SK8 User Guide
Version 0.9

About SK8

SK8 is written for multimedia and authoring tool developers who integrate media components (movies, sounds, still pictures, text, etc.) into a final product.

SK8 can best be thought of as a “meta” tool—a tool used to develop other tools. In this case, the “other” tools are of a multimedia and authoring tool nature.

The objective of SK8 is to provide a multimedia and authoring development environment which could be used by programmers and non-programmers alike and which would enable order of magnitude productivity gains.

Our main objectives have been:

- To provide a direct manipulation user interface for authoring the graphics, animation and multimedia aspects of the application with minimal programming.
- To recognize that programming is necessary at some point, but to provide a language that could be used by a scripter (e.g., someone who has acquaintance with HyperTalk or spreadsheet macro languages) but which had all the power expected by a professional programmer.
- To provide a rich yet elegant, fully object-oriented application framework that would provide all the power needed to develop conventional as well as multimedia applications of the future. This framework would provide very high-level, easy to understand abstractions that would allow developers to focus on their task, rather than on the infrastructure needed before they could get started.
- To provide a powerful graphics and animation system that would encourage experimentation and obviate the need for programmers to get involved with system-level or graphics programming.
- To provide a model which could be used to cross-develop to other runtime models with reduced effort.
- To make the SK8 system as platform-independent as possible.This was solved by having a sufficiently rich framework—one that would abstract away the platform’s system environment.

The key to success for SK8 has been a core creative design and implementation team working with an extended group of colleagues and users over the years to refine the system’s usability.

HardWare Requirements

A Macintosh Quadra or PowerMac running System 7.0 or greater with 32 MB of memory will provide the best results for SK8 development with Release 1.0.

SoftWare Requirements

QuickTime 2.0 for playing movies within SK8.

To develop an authoring tool or title, you will probably desire, at some point in the future, a cross-development runtime toolkit for the platform on which you wish to deliver your titles or applications—such as Kaleida's ScriptX™. We are also considering other target platforms, like Apple's Newton™.

In the future, SK8 will be able to output to other platforms.

Intended Audience

The *SK8 User Guide* is written for the users of SK8 and for the SK8 developer who wants to do any of the following:

- prototype an application in SK8
- develop SK8 tools to be used by other SK8 developers
- develop a SK8 project that will serve as an authoring "tool." The tool can potentially be used by content developers in a particular domain (e.g., electronic books, adventure games) to build multimedia titles or applications.

Assumptions Before Starting

To do SK8 development, you need to be familiar with the:

- basic concepts of object-oriented programming systems (OOPS). A number of books on OOP are available and it is recommended that you become familiar with the concepts before tackling any projects in SK8. A bibliography is provided in the Appendices.
- basic concepts of authoring tools and multimedia development. Refer to the bibliography.
- basic components of SK8—the SK8 Script Language constructs used to develop programs in SK8, the SK8 Project Builder and the SK8 Object Framework. Refer to subsequent sections/chapters in this manual.
- operational fundamentals of the Macintosh.

About the SK8 Documentation

The SK8 Documentation strategy has three parts:

- The User Guide. This book, which provides tutorials as well as a multitude of examples illustrating how to get things done using SK8, its language and its object framework.

- The Object Reference. A book which presents all the object, functions, global constant and variables that are available in the SK8 object framework. Every property and handler of each object is described very briefly. This book is meant to be used as a reference by the experienced SK8 user.
- On-line help. While running SK8 you can bring up the Documentation window and ask for documentation about anything in SK8.

The User Guide

The User Guide is organized in two sections. The first section contains two tutorials and introduces the user to the basic concepts of SK8: the concepts you have to understand in order to use SK8 effectively. Its parts are:

- Overview. This is a high level view of SK8 as a system. The parts that make the system are described and the overall capabilities of SK8 are presented.
- Tutorial. A very gentle hands-on introduction to using the SK8 Project Builder (SK8's User Interface) to create a simple project: a Concentration game that uses sounds instead of images. Introduces the main windows of the Project Builder and how to create objects, layout actors, define handlers, functions and variables, import media and play sounds among other things.
- Basic Concepts. This chapter presents in succinct sections, the basic concepts of SK8. Each section uses examples to explain the concept in question.
- The Project Builder. This chapter describes the tools and menus that are available in the Project Builder.
- SK8Script. A one chapter overview and introduction of the SK8Script language. Designed to cover the language in its entirety answering questions like "how do I do a for loop in SK8Script?"
- Object Oriented Programming in SK8. The general section is closed by another tutorial that focuses on how to do Object Oriented programming using SK8. Presents the features of SK8 that encourage good OOP design.

The second section cover SK8's object framework. While the Object Reference describes objects alphabetically, this section of the User Guide groups objects in functional groups. For example, all the objects involved in importing and exporting data from SK8 are grouped in one chapter, all the objects involved in playing QuickTime movies is in another chapter and so on.

The second section of the User Guide is still to be written.

Conventions and Visual Cues

Special Fonts

All code listings, reserved words, names of objects, properties, handlers, variables, constants, functions and arguments are shown in `Courier` font (`this is Courier`).

Words that appear in **boldface** are key terms or concepts. These terms are defined in the Glossary contained at the end of the User Guide.

Types of Notes

There are several types of informational notes used in this book.

Note

A note like this contains information that is interesting but possibly not essential to an understanding of the main text. ◆

IMPORTANT

A note like this contains information that is essential for an understanding of the main text. ▲

▲ **WARNING**

Warnings like this indicate potential problems that you should be aware of as you design your application. Failure to heed these warnings can result in system crashes or loss of data. ▲

Lexical Notation

`‘symbol’`— A symbol is a series of alphanumeric and symbolic characters that represents the name of a SK8 entity. A symbol is surrounded by apostrophes, i.e., single quotes.

`“String”`— Strings are delimited by double-quote marks. The characters between quotes are the elements of the string. Within the string, white space is significant (it contributes to the string). Strings are surrounded by double quotes.

`Object` — A direct reference to a SK8 object appears in `Courier` font, with the initial letter capitalized. When a SK8 object’s `ObjectName` is used as an adjective (as in, for example, “the rectangle's location”), the `Objectname` is not capitalized and is written in normal text. A general reference to the object (e.g., “an Oval”) is in plain, non-Courier, non-capitalized text.

`aHandler, aProperty, an Event` — A SK8 handler name, property name or event name is also in `Courier` font and starts with a lowercase letter.

`sample code` — All sample SK8Script appears in single-spaced `Courier`.

`{item1, item2}`— A SK8Script list of items is placed in curly brackets. Each item is delimited by a comma.

Other notation conventions

For purely expository reasons, when we say “SK8 sends event X to object Y”, we mean “SK8 invokes or calls handler X of object Y”.

Additional Support Materials

If you are interested in learning more about SK8, additional materials, such as *Getting Started with SK8* videotapes and tutorials will soon be available. To get your name added to the SK8 group address, contact Lori Leahy, SK8 Project Coordinator, at 408-974-2730, AppleLink: LLEAHY.

C H A P T E R 1

SK8 Overview

This chapter introduces you to SK8 (pronounced “Skate”), including:

- What is SK8?
- An overview of SK8 system components.
- A brief overview of the capabilities each component provides.

A more detailed description of SK8 concepts and components are discussed later in this manual.

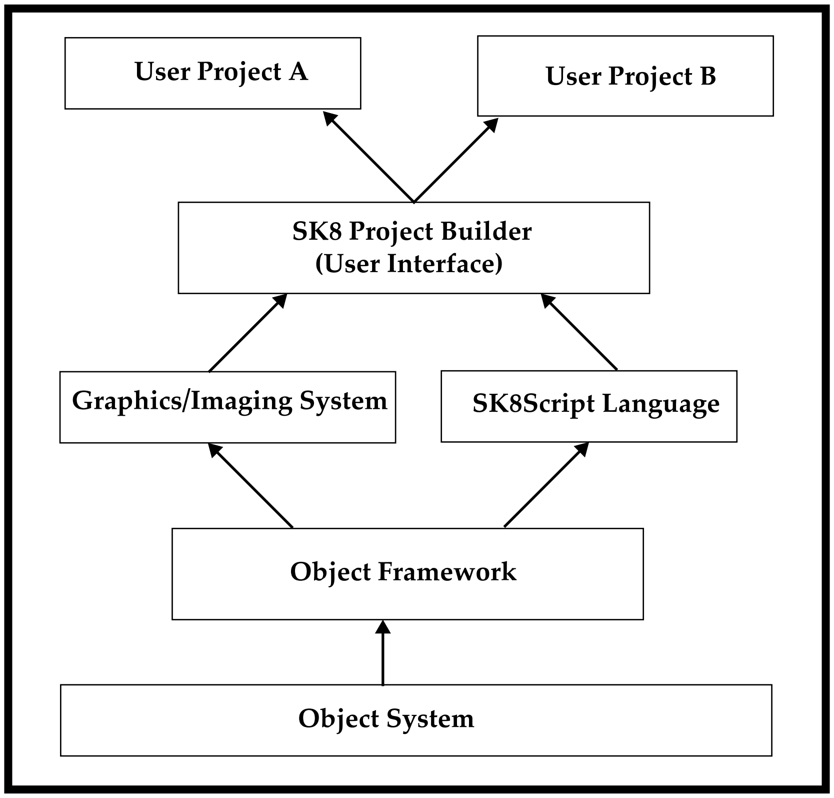
What is SK8?

SK8 is an object-oriented authoring environment designed for rapid development of highly customized authoring tools and titles for multimedia. SK8 is considered a “meta” tool in that it can be used to develop other multimedia tools. SK8 is designed to provide development support for multimedia artists, designers, administrators, and programmers (hackers and non-hackers).

System Components of SK8

SK8 is a prototype-based object system that provides the developer with an Object System, an Object Framework, a Graphics and Imaging System, a Scripting Language, and a powerful user interface called The Project Builder.

The following figure illustrates SK8’s system components. To present SK8’s system components, we will start at the bottom of the diagram and work our way up.



Object System

Three important concepts regarding the Object System are:

- Everything in SK8 is an object. All data and procedures are represented as objects. A character is a `Character` object. An integer is an `Integer` object. Projects are objects. Text, sounds, scripts, graphics, and QuickTime movies are, as far as SK8 is concerned, just different types of objects with different properties and handlers.
- SK8 is based on a prototype-based model where every object can be a template for creating new objects with different types of properties and handlers.
- An object can refer to other objects. In SK8 a property is an object reference. This allows objects to include other objects by reference. In fact, the same property can include different types of objects at different times.

The Object System is the foundation for SK8. The Object System supports many object, handler, and property related functionalities. Examples are: the addition or removal of object properties at any time, the creation of a child of an object, the duplication of an object, etc. A complete list of Object System functionalities is provided in the Object System chapter.

Object Heterarchy

In order to keep track of the thousands of SK8 objects and their associated or inherited properties and handlers, SK8 objects are organized into an **object inheritance heterarchy** where each object is the child of some other object. We use the term heterarchy because an object can have more than one parent and is, therefore, not a strict hierarchy.

The top object of this heterarchy is the object `Object`. This is the only object that is not a child of anything and all objects are descendants of `Object`. `Object` contains all the handlers needed to create new objects or copies of objects, initialize objects, add or remove properties, assign object names, and more.

More Object System Information

"The Object System" and "The Object Framework" chapters provide additional information.

Object Framework

The SK8 Object Framework provides a set of objects for developing applications and tools. The Framework provides objects that represent files, I/O devices, sounds, shapes, dates, clocks, etc. These objects are described in detail in the SK8 Object Reference Manual.

SK8 provides objects that abstract away all the particulars of the operating system and hardware upon which an application runs. For example, devices like disks, keyboards, pointers, monitors, and system clocks are represented by objects with which you can interact with SK8Script. In addition, objects are provided to represent how the lowest levels of the graphics system work, as well as the much higher-level aspects of the graphics and animation system, including:

- actors for creating object-oriented 2-1/2D graphics
- visual effects that can be applied to graphical changes
- translators for converting data from files to internal data formats
- media objects for QuickTime movies, pictures, and other Macintosh resources including sounds
- objects for traditional data types such as integers, characters, and strings as well as arbitrary collections of objects, arrays, lists and vectors
- objects that handle errors and events
- user interface objects like buttons and scrollers

More Framework Information

More information about the object heterarchy can be obtained by using the System Browser in the Project Builder.

Additional details about the Object Framework are found in the SK8 Object Reference Manual and later part of this manual.

Graphics System

The SK8 Graphics System provides two general models for the creation of multimedia projects:

- The 2-1/2-D Media and Event Engine
- The Imaging System

The media engine provides a model for creating object-oriented graphics and animations. A variety of graphic objects for geometrical shapes, as well as an extensive library of user interface objects, is provided. The media engine delivers:

- Complete platform-independence
- Drawing capability for any graphical shape and complex rendering on display devices
- Object-level clipping
- Object level zoom and pan for any view shape
- Object-oriented rendering mechanism
- Many types of pre-defined renderers
- Many ease-of-use controls, such as auto highlighting, for all graphic objects
- Extensible and user-definable graphic objects and renderers
- A library of existing graphic shapes and user interface objects

The engine is responsible for all of the drawing and bookkeeping chores that programmers would typically have to implement to provide smooth and sophisticated graphics to their applications.

The actors supported by the graphics engine may be separated into two categories:

- Geometry actors. The geometry actors draw geometry which is managed by the graphics engine. Geometry actors include Rectangle, Polygon, LineSegment, Oval, and others. The system can be extended by creating new geometries. This can be done quite easily. See the Actor chapter of this guide for details.
- User Interface actors. The user interface actors draw widgets which are the building blocks for user interfaces. These actors are actually created out of geometry actors, but provide more functionality. Among the user interface actors are scrollers, buttons, text fields, lists of text or graphics, and spreadsheets.

IMPORTANT

When you use actors, the graphics engine will manage the imaging for you. If you want to extend the imaging capabilities or implement a different graphics engine (e.g., a 3D engine), you should refer to the Imaging chapter. ◆

2-1/2-D Graphics Engine

This component of SK8 relieves programmers from the burden of writing most of the low-level graphics code required to support sophisticated multimedia titles and applications. It is based on a platform-independent imaging system.

The 2-1/2-D graphics engine is a render-based model providing arbitrary pan and zoom, containment, user-definable Actors and Renderers, and full support of QuickTime.

Technical Note
Render-based is an object architecture in which all screen drawings are done by objects (as opposed to a library function or subroutine). A renderer is the only type of object in SK8 allowed to draw on the screen. These renderer objects are given the special knowledge of how to draw and fill a shape with color via `render` handlers that come with the object or that you have written. (Yes, you too can write a `render` handler!) ◆

Imaging System

The imaging system provides a platform-independent, object-oriented model for imaging arbitrarily and directly onto display devices (e.g., monitors and printers). With the imaging system, you can create your own renderers.

The imaging system provides:

- Complete platform-independence
- Mask objects for clipping draw operations
- Pen objects to save and restore graphics context
- A suite of painting and drawing operations
- Recordability and playback of drawing operations
- A programmer's interface for creating primitive Media Engine renderers

SK8's imaging system allows you to write very low-level graphics in a platform-independent way. This level is reserved for programmers who want to implement their own renderer objects when SK8's renderer library does not support some needed functionality.

Despite the fact that this is a low-level capability, the imaging system is still object-oriented and has been carefully designed to make it as easy to learn as possible, without sacrificing power.

▲ **WARNING**
You should not modify the imaging system unless you are convinced that the SK8 Graphics System does not support the feature that you need. ◆

More Graphics and Imaging System Information

Refer to the Imaging chapter of this Guide. Please be aware that the graphics and imaging information are primarily intended for the advanced user.

SK8Script Language

The SK8Script Language is designed to be easy to program, easy to read, and easy to learn; but it is also a complete, compiled, efficient and object oriented programming language.

SK8Script's genesis was in the early specifications of AppleScript and shares many syntactic and some semantic features with it. While AppleScript was initially targeted for inter-application scripting, SK8Script's goal has been to serve as a full programming language capable of being used by scripters and professional programmers alike, obviating the need to write lower-level code.

SK8Script is fully object-oriented and dynamic. Its objects are, of course, SK8 objects, and its dynamics are found in the interactive and fast way that individual handlers are compiled and run, as well as the ease of use of its editing and debugging tools.

The main features of SK8Script are:

- English-like scripting syntax
- Full programming language providing: user-defined types, error checking, etc.
- Protection against errors
- Simple definition of macros via "with" handlers
- Support for all SK8 object system semantics
- Simpler iteration and processing of groups of objects via the Collection protocol
- Object-oriented internal database queries and modifications through selection expressions
- Complete and easy to use error and condition system
- Efficient native-code compiler
- Low-level system access (e.g., operating system and foreign function calls)

Additional information about SK8Script

Additional information about SK8Script may be found in the SK8Script chapter in this Guide.

SK8 Interface (The Project Builder)

The Project Builder is a project, built in SK8, that provides a general-purpose direct-manipulation user interface for rapid development of your own SK8 projects.

The Project Builder provides a variety of powerful object and SK8Script language browsers, editors, debugging tools, media import and export, and the ability to quickly build your application's user interface.

Using the Project Builder

SK8 projects are modules that are eventually converted to a multimedia title, application, prototype or tool. A project is the workspace in which you do your development. The process of building a SK8 project consists of the following steps:

- Designing and quickly prototyping an overall user interface for your project.
- Customizing your interface by changing properties and scripting the interface to provide your application's behavior.
- Experimenting with the interface's space design, look-and feel, and, hopefully, getting early feedback from potential end-users.
- Continuing with a more detailed, final implementation of your project's functional requirements.

These steps are not necessarily carried in strict succession. The overall design and implementation process is iterative. How many times each step or how many times the whole cycle is repeated depends on the project's complexity and on whether you employ a more or less user-centered design methodology.

More Project Builder Information

More information about Project Builder is located in the Project Builder chapter of this manual.

Tutorial

In this chapter a hands-on introduction to SK8 is presented. In the process of building a simple SK8 application, many of SK8's basic concepts will be introduced. This process will also introduce the SK8 Project Builder, as it will be used extensively while making the application. At various points in the tutorial, you will be referred to a section of Chapter 3: Basic Concepts for further explanation. In addition, as each tool of the Project Builder is introduced, you may refer to the Project Builder chapter for a fuller explanation of the tool and it's capability. It's a good idea to skim through all of these sections as you go through the tutorial in order to flesh out your understanding of SK8.

The parts of the text that are just instructions appear in **bold**.

The Application

We will implement a simple "Concentration" game. Concentration is a game typically played with cards. At the start of the game all the cards are placed face down on the table in a grid. During each turn, a player uncovers two cards. If the two cards uncovered have the same picture, the player gets the keep them and scores a point. The player also gets to go for another consecutive turn. If a match is not found, the two uncovered cards are covered and the next player gets a chance. The game ends when all the matches are found and we run out of cards. The player that has found the most matches wins.

For our version of the game we will introduce the following variations/additions:

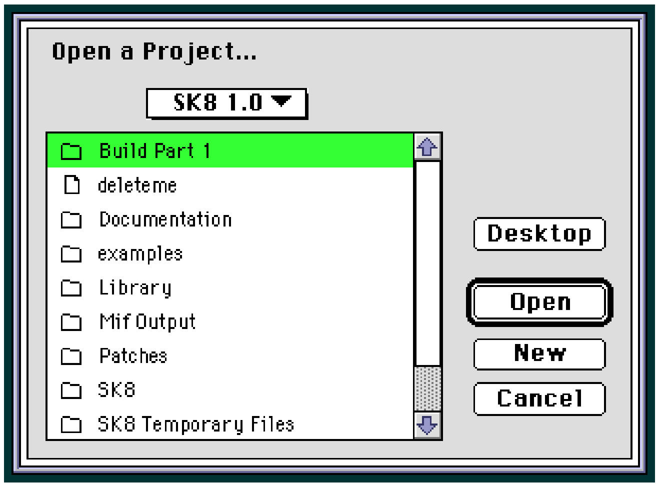
- Instead of pictures we will use sounds. Thus instead of uncovering a card, a sound would be played. This variation makes the game a lot harder than the original game.
- We will provide automatic scoring as well as an indicator to specify the current player.

Using SK8 to Implement Concentration

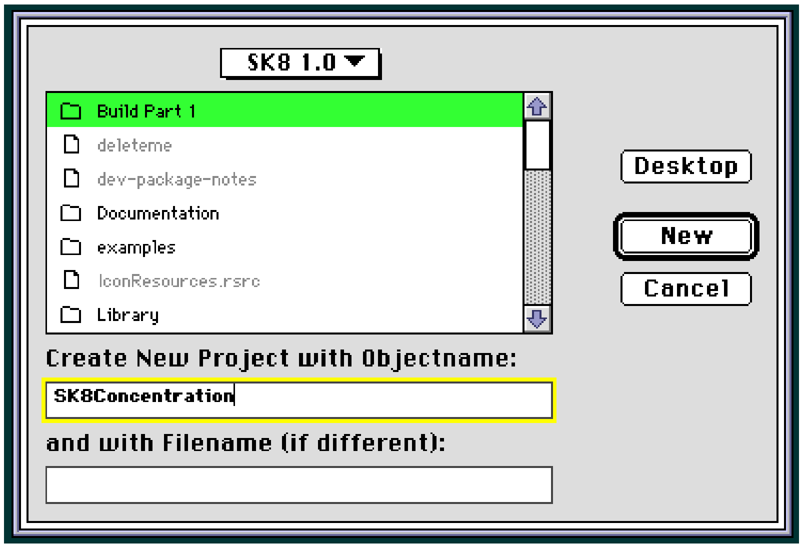
We begin by starting up SK8. To do this, **doubleClick on the SK8 application icon**. SK8 presents you with a startup screen displaying the SK8 Painting and then proceeds to bring up a few of the Project Builder windows. The last window to come up is a dialog box with which you can open an old project or create a new one.

A SK8 project is like a title (or an application). All the work you do is associated to a project. You can think of a project like a bag of objects, constants, global variables and functions. All these together conform your application or title.

(For more information on projects, see the Projects section in the Basic Concepts chapter)



Click on the button labeled "New" in order to create a new project. A new dialog will now appear to let you name the new project. **Type "SK8Concentration" in the field where it asks for the name of the project**.

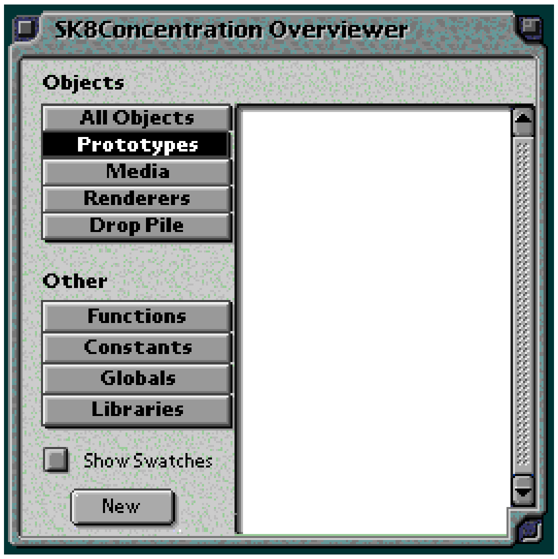


Create New Project with Objectname:

SK8Concentration

and with Filename (if different):

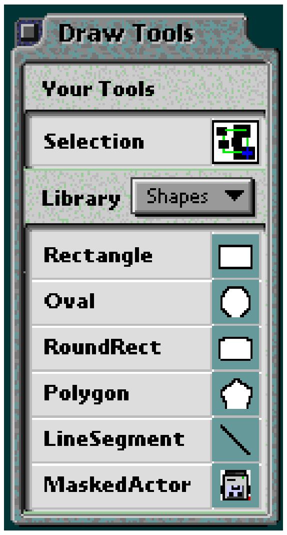
Click on the button labeled "New" to actually create the SK8Concentration project. When SK8 is done doing the required creation, the Project Overviewer window appears as below.



The Project Overviewer allows you to look into the project and examine everything in it. As mentioned above, a project is a collection of objects, globals variables, constants and functions. This list is mirrored in the Project Overviewer. Clicking on the buttons on it shows you all the things you have created in your project.

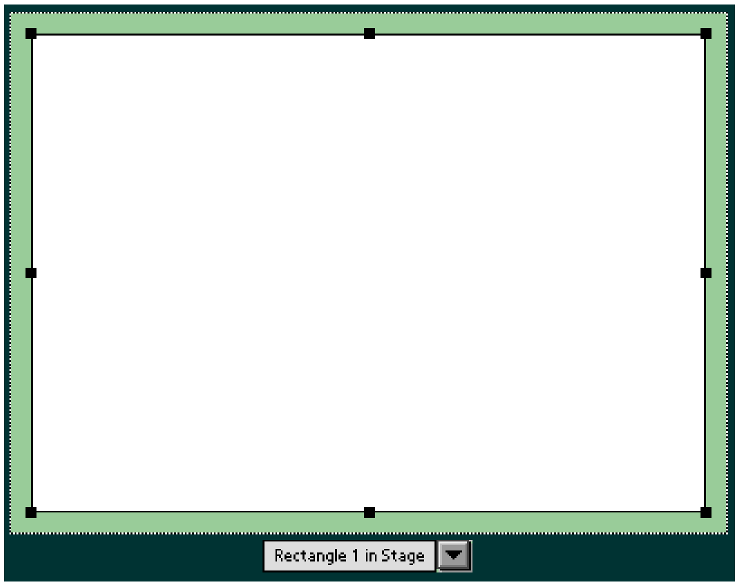
Making the Board

Having created the project, we are ready to start creating the objects that will make up the game. First we need the board: the object where all our cards will be displayed. We can create the board (a Rectangle) by drawing it from the Draw Palette.



The Draw Palette shows a list of actors (graphical objects) that we can create by direct manipulation. To draw our board object **click on the Rectangle tool in the Draw Palette**. The cursor will change into a "+" shape with which you can draw the rectangle. Now **click on an empty point of the screen where you want a corner of the rectangle to be and drag the mouse until you hit the opposite corner. Then let up the mouse**.

The result of this is a new rectangle, created in the SK8Concentration project, that has no objectname. It will be selected by the SelectionHalo: another Project Builder tool.



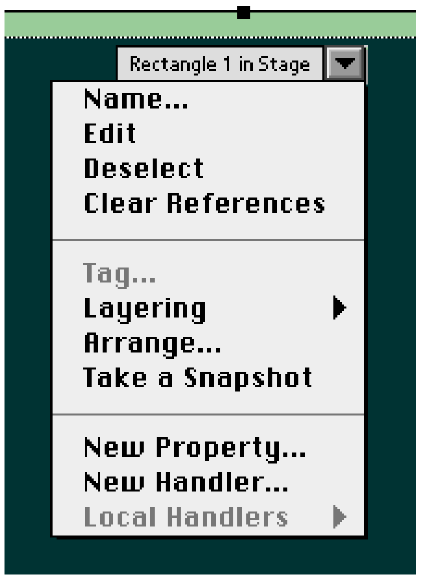
The selection halo has 4 parts that can be seen in the picture above:

- The resizer: these are the little black squares that you can see at the corners and the sides of the selected object. By dragging these squares you can resize the selected object.
- The dragger: this is the thick green frame around the selected object. By dragging this frame you actually drag the object selected.
- The object reference: the rectangle that currently reads “Rectangle 1 in Stage”. This section of the SelectionHalo will always contain a textual description of the object or objects selected.
- The menu: the black triangle pointing downwards. Mouseing on it shows a menu of useful commands for actors.

Notice that the object reference part of the SelectionHalo reads “Rectangle 1 in Stage”. What this means is that the object we have created is the frontmost rectangle in the contents of the Stage. The Stage is the sum of all your monitor space. Windows are nothing more than actors that have been added to the contents of the Stage.

(For more information on the Stage, windows, and actors, see the Actors section in the Basic Concepts chapter)

Next we need to name the rectangle we just created. Let us call it “Board” as we have been calling it all along. To name it, **mousedown on the SelectionHalo’s menu and select the menu item that reads “Name...”**.



In the dialog that comes up **type “Board” and press the Enter key**. Notice that when you are done, the object reference part of the SelectionHalo now reads “Board”.

Making the Board “Special”

At this point our Board is nothing more than a child of Rectangle. As such a child, the Board has inherited all the properties and handlers of Rectangle. The Board, therefore has properties like fillColor, frameColor, frameSize and boundsRect just like the Rectangle object does.

(For more information on the objects and inheritance, see the Objects section in the Basic Concepts chapter)

In this section we start to make our Board object special by changing the value of some of its properties. Further specialization can be achieved by adding new properties and adding new handlers, all of which we will do in this tutorial.

Let us start by adding text to the Board. The text would read “SK8 Concentration”: the title of our game. To add this text all we need to do is set the text property of the Board to “SK8 Concentration”.

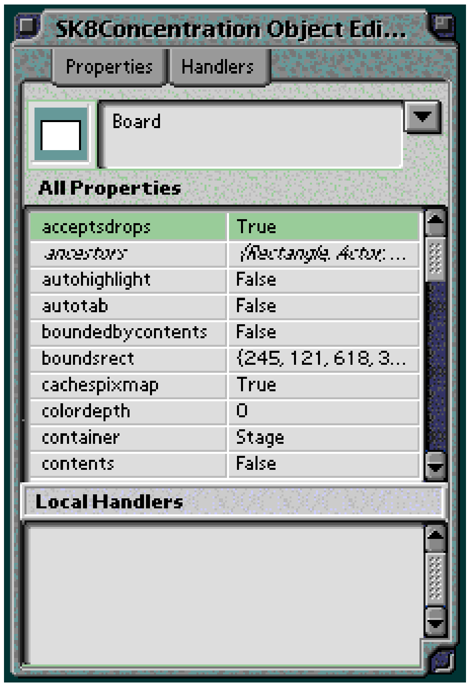
Most of the changes that we will do using the Project Builder can be easily done by typing SK8Script commands into the MessageBox (another Project Builder window). To use this method to add text to the Board, **type the line below in the MessageBox and press Return at the end**. (Always end your MessageBox commands by pressing the Return key.)

```
set the text of the Board to “SK8 Concentration”
```

Notice that the text has shown up in the center of the Board. It is black and its font is “Chicago”. Let us now use the Project Builder to change the appeareance of the text until we are satisfied. The ObjectEditor provides facilities to allow us to change the value of every property of any object by direct manipulation. Let us use an ObjectEditor to edit the Board.

Notice also that as a result of our trip to the MessageBox, the SelectionHalo is no longer selecting the Board. To reselect the Board, **click on the Selection tool of the DrawPalette and then clicking on the Board**. (You can also click on the Board while holding down the Control key).

Mousedown on the object reference part of the SelectionHalo (where it reads “Board”). Do not let up the mouse button and drag the mouse. You will realize that you are dragging the outline of a rectangle. This outline represents the reference to the object that was selected by the SelectionHalo. **Drag the outline to the ObjectEditor** (the window on the top right of the Screen). As you drag over the rectangle at the top of the ObjectEditor (the rectangle on the left of the menu) the rectangle will highlight. When this happens, **let up the mouse to drop the reference to the Board onto the ObjectEditor**. The ObjectEditor is now focused on the Board.

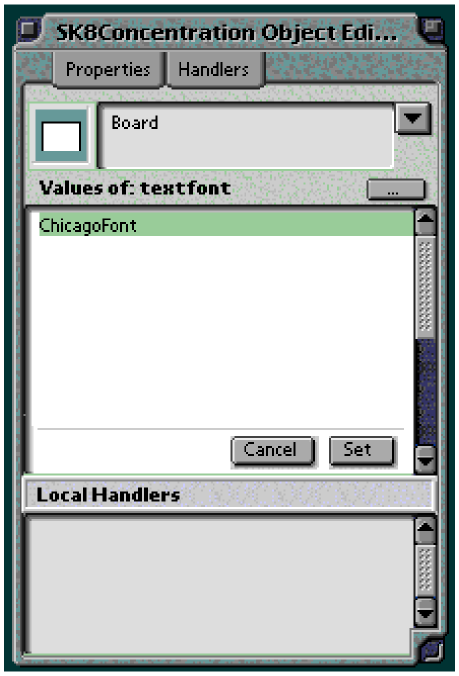


acceptsdrops	True
ancestors	{Rectangle, Actor, ...
autohighlight	False
autotab	False
boundedbycontents	False
boundsrect	{245, 121, 618, 3...
cachespixmap	True
colordepth	0
container	Stage
contents	False

The ObjectEditor shows us all the properties of the Board as well as all the handlers we have defined on it (none so far). At the top left we can also see a snapshot of the Board. The snapshot is not very interesting now because our Board is not very interesting. Let us fix this problem.

We will change the font and size of the text and place it on the top of the Board. The textFont property controls the font used to draw the actor’s text. To change this property **use the scroller in the properties panel to scroll to the property called “textFont”**. Notice that the current textFont is ChicagoFont.

Now **double click on the property**. The property panel of the ObjectEditor is now replaced by a new panel that lets you change the property. Let us change the font to “Times” or TimesFont (we can use either of these formats to specify the textFont).



To change the textFont, **type "TimesFont" (without the quotes)** into the field that now reads "ChicagoFont", or you can click on the button labelled "..." to get a list of options. When you are done **click the "Set" button or press the Enter key**.

The textSize can be changed in the same way. **Find the textSize property, double click on it and type the size wanted**. A textSize of 36 will do nicely.

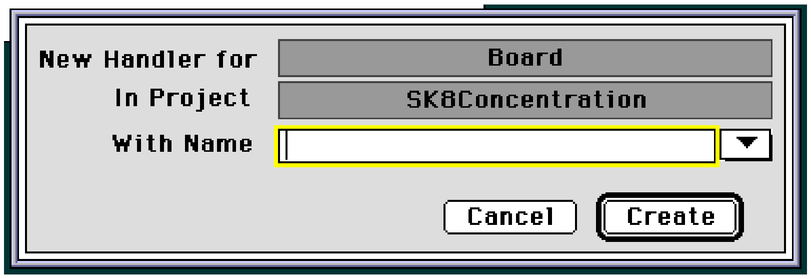
The final task is to move the text of the Board to the top. For this we use the textLocation property. As usual, **find the textLocation property and double click on it**. Notice that in this case, the "..." button has been replace by a menu. **From the menu that appears above the properties panel, select 'topCenter'**.

Making the Board Draggable

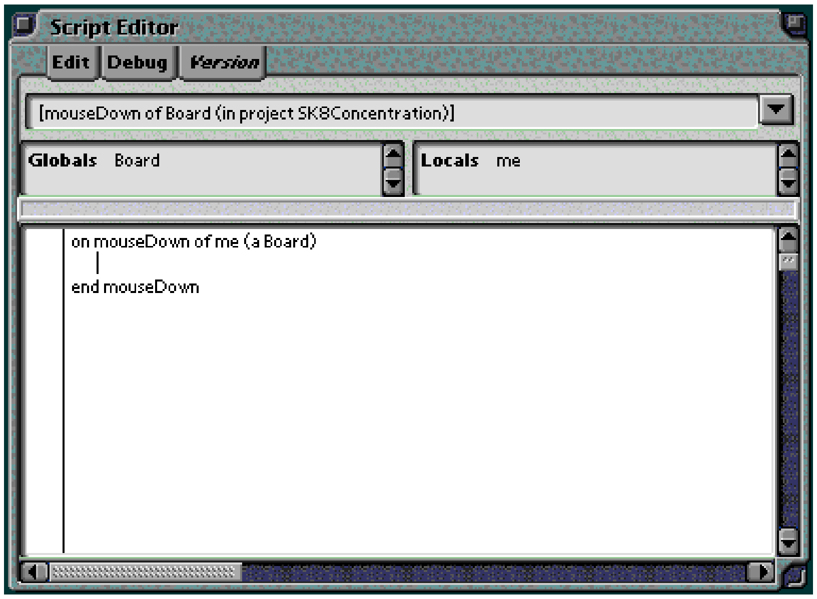
At this point we have specialized the Board by changing the values of some of its properties. The ultimate way to specialize objects is to define handlers on them, to teach them to do new things. Our first example will involve making the Board drag itself when the mouse goes down on it.

To get this behaviour all we need to do is to add a mouseDown handler to our object. This handler will be called by the event system when the mouse goes down on our object.

Pull down the "Handlers" menu from the ObjectEditor's menubar and select "New Handler..." The following dialog will appear:



In the field, enter the name of the handler you want to define: mouseDown. (You can also select mouseDown from the menu on the right. This menu contains the common actor events.) **Then click on the "Create" button** to bring up the SK8Script editor.



```
on mouseDown of me (a Board)

end mouseDown
```

Note that the handler has already been started for us. Let us concentrate briefly on the first line (or header) of this handler. It reads:

```
on mouseDown of me (a Board)
```

The first part ("on mouseDown") means that this program will be executed whenever a mouseDown event happens on the object on which this handler is defined. The object on which this handler is defined is shown between parenthesis: a Board. Note that this handler will apply to any descendant of the Board (not just to the Board itself). Finally, "me" is the way we refer within the handler to the actual descendant of Board that got the mouseDown event.

We want to drag the Board on mouseDown. Thus, in the place where the cursor is, **type the following line**:

```
drag me
```

Our whole handler, therefore looks like this:

```
on mouseDown of me (a Board)
   drag me
end mouseDown
```

Now we are ready to activate this handler. To do so, **select "Activate Current Version" from the Version menu in the SK8ScriptEditor**. You will notice that the Version menu's text ceases to be in italics when you do this. This means that the version of this handler that you are currently looking at is the one that is activated.

We can now try to drag the Board to see if this handler is actually doing what is supposed to do. **Click on the Board**. Now **mousedown on it and drag**. You will see that the Board's outline is dragged by the mouse. When you release the mouse button, the Board moves to the place where the outline ended.

We could drag the Board live (moving it continuously instead of just dragging an outline of it). To do so, add an extra argument to our call to the drag handler. **Replace the line we typed in the handler with the following line**:

```
drag me with live
```

When done, **select "Activate Current Version" to activate the handler and then close the SK8Script editor window by clicking on its close box** (the little rectangle at the top left of the window).

Adding the Cards

We are now ready to add the cards to the board. We will play with 100 cards. They should be arrayed in a grid like fashion. A little planning is in order before we start making the cards. We should notice that every card will do the same thing:

- when clicked on, each card will play the sound associated to it.
- when clicked on, each card will indicate in some form that it has been clicked (for example by changing its fillColor to Red).
- when clicked on, each card has to check whether it is the second card that has been clicked and if so check if a match has been found.

From this simple analysis, we see that all cards need a property into which to store a sound, and a mouseDown handler that does everything we described. We could create 100 rectangles and add the property to each one and then define the mouseDown handler 100 times but if we did so our project would probably miss its deadline... Object oriented programming comes to the rescue!

We can create 1 new rectangle, calling it "Card". We add the property and the handler to this rectangle. When we are done, we just make 100 cards and add them to the board.

Let us start by creating the Card object. **In the top panel of the Object Editor, type the following line**:

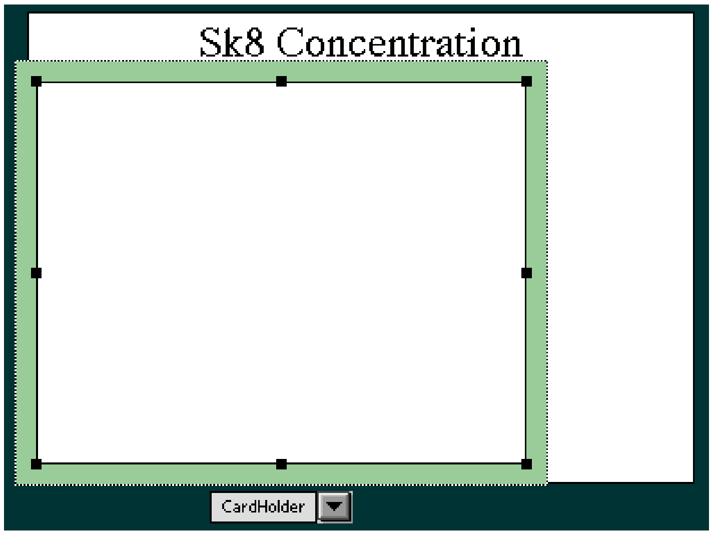
```
new Rectangle with objectname "Card"
```

This creates a new rectangle with our given objectname and sets the Object Editor up to edit the object. Note that you can type any SK8Script expression into this top panel and it will be evaluated. The editor will focus on whatever object or set of objects are returned by the expression.

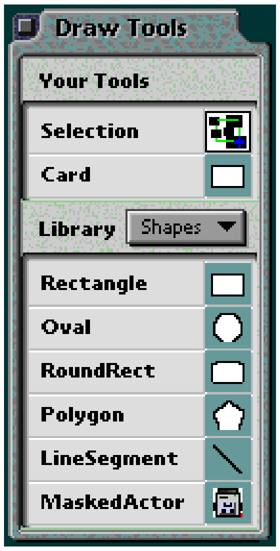
We will now add a property to our Card object. To add aproperty, **select "New Property..." from the Properties menu**. A dialog appears, that you can use to name the property. **In the field provided type "mySound" without the quotes and press Enter** (or click on the "Create" button).

Returning to the ObjectEditor, you see that the new property is selected and its value is set to False. Also notice that the property is bold. This means that it is a "local" property of this object, a property that this object has defined as opposed to a property that was inherited from any of its ancestors.

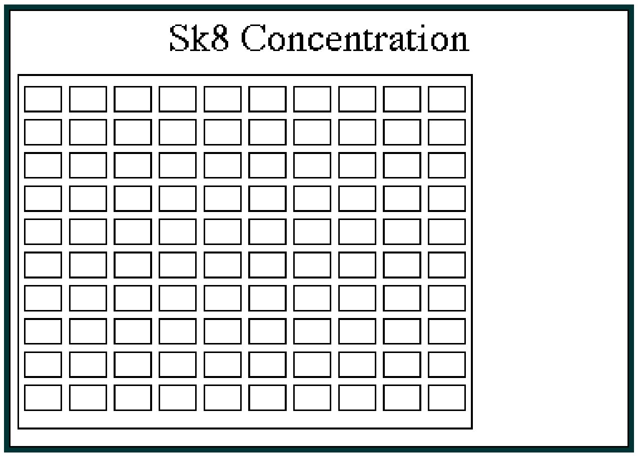
Let us make a rectangle into which the cards will be made. **Click on the Rectangle tool of the DrawPalette and draw a large Rectangle inside of the Board**. Then **name the rectangle "CardHolder"**. The following figure shows the state of the Board.



Now we will use the DrawPalette to draw 100 cards into the CardHolder. For that, we need to add Card to the objects that the DrawPalette can draw. To do this, **mouseDown on the picture of Card that is at the top of the ObjectEditor. When the rectangle appears, drag it to the DrawPalette. Drop it right over the Selection tool**. You will see that Card appears right below the Selection tool. This new tool we have created can be used just like the Rectangle tool we have used before.



Instead of drawing just one Card, we can use the Option key to draw multiple ones in one operation. **Click on the Card tool**. **With the Option key pressed, mousedown on the top left corner of the CardHolder**. **Keeping the Option key pressed, drag the mouse**. You will see that the outline of more and more cards appears. **Keep going until you have 10 rows and 10 columns of cards**. **Then release the Option key and use the mouse to resize all the cards at once to make them fill the space in the CardHolder**. **When satisfied with the result, let up the mouse button**. 100 cards will be created and placed inside the CardHolder. The result is shown below.



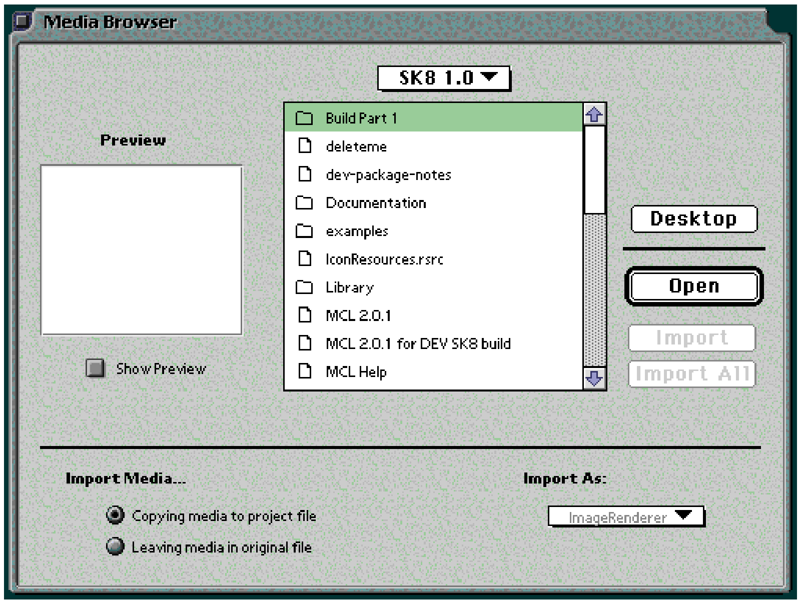
If the Selection Halo does not read "100 items" when you are done, you have created the wrong number of Cards. You can delete all the card you have created by pressing the Delete key. A dialog will come up asking you if you really want to do it. Click the "yes" button. Then you can try creating the 100 cards again.

This is a good time to save your project. **From the file menu select "Save SK8Concentration". When asked whether you really want to save, click on the "Yes" button.**

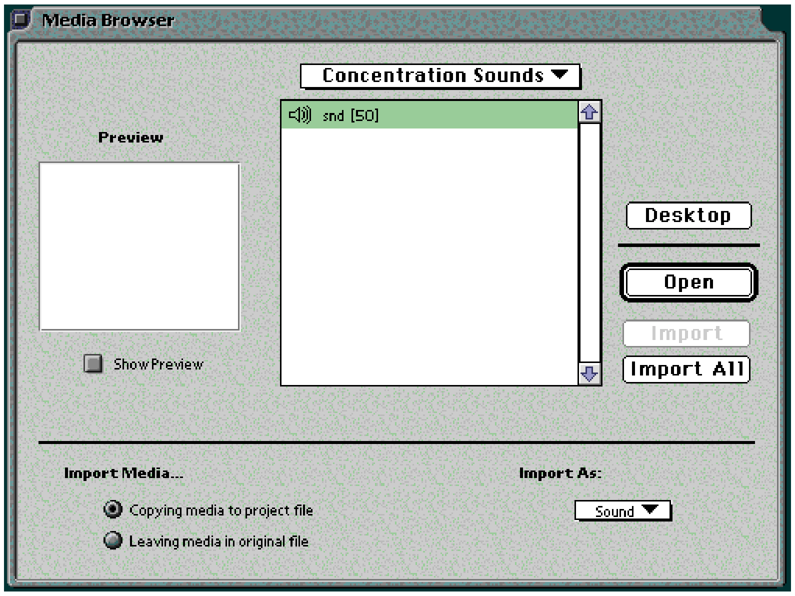
Importing Media

The cards are ready. We now need some sounds to associate to them. Since we have 100 cards we will need 50 sounds. We can get the sounds from any file that contains “snd “ resources. Let us assume that a file called “Concentration Sounds” exists with all the sounds we need.

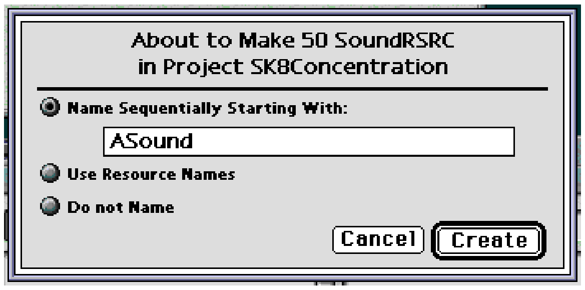
To import the media into our project we will use the Import Media dialog. **Bring it up by selecting the “Import Media” menu item from the File menu**.



The first step of this process is to find the file that has the media. You can use the file list at the middle of the dialog just like the standard Macintosh file dialog. **Once you have found the right file, double click on it** to see a list of the resource types that can be found in it.



Conveniently, there are 50 sounds already in the file. To import them all, **click on the "Import All" button**. Importing the sound resources will create children of Sound. A dialog will come up to allow us to name the new objects.


About to Make 50 SoundRSRC
in Project SK8Concentration
Name Sequentially Starting With:
ASound
Use Resource Names
Do not Name
Cancel
Create

Let us give all our sounds names that start with "ASound" and end with an integer. For that, **click on the RadioButton titled "Name Sequentially Starting With" and type "ASound" in the field provided. Click the "Create" button**. You will then hear your hard disk spinning as all the media is copied to your project's file.

You will now notice that the project overviewer has switched to its drop pile in which it shows all the objects we have just created with the Import Media dialog.

Close the Import Media dialog by clicking on its close box.

Creating Global Variables

It will be useful to keep all our sounds in a global variable. Let us call this variable, "concentrationSounds". To create it, **switch the ProjectOverviewer to global variable mode by clicking on the button labeled "Globals"**. The object list becomes empty since there are no variables defined in our project yet.

To define the variable, **click on the “New” button**. A dialog appears. **Type “concentrationSounds” in the first field. The click the “Create” button to create the variable**. The project overviewer now shows the new variable with its current value: False.

To set the value of the variable to the list of sounds, **double click on the variable**. The list of variables becomes an editor into which we can type the desired value. Now we can use the power of SK8Script to write an expression to return just the sounds we are interested in. A good start would be

```
the knownChildren of Sound
```

This would return every named child of Sound in the system at the moment, which would include not only our 50 sounds but every other sound that was there before. We can refine our search by specifying that we only want those sounds whose objectName starts with “ASound”. In SK8Script,

```
every item whose objectName starts with “ASound” in the
knownChildren of Sound
```

Type this line in the editor. Press the Return key. The variable now contains all our 50 sounds. To test this hypothesis you can type in the MessageBox:

```
get the length of concentrationSounds
```

Note that an alternative way you could get these same sounds with the following SK8Script expression:

```
every item whose project is equal to SK8Concentration in the
knownChildren of Sound
```

Distributing the sounds to the Cards

In SK8 there are two ways to write scripts: as handlers or as functions. A function is just a named piece of script that does something. A handler, on the other hand, is a scripts that is defined to teach an object how to react to a specific message. The name of the handler is just the message that the object can now respond to. In one of the previous sections, where we defined a mouseDown message for our Board, all we did was teach the Board how to react to the mouseDown message. Each time a mouseDown event happens on the Board, the handler we defined will be called.

A question we must ask ourselves often is whether a task should be implemented using a function or a handler. A very nice feature of handlers is that we can define the same handler on many different objects to do many different things. In our example above, we changed the behaviour of the Board from the behaviour it had inherited from its parent (Rectangle).

(For more information on handlers, see the Handlers section in the Basic Concepts chapter)

Our next task is to write a program that randomly assigns eachof the 50 sounds to 2 cards in the board. We could make this program be a handler of the Board or the CardHolder, or we could make it a function with no arguments. In this case it is not really clear which is the best alternative and thus, for the purpose of illustrating how to create functions, we will make it a function.

The function will be called “distributeSounds”. We can use the ProjectOverviewer to add a new function to our project. **Click on the button that reads “Functions”**. The object list becomes empty because we have not yet defined any functions for our project. **Click on the button labeled “New”**. When the naming dialog comes up, **enter “distributeSounds” on the field and click the “Create” button**. A SK8Script editor comes up to let us type the function in. **Type the following**:

```
on distributeSounds
   set the cards to the contents of CardHolder
   repeat with oneSound in concentrationSounds
      -- pick the first card, give it a sound and
      -- remove it from the list.
      set card1 to any item in the cards
      set card1's mySound to oneSound
      remove card1 from the cards
      -- pick the second card, give it the same
      -- sound and remove it from the list.
      set card2 to any item in the cards
      set card2's mySound to oneSound
      remove card2 from the cards
   end repeat
end distributeSounds
```

This program distributes the 50 sounds to the 100 cards randomly. First note that any line that begins with two dashes “--” is considered to be a comment. The script starts by setting the local variable cards to all the cards we want to give sounds to. The repeat loop that follows runs through everything in concentrationSounds (the global variable we defined). At every iteration of the loop, the local variable oneSound is set to one of the sounds in our list. Given a sound, we need to pick two cards at random and assign the sound to them. Let us go through the the three lines that achieve this in detail.

```
set card1 to any item in the cards
```

This line picks a random item from the cards (a collection of cards). The picked card is set to the value of the variable card1.

```
set card1’s mySound to oneSound
```

Now that we have the chosen card, we need to give it the sound. The sound is kept in the loop variable oneSound.

```
remove card1 from the cards
```

Once we are done with the chosen card, we have to make sure it will not be picked again and given another sound. All we need to do is remove the card from the cards that are still available to be picked.

We do this twice for each sound, in order to assign the same sound to two different cards.

To try out this function, **activate it by selecting the "Activate Current Version" menu item in the "Version" menu of the editor. Then go to the MessageBox and call the function by typing the following line followed by the Return character**:

```
distributeSounds()
```

Close the SK8Script editor.

Players and Scores

We are still missing some important parts of our game: a way to remember who is currently playing, a way to remember the score of each player and a way to report these scores.

Let us start attacking this (almost) final part of the game by creating a new object called a Player. The only thing that is important about players is that they have scores. We can create this object and add the "score" property to it in by the use of SK8Script alone. **In the MessageBox type the following**:

```
new object with objectName “Player”
```

And type the following line to add the "score" property to it. (Note that single quotes should enclose the word "score".)

```
addProperty Player, ‘score’
```

And now we can make two players to represent player 1 and player 2. As we create these players we will set their score to 0. **Type in the MessageBox**:

```
new Player with objectName “Player1” with score 0
new Player with objectName “Player2” with score 0
```

To remember who the current player is we can add a property to the Board called "currentPlayer". By setting this property to Player1 or Player2 we will always know who the current player is. **Again in the MessageBox, type**:

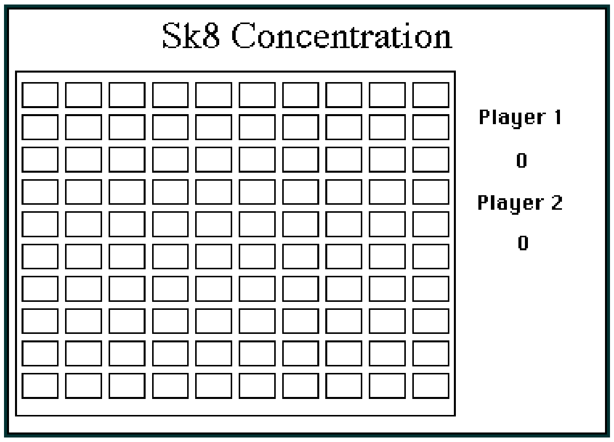
```
addProperty Board, ‘currentPlayer’
```

Let us now provide a mechanism with which our game will show who the current player is and show the score of both players. We will create 4 Label objects and put them in the empty space at the right of the Board. (You can think of a Label as just a piece of text.)

You will notice that the Label tool is not present in the DrawPalette. This is because the palette is only showing the basic shapes in the system. **From the menu in the palette select the menu item that reads "Widgets"** to show the tools to draw all sorts of user interface widgets. The Label tool will be the first one in the list.

To create the labels, **select the Label tool and then click on the place where you want the label to go (the empty area at the right edge of the Board)**. You will see that the label's text will read "Untitled". Now you can use the ObjectEditor to set the label's text. Focus the ObjectEditor on the label by dragging the SelectionHalo's object reference to the ObjectEditor. Then set the text property by selecting the property, double clicking on it and typing the new text. Do this 4 times, creating 4 labels. Their text should be: "Player 1", "0", "Player 2" and "0".

When you are done, **set the objectName of the two labels whose text is "0" to "Player1Score" and "Player2Score"**. The Board will look like this:



Showing the Scores

We want to keep the player's scores always in synch with the text of the score labels on the Board. The way to do this is to ensure that whenever a player's score changes, its score label is updated on the spot. Consider the following line that needs to be use to change the score of Player1:

```
set the score of Player1 to 1
```

What this line actually does is call the "set score" handler of Player1. Whenever a property is added, two handlers are defined: a getter and a setter. In this case the handlers defined are "score" (which returns the score) and "set score" (which sets the score to the value provided).

You can redefine any of these handlers to do something else besides accessing the property in question. You might, for example, change the setter of the "score" property to, in addition to setting the property, updating the player's score label.

To redefine this handler, **type "Player1" in the top pannel of the ObjectEditor**. **Then select the "New Handler..." menu item from the "Handlers" menu**. A dialog comes up to ask us to name the new handler. **Enter "set score" (without the quotes)**. The SK8ScriptEditor appears. **In the editor type the following handler**:

```
on set score of me (a Player1) to newValue
   do inherited
   set the text of Player1Score to newValue
end set score
```

Note that the newValue argument appears in the editor. This argument holds the value that the user wants to set the property to. The first line in this handler actually sets the property. The second line, after the property is set, makes the text of the label devoted to Player1's score be updated, as we wanted.

Now close the SK8Script editor and do the exact same thing for Player2 (with Player2Score instead of Player1Score).

Did we say the "exact same thing"? Mmmm. Seems like some more object oriented thinking could allow us to redesign this in a slightly better way.

And finally, the Game

The big handler is the click handler of the Card object. This is what is has to do:

- if this is the first click, play this card's sound. Then uncover it by, for example, setting its fillcolor to Red. Then somehow we must remember that the next click will be the second click.
- if this is the second click and this card is not uncovered, play its sound and uncover it. If this card's sound is the same as the first card's sound, a match has been found.
- if a match has been found, remove both cards from the board and add 1 to the score of the current player. If the combined score of the two players equals 50 points, the game has ended. Report who won.
- if no match was found, uncover both cards and make the currentPlayer of the Board be the other player.

To remember whether this click is the first or second click we will use a variable called waitingForSecondClick which we will set to False. **Use the ProjectOverviewer to create the variable**.

We will now write the big handler: click of Card, in stages, making it more complex at every stage. You can choose to define each version as we go along or to wait till the handler is complete to type it in.

A good start is a click handler that plays the Card's sound and marks the card as played (by setting its fillcolor to Red). **Focus the ObjectEditor on the Card object by typing "Card" (without the quotes) in its top most panel. Press Return**. **Now add the click handler. When the SK8ScriptEditor appears, type the following**:

```
on click of me (a Card)
   play my mySound
   set my fillcolor to Red
end click
```

But we do not want to allow the user to click on a Card that is already Red (meaning it has already been selected). Thus, if the Card is already red, we should do nothing. We change the handler as follows:

```
on click of me (a Card)
   if my fillcolor ≠ Red then
      play my mySound
      set my fillcolor to Red
   end if
end click
```

You can get the "≠" character by pressing Option-=.

Next we need to determine whether this click is the first or the second click of the pair. If it is the second click, we need to do some work to find out whether a match has occured. To determine this, we use the value of the waiting ForSecondClick variable. If this variable is set to True, this click is the second click. Who sets this variable to True? We do, when we determine that this click is the first click. Again we refine the handler:

```
on click of me (a Card)
   if my fillcolor ≠ Red then
      if not waitingForSecondClick then
         play my mySound
         set my fillcolor to Red
         set waitingForSecondClick to True
      else
         play my mySound
         set my fillcolor to Red
         set waitingForSecondClick to False
      end if
   end if
end click
```

We are now done with the first case (processing the first click). When the second click happens, we need to see whether a match has occured (the two sounds of the cards clicked are the same). To find this out, we need to find the two cards clicked and compare their sounds. We can do it as follows:

```
set playedSounds to the mySound of every Card whose fillcolor = Red
   in the contents of CardHolder
if item 1 in playedSounds = item 2 in playedSounds then
   --a match!
else
   -- no match.
end if
```

And we can define a function to test this case. Create a new function called “weHaveAMatch” with no arguments an the text above. The complete function would be:

```
on weHaveAMatch
   set playedSounds to the mySound of every Card ¬
      whose fillcolor = Red in the contents of CardHolder
   if item 1 in playedSounds = item 2 in playedSounds then
      return True
   else
      return False
   end if
end weHaveAMatch
```

You can get a “¬” (continuation) character by pressing Option-Return to jump to the next line.

Let us now add this test to the second part of the Click handler. If there is no match we want to do the following:

- set the fillcolor of the clicked cards to White.
- change the CurrentPlayer to the other player.

The click handler now looks like this (only the part that processes the second click is shown):

```
else
   play my mySound
   set my fillcolor to Red
   if weHaveAMatch() then
      -- do something
   else
      set the fillcolor of every Card ¬
         whose fillcolor = Red in CardHolder to White
      if the Board's currentPlayer = Player1 then
         set the  Board's currentPlayer to Player2
      else
         set the Board's currentPlayer to Player1
      end if
   end if
   set waitingForSecondClick to False
end if
```

And the last thing is to figure out what to do when we do have a match. Well, we should do the following:

- Add 1 to the score of the current player.
- Set the fillcolor of the two cards clicked to White and remove themfrom the Board. We can remove them by hiding them.
- Check if the total score is 50. If so, the game has ended and we need to report the result.

And here is the result of these changes (only the case with the match is shown):

```
if weHaveAMatch() then
   -- add 1 to the score of the current player.
   set goodPlayer to the Board's currentPlayer
   set goodPlayer's score  to 1 + goodPlayer's score
   -- color clicked cards white and hide them.
   set clickedCards to every Card ¬
      whose fillcolor = Red in CardHolder
   set the fillcolor of every item in clickedCards ¬
      to White
   hide every item in clickedCards
   -- check if the game has ended.
   if Player1's score + player2's score = 50 then
```

```
      messageToUser "Game Over"
   end if
```

And hopefully we are done. The text of the whole handler is presented below:

```
on click of me (a Card)
   if my fillcolor ≠ Red then
      if not waitingForSecondClick  then
         play my mySound
         set my fillcolor to Red
         set waitingForSecondClick to True
      else
         play my mySound
         set my fillcolor to Red
         if weHaveAMatch() then
            -- add 1 to the score of the current player.
            set goodPlayer to the Board's currentPlayer
            set goodPlayer's score  to 1 + goodPlayer's score
            -- color clicked cards white and hide them.
            set clickedCards to every Card ¬
               whose fillcolor = Red in CardHolder
            set the fillcolor of every item in clickedCards ¬
               to White
            hide every item in clickedCards
            -- check if the game has ended.
            if Player1's score + player2's score = 50 then
               messageToUser "Game Over"
            end if
         else
            set the fillcolor of every Card ¬
               whose fillcolor = Red in CardHolder to White
            if the Board's currentPlayer = Player1 then
               set the  Board's currentPlayer to Player2
            else
               set the Board's currentPlayer to Player1
            end if
         end if
         set waitingForSecondClick to False
      end if
   end if
end click
```

Activate the handler and close the SK8Script Editor.

Anything Left?

Yes, but not much. We still need a way to show who the current player is, and a way to let our users start a new game. Both are very simple.

A simple way to show who the current player is, is to redefine the set currentPlayer handler of the Board to highlight the score of the current player. **Focus the ObjectEditor on the Board. Add a new handler: "set currentPlayer". In the SK8Script Editor type the following**:

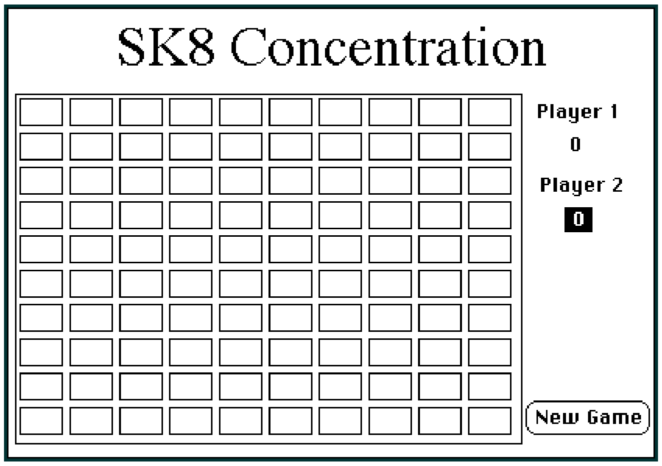
```
on set currentPlayer of me (a Board) to newValue
   -- Do the actual property access:
   do inherited
   if newValue = Player1 then
      set the highlight of Player2Score to False
      set the highlight of Player1Score to True
   else
      set the highlight of Player1Score to False
      set the highlight of Player2Score to True
   end if
end set currentPlayer
```

Activate the handler and close the editor. You can now try this handler out by changing the Board's currentPlayer. In the messageBox, you can type:

```
set the Board's currentPlayer to Player2
```

And finally, we need a way to let users start a new game. We will add a button and define its mouseDown handler to start the game.

We will draw a RoundRect using the DrawPalette. **Go to the palette's menu and select "Shapes"**. This will show the RoundRect tool. **Draw a RoundRect at the bottom right of the Board. Then set its text to "New Game"**. The final state of the Board is,



Let us now define the mouseDown handler of this new button. From the ObjectEditor or the SelectionHalo's menu, **add a new mouseDown handler**. In the SK8ScriptEditor, **type the following text**:

```
on mousedown of me
   do inherited
   show every Card in the contents of CardHolder
   distributeSounds()
   set player1's score to 0
   set player2's score to 0
   set the Board's currentPlayer to Player1
end mousedown
```

And we are done! The perfectionists in the room will be bothered by the fact that our "New Game" button does not highlight when the mouse goes down on it. To appease them, type in the MessageBox:

```
set the autohighlight of RoundRect 1 in the Board to True
```

And now we are really done. Save the project and then try out the game.

Basic Concepts

This chapter introduces the basic concepts of SK8: the concepts you have to understand in order to use SK8 effectively. These are presented in one page sections. Most of the material presented here is revisited and expanded in later chapters of the manual.

This chapter assumes familiarity with the SK8 Project Builder as well as some basic SK8Script.

Objects

Objects in SK8 are like nouns in the real world. An object is a “thing”. If you can refer to it, it is an object. Objects are defined in terms of other objects.

An **object** is defined by its **features** and its relationship to its **parents**.

The features of an object are its **properties** and **handlers**. An object can be a **child** of one or more parents. A child **inherits** the features of its parent or parents. Since a parent object inherits the features of its parents, the child will inherit the features of all of its **ancestors**.

In SK8 everything is an object. A character is a `Character` object. An integer is an `Integer` object. Text strings, lists, and numbers are all considered objects. Base types from which other structures or records are constructed do not exist in SK8.

As an example, consider creating a “Person” object which we could use to store information about people in a Rolodex type application. We create the object and add some appropriate properties to it:

```
new object with objectName “Person”
addProperty Person, ‘name’
addProperty Person, ‘address’
addProperty Person, ‘phoneNumber’
```

Now we can fill in the properties and use the object we have created.

```
set Person’s name to “Julio”
set Person’s phoneNumber to “312-5748”
```

And we can enrich the object by defining a new handler for it that makes some use of the information in it. Assuming the “dial” function existed we could define the following:

```
on call of me (a Person)
   dial my phoneNumber
end call
```

And we can now call the `Person` object. We can also make children of the `Person` object which will inherit all the capabilities of `Person` itself: its properties and handlers.

```
set x to new Person with name “George Smith” with phoneNumber
“543-7865”
```

Properties

Properties in SK8 are like adjectives in the real world. A property is storage space in an object. This storage has a name and can hold exactly one value. This value can be anything ranging from `False` to a collection of values; including a list, an array, or any kind of object.

As an example, we create an object and add a property to it:

```
new object with objectName “Month”
addProperty Month, ‘numberOfDays’with initialValue 31
```

Because a property belongs to an object, both the property name and the object must be specified in order to reference the property’s value.

```
get the numberOfDays of Month
```

An object inherits all of the properties of its parents, its grandparents, and so on. Thus if we create a new `Month` object we will see that it will have a `numberOfDays` property whose value will be initialized to `31` (the `numberOfDays` of its parent).

```
new Month with objectName “September”
```

And we can specialize this month to have 30 days instead of 31.

```
set the numberOfDays of September to 30
```

All properties of an object can be found using the `properties` handler. For example the following line would return all the properties the `Month` object has.

```
get the properties of Month
```

This will return all the properties, including the ones it has inherited from all its ancestors (the object `Object`). We can get a list of the properties that `Month` has added to itself by asking for its `localProperties`.

Properties can be added to or removed from an object at any time. The following line would remove the `numberOfDays` property from `Month` and all its descendants:

```
removeProperty Month, ‘numberOfDays’
```

Note

In other object oriented systems, properties are known as “fields”, “slots”, or “instance variables”. ◆

Propagatable Properties

Propagatable properties transmit their values to descendants. Normal (non propagatable) properties only transmit their values when the children are created: any subsequent changes in value are not propagated.

Let us show the difference between a propagatable property and a normal property with a simple example: representing movies for a video store rental system. In our simple example, every movie has a title and a price. Each individual movies has a distinct title but every movie has its rental price propagated from the object `Movie`. That way, when the price of a rental goes up, all we have to do is change the `rentalPrice` of the `Movie` object and all its descendants will be updated.

```
new object with objectName “Movie”
addProperty Movie, ‘title’ with initialValue “No Title”
```

Since the rental price can change, we will make this property be propagatable.

```
addProperty Movie, ‘rentalPrice’ with propagatedValue
set Movie’s rentalPrice to 3.75
```

Let us make a new movie to see how the values of the two properties are inherited.

```
set x to new Movie
```

When SK8 makes a new object it copies the values of all its properties from the value in its parent. Thus, `x`’s title is `“No Title”` and its price is `3.75`.

In the case of normal properties, when the value of the property changes in the parent it is not updated in the children.

```
set Movie’s title to “Still No Title”
```

`x`’s title has not been updated: it still is `“No Title”`. If we now changed the `rentalPrice` of the `Movie` object, the price will be changed for it and any of its descendants.

```
set Movie’s rentalPrice to 4.00
```

And we can now see that `x`’s `rentalPrice` has also become `4.00`. This automatic update happens until the child overrides the value of the property by setting it directly. So, if `x` was an extremely rare movie we might want to increase its rental price to some ridiculous setting:

```
set x’s title to “The Zaragoza Manuscript”
set x’s rentalPrice to 25.95
```

And now changing the `rentalPrice` of `x`’s parent (`Movie`) will have no effect on `x`’s `rentalPrice`.

Handlers

Handlers in SK8 are similar to verbs in the real world. The behavior of objects and operations on objects are performed by handlers. A **handler** is a named piece of code that is executed in response to a message or event. “Handler” is short for “message handler”. Handlers are procedures available to an object that specify some action to be performed. A handler is the name of a behavior that the object is asked to perform. Other worthy features of handler are:

- they are inherited just like properties. An object inherits all the handlers defined on each of its ancestors.
- they can return values.

To illustrate some of these features, let us create a rectangle called “Beeper” sitting on a window on the `Stage`.

```
new Rectangle with objectName “TheWindow” with container Stage
   with boundsRect {100,100,300,300}
new Rectangle with objectName “Beeper” with container TheWindow
   with autohighlight
```

When the mouse goes down on `Beeper`, a `mouseDown` event is generated. All this means is that the `mouseDown` handler of `Beeper` is called. Note that we have not taught `Beeper` how to respond to `mouseDown` messages, but that this is not a problem because `Beeper` inherits a `mouseDown` handler that is defined on one of its ancestors: .

The `mouseDown` handler of `Actor` takes care of highlighting the actor when the actor’s `autohighlight` property is set to `True`. And this is what we get if we `mouseDown` on `Beeper` since the `autohighlight` property is set to `True`. But we want `Beeper` to beep when the mouse goes down on it. For this, we define its own `mouseDown` handler to beep:

```
on mouseDown of me (a Beeper)
   beep
end mouseDown
```

And now, `Beeper` will beep when we mouse on it. If you try this out you will notice that `Beeper` no longer highlights. This is because when the mouse goes down on `Beeper`, we actually run the most specific handler we can find. Since `Beeper` does define the `mouseDown` handler, that is the handler we run, which just beeps.

There is a way to be able to also run the handler an object has inherited: using the `do inherited` command. In our example, we can get `Beeper` to beep and then do the usual highlighting behaviour by modifying the handler as follows:

```
on mouseDown of me (a Beeper)
   beep
   do inherited
end mouseDown
```

The handlers of an object can be found using a handler called `handler`.

Garbage Collection

SK8 uses automatic garbage collection to manage memory. This means that the memory taken by objects is reclaimed when the objects are no longer used and that you as a SK8 developer do not need to worry about cleaning up the unwanted objects.

SK8 determines whether an object is used by checking if anything refers to the object. If nothing in the environment is pointing to the object, the system concludes that there is no way that you can refer to the object in question and thus the memory can be reclaimed.

Consider the following example in which we create an object and put it in a variable:

```
set x to new object
```

This object will not "go away" since it is being refered to by the variable `x`. If we now set the variable `x` to something else, nothing will be refering to the object and thus it will be reclaimed.

```
set x to False
```

These are the ways in which you can prevent objects from being reclaimed by the system:

- by storing them somewhere: a variable, a constant, a property, a collection, etc.
- by naming the objects (setting their `objectName` property to a string). For example, an object created with the following line would not be reclaimed:

  ```
  new object with objectName "Vegetable"
  ```

- (for actors) by putting them in the contents of other actors or the `Stage`.

Projects

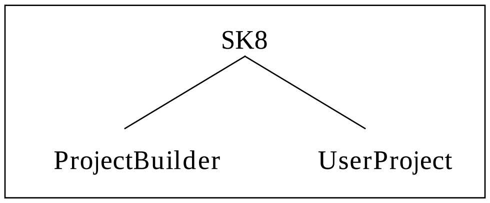
The project is the SK8 unit of work. You do all your work within the context of a project. You save and load whole projects.

More specifically, a project is a SK8 object with which you associate objects, functions, variables and constants.

When you work in SK8, the first thing you are asked to do is create (or open) your own project. Once this project is open, all the objects, variables, constants and functions you create are created in that project.

Each project defines a name space for itself. That means that every named thing that is created in it has to have a distinct name.

Projects are arranged in a hierarchy whose root is the project called SK8. The ProjectBuilder project is a subproject of SK8. User projects are also subprojects of SK8, as shown in the following figure.



A subproject sees everything that is defined in all its super projects. Thus, in your projects, you can use everything that is defined in the SK8 project.

A subproject also shares its name space with the name space of its super project. That means that in your project you cannot create an object called Rectangle because one already exists in SK8.

It is also important to note that a project does not share name spaces with its siblings. Therefore, you could have two distinct objects called “Pipo” in both the ProjectBuilder project and your project.

The ProjectOverviewer is the ProjectBuilder’s window that shows everything that is associated to a project. Thus you can use it to view your project’s objects, functions, variables and constants.

A subtle point is that when you create a project, an actual child of the `Project` object is created for it. In the tutorial, for example, when we created the SK8Concentration project, we actually created a child of the `Project` object, called “SK8Concentration”. We can ask this object, SK8Concentration, for its objects, functions, globals and constants by calling the `objects`, `functions`, `globals` and `constants` handlers respectively.

Functions, Variables and Constants

Besides objects, projects store functions, global constants and global variables.

Functions differ from handlers in that they are pieces of code that are not associated to any object. You can define new functions from the ProjectOverviewer (see the Tutorial for details). Functions are accesed back by using the `functions` handler of `Project`. You can immediately tell the code of a function appart from that of a handler by looking at its header line. Here is an example of a function next to a handler:

```
on add2 of n
   return n + 2
end add2

on mouseDown of me (a Rectangle)
   beep
end muosedown
```

You can see that the handler contains in its header line the `me` argument followed by a specification of the object type that responds to this message. The function just has a list of arguments.

Constants are place holders for values, that live outside the scope of any handlers. The main feature of a constant is that once you define it, it is an error to change its value. This is how a constant is defined in SK8Script:

```
global constant numberOfPlayersInASoccerTeam = 11
```

Global variables are just like constants but their values can be changes as many times as you want. This is how global variables are defined and later changed:

```
global x = 10
set x to 20
```

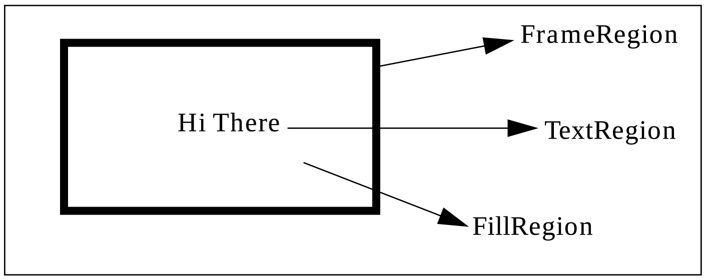
Once created, variables and constants can be removed by using the ProjectOverviewer or by using the `removeVariable` and `removeConstant` handlers of `Project` as shown in the lines below. Assume the project in which we have create our variables is called “MyProj”,

```
removeVariable deleteme with name 'x'
removeConstant deleteme with name 'numberOfPlayersInASoccerTeam'
```

Actors

Actors are graphical objects. Any object in SK8 that has a graphical representation is an `Actor` (descends from the `Actor` object).

Geometrically, an `Actor` is defined by three regions or areas which are shown in the following figure. The `frameRegion` frames the actor's bounds with a thickness that is determined by the `frameSize` property. The `fillRegion` is everything inside the frame and the text region is the area occupied by the actor's text if any.



All drawing in the system is done by objects that descend from `Renderer`. Each actor has a renderer associated with each of its regions. The `fillColor` paints the `fillRegion`, the `frameColor` paints the `frameRegion` and the `textColor` paints the text region.

The most important feature of actors is that they can be put inside other actors. Each actor has a property called `contents` in which it stores all the actors that are inside of it. Actors inside other actors are drawn inside the actor's `fillRegion`. Windows are nothing more than actors whose `container` has been set to the `Stage`.

To summarize, actors provide the following capabilities:

- containment: the ability to contain other actors.
- scale: each actor defines a logical scale for actors that go inside of it. This allows arbitrary zooming of the contents.
- origin: each actor has a local coordinate system whose origin can be moved. This allows arbitrary panning of the contents.

The direct children of Actor define the geometries available in the system. These are: `Rectangle`, `RoundRect`, `Oval`, `Polygon`, `LineSegment`, `MaskedActor` (gets is geometry from the mask of an IconRSRC), `Halo` and `SelectionDots`.

You can easily define new geometries. See the Actor chapter of this guide for details.

The Stage

The `Stage` is the place where everything graphical takes place. More concretely, the `Stage` is the sum of your monitor space (or in the Macintosh, the desktop).

You can find the dimensions of the `Stage` by finding its `boundsRect`. The 4 numbers returned define the rectangle within which all your monitors fit. To get it, you can type

```
get the boundsRect of the Stage
```

Even though the `Stage` is not an actor, it also has contents which are actors. By making an actor an item in the `contents` of the `Stage` the actor becomes a window. In order to be visible, an actor has to be (ultimately) contained by the `Stage`.

Let us create a rectangle and add it to the contents of the `Stage`.

```
new Rectangle with objectname “MyWindow”
set myWindow’s container to the Stage
```

The `Stage` defines what we call the physical (or Stage) coordinate system. The unit of measurement is the pixel. When you get the `boundsRect` of the `Stage`, you get the number of pixels that you can display in the sum of your monitors. The origin of this system, the point {0,0}, is located at the top left of your main monitor (the monitor with the Macintosh menubar on it).

If you entered the lines above in the MessageBox, you will see a small white rectangle appear close to the top left of your main monitor. Let us now use the `Stage`’s coordinate system to change the position and size of our window. We can do this by setting its `boundsRect`, as follows:

```
set the boundsRect of myWindow to {100,100,300,300}
```

What this means is that the top left corner of `myWindow` will start at the point that is 100 pixels down and to the left of the top left of your main monitor. The bottom right corner is located 200 pixels to the right and bottom.

Containment

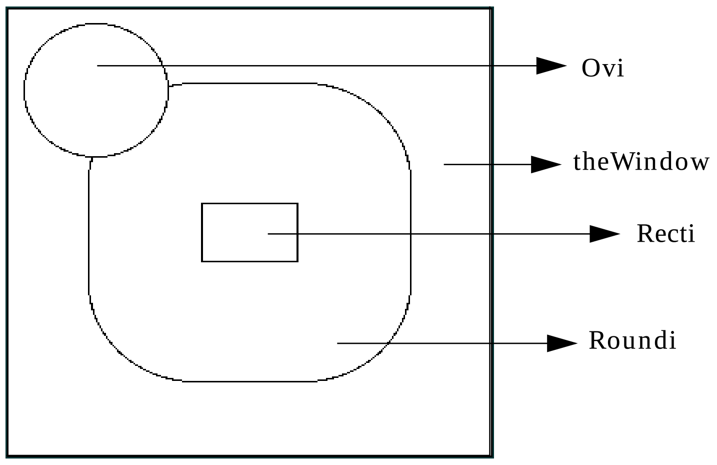
Containment is the mechanism to associate, group or composite separate actors into more complex actors. Each actor has two properties that are used to implement containment: the `container` and the `contents` property.

Actors can be contained by other actors, the `Stage` or False (no container). In order to be visible, an actor has to be ultimately contained by the `Stage`.

Let us create a few actors to illustrate the most important features of our containment model:

```
new Rectangle with objectname "theWindow" ¬
   with boundsRect {200,200,500,500} with container Stage
new RoundRect with objectName "Roundy" ¬
   with boundsRect {50,50,300,300} with container theWindow
new Oval with objectName "Ovi" ¬
   with boundsRect {10,10,100,100} with container theWindow
new Rectangle with objectName "Recti" ¬
   with container Roundy with location {100,100}
```

And here is the resulting actor:



This very simple containment model provides the following features:

- unlimited levels of containment. There no arbitrary limit to the number of containers an actor can have on its way to the `Stage`.
- clipping. Actors are automatically clipped to the `fillRegion` of their immediate containers.

To prove this claim, select `Recti` using the selection tool and drag it. Notice that when `Recti` is taken beyond the `fillRegion` of its container, it is clipped from view. The same thing happens when we drag `Roundi` outside the `fillRegion` of its container (`theWindow`).

- layering. All the actors inside a container are ordered by layer. Actors closer to the front cover actors behind them.

In our example above, notice that `Ovi` is partially obstructing `Roundi` from view. This is because `Ovi` is in the front of `Roundi`. You can change the layering by setting the `layer` property of all actors. You can do this by direct manipulation by selecting `Ovi` and then selecting an item from "Layering" menu inside the SelectionHalo's menu.

- independent coordinate systems with arbitrary scales and origins.

Each actor defines its own, local coordinate system in which its contents live. When the `location` or `boundsRect` of an actor is specified, it is always done with respect to the coordinate system of its `container`.

To explore this, let us examine the lines of SK8Script we used to create the actors above. The first line was:

```
new Rectangle with objectname "theWindow" ¬
   with boundsRect {200,200,500,500} with container Stage
```

Which means that the new rectangle would be place in the contents of the `Stage`, its top left corner would be at location `{200,200}` and its bottom right corner at location `{500,500}`. These locations are expressed in the coordinate system of `theWindow`'s container: the `Stage`. Thus {200,200} means 200 pixels to the left and bottom of the top left corner of your main monitor. The next line reads:

```
new RoundRect with objectName "Roundy" ¬
   with boundsRect {50,50,300,300} with container theWindow
```

Which placed `Roundy` 50 logical units to the right and bottom of the top left corner of its container: `theWindow`. The real (physical) size of a logical unit is given by the `scale` of an actor. Since the `scale` of `theWindow` is set to 1 (the default), a unit equals one pixel. By changing the `scale`, you zoom in or out the contents of the actor.

The `location` (or `boundsRect`) of any actor can be expressed in logical or physical terms. Logical location is the location of the actor relative to the coordinate system of its container. Thus, the location of `Roundy` is `{175,175}` (the center of its `boundsRect`). Physical location is the location of an actor relative to the coordinate system of the `Stage`: the location expressed in pixels from the top left of the main monitor of your system. The following line returns `Roundy`'s physical location:

```
get the location of Roundy with physical
```

The SK8 architecture is devoted to the preservation of the logical location of actors. Thus, when you drag an actor that has contents, everything in the contents moves with the actor since the logical location of the contents does not change as the container moves. For a demonstration, try dragging `Roundy`.

Project Builder Overview

This chapter provides a reference to all of the tools inthe Project Builder. This chapter fleshes out many of the tools used in the previous tutorial.

What is the Project Builder?

The Project Builder is the primary user interface to SK8. It is designed to aid the rapid development of individual SK8 projects and it includes a set of tools which provide access to all of SK8's power in a simple direct manipulation style.

Since the Project Builder was developed and implemented in SK8, it is a SK8 project like any other. Thus the Project Builder provides an excellent illustration of SK8's capabilities and how they can be used to build an authoring tool.

Note
Project Builder prefers the "Espy" font be included in the system fonts folder. ◆

An Overview

Project Builder At Startup

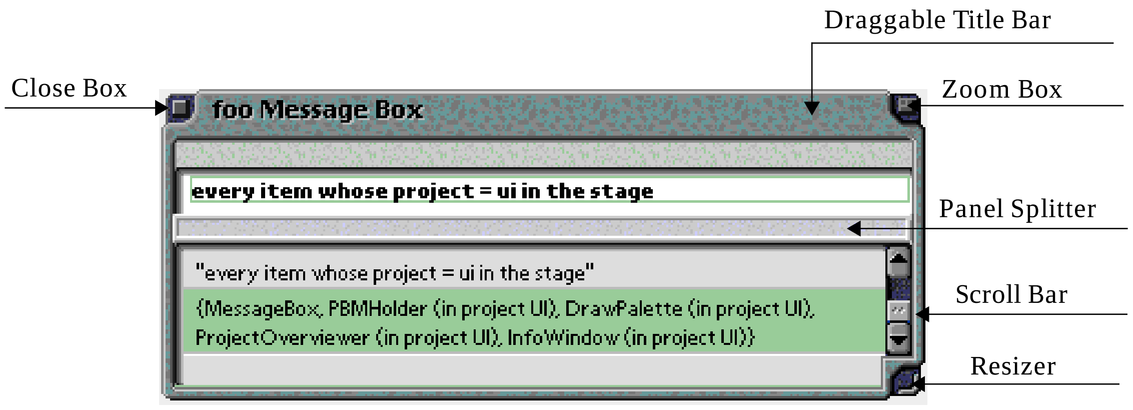
Upon initial startup of SK8, the following Project Builder interface tools are displayed:

- Message Box
- Draw Tools
- Object Editor

After a new project is created, the following Project Overviewer is displayed.

The Project Builder Windows

The Project Builder interface has windows which are characterized by a unique appearance. This appearance is designed to be similar enough in design to the Macintosh to be usable, but different enough so that you will not confuse it with the application that you are building. Below is a sample window with its parts labeled:



foo Message Box

```
every item whose project = ui in the stage
```

```
"every item whose project = ui in the stage"
{MessageBox, PBMHolder (in project UI), DrawPalette (in project UI),
ProjectOverviewer (in project UI), InfoWindow (in project UI)}
```

One additional feature of these window is that doubleclicking on their titlebar activates a "Window Shade" type functionality.

Keyboard Focus

In many of the tools there are multiple panels where you can type. The field which is highlighted in white is the active typing field. For example if the Message Box (shown above) is not selected, both panels are gray. When the window is active, as in the diagram above, one of the panels becomes white to show that any keystrokes will be sent to this panel. Within any window, you can use the Tab key to change the focus from one of the panels to the next:

Drag and Drop

The Project Builder interface fully supports drag and drop. The user can drag and drop references to objects, properties, and handlers between the various editors and browsers. Items may be dragged by:

- using the title on the halo,
- using the icons next to text fields,
- mousing down on any of the pickers for an extended period of time.

For example, in the Project Editor, by pressing the “All Objects” button, you can mousedown on a particular object for half a second and a grayish box will appear around the name of the object. This box can be dragged to the top field of the Object Editor and dropped there to cause the editor to focus on this object. Note that you could shift select several objects and then hold the mousedown for half a second and drag a set of objects to edit simultaneously in one of the editors.

It is important to note that you are not dragging the actual objects around, only references. Dragging a reference around will NOT affect the object. Also note, when dragging a reference, the system will alert you to an appropriate place to drop by highlighting to white and in some cases flashing.

Help Key

The Help key can be used to display balloon help on a selected word of text. For example, if you enter the word “idle” in the Message Box, select it, and press the Help key, balloon help is displayed to show where in the project the `idle` handler is defined along with its argument list. Similarly, the help key will bring up more information on any of the items in the panels displaying properties, functions, etc.

The Help key is the Help function key on most keyboards, or the Escape key on a keyboard without function keys.

Updating Windows

In general, the Project Builder will update itself to reflect any actions such as property setting and handler creation done within the Project Builder. It will NOT detect property changes make in SK8Script or in the Message Box. Whenever you wish to update an editor or window, you can simply place the item in the editor again. For example, in the Object Editor you can reselect the item using the history menu or you can drag the icon of the item(s) to the text field and drop it there.

Clearing References

Throughout the Project Builder, you can select an item and press the delete key to remove it. For example, you can select a handler and press delete to remove the handler. This works for properties, handlers, functions, constants and variables. It also works for objects, but in a less direct manner. SK8 supports full garbage collection of objects. This means that as long as there are no references to an object, it will be removed. However, any reference to the object is enough to keep it around. Thus if you select an object in the Project Builder and press delete, the Project Builder will give you the option to clear the standard references to an object or clear all references to an object.

Clearing standard references is usually enough to get rid of an object. Namely, it sets the object’s `objectname` to `false`, it removes it from it’s parent’s `knownChildren` list, and it removes it from it’s `container` (if it is an actor and has one). It also makes sure that that object isn’t referenced by the Project Builder.

Note that if you have a reference to this object in some other object's properties, it will stay around until you remove that reference. For this reason, you can choose to clear all references to an object by pressing the "All" button in this dialog. In so doing it will first clear the standard references as describe above and then it will do an exhaustive (and time consuming) search of all properties, constants and variables in the project to see if any of these refer to the object. If they do there value is set to False, or in the case of constants are removed.

Note that in some cases, such as when your project has a global or property which stores a list of important objects, setting the value to false may not be the preferred action. If you feel your project may contain such references, then you should use the Searcher window (described below) to find anything that refers to your object and manually set these values to the desired value. You can then use the clear all references capability to get rid of the rest.

Note also that garbage collection takes time. After all your references to the object are gone, it may be a period of time before the system gets around to deallocating it.

The most important thing to keep in mind here is that you don't have to worry about getting rid of objects. The system takes care of this for you.

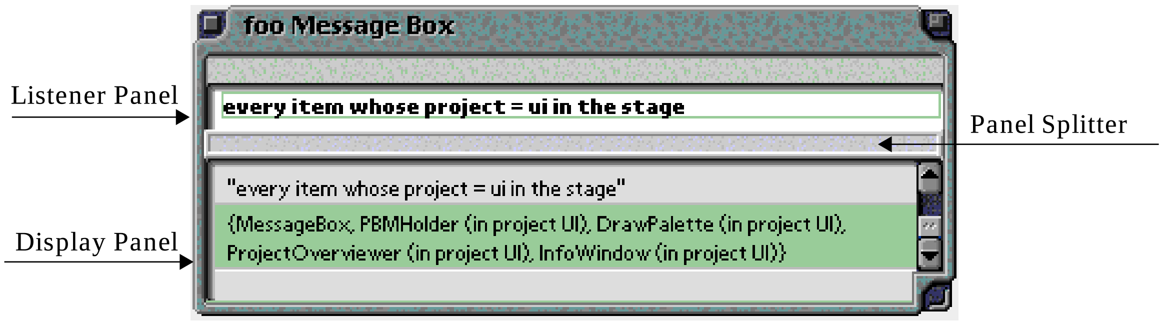
Project Builder Components

Many of the Project Builder windows are designed to provide users with contextual information for their project editing tasks. The Project Builder contains the following main components. A description of each component follows.

- Message Box
- Draw Palette
- Object Editor
- Project Overviewer
- Selection Halo
- Stage Monitor
- Inheritance Overviewer
- System Browser
- Menu Editor
- Media Browser
- Color Palette and Renderer Editors
- Library Editor
- Script Editor
- Stack Watcher
- Handler Tracer
- Documentation Window
- Project Builder Menubar

Message Box

The Message Box serves as both a place to type SK8Script Commands and a place to receive system messages.



```
every item whose project = ui in the stage
```

```
"every item whose project = ui in the stage"
{MessageBox, PBMHolder (in project UI), DrawPalette (in project UI), ProjectOverviewer (in project UI), InfoWindow (in project UI)}
```

Listener Panel

The Listener is a SK8Script command line interface that allows you to talk to SK8 by typing into it. When the Return or Enter key is pressed, Listener evaluates your SK8Script commands and expressions, and responds in the Display Panel.

By typing Option-Return (hitting the Option key and Return key simultaneously), a phrase or script can be continued over more than one line.

By typing a handler name into the Listener Panel and pressing the Help key, a popup balloon appears describing that handler.

Display Panel

SK8 responds via this panel. The display panel prints out a response (error messages, etc.) to the commands entered into the Listener. It also provides status information during the execution of certain operations, such as saving or loading a project. It is the default system “Log”.

If you select an item in the Display Panel and hit Return, the item is copied to the Listener. Double-clicking on the item will also copy it to the Listener. This does not apply in the case of error messages. Pressing enter in this panel will place the item in the Listener Panel and then will evaluate it.

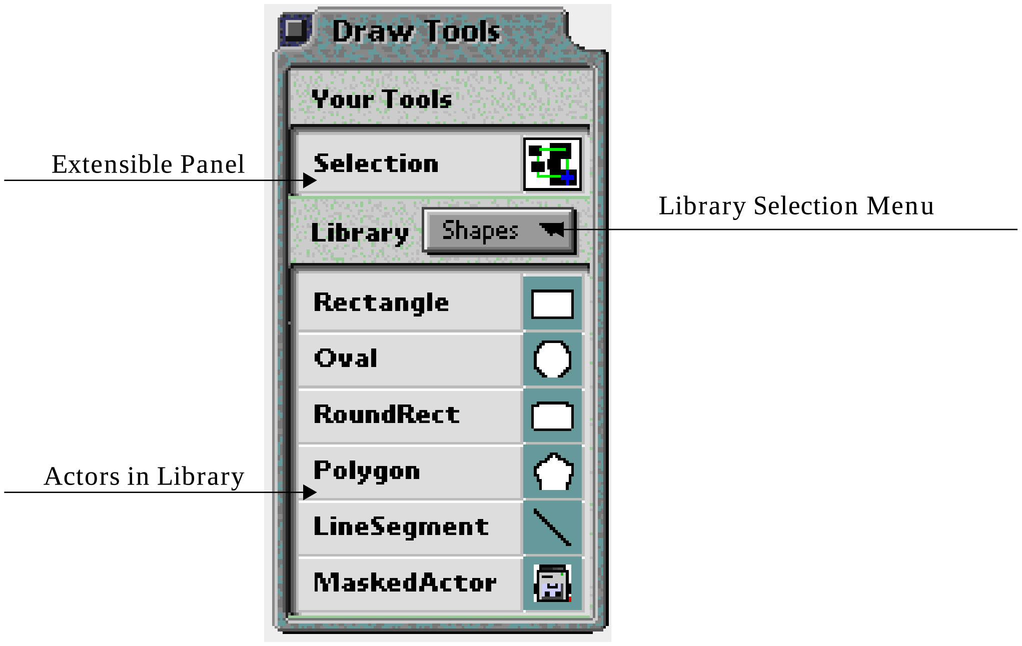
Performing either of the above actions on an error message brings up an error dialog box which provides debugging tools to help the developer.

Items in the display panel are accumulated into a “scrollable history” or history picker. You may pick (select) any item in the history by clicking on it. From here you can move the item to the Listener panel (if not an error message) by the two methods described above. If the item is an error message, you can bring up an error dialog by double-clicking on the error.

The Delete key will remove any selected items from the display panel.

Draw Palette

The Draw Palette provides a set of tools for drawing SK8 actors on the screen.



Description

The draw palette has two panels, each of which displays a set of tools. To use a tool, click on the tool icon and begin drawing. You can also get options for drawing by briefly holding the mouse down on a tool.

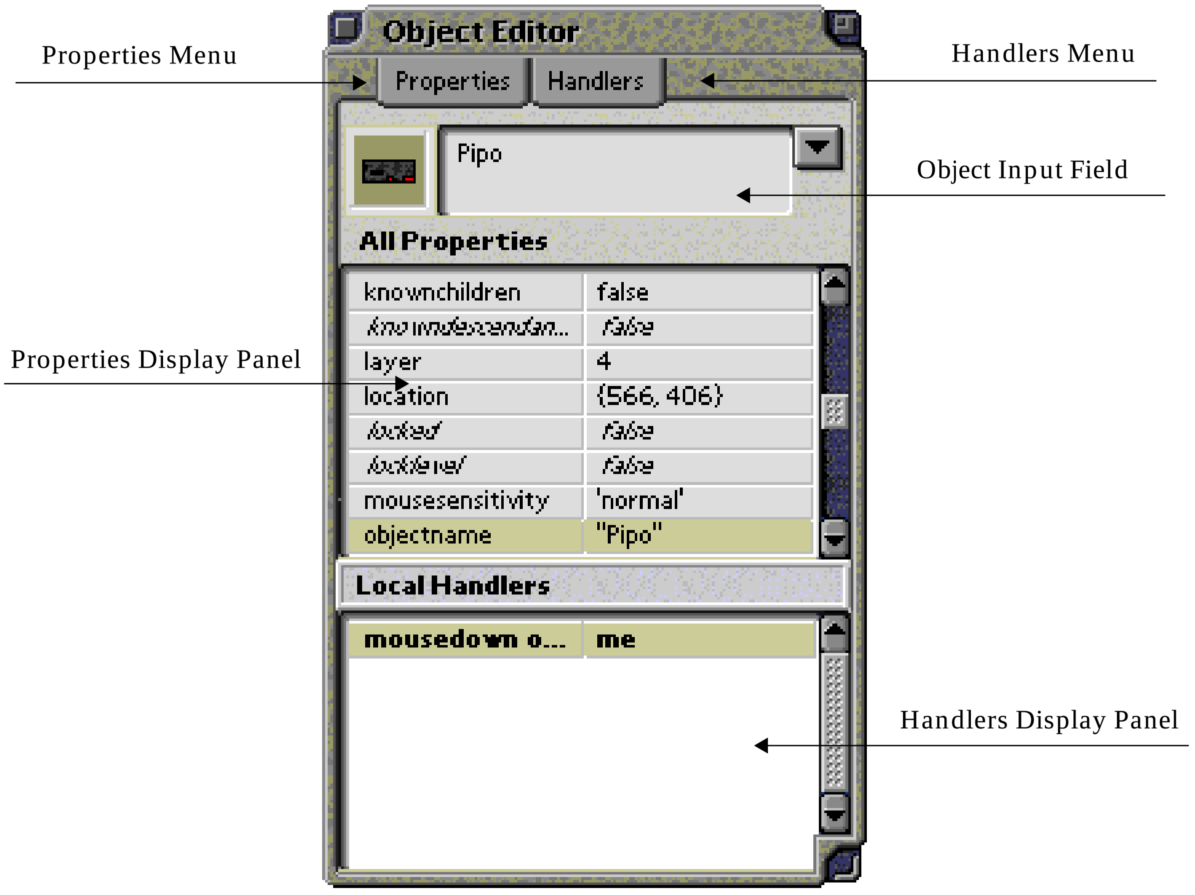
The upper panel displays your personal extensible set of tools. By default, it contains the Selection Tool. You can drag references to one or more named actors and drop them on this panel to create tools for drawing those actors. An actor which is added to a palette is marked as a prototype, (i.e. its `prototype` property is set to `true`).

Remove a tool by selecting it and hitting the Delete key. Note that removing a tool does not affect the actor.

The lower panel displays libraries of actors. What library is shown is controlled by the popup menu just above it. This popup contains a list of all the open libraries in SK8 as well as in your project. Selecting a different library will cause a tool to be created for all of the actors in the library marked as prototypes (their prototype property is set to true). It is possible to drag tools from the lower panel to the upper one by briefly mousing down on the name of the object and dragging it to the top panel. You can also drag these references to any Project Builder editor (e.g. the Documentation Window) to get more information on the object and its functionality.

Object Editor

The Object Editor allows the viewing and modification of objects. It displays all aspects of the objects, including their properties, property values, and handlers. It allows editing of each of these aspects.



knownchildren	false
knowndescendan...	*false*
layer	4
location	{566, 406}
locked	*false*
locklevel	*false*
mousesensitivity	'normal'
objectname	"Pipo"

mousedown o...	**me**

Description

Each object editor consists of the components in the above diagram. Each of which is described below. Note that it is possible to have multiple Object Editors active at one time. In addition, note that an Object Editor can edit more than one object at a time. The user can do this by dragging a set of objects into the Object Editor (using shift selection in one of the pickers or dragging a list of objects) or by typing a SK8Script expression that returns multiple objects (e.g. every rectangle in the stage whose fillcolor is blue).

Properties Menu

Provides the following options for property display:

Add Property	Brings up a dialog to add a property to the edited object(s).
Remove Property	Brings up a dialog to remove the selected property.
Show Private Properties	Cause the editor to include private properties in the properties panel.
Edit Property Attributes	Brings up the Property Control Panel to view and edit the selected property.
Add Port	Brings up a dialog to add a port to the selected property.
Show All	Show all properties of given object.
Show Inherited	Show only inherited properties
Show Parents	Show only local and parents local properties
Show Local	Show local properties only.
Show Graphic	Show only properties related to appearance

Handlers Menu

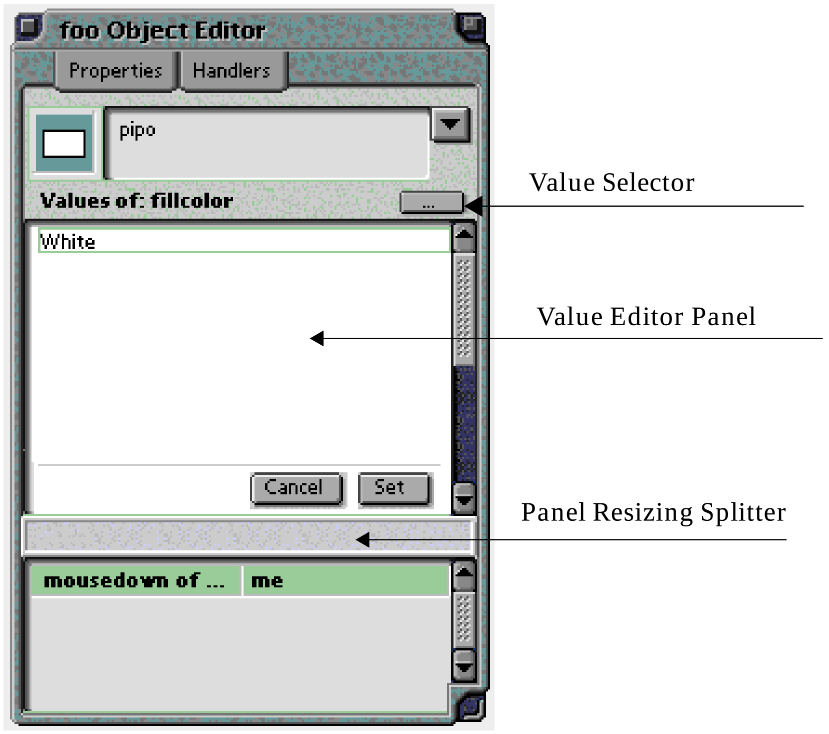
Provides the following options for handler display:

Add Handler	Brings up a dialog to add a handler to the edited object(s)
Remove Handler	Brings up a dialog to remove the selected handler.
Show Private Handlers	Cause the editor to include private handlersin the properties panel
Show All	Show all handlers of given object.
Show Inherited	Show only inherited handlers
Show Parents	Show only local and parents local handlers
Show Local	Show local handlers only.
Show Graphic	Show only handlers related to appearance

Properties Display Panel

Shows the selected object’s properties and values

A property value for an object is edited by double-clicking on the property or selecting the property and pressing return or enter. The properties display panel is replaced by a Value Editor panel shown below:



You can type any SK8Script expression (e.g. “the fillColor of SuperButton”) in the Value Editor panel. Press the Set button or Return key to evaluate the expression and set the property, or press the Cancel button or Escape key to leave the property unchanged.

Note that there is a menu item to bring up a property control panel on the selected item. As a short cut, command double-clicking on a property will do the same.

The Value Selector button (top right) displays different value lists from which to select a value, such as, a list of window styles, a list of colors, etc.

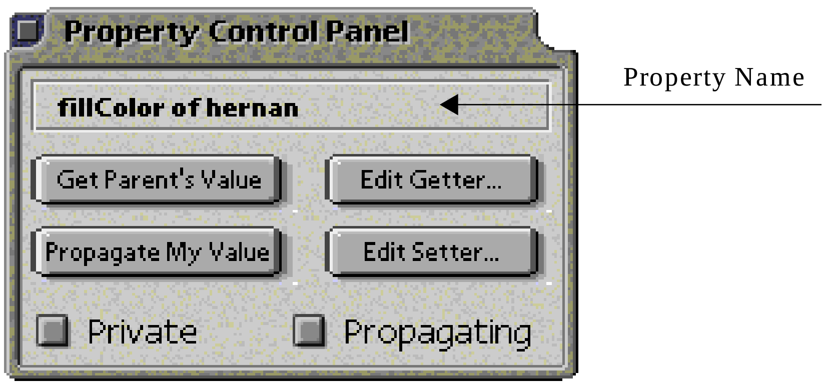
Note
As a special convenience, if you hold down the Command key while clicking the Set button, not only will the property be set for the object, but for all the descendants of this object. This is useful, for example, while editing the graphical properties (e.g. fillcolor) of a prototype which already has many children. ◆

Handlers Display Panel

Shows the handlers and their arguments, both positional and keyword, currently defined on the selected object. Double clicking on a handler will bring up a Script Editor for that handler (if the script is available). Alternatively selecting a handler and pressing return or enter will bring up a Script Editor for that handler.

Property Control Panel

The Property Control Panel allows the editing of the meta-properties of a particular property; such as making a property propagating or private. This can be brought up through the Edit Property Attributes menu item ofthe Object Editor (see above).



Description

The top portion shows the name of the property you are editing. You can drop a property into this field to change what you are editing.

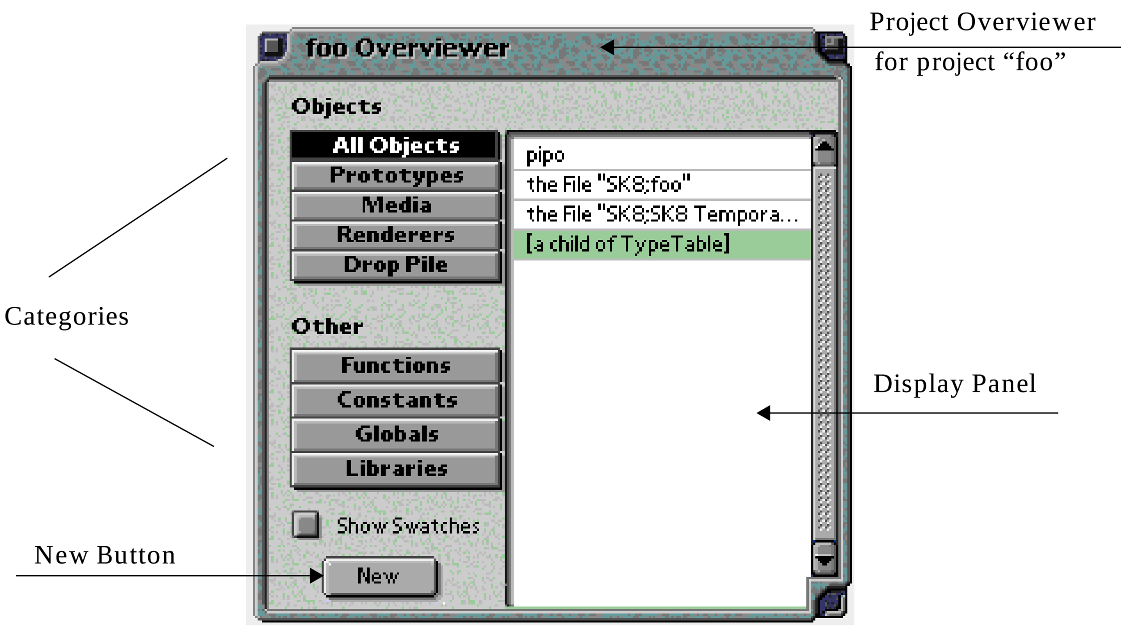
The Private checkbox and the Propagating checkbox at the bottom of the panel allow you to specify whether the property is private or propagating.

The following four buttons allow you to control various aspects of inheritance:

Get Parent's Value	Sets the edited properties value to the value of its parent, or if the property is propagating, it is forced to re-inherit the value from its parent (or wherever the value comes from).
Propagate My Value	Sets the value of this property on all of the object's descendants to its current value, or in the case of propagating property it forces all descendants to re-inherit from this value. This button is particularly useful for graphical properties. For example, if you changed the `fillColor` of a prototype button, you can use this button to make sure that `fillColor` is propagated to all instantiations of that button.
Edit Getter...	Brings up a Script Editor to edit the getter of the property.
Edit Setter...	Brings up a Script Editor to edit the setter of the property.

Project Overviewer

The Project Overviewer provides a common browser where every object in a SK8 project is organized and manipulated. It does this by giving a “bird’s eye view” of a project which can be used for both viewing and editing.



Description

On the lefthand side of the Overviewer, there is a list of selectable categories. These represent the various types of information contained in a SK8 project. Selecting a category, which can be done by clicking on it, causes all members of that category in the current project to be displayed in the scrolling list on the righthand side of the window. There items can be selected, double clicked, deleted and dragged off as references in the standard manner.

Note that items can be deleted by selecting them and pressing the Delete key. Double clicking on an item brings up an editor appropriate for that item. The New Button on the bottom left allows you to create a new item of the selected type (e.g. a new variable).

Note also that when viewing the variables of a project, you can edit the value of a variable by double clicking on itor selecting it and hitting Return or Enter. This brings up a Value Editor panel similar to the one in the Object Editor to allow you to set the value.

There are two special categories in the list. The first is the “Drop Pile”. When the Drop Pile category is selected, a free space appears in the right hand panel. This free space can serve as a storage place for frequently used references. For example, you can drag and drop a set of objects and handlers that you are currently editing. Double clicking on ahandler will bring up a Script Editor for the handler. Note that pressing the delete key in the pile simply removes the item from the pile. It does not get rid of the item.

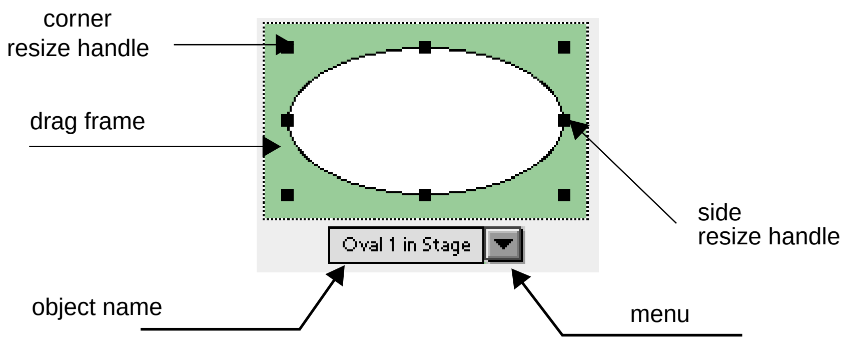
The second special category is “Prototypes”. This category displays every item in the project whose ‘prototype’ property is set to true. Remember that the `prototype` property has no functional effect on an object. It is simply a useful way to mark certain objects as “important”. Thus this category provides a place for you to mark and store “special” objects. The Project Builder marks any objects dropped in this panel (or in the Draw Palette) as a prototype by setting this property to `true`. Selecting an item and pressing delete sets an object’s `prototype` property to `false`, thus removing it from the list.

Note that sometimes you may wish to drag an item from, say, the Drop Pile to the Prototypes category. You can do so by dragging the item from the Pile to the Prototypes category button and dropping it. Thus you drag an item to the category button for either the drop pile or the prototypes.

Note
The Project Overviewer isn’t always immediately updated. If you press on the category a second time it will update the display. For example, if you are viewing prototypes, and you made a new prototype via a script, you can press the “Prototypes” button again and the list will be updated. ◆

The Selection Halo

The Project Builder suports several styles of selecting and editing actors on the stage. By default, actors are “live” and respond to mouse events normally. Thus to modify these objects, there is a “Selection Halo” which surrounds the currently selected actor(s) to allow you to drag, resize and edit.



Description

The selection halo contains four main parts:

- the object name / title

- the menu
- the drag frame
- the resize handles

The object name / title bar, with white text, displays the object name of the object(s) selected. The title is draggable and is used to drag the name to other windows in the Project Builder.

The drag frame allows you to drag objects to any location within a container. An actor's container is changed to the deepest actor the cursor is over after dragging and dropping.

The resize handles let you resize the actor in any direction.

Note

Although the actor can be resized in a particular direction using the resize handles, the origin of the actor remains fixed, so its contents will appear to shift to the new `boundsRect` as their position is defined relative to the upper left-hand corner.

Using the Option Key

Pressing the Option key while resizing, allows toggling between live and non-live resizing. Non-live resizing shows only the outline of the object being resized. Since some objects (e.g., like a rectangle with a large uncached color bitmap) may be slow to resize, the non-live option permits you to resize these objects quickly.

Pressing the Option key while dragging an object or actor , lets you toggle between live and non-live dragging. Note that resizing windows is always done non-live.

Selection Halo Menu

The Selection Halo menu items have the following actions:

Name	Sets the `objectName` of the actor(s).
Edit	Brings up an Object Editor for the selected object.
Deselect	Removes the selection halo and deselects the selection(s).
Clear References	Allows the user to clear standard references to the actor(s) to allow them to be garbage collected. See below for more information on reference clearing.
Tag	Creates a Tag for a part of a complex actor.
Layering	Allows the user to change the graphical `layer` of the selected actor(s) in their container.
Arrange	Brings up a dialog that allows the user to align, distribute or tile the selected objects.
Take a Snapshot	This allows you to take a graphical snapshot of the actor and creates an `imageRenderer` to display it.
NewProperty	Add a property to the selected actor(s).

NewHandler	Add a handler to the selected actor(s).
LocalHandlers	Bring up and editor for one of the existing local handlers from the selected actor.

Keyboard Shortcuts

The following keyboard shortcuts can also be used while the Selection Halo is active:

Clearing:

Delete	Clears references to the selected actor(s). See below for more information on reference clearing.

Changing Selection:

Up Arrow	Select the object that contains the current selection
Down Arrow	Select the first item in the contents of the current selection
Left Arrow	Select the next item in the selection's container's contents
Right Arrow	Select the previous item in the selection's container's contents

Layout:

Option-Up Arrow	Move the selection 1 pixel up
Option-Down Arrow	Move the selection 1 pixel down
Option-Left Arrow	Move the selection 1 pixel left
Option-Right Arrow	Move the selection 1 pixel right
Shift-Option-Up Arrow	Move the selection 10 pixels up
Shift-Option-Down Arrow	Move the selection 10 pixels down
Shift-Option-Left Arrow	Move the selection 10 pixels left
Shift-Option-Right Arrow	Move the selection 10 pixels right

Layering:

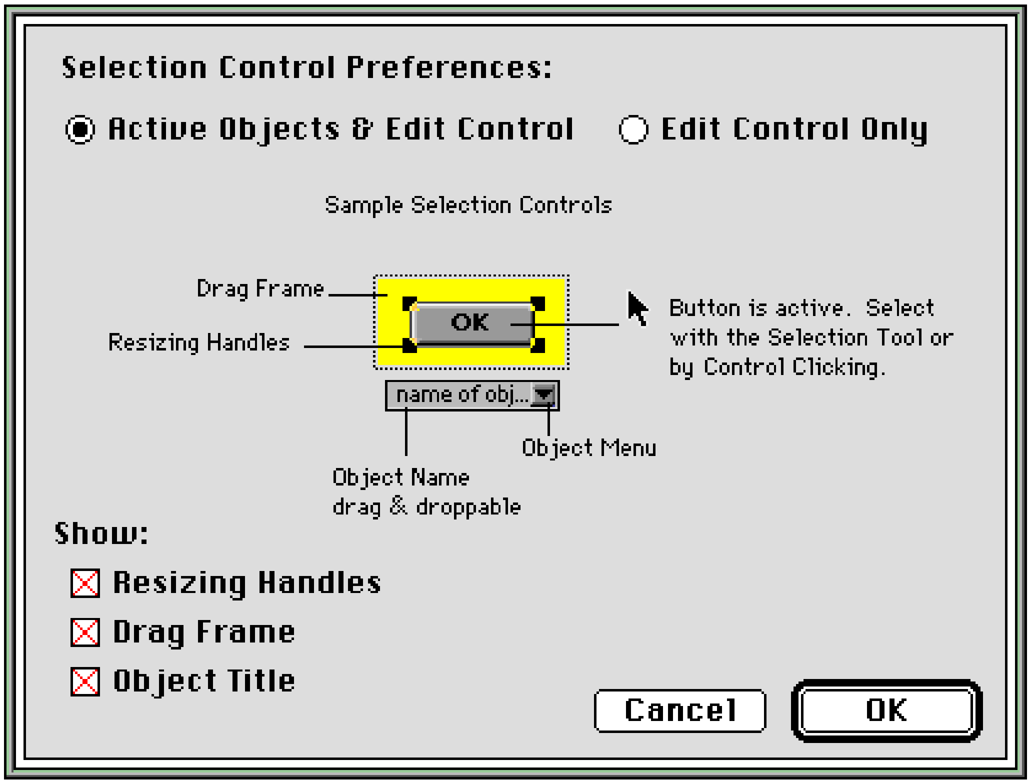
Actors are arranged in a front-to-back order. Thus, an actor can cover another actor. The following keys allow you to move actors through this front-to-back ordering.

+	Bring the selected actor one step closer to the front
-	Send the selected actor one step further to the back
Shift +	Bring the selection all the way to the front
Shift -	Send the selection all the way to the back

Actors also have a `layer` property that can be modified to change layering. The top layer is layer number 1. If you change the `layer` of an actor in the Object Editor the `layer` properties of the other actors in that container will be renumbered accordingly.

Configuring the Selection Halo

In the Workspace menu, there is an item called “Selection Preferences” which is shown below:


Selection Control Preferences:

[● Active Objects & Edit Control] ○ Edit Control Only

Sample Selection Controls

Drag Frame
Resizing Handles
OK
name of obj...
Button is active. Select with the Selection Tool or by Control Clicking.
Object Menu
Object Name drag & droppable

Show:

- [✓] Resizing Handles
- [✓] Drag Frame
- [✓] Object Title

Description

This dialog allows you to configure how you wish to select objects. It provides two main modes:

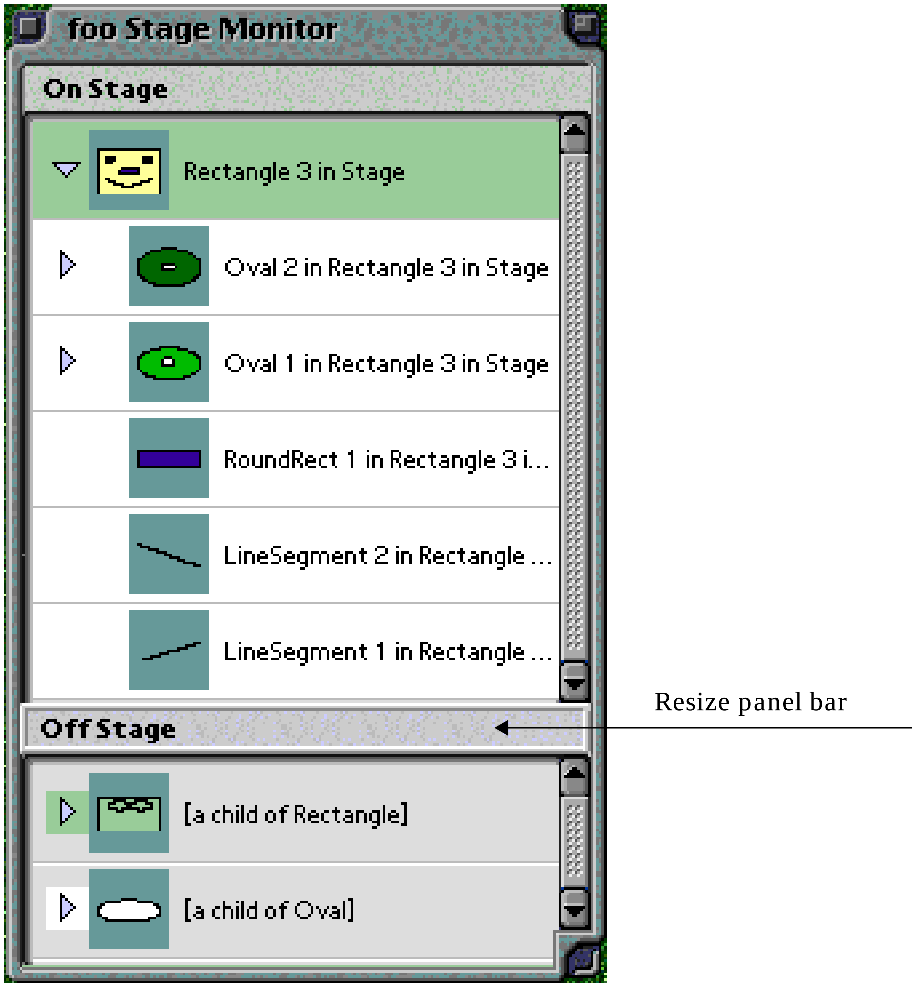
Active Objects	Allows object selection by the standard method of clicking to select an object and mouseDown to drag the object.
Edit Control Only	A mode where the objects do not get mouse events. This mode also determines the configuration of the halo. In general, when you have active objects, you will want all the options of the halo available. For example, when objects are active, the drag frame is needed to move them around. When objects are not active, you can drag them by mousing down on them and therefore a drag frame is not needed. Thus by default, when in Edit Control Only mode, only the resize dots of the halo are shown.
Part Checkboxes	Options of the halo. See Edit Control Only.

Stage Monitor

The Stage Monitor provides an overview of all actors in a project and their containment hierarchies. It also provides the user with a place where visual elements are manipulated on and off the `Stage` (in and out of sight).

The Stage Monitor provides:

- an overview of all Actors in a project, both on and off stage
- direct manipulation of hard to select items (due to the obstructions possibly induced by containment and layering)
- references to all actors
- information about a project's containment hierarchy


foo Stage Monitor
On Stage
Rectangle 3 in Stage
Oval 2 in Rectangle 3 in Stage
Oval 1 in Rectangle 3 in Stage
RoundRect 1 in Rectangle 3 i...
LineSegment 2 in Rectangle ...
LineSegment 1 in Rectangle ...
Off Stage
[a child of Rectangle]
[a child of Oval]
Resize panel bar

Description

The Stage Monitor window has two panels.

The OnStage panel (top panel) displays a list of every Window (top level actor) on the `Stage`. The Stage Monitor is dynamically linked to selected items on the stage. For example, when a new rectangle is drawn on the `Stage`, a reference to that rectangle automatically appears in the Stage Monitor. . Double-clicking on an item in the OnStage panel causes the item to be selected with the Selection Halo.

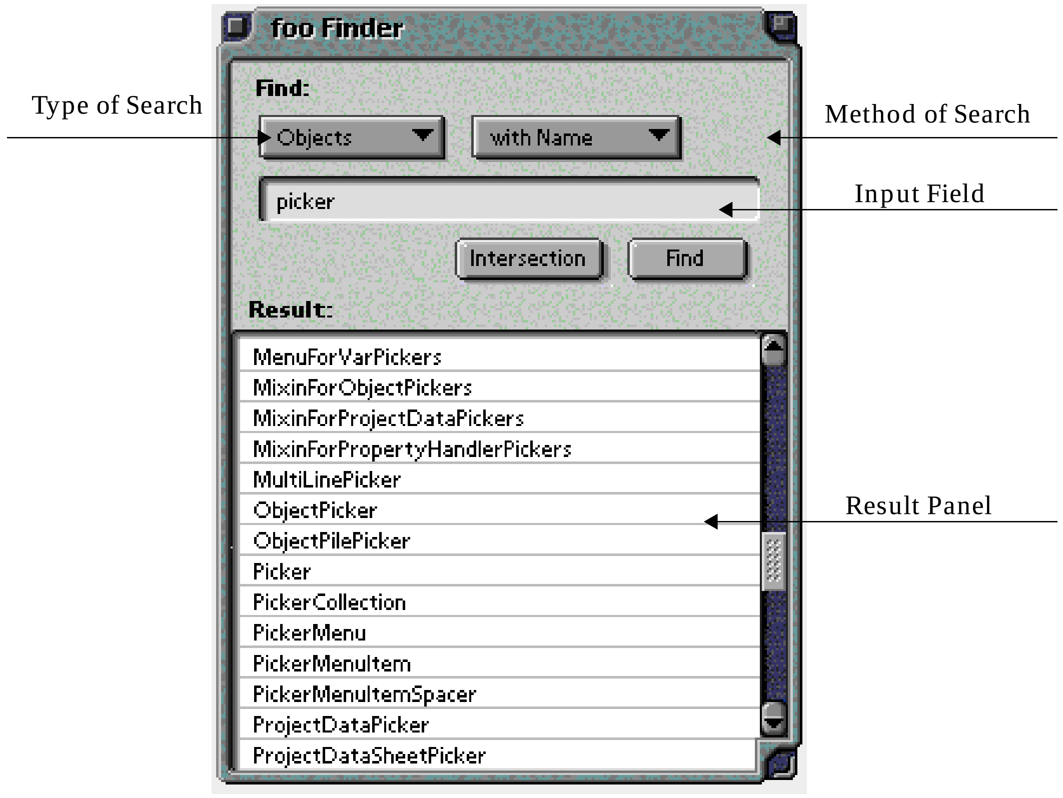
The OffStage panel (bottom panel) shows a list of all Windows that do not appear on the `Stage` at the present time. In other words, these are the windows whose container is equal to `false`. Those actors can be thought of as “waiting in the wings” of the `Stage`.

In both panels, the containment hierarchy of actors is shown in a finder style list fashion. A user can traverse the hierarchy using the standard finder opening/closing triangles. For example, to see a list of what objects are contained in a window on the screen, a user can “open up” a list of contained items by clicking on the finder triangle next to the window's reference in the Stage Monitor.

Items in the Stage Monitor can be edited in the standard manner. References to them can be dragged away in the standard manner (see global changes) as well. In addition, users can edit the containment hierarchy of actors by changing the order of actors in the list by dragging and dropping them into place. For example, you can drag an item from the bottom panel to the top panel to put it on `Stage`. You can drag an item from within one actor and drop it on the swatch of another actor to change its container. Note that you can ensure an actor goes on stage or off stage by dropping it on the title label of the lists.

Searcher

The Searcher helps find objects, properties, handlers, functions, constants and variables in SK8.



```
MenuForVarPickers
MixinForObjectPickers
MixinForProjectDataPickers
MixinForPropertyHandlerPickers
MultiLinePicker
ObjectPicker
ObjectPilePicker
Picker
PickerCollection
PickerMenu
PickerMenuItem
PickerMenuItemSpacer
ProjectDataPicker
ProjectDataSheetPicker
```

Description

The Searcher has two Find menus to let you specify the kind of search you wish to perform. A search is specified by the two menus at the top of the window and by the input field. The first menu, located at the top left, allows you to specify the type of item you are looking for (e.g. objects or handlers). The second menu, located at the top right, allows you to specify the method to use to search for the items. The Input Field allows you to specify the object or string used while searching. Pressing Return in this field or clicking on the Find button will perform the search. The results are displayed in the Result Panel.

This window allows you to do several different kinds of searches. For all the types of items (i.e. object, properties, handlers, functions, constants and globals), you can search based on a substring of the item’s name. Note in the example above, the string “picker” is in each of the object names listed in the result panel. For handlers and functions, you can search for scripts which contain some string of text. For globals, constants and properties you can search for items with a particular value or whose value references a particular object. These two types of searches can be useful when you wish to ensure that an object will be garbage collected. Searching for a value will see if any item of the specified type(s) has the object as it’s value. Searching for a reference will see if any item of the specified type(s) has the object somewhere in it’s value. This means that the search will look “deeply” into values searching through collections to see if the specified object is part of the value.

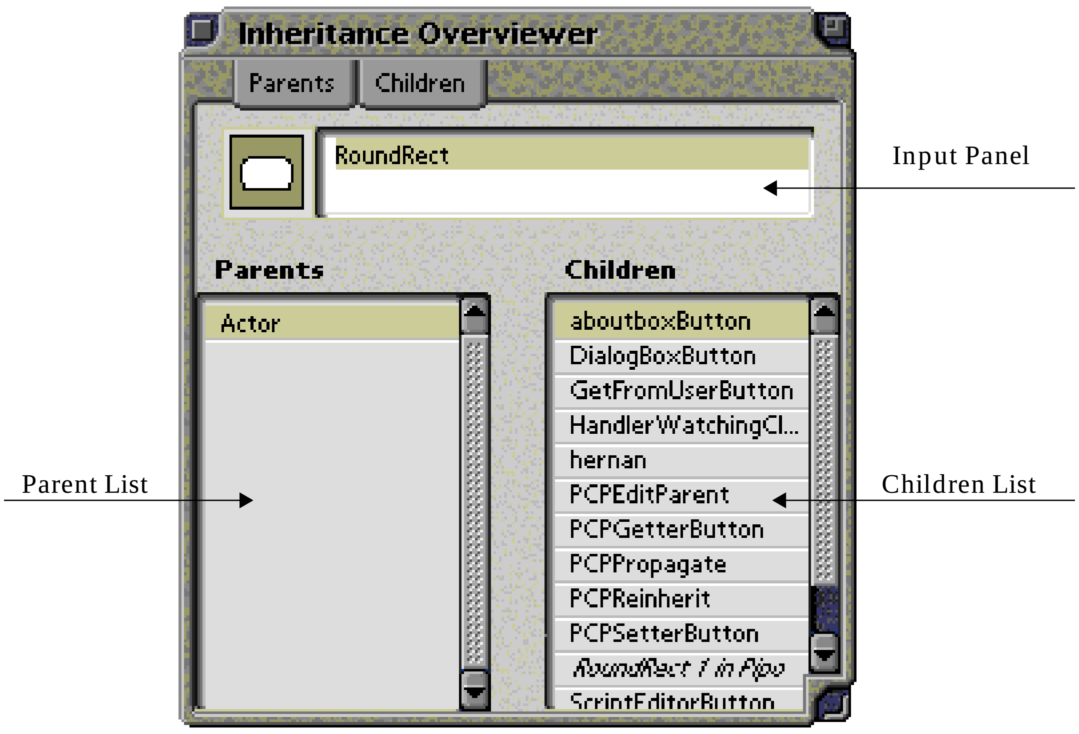
Items can also be dragged from the Results Panel to other Project Builder windows, such as the Documenation Window, for more information. In addition, you can double click any of the items to bring up an editor for the item. Thus if you wish to clear a reference to an object from some property, you can double click on the property and edit it’s value to remove the reference.

Finally, there is a button labeled “Intersection” displays the intersection of the results of this search with the results of the previous search. This is useful for cases such as if you know an object's name contains both the words “Button” and “Push”.

Inheritance Overviewer

The Inheritance Editor provides:

- information to the user about the inheritance hierarchy (including multiple inheritance)
- navigation of the inheritance hierarchy
- a view from which users can abstract a set of objects into a prototype
- an appropriate view on which to indicate inheritance “meta information” such as whether a property is editable or inheritable



Description

The Inheritance Overviewer window is divided into three panels.

The top panel is an input panel for specifying the object (or group of objects) you wish to view or edit.

The left panel, the Parent panel, provides for viewing and editing of the parents of the selected object(s). If multiple objects are selected, shared parents are displayed. New parents can be added via the Parents menu or via dropping a new parent into the Parent panel. Parents can be reordered via drag and drop within the panel. They can be removed by selecting the parent and pressing delete.

Finally, the menu has an option for splicing in a new parent between the selected object(s) and their existing parents. This is useful when you have made several objects which should descend from a common prototype. You can “splice” by selecting the objects, dragging them to the top field and then using the “Splice in a New Parent” option in the Parents menu.

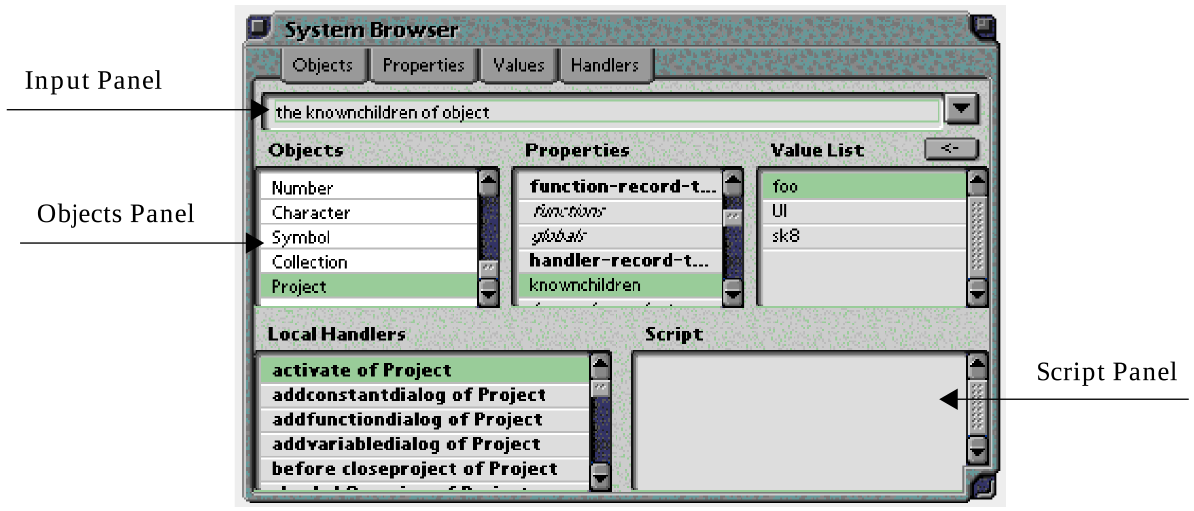
The right panel, the Children panel, displays the children of the selected object(s). If multiple objects are selected, shared children are displayed. Note that children will show all of the “knownChildren” as well as any other children in the currently edited project. You can add and remove children to this “knownChildren” list by selecting the children and using the appropriate menu item in the Children menu. In addition you can create new children of the selected objects as well as clear references to children.

▲ **WARNING**

Any change to parents is a serious or major operation which can seriously alter your object. Use this with caution. ▲

System Browser

The System Browser is useful for browsing sets of objects, properties, values, and handlers, as well as, traversing through the various hierarchies of the system. It can be used to find objects with similar names. This is useful to determine if inherited handlers are overridden by a specific object.



Description

The user can type a query (or drop an object or property reference) into the query field at the top of the panel, and the objects panel will display the results of that query. The objects panel controls what the other panels (properties, value list, and handlers panels) display. The selected item(s) in the objects panel will cause the appropriate properties and handlers to appear in the properties and handlers panels. Note that you can select and edit multiple objects at the same time by shift selecting in the object panel.

The value of the selected property in the property panel appears in the value panel. You can edit the value by either double clicking on the value or selecting the panel and pressing Return or Enter.

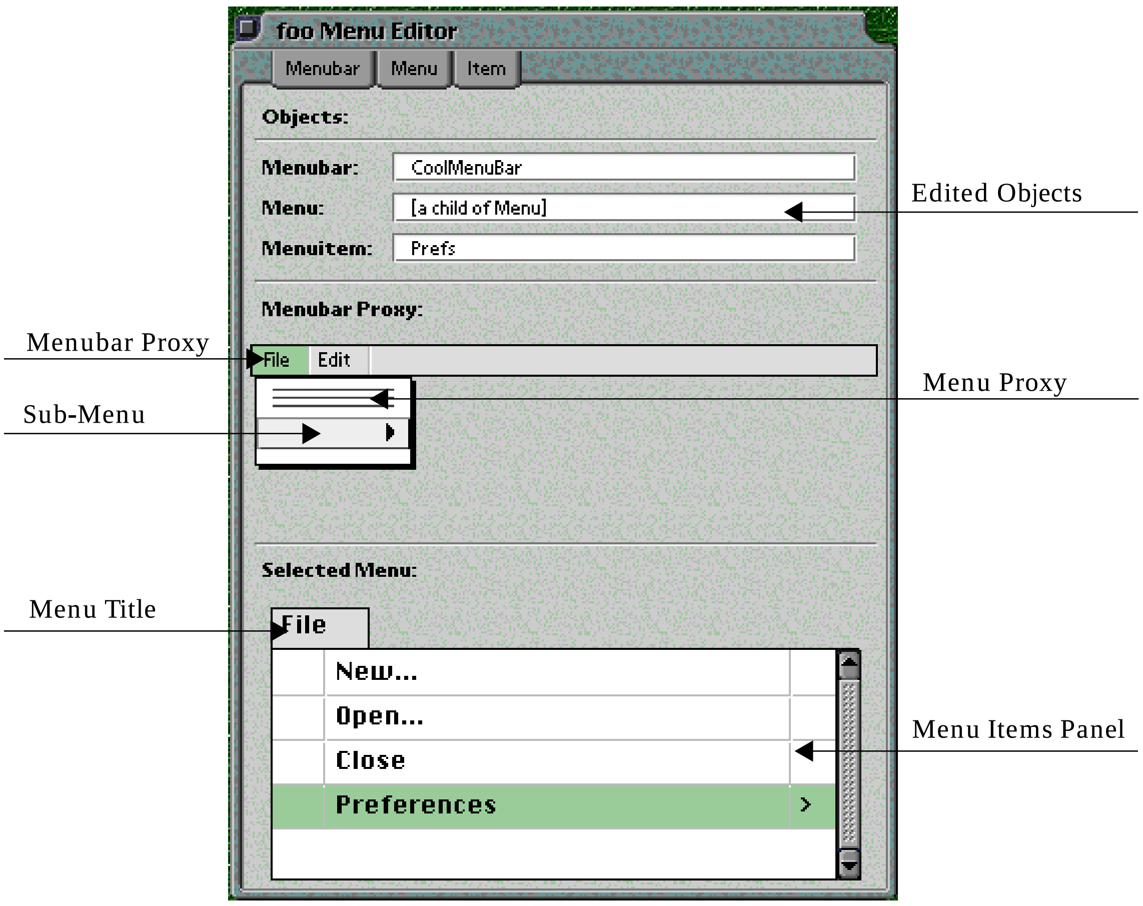
The button above the Value List allows you to move the contents of the Value List to the objects list on the left side. For example, if you are browsing the `knownChildren` of object, pressing this button would place "The knownChildren of object" in the field at the top and the children would appear in the object list.

Double-clicking on a property will bring up a Property Control panel. Double clicking or pressing return in the value list will let you edit that value in the same way as the object editor.

Finally there is a Script Panel which displays any script which is available for the selected handler. Note this is not editable.

Menu Editor

The menu editor allows the building of menubars, menus, menu items, pop-ups, and hierarchical menus through direct manipulation.



Description

The menu editor can be used to create a single menu or an entire menubar. Editing is controlled predominantly by the three menus at the top, which are described in detail below. Navigation is controlled via direct manipulation. Note that drag and drop is particularly useful within this editor. Specific cases are described below

Edited Objects List

These three rectangles display which objects are currently being focused upon. You can mousedown on these rectangles and drag away references to these objects. In addition, you can drop objects of the appropriate type on these rectangles to have the editor focus on them.

Menubar Proxy

When editing a menubar, a proxy displaying the menubar is shown in the center of the editor. Clicking on one of these menus displays a proxy of the menu underneath it and focuses the bottom part of the editor on the selected menu. You can drag menuitems from the bottom part of the editor and drop them on a different menu in this proxy to move them to a new menu. You can drag around the menus within the proxy to reorder them. Note that when the keyboard is focused on this proxy, you can type Command-N to add a new menu and delete to remove one.

Menu Proxy

When one of the menubar's menus is selected, this proxy appears underneath it. This proxy serves two purposes. First it shows the approximate size of the menu with one line for each item. Second it displays any sub menus in the menu (see the above diagram). By clicking on a submenu, the lower section of the editor will be focused on the submenu.

Menu Title

When a menu is being edited, it's text appears in this field. Typing in this field allows the user to directly change the text.

Menu Items Panel

When a menu is being edited, it's menu items appear in this picker. By doubleclicking on an item or by selecting an item and pressing return, you can edit the text of the menu item. Press return when finished to set the text. You can drag and drop items to reorder them, as well as drop a menu into this list to serve as a subMenu. Note that when the keyboard is focused on this panel, you can type Command-N to add a new menuitem and delete to remove one.

Menubar Menu

Edit New Menubar	This brings up a dialog to create a new menubar to edit.
Edit Existing Menubar	This brings up a dialog which hierarchically displays all of the menubars visible to your project. Selecting one focuses the menu editor on that object.
Put Menubar in Stage	This makes the currently edited menubar the menubar of the stage. In other words, it installs this menubar on top of the screen.
Clear References	Clears all references to the edited menubar. See the section on Clearing References at the end of this chapter.
Edit Update	Brings up a Script Editor to write the Update handler for the selected menubar. Note that by default, a menubar gets an update event whenever the user mouses down on one of it's menus.

Edit Menuselect	Brings up a Script Editor to write the Menuselect handler for the selected menubar. Note that by default, a menubar gets a menuselect event whenever the user selects and item from one of it's menus.

Menu Menu

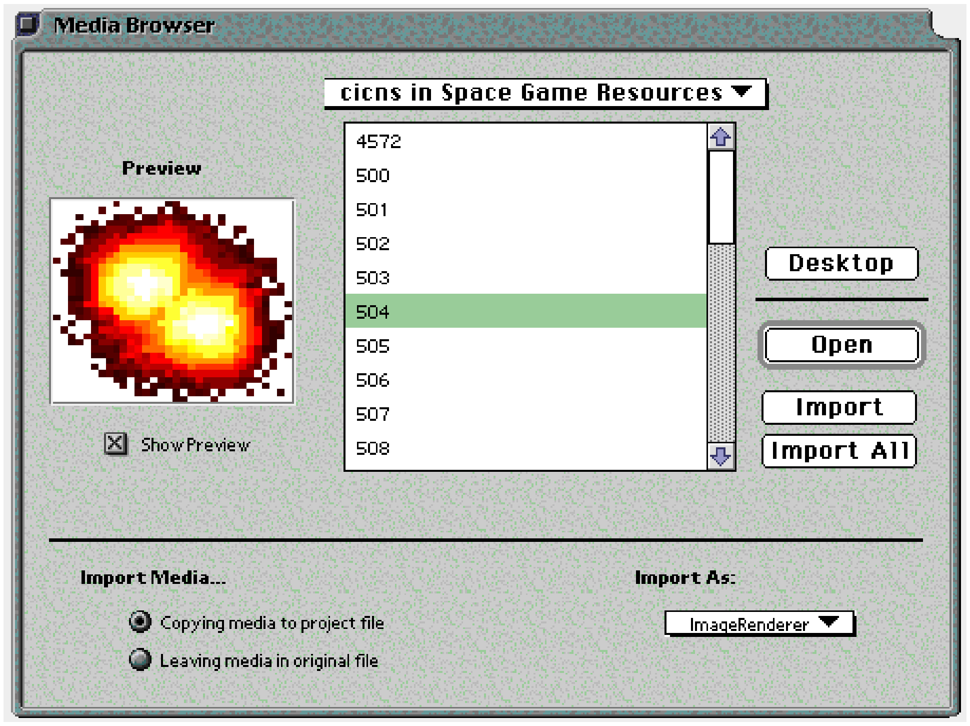
Edit New Menu	This brings up a dialog to create a new menu to edit.
Edit Existing Menu	This brings up a dialog which hierarchically displays all of the menus visible to your project. Selecting one focuses the menu editor on that object.
Add Menu to Menubar	This adds a new unnamed menu to the edited menubar.
Add subMenu to Menu	This adds a new unnamed submenu to the edited menu. This is used to build hierarchical menus.
Clear References	Clears all references to the selected menu. See the section on Clearing References at the end of this chapter.
Edit Update	Brings up a Script Editor to write the Update handler for the selected menu. Note that by default, a menu gets an update event whenever the user mouses down on it.
Edit Menuselect	Brings up a Script Editor to write the Menuselect handler for the selected menu. Note that by default, a menu gets a menuselect event whenever the user selects one of it's items.

Items Menu

Add Menuitem to Menu	This adds a new unnamed menuitem to the edited menu.
Clear References	Clears all references to the selected menuitem. See the section on Clearing References at the end of this chapter.
Edit Update	Brings up a Script Editor to write the Update handler for the selected menuitem. Note that by default, a menuitem gets an update event whenever the user mouses down it's menu.
Edit Menuselect	Brings up a Script Editor to write the Menuselect handler for the selected menuitem. Note that by default, a menuitem gets a menuselect event whenever the user selects it.

Media Browser

The Media Browser is used to select media from files, to view them, and if desired to import them into your project. The media will be translated into image renderer objects, sounds, cursors, etc.



Description

The Media Browser provides an interface for importing media into your SK8 application. It consists of two main sections. The upper section specifies what is to be imported. The lower section specifies how the media is imported.

Finding the Media

The upper section consists of a psuedo-file dialog that allows you to browse through the files available on disk. When you find a file which contains the media you wish to import, you can double click on the file to see the list of media types that file contains. You can then double click on a media type to see the list of media of that type. Note that on the right hand side is a preview box which is activated by the checkbox underneath it.

When you have found the media you wish to import, you can shift select the choice you wish to import and press the import button, or you can choose to import all the media of the selected type by pressing the import all button.

Choosing the Style of Importation

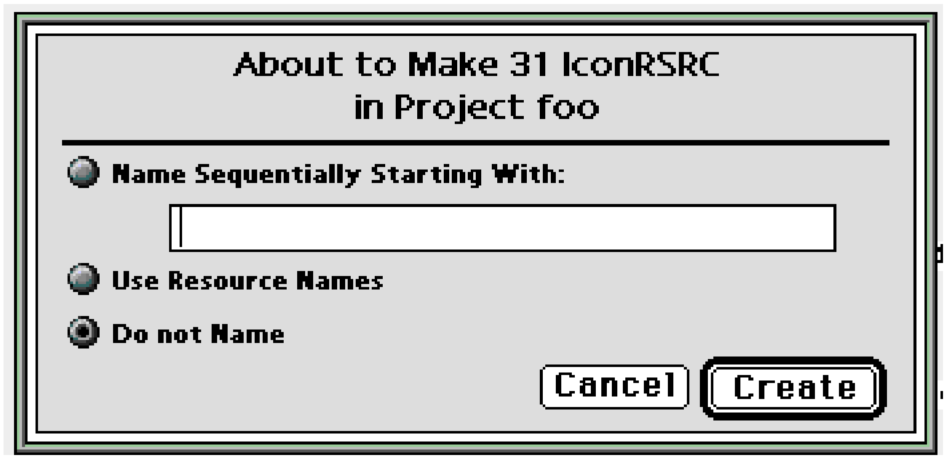
Media will be imported based on the selections made in the lower section. The first two radio buttons control where the media is to be located:

Copying media to project	Copies the media directly into the project
Leaving media in original	Copies the media's filename only into the project. The media is not copied into SK8, but imported by reference to the filename.

The Import As option (lower right) provides the ability to specifythe object representation of the imported media. For example, with a PICT you might wish to have an imageRenderer which can be used to color objects or you might wish to just have a low level media object which points to the media. The popup menu gives you such choices.

The Import Naming Dialog

Clicking on the import or the import all buttons brings up the following dialog:



About to Make 31 IconRSRC in Project foo

Name Sequentially Starting With:

Use Resource Names

[✓ Do not Name]

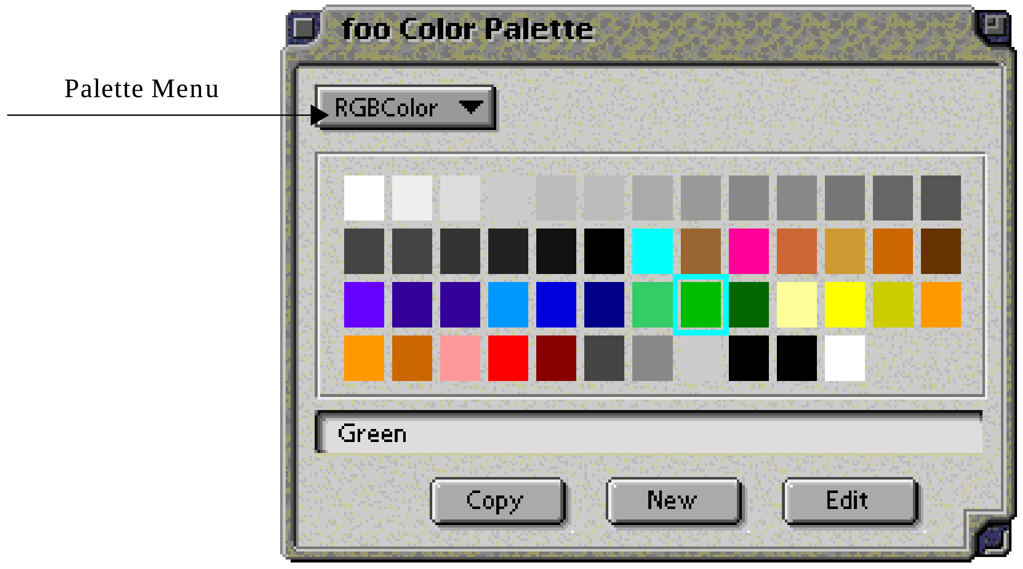
This dialog allows you to specify the objectnames of the objects created for the media. It has three radio buttons:

Name Sequentially	Set the objectnames of the objects to be the name typed in the field followed by a number. Thus if the user imported three PICTs and typed “mypict”, it would create “mypict1”, “mypict2”, and “mypict3”.
Use Resource Names	This names the objects according to their resource names. Unnamed objects are left anonymous. Spaces and other illegal objectname characters are removed.
Do not name	This leaves the objects anonymous.

Once the objects are created, they will show up in the object pile of the project overviewer . This provides a way for the user to hold on to and examine the set of media, even if it is anonymous.

Color Palette

The Color Palette provides an easy way to use the color facilities of SK8. It allows the user to browse the renderers available in the system and use them to color objects.



Description

The Palette menu, top left, displays a menu of many palettes. In the example, above, the RBGColor panel is displayed.

You can drag colors from the Color Palette and drop them on objects. By default the object's `fillColor` is set to the new color.

The `frameColor` of an object is set by holding down the Option key as the color is dropped on the object.

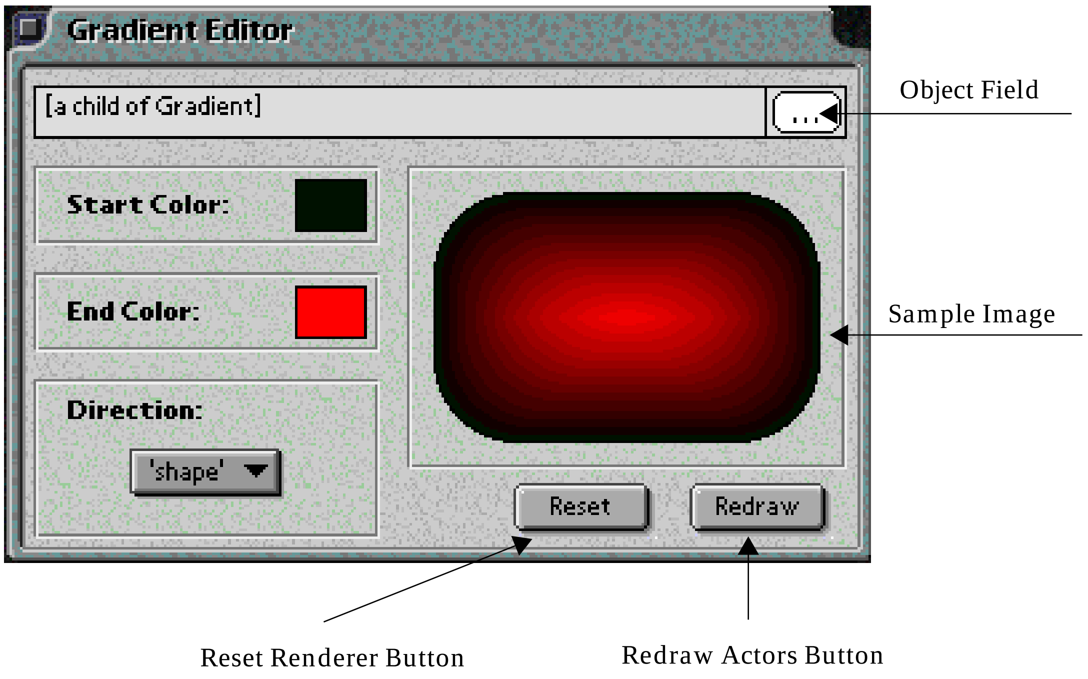
The `textColor` is set by holding down the Command key as the color is dropped.

Copy button	Duplicates the currently selected renderer in your project.
New button	Creates a new renderer of the currently viewed type.
Edit button	Displays the appropriate renderer editor for the selected renderer. (see below)

Renderer Editors

The Project Builder provides direct manipulation tools for building and manipulating renderers in your project. There are tools for building the following renderers:

- RGBColor
- ComplexRGBColor
- Gradient
- ComplexGradient
- BevelRenderer
- MultiRenderer
- Hatch
- ImageRenderer



Description

Most of the Renderer Editors described below follow a design similar to the sample shown above. At the top of the window is a field for specifying a renderer to edit. You may specify this by typing a renderer's name in this field, by dropping a renderer in this field, or by clicking on the "..." button and choosing from a dialog of choices.

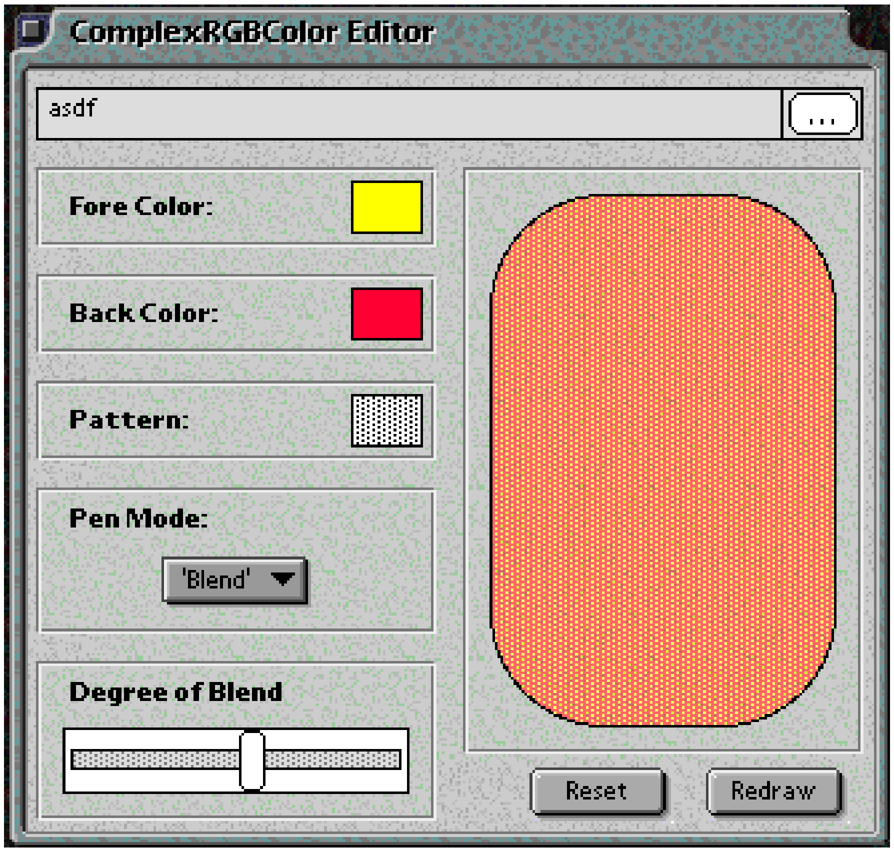
The left side of the window provides you with controls for the various important properties for the renderer. These controls come in four flavors. The first kind is for choosing a renderer or an RGBColor (see the Start Color and End Color controls in the above example). Clicking on these controls brings up a dialog for selecting an appropriate color for theproperties. In addition you can drag and drop a color of the appropriate type onto these controls to set the property to that value. The second kind of control is a popup menu of options. The third is a slider which is used in specifying translucency. The fourth kind is a number entry field for entering integer values. These are used in specifying offsets andline sizes. Note that you can undo any changes you make using these controls in the standard fashion.

The right side of the Renderer Editor provides a sample imageof the renderer so you can view the changes. There are two buttons at the bottom of the window. The first button will revert the renderer back to the way it was when you began editing. This is useful when you've made a whole bunch of changes and are unsatisfied with the result. The second button is the "Redraw" button which will force every actor on the stage that uses the edited renderer to redraw itself to reflect the changes you have made. Thus you can click this button whenever you want to see the changes in the context of your project.

RGBColor Editor

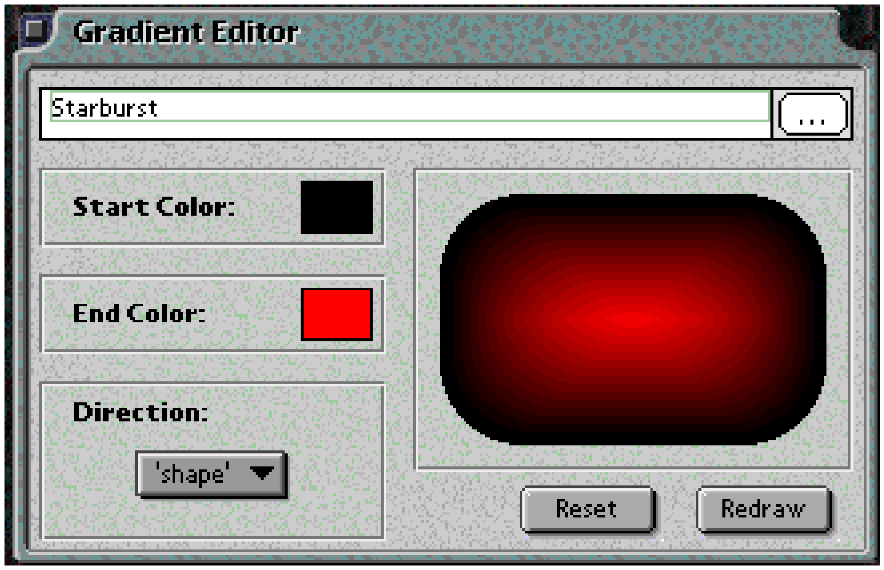
RGBColors are specified by their forered, foregreen and foreblue properties. This editor is unique in that it brings up the standard system dialog for choosing an RGB color to set these values.

ComplexRGBColor Editor



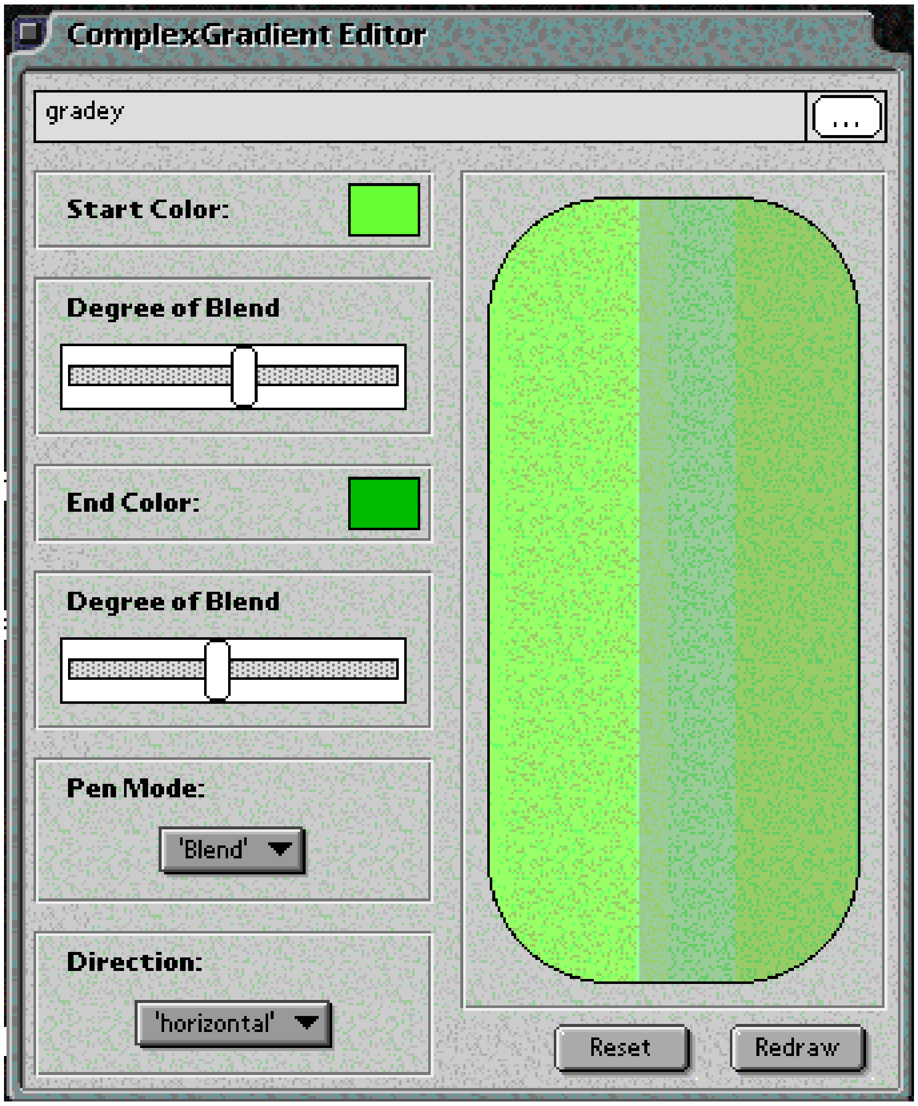
This editor allows you to editor to generate ComplexRGBColors. ComplexRGBColors allow you to control two main parameters. The first is the pattern of the color drawn. You can choose a pen pattern, a foreground color and a background color, and the renderer will draw the pattern using these colors. The second parameter is the Pen Mode. Pen Modes affect how the pattern is drawn on the screen. The default is ‘SrcCopy’ which just draws normally. Others, such as ‘Blend’, allow you to create translucent colors. The slider at the bottom is active for some pen modes to specify an additional parameter. For example, in the case of ‘Blend’ it specifies the percentage of translucency.

Gradient Editor



This editor allows you to create Gradient renderers. This type of renderer draws a gradient of color from a start color to an end color in a specified direction. You can choose an RGBColor to use as a start color and as an end color by clicking on those two controls or by dropping a color onto the controls. Note that the quality of gradients depends heavily on the number of colors your system can display.

ComplexGradient Editor



gradey

Start Color: 🟩

Degree of Blend

End Color: 🟩

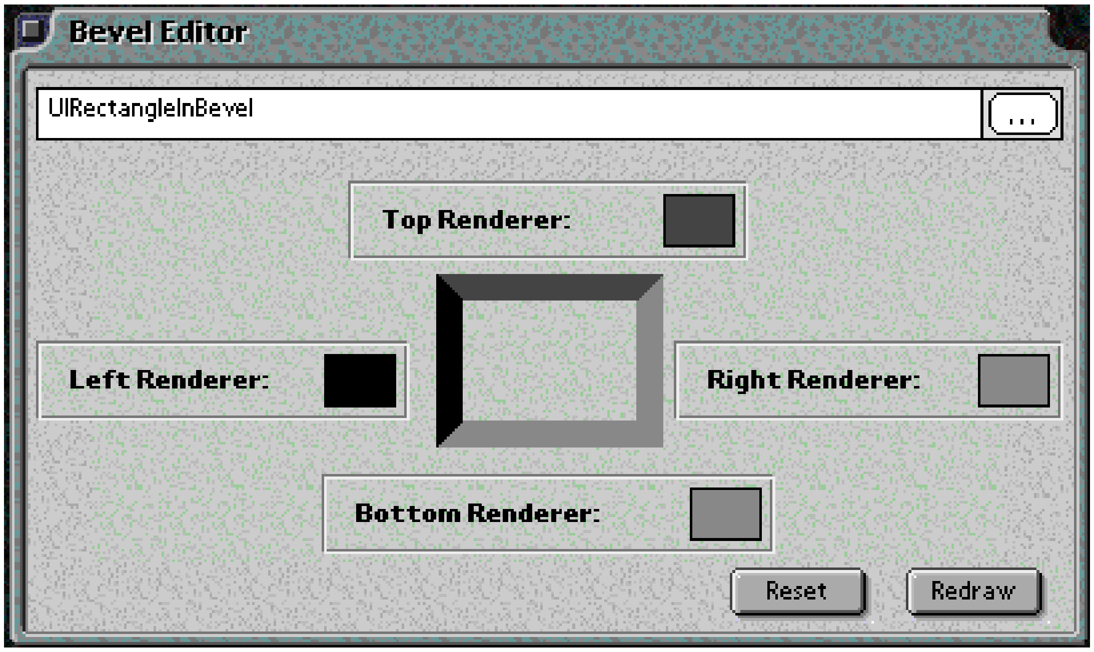
Degree of Blend

Pen Mode: 'Blend'

Direction: 'horizontal'

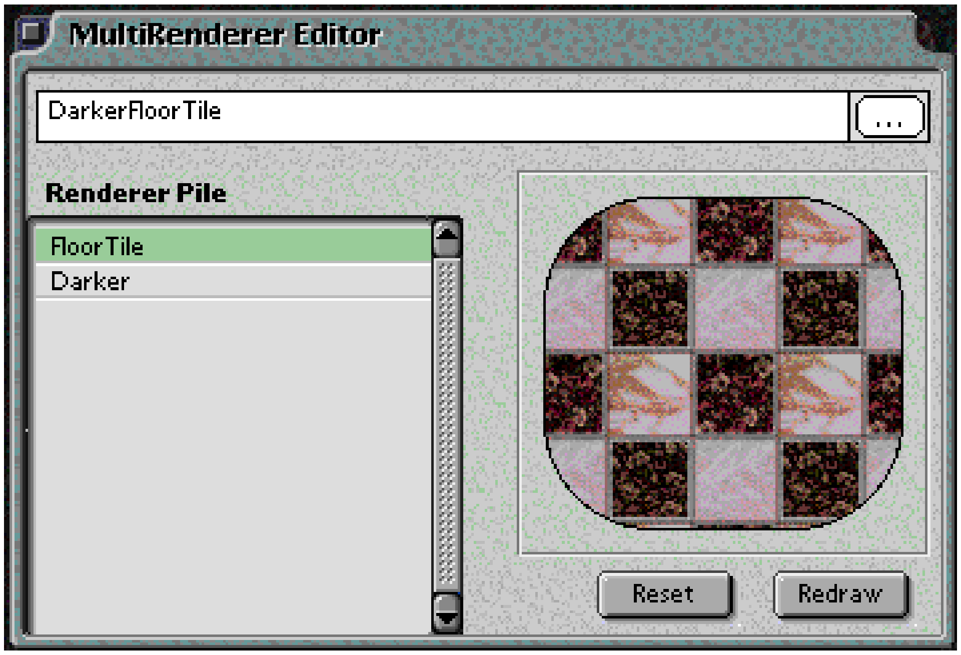
ComplexGradients are similare to normal Gradients except that you can specify the Pen Mode used to draw the gradient. Pen Modes can be used to createa variety of effects, the most common being translucency. In the above example, note that the 'Blend' pen mode has been selected. The two sliders control the percentage of translucency of the start and end points. This amount of translucency is interpolated between the start and the end in the same way the color is.

BevelRenderer Editor



The BevelRenderer takes four renderers and draws them in a rectangular shape where each renderer corresponds to a side. It is designed to be used in the frame of an actor to give it beveled three dimensional look. This editor simply allows you to experiment setting the values for each of the sides. You make click on one of the four controls to choose a renderer, or you may drop any renderer onto these controls. Note the layout for this window is slightly different than the other editors in that the sample image is in the middle.

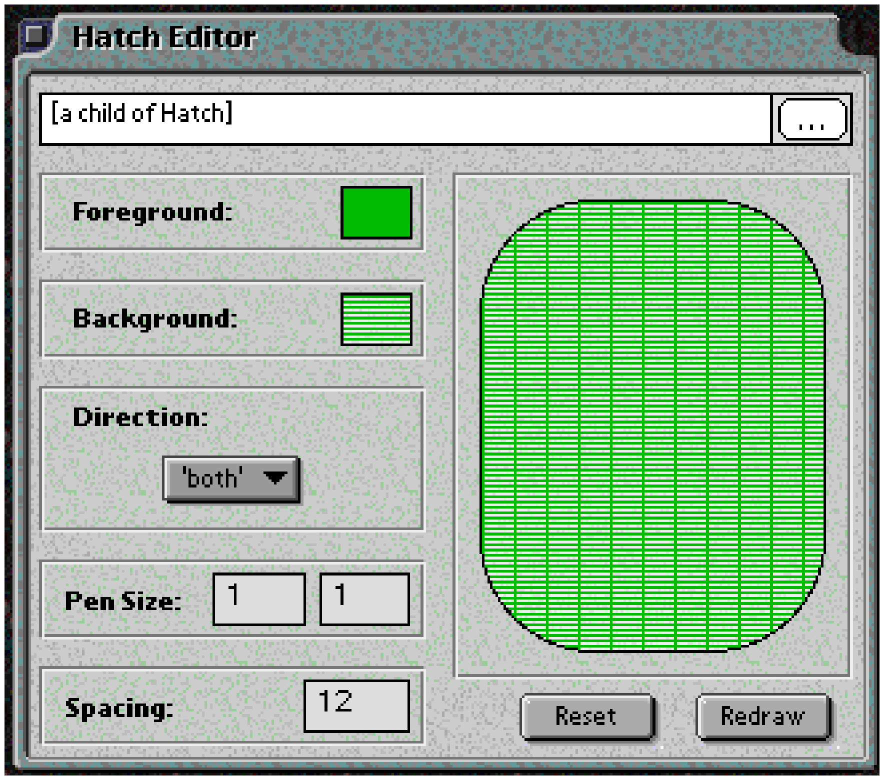
MultiRenderer Editor



AMultiRenderer is simply a renderer that sequentially draws a series of renderers into the same space. In the above example, first the floor tile is rendered into the rounded rectangle and then it is made darker by the darker renderer. The editor allows you to control this series of renderers in a pile like fashion. You can drag and drop renderers into the Renderer Pile to add them. You can drag and drop renderers within the pile to reorder them. You can select a renderer and press the Delete key to remove a renderer from the pile.

Note that adding and removing renderers to this pile will not affect the existing renderers themselves.

Hatch Editor



[a child of Hatch]

Foreground: 🟩

Background: ▤

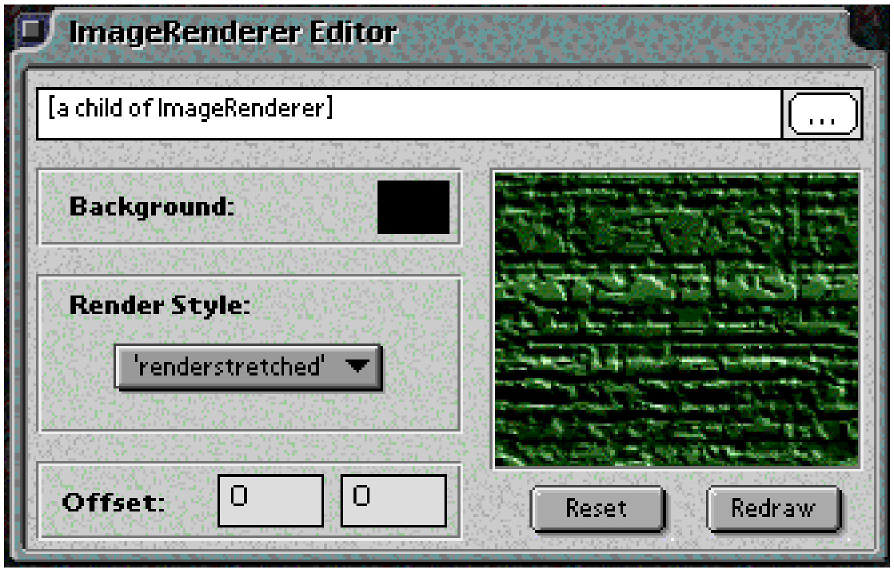
Direction: 'both'

Pen Size: 1 1

Spacing: 12

Hatches are used to draw a grid of lines on top of another renderer. The background color specifies the renderer which is first drawn before lines are drawn on top it. The foreground color specifies the color of the lines. The pen size, which must consist of two integers, specifies the size of the lines. The spacing, which also must be an integer, is the distance between the lines. The direction specifies whether to draw only horizontal, only vertical or both horizontal and vertical lines.

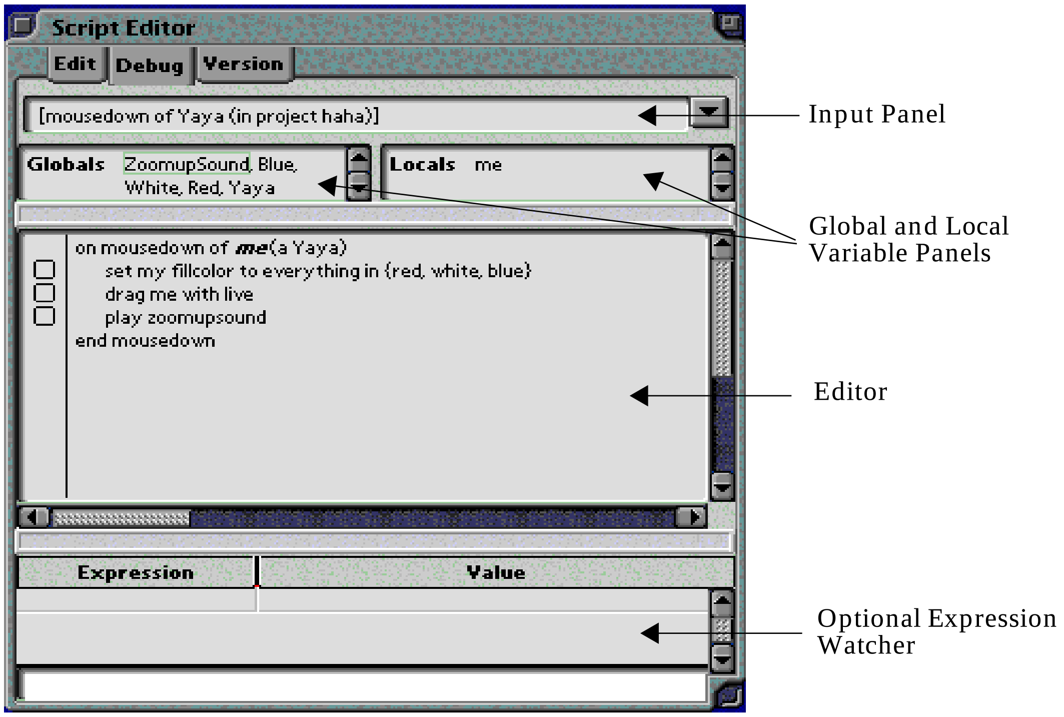
ImageRenderer Editor



An ImageRenderer can be used to display a large number of media types, e.g. PICTs, PPATS, etc. Generally these objects are first created via importation using the Media Browser. This editor lets you control some of the finer parameters of these renderers after their media has been set. You can specify the style you want the media to be drawn. Options are to draw regularly (i.e. draw the media once unstretched), draw stretched to fit the space alloted, or draw unstretched in a tiled grid. You can specify a background renderer to draw behind the media. You can also specify an inital offset from the upper left hand corner to draw the media. Note that the background renderer and the offset only are used when the media is drawn unstretched.

Script Editor

Whenever you choose to add a handler to an object, a Script Editor is presented for writing and compiling the new handler. The Script Editor is also used to edit the scripts of existing handlers. The Script Editor is also the place where functions can be edited. The Script Editor can also be invoked by typing Command-N in a handlers or functions panel (e.g. the handlers section of the Object Editor), or by double clicking on a handler or function in a one of these panels.


Script Editor
Edit
Debug
Version
[mousedown of Yaya (in project haha)]
Globals ZoomupSound, Blue, White, Red, Yaya
Locals me
Expression
Value
Input Panel
Global and Local Variable Panels
Editor
Optional Expression Watcher

```
on mousedown of me(a Yaya)
    set my fillcolor to everything in {red, white, blue}
    drag me with live
    play zoomupsound
end mousedown
```

Description

The Script Editor is a context sensitive SK8Script language editor. It automatically adds some initial and closing syntax and indicates syntax errors while they are being typed.

Edit Menu:

New Property...	Adds a property to the object of the handler being edited.
New Handler...	Adds another handler to the object of the handler being edited.
Remove this Handler	Deletes this handler from the object.
Print	Prints the text of the handler.
Stack Editors	Stacks up all Script Editors currently on the stage.

Debug Menu

The Script Editor Debug Menu controls several aspects of SK8's debugging environment. The first item in Debug menu, "Watching On", is the master switch for the next three items in the Debug menu. The last three items in the debug menu cause additional debugging information to be revealed in windows or sub-panes. See the next section , Debugging with the Script Editor, for more details.

Log Calls	If Log Calls and Watching On are both checked, a message will be written in the Message Box each time the handler is called and each time it returns.
Pop Up When Running	Pop Up When Running and Watching On are both checked, the handler's Script Editor window will be displayed as the topmost SK8 window, every time the handler is called
Trace Execution	If Trace Execution and Watching On are both checked, each line of the handler will be highlighted as it is executed..
Expression Watcher	Displays or Hides the Expression Watcher component of the Script Editor. See the next section for more details.
StackWatcher	Brings up the Stack Watcher. See the section below on the Stack Watcher.
Handler Tracer	Brings up the Handler Tracer. See the section below on the Handler Tracer.

Version Menu:

Activate Current Version	This takes the existing SK8Script in the editor, compiles it, and activates it for the object.
Delete Current Version	Deletes the current version from the version history.
Delete Earlier Versions	Deletes all versions of the current handler from the version history which are previous to the selected one.
<Date, Time>	These menu items represent the different versions of the current handler which are in the version history. Choosing one of these items will display the script of this version. Note that this version will only become active if "Activate Current Version" is chosen. Note also that the menu places a check next to the current version.

Debugging with the Script Editor

SK8 provides an advanced source code level debugging environment to help you isolate and fix errors in your handler and function scripts. The debugging facilities are available via SK8 Script Editor windows. This section describes how to use the SK8 debugging facilities.

The basic steps for debugging in SK8 are:

• Set a breakpoint in your handler

• Step through the execution of your handler and handlers it calls

• Examine values referenced in your handler

• Display and examine the call chain (the stack). There are several ways to do this.

Breakpoints

To the left of each script statements is a column with a breakpoint button. Clicking on a breakpoint button toggles the breakpoint on and off. When a breakpoint is on, SK8 will halt execution of handler immediately before executing the next statement. An arrow icon will appear next to the breakpoint button, indicating which statement will be executed next.

If you double click on a breakpoint box, the condition for a conditional breakpoint can be entered as a SK8Script expression. Conditional breakpoints are indicated by a dot in the center of the breakpoint box.

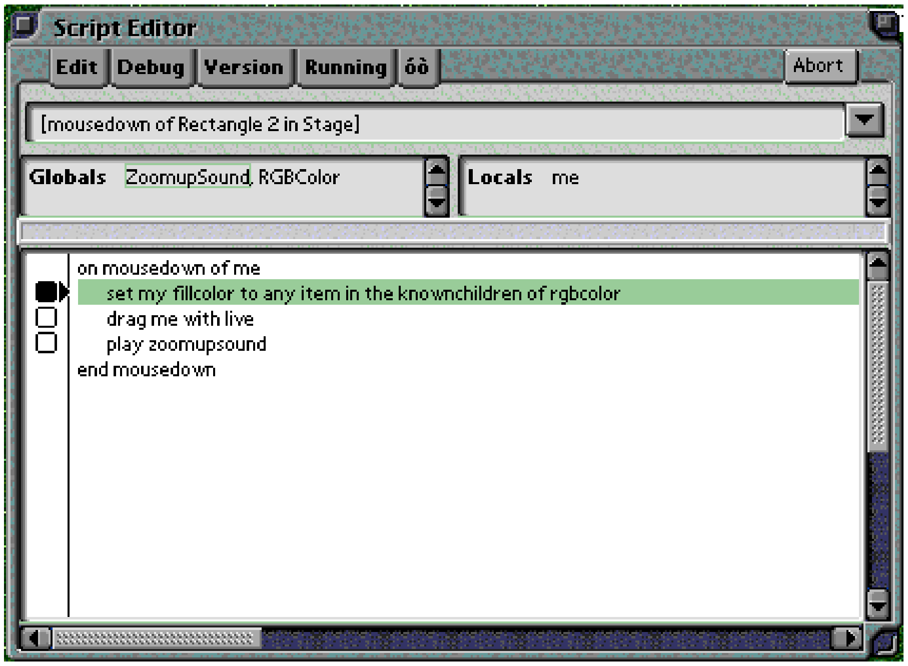
Halt Due to Error

Similarly, if SK8 encounters an error while running your handler, it will halt execution of your handler, and put a yellow triangular warning marker over the breakpoint mark of the instruction that caused the error. Note there are a couple of important differences between halting at a user-defined breakpoint and halting due to an error:

■ An error indicator appears next to the statement that failed to execute. On the other hand, when a handler is halted at a breakpoint, the arrow indicator is next to the statement that is about to execute.

■ You cannot continue execution of a handler that has been halted at an error. The Go and Step commands will be disabled. The abort command is still available, and you can still examine the state of the handler using the Expression Watcher.

When handler execution has been halted, two new menus appear at the top of the Script Editor.


Script Editor
Edit
Debug
Version
Running
óò
Abort
[mousedown of Rectangle 2 in Stage]
Globals ZoomupSound, RGBColor
Locals me

```
on mousedown of me
   set my fillcolor to any item in the knownchildren of rgbcolor
   drag me with live
   play zoomupsound
end mousedown
```

Running Menu

The “Running” menu contains commands specific to controlling the execution of the handler under debug mode.

óò Menu

The “óò” menu shows the chain of handlers that were called to get to the handler now under examination. To examine one of these handlers, select it from this menu.

Also when handler execution is halted, three new buttons appear at the top of the Script Editor. These buttons provide a convenient way of executing three debugger commands. These three commands can also be executed by selecting them from the Running menu, or typing a command key combination.

Go, command-G

To continue normal execution of the handler:

1) Click on the “Go” button at the upper right of the Script Editor window.

2) Select “Go” from the Script Editor’s Debug menu.

3) Type command-g when the Script Editor is the active SK8 window.

Step, command-S

Once a program has halted for a user-defined breakpoint, you can step—statement by statement— through the rest of the handler. There are three ways to execute the current statement and step to the next statement. You can:

1) Click on the “Step” button at the upper right of the Script Editor window.

2) Select “Step” from the Script Editor’s Debug menu.

3) Type command-s when the Script Editor is the active SK8 window.

Abort, command-period

To terminate execution of the current handler and all handlers in the calling chain:

1) Click on the “Abort” button at the upper right of the Script Editor window.

2) Select “Abort Event” from the Script Editor’s Debug menu.

3) Type command-period when the Script Editor is the active SK8 window.

The other menu items in the Running Menu are:

Step Into

This command enters the first called handler (or function) it finds in the current statement. It will display a Script Editor for the called handler and execution will be halted before its first line.

“Step into” will only work for handler and functions for which you have the SK8 source code; in other words you cannot step into functions from the SK8 Project or UI (User Interface) Project. If there is no such handler in the current statement, “step into” works just like the normal “Step” command.

Go To End

Execute the handler until the end statement. When SK8 has halted on a handler’s “end” statement, it has lost the handler’s context so you cannot use the expression watcher for any meaningful evaluation.

Restart

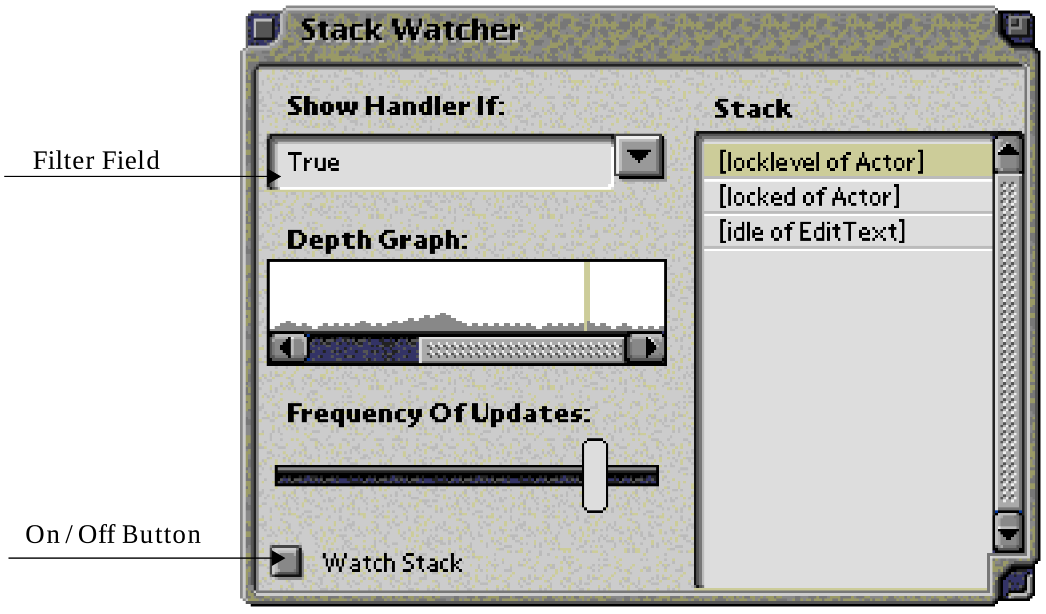
This debugger command is not implemented in the current version of SK8. When it works, handler execution will proceed from the first line of the handler.

Expression Watcher

Selecting Expression Watcher from a Script Editors Debug menu reveals an evaluation area at the bottom of the Script Editor window. The Expression Watcher consists of two column lists, and a type-in field. In the type-in field you can type, or copy, the name of a handler variable, or a more complex expression, followed by the enter key; the text you typed will appear in the left column above and its current value will appear in the adjacent row on in the right column. Note the handler's local variable will only be defined during the execution of the handler.

Stack Watcher

The Stack Watcher, as the name implies, provides the ability to view and navigate through the stack. The Stack Watcher remembers every handler call and captures all the information. This is an excellent tool for observing the layering of handler calls, starting at the beginning call and filtering down the stack to a specific location in code.



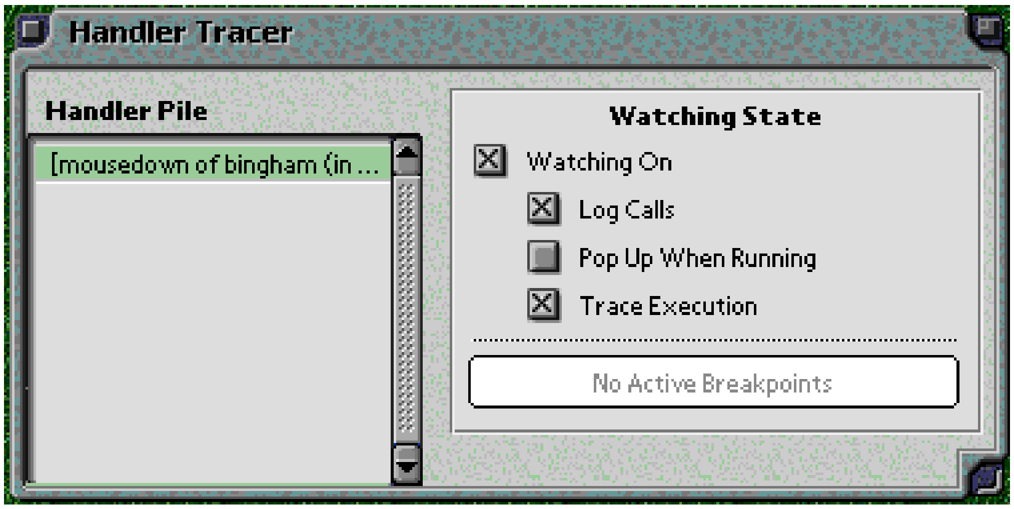
Description

On / Off Button	The Stack Watcher is activated / deactivated by this button.
Filter Field	Allows you to write a filter for the Stack Watcher. The filter is a SK8Script selection expression that you apply to all the handlers in the Stack Watcher. For example, maybe you only want to look at the handlers in a particular project or when "`myglobal is red`", etc. The current handler in the Stack is referred to as "it". The output is displayed in the Stack panel on the right. The handler called last is at the top of the Stack panel.
Depth Graph	Allows viewing of a previous part of the stack. This is useful for showing the handler call sequence for each new handler. Click on a part of the graph to see the stack at that point. Use the left and right arrows to navigate backwards and forwards in time to examine the order of handler execution.

Frequency of Updates	Controls the update frequency of the stack. The fastest setting will cause all handler calls to be displayed in the Watcher exactly as they happen. This will result in some degradation of SK8's performance. Slower settings will cause the Watcher to update only periodically, allowing the system to perform better.

Handler Tracer

The Handler Tracer window lets you control "watching" the execution of multiple handlers.



[mousedown of bingham (in ...

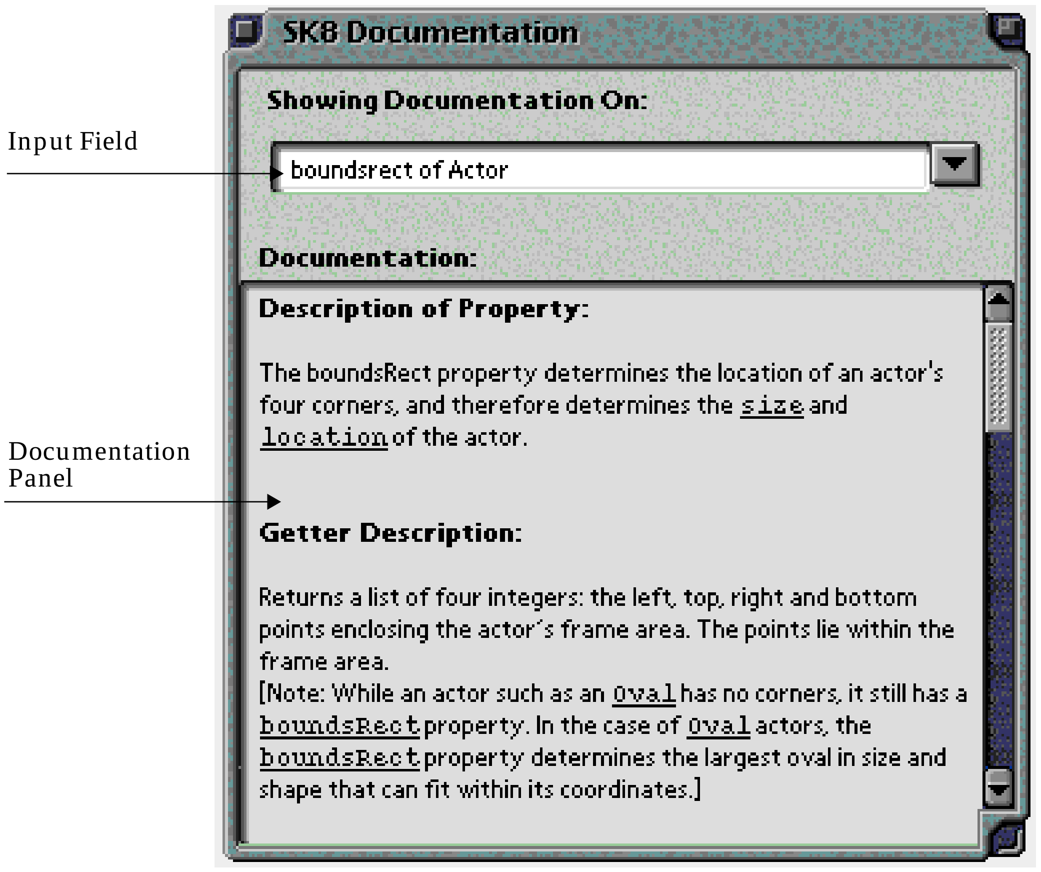
[✓] Watching On
[✓] Log Calls
[ ] Pop Up When Running
[✓] Trace Execution

Description

Handler Pile	Allows you to specify the handlers you wish to watch or not watch. You can drag handlers into the "Handler Pile" from Object Editor's and other SK8 windows.
Watching State	For all the handlers in its "Handler Pile", Handler Tracer simultaneously provides the functionality described above for theScript Editor's menu items "Log Calls", "Pop Up When Running", and "Trace Execution". See the Script Editor description for more detail.

Documentation Window

The documentation window allows you to browse the SK8 object reference as a hypertextual document online.



Description of Property:

The boundsRect property determines the location of an actor's four corners, and therefore determines the size and location of the actor.

Getter Description:

Returns a list of four integers: the left, top, right and bottom points enclosing the actor's frame area. The points lie within the frame area.
[Note: While an actor such as an Oval has no corners, it still has a boundsRect property. In the case of Oval actors, the boundsRect property determines the largest oval in size and shape that can fit within its coordinates.]

Description

Input Field	Here you can drop an object, function, property, etc. to get documentation on it. Note that if there is no documentation on the particular object it will search up the inheritance hierarchy to find the next highest piece of relevant documentation. Note you can search for an object by typing it's name. You can search for a property or handler by typing the form "<property / handler> of <object>". You can search for a function, constant or variable by typing "the function beep" or "the constant pi", etc..

Documentation Panel	This panel shows documentation on the specified item. The documentation may contain words or phrases which are underlined. These are hotlinks which may be clicked upon to find more information. Note you can use the history menu of the input field to go back.

Note

The documentation is unmodifiable. You can neither change existing documentation nor add additional documentation. You can however select and copy the examples to try them out. ◆

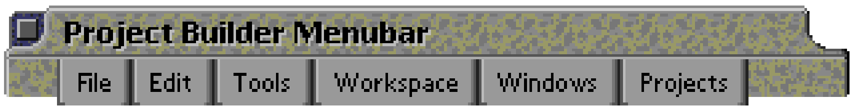
Project Builder Menubar

The Project Builder Menubar allows you to access the various tools and functions of the Project Builder. It is designed to allow you to both edit and use your project at the same time. Thus it is replacable (see the first item below).

Most of the menu items are self explanatory. Below a few of them are described in more detail.

Menubar Replacement

If you replace the project builder menubar at the top of your screen with your own, the Project Builder menubar will slide out onto the stage in a window which looks like below.



Note that in this case, if you have a Project Builder window active, command keys will go to the Project Builder menubar. If the windows of the user's project are active, command keys will act normally, (i.e. go to the menubar on the stage).

Hide Project Builder

The Hide Project Builder menu is located under the Project Builder Workspace menu. The Hide Project Builder option provides the capability of clearing the Project Builder environment from the Desktop without quitting SK8. When this command is invoked, all Project Builder’s windows are hidden and disabled except for a small Project Builder window or icon. Clicking on the icon brings the Project Builder environment back to the Desktop.

This feature allows you to quickly return to the Project Builder, after doing some other task, while avoiding the time and overhead of bringing up the Project Builder from ground zero.

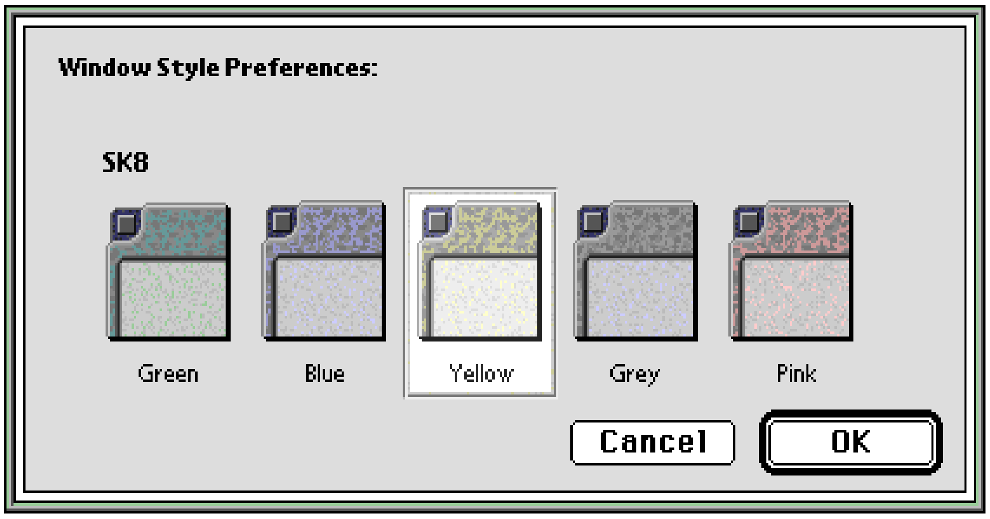
Undo Menu Item

The Undo menu item is located under the Project Builder's Edit Menu. Undo provides users the ability to “undo” typing in fields and the setting of properties.

The Project Builder's Edit menu has the standard “Undo” menu item whose text changes to specify what could be undone and is disabled when not available. There is only one level of undo. Property settings are undone by resetting them to their old value.

Window Preferences Dialog

This dialog lets you select the window color the interface uses. Click on a color to select it and press OK.



Graphic Intensive Tasks: Using Project Builder

During Graphic intensive tasks, such as importing source artwork (as Picts or fill colors), graphics integration, and direct manipulation graphic layout, a user is likely to use the following items together on the stage:

- The Import dialog for importing outside artwork.
- Graphic Source materials.
- Selection handles (instead of Halos) for easy selection and direct manipulation of graphics.
- Drawing tools for making new graphics.
- Color palettes for adding colors and textures to graphics.
- The Project Overview Window for showing items in the media category.
- The Stage Monitor to allow selection of objects in containment layers.

Code Intensive Tasks: Using Project Builder

During code intensive tasks, such as editing the basic object architecture of a project, a user is likely to have the following items out on the stage:

- The Project Builder Application menu—in its detached state, to allow room for a new application menu bar to be built in place.
- Graphical interface elements of the project being built.
- The Color Palette—for changing colors easily.
- Selection Halos (instead of handles)–to enable the composite edit and use style of editing, useful for immediately trying out coded interaction behaviors.
- A Script window—for writing new handlers.
- The Message Box—for trying out SK8Script queries.
- The Project Overview Window—showing all objects included in the project.
- The Object Editor—where properties and handlers of objects are edited.
- The Inheritance Editor—which shows the inheritance context of the currently selected object in the Object Editor.
- The Stage Monitor—for viewing actors and their containment hierarchies.

The user is also likely to use the Import command heavily to import outside work into the project media collection (viewable in the Project Overviewer).

SK8Script

The purpose of this chapter is to cover all of the significant aspects of the SK8 scripting language, SK8Script, in sufficient detail to answer almost all questions that arise while using the language. Each SK8Script construct is explained and detailed working examples are provided. See Chapter 2 for a more gentle tutorial introduction and the SK8 Language Reference for a more concise, technical description of SK8Script.

Initial Considerations

General Language Design

SK8Script is an object oriented language. As such most SK8Script code is associated with some particular named data structure or 'object'. Such code is called a 'functional-handler' or when unambiguous simply a handler. Each handler is owned by exactly one object. New objects are created from existing objects. By default a new object shares or 'inherits' all the handlers owned or inherited by the objects it is created from (its 'parents'). Unlike many other object oriented languages SK8Script does not distinguish between 'prototypes' and 'instances'. Loosely speaking every SK8Script object could be considered to be an instance of its parents and a prototype for its children.

In addition to functional-handlers, SK8Script has 'functions' which are not associated with a particular object. Functions are more familiar to new SK8 users who have never worked in an object oriented language before. However, they are rarely used by experienced SK8Script programmers. Functions and functional-handlers share many features so the term 'executables' is used to refer to both.[*12]

Finally, SK8Script is used to create user defined 'with-forms'. With-forms are used to guarantee that specially designated cleanup code always runs. These forms can either be globally defined or associated with a particular object. In the latter case they are referred to as 'with-handlers'. With-forms are discussed in more detail later in this chapter.

Another important part of the design of SK8Script is ‘garbage collection’. Garbage collection provides an automatic mechanism for destroying objects that are no longer usable. In general this occurs when an object can no longer be accessed in any way by the system. This means that any named object will not be garbage collected. In addition, it means that clean-up of complicated data structures the programmer creates out of unnamed objects happens cleanly and correctly without the programmer doing any explicit destruction of the objects.

Syntax Design

The syntax of SK8Script is designed to be English-like. Many commands in SK8Script read as English sentences and are easily understood by anyone familiar with English.

Example:

```
Get the first rectangle in the stage
Set the width of the result to 150
```

However, most English sentences are not valid SK8Script.

Example:

The follow is not a valid SK8Script command

```
Add the width and height of the first rectangle in the stage
```

and some SK8Script commands are not correct English.

```
set text of rectangle 1 of stage to "hello" & space & "world"
```

Understanding exactly how to craft SK8Script commands is the purpose of the rest of this chapter.

Case in SK8

Similar to English, SK8Script has common capitalization conventions, but these are not enforced in any way and the meaning of a command is in general not effected by the case. The following conventions are recommended for all SK8 code:

- Global variables, global constants and object names have a leading capital.

Examples:

```
Pi
Actor
```

- The names of functions, handlers and properties as well as local variables, local constants, and arguments start with a lower case letter.

Examples:

```
x
factorial
width
```

```
contents
update
```

■ Words within any name are capitalized.

Examples:

```
fillColor
RoundRect
```

■ All other letters are in lowercase.

Example:

```
set the fillColor of MyWellNamedRectangle to LargeBrickWeave
```

Remember that these are just conventions so the above line has exactly the same effect as

```
seT thE filLcoloR oF mYwelLnameDrectanglE tO largEbricKweavE
```

As an example where case does matter the expression:

```
(ascii of "h") equals (ascii of "H")
```

has the value `False`. However, the simpler statement

```
"h" equals "H"
```

is case insensitive and has the value `True`.

For object names, symbols and other identifiers the case pattern of the first occurance is considered to be ‘correct’ and will be used by SK8Script when values are returned. For strings the case pattern depends on the original source of the string.

The following examples show the result of each command or expression as returned when they are entered into the SK8 Message Box. The entered forms are given in bold-face.

```
new rectangle with objectname "Rect1"
Rect1
rect1
Rect1

set x to "hello world"
"hello world"
set y to "Hello World"
"Hello World"
x equals y
True
x
"hello world"
y
"Hello World"
```

Statement Continuation

Unlike English, line breaks are significant to SK8. The character '¬' (Option-Return in the SK8 environment) at the end of a line tells SK8 to ignore that line break. Such lines are normally indented with a tab character. Tabs and space are significant only as token separators.

Example:

```
set the fillColor
of rectangle 1 of stage to blue
```

is syntactically incorrect. Whereas

```
set the fillColor ¬
   of rectangle 1 of stage to blue
```

is correct SK8Script.

Declarations

Most lines of SK8Script code are 'Commands' which usually have some direct effect on some variables or objects. Declarations are lines of SK8Script code that provide extra information either for the reader or the SK8Script compiler.

Comments

Comments provide information for the reader and ignored by the SK8Script compiler. Comments are indicated in SK8Script by two dashes (`--`) and may be placed anywhere in a script. A comment consists of all characters after the comment marker up to the next carriage return.

The comment marker is ignored if it occurs within a string or symbol literal.

Examples:

The following are two examples of valid comments:

```
set x to 1 -- it’s important to do this!

on add a,b
   -- add: this function adds two numbers
   return a + b
end add
```

The following statements do not contain a comment because the comment marker is within a string (double quotes) and a symbol (single quotes), respectively.

```
set x to "-- somthing"
set x to '--hello'
```

Identifiers

Identifiers are the names used in SK8Script. They are used as objectnames, variable names, property names, handler names etc. They consist of any sequence of letters, digits, and the underscore (_) character. The first character must not be a digit. In addition, identifiers cannot conflict with the reserved words in the SK8Script language.

A complete list of SK8Script's reserved words is given in Appendix <???>.[*1]

These are legal identifiers:

```
Garbo
thing2
ho_ho
brewCoffee
```

These are not legal identifiers:

```
4onTheFloor    -- begins with a number
x+y            -- contains illegal special character
Hi there       -- contains illegal special character
end            -- conflicts with reserved word
```

Variables

Variables in SK8Script provide a way to assign a value to an identifier. By default variables are not typed so they many get assigned any value.

Example:

```
set X to 1
1
X
1
set X to "Hello world"
"Hello world"
X
"Hello world"
```

Variables do not have to be explicitly declared in SK8Script. However both for readability and reliability it is recommended that variables be declared in all SK8Script code. Variables may be declared either 'local' or 'global'. A local variable is only available to the commands that are part of the handler or function in which they are declared. Values assigned to local variables to not persist between executions of their containing body. Global variables are available to any code executed within the project they are declared in. Values assigned to global variables persist until explicitly changed. SK8Script assumes that any undeclared variable in a function or handler is a 'local' unless it has been declared global prior to the compilation. The parameters to a handler or function are always considered to be local.

Declaring a variable is done by giving the name of the variable (or a list of variables) on a line preceeded by either the word '`local`' or '`global`'. The declaration of a variable can be anywhere within a block of code as long as it is before the first use and within a context that contains all uses of the variable. However, SK8 has only one local scope within a function or handler, so by convention all variable declarations appear immediately after the header line of a handler or function.

Example:

```
global count = 0 -- evaluated somewhere else like the MessageBox

on example_func a1, a2
   local l1 = a1 + a2 + count
   global g1
   set count to count + 1
   if not g1 then
      set l2 to a1 * a2
      set l1 to l1 + l2
   end if
   set g1 to not g1
   return l1
end example_func
```

In this example the variables `g1` and `count` are globals and `l1`, `l2` (by default), `a1` and `a2` are locals. Note that variables can be initialized when they are declared.

In general global variable should only be initialized outside any function or handler. A global declaration with an initialization suffix will reset the global variable each time execution reaches that line. If no initializer is given then the variable is automatically given the value '`#Undefined#`'.

Constants

Globals may also be declared constants by inserting the word '`constant`' after the word '`global`' in their declaration. Constants may not be changed after they have been initialized.

Example:

```
global constant ShortPi = 3.14
```

After this command has been executed any attempt to change the value of `ShortPi` will result in an error.

Type Declarations

Variable declarations, both local and global, may assign a **type** to a variable. This allows the compiler to check for typing errors and to optimize the code that it generates. Type declarations are optional.

Examples:

```
   local b (an Integer) = 3
```

This declares the variable `B` as local, initializes it to the integer `3` and declares its type to be `Integer` (that is, the value of b must be an object that descends from the object `Integer`).

```
   global C (a Rectangle) = my container
```

This declares the variable `C` as global, initializes its value to the value in `my container`, and types this value as a `Rectangle`. (See Data Types, later in this chapter, for more information on typing.)

```
   global SomeNumber (a Number)
```

This declares the variable `SomeNumber` to always contain a descendant of `Number`. Since `Complex`, `Integer`, and `Float` are descendants of `Number`, `SomeNumber` may be any of these.

Version 1.0 Note
Type declarations are not enforced in this version of SK8.

Expressions

An expression is a phrase that can be evaluated into some object. Evaluating an expression may involve anywhere from a negligible computational overhead to time-intensive search through databases or external networks.

The SK8 MessageBox can evaluate most expressions, but it is important to remember that expressions are not the same as commands. In particular, the code bodies of functions and handlers are made of commands not expressions. In general expressions appear as part of some larger command.

Literals

The simplest expressions are **object literals** which directly represent objects. A literal always evaluates to itself.

Examples:

```
101
"Hello"
the character "g"
```

Booleans

`True` and `False` are global constants whose values denote truth and falsity, respectively, in the SK8Script language. They are vital concepts in the control of execution.

Any expression that evaluates to `False` is, by definition, false.

Any expression that evaluates to something other than `False` implies `True` and thus is true.

Examples:

The following expressions are considered to be logically true:

```
not False = True
19
```

```
"Human"
the third character in "Hello"
0 -- zero is NOT false in SK8
not (3 = 2)
```

The following expressions evaluate to false:

```
not True
True and False
19 and False
not 3 = 2 -- this is equivalent to (not 3) = 2
the eighth character in "Hello"
```

Numbers

SK8Script supports a wide variety of numeric classes arranged in a hierarchy given in Figure 5.1. SK8 tries to make this hierarchy as transparent as possible. From the standpoint of entering numeric literals, the user is typically only concerned with the distinction between integers and floating point values.

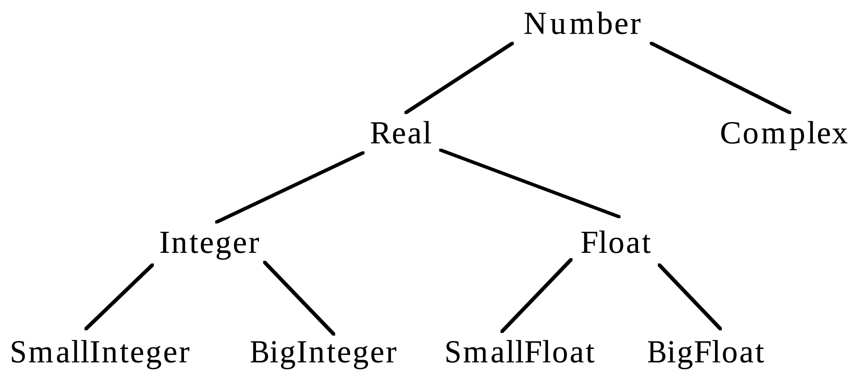


Figure 5.1. SK8's numeric hierarchy.

Integers

An integer literal consists of an optional unary sign (either "+" or "-"), a combination of the digits 0 through 9 and an optional decimal point. If no sign is given, the number is considered to be positive. If the integer value is very large ($\geq 2^{28}$), SK8 will automatically create a `BigInteger` which uses a 'bignum' representation to represent arbitrarily large integer values. Note that this is different from other common languages which either silently overflow or automatically switch to a floating point representation when integer values become too large for the standard representation.

Examples:

```
0
1
-59
+12345678901234567890.
```

Hexadecimal Integers

SK8Script will read integers in hexadecimal when they are preceded by `0x` (the digit zero followed by the letter x). Hexadecimal numbers may be given a sign, but no trailing decimal point is permitted.

Example:

```
0x1F -- evaluates to 31
-0xFF -- evaluates to -255
```

Floats

A floating point literal begins with the same components as an integer, but it must be followed either by a combination of digits after a decimal point, or an exponent. An exponent consists of the letter '`e`' followed by an optional unary sign and a combination of digits. See the on-line documentation for details on the limits of the different representations.

Objects of the class `SmallFloat` can be created using General Object Literals. However, these objects should be used with caution since any overflow causes an `ArithmeticOverflowError` to be signalled.

Examples:

```
0.0
-3.14159
1e10
+1.234e-56
the SmallFloat 1.5
```

Complex

Objects of the class `Complex` do not have a literal form but can be created with the function '`complex`'. The components of a complex value can be accessed with the functions '`realPart`' and '`imagPart`'.

Examples:

The following expressions all have the value `True.`

```
complex(0,1) equals squareRoot of -1
3 equals realPart of (complex(1,4) + complex(2,6))
0 equals imagpart of cos of complex(0,1)
```

Symbols

A **symbol** is an unchangable sequence of characters. They are used to provide informative symbolic values where appropriate. For users unfamiliar with the use of symbols, they can be thought of as fixed strings. In general they are significantly more efficient than strings. Symbols are indicated by single quotes. Any characters may appear inside the single quotes except single quotes themselves. In addition the empty symbol, `''`, is not permitted.

Examples:

```
'center' -- the symbol used to center the text of an Actor
'renderUnstretched' -- a symbol used in ImageRenderers
'Hello world' -- note embedded spaces
```

Collections

Collections provide a way to combine simpler objects into more complex objects, e.g. strings, lists or arrays. More specifically collections are descendants of the `Collection` class. Many of the special properties of collection come from their use of the *collection protocol*. It is very rare that a SK8 programmer really needs to understand the details of the collection protocol. However the details are available in the Object Reference Manual. Conceptually, the collection protocol allows a data type to be used in special ways in 'selection expressions' and special collection-related functions all of which are described later in the chapter.

Strings

A **string** is a collection of characters. A string literal is a child of `String`. It can be created by enclosing the appropriate characters with double quotes. The double quote character itself is represented by a backslash (\) preceeding the double quote. The backslash character is represented by two backslashes. See <???> for a discussion of standard string functions.[*2]

Examples:

```
"Hello world!"
"\"The time has come,\" the Walrus said"
"Here's a backslash \\"
""
```

Lists

A **list** is a collection of objects. All lists are children of the `List` object. Lists can be created by enclosing a comma separated set of expression with braces, `{}`.

Examples:

```
{1,2,3}
{1+4,{"Hello", 'world'}}
{} -- The empty list
```

Arrays

An array is another type of collection of objects. Arrays may be created using '*general object literals'* (see below). Lists can be converted into arrays using this method. If the list is a list of lists of the same size then a multi-dimensional array is created. The object ‘`vector`’ is a child of ‘`Array`’ that is constrained to be one dimensional.

Examples:

```
the Array {1,2,3} -- A one dimensional array of numbers
the Vector {4,5,6,7} -- Another one dimensional array of numbers
the Array {{1,2},{3,4}} -- A two dimensional array of numbers
the Vector {{1,2},{3,4}} -- A one dimensional array of lists
the Array {{1,2},{3,4,5}} -- A one dimensional array of lists
the Array {{1,2},3,rectangle} -- A mixed one dimensional array
```

In addition to general object literals, the `new` handler is commonly used to create arrays. See the **`new` Handler** section below. To create an array with the new handler, it is necessary to indicate its size. A one dimensional vector can be created by giving the array a `length`. A multi-dimensional vector can be created by giving a set of `dimensions`. Arrays can contain any type of objects and are initialized so that all entries are `False`. unless an ‘`item`’ is given.

Examples:

```
new array with length 10 -- A vector of 10 elements
new array with dimensions {2,3,4} -- A 3 dimensional array
-- with 24 elements.
new array with dimensions {2,3} with item rectangle
```

Calls to Executables

Calls to executables, that is function or functional-handlers, can be used as either expressions or commands. If an executable does not explicitly return a value in its definition, then its value is always `False`.

Examples:

```
on example_func
   beep
end example_func

on example_func2
   if not example_func() then
      example_func()
   end if
   return 4
end example_func2
```

After creating these functions entering `example_func2()` into the Message Box would cause the system to beep twice and return the value 4. See the Handlers and Functions section for details on the possible ways of declaring and invoking them.

#Undefined#

'`#Undefined#`' is a special reserved word that corresponds directly to the special object `#Undefined#`. This object is meant to be used as a value for variables or properties to indicate that they don't currently hold a meaningful value. `#Undefined#` will cause a "`#Undefined# is not of the expected type ...`" error for almost any operation besides writing itself.

Operators

Operators are used to construct more complex expressions from simpler ones. Operator syntax follows standard mathematical notation: unary operators are given as a prefix to their sub-expression and binary operators appear infix, that is between their two sub-expressions.

Arithmetic Operators

Any expression with a numeric value may be used with an arithmetic operator. The unary arithmetic operators are `+` which is just the numeric identity function and `-` whose value is the negation of the argument's value. The binary arithmetic operators are:

■ `+`	addition
■ `-`	subtraction
■ `*`	multiplication
■ `/`	division
■ `div`	integer divide (division rounded towards zero)
■ `mod`	modulo (sign is the same as sign of the second argument).

Integer divide and modulo are **not** restricted to positive integer classes.

Examples:

```
1+3
a*b/-c
(x div y)*y + x mod y-- Should always equal x
```

Logical Operators

Any expression may be used with any of the logical operators. As stated previously any value other than `False` is considered to be true.

The only logical unary operators is **`not`** which returns `True` if its argument is `False`. Otherwise it returns `False`.

Examples:

The following expressions return `True`.

```
not False
not (not True)
```

The logical binary operators are **`and`** and **`or`**. Both of these operators are 'short-circuiting'. This means that they may only evaluate one their operands. In the case of `and` if the first operand has the value `False`, then the value of the entire expression is `False` and the second operand is not evaluated. Otherwise, the second operand is also evaluated and its value is the value of the entire expression. For the `or` operator, only when the first operand evaluates to `False` is the second operand evaluated. If the first operand evaluates to something other than `False`, then the value for the entire expression is the value of the first operand. Otherwise the value of the entire expression is the value of the second operand.

Examples:

The following expressions all evaluate to `True`:

```
True and True
True or True
True or False
False or True
```

The following expression all evaluate to `False`:

```
False and True
True and False
False and False
False or False
```

The following examples demonstrate the short-circuiting feature:

```
beep() or beep() or 3 or beep() -- beeps twice and returns 3
3 and beep() -- beeps once and returns False
beep() and 3 -- beeps once and returns False
(False = beep()) and 3 -- beeps once and returns 3
```

Collection Concatenation Operator

The operator ‘&’ is a binary operator that takes two collections and returns a collection that contains the concatenation of the contents of its operands. The type of the result of the concatenation operator depends on the types of its operands. The type of the result is determined using the following rules in the given order:

- If ‘`is a text`’ is true of any arg then the result is a ‘`String`’
- else if ‘`is a vector`’ is true of any arg then the result is a vector
- else the result is a list

Examples:

The following expressions all evaluate to `True`:

```
"Hello" & " " & "world" = "Hello world"
"Hello" & the vector {1,2} = "Hellothe Vector {1,2}"
{1,2,3} & the vector {4,5} = the vector {1,2,3,4,5}
the array {{1,2},{3,4}} & {1,2} = {1,2,3,4,1,2}
the array {{1,2},{3,4}} & the vector {5,6} = ¬
   the vector {1,2,3,4,5,6}
the array {{1,2},{3,4}} & the array {{5,6},{7,8}} = ¬
```

```
{1,2,3,4,5,6,7,8}
```

Comparator Operators

Comparators always return either `True` or `False`. The relationships they test include object identity and type, and magnitudes between numbers. Unlike the operators discussed so far, comparators are often short lists of words rather than just single characters or words. To express all possible syntaxes the following conventions are used. **`Boldface`** indicates words that should appear exactly as they are. *`Italics`* are for components that get substituted for something meaningful from the surrounding context like a variable or object name. Components in square braces, `[]`, are optional. Sets of components in parentheses, `()`, separated by vertical bars, `|`, are syntactically interchangable though they may change the meaning.

Object Identity

Two objects are identical if the '`is the same object as`' comparator returns `True`. The word '`is`' may be replaced by '`is not`', '`isnt`' or '`isn't`' to test if the objects are different. There are two forms for this comparator. In the first form the word '`the`' is optional:

object (**is|is not|isnt|isn't**) **[the] same [object] as** *object*
[*4]

In the second form the word '`the`' is not optional

object (**is|is not|isnt|isn't**) **the object** *object*

Examples:

The following expressions all have the value `True`:

```
4 is the same object as 3+1
4 isnt the same object as 4.0 -- Integers aren't floats
Rectangle isnt the same as new rectangle [*4]
"hello" isnt the object "hello" -- Two objects are created
'hello' is the object 'Hello' -- These really are exactly the same
```

Equivalence

All objects can be tested for equivalence to other objects. The precise meaning of equivalence depends on the types of the objects. For numeric classes simple numeric equivalence is used. For characters and symbols a case insensitive comparison is done. For collection objects (e.g. strings, arrays and lists), the members are recursively checked for equivalence. If they are all equivalent, then the collections are considered equivalent. For other objects the comparison is object identity. Many syntaxes are accepted for equivalence testing:

thing [(**is**|**is not**|**isn't**|**isnt**)] (**=**|**equal**) [**to**] *thing*
thing [**is**] ≠ [**to**] *thing*
thing **equals** *thing*
thing (**does not**|**doesn't**|**doesnt**) **equal** *thing*

Note that '*thing* **is** *thing*' is **not** a valid equivalence comparator.

Examples:

The following expressions all have the value `True`:

```
4 = 3+1
"Hello" is equal to "hello"
"jello" ≠ "hello"
{1,{1,"hello"},2,3} equals {1,{1,"Hello"},2,3}
{1,2} doesn’t equal {2,1}
```

Magnitude Comparators

The magnitude comparators apply to numeric classes, characters, collections of these classes and any other objects on which the handlers '`greaterThan`' and '`equalTo`' have been defined, e.g. `DateTime`. Numeric comparisons are as expected. For characters lowercase alphabetic characters are converted to uppercase and then the ASCII values are compared. For collections the earlier elements are considered more significant.

thing [**is**] (**>|greater than**)*thing*
thing [**is**] (**<|less than**)*thing*
thing [**is**] (≥|**>=|greater than or equal [to]**)*thing*
thing [**is**] (≤|**<=|less than or equal [to]**)*thing*

Examples:

The following expressions evaluate to `True`.

```
(9 + 2) > 10
"Hello" < "Jello"
{3,2,1} ≥ {3,1,2}
"h" ≤ "H"
(ascii("h")) is greater than (ascii("H"))
```

Type Comparator

By default the type comparator tests if `A` is a descendant of `B`. See the section on Virtual and Enumerated Types for objects with more complex type relations. The syntax of the type comparator is:

object (**is|is not|isn't|isnt**) (**a|an**) *object*

Examples:

The following expressions return `True`.

```
19 is a Number
"Mike" isnt an Integer
Rectangle is a Rectangle
Number is an Object
```

Collection Comparators

There are a set of comparators that only work with collections. Examples of collections include lists, arrays, and strings. The comparators test for membership within a collection. All of them take a collection as one operand and either an object that is a valid member of the collection or another collection of such objects as the other operand. The following syntaxes can be used:

`collection` `[(`**`does not|doesn't|doesnt`**`)]` **`starts with`** *`object`*
`collection` `[(`**`does not|doesn't|doesnt`**`)]` **`ends with`** *`object`*
`collection` `[(`**`does not|doesn't|doesnt`**`)]` **`contains`***`object`*
`object` `(`**`is|is not|isn't|isnt`**`)` **`contained by`** *`collection`*

If the *`object`* operand is itself a collection, then the *`collection`* argument is searched for exactly that subsequence.

Examples:

The following expressions return `True`.

```
"Morning Maniacs" starts with the character "M"
"Morning Maniacs" starts with "Morn"
"Hello world" ends with "ld"
{1,2,{3,4}} contains 2
{1,2,{3,4}} contains {2}
{1,2,{3,4}} contains {1,2}
{1,2,{3,4}} doesn't contain 3
{1,2,{3,4}} contains {3,4}
5 is contained by {1, 4, 5, 10}
```

The following expressions return `False`.

```
{1, 2} starts with "1"
{1,2,{3,4}} contains {2,1}
"hello" contains "X"
```

Selection Expressions

Selection expressions provide a powerful, consistent way to gather together sets of objects within the SK8 environment. For example,

```
every rectangle in the stage
```

Is a valid SK8 expression that returns a list of every rectangle on the SK8 stage.

The components of a selection expression include a*source*, a *filter*, a*selector*, and a *preposition*.

The *source* is a collection which is the computational starting point of the selection expression. The source must be an expression that evaluates to an object inheriting from `Collection`. Often the source is itself a selection expression.

A *filter* limits the search within the *source* to objects that are of a specific type or objects that pass an arbitrary test. The first kind of filter is called a *type filter*, while the latter is called a *test filter*.

The *selector*, as its name indicates, is used to select a particular item or subset of the filtered source.

The *preposition* guides how a selection expression positions the reference within the items in the source.

Each of these elements and the possible syntactic variations for selection expressions are described in detail in the following sections.

The result of a selection expression is normally a member of the source collection or a list of members. However, the '`range over`' operator can be used to return a collection of the same type as the source. In addition, 'embedded selection expressions' may be used to return the value of an arbitrary expression on members of the source collection.

Examples:

The following examples are intended to give the reader a feel for how selection expressions are generally used.

```
Oval 3 in x
```

In the above example, the source collection is `x`, so the collection stored in the variable `x` is the source. The filter is the type filter '`Oval`', so the search is limited to objects in `x` that are descendants of the `Oval` object. The selector is the index `3`; that is, it selects the third object returned by the search. Finally, the preposition '`in`' searches the entire contents of the source. In summary, this selection expression represents the third object of type `Oval` found in the collection in the variable `x`.

```
character 3 through 4 in "I am okay!"
```

In this case the source is the string `"I am okay!"`. The preposition is `in`. `Character` is the type filter and `"3 through 4"` is an index range selector. This selection expression returns the list `{"a","m"}`.

```
the range over character 3 thru 4 in "I am okay!"
```

Is very similar to the previous example, except that the resulting collection is of the same type as the source. In this case the result is the string `"am"`.

```
every char before the middle word in ¬
   the text of Rectangle 1 in stage
```

This is a compound selection expression. Component selection expressions include “`Rectangle 1 in stage`” and “`the middle word in the text of Rectangle 1 in stage`”. The subexpression “`the text of Rectangle 1 in stage`” is a property access expression. The property being accessed is `text`. Note that the collection “`the text of Rectangle 1 in stage`” is used both to find “`the middle word`” and “`every char before the middle word`”. The word “`before`” is a preposition which takes an additional selector and restricts the search to those items in the source that occur before the selected item. Finally, the word '`every`' is a selector that simply returns a list of all the members of the source that match the filter.

Thus, as expected, this expression returns the list of characters from the selected rectangle's text property that occur before the middle word in that text.

The final example shows another compound selection expression used to set the `fillColor` of a certain subset of rectangles (blue above 250) in a specified top level actor (mainWindow) to random colors. This is an example of a test filter.

```
set the fillColor of every rectangle whose fillColor = blue and ¬
   whose top < 250 in mainWindow to any item in the ¬
   knownChildren of RGBColor
```

Filters

In general, filters are used to reduce the set of objects that can be returned based on properties of those objects. Type filters are used to limit the scope of a search to a specific type of object. Test filters are tests that use arbitrary user-defined criteria to filter the searched objects.

Type Filters

Type filters limit the scope of selection expression elements to things that are "of the specified type" as determined by the …`is a`… type-checking operator. A type filter must always be given. Type filters may be the name of a particular object or a variable containing an object. In addition the words '`item`' and '`thing`' are considered equivalent to the object '`object`' and thus may be used as filters that have no effect. Note that arbitrary expressions are **not** permitted as type filters.

Examples:

```
every item in stage -- The contents of stage are returned
every rectangle in the stage
number 2 in {35,20,10} -- Returns 20
word 1 in "Herr Buddenbrook" -- Returns "Herr"
```

Test Filters

Test filters are tests that use various criteria to filter the searched objects, including arbitrary tests defined by users. The possible syntaxes for test filters are

`where` *testExpression*
`whose` *continuedTestExpression*
`that` *continuedTestExpression*
`named` *nameExpression*

These test filter forms allow various types of test expression to be specified. The test filter limits the scope of a selection expression element by only permitting members for which the test expression evaluates to a non-false value.

Generally, the test expression will refer to properties of the *current target* as the search is performed. Expressions that don't refer to the current target are allowed but are rarely correct.

Example:

The following is an example of the rare case where the test expression does not refer to the current target.

```
the 1st item where (1 equals random (2)) in {1,2,3,4,5,6,7}
-- Random selection favoring the beginning of the collection
```

The reserved word ‘`where`’ is the most general form of test filter and is followed by a complete expression that is evaluated. Within the expression following the ‘`where`’, the variable ‘`it`’ may be used to refer to the object in question. In addition, the word ‘`its`’ is supported for extracting values for particular properties. E.g. the fragment ‘`where its top is greater than 100`’ is equivalent to ‘`where the top of it is greater than 100`’.

Most test filters begin with the reserved word ‘`whose`’ which is equivalent to ‘`where its`’. Thus ‘`whose top is greater than 100`’ is another equivalent expression for the example above.

Example:

```
every rectangle whose fillcolor is blue in the stage
```

The reserved word ‘`that`’ is equivalent to ‘`where it`’. This form is useful for applying a functional test or some direct comparison operator.

Example:

```
every number that equals 4 in {1,4,3,4,2,3,4}
```

Finally the reserved word ‘`named`’ is equivalent to ‘`whose name equals`’.

Example:

```
every item named "Rectangle" in {Rectangle, Oval, Rectangle}
```

Test filters may be combined using the operators ‘`and`’ and ‘`or`’. Within each test filter the reserved words may be used with the expected meanings. If the standard precedence of ‘`and`’ before ‘`or`’ is not desired, parentheses and the reserved word ‘`where`’ must be used.

Example:

```
every rectangle whose top < 100 or whose bottom > 360 in the stage

every rectangle whose top < 100 or whose bottom > 360 and whose ¬
   fillcolor equals blue in the stage
-- Picks out any rectangle whose top is < 100 along with any ¬
-- blue rectangles whose bottom is > 360

every rectangle where (its top < 100 or its bottom > 360) and ¬
   whose fillcolor equals blue in the stage
-- Picks out only blue rectangles whose top is < 100 or whose ¬
-- bottom is > 360
```

When a selection expression is evaluated, the test filter is always applied *after* the type filter. This means the expression in the test filter may assume the current target (i.e., `it`) meets the type constraint specified in the type filter.

Examples:

```
every number whose squareRoot < 5 in {1,Blue,16,24,Rectangle,25}
```

Selectors

The selector of a selection expression specify which element or elements that successfully pass any filters are returned as the result of the selection expression. Selectors can be either *singular* or *plural*. Singular selectors return a member of the source. Plural selectors return collections of members of the source.

Examples:

The next two examples are both singular. In the first case the selector is the index '`1`'. In the second case it is the reserved word '`middle`'.

```
get Rectangle 1 in the Stage
get the middle word in "The Origins of English Words"
```

The next two examples are plural. In the first case the selector is the reserved word '`every`'. The second example selects a range of elements using indices and the reserved word '`through`'.

```
get every Rectangle in the Stage
get item 2 through 4 in {1,2,3,4,5,6}
```

Singular Selection Expression Elements

The following singular selectors are defined in SK8. In each case the selector is underlined. The relative position of the selector and the type filter depends on the natural position in an English phrase. Regardless of the position of the selector the type and test filters are always applied to the source collection first, yielding a sub-collection of it; then the given selector is used to choose an item from that sub-collection. Not all singular selection expressions can be used with test filters. Those that can will explicitly include them in the given syntax expressions. Test filters are described above in the Filters sub-section.

`type` *index*

In first form, an arbitrary index is used as the selector. If the index is beyond the bounds of the sub-collection, the value of the selection expression element is `False`.

The form of the index in this form depends on the type of the collection. For linear collections, `List`s, `String`s or `Array`s, only numeric indices are expected. For multi-dimensional `Array`s a list of numbers, one for each dimension, may be used. Single numeric indices are also supported for multi-dimensional arrays. In this case the array is considered to be 'flattened' so any element may be accessed. E.g. '`item 2 of Some3DArray`' is the same as '`item {1,1,2} of Some3DArray`'. Finally, `Table`s may be indexed by arbitrary SK8 objects. See the Collection subsection for more details on tables.

Note that the simple index form **cannot** be followed by a test filter.

Examples:

The following expressions may confuse the message box if they are entered as they are. However, they all work correctly if they are transformed into commands by prepending the word ‘`get`’.

```
rectangle 1 in the stage
item 3 in "Hello world"
item {i, j} in Some2DArray
item 'foo' in SomeTable
```

[**`the`**] *ordinalIndex* *type* [*testFilter*]
In the above form, the index must be given in ordinal form: it may be one of first twelve ordinals spelled out (i.e. `first` through `twelfth`), `front` (a synonym for `first`), or a number followed immediately by one of the four ordinal endings: `st`, `nd`, `rd` or `th`.[*3]

Example:

```
the 2nd word of "say goodnight"
the (i)th word character of "some string"
```

[**`the`**] *type testFilter*
returns the first element that matches the test filter. In this case a test filter is **required**.

Example:

```
the rectangle whose top > 100 in the stage
```

[**`the`**] (**`beginning`**|**`front`**)
are contractions of ‘`first item`’. Most commonly used as part of ranges for plural selection expressions describe below.

Example:

```
the beginning of {1,2,3,4}
```

[**`the`**] (**`last`**|**`back`**) *type* [*testFilter*]
returns the last element.

Example:

```
the last number that is < 4 in {1,2,3,4}
```

[**`the`**] (**`end`**|**`back`**)
are contractions of ‘`last item`’. Most commonly used as part of ranges for plural selection expressions described below.

Example:

```
the end of {1,2,3,4}
```

[**`the`**] **`middle`** *type* [*testFilter*]
returns the middle element, that is, the `((n + 1) div 2)`th item is returned.

Example:

```
the middle word in "I came, I saw, I conquered"
```

any *type* [*testFilter*]
returns a randomly chosen member of the filtered source.

Example:

```
any card in the deck
```

anything [*testFilter*]
is a contraction of ‘`any thing`’.

Example:

```
anything whose fillColor is blue in stage
```

any of
is a contraction of ‘`any thing in`’.

Example:

```
any of the standardColors
```

Plural Selection Expression Elements

Plural selection expressions always return lists containing zero or more members of the source. If no elements match the filter criterion then an empty list will be returned.

The most general plural selection expressions are

every *type* [*testFilter*]
everything [*testFilter*]
all of

These forms evaluate to *all* of the items of the source collection that satisfy the given type and test filters. The reserved word ‘`everything`’ is a contraction of ‘`every thing`’. The reserved word ‘`all of`’ is a contraction of ‘`every item in`’.

Examples:

```
every number in {1,2,rectangle,{3,4}} -- returns {1,2}

everything whose top > 100 in stage

all of new array with dimensions {2,2}
-- returns {False, False, False, False}
```

Plural selection expressions can also return a subset of the filtered source.

type startIndex (**through** | **thru**) *endIndex* [*testFilter*]

The above form selects based on index. See the discussion of indexed singular selection expressions above for the valid set of indices.

Examples:

```
item 2 thru 3 of {1,2,3,4,5} -- returns {2,3}

item {1,2} thru {2,2} of new array with dimensions {2,2}
-- returns {False,False,False}

item 'foo' through 'bar' of SomeTable
```

Another techinque for selecting a subsequence combines two singular selectors and returns the elements between them.

singularDeterminer `(through | thru)` *singularDeterminer type* [*testFilter*]

See the section on singular selection expressions for a complete explanation of singular determiners.

Example:

```
the middle thru the last item of {1,2,3,4,5}
-- Returns {3,4,5}
```

`every` *type* [*testFilter*] **`from`** *startPathExpr* **`to`** *endPathExpr*
`everything` [*testFilter*] **`from`** *startPathExpr* **`to`** *endPathExpr*

The last two forms are known as the general range forms. Two arbitrary selection expressions are used to determine the range. First the range is computed using the two selection expressions, *startPathExpr* and *endPathExpr*; then the type and test filters are applied to the resulting range of items.

Examples:

```
everything from rectangle 1 to number 4 of {3,rectangle,4,5,6,7}
every number from rectangle 1 to number 4 of {3,rectangle,4,5,6,7}
```

Prepositions

Prepositions in selection expressions provide a way to modify how the filters and the selector view the source.

`(`**`in|of`**`)`

These are the identity prepositions. They simply pass the source collection directly to the filters unmodified. Most of the examples above use this preposition.

`(`**`after|behind|in back of`**`)` *`selectionPhrase`* `(`**`in|of`**`)`

These prepositions reduce the source by eliminating members up to and including the one specified by the selection phrase. A selection phrase consists of a selector, an additional type filter and an optional test filter.

Example:

```
item 1 after the middle item of {1,2,3,4,5} -- returns 4

item 1 through 3 after the first item that is > 2 in {1,2,3,4,5,6}
-- returns {4,5,6}
```

(**before|in front of**) *selectionPhrase* (**in|of**)

Finally, this preposition eliminates the selected member and any members following. In addition, it causes indexed searches to be done from the selected member towards the beginning of the collection rather than the default of always starting from the beginning.

Examples:

```
item 1 before the middle item of {1,2,3,4,5} -- returns 2

item 1 thru 3 before the first item that is > 4 in {1,2,3,4,5,6}
-- returns {2,3,4}
```

Embedded Selection Expressions

In addition to using selection expressions to derive single specific elements or sets of elements from the source, selection expressions can be applied to more complex expressions. In particular, expressions can appear in place of the selection phrase (everything before the preposition) of a selection expression. Within such an expression singular selection phrases may appear. When the expression is evaluated the selection phrases are evaluated with respect to the source appearing after the expressions.

Example:

```
{item 1,item 5} in {1,2,3,4,5,6} -- returns {1,5}
{any item, any item} in x -- returns a randomly selected pair
(the 1st word & space & the last word) of line 9 in x
```

Plural selection expressions can also be used in a special way as part of an expression. When a plural selection expression appears as a subexpression the entire expression is evaluated repeatedly once for each member of the plural selection expression.

Example:

```
the squareRoot of every item in {1,4,9,16} -- returns {1,2,3,4}
```

This can occasionally lead to unexpected behavior. For example, the phrase

```
the length of every item that is > 2 in {1,2,3,4,5}
```

might seem to be a valid SK8 expressions returning the value 3. However, '`every item that is > 2 in {1,2,3,4,5}`' is a plural selection expression so SK8 tries to apply the function `length` to each member of the result and fails. To avoid this problem, a plural selection expression must be converted into a list. The easiest method is to enclose it in parentheses and use a General Object Literal to coerce the result into a list. Note that parentheses by themselves are insufficient.

Example:

```
the length of the list (every item that is > 2 in {1,2,3,4,5})
```

Finally, if more than one plural selection expression appears within a given expression, then the results are all stepped through simultaneously until one of them terminates.

Example:

```
all of {1,2,3} + all of {2,4,6,8} -- returns {3,6,9}
```

Operator Precedence

As with most programming languages, the interpretation of an expression containing operators depends on the priority or 'precedence' of the operators. In general, SK8 follows the standard conventions for precedence.

Example:

```
4+8*3 equals 28
(4+8)*3 equals 36
```

The order of precedence from highest to lowest between the SK8 operators is:

`()`
`+` (unary), **`-`** (unary)
`^`
`mod`
`*,/, div`
`+, -`
`and`
`or`
`&`
`... as a(n) ...` (coercion operator. See Data Types section.)

Operators on the same line in the above table are considered to have the same precedence.

Parentheses

In most expressions, sub-expressions enclosed in parentheses are evaluated first. Any expression can be enclosed in parentheses. The resulting expression may be combined with the standard operators. Because of the behavior of embedded selection expressions described in the previous section, parentheses can behave unexpectly within selection expressions. In such cases the expression should be either simplified or broken across multiple lines storing the intermediate results in variables.

Functions and Handler Expressions

Function and handler expressions have precedence below all operators.

Examples:

The following expression is valid SK8 script

```
length of 1000 as a string
```

However,

```
length of {1,2,3} + 4
```

is an error since adding a list to a number is not allowed. The correct way to get the intended value is

```
(length of {1,2,3}) + 4
```

Finally, note that

```
length({1,2,3}) + 4
```

is also **invalid**.

Selection Expression Precedence

Expressions within selection expressions takes precedence over that of binary operators.

Examples:

Assume that the variable `x` is set to `2`. The following is illegal:

```
item x + 1 of {2,4,6,8}
```

SK8Script interprets this line as:

```
(item x) + 1 of p to 10
```

which is contains the illegal selection expression '`1 of p to 10`'. Parentheses should be used to clarify the meaning:

```
item (x + 1) of {2,4,6,8} -- returns 6
(item x + 1) of {2,4,6,8} -- returns 5
```

Get Command

`get` *expression*

This command is a shorthand for:

`set result to` *expression*

That is, the variable `result` is set to *expression*. The get command is the simplest way to transform an expression into a command. It also is the motivation for calling the property value accessing functions "getters". However, in actual practice `get` is rarely used.

`Example:`

The following statement sets `result` to the value of the property `name` in the object `John`.

```
get John's name
```

Assignment

The value at any place or location where objects can be stored (e.g., variables, collections, and properties) is changed using the `set` command.

`set` *location* `to` *expression*

Location can be a variable, the name of an executable, a selection expression, or a list destructurer. Each of these are described in the following subsections.

Assignment to Variables

`set` *variableName* `to` *expression*

The value of*expression* is stored in the variable *variableName*..

Examples:

```
set x to "Good" -- x now has the value "Good"
set x to x & " morning!" -- x new has the value "Good morning!"
```

Assignment Using Executables

`set` *setterName* `to` *expression*

This form is used for changing the value of properties and for calling any user defined ‘setters’. Setters are specially defined executables whose conceptual purpose is to change some value. It is possible for setters to use additional arguments as described in the Handlers and Functions section below. The setter name can be either a function name or a handler name with the name of the object to be effected.

Examples:

```
set fillColor of SomeRect to Blue -- a property (also a setter)
set width of SomeRect to 100 -- a setter
```

Assignment Using Selection Expressions

`set` *selectionExpression* `to` *expression*

Any valid selection expression may be given. The result is that the item (or items) that would normally be returned as the result is modified in the source collection to have the value of the expression.

Example:

The following statements illustrate the use of `set` with some simple selection expressions. In each case assume that the variable ‘`x`’ has been set to the value `{1,2,3,4,5}`. The resulting value of x is given as a comment.

```
set item 3 of x to 6 -- {1,2,6,4,5}
set item 1 to 3 of x to 7 -- {7,7,7,4,5}
set every number that is > 2 in x to 0 -- {1,2,0,0,0}
set item 1 of x to item 3 of x -- {3,2,3,4,5}
```

Assignment Between Selection Expressions

In the last example above a singular selection expression is assigned to another singular selection expression. In this case the semantics are obvious. However when either or both are plural the semantics become somewhat more complex. The behavior of `set` for each case is described below.

Plural location and Singular value

When only the location is plural, the value is recomputed for each iteration of the `set` operation over the given collection of places.

Examples:

In the following example, each oval in `Stage` is assigned a potentially different color. The clause "`any item in the knownChildren of RGBColor`" is recomputed each time a `fillColor` is assigned to one of the ovals in `Stage`.

```
set the fillColor of every oval in the stage to ¬
   any item in the knownChildren of RGBColor
```

Singular location and Plural value

The whole collection represented by the plural value is stored into the location and no iteration occurs.

Example:

If '`x`' starts with the value `{1,2,3,4}`, the following command changes its value to `{{1,3,5},2,3,4}`.

```
set item 1 in x to everything in {1,3,5}
```

Plural location and Plural value

When both the location and value are plural, the `set` operation iterates over the collection of locations and the collection of values in parallel. If the length of the two collections is different, then the number of iterations is equal to the length of the shorter of the two collections.

Example:

In the following example, the first three ovals in the stage will receive the `fillColor` `Red`, `Blue`, and `Green`, respectively.

```
set the fillColor of every oval in the stage ¬
   to everything in {Red, Blue, Green}
```

In the next example, each oval in the stage is assigned a fill color from the rectangles on the stage.

```
set the fillColor of every oval in the stage ¬
   to the fillColor of every rectangle in the stage
```

Assignment Using Destructurers

`set` *listOfPlaces* `to` *expression*

When the location of the `set` command is a list of valid location, SK8 automatically 'destructers' the list into a set of assignments. In order to be successful the result of the expression given must be a collection containing the same number of elements as there are places. An error will be signalled if the dimensions of the list of places and the expression don't match.

Example:

The following statements illustrate list destructurers.

```
set {h,v} to the location of SomeWindow
```

Destructurers are very efficient and should be used whenever possible. For example, the above command will be noticably faster and will use less temporary memory than the separate commands '`set h to item 1 of the location of SomeWindow`' and '`set v to item 1 of the location of SomeWindow`'.

Flow of Control

SK8Script provides a variety of flow of control constructs for conditional execution, looping and a special command for pausing execution.

Conditionals

The conditionals in SK8Script include a simple single line form, a multi-line form and a multi-branch form similar to the 'case' statements from other languages.

Single Line Conditional: If… then… else

The syntax for the single line conditional is:

`if` *expression* **`then`** *command1* `[`**`else`** *`command2`*`]`

If *`expression`* evaluates to something that is not `False`, *`command1`* is executed. Otherwse *`command2`* is executed if it is provided. If there is ambiguity of the association of an '`else`' with a set of nested '`if`', it associates with the closest '`if`'.

Example:

```
if today's dateString ≠ "December 31, 1994" then beep
if today's dateString = "December 31, 1994" then beep 2 ¬
   else beep 3
if true then if false then beep 2 else beep 3 -- beeps 3 times
```

Multi-line Conditional

The syntax for the multi-line conditional is

if *expression* **then**
   commandList1
[else
   commandList2]
end if

or

if *expression* **then**
   commandList1
else *elseCommand*

If *expression* evaluates to something that is not False, the *commandList1* is executed. If *expression* is False, then the *commandList2* or *elseCommand* (if present) is executed. Note that 'end if' is part of only the first form. The second form allows for the common 'else if' syntax.

Example:

```
if Rect1's fillColor = Red then
   beep 2
   set Rect1's fillColor to Blue
else
   beep 3
   set Rect1's fillColor to Red
end if
```

In the next example note the nested conditionals.

```
if today's day is odd then
   beep
   sendToLog "Today's an odd day"
   if today's year = 1994 then
      sendToLog "and it's 1994"
   end if
else if today's day is > 14 then
   beep 2
   sendToLog "today's an even day near the end of the month"
   if today's year = 1995 then sendToLog "and it's 1995!"
end if
```

Multi-branch Conditional: If… is one of…

This conditional allows the kinds of decisions that are usually performed using 'case' or 'switch' statements in other languages. The syntax is

if [*caseExpression* (**is**|*TestComparator*)] **one of**
   (*conditions* : *conditionalCommandList*)*
[**else**
   alternateCommandList]

end if

If the alternate command list is a single command, then the following alternative syntax maybe used.

```
if [caseExpression (is|TestComparator)] one of
   (conditions : conditionalCommandList)*
else alternateCommand
```

When this statement is executed, at most one of the enclosed command lists are executed. The case expresson is evaluated first. If no case expressions is given then it is assumed to have the value `False` and the test operator is '≠'. The conditions are comma separated 'condition expressions'. Each condition expresssion is evaluated in turn and compared to the value of the case expression using the test comparator. The conditional command list corresponding to the first condition expression that matches the case expression is executed. The alternate command list is executed only if none of the condition expressions are matched. The test comparator can be any of the comparators described in the Expressions section above. If '`is`' is used instead of a test comparator then the rest of the comparator is prepended to the condition expression. This allows different comparators to be used in each case.

Examples:

The first example has no case expression, so the first condition expression that evaluates to a value other than `False` is chosen.

```
if one of
   Person is sleeping: set person's partying to False
   today's dateString with dayOfWeek starts with "S", ¬
      (today's hour) >= 17: -- Saturday or Sunday or after 5pm.
      sendToLog "Let's party!"
      set person's partying to True
else sendToLog "Nothing to do!"
```

In the next example a fixed case expression and test comparator are given.

```
if today's dateString contains one of
   "r", "m": sendToLog "It's not summer!"
   "j": sendToLog "Relax, it's summer"
else
   sendToLog "School is on the horizon!"
end if
```

The final example, uses the word '`is`' to vary the test comparator.

```
if  selectedObj is one of
   an actor:  insert selectedObj into the viewArea
   a collection:
      set the items of myListViewer to selectedObj
      set viewArea's text to "It's a collection"
   not a number:
      beep
      set viewArea's text to "Who knows."
   > 0:  set viewArea's text to "It's a positive number."
   < 0:  set viewArea's text to "It's a negative number."
else set viewArea's text to "It must be zero!"
```

Iteration (Looping)

Most explicit looping in SK8 is done with the `repeat` command. The general syntax of this command is:

repeat [*loopControlClause*]
   commands
(**end repeat**|*loopControlClause*)

Exactly one of the two loop control clauses must be used. In addition, only the `while` and `until` loop control clauses can appear in the end position. When the clause appears at the end, the loop is guaranteed to be executed at least once.

Clauses

Different types of iteration are provided by varying the loop control clause.

Forever

If the loop control clause is just the reserved word '`forever`', the loop to repeated continuously until it is explicitly ended by a '`Condition`', '`return`' or '`exit`'. All of these are described later in this chapter.

Example:

```
repeat forever
   beep  -- beeps forever; very annoying. Cmd-. will stop it.
end repeat
```

Times

This clause executes the given commands some number of times. In this form the index of the loop is not available. If the current loop index is needed, see the 'with…from…to' clause below.

[**for**] *integerValue* **times**

Examples:

```
repeat 5 times    -- beeps 5 times; less annoying
   beep
end repeat

repeat for 3 times      -- beeps 3 times; just annoying
   beep
end repeat
```

With…from…to [*5]

This clause is used to iterate over a sequence of numeric values.

`with` *`var`* **`from`** *`start`* `[`**`to`** *`end`*`]` `[`**`by`** *`step`*`]`

`Var` must be a variable name which is automatically declared as a local. `Var` is initially set to the value of `start` and incremented by `step` as long as the value is approaching or at the value of `end`. The exact meaning of approaching depends on the value of `step` in the obvious way. The `start`, `end` and `step` must be expressions whose values are descendants of the object `Real`. Each expression will be evaluated exactly once at the beginning of the loop. The value of `step` defaults to `1`. If no value is given for `end`, the loop continues forever. The value of `var` at the end of the loop is undefined.

Example:

```
-- sends 3, 4, 5, 6, and 7 to the MessageBox
repeat with index from 3 to 7
   sendToLog index
end repeat

-- sends 7.1, 5.0, and 2.9 to the MessageBox
repeat with index from 7.1 to 2.3 by -2.1
   sendToLog index
end repeat

repeat with index from 3
   sendToLog index
   if index is greater than random(index*3) then exit repeat
end repeat
```

With…in [*5]

This clause allows for iteration over an arbitrary collection.

`with` *`var`* **`in`** *`expression`*

`Var` must be a variable name which is automatically declared as a local. The loop body is executed with `var` set to each member of the collection returned by the expression in turn. The value of `var` at the end of the loop is undefined.

Example:

```
--print "red", "green", and "blue" into the MessageBox
repeat with aWord in {"red", "green", "blue"}
   sendToLog aWord
end repeat
```

While

This clause repeatedly evaluates an arbitrary expression and executes the loop body as long as the value of the expression is not `False`. The syntax is

while *test*

Example:

The following example is has the same output as the first example for the ‘with…from…to’ clause. In this case `aNumber` is guaranteed to have the value 8 at the end of the loop.

```
global aNumber = 3

repeat while aNumber is less than or equal to 7
   sendToLog aNumber
   set aNumber to aNumber + 1
end repeat
```

This loop can be rewritten to use the end clause syntax as

```
global aNumber = 3

repeat
   sendToLog aNumber
   set aNumber to aNumber + 1
while aNumber is less than or equal to 7
```

Note that if the 7 in the above example were replaced with 2, the loop would still execute once, as compared to the previous example which would never execute the body of the loop.

Until

This clause is identical to the while-clause except that it terminates as soon as the value of the test is non-`False`. The syntax is

until *test*

Example:

This example has the same output as the example for the while-clause. Again `aNumber` is guaranteed to have the value 8 at the end of the loop.

```
global aNumber = 3

repeat until aNumber is greater than 7
   sendToLog aNumber
   set aNumber to aNumber + 1
end repeat
```

Again this may be rewritten to use the end clause syntax as

```
global aNumber = 3

repeat
   sendToLog aNumber
   set aNumber to aNumber + 1
```

```
until aNumber is greater than 7
```

Loop Exits

Within the context of a repeat loop special commands are provided to exit the entire loop and to skip to the next iteration. These commands are almost always used in conjunction with conditionals.

Local Exits

The command

`exit repeat`

causes the containing loop to terminate and control to pass to the command after the appropriate ‘`end repeat`’ statement.

Example:

This loop waits until the current minute is odd, then beeps and exits.

```
repeat forever
   if now's minute is odd then
      beep
      exit repeat
   end if
end repeat
```

Exit to Next Iteration

The command

`next repeat`

skips the remaining loop body and starts the next iteration of the `repeat`.

Example:

The following loop outputs the numbers from 1 to 10, but only beeps 5 times.

```
repeat with i from 1 to 10
   sendToLog i
   if i is odd then next repeat
   beep
end repeat
```

Wait

The wait command provides a mechanism for suspending SK8 either for a specified period of time or until some condition is met. To wait for a period of time the syntax is:

`wait` `[`**`for`**`]` *`numberOfUnits timeUnit`* `[(`**`with`**`|`**`without`**`)` **`events`**`)])`

The number of units is an expression that evaluates to a real and time unit may be any one of `minute`, `minutes`, `second`, `seconds`, `tick`, `ticks`, `millisecond` or `milliseconds`. A tick is one sixtieth (1/60) seconds. If `without events` is used then all event processing is suspended until the time elapses.

Examples:

```
wait 20 ticks
wait for 1 second
wait 2.4 seconds
wait 0.2 minutes
```

The second form of the wait command is

`wait` `[(`**`with|without`**`)` **`events`**`)] (`**`while|until`**`)`*`conditionExpression`*

The condition expression is repeatedly evaluated until it evaluates either to `False` or to a value other than `False` depending on whether `while` or `until` is used, respectively. If `without events` is used then all event processing is suspended until the condition is met.

Examples:

```
wait while mouse is down
wait without events until commandKeyDown()
```

Collection Commands

As discussed in earlier sections, collections are objects which contain sets of other objects. The creation of most type of collections is discussed in the Expressions section. The remaining common collection type, `Table`, is discussed below. Values within collections can be accessed or changed using Selection Expressions. The principle subject of this section are the commands `insert` and `remove` for adding and deleting elements from collections respectively.

Not all collections support these commands, e.g. insert is not supported for children of the `Table` object. In this case the same functionality can be achieved with Selection Expressions. Collections that completely support the commands include descendants of `List`, `String` and `Array`.

These commands only guarantee that the value of a location explicitly described in the command will be changed. It does not guarantee either that a new object will be created or that an existing object will be modified. This choice depends on the type of the collection.

[*6]In particular, `Vector` and `Array` objects will in general return different objects, whereas `List` objects will usually modify the existing list.

Tables

The `Table` object provides and easy to associate pairs of objects. The use of tables is simiilar to arrays except that rather than using numeric indices, the indices can be any object (including numbers). If a value is requested for an element that has not been set, the value `False` will be returned. There is, however, a difference between an unset value and a value set to `False`. In the later case the value is contained in the Table. This difference is most evident when using plural selection expressions. Tables are created using the standard `new` and `copy` discussed later in this chapter.

Examples:

The following example both shows how to create and use a `Table`.

[*7] Check all the following examples

```
new Table with objectName "SomeTable" -- Create a Table

-- Add some items
set item 'foo' of SomeTable to 5
set item Oval of SomeTable to Yellow

get item 'foo' of SomeTable -- Returns 5
get item Oval of SomeTable -- Returns Yellow
get item 'Oval' of SomeTable -- Returns False (no entry)

get every item in SomeTable -- Returns {Yellow, 5}

set item 'foo' of SomeTable to False
get item 'foo' of SomeTable -- Return False (explicitly set)
get every item in SomeTable -- Returns {Yellow, False}

remove item 'foo' of SomeTable
get item 'foo' of SomeTable -- Return False (no entry)
get every item in SomeTable -- Returns {Yellow}
```

Insert Command

The `insert` command is used to add objects to collections. The general syntax of the insert command is

`insert` *`source target`*

The *`target`* describes the collection or collections to be effected and the position where the value or values are inserted. Multiple collections and multiple values are specified using plural selection expressions. The *`source`* can be any arbitrary expression. The following subsections describe the different type of ‘insertion targets’.

Into...

The syntax of this target is

`into` *place*

This insertion target leaves the point of insertion unspecified. The type of *`place`* determines the insertion point. In general, this means the most efficient position will be used.

Examples:

Assuming that `x` has the value ‘`{1,2,3}`’

```
insert 0 into x
```

results in `x` having the value ‘`{0,1,2,3}`’.

On the other hand, if `x` has the value ‘`the Vector {1,2,3}`’, the same insertion command results in `x` having the value ‘`the Vector {1,2,3,0}`’.

If `x` has the value `{{1,2,3},the Vector {1,2}}`,

```
insert 0 into x
```

results in `x` having the value ‘`{0,{1,2,3},the Vector {1,2}}`’.

However, if the target is a plural selection expression, then the source is inserted into each member in turn. Thus

```
insert 0 into every item in x
```

results in `x` having the value ‘`{{0,1,2,3},the Vector {1,2,0}}`’.

Similarly if the source is a plural selection expression, then each member of the source is inserted into the target in turn,

```
insert every item in {4,5} into x
```

results in `x` having the value ‘`{5,4,{1,2,3},the Vector {1,2}}`’.

Finally, if both the source and the target are both plural selection expressions, the iteration proceeds down both collections in parallel and terminates when one of them is exhausted.

```
insert every item in {4,5,6} into every item in x
```

results in `x` having the value ‘`{{4,1,2,3},the Vector {1,2,5}}`’.

At Beginning...

The syntax of this target is

`at` `[`**`the`**`]` `(`**`beginning|start|front`**`)` `(`**`of|in`**`)` *expression*

This target inserts the value at the beginning of the collection(s) given by *expression*.

Examples:

```
insert 10 at the beginning of x
```

The statement above inserts `10` at the front of the collection in `x` regardless of its type.

```
insert 10 at the beginning of every item in x
```

The statement above inserts `10` at the front of the collections contained in `x`. Note that if `x` contains elements that are not collections, an error will be signalled and any collections in `x` may or may not of been changed. A safer command would be

```
insert 10 at the beginning of every collection in x
```

at End...

The syntax of this target is

at [**the**] (**end**|**back**) (**of**|**in**) *expression*

This target inserts the value at the beginning of the collection(s) given by *expression*.

Examples:

```
insert 10 at the end of x
```

The statement above inserts `10` at the end of the collection in `x` regardless of its type.

General Insertions

These targets use a selection expression to choose a location or locations within the source collection to insert the value. To insert the value in front of the specified location(s) the syntax is

(**before**|**in front of**)*selectionExpression*

To insert the value after the specified location(s) the syntax is

(**after**|**behind**|**in back of**) *selectionExpression*

If a singular selection expression is given, the insertion is performed at the position specified by that expression; if a plural selection expression is given, the insertion is performed at each of the positions it specifies.

Examples:

Each of the following example assumes that `x` has been set to `{1,2,3,4}`. The new value of `x` after executing the command is given as a comment

```
insert 2.5 after item 2 in x -- {1,2,2.5,3,4}
insert 2.5 before item 3 in x -- {1,2,2.5,3,4}
insert 2.5 before the item that is > 2.5 in x -- {1,2,2.5,3,4}
insert 0 before every item in x -- {0,1,0,2,0,3,0,4}
insert 0 before every item that is odd in x -- {0,1,2,0,3,4}
insert every item in {4,5,6} before every item that is odd in x
-- {4,1,2,5,3,4}
```

Remove Command

The `remove` command is used to delete items from collections. The syntax of the general form of this command is

remove *selectionExpression*

The item or items described by the selection expression are removed from the source collection. In addition to the general form there is a special syntax

remove *value* **from** *expression*

This form is equivalent to the general form

[*8][remove the item that is the object *value* of *expression*

This means that only the first item that matches `value` is removed. Note that `from` is **not** an accepted preposition in a selection expression.]

After any `remove` command has been executed, the value of `result` is set to a list of the removed members of the source collection.

Examples:

Each of the following example assumes that `x` has been set to `{1,2,3,4,2}`. The new value of `x` after executing the command is given as a comment

```
remove item 1 of x -- {2,3,4,2}
remove the item that is > 2 in x -- {1,2,4,2}
remove every item that is > 2 in x-- {1,2,2}
remove 2 from x -- {1,3,4,2}
remove every item that equals 2 in x -- {1,3,4}
[*15]remove the last item that equals 2 in x -- {1,2,3,4}
[*8]remove every item in {1,2} from {1,2,3,4} -- {3,4}
[*8][If the above example is made to work the text above in square
braces should be revised. If this feature is not included in the
final release then the example should be removed]
```

Creating New Objects

Most simple objects such as `Numbers`, `Strings`, `Lists` or even `Arrays` can be created with literals or general object literals as described in the Expressions section. However, more complex object such as `Actors` are created with one of two special handlers, `new` and `copy`. The first of these creates an new object that is a child of a given object. The second creates an identical sibling object.

Both the `new` and `copy` handlers call the handler `initialize` after the object has been created and its properties have been initialized. This handler is provided as a hook for the programmer so that any special initialization work they desire can be handled automatically. The default `initialize` handler does nothing.[*16]

The New Handler

```
new object [with objectName (a String)] ¬
   [with project (a Project)] ¬
   [with properties (a List)] ¬
   [with otherParents (a List)] ¬
   [SpecializedKeywordArguments]
```

The `new` handler is used to create a child of *`object`*. It returns the child. The new child will inherit all of the properties of the given object, except as described below.

`objectName`	The optional object name you may wish to assign to the child. If you do not assign an object name, you may still assign it at a later time using the `objectName` handler. Objects can also be renamed by setting the `objectName` property. The default is that the object is not named. This value is **never** inherited.
`project`	The project that will contain the new object. The default is the project to which the surrounding handler or function definition belongs. In the special case of the message box, the default project is shown in the title bar.
`properties`	A list of symbols that are the names of desired properties for the new object. Properties created in this way are public, not propagated, and have an initial value of `False`. See the following section on Properties for an explanation of these attributes.
`otherParents`	An optional list of objects that represent other objects, in addition to the given *`object`*, from which the child should inherit handlers and properties. That is, *object* and `otherObjects` will be the parents of the child. *Object* is called the 'base parent' of the child.
`SpecializedKeywordArguments`	Used to set values for any property of the new object or to automatically call any defined setter-handlers. Example: `with location {60, 60}`

Examples:

```
new object with objectName "TooSimple"
```

The children of very simplest object, `object`, are useful for creating compound data structures similar to the records or 'structs' of other languages.

```
new object with objectName "Address" ¬
   with properties {'name', 'street', 'city'}

new Actor with objectName "MediaCollection"
   with project MyProject
   with otherParents {Collection}
   with location {60,60}
```

IMPORTANT

[*9][More like bogus! Try

```
new rectangle with objectname "rect1"
new rectangle with objectname "rect2" with otherparents {rect1}
new rect1 with objectname "rect3" with otherparents {rectangle}]
```

You should not supply objects in `otherParents` that inherit from the same parents (except from `Object`), or else you will be introducing a circularity into the descendancy of your new object. This will generate an error. ▲

The Copy Handler

```
copy object ¬
   [with objectName (a String)] ¬
   [with project (a Project)] ¬
   [SpecializedKeywordArguments]
```

The resulting copy will have the same parents as the original and it will have the same local handlers, properties and property values defined with the exception of the object name. Unlike the `new` handler these components of the new object are independent of those of the original object. Thus changes to one of them after the initial copy do not effect the other.

`objectName`	The optional object name for the copy. The default is for the object to have no value for `objectName`.
`project`	The project in which the copy should be created. The default is the project of the original object.
`SpecializedKeywordArguments`	Optional arguments used to override the property values (virtual or otherwise) in the original object from which the copy is made.

Example:

Suppose you created a rectangle called `SomeRectangle`, specialize it as shown, and want an exact copy of it:

```
set SomeRectangle's fillColor to Red
set SomeRectangle's frameColor to Blue

on Click of me (a SomeRectangle)
   beep
end Click

copy SomeRectangle ¬
   with objectName "CopyOfSomeRectangle" ¬
   with frameColor Green
```

The `fillColor` of `CopyOfSomeRectangle` will be `Red` and its `frameColor` will be `Green`. `Click` will also be defined for `CopyOfSomeRectangle`.

If you now redefine the `SomeRectangle`'s click handler as

```
on Click of me (a SomeRectangle)
   set my fillColor to Blue
end Click
```

The click behavior of `SomeRectangle` will be changed, but the behavior of `CopyOfSomeRectangle` will remain the same. If, on the other hand, `CopyOfSomeRectangle` had been created using `new` instead of `copy`, then its behavior would have also changed.

Creation Relations

Creation relations provide a mechanism modeling complex compound objects in SK8.

Consider a familiar object such as a guitar:

- We generally think about a guitar as a unit, even though we recognize that it is made up of several distinct objects; in particular, each string is a distinct object but one that is closely related to the guitar, one that is part of the guitar.
- On a given guitar we refer to the d-string, for example, as “the d-string of Hernán’s guitar” — that is, we refer to it via its relation to another object, rather than with a more absolute identification.
- If we create a new guitar, we know we also need to create new strings and attach them to this new guitar just as the other strings are attached to the other guitar.

Creation relations do the following three things:

- They enable compound objects in an object-oriented programming environment to be modeled as we tend to model such objects in the real world.
- They allow sub-objects to be referenced and identified in terms of their relation to their root object.
- They automate the creation and linking up of the corresponding sub-objects when a copy (or instance) of the root object is created.

Declaring Creation Relations (Modeling Interrelations)

The first step in creating a compound object is simply to build the prototype; that is, build all the prototypical pieces with appropriate properties tying them together.

Example:

Using the guitar example, we create a guitar object with a `strings` property. Then we create six guitar string objects, instances of a prototypical guitar string object with a `guitar` property. We set `guitar's strings` to a list containing the six `GuitarString` objects, and we set each guitar string’s `guitar` property to the `guitar` object.

```
new object with objectName "Guitar" with properties {'strings'}
new object with objectName "GuitarString" with properties ¬
   {'guitar'}

set Guitar's strings to {}

repeat 6 times
   insert a new GuitarString into Guitar's strings
end repeat

set the guitar of every item in Guitar's strings to Guitar
```

Note that the `guitar` and `guitarString` objects could, of course, have additional properties such as `make` and `model`, or `thickness` and `tension`.

Once the prototype object is built and its interrelations set up, the creation relations are declared; they describe the way in which the properties of sub-objects embody their interrelations in forming a compound object or “creation group”.

The `guitar`’s creation relations would include its `strings` property, along with an indication (an asterisk) that this property is considered plural (i.e. it holds a collection of objects rather than a single object). Our prototypical guitar string’s creation relations would include its “guitar” property, along with an indication (square brackets) that this relation is a simple, non-creation relation. If this creation relation were left out, then the value of `guitar` in each of the new `GuitarString`s would be just be copied so their value would be the prototype object `guitar` rather than the newly created child of `guitar`. Note that the six string instances we made automatically inherit the creation relations from their prototype.

```
set Guitar's localCreationRelations to {'strings*'}
set GuitarString's localCreationRelations to {'[guitar]'}
```

The complete syntax for creation relations is

```
'[[]propertyName[*][]]'
```

Identifying Objects Via Relations

Notice that we have named the prototype guitar string object (i.e. supplied it a global `objectName` by which it can be directly referenced), but we haven’t named any of the six instances we made. However, we can still refer to a given string relative to the guitar object. For example, the third string in our guitar can be referred to as:

```
GuitarString 3 in the strings of Guitar
```

And, in fact, since each string “knows” which guitar it belongs to (via its “guitar” property), that string will identify itself (i.e. write out its textual representation) as the same expression:

```
GuitarString 3 in the strings of Guitar
```

The Purpose of Creation Relations: Automatic Instantiation of Creation Groups

Though most object systems (including SK8’s) give the programmer a hook into the creation of objects (thus allowing the installation of code that creates sub-objects and links them to the root), most such systems require code to be written specially to deal with each new type of compound object.

Creation relations handle the problem of creating compound objects generally enough that, in most cases, the programmer does not need to write any such “object initialization” code at all. Once the creation relations are specified, the creation group associated with a given root object can be automatically determined by traveling through the object’s creation relations —keeping track of which objects have been visited— until closure is reached. Instantiating the root object then automatically causes all objects in the creation group to be instantiated and interrelated in the same way the prototypes are interrelated.

To finish our example, since the guitar's and guitar strings' creation relations have already been declared, we can simply make a new guitar and notice that it has six new guitar strings:

```
new Guitar with objectName "YamahaGuitar"

get item 1 in YamahaGuitar's strings
-- returns an object that identifies itself as:
--   item 1 in the strings of YamahaGuitar

get whether item 1 in YamahaGuitar's strings ¬
   is the same object as item 1 in Guitar's strings
-- returns False

get YamahaGuitar's strings
-- returns the list:
--   {item 1 in the strings of YamahaGuitar, ¬
--    item 2 in the strings of YamahaGuitar, ¬
--    ...
--    item 6 in the strings of YamahaGuitar}
```

Creation Relations and Actors

SK8 users who are familiar with creating and using complex graphical objects have actually been making use of creation relations. The `creationRelations` of `Actor` are `{'contents*', '[container]', 'lines*'}`, which explains why new content objects are created whenever a new compound `Actor` is created.

Properties

A property is much like a variable except that, since a property belongs to an object, both the property name and the object must be specified in order to reference the property's value. Values are stored in and can be retrieved from the properties.

Accessing Properties

There are two equivalent syntaxes for accessing a property of an object.

propertyName **of** *expression*
(*expression*'**s**|**my**|**its**) *propertyName*

In both cases the value of the named property is accesssed from the object returned by the expression. By themselves these forms are expressions with the given value. The forms may also be used as part of a `set` command in order to change the given value.

The reserved word '`my`' may only be used within a handler definition and is used to access values of the invoking object. It is the natural contraction of '`me's`'. Similarly the reserved word ‘its’ may only be used within selection expressions and is used to access values of the current element.

Examples:

```
frameColor of Rectangle
Rectangle's fillColor
set frameColor of SomeRectangle to Green
set (anything in stage)'s fillColor to Blue
```

The next example uses the reserved word ‘my’

```
on Click of me (a SomeRectangle)
   if my fillColor equals Red then
      set my fillColor to Blue
   else
      set my fillColor to Red
   end if
end Click
```

Finally, here an example using ‘its’

```
set container of everything where 100 > its top in stage to False
```

Accessors (getter or setter)

In SK8 Script there is a close relationship between handlers and properties. At the syntactic level there is no difference between accessing a property value and calling a handler with the same name. It is common in SK8 to create ‘virtual properties’ by creating a handler with a given name and setter handler that shares that name. Obviously it is up to the programmer to ensure that these handlers perform in the expected way.

A good example of a virtual property is the `width` of the `Actor` object. The position of an actor within its container is controlled by the `boundsRect` property. The `width` handler uses this value to compute its value. Similarly, the `set width` handler modifies the value of this property in such a way to guarantee that a later call of the the `width` handler will have the expected value.

In addition, to virtual properties, a handler with the same name as one of an objects properties can be created. Such handlers take priority over the default behavior of getting or setting the value of the property. In this case, the actual property can only be accessed by calling the special command ‘`do inherited`’ within these handlers. This type of handler is commonly used to guarantee some attribute like the type of a property.

Example:

The following SK8Script code creates a ‘beep switch’ object that can be on or off and beeps whenever its value is read. A setter handler is defined to ensure that it can only get set to one of these two values. A getter handler is also defined to do the beeping. In addition two virtual properties ‘`isOn`’ and ‘`isOff`’ are defined.

```
new object with objectName "BeepSwitch" with properties {'state'}
```

```
on set state of me (a BeepSwitch) to newValue
   -- Define the setter of the state property to ensure
   -- that only valid values are given
   if newValue is the object one of
      'on', 'off': return do inherited
   else General Error ¬
      with strings {"A beep switch can only be 'on' or 'off'"}
end set state

on state of me (a BeepSwitch)
   -- Let it beep
   beep
   -- Do the actual property access:
   return do inherited
end state

on isOn of me (a BeepSwitch)
   -- Getter for virtual property isOn
   return 'on' is the same as my state [*4]
end isOn

on set isOn of me (a BeepSwitch) to newValue
   -- Setter for virtual property isOn
   if newValue then
      set my state to 'on'
   else
      set my state to 'off'
   end if
end set isOn

on isOff of me (a BeepSwitch)
   -- Getter for virtual property isOff
   return 'off' is the same as my state [*4]
end isOff

on set isOff of me (a BeepSwitch) to newValue
   -- Setter for virtual property isOff
   if newValue then
      set my state to 'off'
   else
      set my state to 'on'
   end if
end set isOff
```

Property Attributes

Properties have two attributes, *private* and *propagated*, that determine the manner in which they are inherited.

If a property is *private* then it can only be accessed using the object that owns it or handlers defined on this object.

1.0 Note
Private properties are incompletely supported. Properties may be declared as private, but the access limitations are not enforced.

If a property is *propagated* then changes to the value of a parent object are automatically propagated to its children unless its value has been explicitly overridden.

A property can be made to propagate or not using the '`makePropertyPropagate`' and '`makePropertyNotPropagate`' handlers respectively.

Example:

Continuing the `BeepSwitch` example from above, first set the value of the `BeepSwitch`'s `state`

```
set state of BeepSwitch to 'on'
```

Make the state property propagate

```
makePropertyPropagate BeepSwitch, 'state'
```

Create a child

```
new BeepSwitch with objectName "NewSwitch"
```

At this point the state of `NewSwitch` is copied from `BeepSwitch` so it is 'on'. This is the normal behavior whether or not the property is propagated. Now we turn `BeepSwitch` 'off'.

```
set state of BeepSwitch to 'off'
```

Since the state property propagates, `NewSwitch's state` is also set to 'off'. If the state property did not propagate, `NewSwitch's state` would have remained 'on'.

However, if we now explicitly set `NewSwitch's state`

```
set state of NewSwitch to 'on'
```

This overrides the propagation, so the value of `BeepSwitch's state` is still 'off', but `NewSwitch's state` is now 'on'. At this point changes to `BeepSwitch's state` no longer effect `NewSwitch's state`. However, the property still propagates, so the values of other children of `BeepSwitch` that haven't overridden the value will continue to propagate. Furthermore, any children of `NewSwitch` will now derive their values from `NewSwitch`. The connection between `NewSwitch` and `BeepSwitch` can be reestablished using the '`propagateValue`' function,

```
propagateValue 'state' from BeepSwitch to NewSwitch
```

`NewSwitch's state` now has the same value as `BeepSwitch's state`. However, any children of `NewSwitch` created while the two were disconnected will remain, disconnected. [*10]

Adding and Removing Properties

When dealing with properties for the purposes of listing, adding or removing, a special set of symbols known as ‘property symbols’ are used. Valid property symbols are composed only of letters, digits, and the underscore character, ‘_’. In addition, they cannot start with a digit. These restrictions allow the properties to be used later as handler names (identifiers) in SK8 script.

The properties owned by an object are stored in the property ‘`localProperties`’. Properties may be directly added or deleted from this list using the usual ‘`insert`’ and ‘`remove`’ commands. However, the two handlers `addProperty` and `removeProperty` provide a simpler interface.

AddProperty

```
addProperty object, propertySymbol ¬
   [with private] ¬
   [with propagatedValue] ¬
   [with initialValue [value]] ¬
```

The `private` and `propagatedValue` with-parameters set these attributes of the new property. The `initialValue` provides an initial value for the property. If no inital value is given, the property is initialized to `False`.

`AddProperty` only adds a new property if the object does not already have a property (local or inherited) by that name.

When a property is added, all of its descendants are given that property. If an initial value is provided then all the descendants get that value. However, future changes are only propagated if the propagated attribute of the property is set.

Examples:

```
addProperty BeepSwitch, 'color'
addProperty BeepSwitch, 'kind' with propagatedValue with ¬
   initialValue 'wall'
```

RemoveProperty

```
removeProperty object, propertySymbol
```

The given property is removed from object only if it is a local property of that object.

Examples:

```
removeProperty BeepSwitch, 'color'
removeProperty BeepSwitch, 'kind'
```

Forms[*12]

Forms include executables (functions and functional-handlers) as well as with-forms (see below). Forms are the principle places where SK8Script commands are stored and from which the commands are executed. Functions are named sets of commands that have no special relationship to particular SK8 objects. Functions must have a unique name within a given project. Similarly, globally defined with-forms must have a unique name within a given project.

Functional-handlers and with-handlers, on the other hand, are always associated with a particular SK8 object that is considered to 'own' the handler. Such handlers can only be invoked using the object that owns it or a descendant of that object. It is common (and part of standard object oriented programming) for the descendants of an object to redefine or modify a handler defined by one of its ancestors.

With-Forms

With-forms are used in SK8Script to create 'contexts'. Contexts are most simply thought of as wrappers that go around a body of code. Contexts normally consist of some SK8Script that initializes the context and runs before the enclosed body of code, and another piece of SK8Script that cleans up the context and runs after the enclosed body of code. The key feature of contexts is that the clean up code is guaranteed to execute no matter how the enclosed body of code is terminated.

A simple example is the with-form '`Cursor`'. This context changes the screen cursor to some child of `CursorRSRC`. and then restores it to its previous state when the context is exited.

Example:

```
on click of me (a SpecialRect)
   with cursor WatchCursor
      repeat with i in the knownchildren of rgbcolor
         set my fillcolor to i
      end repeat
   end with
end click
```

The above object-handler causes the standard watch cursor to appear while the color of the invoking object is sequentially set to each of the possible solid color renderers.

Header Line Syntax

All types of forms maybe created by a SK8 programmer. Handlers are distinguished from other forms both by the way they are created and by the syntax of the first line of their definition (the header line). For functions and globally defined with-forms the header line has the syntax:

on [(**set|with**)] *formName* [*parameterList*]

Object-handlers have the following syntax

on [(**set|with**)] *handlerName* **of me (a** *objectType***)** [*parameterList*]

Technically, the optional `set` or `with` are part of the form name, but they are separated out since they are used to defined specific types of forms, setters and with-forms, respectively.

Parameter List

SK8 supports required positional parameters, optional positional parameters and position-independent keyword parameters for all forms.

Required Positional Parameters

Required parameters are regular positional parameters that must always be provided in order to execute a form. They are variable names separated by commas.

Examples:

Header lines with required parameters:

```
on myFunc X, Y
on keydown of me (a SpecialRect), TheChar
```

Using required parameters:

```
myFunc 100, {1,2,3}
keydown of SomeSpecialRect, the character "a"
```

Optional Positional Parameters

Following any required parameters a set of optional parameters may be given. The optional parameters are enclosed in square braces, `[]`, and separated by commas. Any number of optional parameters may be given, but they are always positional. This means that in order to provide a value for a given parameter, any previous optional parameters must also be provided. If an optional parameter is not provided, it defaults to the value `False`.

Example:

Header lines with optional parameters:

```
on myFunc1 [X]
on myFunc2 Y[, X] -- Note the comma is inside the braces
on mumble of me (a SpecialRect), X[, Y, Z]
```

Using optional parameters:

```
myFunc1
myFunc1 100
myFunc2 100
myFunc2 100,200
```

```
mumble of SomeSpecialRect, 100 -- X = 100, Y = False, Z = False
mumble of SomeSpecialRect, 100, 200 -- As above except Y = 200
mumble of SomeSpecialRect, 100, 200, 300 -- Now Z = 300
```

‘With’-Parameters

Instead of optional parameters the more general mechanism of ‘with-parameters’ may be used. With-parameters are also optional, but they are not positional. With-parameters are chosen using a unique ‘keyword’ associated with each possible parameter. This means that the user may provide values for only the with-parameters they desire. In a header line, with-parameters are usually specified with the word ‘`with`’ followed by the keyword and the variable name to be used in the body. However, there are also a set of special words that automatically create with-parameters with those names. The complete syntax for with-parameters is:

((**with** *keyword* [*name*]|*specialName name*) [*typeOrDefault*])*

Where a *specialName* is one of the reserved words

(**by**|**for**|**from**|**in**|**into**|**on**|**to**|**thru**|**through**)

and *typeOrDefault* has the form

([(**a**|**an**)*type*][**defaulting to** *value*]**)**

When a handler is called and a with-parameter is not specified it is given the value `False` unless a different default has been given using the `defaulting to` syntax. See the section on Data Types for details on the optional type specifier. If the `name` parameter component is left out, it defaults to the `keyword`.

For providing values for with parameters there is an addition syntax [*13]

keyword : *value*

Examples:

Header lines with keyword parameters:

```
on myFunc3 with border b
on myFunc4 X from start to end with border b
on myFunc5 with border (a Number defaulting to 0)
```

Using keyword parameters:

```
myFunc3 -- b = False

myFunc3 with border 100 -- b = 100

myFunc4 100 -- start = False, end = False, b = False

myFunc4 100 with border 100 -- start = False, end = False, b = 100

myFunc4 100 from 10 to 20 border: 100 -- start = 10, end = 10
-- b = 100
```

```
myFunc5

myFunc5 border: 100
```

Results of Executables

By default executables, functional-handlers and functions, return the value `False` when they are invoked as part of an expression. If another value is desired then the '`return`' command must be used. Note that with-forms have no return value. The syntax is:

`return` *`expression`*

The value of *`expression`* is returned as the value of the executable. The return command also terminates the execution of its enclosing executable. When a return command is executed the value of the expression is always computed regardless of whether that value is by the caller used or not.

Example:

```
on bark
   beep
   return 400
   beep
end bark
```

Invoking this function will always beep once and return 400.

Handlers

There is a special command and a couple special variables that are used in only in handlers.

Calling Parent's Handler

By default creating a handler for some object with the same name as a handler defined in one of that object's ancestors completely redefines the actions taken when the handler is invoked on that object or one of its children. However, the handler defined by the 'nearest' ancestor may be invoked from within a child's handler with the command:

`do inherited` `[`**`(`***`object positionalParams`***`)`***`withParams`*`]`

With no arguments `t`his command causes the ancestor's handler to be called with the same objects as were given to the containing handler. If any new arguments are given then an entirely new argument list is created. The *`object`* argument is almost always `me`.

Examples:

```
do inherited
do inherited (me, X, Y) with border 100
```

If an object has more than one parent with a definition of the handler or which inherits such a definition, then the handler associated with parent nearest the beginning of the `parents` list is called. This rule extends to more distant ancestors in the obvious way.

Calling ‘`do inherited`’ is particularly important for ‘setters’ since otherwise the value of the property will not be modified. It is also often important for various event handlers in order to maintain the expected behavior.

Special Handler Variables

Within a handler the variables ‘`me`’ and ‘`myself`’ are set to the object used to invoke the handler. In addition, the word ‘`my`’ may be used to refer to the invoking object.

Examples:

```
on click of me (a SpecialRect)
   -- All three of these lines modify a property of
   -- the invoking object
   set fillcolor of me to blue
   set left of myself to 100
   set my right to 200
end click
```

Defining With-Forms

There is a special command and a special marker that are used in with-forms but not in executables. Conceptually there are four sequential components to the execution of a with-form: the initialization code, the contained body of code, unnecessary follow-up code and necessary clean-up code. None of these components is required to create a valid with-form, but in practice most with-forms include all the components except the unnecessary follow-up code. Since the contained body of code varies between uses of the with-form, this component is represented by the command ‘`do body`’ in the definition. The necessary clean-up code is the only component that is guaranteed to execute once the form has been entered. In particular, if a condition is raised while executing the body, the unnecessary follow-up code will be skipped and execution will resume with the clean-up code. The unnecessary follow-up code and the necessary clean-up code are separated by the reserved keyword ‘`cleanup:`’.

Example:

```
on with beep
   beep
   do body
   beep
   cleanup:
      beep
end with beep
```

The following code will beep four times:

```
with beep
   beep
end with
```

However, if we insert an `AbortCondition` into the body it only beeps twice once during initialization and once during clean-up.

```
with beep
   abort condition
   beep
end with
```

The above example demonstrates how with-forms work, but it is not a good example of the reasons for using with-forms. The following with-form enforces a proper ice-cream protocol. In particular, it guarantees that the ice-cream always gets put away. Note that `PutTheIceCreamAway` should not assume that the ice cream was ever successfully taken out.

```
on with IceCream
   GetOutTheIceCream()
   do body
   cleanup:
      PutTheIceCreamAway()
end with IceCream
```

Invoking Forms

The syntaxes for invoking executables are:

[**set**] *name* [(**of** *positionalParams*|**(***positionalParams***)**)] [*withParams*]
[**set**] *firstPositionalParam***'s** *name* [**(***positionalParams***)**] ¬
   [*withParams*]
[**set**] *firstPositionalParam* **is** *name* [**(***positionalParams***)**] ¬
   [*withParams*]

The last form is intended for use with executables that are used as boolean tests. However, any excutable may be called using this syntax. When unambiguous the parentheses around the positional parameters may be left off. Note that the parentheses never go around the with-parameters. Again the optional `set` is used to invoke setter-handlers. Setter-handlers always have a required special with-parameter '`to`'.

For with-forms the syntax is:

with *name* [(**of** *positionalParams*|**(***positionalParams***)**)] [*withParams*]
   body
end with

For all handlers the first positional parameter must be an object which owns or inherits a correctly named handler. The positional parameters are expressions separated by commas. The with-parameters are generally given as the word `with` followed by a valid keyword and an expression for the value. This pattern is repeated for addtional with-parameters. Certain with-parameters (see subsection on with-parameters above) do not require the presence of the word `with`. For other with-parameters, the value may be left off in which case the value for that with-parameter is `True`. Similarly, the word `without` may be given followed by a keyword. In this case the value is `False`.

Examples:

```
squareRoot of 4
squareRoot(4)
squareRoot 4
4’s squareRoot
set SpecialRect’s fillcolor to blue
set fillColor of SpecialRect to blue
set left of SpecialRect to 10 without resizing
SomeObject’s someHandler (100, 200) from 4 to 10 with border 300
```

Data Types

In most situations SK8 relieves the programmer of the need to be explicitly aware of the data types of the values used in the language. However, in some cases the programmer may wish to have control over the automatic type conversions or to provide additional explicit type information to improve error detection or to increase the efficiency of the code.

Advantages of Type Declarations

With type declarations the SK8 compiler may:

- perform type-checking during compilation as a debugging aid.
- invoke specialized runtime type checking enabling early detection of assignment errors -- before they manifest into more obscure bugs.
- use type information to create more optimized code.

In addition, type declarations can improve code readability and maintainability by clarifying the programmer’s intent for the possible values of a variable.

Note

Early versions of the SK8 compiler may **not** take full advantage of type declarations to produce efficient code or even to guarantee the contents of a particular location. However, the explicit type checking mechanisms described below are supported. ◆

Associating a Type with a Variable

Anywhere a variable is declared, the intended type for the variable may be given. The syntax is:

varName **(**(**a**|**an**) *type***)**

Examples:

```
local X (a Number)

on mumble Str (a String) from start (a Number defaulting to 1) ¬
   to end (a Number defaulting to Str's length)

on set border of me (a SpecialRect), b (an Integer)
```

Type Coercion

Coercion Operator

The `... as a ...` and `... as an...` operators allow you to coerce one type of object into another type. The most frequently desired types of coercion are supported. You can write your own coercion functions for ones that are not supported.

[*11]Note
An enumeration of the supported coercions and instructions on how you can write your coercion functions will be available in a later release of this manual. ◆

▲ **WARNING**
SK8Script does not support complete automatic coercion in the same way as HyperCard and other systems. Restricting the allowed kinds of automatic coercion makes your code easier to understand and maintain by others. In general, if you need an object to be interpreted as a certain type, you should explicitly coerce it to that type; e.g. `10 equals "10" returns False`. ▲

Example:

The following expressions evaluate to `True`.

```
("10" as a Number) = 10
(10 as a String) = "10"
```

General Object Literals

Coercions using the coercion operator are always reevaluated each time the expression is evaluated. If the same particular object is always expected, the more efficient mechanism of general object literals may be used. The most common use of general object literals are to create objects that can intuitively be defined by more easily read objects such as strings, e.g. Dates. The syntax for a general object literal is:

`the` *`desiredParentObject valueToCoerce`*

Examples:

```
the Number "10"
the String 10
the Date "January 15, 1994"
the Date 2841436800
the Number the Date "January 15, 1994" -- Returns 2841436800
```

User defined objects can take advantage of this mechanism as long as the appropriate coercion functions are defined.

SK8 will reevaluate a general object literal if it cannot guarantee that the given expression is constant. However, SK8 does **not** consider a non-constant value an error. A good example of this feature is converting a plural selection expression into a list.

Example:

```
length of the list (every rectangle in the stage)
```

Note that the statement '`length of every rectangle in the stage`', would attempt to call the function `length` on each rectangle on the stage in turn which is invalid.

Virtual & Enumerated Types

Virtual types are a SK8Script mechanism used for extending the SK8 type system beyond the standard inheritance type system. For example, the concept of a positive integer is available through the standard virtual type `PositiveInteger`.

Example:

```
3 is a PositiveInteger -- Returns true
-3 is a PositiveInteger -- Returns false
```

Virtual types are all descendants of the `VirtualType` object. They are implemented using through the `typeSatisfied` handler. This handler is expected to take an arbitrary SK8 object and return `False` unless the object should be consider to be of the given virtual type.

Example:

```
new VirtualType with objectName "OddInteger"

on typeSatisfied of me (an OddInteger), obj
   return obj is an Integer and obj is odd
end typeSatisfied

1 is an OddInteger -- Returns True
42 is an OddInteger -- Returns False
```

`EnumeratedType` is a predefined virtual type that is used for representing arbitrary discrete collections of options. The possible values are stored in the `options` property of a child of `EnumeratedType`.

Example:

```
new EnumeratedType with objectName "GeomPrimitive"
set options of GeomPrimitive to {Rectangle, Oval, RoundRect, ¬
   Polygon, Line}

Rectangle is a GeomPrimitive -- Returns True
```

```
new Rectangle with objectName "Rect1"

Rect1 is a GeomPrimitive -- Returns False
```

SK8Script Condition System

SK8Script provides an easy to use, fully object-oriented, extensible condition system. The condition system is comprised of condition objects, condition handlers and condition response objects.

Condition objects are created when a condition is detected and are intended to store all of the information needed to correctly respond to the condition. Condition handlers are defined within a function or handler and are the first line of response to a condition. Finally, condition response objects are special objects that know how to respond to some set of conditions. Condition response objects are normally invoked in response to a condition object for which no condition handler has been defined in any of the currently active functions or handlers. A table associating condition response objects with condition objects is stored in the `conditionResponses` property of every project.

Condition Object

A condition object is any descendant of the object `Condition`. In order for the condition to be invokable from SK8Script, the name of the object must end with either '`Error`' or '`Condition`' and must have some prefix, e.g. `GeneralError` or `AbortCondition`. Condition object often store useful information in local or inherited properties. For example, the `GeneralError` object stores a list of objects and strings that are used to contruct an error message.

Signaling a Condition

Conditions are signalled or 'raised' with a special condition command. The syntax of a condition command is

`conditionType` (**`error|condition`**) *`KeywordArgumentList`*

The condition type is the prefix of the name for some condition. The reserved word `error` or `condition` must correspond to the suffix of the desired condition. When a condition command is executed the containing executable [*14] [form] is checked for a matching condition handler. If one is found, then it is invoked. If none is found, the condition is treated as though it were created in the executable [form] that called the previous function or handler. This process is repeated until a matching condition handler is found. If the top level SK8 event loop is reached, then the `conditionResponses` of any relevant projects are checked. The exact order in which the projects are search in this case is discussed in the Condition Response subsection below. In any case, the SK8 project is always checked eventually and since its `conditionResponses` includes an entry for the `Condition` object itself, all conditions are ultimately handled in some way.

Example:

The following command creates a `GeneralError` which by default will get reported in the message box if it is executed from there.

```
general error with strings {"We encountered some problems!"}
```

SK8 provides a set of common conditions which are documented in the Reference Manual.

Condition Handlers

Condition handlers are defined inside functions or handlers and catch condition objects when they are created with some condition command. The syntax for a condition handler is

```
on [conditionType] (condition|error)
   CommandList
end (condition|error)
```

The choice of the `condition` or `error` keyword is again determined by the name of the condition being raised. If no condition type is given then all conditions or errors are caught depending on which keyword is used. The `error` object is a descendent of the `condition` object, so all errors will be caught if all conditions are caught, but the reverse is not true.

Example:

```
new condition with objectName "DiskCondition"

on TestDisk
   if DiskIsBad() then Disk Condition

   on Disk Condition
      sendToLog "Time to buy another disk?"
   end Condition

   on error
      sendToLog "Some error happened"
   end error
end TestDisk
```

Condition Responses

Condition responses are objects which encapsulate common protocols for responding to conditions. For example the `LogErrorMessage` condition response calls the `writeObject` handler of a condition on the stream associated with the MessageBox.

Condition responses are used by installing them into the table stored in the `conditionResponse` property of a project. Empty tables are created automatically when a new project is created. Such tables are all children of the `TypeTable` object. `Table` objects are normally used to associate specific objects with other objects. A

TypeTable is a special type of Table object that returns the object associated with the closest ancestor of that object present in the table.

When a condition command is executed and no condition handler is found, the TypeTable of various associated projects are checked. Conceptually the projects 'closer' to the point of execution at which the condition command is executed are checked first and the top level project, sk8, is checked last. See the Search Algorithm sub-section below for more details.

Example:

We will restrict ourselves to the simple, though common, case where there are only two projects involved. The sk8 project and a user project, say play. If we create a new condition,

```
new condition with objectName "HappyCondition"
```

and then invoke it in the message box

```
happy condition
```

nothing seems happen. What actually happened is the condition response NoResponse was applied to the child of HappyCondition so the condition was resolved by taking no action. You may be familiar with this approach often taken by large bureaucracies.

To find out that this was what happened we can look up the responses the different projects have for HappyCondition.

```
get item HappyCondition of conditionResponses of play
get item HappyCondition of conditionResponses of sk8
```

The first of these returns False indicating no applicable condition response was found. The second, returns {NoResponse}. A list is returned because it is sometimes desirable to associate more than one condition response with a condition.

We can change this behavior by inserting a condition response into the conditionResponses of play

```
set item HappyCondition of conditionResponses of play to ¬
   {LogErrorMessage}
```

Now when we raise the condition an error message appears in the message box. By default the message simple prints out the name of the condition. This can be changed by providing a writeObject handler for the condition.

```
on writeObject of me (a HappyCondition), thestream, rereadably
   if rereadably then
      do inherited
   else
      writeObject  "I'm really happy", theStream, rereadably
   end if
end writeObject
```

Creating Condition Responses

New condition responses are created in the usual way as children of the conditionResponse object. Two handlers are used by the condition system.

The invoke handler actually implements the protocol.

The invokable handler ensures that this condition response knows how to handled the raised condition which is stored in the global variable currentCondition. If the invokable handler returns False for some condition, then the system will continue the search for a condition response. This is one of the reasons that a list of condition responses is associated with a condition rather than just a single condition response.

Example:

```
new conditionResponse with objectName "BeepResponse"

on invoke of me (a BeepResponse)
   beep
end invoke

-- invokable defaults to being True

set item HappyCondition of conditionResponses of play to ¬
   {BeepResponse}

happy condition -- Now causes a beep.
```

Search Algorithm

Each executable [form] on the call stack indicates a “chain” of required projects originating from the project to which the handler belongs.

If all the chains, when merged, make up a single chain then the condition response to invoke is determined by searching up this chain from its deep end for a project whose conditionResponses property has a match to the raised condition

Otherwise, if the chains are divergent, the “pivot project” is defined as the deepest intersection of all the chains, i.e. the project below which the chains diverge. When searching up a chain to find an applicable condition response, the traversal stops short of the pivot project (since a condition response found in or above the pivot project would be applicable in any project in any of the chains). The portion of a chain below the pivot project is called the “exclusive chain”.

The condition response to invoke is determined by searching up the stack for a frame whose project’s exclusive chain has a condition response applicable to the given condition. If the entire call stack is searched without finding an applicable condition response then, as a final resort, the chain from the pivot project up is searched. Since this chain always ends with SK8, which has a condition response for Condition, this final search is guaranteed to find an applicable response.

Other Features and Issues

This section discusses several issues that commonly arise for new SK8 users.

Syntactic Sugar

Observant SK8 programmers often notice that some of the words in SK8 do not have an obvious functional role in some SK8 commands, but appear to simply make the command more English-like. In some situations this is true and where possible SK8 does not require such 'syntactic sugar'. However, it is important not to over generalize and assume that just because a word is not needed in one command that it is always unnecessary.

Examples:

The following pairs of commands are equivalent:

```
get 10 is the same object as 5 + 5
get 10 is same as 5 + 5

get every item in the knownchildren of the rectangle
get every item in knownchildren of rectangle

get 10 as a string
get 10 as string

get whether x is = to y
get x = y
```

However the following 'commands' are not equivalent:

```
get the Date "January 10, 1994" -- valid general object literal
get Date "January 10, 1994" -- call to a function called 'Date'

get "10" is a list -- Is "10" a child of List? Value is False
"10" is list -- list("10"). Value is {"10"}

get every oval in contents of someRect
every oval in contents of someRect -- An expression not a command

get whether every item in {10,20,30} is < 20 -- False
get every item in {10,20,30,40} is < 20 -- {True, False, False}
```

in vs. of

It is sometimes confusing when to use the terms '`in`' and '`of`'. There are two different places this keyword gets used. One is in selection expressions. The other is the parameter list of a form. In selection expressions the two words are precisely equivalent.

Example:

```
(rectangle 1 of stage) is equal to (rectangle 1 in stage)
```

In parameter lists, on the other hand, '`of`' is one method for beginning the list of positional parameters, whereas '`in`' is one of the special with-parameters.

Example:

```
someFunc of 4,5 with border 300 in {10,20,30}
```

Message Box Results

When an expression is evaluated in the Message Box, SK8 attempts to provide a result that uniquely describes the same value. For objects that have literal forms, the literal is given. For named objects, the name is returned. If the object is unnamed, but can be described using its creation relations, an appropriate selection expression is built. If none of these expressions can be created, a brief description of the object will be given enclosed in square braces, `[]`. The square braces indicate an *unreadable form*. Unreadable forms are not valid SK8Script. Unreadable forms can actually appear any time the name of an object that is otherwise undescribable is requested.

Examples:

In the following examples the result as it would appear in the Message Box appears as a comment.

```
4 + 8 ≈ 12
new rectangle with objectName "SpecialRect" --SpecialRect
new rectangle with container Stage -- Rectangle 1 in Stage
new rectangle -- [a child of Rectangle]
```

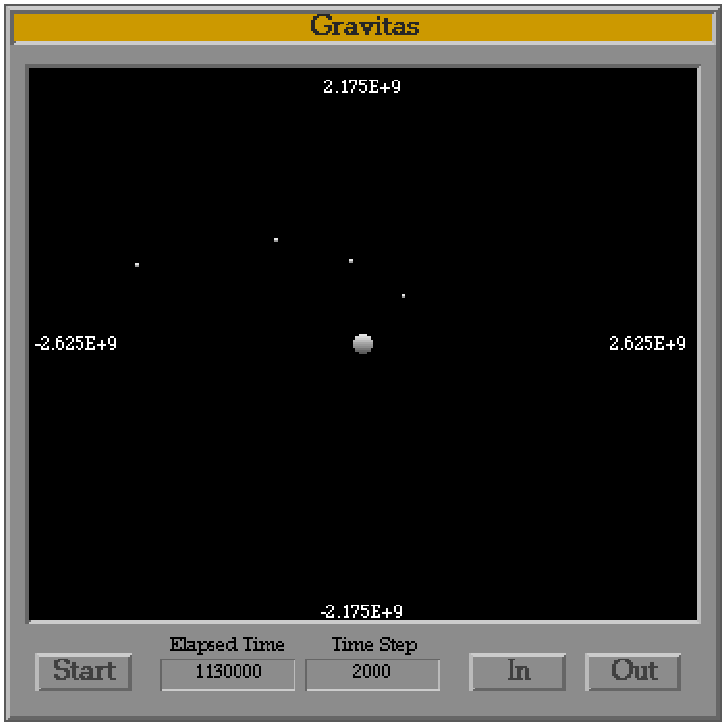
Tutorial 2: SK8 as a Meta-Tool

Introduction

This tutorial will demonstrate some of the features discussed in the earlier chapters which make SK8 such a powerful development environment. While following this tutorial, you may find it instructive to refer back to the “Basic Concepts” and “Project Builder Overview” chapters for more information. The aim of this tutorial will be to emphasise the use of SK8 as a *meta-tool*, that is, as a tool for creating other tools. Accordingly, the tutorial will be based around the creation of a tool kit which we will then use to build real applications.

The class of applications our tool set will allow us to build can be characterised as *dynamic simulations*. These are commonly used in science education and also in the field of scientific visualization. They bring together a mathematical model of a process and an animation engine which then allows the user to investigate the evolution of the process under a range of conditions.

Of course, our hope is that by following this tutorial you will learnsome *general* skills which they will be able to apply to your own problems. Dynamic simulations are just convenient examples that allow us to introduce some of the key concepts behind SK8 and its approach to object-oriented programming.



The figure shows the actual application we will build with our tool set. The picture shows four satellites orbiting a planet. In fact, the objects represented are Jupiter and its four largest moons, but in principle they could be *any* astronomical system - the program works out orbits from initial conditions and Newton's Laws of Motion, not from preset data.

In this tutorial we will be using SK8 to build a set of tools that, in turn, make the construction of such programs relatively straightforward.

Aims and Prerequisites

The broad aims of this tutorial are to:

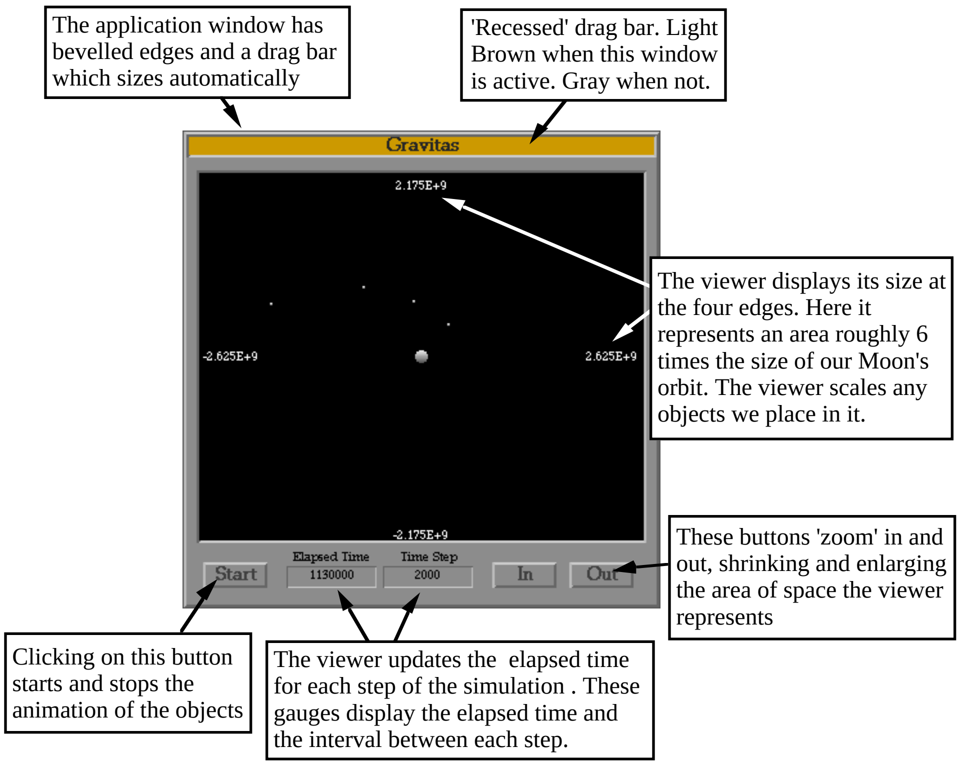
- Demonstrate SK8 being used to create a set of tools. In other words, to show SK8's power to be a meta tool.
- Illustrate a typical extended user interaction with SK8.
- Raise some of the issues arising from Object-Oriented Programming in SK8.
- Show that SK8's support for both Direct Manipulation *and* Command Line styles of interaction creates a powerful synergy.
- Point at further directions in which you may like to explore SK8 programming.

To follow this tutorial you should have some experience of scripting languages such as HyperTalk or AppleScript and you should also know a few things about SK8. In particular you should be familiar with:

- The SK8 Project Builder and its major components: the Message Box, the Object Editor, the Drawing Tools and the Project Overviewer.
- The fact that SK8 has its own scripting language, SK8Script.
- The basic Object-Oriented structure of a SK8 program: All parts (including visible parts like windows and controls) of an application are *objects* which can send and receive *events* generated by other objects, the user or the system.
- The way objects respond to events by executing scripts called *handlers*, which may be their own or which they may have *inherited* from other objects.
- The fact that objects can have their own data items, known in SK8 as *properties*, and that like handlers. these may also be their own or be inherited.
- The way SK8 compartmentalizes the user’s work in *projects* which parcel groups of objects together in a protected name space.

The Goal

Before starting we’ll take a slightly more detailed look at the simulation we will build with the finished tool kit.



The application window has bevelled edges and a drag bar which sizes automatically

'Recessed' drag bar. Light Brown when this window is active. Gray when not.

The viewer displays its size at the four edges. Here it represents an area roughly 6 times the size of our Moon's orbit. The viewer scales any objects we place in it.

These buttons 'zoom' in and out, shrinking and enlarging the area of space the viewer represents

Clicking on this button starts and stops the animation of the objects

The viewer updates the elapsed time for each step of the simulation . These gauges display the elapsed time and the interval between each step.

This program is a simplified version of Gravitas (Sellman, 1992), a dynamic simulation designed to let students investigate the motion of gravitating bodies. Users can drop new planets into the viewer, adjust their velocities and masses and watch their trajectories evolve. The program computes the paths of the bodies using Newton's Laws of Motion and Gravity. The interesting point in the context of SK8 is that Gravitas is an example of a *class* of possible simulations, all with a similar structure, based on the animation of interesting objects. These simulations need a place to display the objects, a method for generating the next step in the animation, controls for driving the animation and setting object properties, and gauges to display the properties.

Gravitas deals with planets, but the central objects could just as well be charged particles, or molecules, or organisms of some kind. We would have to change the detailed working of the viewer, the controls and the animation method, but the overall structure would stay pretty much the same. With this point in mind we will begin work on our tool kit.

The SimKit Window Tool

The first thing we will construct is the window tool for our kit. After you have loaded SK8 a dialog box will appear which will ask you to open an existing project or create a new one. We will start with a new project so type SimKit (from **Sim**ulation Tool **Kit**) into the text box and click the New button. SK8 creates a new project object and names it SimKit. After loading SK8 and creating the new project you should have four windows on the screen: at top left the Draw Tools, at top right the SimKit Object Editor. At bottom left you should have the SimKit Overviewer and at bottom right the SimKit Message Box.

Creating a Window

In SK8 a window is simply a rectangle, so choose the Rectangle from the Draw Tools and drag out a rectangle a few inches across on the screen. The notion of *graphical containment* is at the heart of SK8 and we must now consider what this concept means for the rectangle you have just drawn. It doesn't actually appear to be contained by anything. However, SK8 treats the entire screen as a container called the stage, and the new rectangle is then said to be contained by the stage. In turn, it will become the container of other things we will be building.

This new window is rather plain - a white interior with a black frame. We could now choose to set its windowstyle property to make it look like one of the standard Macintosh windows. However, to show that SK8 is not tied to a particular desktop 'look and feel' we will be creating a simple one of our own.

Naming an Object

If you haven't clicked the mouse anywhere else the new window should still be selected, if not, select it now (use the selection tool from the Draw Tools palette) so that the *selection halo* is surrounding it. Use the halo's pop up menu and select Name... Type SKWindow into the dialog box which appears.

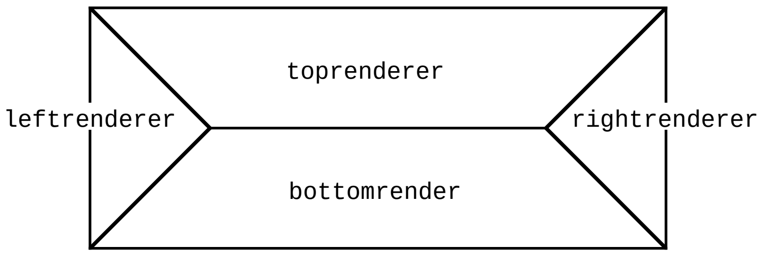
Editing the SKWindow Object

If you hold the mouse button down on the name in SKWindow's halo it will highlight. You can drag the highlighted name into the Object Editor (actually, into the text box at the top of the editor) which will fill up with the new window's properties. Now, in the Object Editor, find the window's fillcolor property and set it to gray. SK8 defines a handy range of grays between white and black and we'll use one of medium density called GrayTone50. Double click on the fillcolor property to edit it and type in GrayTone50.

Regions and Renderers

We still have a plain looking window but for our new look and feel we are going to produce something with a simple 3-D appearance. To do this we have to take a quick look at SK8's imaging methods. First of all, it is important to understand that a graphic object in SK8 defines one or more regions. The rectangle we just created defines two: its fill region and its frame region (the black border). In most applications setting the color of a graphic object is just a matter of setting all the pixels contained by that object to the desired color. In SK8 we must think about the process a little differently. Setting a SK8 object's fillcolor is best thought of as assigning the job of painting its fill region to a special new object called a renderer.

Renderers come in many forms but what they all have in common is the ability take a region of a graphic object and act on the pixels inside it. They can paint the region with a solid color, or a pattern, a picture, a transparent color, or even a complex predefined sketch. For our present purposes we are going to use an object called a bevel renderer, which, as its name implies, makes a region look beveled.



As the diagram shows, when acting on a rectangle a bevel renderer actually views the fill region as four sub-regions and applies a renderer to each. By choosing the right shades we can make the bevel appear to be illuminated from any angle and to be recessed or raised. We'll now build a couple of useful bevel renderers of our own.

Go to the SimKit Project Overviewer and press the New button. A dialog box will appear, giving you the chance to create and name a new object. Type BevelRenderer into the Object field and SKBevelIn in the ObjectName field. Press the Create Button and your new object will appear in the Project Overviewer.

SKBevelIn's default shading is unsuitable for our purposes but you can double click on it in the Project Overviewer to bring up a specialized BevelRenderer editor. This will allow us to set the colors for each region. Click on the color tab of the Left Renderer. A palette will appear showing, among other colors, a range of grays. Find the gray called

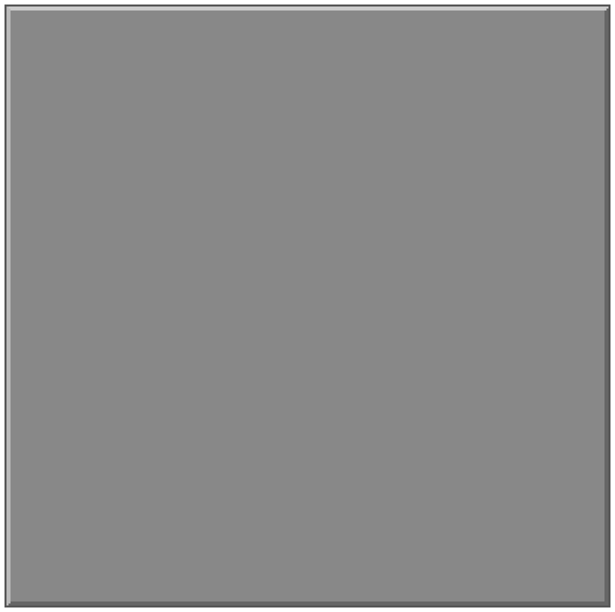
GrayTone40 and click on OK. The Left Renderer of SKBevelIn is now set to this shade. Carry on in the same way, setting the Top Renderer also to GrayTone40, the Right Renderer to GrayTone70 and the Bottom Renderer to GrayTone80. That completes SKBevelIn.

Our second renderer is going to have the opposite appearance so we will call it SKBevelOut. Return to the Overviewer and press the New button again. This time type BevelRenderer into the Object field and SKBevelOut into the ObjectName field. When you press the Create button SKBevelOut will appear in the Overviewer where you can double click on it to bring it into the Bevel Renderer editor.

In the same way as before set the Left Renderer of SKBevelOut to GrayTone70, the Top to GrayTone80, the Right and the Bottom to GrayTone40. You can put away the Bevel Renderer editor now, we are finished with it.

Incidentally, note that no new objects corresponding to SKBevelIn or SKBevelOut have actually appeared on the screen. This is because renderers are not themselves graphic objects. We only see them in the Overviewer.

Now we can *use* our bevel objects but we won't be assigning them to the SKWindow's fill region. Instead we will use one of them on the window's frame. Return to the Object Editor where SKWindow should still be the selected object with its properties and their values displayed. Find and double click on the framecolor property. Type in SKBevelOut and press the Set button. We also need to enlarge SKWindow's frame to see the effect. Double click on the framesize property and type in {2,2} and press the Set button. Framesize has two components - width and height - hence the list of numbers.



This is what you should have on the screen. Now we'll add a drag bar, so that we can move the window around the screen. Our drag bar is going to be another rectangle, so select the rectangle tool and drag out a long thin rectangle *inside* of SKWindow.

Tagging Component Objects

The shape of the new rectangle doesn't matter now as we will be setting it precisely in a moment. Using the menu on the new rectangle's halo, select Name... and type in SKDragBar. Next, choose the Tag... item on the halo menu. A dialog box will appear asking you to "Tag" SKDragBar as a named part of its *container*, SKWindow. Simply type in dragbar. Tags establish a relationship between a complex object and its components and SKDragBar is going to be a component of SKWindow.

Now we can refer to SKDragBar in two ways - by its name or by its tag. Thus the two statements:

```
set the framecolor of SKDragBar to SKBevelIn
set the framecolor of SKWindow's dragbar to SKBevelIn
```

are equivalent. Type either one of them into the Message Box to set the dragbar's framecolor. Of course, you could also have done this in the Object Editor but sometimes the Message Box is handy for a one off task.

We'll explain the rationale behind tagging soon but first we'll put some functionality into SKDragBar and SKWindow. We can do this by defining some handlers on the objects.

Defining Handlers

First we'll add a handler to SKDragBar. It should still be selected so just use the halo menu again and select the New Handler... item. A dialog box will appear asking you to name your new handler. A handy pop-up menu allows you to rapidly select one of SK8's standard events to handle. Use this and choose the mouseDown item. A *handler editor* will now appear with the template of our new handler already installed and the cursor positioned at the right place. Type a new line in so that the completed handler reads:

```
on mouseDown of me (a SKDragBar)
   drag my container with live
end mouseDown
```

and pull down the Version menu. Choose the Activate Current Version item. If you mouse down on the drag bar now you will find that it does indeed drag its container, SKWindow, around the screen. The with live statement is what makes the movement of the window immediate - if we had left it out a frame would be dragged and SKWindow would snap to the new position upon release of the mouse.

The SKDragBar however, is not really in the appropriate place. We have to do something about that. Again, a handler is the answer but this time we'll define it on SKWindow so that the SKDragBar is correctly positioned whenever the window is resized. One of SK8's standard events is generated when something gets resized so we simply have to supply a corresponding handler. Select SKWindow (Use Selection from the Draw Tools and click the mouse on SKWindow) and choose New Handler... from the halo menu. From the pop-up menu in the dialog box choose the resized item and a handler editor will appear as before but with a template for the new event. Type in the following SK8Script:

```
on resized of me (a SKWindow)
   if my dragbar is an object then
      set {dx, dy} to my framesize
      set my dragbar’s boundsrect to {dx,dy,my width-dx,dy+18}
   end if
end resized
```

There are several features in this handler we should explain but first give it a try. Select SKWindow again and use the halo’s selection dots to resize it. You should see SKDragBar adjust itself to the correct size also.

An Explanation of the Resized Handler

Looking back at the handler, the first thing to note is the syntax of the opening line: `on resized of me (a SKWindow)`. The code is saying that this handler deals with resized events for objects of the type SKWindow. At the moment we only have one object of this type, the original itself, but we will soon be creating copies which will *inherit* this same handler. The me in the definition carries the identity of the particular object receiving the event. Incidentally, the resized event is sent by SK8 to an object *just after* its size has been changed.

The next line is a check to see that the dragbar of this particular SKWindow actually exists (we might delete it for some reason in which case our handler would crash without this line).

We then use a couple of *local variables* (that is, variables which only exist briefly, while the handler is executing) to hold the SKWindow’s framesize. As you can see, SK8Script allows multiple assignments in one line such that:

```
set {name, height} to {“Lhotse”, 8516}
```

is equivalent to:

```
set name to “Lhotse”
set height to 8516
```

The local variables are then used to set the boundsrect property of the SKWindow’s dragbar, that is the coordinates of the dragbar’s bounding rectangle. The coordinate system for a graphic object is based on the top left corner of its container: the stage for SKWindow, and SKWindow for SKDragBar. The boundsrect property has four components: {left,top,right,bottom}. Thus:

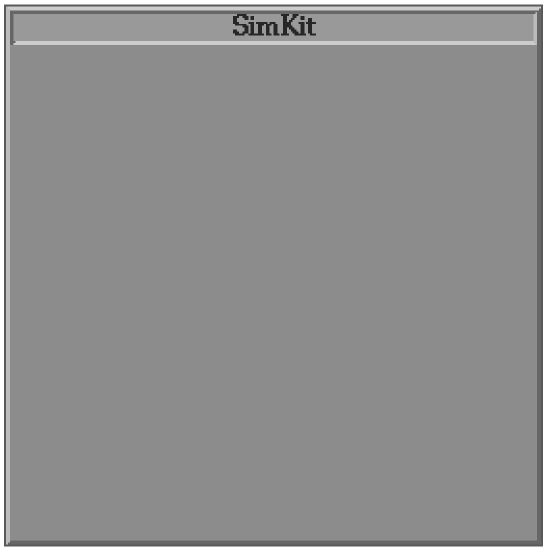
```
set my dragbar’s boundsrect to {dx,dy,my width-dx,dy+18}
```

places SKDragBar just inside SKWindow’s frame, with height 18 pixels. By the way, notice the use of the apostrophe to form the possessive in this line of code. It is a shorter equivalent to:

```
set the boundsrect of my dragbar to {dx,dy,my width-dx,dy+18}
```

To improve SKWindow’s look we’ll set a few more of SKDragBar’s properties to new values. As we are going to make a few changes, the Object Editor is a good place to do this. Select SKDragBar and drag its name bar into the Object Editor. In the usual way, set its fillcolor to GrayTone60, its text to “SimKit” and textfont to “Palatino”, its framesize to {2,2}, its textsize to 14, its textstyle to {'bold'} and finally its textcolor to GrayTone15.

You should now have a window which looks something like this:



Highlighting the Active Window

The last two handlers we will add to SKWindow will make it highlight itself when it is the active window. This feature will complete our rudimentary new look and feel. Again, we’ll use standard SK8 events, in this case activate and deactivate, which windows receive when they are clicked on or when another window is clicked on. As before, choose New Handler... from the halo menu and this time choose the activate item the pop-up menu in the dialog box and type the following SK8Script into the handler editor:

```
on activate of me (a SKWindow)
   if my dragbar is an object then
      set the fillcolor of my dragbar to LightBrown
   end if
end activate
```

Pull down the Version menu in the handler editor and choose Activate Current Version to save the handler. Now pull down the Edit menu and choose New Handler... When the dialog box appears choose the deactivate preset handler and type in:

```
on deactivate of me (a SKWindow)
   if my dragbar is an object then
      set the fillcolor of my dragbar to GrayTone60
   end if
end deactivate
```

Pull down the Version menu in the handler editor and choose Activate Current Version. Now when you click on SKWindow its dragbar will change color to indicate that it is the selected window.

Creating the SKWindow Tool

We have a nice looking window now but how would we go about producing more than one of them? We could type something like this:

```
new SKWindow with container stage
```

and a new object, a perfect copy of SKWindow, would appear on the screen. However, SK8 gives us a very simple way to turn any graphic object into a Direct Manipulation tool. Select SKWindow and drag its name bar into the top part of the Draw Tools window, by the Selection Tool. The new tool will appear, alongside a miniature representation of itself.

It doesn't make much sense to have the original SKWindow on the screen now. It has become our *prototype* for the creation of others like it. To hide it we simply remove it from the Stage. Type the following into the Message Box:

```
set SKWindow's container to false
```

SKWindow will disappear from the screen but you can select the new tool and draw a few SKWindows out on the stage. Give it a try. Actually, in Object-Oriented terminology, the objects you are drawing are called *children* of SKWindow and we say that the children *inherit* their properties and handlers from their *parent*, SKWindow.

Draw three or four child SKWindows out on the stage and click on them in turn. You'll see our simple highlighting mechanism working.

Tags versus Named Objects

Now we should spend a moment to reflect on a detail of our programming that made it possible for us to turn SKWindow into a tool. First of all, SK8 insists that all objects in a project have different names. This is sensible, as SK8 would find it difficult to distinguish between two objects with the same name. But every *child* of SKWindow needs to be able to refer to their copy of SKDragBar by some reliable means. This is why we *tagged* SKDragBar to be the dragbar of SKWindow. It allowed us to refer to my dragbar in each of SKWindow's three handlers almost as if it was a property. In each of the children of SKWindow we just created, the expression "my dragbar" is resolved to mean the child's local copy of SKDragBar. Imagine that we had typed the following handlers instead (the changes are in bold):

```
on resized of me (a SKWindow)
   if SKDragBar is an object then
      set {dx, dy} to my framesize
      set the boundsrect of SKDragBar to {dx,dy,my width-dx,dy+18}
   end if
end resized
```

```
on activate of me (a SKWindow)
   if SKDragBar then
      set the fillcolor of SKDragBar to LightBrown
   end if
end activate

on deactivate of me (a SKWindow)
   if SKDragBar then
      set the fillcolor of SKDragBar to GrayTone60
   end if
end deactivate
```

The explicit references to SKDragBar would work fine for SKWindow itself because its dragbar really is the object SKDragBar. But, our copies of SKWindow can't have dragbars called SKDragBar as well. In fact, their dragbar's don't have names at all. If we had typed in the above text, resizing one of the children of SKWindow would lead to SKDragBar being resized, not its own dragbar. The tags though, allow us to give an unambiguous *relative* reference to an object, which will work for all SKWindow's children.

Properties versus Named Objects

The second feature of our programming which allowed us to turn SKWindow into a tool was used in our very first handler:

```
on mouseDown of me (a SKDragBar)
   drag my container with live
end mouseDown
```

The key phrase here is `my container`. The handler

```
on mouseDown of me (a SKDragBar)
   drag SKWindow with live
end mouseDown
```

would have failed for SKWindow's children just like the other handlers. The symbol container is not a tag but a property. The reference avoids using an explicit object name by using an existing property.

Using tags and properties in this way allows us to build objects that we can easily turn into tools. In the common terminology of Object-Oriented Programming, it allows us to do *sub-classing*, that is, create new *instances* of objects from *class prototypes*.

Now we move on to the construction of our next tool.

The SimKit Viewer Tool

The next tool we will build is a viewer for the objects at the center of the simulations we intend to construct with our kit. The viewer will display the objects in a window which can represent an arbitrarily large region of space and it will scale the objects correctly to this space.

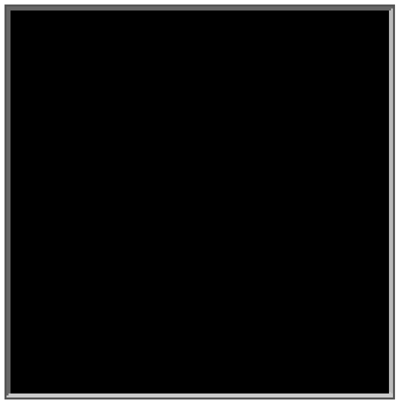
If you have been following the tutorial up to this point you will have three or four SKWindows on your screen. Get rid of them for the time being by selecting them and choosing Clear References. SK8 automatically reclaims the memory used up by unreferenced objects so this is equivalent to deleting them. Pressing the delete key when an object is selected accomplishes the same thing.

Using the Object Editor

Now, use the rectangle tool and draw out a three or four inch square on the screen. Drag the new object's name box into the top field of the SimKit Object Editor. The editor will fill up with the square's properties. Incidentally, this "drag and drop" method is the Direct Manipulation equivalent of the Edit item on the halo menu. SK8 usually offers more than one way to carry out an operation and you can choose the method you prefer.

Now name the new object: scroll the properties list down until you can see the objectname property. Double click on it and type in SKViewer and press return. This is equivalent to the halo's Name... menu item that we used before.

Next, scroll the property list to fillcolor, double click as before and type Black. In the same way set SKViewer's framesize to {2,2}, and its framecolor to SKBevelIn. You should have something like this:



The viewer is black so that it will resemble outer space when it contains the planets we will be building later.

Adding New Properties to Objects

So far in this tutorial, we have simply changed the *values* of properties (like framecolor and boundsrect) that came built in with our newly hatched objects. We are now going to add *new* properties to an object, SKViewer, to help us give it more functionality.

The Object Editor has a menu called Properties. Pull this down and choose the New Property... item. A dialog box will appear allowing you to name the new property. Type metersperpixel. Now repeat the process and add two more properties: timestep and elapsedtime. We will see the purpose of these new properties later. For now we will just set their values. In the same way you edited the values of the built in properties framecolor and so on, find the new properties in the Object Editor’s list and set their values. Set metersperpixel to 1, timestep also to 1, and elapsedtime to 0.

We could have done all this work on SKViewer from the halo menu and the message box. However, it was useful practice as the Object Editor is convenient and quicker if you know you are going to do several things to an object in quick succession.

Labels

The new property, metersperpixel, that we have added to SKViewer will be used to define the size of the space that SKViewer represents: Its width and height in meters will be its width and height in pixels times its metersperpixel value. It would be handy if we could somehow display the size of the space at all times. To do this we’ll use a new kind of object, called a *label*.

We will need four labels, one each for the x and y maxima and minima of our space. We’ll start this work in the Message Box and illustrate a couple more features of SK8Script along the way. Type:

```
new label with objectname "SKXMin" with container SKViewer
tagpart SKViewer, SKXMin, ‘xmin’
```

Previously we created new objects either inthe Overviewer or by using one of the Draw Tools. Here we are using a third method - a command from the Message Box. We have created a new label and set *two* of its properties in the creation line using the “with” statement. In fact, you can set as many properties as you like this way. Secondly, we have tagged SKXMin as the xmin of SKViewer (Note the order of the arguments to tagpart). The other labels are created in the same way. Type:

```
new label with objectname "SKXMax" with container SKViewer
tagpart SKViewer, SKXMax, ‘xmax’

new label with objectname "SKYMin" with container SKViewer
tagpart SKViewer, SKYMin, ‘ymin’

new label with objectname "SKYMax" with container SKViewer
tagpart SKViewer, SKYMax, ‘ymax’
```

At the moment you won’t see any new objects in SKViewer because the default textcolor for labels is Black! We can soon fix that, and learn a new trick on the way. Type {SKXMin, SKYMin, SKXMax, SKYMax} into the edit box at the top of the Object Editor. We can actually edit several objects at the same time! This makes a lot of sense whenever we want to set a common property of a set of objects to the same value - which is exactly the present situation. Of course, it makes less sense the more different the objects are.

Set the textfont to “Times”, the textsize to 10, and the textcolor to White.

Now you will see a label up at the top left of SKViewer. Actually, all four are there but they are on top of each other. We will write some handlers to position them correctly.

Resized Revisited

The new labels are going to display the x and y values (in meters) of the edges of SKViewer. The only time these values will change is if we resize SKViewer or if we change the value of metersperpixel. We can handle the first possibility with a resized handler defined on SKViewer. Type SKViewer into the Object Editor and choose New Handler... from the Handlers menu. Choose a standard resized handler from the pop-up menu in the dialog which appears and type in:

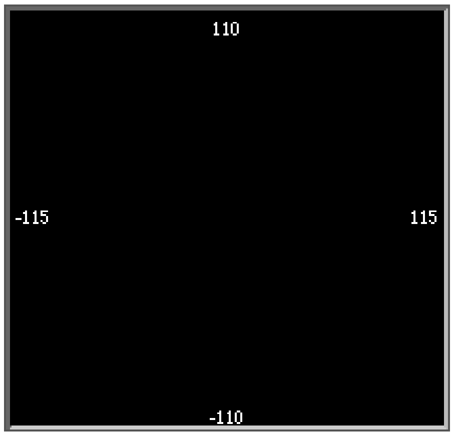
```
on resized of me (a SKViewer)
   set mpp to my metersperpixel
   set w2 to my width / 2
   set h2 to my height / 2
   set the text of my xmin to -mpp * w2
   set the text of my ymin to -mpp * h2
   set the text of my xmax to mpp * w2
   set the text of my ymax to mpp * h2
   set my xmin’s location to {my xmin's width / 2, h2}
   set my ymin’s location to {w2,my height-my ymin's height/2}
   set my xmax’s location to {my width-my xmax's width/2, h2}
   set my ymax’s location to {w2, my ymax's height}
end resized
```

Choose Activate Current Version from the handler editor’s Version menu then select SKViewer and resize it. The labels should now all snap to the correct positions and show the extent in meters of the space that SKViewer is representing.

The first three lines of the handler we just typed in set up three local variables. We did this partly to make the rest of the code more readable. But local variables are also quicker for SK8 to access than object properties so they *can* speed up a handler that makes repeated use of the same properties. In this case the speed up is probably negligible, but local variables are a good habit to get into.

We next set the text property of each of the labels to plus or minus half the size of the space represented in meters. Finally, the handler sets the locations of the labels to reasonable values.

Your SKViewer should look something like this:



At the beginning of this section we mentioned that the extent in meters of SKViewer can also change if metersperpixel's value is changed. We can arrange for the labels to update on *this* event by defining a new kind of handler called a *setter*. Use the Object Editor's Handlers menu and select the New Handler... item again. This time we will not add a standard handler but create a new one. Type set metersperpixel into the edit box and press return. Type into the Handler Editor:

```
on set metersperpixel of me (a SKViewer) to newValue
   do inherited
   set mpp to my metersperpixel
   set w2 to my width / 2
   set h2 to my height / 2
   set the text of my xmin to -mpp * w2
   set the text of my ymin to -mpp * h2
   set the text of my xmax to mpp * w2
   set the text of my ymax to mpp * h2
end set metersperpixel
```

Choose Activate Current Version from the handler editor's Version menu. This handler will be invoked whenever we try to change the value of metersperpixel.

The first line of code, do inherited, ensures that metersperpixel actually does get set to newValue by invoking the property setting mechanism SKViewer, like all objects, inherited from its ancestors.

The next three lines do our customary thing with local variables with even less justification than last time although it does make the code a little more readable. Finally, we update the text of the labels. Obviously, there's no need to reposition them this time.

You can try this handler out now by typing the following into the Message Box:

```
set SKViewer's metersperpixel to 2
```

The labels will all update. Be sure to set metersperpixel back to 1.

You may be wondering why we didn't have to insert a do inherited into handlers we defined previously. The reason is that in those cases we were defining handlers which respond to a message from SK8 that something has been done - a window has been resized or activated, or there has been a mouseDown on it. There was no work left for us to do to *complete* the event, we simply wanted to do something of our own immediately *after* the event. Our setter handler however is tinkering with the actual mechanism for setting a property which SKViewer inherited from its ancestors. The do inherited makes sure that the inherited mechanism gets done, that is, the value of metersperpixel gets set to newValue, before our new code is run.

Making the SKViewer Tool

As before, we now want to turn our *instance* of SKViewer into a tool for constructing others like it. Once again, select SKViewer and drag its name box into the top section of the Draw Tools window. Then type the into the Message Box:

```
set SKViewer's container to false
```

and SKViewer will disappear from the stage.

Notice that, as with SKWindow, we used tags rather than explicit references to the label objects SKXMax, SKXMin, SKYMax and SKYMin. Once again, this was akey feature allowing us to turn SKViewer into a tool.

The SimKit Oval Tool

The next item we will build is an oval with some special functionality. It will be useful when we come to build planet objects later in the tutorial, and could be the basis for atoms, molecules and so on.

SK8 already has a basic oval tool in the Draw Tools Palette. We will be producing a similar tool which creates our new, special ovals. We could go on and build similar tools for other SK8 graphical objects like rectangles or polygons but we do not have the space here. The principles of their construction would however be the same.

Our SimKit ovals, or SKOvals, will have some extra properties: an xsize and a ysize, and xloc and a yloc, to match the meter based coordinate system we have given to SKViewer. We will also give SKOvals velocity components in the x and y directions, xvel and yvel. These will be used when we come to animate the new objects.

We'll also need to give SKOvals some extra handlers: We will want them to set their meter based coordinates automatically when they are drawn or moved in a SKViewer.

First, we create the new object. Type into the Message Box:

```
new oval with objectname “SKOval” with container stage
```

A new oval will appear at the top left of the stage. Use the halo to drag it to a convenient position.

Adding the Properties

We're going to add and set quite a few properties so we will work from the Message Box as before. Type in:

```
addProperty SKOval, 'xsize'
addProperty SKOval, 'ysize'
addProperty SKOval, 'xloc'
addProperty SKOval, 'yloc'
addProperty SKOval, 'xvel'
addProperty SKOval, 'yvel'
```

```
set SKOval's xsize to 1
set SKOval's ysize to 1
set SKOval's xloc to 0
set SKOval's yloc to 0
set SKOval's xvel to 1
set SKOval's yvel to 1
```

Adding the SKOval Handlers

We want a handler which will automatically set the meter based properties of a new SKOval when it is drawn in a SKViewer. Once again, the best event for the job is our friend resized. Pull down SKOval's halo menu and select New Handler.... Choose the resized standard handler again and type in:

```
on resized of me (a SKOval)
   set mycon to my container
   if the properties of mycon contains ‘metersperpixel’ then
      set mpp to mycon's metersperpixel
      set my xloc to (my h - mycon's width / 2) * mpp
      set my yloc to (mycon's height / 2 - my v) * mpp
      set my xsize to my width * mpp
      set my ysize to my height * mpp
   end if
end resized
```

Choose Activate Current Version from the Version menu. This handler checks first to see if it is contained by an object with the metersperpixel property (only SKViewer has this at the moment but we might create other viewers in the future). If it is not it does nothing but if it is then SKOval calculates its meter based property values from its height, width and location within its container (actually, as well as location we are using the separate horizontal and vertical pixel based values h and v). Unless you are particularly interested, don’t worry about understanding the scaling calculations in the above handler. You can always come back to them after you have got the whole package working.

The second handler we will add allows us to drag SKOvals around in an SKViewer, keeping their meter based coordinates up to date. We’ll use the mouseDown event again so select New Handler... from the halo. Choose the mouseDown standard handler and type in:

```
on mouseDown of me (a SKOval)
   set {oldh, oldv} to my location
   set mycon to my container
   if the properties of mycon contains ‘metersperpixel’ then
      drag me with live
      set mpp to mycon's metersperpixel
      set my xloc to my xloc + (my h - oldh) * mpp
      set my yloc to my yloc - (my v - oldv) * mpp
   end if
end mouseDown
```

First of all, this handler saves the SKOval’s current h and v. Then it checks to see if the SKOval’s container has the metersperpixel property. If it has it lets the SKOval be dragged by the mouse. When the mouse button goes up it calculates the increments to the meter based coordinates from the increments to its h and v within its container.

The last handler we have to add to SKOval is almost an inverse of the resized handler. That is, it takes the meter based properties of a SKOval and it works out the corresponding boundsrect for it (Incidentally, boundsrect, is the fundamental pixel based property of graphic objects - location, h, v, height and width are all *virtual* properties which are calculated from the boundsrect). Pull down SKOval’s halo menu and select New Handler.... We’ll call the new handler scaletoviewer, so type that name into the dialog box and click the create button, then type the following code into the handler editor:

```
on scaletoviewer of me (a SKOval)
   set mycon to my container
   if the properties of mycon contains ‘metersperpixel’ then
      set ox to mycon's width / 2
      set oy to mycon's height / 2
      set mpp to mycon's metersperpixel
      set xl to ox + (my xloc - (my xsize / 2)) / mpp
      set yt to oy - (my yloc + (my ysize / 2)) / mpp
      set xr to ox + (my xloc + (my xsize / 2)) / mpp
      set yb to oy - (my yloc - (my ysize / 2)) / mpp
      if xr - xl is less than 1 then set xr to xl + 1
      if yb - yt is less than 1 then set yb to yt + 1
      set my boundsrect to {xl,yt,xr,yb}
   end if
end scaletoviewer
```

Choose Activate Current Version from the Version menu.

After checking that it is contained by an SKViewer (or at least something with the metersperpixel property) this handler works out the corners of the boundsrect for its SKOval. We need to work out the center of the containing SKViewer (ox and oy) because we are using it as the origin of the meter based coordinates.

We are finished with SKOval for now so we can add it to the Draw Tools as before. In the usual way, select SKOval, drag and drop its name into the top section of the tools window, and then set its container to false.

Making Things Move

We have done almost all the work needed to display SKOvals in SKViewers and maintain the meter based coordinate system. The reason for all this effort is that the new coordinate system will allow us to represent spaces of widely differing sizes: simply by altering metersperpixel we'll be able to go from atomic to cosmic scales. This feature will give SimKit a valuable generality.

Now, simply by modifying two existing handlers and adding two new ones we will give SKViewers the ability to animate the objects we place in them.

Making SKViewer Re-scale SKOvals

We made sure that SKOvals update their meter based properties from their pixel based counterparts by giving them resized and mouseDown handlers. We also made sure that their pixel based boundsrect can be recomputed from their meter based properties using the scaletoviewer handler. However, the two sets of properties could still be put out of step by changes to the size or metersperpixel of SKViewer. We can quickly put that right. Note that SK8 allows us to edit the handlers (or properties) of objects that are in the Draw Tools Palette as SKViewer and SKOval now are.

Find the handler editor for set metersperpixel of SKViewer. You probably have quite a few handler editors open at the moment but click on the one you want and bring it to the front. Add the extra line (last but one) shown below and Activate Current Version.

```
on set metersperpixel of me (a SKViewer) to newValue
   do inherited
   set mpp to my metersperpixel
   set w2 to my width / 2
   set h2 to my height / 2
   set my xmin’s text to -mpp * w2
   set my ymin’s text to -mpp * h2
   set my xmax’s text to mpp * w2
   set my ymax’s text to mpp * h2
   scaletoviewer every SKOval in me
end set metersperpixel
```

To cope with changes in the size of SKViewer we need to modify its resized handler. Make sure you get the correct one - we have defined resized handlers on more than one object. Add the same single line as above to the end of the handler so that it reads:

```
on resized of me (a SKViewer)
   set mpp to my metersperpixel
   set w2 to my width / 2
   set h2 to my height / 2
   set my xmin’s text to -mpp * w2
   set my ymin’s text to -mpp * h2
   set my xmax’s text to mpp * w2
   set my ymax’s text to mpp * h2
   set my xmin’s text to {my xmin's width / 2, h2}
   set my ymin’s location to {w2,my height-my ymin's height/2}
   set my xmax’s location to {my width-my xmax's width/2,h2}
   set my ymax’s location to {w2, my ymax's height}
   scaletoviewer every SKOval in me
end resized
```

Now Activate Current Version to save the modified handler.

Teaching SKOvals How to Move

Now for the new handlers. The first one will give SKOvals the ability to move themselves to a new position. All SK8 graphic objects can already be moved without the programmer needing to deal with erasing them in the old position and redrawing in the new. But our new method of moving SKOvals is based on their velocity (remember that we gave SKOvals xvel and yvel properties) and a time increment. To add a handler to SKOval, which is now in the Draw Tools Palette, go to the Object Editor and type SKOval into the text box. The Editor will fill up with SKOval’s details and you will be able to pull down the Handlers menu and choose the New Handler... item. Type updateposition into the dialog box and click on the Create button. Type in the following code:

```
on updateposition of me (a SKOval)
   set mycon to my container
   if the properties of mycon contains ‘timestep’ then
      set dt to my container's timestep
      set my xloc to my xloc + (my xvel * dt)
      set my yloc to my yloc + (my yvel * dt)
      scaletoviewer me
   end if
end updateposition
```

Choose Activate Current Version to save the new handler. The workings of updateposition are quite simple. It checks to see whether its container has a property called timestep (and remember, we added this to SKViewer back at the beginning). If it does then its value is multiplied by the SKOval’s velocity components to work out the meter based position increments. These are added to the existing x and y positions.

Telling SKOvals to Move

We are going to give the task of running the animation to SKViewer. It will store the timestep, send messages to SKOvals at regular intervals telling them to updateposition, and it will keep track of the elapsedtime. This job needs us to define a new handler on SKViewer.

Go to the Object Editor and type SKViewer into the text box. The Editor will fill with SKViewer’s details. Pull down the Handlers menu and choose the New Handler... item (by now, this routine should be familiar to you!). Once again we are going to define a standard handler. When the dialog box appears use the pop up menu and choose the idle item. Type in the following code

```
on idle of me (a SKViewer)
   lock me
   updateposition every SKOval in me
   unlock me
   set my elapsedtime to my elapsedtime + my timestep
end idle
```

and Activate Current Version. SK8 sends idle events when no other processing is going on to any object that asks for them. The lock me statement stalls all graphical updating of SKViewer until after the updateposition message has been sent to all the SKOvals in its contents. The unlock me statement allows graphical updating to proceed, making it look as though all the SKOvals are moving at the same time. Finally, the SKViewer updates its record of elapsedtime.

All the ground work is done now. We are ready to start making things work. In the next section we'll build our first dynamic simulation.

But We Have Only Done Ovals...

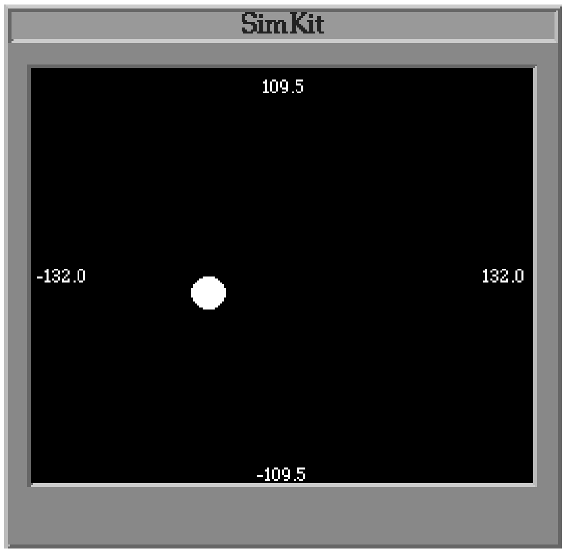
It's true, we have only prepared our own SimKit versions of Ovals. That's all we have space for here. However, you could use the same techniques for other objects, such as rectangles and polygons.

Using the Tool Kit

We can now give our tool kit a first try out. Make some space on the screen by closing all the Handler Editors.

Building a Basic Simulation

Select the SKWindow tool and drag out a new window about three or four inches square. Now select the SKViewer tool and drag out a new viewer inside the new SKWindow. Finally, select the SKOval tool and draw a small SKOval *inside* the new viewer. Try to produce something a little like the figure below. Deselect the new SKOval by clicking the mouse on either of the other objects.



The objects you have just drawn are *instances* of the tools you used to draw them. In SK8Script we say they are in the knownchildren of the tool objects. Thus the window is one of SKWindow's knownchildren, the viewer is one of SKViewer's knownchildren, and so on.

Try out the instance of SKOval's mouseDown handler. Hold the mouse down on the oval and drag it around. Notice that the oval is clipped to its container, the viewer, if you move the mouse outside the viewer's frame.

However, nothing is moving of its own accord yet. This is because we have not told SK8 that the viewer wants to be sent idle events. We do this by setting the viewer's wantsidle property to true. To make this easier we will give the new window a name and then tag the viewer inside it.

Select the window (choose the Select Tool and click somewhere in the gray area of the window), then pull down the halo menu and choose the Name... item. Type in SKSimWin and press return. Now we wish to select the viewer so that we can tag it. However, don't deselect SKSimWin but instead press the down arrow key on your keyboard.

When a complex object is selected, the arrow keys allow you to move around the containment hierarchy. In this case we have moved down a level into the layer containing the window's dragbar and the new viewer. If you now press either the left or right arrow key you will select each of these objects. Select the viewer. Now you can press the down arrow once more and move around the labels and the oval.

Come back up to the viewer with the up arrow key and choose Tag... from its halo menu. Type in viewer and press return. The instance of SKViewer has become the viewer of SKSimWin and is now easy for us to refer to in SK8Script. For example, in the Message Box, type:

```
set the wantsidle of SKSimWin’s viewer to true
```

The oval will start to move. It moves up and right because we set SKOval's default values for xvel and yvel to 1. Notice that you can still drag the oval around with the mouse (providing it didn't move off the screen before you got to it!)

Now type:

```
set the wantsidle of SKSimWin’s viewer to false
```

and the animation will stop. We have built our first simple (*very* simple) simulation of a moving object.

Using the SimKit Overviewer

Don't worry if the oval moved out of sight. To find it we can make use of the SimKit Overviewer again. The Overviewer is useful for getting references to objects we can't see (either because they are off the screen or because they are not graphical objects, such as Renderers) or whose name we do not know. Even if you haven't lost the oval, click on the All Objects button so that the list box fills with a host of items.

You are looking for SKOval 1 in the viewer of SKSimWin, although probably not all of the name will be visible. As usual, if you hold the mouse down on this expression in the list box it will highlight and show a gray frame indicating that you can now drag the reference and drop it into the name box of the Object Editor. Do this and scroll the Object Editor's property list down to xloc and yloc. Set them both to 0. Then type into the Message Box:

```
scaletoviewer SKOval 1 in the viewer of SKSimWin
```

The oval will immediately move to the center of the viewer.

The Need for Controls

Controlling the animation by typing commands into the Message Box as we did above is rather clumsy. It would be better if we could switch it on and off with a button. In fact, it's easy to imagine that buttons will come in useful for many interface requirements so in the next section we will make an SKButton prototype and add it to our tool kit.

The SimKit Button Tool

Our SimKit Button tool will be quite a simple object, based on Rectangle. We'll create it and set its properties from the Object Editor.

Creating SKButton

We start with a new rectangle on the stage. Type into the Object Editor:

```
new Rectangle with objectname “SKButton” with container Stage
```

Notice that you can even create new objects in the Object Editor. Now set the following properties of SKButton one by one: set the size to {50,20}, the fillcolor to GrayTone50, the framesize to {2,2} and the framecolor to SKBevelOut.

We've given the SKButton a simple 3-D look. Carry on typing: set the textfont of SKButton to Palatino, the textsize to 14, the textstyle to {'bold'}, the textcolor to DarkGray and the text to "SKButton"

That completes its appearance but we also want SKButton to highlight when we click on it, so type:

```
set the autohighlight of SKButton to true
```

But the default rectangle highlighting behaviour is to invert it. We can override this and make the button look pressed in. First type:

```
set the inverts of SKButton to false
```

Now we'll add our own code for handling the highlighting. We'll set SKButton's framecolor to SKBevelIn when the highlight is true.

```
on set highlight of me (a SKButton) to newValue
   do inherited
   if newValue then
      set my framecolor to SKBevelIn
   else
      set my framecolor to SKBevelOut
   end if
end set highlight
```

That's SKButton defined. Add it to the Draw Tools as before by dragging its name into the Draw Tools, then set its container to false.

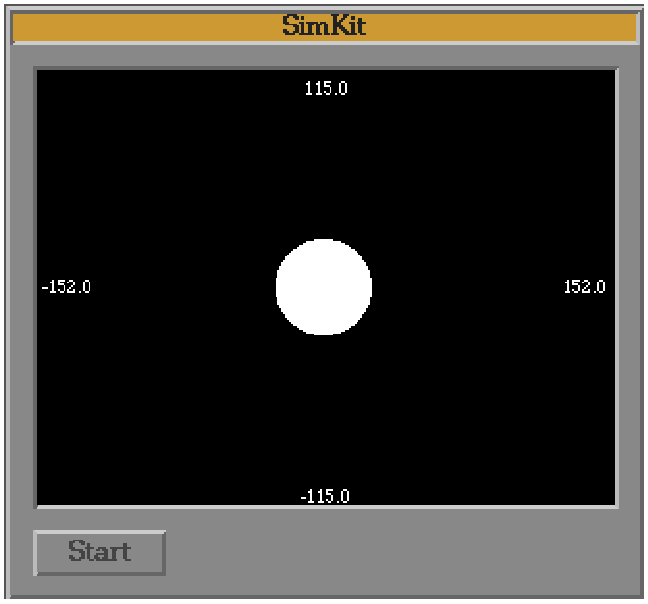
Using SKButton

Now you can create a button to turn the animation on and off. Start by using the SKButton tool to draw a new button in SKSimWin. Make sure your button really*is* in SKSimWin, you may have to stretch it a little (Select it and use the halo's resize dots) to give yourself room.

After you have drawn the new instance of SKButton, pull down its halo menu and Name it SKStartStopButton. Now choose the New Handler... item. When the handler dialog box comes up use the pop up menu and choose the standard click handler. This is the handler that deals with a mouse click on a graphic object. Type into the handler editor:

```
on click of me (a SKStartStopButton)
   if (my container’s viewer)'s wantsidle then
      set (my container’s viewer)'s wantsidle to false
      set my text to "Start"
   else
      set (my container’s viewer)'s wantsidle to true
      set my text to "Stop"
   end if
end click
```

Use the Handler Menu to Activate Current Version and try it out.



The handler checks to see if its container’s wantsidle is true (which is why it is important that its container is SKSimWin). If it is asking for idle events then the next two lines tell it to stop doing so and set the text of the button to “Start”. Otherwise, its wantsidle is set to true (and the animation starts) and the text is set to “Stop”.

More SKButtons and Using “its”

We’ll create a couple more buttons now, one to magnify the region represented by the viewer, the other to shrink it. This turns out to be easy to do. Use the SKButton tool to draw two buttons like those in the diagram below. Select them in turn and Name one SKZoomInButton, the other SKZoomOutButton. In the Message Box type:

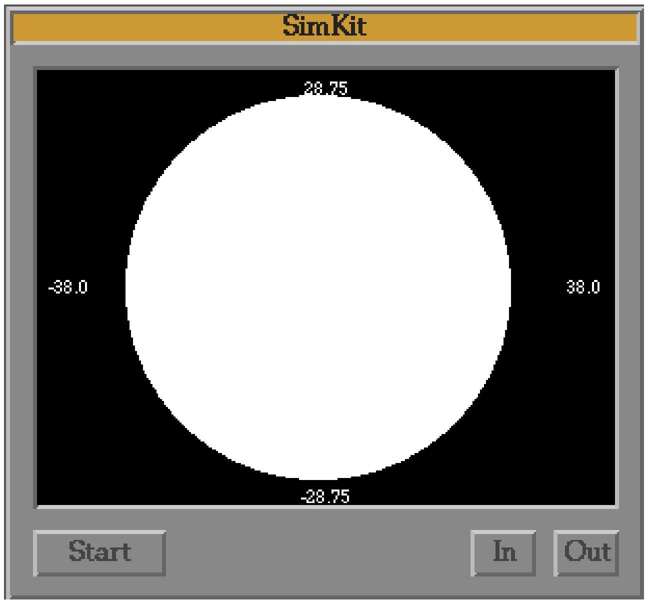
```
set the text of SKZoomInButton to “In”
set the text of SKZoomOutButton to “Out”
```

Now define a Click handler on each (Use the halo menu’s New Handler... item, and don't forget to Activate Current Versions):

```
on click of me (a SKZoomInButton)
   set myCon to my container's viewer
   set myCon's metersperpixel to its metersperpixel / 2
end click

on click of me (a SKZoomOutButton)
   set myCon to my container's viewer
   set myCon's metersperpixel to its metersperpixel * 2
end click
```

This is what you should have now. Try it all out.



The two click handlers work by altering the viewer’s metersperpixel. All the redisplaying of the oval and the labels is taken care of by the viewer’s set metersperpixel handler. Notice how SK8Script allows you to use the provisional form of its to refer back to an object you have previously referenced in a line of code. This feature can make scripts more readable because it makes it clear that the code is refering to just one object. When the references to objects are complex (and they can become much more complex than our example) this can be difficult to see at a glance.

The SimKit Gauge Tool

Another useful tool for our kit would be a gauge for displaying a named quantity. We could use such a device to display SKViewer's timestep or elapsedtime. We'll build the gauge tool on the stage and then pop it into the Draw Tools before using it in SKSimWin.

Begin by selecting the Rectangle tool and drawing out a rectangle about an inch long and half an inch high. Use the Name... halo menu item and call it SKGauge. Then draw another rectangle just inside the first, name it SKGaugeValue and then Tag it as the valubox of SKGauge.

We will set the graphical properties for these two new objects from the Message Box. Type SKGauge into the Object Editor and set the following property values: set the fillcolor to GrayTone50, the framesize to {0,0}, the text to "SKGauge", the textfont to Palatino, the textsize to 9, the textstyle to {'bold'}, and the textlocation to 'topcenter'.

A few points here deserve explanation. We have set SKGauge's framesize to zero in both directions, which is the same as telling it not to have a frame. We set the style of its text to bold - the {'bold'} syntax arises because we may wish to set a compound style and SK8Script allows us to do this in a *list* (signified by curly braces) each item of which must be a *symbol* (signified by single quotes). Thus we could use a textstyle such as {'bold', 'outline', 'italic'} but we will resist the temptation. Finally we have changed its textlocation to 'topcenter' (another symbol) from the default position, 'center'. Now we'll set up SKGaugeValue. As before, type its name into the Object Editor and set the following property values: set the fillcolor to GrayTone50, the framesize to {1,1}, the framecolor to SKBevelIn, the text to 0, the textfont to Palatino, and the textsize to 9.

The plan for SKGauges is that we can set their text and their valuebox's text to show the name and value of the property they are displaying. We'll make sure the valuebox always fills the bottom half of the gauge by giving SKGauge a resized handler. Select the SKGauge and choose the halo menu's New Handler... item. Choose the standard resized handler and type in:

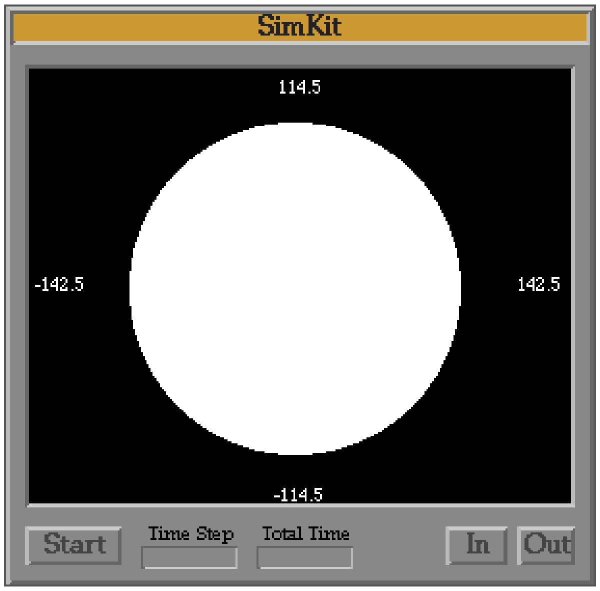
```
on resized of me (a SKGauge)
   set my valuebox’s boundsrect to {0,my height/2,hsize,my height}
end resized
```

We'll also make sure that gauges don't go below a certain size. This is done not by setting properties but by defining another handler on SKGauge. This time call the handler minimumsize. Its definition should be:

```
on minimumsize of me (a SKGauge)
   return {50,25}
end resized
```

SK8 uses this handler internally to make sure an object's size never gets set smaller than the value it returns. Now we can put SKGauge into the Draw Tools. Once again, set its prototype to true, drag its name into the Library Editor, then set its container to false.

Use the new tool to draw a couple of gauges inside SKSimWin as shown below:



Select each in turn (make sure you are selecting the gauge and not just its valuebox) and Tag one of them as the timestepgauge of SKSimWin, and the other as the elapsedtimegauge of SKSimWin.

You can set their texts quite easily now from the Message Box. Type:

```
set the text of SKSimWin’s timestepgauge to “Time Step”
set the text of SKSimWin’s elapsedtimegauge to “Total Time”
```

Making the Gauges Update

There is nothing at present to make the values of these gauges update when the values of the quantities they represent change. To fix this we once again use set handlers. The actual properties timestep and elapsedtime belong to SKSimWin’s viewer and so it is on that object that we should define the setters. Of course, that means we now have to name this instance of SKViewer so select the viewer, choose Name... and call it SKSWViewer. Then choose New Handler... and type set timestep into the dialog box. When the Handler Editor appears type in:

```
on set timestep of me (a SKSWViewer) to newValue
   do inherited
   set ((my container's timestepgauge)'s valuebox)’s text to
   newValue
end set timestep
```

Repeat the process for set elapsedtime:

```
on set elapsedtime of me (a SKSWViewer) to newValue
   do inherited
   set ((my container's elapsedtimegauge)'s valuebox)’s text to
   newValue
end set elapsedtime
```

Don't forget to use the Handler Menu to Activate Current Version for both the handlers.

If you now type into the Message Box

```
set SKSWViewer’s timestep to 1
```

the value in the Time Step gauge should update. Press the Start button and the Total Time gauge will display the value of elapsedtime.

Creating a Simulation Tool

The object we have built is one important step short of being a useful simulation: the objects we place in the viewer (SKOvals) have a completely uninteresting behaviour. If they could be made to feel a force or to interact with other objects in the viewer then we would have a system worth investigating. However, this shortcoming is also an opportunity: we can turn what we have now into a high level tool for producing simulation frameworks. Draw one of these out on the screen, add some interesting behaviour and *voila*, we have something of value. We'll do a little bit of tidying up and create this high level tool then finish the tutorial by using it to produce a gravitational simulation.

Tidying Up

First, get rid of the SKOval. We want the tool to start with an empty viewer. Select it and press the delete key. Next, change the text of SKSimWin's dragbar to "SKSimWin". Type into the Message Box:

```
set the text of SKSimWin's dragbar to "SKSimWin"
```

Sensible Layout for SKSimWin

We'll change the code of SKSimWin's resized handler so that it positions its component objects in a sensible way when we draw out a new one or resize one that already exists. You may prefer to work out your own layout but we provide an example any way:

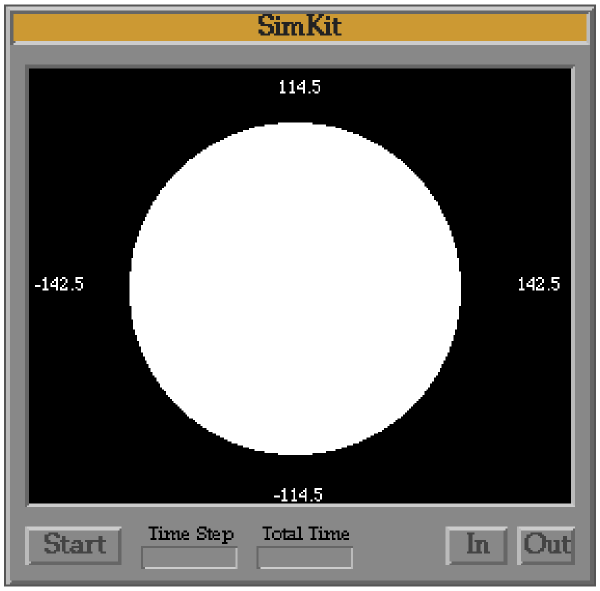
```
on resized of me (a SKSimWin)
   set {dx, dy} to my framesize
   set {newW, newH} to my size
   set my dragbar's boundsrect to {dx,dy,newW-dx,dy+18}
   set my viewer's boundsrect to {10,30,newW-10,newH-40}
   set my startstopbutton's boundsrect to {10,newH-30,60,newH-10}
   set my timestepgauge's boundsrect to {70,newH-32,120,newH-8}
   set my elapsedtimegauge's boundsrect to
      {130,newH-32,180,newH-8}
   set my zoominbutton's boundsrect to
      {newW-72,newH-30,newW-42,newH-10}
  set my zoomoutbutton's boundsrecT to
      {newW-40,newH-30,newW-10,newH-10}
end resized
```

Use the Handler Menu to Activate Current Version.

Just as we did with the gauge, it is a good idea to set SKSimWin's minimum size to a value which leaves room for all the components. The following handler works for the layout we specified above. Select SKSimWin again. Choose New Handler..., and type minimumsize into the dialog box. When the Handler Editor appears type in:

```
on minimumsize of me (a SKSimWin)
   return {260,250}
end minimumsize
```

Now you can select SKSimWin and resize it with the halo's dots. Make it as small as you can. You should have something like this:



Creating the SKSimWin Tool

Finally, we can turn SKSimWin into a tool. As before, drag its name into the top part of the Draw Tools, then set its container to false.

We are ready to begin creating a simulation of real interest. Before moving on, close all the handler editors you have on the screen then use SK8's File menu to Save SimKit.

Building Planets and SKGravitas

Gravitas is an educational dynamic simulation which was built to allow students to investigate the behaviour of gravitating objects. The program has direct manipulation tools with which users can create planets and stars. These may be positioned and set in motion and the user can observe their trajectories as they interact with each other under the force of gravity in two dimensions.

We will be building SKGravitas, a simple version of this application using the tools we have built in the tutorial. To do this we need to modify just two objects: we must create a child of SKOval that has the added property of mass, and we must augment SKViewer's animation method so that it can apply the gravitational force to these new ovals.

Creating the SKGravitas project

Pull down SK8's File menu and choose the New... item to open a new project. A dialog box will appear asking you whether you want this new project to be a sub-project of SK8 or SimKit. Choose SimKit - our new project is going to need access to the objects we have defined so far. We are however, free to define any number of independent sub-projects in SimKit. This means we will be able to have different kinds of simulation running independently under SimKit if we like. Type SKGravitas into the Objectname field of the file dialog box and press the New button.

Creating Planets

Use the SKOval tool to draw out a small circular object on the stage. Make it about half an inch in diameter and name the new object Planet. Drag Planet's name bar over into the Object Editor's name box and use the Properties menu's New Property... item. Type mass into the dialog box which appears then select this property in the Object Editor and set its value to 0. Next, select Planet's framesize property and set it to {0,0}. Finally, select Planet's fillcolor property and set it to metal2. This last adjustment gives Planet a simple (you might say 'bogus') 3-D look.

Now we can turn Planet into a tool. As before, drag its name into the Draw Tools, then set its container to false.

Creating Gravitas

Now use the SKSimWin tool to draw out a new simulation window on the stage. Use the Name... item of the halo menu and call it Gravitas. Also, select the viewer inside Gravitas and name it GravitasViewer. While it is still selected, choose the halo menu's New Handler... item. Pick the standard idle handler and type in:

```
on idle of me (a GravitasViewer)
    set planetlist to every planet in me
    if planetlist is not empty then
        repeat with thisPlanet in planetlist
            repeat with otherPlanet in planetlist
                if otherPlanet is not the same object as thisPlanet then
                    set rx to (otherPlanet's xloc) - (thisPlanet's xloc)
                    set ry to (otherPlanet's yloc) - (thisPlanet's yloc)
                    set modr to sqrt((rx * rx) + (ry * ry))
                    if modr < (thisPlanet's xsize + otherPlanet's xsize) / 2 then
                        beep
                        set the wantsidle of me to false
                        set the text of my container’s startstopbutton to “Start”
                    end if
                    set F to 6.67e-11*(otherPlanet's mass)/(modr*modr *modr)
                    set thisPlanet's xvel to thisPlanet's xvel+rx*F * my timestep
                    set thisPlanet's yvel to thisPlanet's yvel+ry*F * my timestep
                end if
            end repeat
        end repeat
    end if
    do inherited
end idle
```

Use the Handler Menu to Activate Current Version

We have printed this handler in a small font so that the long lines fit across the page. This increases readability, hopefully not at too much cost to your eyesight. What the handler actually *does* can be explained in words: GravitasViewer builds a list of all its planets. Then for each one of these it sums the changes in velocity the current planet will experience due to the gravitational force of each of the others. While it is doing this it also checks for collisions. If a collision occurs the handler stops. Finally, the handler calls its inherited version which updates the positions of all the planets and sets the elapsedtime.

You don’t have to worry about understanding this handler thoroughly now. However, it could be a useful model if you decide to build a simulation of your own with different behavior.

That completes the work for SKGravitas. All we need now is a few realistic planets. We will create the objects shown at the beginning of the tutorial, that is, Jupiter and its four principal moons (it has several other much smaller satellites).

Creating Jupiter and its Moons

We could use the Planet tool, draw five objects in GravitasViewer and then set their properties in the Object Editor. However, this would be a laborious and error prone method. Equally laborious, but less error prone is to create the objects and set their properties in two special handlers which you can check and correct. Select SKGravitas and choose New Handler... from the halo menu. Call the first handler createjupiterandmoons and type in:

```
on createjupiterandmoons of me (a SKGravitas)
  set my viewer's metersperpixel to 1.5e7
  new Planet with objectname "Jupiter" with container GravitasViewer
  new Planet with objectname "Io" with container GravitasViewer
  new Planet with objectname "Europa" with container GravitasViewer
  new Planet with objectname "Ganymede" with container GravitasViewer
  new Planet with objectname "Callisto" with container GravitasViewer
end createjupiterandmoons
```

Now define a second handler called setupjupiterandmoons. Type:

```
on setupjupiterandmoons of me (a SKGravitas)
   set the mass of Jupiter to 1.9e27
   set the xsize of Jupiter to 1.428e8
   set the ysize of Jupiter to 1.428e8
   set the xloc of Jupiter to 0
   set the yloc of Jupiter to 0
   set the mass of Io to 8.89e22
   set the xsize of Io to 7.26e6
   set the ysize of Io to 7.26e6
   set the xloc of Io to 0
   set the yloc of Io to -4.25e8
   set the xvel of Io to -17348
   set the mass of Europa to 2.27e22
   set the xsize of Europa to 6.28e6
   set the ysize of Europa to 6.28e6
   set the xloc of Europa to 0
   set the yloc of Europa to 6.66e8
   set the xvel of Europa to 13741
   set the mass of Ganymede to 1.48e23
   set the xsize of Ganymede to 1.05e7
   set the ysize of Ganymede to 1.05e7
   set the xloc of Ganymede to 0
   set the yloc of Ganymede to 1.07e9
   set the xvel of Ganymede to 10875
   set the mass of Callisto to 1.07e23
   set the xsize of Callisto to 9.6e6
   set the ysize of Callisto to 9.6e6
   set the xloc of Callisto to 0
   set the yloc of Callisto to 1.88e9
   set the xvel of Callisto to 8205
   scaletoviewer every planet in my viewer
end setupjupiterandmoons
```

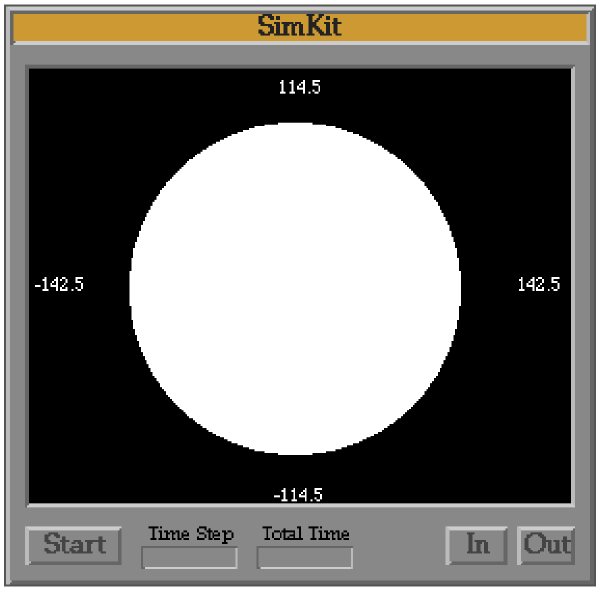
Remember to Activate Current Version for both these handlers.

The data above can be found in almost any text on the Solar System although they usually do not specify orbital *velocity* (which is what we have used to set the xvels) but instead give the orbital *period* and *diameter*. Of course, it is easy enough to convert between these representations. You could even write a SK8 handler to do it.

Now we will set things up. Type the following lines one by one into the Message Box:

```
createjupiterandmoons SKGravitas
setupjupiterandmoons SKGravitas
```

The Planets will be created and this is what you should see:



You may have to resize your SKGravitas to bring all of the objects into view - Callisto is almost 2 x 109 meters from Jupiter. Now press the Start button. Sadly, not much seems to happen except that the Total Time indicator begins to count up. However, this is a clue to what is wrong - the timestep is so small that the positions are changing too slowly to notice. We can easily change the timestep by typing into the Message Box:

```
set GravitasViewer's timestep to 2000
```

Now things start to happen. You can watch the moons revolve around Jupiter or drag them out of position and watch their orbits change. ou can also zoom in or out while the system is running. Take a look at Io, the closest moon. Io is roughly the same distance from Jupiter as our moon is from the Earth. However Io takes less than two days to orbit Jupiter while our moon takes about 28 days. This is the effect of Jupiter’s much stronger gravitational pull.

You should save your project now. We have finished work on the tool kit and shown how it can be used. The application we have built already allows us to model and examine some interesting phenomena. And as we have noted, the tool kit should also make it a much simpler task to produce simulations of other objects and forces. We will finish off this tutorial with some suggestions for further work.

Going Further With SimKit

There are several ways in which you could modify and extend SimKit. First of all, you could fix a bug in the Resized handler of SKOvals (and therefore Planets). You may already have noticed that if you zoom the Jupiter’s moons system out a few times and then zoom back in the same number of times, the objects have changed size. In fact they will probably all appear to be the *same* size. This is because our current Resized handler re-computes the meter based size of an SKOval every time its boundsrect is changed. It should in fact only do this the very first time an SKOval is resized, that is, just after it is created.

In the interest of good pedagogy we should probably leave this fix as an excercise for the reader. However, that seems too cruel, so here is a suggestion:

```
on resized of me (a SKOval)
   global SKViewer
   set mycon to my container
   if not my meterSizeSet then
      if the properties of mycon contains 'metersperpixel' then
         set mpp to mycon's metersperpixel
         set my xloc to (my h - mycon's width / 2) * mpp
         set my yloc to (mycon's height / 2 - my v) * mpp
         set my xsize to my width * mpp
         set my ysize to my height * mpp
         set my meterSizeSet to true
      end if
   end if
end resized
```

Notice that this handler uses a new property called meterSizeSet. To make the handler work you will have to add this property to SKOval and set its value to false.

A feature of SKGravitas that could definitely be improved upon is the method of creating planets. This is really quite clumsy as it stands. A better method would allow us to set the size, location, velocity and mass of Planets by direct manipulation. Direct manipulation though is poor at providing the kind of precision we need in this application. Can you think of ways around this?

Another weakness of SKGravitas is that the Planets are not labeled and it is therefore difficult to identify them. A simple solution would be to set the text of a Planet to its objectname, but this would not be very useful as the objects are often much smaller than their text.We saw how useful SK8's label object could be in the creation of the SKViewer. Can you think of a way to attach labels to Planets and have them always move with them? You will have to modify more than one handler.

Finally, you may want to investigate other physical domains. Good examples would be the dynamics of charged particles, movement in the presence of friction or air resistance, or motion in a varying potential field. Your ingenuity rather than SK8 will probably be the limiting factor to what is possible. Good luck.

Actor and Stage

Actor and Stage are the objects which provide the core functionality for the graphics and windows that might be associated with a particular SK8 application / title.

All Actor objects descend from the Graphic object and provide most of the drawing capabilities for SK8. The Stage Object controls the placement and display of actors on the screen. Actors, through their containment hierarchy, can become SK8 Windows.

This chapter provides an introduction to Actor, Stage, Windows, and the containment hierarchy.

Introduction

Stage and Actor Metaphor

The main graphic objects of a SK8 project are named after the metaphor of a theatrical play. Like a play, all the visible action of the computer program takes place on the `Stage`. Just as a theater's stage can be of any size, SK8's `Stage` object—at least the part that is visible—varies in dimensions according to the size of the monitor(s) in use. The Stage is the plane in which all the configured monitors exist. For the Macintosh, the Stage is the Desktop.

Actors can take on a variety of forms and functions. Ovals, polygons, and line segments are just a few of the simple geometry actors which let you draw. SK8 also features a variety of other actors that allow interaction. Some of these actors are designed to accept text entry, while others allow graphical selections or changes.

Just as a playwright writes dialogue telling actors how to perform, the SK8 programmer can write scripts to tell SK8 actors how to perform.

Because of its object-oriented architecture, scripts written in SK8 can make actors — which are objects — behave and respond in almost every conceivable way.

For example, SK8 actors can:

- change their location on the stage
- alter their appearance
- interact with other actors you create or from a library of actors
- disappear, or change behavior
- respond to user input

The costume and makeup for each actor is determined by a set of Renderer objects. These can be anything from simple colors to Quicktime^{TM} movies.

Actor

As previously mentioned, the Actor object provides most of the drawing capability necessary to build a title or multimedia application for the Macintosh or other computer platforms. The Actor object provides the default handlers for drawing, animating, and controlling groups of objects that appear on the screen.

Actor supports two major kinds of objects:

- simple geometric objects, such as `Polygon` and `Ovals`
- user interface objects, such as `Scroller`, `TextList`, and `RadioButton`

An actor can contain one or more actors of any type. For example, an Oval in SK8 can contain a Rectangle (that is, the contained Rectangle would be visible inside the boundary of the Oval).

Actors are used to create windows, clip the drawing of other actors, group other actors together, zoom and pan other actors, and more.

With the exception of `MenuItem` and `Cursor`, every graphic object used in SK8 is a descendant of Actor. The simplest actors take shapes such as circles or squares and have no function other than being visible on the screen. In addition, most types of actors are capable of displaying text.

Complex actors perform a diversity of tasks. `Scroller, RadioButton, EditText,` and `TextList` are some of the SK8 actors that interact with the user. Their actions depend on input from the user.

The Stage

The `Stage` object controls the coordinate system used for placement of all windows in SK8. Windows are located within the boundaries of `Stage`, and have their own logical coordinate system for the objects they contain.

The `Stage` object can be thought of as a huge polygon that matches the shape of the monitors as placed in the Monitors' control panel. The origin of the `Stage` object is located at the upper-left corner of the monitor that contains the menu bar.

There is only one `Stage` object. The `Stage` object automatically configures itself to the current monitor setup. When multiple projects are in use, they all share the same `Stage` object.

Containment Hierarchy

The **containment hierarchy** presents an image of the relationships of the visual elements on the screen. When a child of Actor is moved to the Stage, it is added to the Stage's contents. Each descendant (child) of Actor has a contents property, that lists its contents, and a container property, that indicates which object it is contained by in the containment hierarchy.

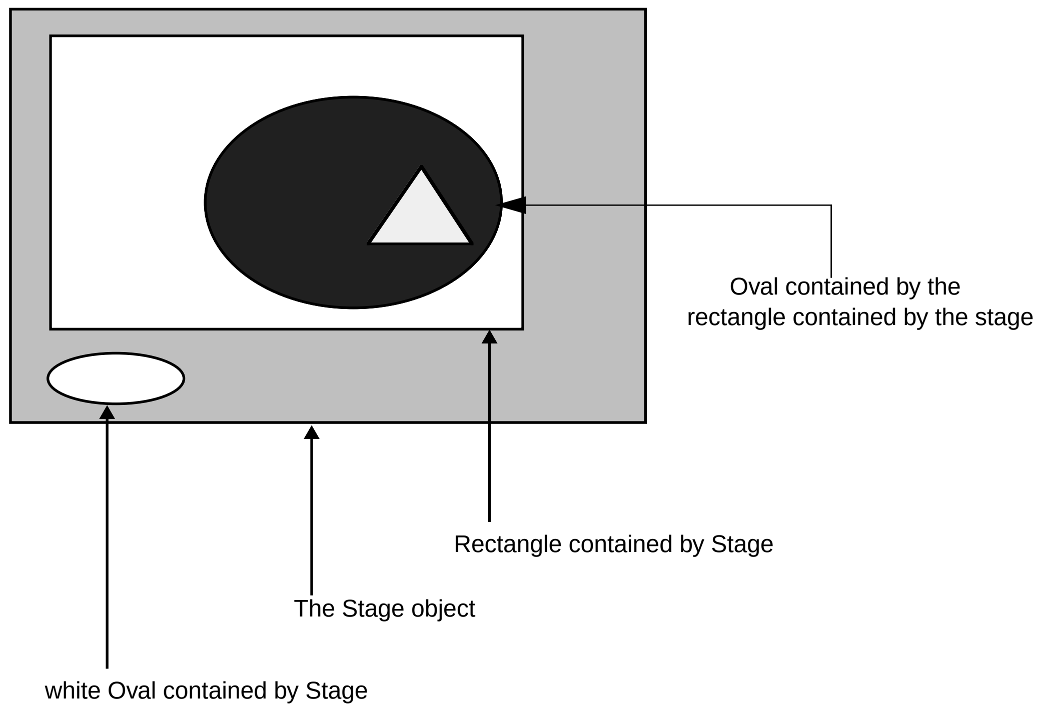
To be visible on the screen, an object must be contained by the Stage or contained by an object that is contained by the Stage (i.e. in the Stage's containment hierarchy) or, to carry this further, contained by an object that is contained by an object that is contained by the Stage.

The containment hierarchy is the order in which Actors in SK8 are contained by other Actors.

Actors must be attached to a container to be visible inside that container. A container is either another Actor or the Stage. The top of the containment hierarchy is the object Stage. When an Actor is contained by the Stage, the Actor appears directly on the desktop. Other Actors may, in turn, be attached to the Actors that are attached to the Stage.

Figure 1-1 illustrates the containment hierarchy.

Figure 1-1 Containment Hierarchy


Oval contained by the
rectangle contained by the stage
Rectangle contained by Stage
The Stage object
white Oval contained by Stage

The large gray rectangle, the Stage object, contains a white Oval and a white Rectangle. The white Rectangle contains a dark Oval, and the dark Oval contains a triangle. The triangle is contained by the dark Oval, which in turn is contained by the white Rectangle, which is contained by the Stage. All objects are visible because they are all, in some way or another, contained by Stage.

Actor

Actor Properties

An actor’s properties determine its appearance and function, such as: color, width, size, draggable by the mouse, etc.

All properties can be read by the user, and most properties can be set by the user.

Examples:

To get the color of the interior of the rectangle:

```
get HotRectangle's fillColor
```

To store the value of `CoolOval's` visible property into the variable x:

```
set x to the visible of CoolOval
```

To set the location of `MyOval` to the coordinates {30,30}:

```
set the location of CoolOval to {30,30}
```

To set the `frameHeight` property of `CoolOval` to 4:

```
set CoolOval's frameHeight to 4
```

Many actors also have properties that are specific to themselves and their descendants. For example, `CheckBox` actors have a `checkColor` property that determines the color of the X in a highlighted `CheckBox.` Other types of actors do not have a `checkColor` property.

Attaching Actors to The Stage

When a new child actor is created in SK8Script, the new actor is not placed in any container. The actor must be contained by the Stage or another actor eventually contained by the Stage in order to be seen.

Example:

The examples below show two actors created as children of SK8's built-in Rectangle and Oval actors. The first actor is then made to be contained by the Stage, while the second actor is made to be contained by the first actor.

Create a Rectangle actor:

```
set actor1 to new Rectangle
```

Create an Oval actor:

```
set actor2 to a new Oval
```

Make actor1 contained by the Stage:

```
insert actor1 into the Stage
```

Make actor2 contained by actor1:

```
insert actor2 into actor1
```

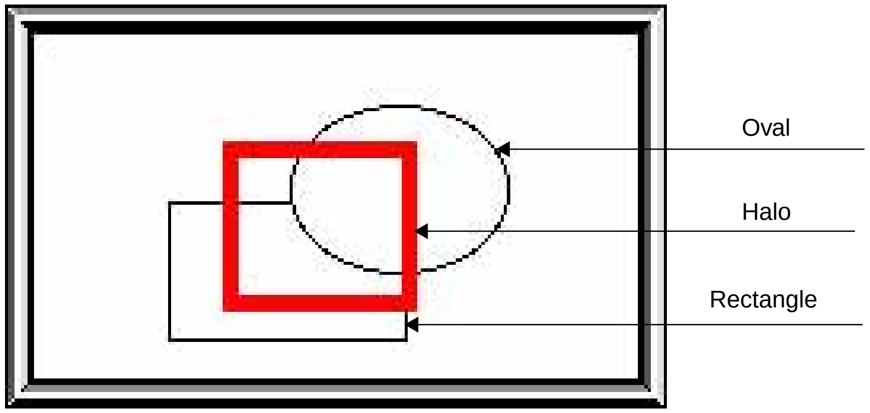
Note

Container here only refers to Containment Hierarchy. ◆

Halo (an Actor)

The Halo is a rectangle with a "hole" in the middle. Think of it as a rectangle whose fillRegion has been taken away. It is used by the User Interface of SK8 to implement the Selection Object. The following figure shows a Halo object on a window.

Figure 1-2 A Halo in front of two other actors.



Complex Actors, SubActors, and Tags

Complex SK8 actors are often made up of a collection of subActors. A subActor is an actor that is used in constructing a complex SK8 actor. In SK8 terms, a subActor of Actor is an actor in Actor's contents. A vertical Scroller, for example, is made up of four subActors:

- a Rectangle with an up arrow on it
- a Rectangle with a down arrow on it
- a shaded Rectangle between the up and down arrows
- a final Rectangle subActor (the "thumb") that moves on the shaded subActor.

Repeatedly accessing an actor's subActors is inconvenient. The objectName of a Scroller's thumb, for example, has nothing to do with the Scroller to which it is attached. Without tags, changing the `fillColor` of the Scroller's thumb would require specifying the subActor by its objectName:

```
set the fillColor of Rectangle255 to brown
```

Tags provide a more efficient method of specifying actors. Tags allow a subActor to be referenced through the name of the object it composes. If, in the above example, the Scroller's thumb had been tagged, the following line could be used to change the thumb's `fillColor`:

```
set the fillColor of the thumb of the Scroller266 to brown
```

Better still, if the Scroller had a handler to modify its thumb, it could include a line like:

```
set the fillColor of my thumb to brown
```

The Stage

Actor Coordinates

All drawing in SK8 occurs on the Stage. The Stage, which is represented in SK8 by the Macintosh desktop, defines a coordinate system on which SK8 actors are positioned and sized according to standard coordinate systems.

The coordinate system uses a horizontal and a vertical value to specify each coordinate. The actor is positioned using coordinates that reference the upper-left corner of either its container or the `Stage` object.

A point on the Stage is specified in two ways: physically or logically. In SK8, each actor has both logical and physical coordinate properties.

The physical coordinates of an actor define its location relative to the origin of the Stage. In Figure 1-3, the physical top left coordinates of the gray rectangle specify its distance from the origin of the Stage. With physical locations, a value of {152,104}, for example, means that the specified point is located 152 pixels to the right and 104 pixels below the Stage's upper-left corner of the start-up monitor.

The logical coordinates of an actor define its location relative to the origin of the object that contains it. In Figure 1-4, the logical top left coordinates of the gray rectangle specify the distance from the origin of its immediate container.For example, if you set the logical location of an Oval to {32, 54}, the Oval will be placed 32 units to the right and 54 units below the upper-left corner, or position {0,0}, of its container.

Each SK8 actor defines its own logical coordinate system for the actors it contains. Note that if the container of an actor is the Stage, then its logical and physical coordinates are the same.

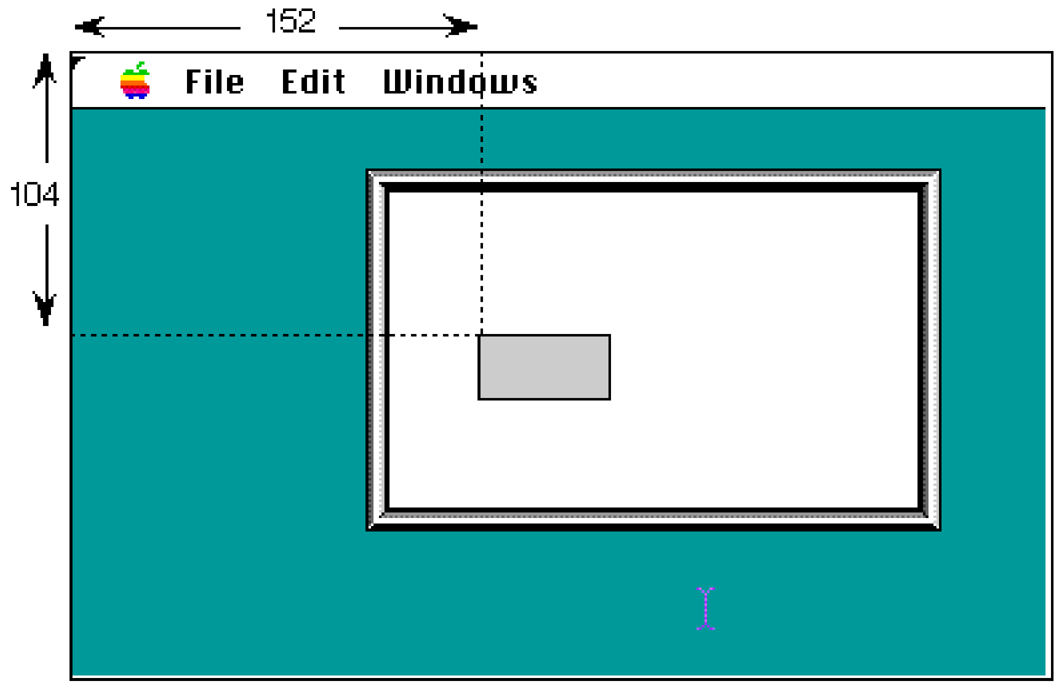
Physical Coordinate System Properties

A physical coordinate is one that determines the actor's **size** and **location** on the Stage. Performing operations such moving an actor or its container changes the physical position of an actor.

Changing any one of the physical location properties usually changes many or most of the other physical coordinate properties. For example, changing the `physicalBoundsRect` property affects physical properties pertaining to: size, left coordinates, right coordinates, top coordinates, bottom coordinates, location, etc.

Physical properties also include `scale`, `hScale`, and `vScale`. These properties display the specified actor at a multiple of the actor's logical dimensions. For example, if a Rectangle contained an Oval whose logical `size` was 20, and the Rectangle's `scale` was set to 3, the Oval would take up 60 pixels when displayed, even though its logical `size` remained at 20 units.

Figure 1-3 Physical Coordinates



The physical coordinates of the top left corner of the gray rectangle {152,104} are the number of pixels to the origin of the stage.

Logical Coordinate System

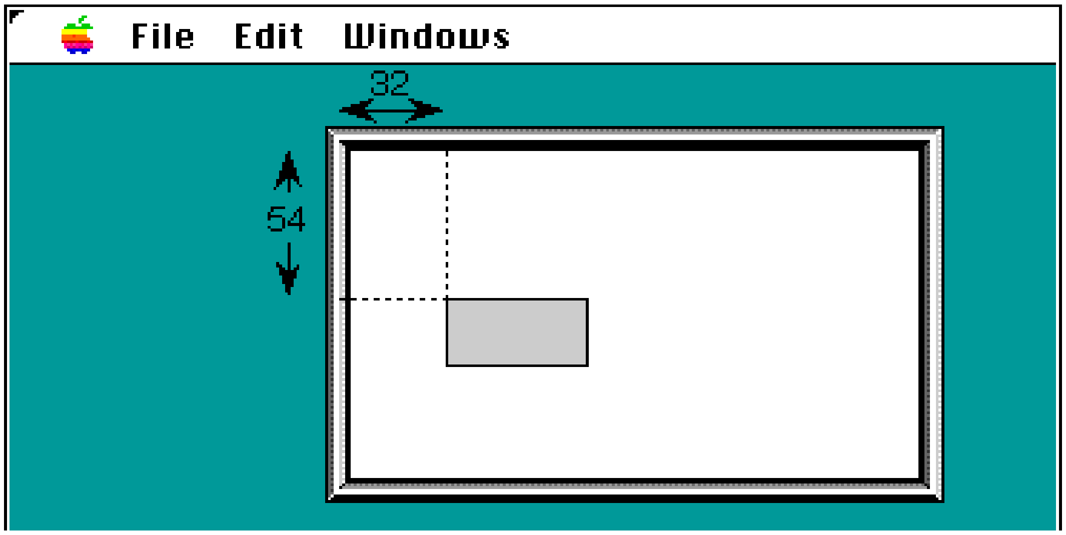
A logical coordinate system is one that is determined by the actor's position relative to the origin of its immediate container. The origin is the upper left corner of the boundsRect of the container.

Performing operations such as panning, zooming, or moving the actor's container in no way alters the logical position of an actor. The only way to change an actor's logical coordinates is to change its size or position within its container.

Changing any one of the logical coordinate properties usually changes many or most of the other logical location properties. For example, changing the `boundsRect` property affects properties pertaining to: size, left coordinates, right coordinates, top coordinates, bottom coordinates, location, etc.

The arguments assigned to logical location properties are based on units. A unit is equal to one pixel, if the actor's container's `hScale` and `vScale` properties are set to their default value of 1.0. As the scale changes, the unit size changes proportionately. To change the way an actor is scaled or panned, you change the `scale` or `origin` properties of the actor's container.

Figure 1-4 Logical Coordinates



The logical coordinates of the top left corner of the gray rectangle, {32,54}, are the number of units to the origin of its immediate container.

origin

The `origin` property of Actor accepts or returns two numbers. The first number specifies the container’s horizontal origin. The second number specifies the container’s vertical origin.

```
get the origin of Actor
set the origin of Actor to {h, v}
```

The `origin` property has a two-item list argument that selects a point based on the specified actor's logical coordinate system. When the actor's `origin` property is set, the point specified by the argument is moved to the upper-left corner of the actor's `boundsRect.` As a result, the entire contents of the actor pan to accommodate the new origin.

Note
It is possible to set a container’s `origin` to a point either outside or inside the container.◆

Setting the `origin` property effectively moves the specified point to the upper-left corner of the container. As a result, actors inside the container will pan vertically and / or horizontally to fit within the new coordinate system. For example, setting the origin of a container (whose `origin` is currently set to {0,0}) to {-5, 5} will move all contained actors 5 logical units to the right and five vertical units up. Until the `origin` is again changed, the point at the actor's `boundsRect's` upper-left corner would have a value of {-5,5}.

FillColor Of The Stage

The fillcolor of the Stage can be set to an `RGBRenderer.` The stage will be rendered with this color (provided the `Stage` is covered).

Windows

Actors Directly Contained by Stage (Windows)

In SK8, a **Window** is any actor whose container is the Stage. **Window** refers to all actors that are contained **directly** by the `Stage.` In Figure 3-1, the white oval and the white rectangle are the only Windows. Any actor moved to the Stage becomes a Window and behaves accordingly. All actors may become Windows, since all actors can be attached to the `Stage`. Because windows are actors, they are ordered, selected, shown, hidden, and manipulated the same way as other actors.

The following statements create an actor that behaves like a traditional Macintosh window from the built-in `Rectangle` object (the new child of `Rectangle` is a window because it is attached directly to the `Stage`):

```
set MacWindow to a new Rectangle

set the container of MacWindow to the Stage
```

The first statement creates a new rectangle actor and binds it to the variable `MacWindow`. The second statement sets the container of the rectangle to the `Stage.`

If we follow the above example but create a new `Oval` instead of a new `Rectangle,` we get an oval-shaped window.

Note
This is not in compliance with the Macintosh Human Interface Guidelines. If you are creating an application to run on other platforms, capabilities such as this are useful. ◆

colorDepth and windowStyle

When an actor becomes a Window, some of the actor's behavior as a Window may be affected by the way that it is displayed. This is the case with the following two properties:

- `colorDepth`
- `windowStyle`

`ColorDepth` is the bit depth (number of bits per pixel) the actor wants to use for itself as it becomes a window. This specifies whether offscreen graphics memory will be used, and how many colors the window accepts. Note that an actor can only dictate the `colorDepth` of its window when the actor is the window.

`WindowStyle` is the type of window used by the actor when it attaches itself to the stage. The default is for actors to show up on a blank window, but you can here specify the usual Macintosh window types, such as document with zoom, etc.

Activate and Deactivate

When you click on the window and it is not the front most one, then the front most window is sent a `deactivate` event and the window that received the click receives an `activate` event.

You can use the `activate` event, for example, to set up a menubar which is active when your window is in the front. You would then remove the menubar when your window is no longer the front most window.

Only one window can be active at a given time.

KeyTarget

When an actor becomes a Window, the actor acquires some extra properties and capabilities, such as: `keyTarget. KeyTarget` is a very important property that specifies which actor inside a window receives the window's key events. When a Window actor receives the `activate` event described above, we want a specific actor to be the `keyTarget` for that window. Each Window can have only one current `keyTarget.` In the case where the Window actor may contain many subActors, **only** one subActor can be the `keyTarget` for a window at a time. The Window remembers which subActor is its `keyTarget.` You can set the Window's `keyTarget` as follows:

```
set the keyTarget of Window to someActor
```

where **`Window`** is the window object of desire and **`someActor`** is a subActor of that Window.

Note
If this call is made to an actor that is not a Window, there is no effect on the actor. ◆

Note
Additional information regarding key events is located in the Event System chapter. ◆

Windows of Stage

By using the `windows` property, you can get the windows of the stage.

Example:

```
get the windows of the stage
```

Hide and Show

To hide or show a window, you may use the `hide` or `show` handlers. Another way is to use the `visible` property.

Example:

To show window x, and then check if it is visible:

```
show x
if x is visible then sendToLog "x is visible"
```

To hide a window, the `hide` handler can be used or the window's `visible` property can be set to `false.`

Window Styles

The `Rectangle` object supports specialized styles used in standard Macintosh windows.

Example:

The following code sequence, from the example above, creates a Macintosh document window:

```
set MacWindow to a new Rectangle with container Stage ¬
   with windowStyle 'Document'
```

Table 1-1 SK8 Window Styles.

Window Style	Description
'blank'	a “plain vanilla” window with no standard objects attached
'document'	a standard Macintosh window with close box and menu bar
'documentWithZoom'	a standard Macintosh window with close box, menu bar, and zoom box
'doubleEdgeBox'	a window that is generally used for messages or dialogs, and is surrounded by a double line
’movableDialog’	a Macintosh movable dialog like the Copy File dialog
'singleEdgeBox'	a window that is generally used for messages or dialogs, and is surrounded by a single line
'SK8Window'	the standard window available under SK8 with a beveled frame that can be dragged
'tool'	a Macintosh window with rounded corners, a close box, and a draggable title bar
’documentwitharrow’	<need info>
’shadowEdgeBox’	<need info>

C H A P T E R 8

Browsers

This chapter is under development. It will appear in the finished manual.

Clipboard and Import/Export

SK8 includes built-in support for exchange data with the outside world. This chapter introduces the SK8 import/export architecture, including Translators and the Clipboard.

The SK8 Import/Export Architecture

The SK8 Import/Export Architecture supports data exchange with the world outside of SK8. In SK8, we only manipulate SK8 objects; while in the outside world many data formats are available depending on the current hardware and software configuration. For translation purposes, SK8 includes built-in support for common translations for the Macintosh and is extensible for more esoteric translations and translations to different platforms.

Translators address the source and destination in all import and export activity. You can read and write data into any form of file. For this, you read and write data using the `Stream` object abstraction to avoid problems related to arbitrary file representations.

Translators know how to turn SK8 objects into various formats (e.g. turning an `Actor` into a 'PICT' `Resource`) and various external formats into SK8 objects (e.g. turning a '`cicn`' into a `MaskedActor`). Specific objects are preferred to a suite of conversion functions because they can be used to query the state of our system: we can ask the objects whether they can be invoked to perform the translation or not.

Exporting

To export a SK8 object into a stream, do the following:

1. Find all the `Translator` objects that know ways to export the object.
2. Select the translator desired.
3. Do the conversion and write the result to the stream specified.

Importing

To import data from a file into SK8, do the following:

1. Given the data available, find all translators that know how to turn it into SK8 objects.
2. Select the translator desired (yields the thing we want).
3. Convert the data into a new object of the desired type in the specified project.

Note
Given a stream, you need a function that tells you what type of data can come from it. ◆

Translator (an Object)

Translators inherit from the `Translator` object. Each translator handles conversions between an internal object and an external object. The translation is done by an export and an import handler.

Translator Properties and Handlers

`internalObject`	The SK8 object that the translator translates or yields as a result of a translation.
`externalObject`	A keyword, specifying the external representation of the data the translator is translating. Examples: 'PICT', 'cicn' 'snd'.
`import`	A handler that reads the type required from the stream specified and creates the appropriate SK8 object in the specified project. `import Translator, stream with Project (a Project | false), thing` If `Project` is `false`, the currentProject is used for the creation.
`export`	A handler that translates a SK8 object into an external representation and writes the result to the stream provided.
translatorsApplicable stream	Returns all the translator objects that can read data from the stream and converts it.
importTypesAvailable stream	Returns all the SK8 objects you could create from the data in a stream. This is done by checking the data in the stream with the available translators. All applicable translators return their SK8 types.
exportTypesAvailable listOfObjects	

Returns a list of keywords specifying all forms you can export given a list of SK8 objects. This is done by finding all applicable translators and returning their external types.

exportTypes object

Returns all the forms into which you could export the object (a list of keywords).

importTypes type

Given a `type` keyword, returns the SK8 object types you could make from it.

The SK8 Clipboard (an import/export application)

The `SK8clipboard` is a structure that provides SK8 a way to cut and paste within SK8 as well as across applications. Across applications, we use the import/export architecture described above, treating the system's clipboard as a stream (see discussion below).

Overview and Requirements

The `SK8clipboard` is the object that supports all cut and paste activity in SK8. Within SK8, it serves as a temporary receptacle for objects that have been copied or cut. When an object is copied or cut, it is placed into the clipboard, from where it is retrieved (and possibly copied) when a paste operation takes place.

The clipboard also serves as an interface to the world outside of SK8. This importing and exporting takes place when SK8 becomes or ceases to be the current application.

The clipboard object supports the following operations:

- temporary store of objects (one or many), returned on demand.
- querying of objects currently on hold.
- automatic importing and exporting of objects when applications are switched.

The Clipboard Object; its Properties and Handlers

The `SK8Clipboard` object is the recipient of things that have been cut or copied. For all user purposes, the clipboard only deals with SK8 objects.

The `SK8Clipboard` object has the following properties:

`objectsOnHold`	The objects kept in the clipboard themselves (this property is private and should only be accessed by the user through the API).
`clipboardOpen`	Determines if the clipboard is open to activity outside SK8. If this property is false, no importing and exporting activity is automatically triggered by the clipboard. Defaults to true.

`incomingData`	Determines if the clipboard's contents are the contents of the system scrap. This has to do with the use of the clipboard as an importer of data from the system (see below for details).

These capabilities allow SK8 authors (including the SK8 Project Builder) to write their own cut/paste system. Handlers act on the clipboard object. The `SK8Clipboard`, a child of `Clipboard`, is SK8's base clipboard, being the only one that talks to the outside world. Authors are welcome to use it for their purposes, which should be sufficient in most cases. They can also define their own clipboards to do more esoteric things.

The Clipboard within SK8

In this section we describe the handlers that allow the user to implement common cut and paste behavior within SK8. They provide the means to place things in the clipboard, remove things, and query the clipboard for its contents.

addToClipboard — Adds something to the clipboard.

```
addToClipboard thing, clipboard
with copySelectionToClipboard (a Boolean)
with overWrite (a Boolean)
```

If `copySelectionToClipboard` is `true` (the default), the thing is copied before being placed in the clipboard. Otherwise it is placed directly. If `thing` contains actors, they are removed from any containers with which they are associated. If `overWrite` is `true` (the default), `thing` replaces whatever was in the clipboard before (this can cause disposal of objects by calling `clearClipboard`). Otherwise, `thing` is added to whatever is in the clipboard already.

clearClipboard — Clears the clipboard. This handler is called just before the clipboard is overwritten by something else. All the items in the clipboard are disposed.

```
clearClipboard clipboard
```

objectsInClipboard — Returns a list of all the objects in the clipboard. These are the objects stored in the `objectsOnHold` property of the `Clipboard` object.

```
objectsInClipboard clipboard
```

typesInClipboard — Lists all the types of objects on the clipboard. In its simplest form, this is a list of all the parents of the objects in the clipboard with duplicates removed. For example, if three `RGBColors` are in the clipboard, only `RGBColor` is listed.

```
typesInClipboard clipboard
```

typeInClipboard — Returns true if a descendant of type is in the clipboard.

```
typeInClipboard clipboard, type
```

getFromClipboard — Retrieves items from the clipboard. The second argument specifies the type of object we want.

```
getFromClipboard clipboard,  thing
   with copySelectionToClipboard (a Boolean)
      with removal (a Boolean)
      with everyone (a Boolean)
      with project (a Project)
```

If `copySelectionToClipboard` is true, the default, copies of each item found are returned.

If `removal` is true, the items returned are then removed from the clipboard. The default is false.

If `everyone` is true, all items of the specified type which are found are returned. Otherwise, only the first item found is returned. The default is false.

If a `copySelectionToClipboard` needs to be made, it is made in the project specified (or the current project if no project was passed).

Cut and Paste within SK8

The event system is responsible for generating the events required to drive the cut and paste system. The system events are:

```
cutSelectionToClipboard

copySelectionToClipboard

pasteClipboardToSelection
```

We write cut and paste handlers for the objects that require this functionality; like `actor` and `EditText`.

Cutting and Pasting Text

For `EditText`, the cut and paste process is fairly simple.

`copySelectionToClipboard` gets the selected text and destructively (wipe out what is there) places it in the clipboard.

```
on copySelectionToClipboard of me (a EditText)
   set theText to my selection
   addToClipboard sk8Clipboard, theText
end copySelectionToClipboard
```

`cutSelectionToClipboard` does the same, but the text that is currently selected is deleted.

```
on cutSelectionToClipboard of me (a EditText)
   set theText to my selection
   set {start, end} to my selection
   -- clear the text.
```

```
      set my text with start start with end end to ""
      -- add the string to the clipboard without copying.
      addToClipboard sk8Clipboard, theText without¬
         copySelectionToClipboard
   end cutSelectionToClipboard
```

For `pasteClipboardToSelection`, we insert, at the cursor's position, any text that we find in the clipboard. For simplicity, we assume there is only one string in the clipboard.

```
   on pasteClipboardToSelection of me (a EditText)
      set {start, end} to my selection
      set theText to getFromClipboard sk8Clipboard, text
      set my text with start startChar with end endChar to theText
   end pasteClipboardToSelection
```

Cutting and Pasting with the `selectionHalo`

A slightly more interesting example has to do with the `selectionHalo`, the Project Builder actor in charge of selecting actors and changing them using direct manipulation. Let us write the `cutSelectionToClipboard`, `copySelectionToClipboard` and `pasteClipboardToSelection` handler for the halo. When the halo is selecting an actor, it can copy or cut the actor, or it can choose to paste into it something in the clipboard.

Copying is simple, just gather everything in the selection and put copies of each item in the clipboard.

```
   on copySelectionToClipboard of me (a SelectionHalo)
      addToClipboard sk8Clipboard, my selectedItems
   end copySelectionToClipboard
```

The `cutSelectionToClipboard` handler is exactly the same except the copy optional argument is set to false. In this case, the actors are automatically removed from their containers.

In the example below, the `pasteClipboardToSelection` handler assumes the selectionHalo is currently selecting an actor. Every actor in the clipboard is added to the contents of the selected actor.

Note
If more than one actor is selected, we choose as the recipient of the `pasteClipboardToSelection` action the front most container on the screen that contains everything that is selected. The Stage itself might be such container. This is all done for you by the Project Builder.

```
   on pasteClipboardToSelection of me (a SelectionHalo)
      set theActors to getFromClipboard sk8Clipboard, actor¬
      with everyone
      -- for simplicity, just get the first item.
      set theRecipient to item 1 in my selectedItems
      withLockedActor theRecipient
         set the container of every item in theActors to¬
         theRecipient
      end with
   end pasteClipboardToSelection
```

In the example above, `pasteClipboardToSelection` takes all the actors in the clipboard and sets the container of each of them to the “first selected” actor in the selection halo. In other words, `pasteClipboardToSelection` places all the actors from the clipboard into the “first selected” actor. Note that the selection halo may contain several actors. The selection of the “first selected” actor, of the several that may be in the selection halo, is strictly arbitrary. The actor that happens to be the first in the `selectedItems` list become the “first selected”.

The Clipboard and the System

The `SK8Clipboard` is the only clipboard object specialized to interface with the outside world. If the `clipboardOpen` property is true, then the `SK8Clipboard` interacts with the system and can be used to import and export things in and out of SK8. In this section we discuss what happens when SK8 suspends and resumes when the `SK8clipboard`'s `clipboardOpen` property is `True`.

Note that the discussion that follows relies heavily on the Import/Export architecture outlined above.

In order to use the Import/Export system, we need to treat the system's clipboard as a stream. A stream will be created to represent the system's scrap, and write and read operations will translate to `putScrap` and `getScrap`.

What happens on resume...

Basically nothing happens. If the scrap has changed from the time we left SK8, the `SK8Clipboard`'s `incomingData` property is set to `True`, meaning that there is new data in the scrap. The old contents of the `SK8Clipboard` are disposed. All further action happens when the user tries to get something from the `SK8Clipboard`.

In the first place, the `typesInClipboard` handler behaves differently when the `incomingData` property is set to `true`: it calls `importTypesAvailable` to find out what SK8 objects could be created. The stream argument is the `SK8Clipboard` itself.

The user can then call `getFromClipboard` with the appropriate SK8 object type, and the object will be read in from the scrap using the import handler of the appropriate translator. All work required will be done by the translator.

What happens on suspend...

Given the objects in the `SK8Clipboard`, we call the `exportTypesAvailable` to find all the forms of outputs we can produce. With the resulting list, we have to select a target type and export it, using the export handler of the chosen translator. Two decisions still have to be made:

- which of the objects in the `SK8Clipboard` is chosen for the export.
- which of its translators is used.

Clocks

This chapter is under development. It will appear in the finished manual.

Collections

SK8 provides a powerful and flexible notion of a Collection. This chapter explains collections, their use, and how they can be extended.

Introduction

The `collection` object is the basic behavior for representing a collection of other objects in SK8. SK8 currently represents many different types of collections. They appear as descendants of the `collection` object. They are: list, vector, string, text, and string (string resides under text).

The `collection` object is not a functional working object per se. It is strictly an abstract prototype. It contains no items. You can not look into it. The `collection` object strictly defines the basic behavior of collections.

A collection in SK8 is a descendant of the `collection` object, but the true meaning of a collection, from SK8 ‘s view, is any object that properly implements the collection protocol. All the built-in SK8 objects properly implement the collection protocol.

Generally when dealing with collections, a scripter will only create children of existing collection prototypes— lists, strings, vectors, and arrays.

The way that the average scripter creates a collection to work with is either by starting with a list or string literal form.

Or the average scripter will grab an existing collection. For example, doing some sort of operation on the Stage to come up with a subcollection like “`every oval whose fillColor is red`”. This will create a list that corresponds to some subcollection of Stage.

In general, to use SK8 you should not need to create a new `collection` object. You can get an existing collection, use string and list constructor to create string and list collections respectively. If you do need to create a new array, -you want to use the correct prototype. Do not instantiate collection to make a new collection.

The concept of Collection in SK8 does not imply any particular representation (linked list, block of memory, etc.). This allows all the different types of collections, but all accessed the same way via the SK8 scripting language. The implementation is not as important because Collection is a protocol. As long as the object responds to the messages you send it, it appears to be a collection to SK8.

In addition to the representation not being specified, a fixed ordering is not really a part of the protocol, although every collection that exists in SK8 has a fixed order. You can potentially have a collection that gives you a different order each time you iterate over it. In this case you would want to specialize the `index selection` handler to get the correct response back, given a particular key.

Dialogs

This chapter is under development. It will appear in the finished manual.

C H A P T E R 1 3

Tools and Palettes

This chapter is under development. It will appear in the finished manual.

EditText

This chapter is under development. It will appear in the finished manual.

CHAPTER 15

The Error System

This chapter is under development. It will appear in the finished manual.

Event Interests

This chapter is under development. It will appear in the finished manual.

C H A P T E R 1 7

The Event System

Introduction

The SK8 Event System allows your project to receive user-initiated and other kinds of system events. User events, for example, are generated when the user depresses a key on the keyboard, or presses or releases the mouse button. SK8 identifies these events and lets your project know that the event has taken place by invoking event handlers which you define in your project's objects.

The SK8 Event System provides the following services:

- **Delegation:** The ability to easily delegate events up the containment hierarchy.
- **Event Modes:** The ability to circumvent the normal event system in order to perform sensitive, non-interruptible operations.
- **Mouse Sensitivity:** The user's ability to contextually specify SK8's interpretation of mouse operations in order to achieve a high degree of flexibility over how and which events are sent to a user's actors.
- **Event Tracking:** The ability to track and intercept events.

The event engine supports:

- A full suite of low-level user input events, such as click, doubleClick, mouseEnter, etc.
- A suite of high-level protocols, such as drag-drop, connect-disconnect, started-stopped, used in conjunction with graphics, animation and other activities.
- A suite of high-level SK8 system events, (e.g., suspend-resume, tick) for communication with the operating environment.
- A suite of event modes for performing critical operations by partially or totally inhibiting the event system.
- User-defined event modes

- Event trapping

Delegating Events

SK8 allows one to **delegate** events up the containment hierarchy. In other words, SK8 event delegation allows the handling of an event at a higher place in the containment hierarchy than that of the original event target.

For example, if you don't implement a `mouseDown` handler for the `Oval` actor, then SK8 will pass the `mouseDown` event to the actor that contains the oval. Suppose that your oval is contained by a rectangle. You have an opportunity to "trap" the oval's `mouseDown` event, and that of any other actor contained by the rectangle, in the rectangle's `mouseDown` handler. (See below for "trap" definition.)

Event delegation provides another dimension for programming in SK8. The specific actor to which SK8 originally sends an event is called the **event actor**. The default behavior of all event handlers is to pass the event to the event actor's container unless you have written a script for the handler of the event actor. The container to which the event is passed is called the **secondary target** of the event. If you haven't written a script for the secondary target, SK8 will automatically pass the event to the secondary target's container. This delegation continues up the containment hierarchy until either the Stage is reached or the actor has no container.

Whenever you write a script to handle an event, you **trap** the event in that handler. Once trapped, an event will no longer be delegated up the containment hierarchy. However, if you wish to delegate the event, even though you have trapped it, you may do so by using the **pass** command.

Delegation programming techniques are not a substitute for using SK8's inheritance capabilities, but in some cases they provide an elegant solution to an otherwise complex problem. For example, suppose you want to create a button which beeps when someone presses it. The ideal solution might be to create a prototype button actor with a `click` handler that beeps. Then make as many children or copies of this button as you need. All buttons would inherit the `click` handler.

But suppose you have twenty different actors contained by a rectangle and you want each of them to print something into the message box upon receiving a click event. Instead of defining a `click` handler for each actor (20 handlers in total), you can define one `click` handler for the rectangle. When any one of the actors receives a click event, the event will be delegated from the individual actor to the `click` handler of the rectangle (the container of the twenty actors). Thus, one handler on the container replaces 20 potential handlers. Quite a savings, in terms of lines of code!

Handling Events

Event Functions

`eventActor`	Holds the actor to which SK8 originally sent an event. For example, on a click, this is the actor on which the user intended to click.
`eventH`	Holds the h (horizontal coordinate) of the position where the event took place. The value of this global depends on the event. For example, on a click event, this global contains the h position of the mouse at the time it was clicked.
`eventV`	Holds the v (vertical coordinate) of the position where the event took place.
`eventTime`	Holds the time, in SK8 system ticks, the event took place.

Pass Command

`Pass [`**`arguments`**`]`

Use the `Pass` command only if you wish to delegate a trapped event to your actor's container. For example, suppose you wish to create a beep sound when any key is depressed on a text field actor. To pass the `keyDown` event to your field's container, which might be a document which counts the number of keystrokes, you would write a script such as this one:

```
on keyDown of me (a BeepField)
   beep
   Pass
end keyDown
```

In this example, the handler nullifies normal processing of the `keyDown` event by the `BeepField.` If `BeepField` were, e.g., a `SimpleText` field, the character corresponding to the depressed key would not print in your field. To have it print and still delegate the `keyDown` event to your field's container, do the following:

```
on keyDown of me (a BeepField)
   beep
   do inherited
   Pass
end keyDown
```

The above is an example of how inheritance and delegation can be used in one handler to achieve different effects.

Note that the `pass` command is a logically equivalent, but more concise way of doing the following:

```
   on keyDown of me (a BeepField)
      beep
      if my container ≠ false then keyDown my container
   end keyDown
```

Note
Using `Pass`, instead of “passing” an event, allows SK8’s event- tracking system to more fully track event processing.◆

Event Modes

Event modes allow circumvention of normal event processing so critical operations are performed without disruptions caused by unanticipated processing of certain events. Event modes are objects which descend from `EventMode.`

Defining your own event mode is straightforward. Create a child of `EventMode` and assign one or more handlers to the event mode object.

The following event modes are predefined in SK8:

dragOnStageMode	Drags an actor on the stage.
MenuSelectMode	Used during popup menu selection.
ModalDialogMode	Makes dialogs modal.
Port Edit Layer Mode	Shows port wiring.

By using an event mode, you instruct the event system to circumvent its event-processing procedure. While your event mode is active using `enterMode,` SK8 will not send the normal events to your actors.

Instead, SK8 sends an alternate set of events to your event mode object, so that you may handle the event in a manner appropriate for your event mode. In addition, you may suspend certain types of event-processing altogether by entering your event mode using the `enterModalState` handler instead of `enterMode.`

Creating Your Own Event Mode

You create your own event mode by creating a child of `EventMode.` You can override, or “shadow” the `EventMode’s` handlers as required by your application. You can add any properties you wish to the event mode object in order to preserve its state.

IMPORTANT
Never delete a property or handler inherited from EventMode. ◆

Mouse Sensitivity

When a user points at an object by clicking the mouse, the user’s intention may not coincide with SK8’s interpretation of the pointing gesture. For example, if an oval is contained inside a rectangle, and the user intends to click on the rectangle, but the pointer is over the oval, the `click` event will be sent to the oval instead of to the rectangle.

SK8 lets you manage this situation in a number of ways. One is to use delegation, in which case the oval's `click` handler would delegate the `click` event to its container (the rectangle). The rectangle would get the `click,` as the user intended.

You can also change SK8's interpretation of mouse events by changing the value of the `mouseSensitivity` property of your actor.

Click Interpretation

Sometimes you may have conflicting requirements on how you want a click, double-click, or mousedown events to be interpreted by the system. Sometimes, when the user clicks or double-clicks, you may not want to have the actor receive a `mouseDown` event. The `doubleClickStyle` property, defined on actors, permits you to control this. More information on the `doubleClickStyle` property is in the Actor object description.

The possible styles are:

`'standard'` (the default) `'doubleClickOnly'` `'clickOnly'`

Note
See the "SK8 Projects and Libraries" chapter for additional information on openedProject and openedLibrary. ◆

Event Tracking

Event tracking is a category which subsumes a variety of services provided by SK8. These include:

`MouseSensitivity`	Controls SK8's responses to a user's pointing actions (e.g., with the mouse).
`ClickInterpretation`	Controls SK8's interpretation of a click, versus a double-click versus a pointer-down and pointer-up combination.
`EventRecording`	Records or logs SK8 events.
`EventInterception`	Intercepts SK8 events.

Files

SK8 provides a simple interface to long term data storage in the form of:

- Files
- Streams

File Objects

Every computer operating system that SK8 supports represents long term data storage (a.k.a. disk storage) in operating system structures known as files. Furthermore, the operating systems all organize files in a hierarchial system of volumes (disks), folders (directories).

Each file can be uniquely identified by a file name and its position in the hierarchy. For example, there can only be one file named Folio in the same folder (directory). A file can also be specified by telling the operating system its disk, and all the directories that lead to it; e.g. on a Macintosh “MyHardDisk:MyFolder:Folio”; on a PC “MyHardDisk\MyDirectory\Folio”; on a Unix system “/MyDisk/MyDirectory/Folio”. This specification is sometimes called a “PathName”. In SK8 this specification is manifested by the “File” object.

Please note that a file object is not a file itself nor is it a file’s data. It is merely the unique identification of a file that gets correctly translated to the host operating system. File objects are just “smart name strings.” In fact, just because you have a valid file object, it does not mean it refers to a file that actually exists. You may have a valid file object that refers to a file that has not yet been created, or that has been deleted. Similarly there can be several different file objects that refer to the same file.

An analogy: a name tag may have your name on it; it refers to you. It is easy to image several name tags with your name on them; each one refers to you. It is also easy to image a name tag of someone who is not present, or a name tag for a fictional character (i.e. someone who does not exist). File objects are very much like these name tags, only they refer to files instead of people.

So what can you do with file objects? Actually quite a lot. There is a rich set of object handlers that give you one standard programming interface to access files from whatever operating system your SK8 application happens to be running on. Below are a few examples of these handlers. Examine a File object with the SK8 Object Editor and refer to the SK8 Object Reference Manual for more detail descriptions.

physicalName and logicalName

File Objects use two separate ways for specifying a file:

- logicalName
- physicalName

PhysicalNames specify the file with the explicit name of the disk, every directory leading to the file and finally the filename itself.

For example:

MyDisk:MyTopDirectory:MyLocalDirectory:Folio

Note the directories are separated by colons. Also note no colon appears before the first thing in the path; i.e. the disk name.

LogicalNames provide a shortcut from having to supply the file's entire path. Instead of starting with the disk name, logicalNames start with the name of a `FileAlias` object. Two `fileAlias` objects are defined in SK8:

- "SK8" which is the directory in which the SK8 application resides, and
- "Root" the top level of all operating system paths. On a Macintosh, Root is the DeskTop.

An example of a logical name would be:

SK8;SK8:SK8

which is the SK8 file, inside the SK8 directory, inside the directory in which resides the version of the SK8 application you are running. Note the first separator in logicalNames is a semicolon.

File Object Handlers

FileExists

Returns `True` if the file that the File Object refers to actually exists and can be accessed.

```
Set x to new file with logicalName “SK8;SK8:sk8”
FileExist of x -- returns true
```

```
Set y to new file with logicalName “SK8;NotHere”
FileExist of y -- returns false
```

CreateFIle

Given a File Object that refers to a file that does not exist, the handler will instruct the operating system to create a file as specified by the file object.

```
Set y to new file with logicalName “SK8;ImNewHere”
CreateFile y
```

Delete

Given the actual file exists, this command instructs the operating system to delete the file.

```
Delete y
```

Note
This does not end the File Object’s existence. ◆

isDirectory

Returns `True` if the File Object refers to an operating system directory (i.e. folder on a Macintosh) and not a data file.

```
New file with objectName “HereTis” with ¬
   logicalName “SK8;SK8 Library”
isDirectory  HereTis  -- returns true
Set x to new file with logicalName “SK8;SK8:sk8”
isDirectory x  -- returns false
```

Files

If the File Object refers to a data file, this handler returns a list of files it contains.

```
new file with objectname “HereIAm” with logicalname “SK8;”
files of HereIAm
-- returns: {the File “SK8;Library:”, the File ¬
      “SK8;Patches:”,the File
--  “SK8;SK8:”, the File “SK8;SK8 v1.0”, the File “SK8;SK8 ¬
      Temporary  Files:”}
```

Directories

If the File Object refers to an operating system directory, this handler returns a list of directories it contains.

```
directories of HereIAm
-- returns: {the File “SK8;Library:”, the File ¬
   “SK8;Patches:”, the File
--  “SK8;SK8:”, the File “SK8;SK8 Temporary Files:”}
```

Name

Return various forms of the name of the file that the File Object refers to.

```
Set x to new file with logicalName “SK8;SK8:sk8”
Set y to new file with PhyiscalName “MyDisk:Applications; ¬
   Sk8 V1.0:SK8:SK8:sk8”
```

With no parameters it returns the name of the file as it was originally specified so:

```
name of x -- returns “SK8;SK8:sk8”
name of y -- returns “MyDisk:Applications;Sk8 V1.0:SK8:SK8:sk8”
```

If the directory parameter is set to `False`, only the name of the file is returned:

```
name of x with directory false -- returns “sk8”
name of y with directory false -- returns “sk8”
```

Streams

`Streams` are used to perform input and output operations, both sequential and random access. Two types of streams exist:

- `TextStreams` are used for text and characters. TextStreams have an understanding of words, paragraphs, lines, and so forth.
- `ByteStreams` are used for binary data. You are responsible for reconstituting the byte stream data when you read the file. SK8 provides minimal functionality to help you with input/output of SK8 objects. Refer to the `writeObject` handler in the SK8 Object Reference Manual.

Stream Handlers

Some of the handlers for the `Stream` object API (application program interface) are:

`writeStreamItem` stream, item

`readStreamItem` stream

`atStreamEnd` stream

`streamPosition` stream

Virtual property which has a setter.

`streamWriterInfo` stream

Provides fast repeated performance of `writeStreamItem.`

`streamReaderInfo` Provides fast repeated performance of `readStreamItem.`

Streams as Collections

Since Streams are collections, all of their operations are available via selection expressions. Read and write operations are supported in ByteStreams and TextStreams via the collection protocol so that one may read or write to a file from within a repeat loop or by using a complex selection expression.

Example:

```
get item 410 in myByteStream
set item 410 in myByteStream to 255
```

TextStreams may also use text-oriented selection expressions.

Example:

```
get the second word in myTextStream

set the fifth character of the 1004th word¬
   in myTextStream to “x”
```

Note
Refer to the “SK8Script Language” chapter for more information on Selection Expressions. ◆

▲ **WARNING**
Writing into a Stream in such a way as to cause items to be inserted in the middle of the Stream is not currently supported. ◆

Current Directory

The `currentDirectory` is a property of the system. It is a file path name naming a directory that is used as a default for relative file path names—names without a top level directory reference (or syntactically, a file path name beginning with a colon).

Absolute filename: “`Socrates:Stuff:Folder:myfile`”

Relative filename: “`:Folder:myfile`”

If the `currentDirectory` is “`Socrates:Stuff:`”, the above relative file path name will reference the same file as the above absolute file path name.

Foreign Function Interface

Intro / Warnings

The SK8 **Foreign Function Interface (FFI)** allows you to make calls to functions and procedures that were written, compiled, and linked in other programming environments like C, Pascal, or assembly language. SK8's FFI is a fairly minimal interface. To use it, you must understand several low level concepts for the specific language and computer you are interfacing with, such as: the bitwise representation of the various data structures you will be passing, how parameters are passed on the stack or in registers, and code linking conventions. In general, code you write in SK8 for the FFI will *not* be cross-platform compatible; i.e. it will *not* work on every computer that SK8 may run on (Macintosh, Windows PC, or specific Unix platforms, etc.).

▲ **WARNING**
Users of the FFI have full power and control to corrupt memory, create dangling pointers, cause storage leaks, and otherwise crash the system. If you are not comfortable with low level runtime details of your computer system, you should not use this interface. ▲

Note
The current incarnation of SK8's Foreign Function interface is guaranteed to work only for ATG releases of SK8. Subsequent Applesoft releases may contain an FFI which is significantly different. ◆

Note #2
Most of the examples of foreign code in this chapter use C, with only a few of the examples illustrating how to use Pascal or Mac traps, but extrapolation to these should be reasonably straightforward. ◆

Using the FFI

Here are the steps generally necessary when using the foreign function interface:

1) Compile your foreign functions.

2) Use the `loadForeign()` function to load the foreign functions into your project

3) Write your SK8Script code that uses each foreign function, and put it in a separate script file. These functions are often called the “wrapper” code, because they wrap the foreign code in SK8Script. Since SK8 is an object-oriented environment, you should try to design your wrappers so that they can work inside object handlers, in order to preserve the object-oriented paradigm.

4) Use these wrapper functions, just as you would any other SK8 function.

Compiling your foreign functions

You can write and compile your foreign functions in any programming environment that produces **MPW format object-code** (binary) files (for Pascal or C). For more information, you can refer to the manual for *MPW C, Version 3.0, Appendix C, Calling Conventions and Type Correspondence.*

All of the foreign object files must be linked into one object-code file, because (as of version 0.9) SK8 can only have one object file loaded at a time.

Loading a foreign object file

Once your foreign object-code file has been created, it must be read into the SK8 environment before it can be used. To read an object-code file use the following command:

`loadForeign (`*loadSpec*, *pathName*`)`

loadSpec, the first argument, is the load specifications. It lets SK8 know what the object file looks like. There are the only two load specifications currently supported:

`Mac68KMPWCLoadSpec`

`Mac68KPascalLoadSpec`

As you might surmise from their names, both of these loadSpec arguments specify that the object code uses Motorola MC680x0 instructions and addressing modes. Furthermore, the first *loadSpec* specifies that the instructions use C conventions for passing arguments and return values, whereas the second loadSpec specifies that Pascal conventions were used. Specifically, the C convention is to push the arguments in right to left order, call the function, and when the function returns, the function's caller is expected to remove the arguments from the stack. The Pascal convention, on the other hand, is to push the arguments in left to right order, call the function, and when the function returns, only the result (if any) will be left on the stack. Other loadSpec values may be available in later versions of SK8 (later than version 0.9).

pathName, the second argument, is the location of the object-code file. Currently (i.e. version 0.9) the pathname must be a fully specified OS specific pathname string. For example, on a Macintosh the pathname string might be"Mac HD:SK8:MyCode:Stuff.o"; on a PC the pathname string might be "C:\SK8\MYCODE\STUFF.O". SK8's OS independent physical or logical file names (e.g. "SK8;MyCode:coolStuff.o") cannot be used. This restriction will likely disappear in later versions of SK8 (later than version 0.9).

Some linking loaders only load referenced code rather than loading the entire binary. This is called "searching link libraries" (for referenced code). SK8's FFI, however, does not currently support searching link libraries.

Note
Foreign functions are not saved with your SK8 project; they must always be loaded using the `loadForeign()` function. When you use the project store to save a project which has foreign function calls in it, the foreign function will not be loaded the next time you open that project. You need to guarantee that foreign functions are loaded before any of your code calls them. One approach would be to write an initialize handler for your project that includes a call to the loadForeign function.

Example:

```
on initialize of me (a yourProject)
   do inherited
   loadForeign (Mac68KMPWCLoadSpec, "MacHD:SK8:MyCode:Stuff.o")
   --Then we need to load the wrapper functions...
   loadScriptFile (“SK8;myWrappers.sk8”, me)
end initialize
```

Before we elaborate upon how you can call foreign functions, and how foreign functions can call SK8 handler/functions (a.k.a. Foreign Callins), in the next two sections we introduce foreign data types and foreign memory.

Foreign Data Types

SK8 is able to communicate with foreign functions and memory using a limited set of **simple data types**. In addition, SK8Script allows for the declaration of complex data types (known as **records**) which are built by composing simple data types and other complex data types.

FFI type declarations are SK8Script descriptions that the SK8Script compiler uses to generate machine code. FFI type declarations do *not* instantiate or describe an actual SK8 object. These declarations only cause a SK8 constant to be created with the specified type name.

Simple Types

The simple data types are the following:

- `SInt8, UInt8` (8 bit signed and unsigned integers)

    - Example: `ourIndex (a SInt8)`
- `SInt16, UInt16` (16 bit signed and unsigned integers)
    - Example: `ourIconID (a SInt16)`
- `SInt32, UInt32` (32 bit signed and unsigned integers)
    - Example: `ourOSErr (a SInt32)`
- `Char8` (8 bit extended-ASCII Characters)
    - Example: `ourLetter (a Char8)`
- `MemPointer` (an address in system memory)
    - Example: `ourGrafPtr (a MemPointer)`
- `MemHandle` (an indirect address to memory, a pointer to a MemPointer)
    - Example: `ourWindowHdl (a MemHandle)`
- `MemArray` (an array's base address in system memory)
    - Example: `ourRect (a SInt16 MemArray with numItems 4)`

Complex Types

Complex data types specify a block of physical memory, with individually accessible parts, or components. Access to the components is via a syntax which is identical to accessing properties of a SK8 object. Compound data types can be used as components in other compound data types. Since Compound data types are specified using a universal set of simple data types, their structure can be mapped onto data structures declared in other programming languages.

Form:

`type MemRecord with typeName` *newType* `with fields ¬`

fieldName `([a|an]` *type*`) [,` *fieldName* `([a|an]` *type*`)]*`

where type is a base type (defined above), a memHandle or memPointer to

another type, or a <type> memArray with numItems <n> [<memType>].

Examples:

```
Type MemRecord with typeName "personType" with fields ¬
   name (a Char8 MemArray with numItems 32), ¬
   age (a UInt8), ¬
   badgeID (a UInt32)

Type MemRecord with typeName "departmentType" with fields ¬
   manager (a Char8 MemArray with numItems 32), ¬
   deptNum (a UInt16), ¬
   employee (a personType MemArrya with numItems 10)
```

Corresponding C Code:

```
typedef struct {
   char           name[32];
   unsigned char  age;
   unsigned long  badgeID;
} personType;

typedef struct {
   char           manager[32];
   short          deptNum;
   personType     employee[10];
} departmentType;
```

Foreign Memory

Memory Access

Foreign memory values may be accessed via a specialized expression syntax, which may be used in both setters and getters.

Typed foreignMemory may omit the specific type information, but the dereference will be generated at run-time, will be slower, and will cause some temporary SK8 memory allocation. Giving the full type allows for static, compile-time code generation which is more efficient, but gives up type error checking. Giving partial type information gives fairly quick dereferences which do type check the memory.

Form:

foreign [item <n> [in | of] <field> of <foreignRef> [(a <type> <memType>)]

<memType> ::= memHandle | memPointer | foreignMemory

Use of the `foreignMemory` type (the parent class of both `memHandle` and `memPointer`) indicates that a memory check at runtime should be performed to find if a memory pointer or handle is used.

Similarly to C and Pascal, nested record fields use a dot notation for field names (e.g. person.age).

Examples:

```
--the following line is described in the next section;
--it just allocates foreignMemory which is the right size for
--departmentType
set newDept to newMemHandle(departmentType)

--fully static reference; this works with raw foreignMemory
--and does not type check the memory. (The fastest way.)
set foreign item 1 in manager ¬
   of newDept (a departmentType memHandle) to the character "J"
get foreign item 1 in manager¬
```

```
   of newDept (a departmentType memHandle)

--now, pointer vs handle is decided at runtime; this does a type
--check, but is relatively quick.
set foreign deptNum of newDept (a departmentType foreignMemory) ¬
   to 15
get foreign deptNum of newDept (a departmentType foreignMemory)

--finally, a fully dynamic reference. This takes a relatively long
--time, but does type check.
set foreign deptNum of newDept to 15
get foreign deptNum of newDept
```

Memory Allocation

To actually allocate memory based on a type specification, you use the SK8 functions `newMemPointer()` and `newMemHandle()`. Memory allocated in SK8 via these functions is typed and autodisposed if the object representing the memory is garbage collected.

Getting New Pointers and Handles

The initialize handler for each foreignMemory type can be overridden for customized behavior. When `newMemPointer()` or `newMemHandle()` is called, memory is allocated and the `initializeMem()` handler is called with the typed foreign memory as an argument. `doInherited()` may not be called inside an `initializeMem()` handler.

Form:

```
newMemPointer(<memRecordType>)
newMemHandle(<memRecordType>)
```

Examples:

```
set newPersonPointer to newMemPointer(personType)
set newDeptHandle to newMemHandle(departmentType)

on initializeMem of me (a personType)
   --You can override the handler here if you want to manage
   --your own memory
end initializeMem
```

Disposing of Foreign Memory

Similarly, you may override the default memory deallocator (which SK8 calls when it's releasing your memory). If a `disposeMem()` handler is defined by the programmer, the handler is expected to cause the foreign memory to be deallocated. `disposeMem()` handlers are typically used either to deallocate interior memory references or the call is used to deallocate the memory reference in some non-standard way. Unlike normal SK8 handlers, `doInerited()` may *not* be called inside a `disposeMem()` handler. An example of this second use is to call the foreign deallocator corresponding to the foreign call which allocated the memory initially (e.g. to call `DisposeWindow()` for a reference allocated by `NewWindow()`).

▲ **WARNING**
Default memory deallocation is "precooked" for the Mac. When a memory dispose handler is defined, it replaces the default deallocator. It is the responsibility of the programmer to call the appropriate memory deallocation function inside the dispose handler. ▲

Form:
`disposeMem(<pointerOrHandle>)`

Example:

```
on disposeMem of me (a personType)
   --do some stuff here
end disposeMem
```

Typed foreignMemory is considered to be owned by SK8, which means that it is either disposed when garbage collected by SK8 or a user defined disposeMem handler is called. When typed foreignMemory is passed as an argument to disposeMem, it's deallocator (i.e. it's dispose handler) is invoked if it is owned by SK8. The programmer may change this behavior using the functions `memOwn()` and `memDisown()`. Disowning typed foreignMemory means that the user takes responsibility for deallocation by directly calling a Mac trap routing such as `DisposePtr`, `DisposeHandle`, or `DisposeWindow`. `disposeMem()` has no effect on typed foreignMemory that is unowned.

Note
Owned typed foreignMemory is auto-disposed whenever its object is garbage collected. Therefore you should keep a reference to an owned, typed foreignMemory in SK8 to avoid having it inadvertently deallocated. ◆

Form:

```
memOwn(<typedForeignReference>)
memDisown(<typedForeignReference>)
```

Examples:

```
--Doing this means that SK8 will take care of disposing
--the memory which newPersonPointer references
memOwn(newPersonPointer)

--But now we're responsible for deallocating newDeptHandle
memDisown(newDeptHandle)
```

Converting From raw foreignMemory to typed foreignMemory

To this point, we have been primarily concerned with memory allocated by SK8, known as **typed foreignMemory**. Sometimes foreign function or trap calls will return pointers or handles to memory which they allocated. These references are known as **raw foreignMemory**. A typed foreignMemory reference may be derived from a raw foreignMemory reference using either the `toMemPointer()` or `toMemHandle()` function. Once you've converted raw foreignMemory to typed foreignMemory, you should no longer reference the raw foreignMemory.

Form:

```
toMemPointer(<rawForeignReference>, <typeName>, <own?>)
toMemHandle(<rawForeignReference>, <typeName>, <own?>)
```

Setting <own?> to true means that SK8 owns the memory, and will be

responsible for deallocating it, while setting <own?> to false means that

the programmer is responsible for deallocating the memory.

Examples:

Say we have a foreign function called `NewPersonPointer()` which returns araw foreignMemory pointerto a personType which it allocated, and, similarly, a function called `NewDeptHandle()`, which returns a raw foreignMemory handle to a departmentType. Then, to be able to access the individual fields of each within SK8, we would do this:

```
--Here, SK8 will do the deallocation for us.
set typedPersonPointer to ¬
   toMemPointer(NewPersonPointer(), personType, true)

--Now, however, we're responsible for it ourselves.
set typedDeptHandle to ¬
   toMemHandle(NewDeptHandle(), deptType, false)
```

Foreign Function Calls

When a SK8 function or handler invokes a function written in another language (specifically C, Pascal, or MC680x0 Assembly), it is known as a **foreign function call**, which has the following form:

Form:

```
on <SK8Name> [[of] <name> <argSpec> [, <name> <argSpec>]* ¬
      [returns a <resultSpec>]
   foreign function "<foreignName>" with callSpec
      <callPatternSpec>
end <SK8Name>
```

```
<argSpec>   ::=   ([a|an] <type> <usage> [<register>] <pass>)
<register>  ::=   RegD0 | RegD1 | RegD2 | RegD3 | RegD4 | RegD5 |
                  RegD6 | RegD7 | RegA0 | RegA1 | RegA2 | RegA3 |
                  RegA4 | RegA5
<usage>     ::=   in | out | inOut
```

In arguments are input only, out arguments are results only, and inOut arguments may be side effected by the foreign call.

```
<pass>            ::= byValue | byReference
```

Foreign parameters may be passed byReference or byValue. Arguments that fit in a machine register are typically passed byValue (which copies the argument). Arguments larger than that are usually passed byReference (which passes the address of the argument). Note that foreign memory references are already addresses and so are generally passed byValue (i.e. the value of the address).

Because SK8 objects may safely and freely move about in memory, the SK8 system takes special care when passing values by reference--it makes a copy of the bits in the passed object in a static are of memory and passes the static address. This makes a difference between in and inOut byReference parameters:

- in byReference parameters do not copy the bits back after the foreign call.
- inOut byReference parameters do copy the resulting bits back to their SK8 objects (with some data representation coercions) after the foreign call. This means that if you pass a reference parameter in rather than inOut and side effect it, the result will not be visible to SK8, so use caution.

```
<resultSpec>      ::=   [a|an] <foreignType> [out <register>]
<callPatternSpec> ::=   Mac68KMPWCCallSpec |
                        Mac68KPascalCallSpec |
                        Mac68KTrapCallSpec
```

CallPatternSpec's are used to abstract over common call pattern conventions which are in our case MPW C or Pascal, and MC68K traps. They maybe thought of as information for the compiler's code generator. Currently, there is no supported way for you to extend the supported callPatternSpec's or define your own.

Note

Not all possible <argSpec>'s or <resultSpec>'s are currently supported; a list of supported types occurs at the end of this chapter. ◆

IMPORTANT

Data coercions take place at the interface between SK8 and the foreign code. For example, SK8 integers may have any number of bits, but you can only pass an integer which is representable in a single machine register which, in the case of SK8, is a smallInteger, represented by 29 bits. ▲

Examples:

MacTrap Example:

```
on SysBeep of duration (a SInt16 in byValue)
   foreign function 43464 with callSpec Mac68KTrapCallSpec
   --this is the address of the system beep function
end SysBeep
```

C Example:

```
   C:    Handle Sk8TCPCreate(char *tcpBuffer, long tcpBufferSize,
            char *inputBuffer, long inputBufferSize)
         {
            // Do some stuff in here, then return a long
            return s;
         }

   SK8:  type MemRecord with typeName "tcpCharBuffer" with fields¬
            buff (a Char8 MemArray with numItems 16000)

         on TCPCreate of ¬
               buffer (a tcpCharBuffer MemPointer in byValue),  ¬
               buffSize(a SInt32 in byValue), ¬
               inBuffer (a tcpCharBuffer MemPointer in byValue), ¬
               inBuffSize(a SInt32 in byValue) ¬
               returns a SInt32 out RegD0
            foreign function “Sk8TCPCreate” ¬
               with callSpec Mac68KMPWCCallSpec
         end TCPCreate
```

Pascal Example:

```
   Pascal:  FUNCTION PBOffLine (paramBlock: ParamBlkPtr): OSErr;

   SK8:     on FF_PBOffLine of ¬
                  paramBlock (a RegA0 inOut byValue) ¬
                  returns a SInt32 out RegD0
               foreign function "PBOffLine" ¬
                  with callSpec Mac68KPascalCallSpec
            end FF_PBOffLine
```

Since SK8 currently doesn't allow many things to be passed by reference, interfacing with some functions can be a little tricky. What follows is an example of how to pass longs byReference. Of special note is the fact that you MUST allocate memory for the memRecord which contains the integer.

```
C:    void DoTheRightThing(long *num)
      {
         //Do some stuff to *num
      }

SK8:  type memRecord with typeName “longPtr” with fields ¬
         theLong (a SInt32)
```

```
on DoTheFFIThing of num (a longPtr memPointer in byValue)
   foreign function "DoTheRightThing" ¬
      with callSpec Mac68KMPWCCallSpec
end DoTheFFIThing

set myInteger to newMemPointer(longPtr)
DoTheFFIThing (myInteger)
```

Foreign callins

During a trap or foreign function call, there may be a need for the foreign code to call a function in SK8. You can create SK8 functions, called **callins**, which may be passed (as addresses) to and called from foreign code.

Form:

```
on foreign callin <SK8Name> ¬
   [[of] <name> <argSpec> [, <name> <argSpec>]* ¬
   [returns a <resultSpec>] ¬
   with callSpec <callPatternSpec>
   <just put standard SK8script code here>
end foreign callin <SK8Name>
```

<argSpec>'s currently supported are:

- (a SInt16 in byValue)
- (a SInt32 in byValue)
- (a MemPointer inOut byReference)

<resultSpec>'s currently supported are:

- SInt16
- SInt32
- MemPointer

Example:

Say, for instance, that we wanted to call a C function which takes an integer argument, beeps that number of times, then returns the number plus 2 (which we'll do in the example by incrementing it once in the callin and once in the C function. In the `add2Beep` wrapper, you should particularly notice that the argument `funPtr` is passed as a `T memPointer`, which tells the compiler that it is a generic pointer.

```
C:    long Add1(long n, void (*sk8Function)())
      {
```

```
      sk8Function(n);
      return (n + 1);
   }

SK8:  on foreign callin newBeep of num (a SInt32 in byValue) ¬
            returns a SInt32 with callSpec Mac68kMPWCCallSpec
         beep num
         return (num + 1)
      end foreign callin newBeep

      on add2Beep of num (a SInt32 in byValue), ¬
            funPtr (a T MemPointer in byValue) ¬
            returns a SInt32 out RegD0
         foreign function "Add1" with callSpec Mac68KMPWCCallSpec
      end add2Beep

      add2Beep(5, newBeep)
```

Supported Argument and Result typeSpecs for FFI Calls

Following are the supported argument and result typeSpecs for foreign function calls. The types listed may not be the only ones that work; they are, however, the only ones supported at this time. Unless otherwise noted, arguments are passed on the stack.

Table 2-1 Supported Argument TypeSpecs for Pascal and C Calls

SK8 Type	Foreign Type
Character	a Char8 in byValue
Integer	a SInt16 in byValue
	a SInt32 in byValue
	a RegDx in byValue (reg D0..D7)
Float	an IEEEFloat inOut byReference
	an IEEEFloat as RegAx inOut byReference(reg A0..A4)
MemPointer	a *pointerType* in byValue
	a *pointerType* in RegAx byValue (reg A0..A5)
MemHandle	a *handleType* in byValue
	a *handleType* in RegAx byValue (regA0..A5)
String (*)	a String inOut byReference
	a String inOut RegAx byReference (register A0..A4)
	a String with numItems n inOut byReference
	a CString inOut byReference

Table 2-1 Supported Argument TypeSpecs for Pascal and C Calls

SK8 Type	Foreign Type
	a CString inOut RegAx byReference (reg A0..A4)
	a PString inOut byReference
	a PString inOut RegAx byReference (reg A0..A4)

(*) Using the generic String type tells SK8 to default to the type of string appropriate to the language which you are using (e.g. a null terminated string for C). You may also pass strings as buffers, using the ...with numItems <n> syntax, which allows for foreign code to extend the strings (up to <n> characters). Using this method, trimmed, native SK8 strings will be returned. You may also specify that a SK8 string be coerced to a C string within a Pascal call or vice versa using CString or PString, respectively

.

Table 2-2 Supported Argument TypeSpecs for MacTraps Calls

SK8 Type	Foreign Type
Boolean	a Boolean in byValue
Character	a Char8 in byValue
Integer	a SInt8 in byValue
	a SInt16 in byValue
	a SInt32 in byValue
	a UInt8 in byValue
	a UInt16 in byValue
	a UInt32 in byValue
Float	an IEEEFloat32 inOut byReference
	an IEEEFloat32 as RegAx inOut byReference (reg A0..A4) foreign
memPointer	*pointerType* in byValue
	pointerType in RegAx byValue
memHandle	*handleType* in byValue
	handleType in RegAx byValue
String	a String inOut byReference
	a String inOut in RegAx byReference (register A0..A4)
	a PString inOut byReference
	a PString inOut in RegAx byReference (reg A0..A4)

Table 2-3 Result typeSpecs forPascal and C Calls

Result typeSpec	SK8 Type Returned
Char8	Character
Char8 out RegDx	Character
SInt16	Integer
SInt16 out RegDx	Integer
SInt32	Integer
SInt32 out RegDx	Integer
IEEEFloat	*Float*
type memPointer	*type* memPointer
type memPointer out RegAx	*type* memPointer
type memHandle	*type* memHandle
type memHandle out RegAx	*type* memHandle
String memPointer out RegAx	String
String memHandle out RegAx	String

Table 2-4 Result typeSpecs for MacTraps Calls

Result typeSpec	SK8 Type Returned
Boolean	Boolean
SInt16	Integer
SInt16 out RegDx	Integer
SInt32	Integer
SInt32 out RegDx	Integer
IEEEFloat32	Float
type memPointer	*type* memPointer
type memPointer out RegAx	*type* memPointer
type memHandle	*type* memHandle
type memHandle out RegAx	*type* memHandle
String memPointer out RegAx	String
String memHandle out RegAx	String

CHAPTER 20

Imaging

This chapter is under development. It will appear in the finished manual.

CHAPTER 21

Projects

SK8 provides a framework that allows the user to organize their objects, functions, constants, and variables into one unit, the Project.

This chapter discusses:

- Projects
- Object Store (Project Store)—a mechanism for storing (saving) projects

Projects

A SK8 Project is a workspace for manipulating a collection of objects. You can save, store, and retrieve all the objects in a Project at once. Projects provide a method for organizing all the objects associated with a specific SK8 application into modular workspaces that you can save and load. It is possible to create a Project with the Project Builder or programmatically. Project workspaces are the basis for final applications and titles that you build using SK8.

Not only is a Project a collection of objects, a Project is itself an object.

Superprojects

A project can be part of a superproject. In the following diagram, Project Curly is part of the superproject Stooges.

In real life, your project's ultimate superproject is the SK8 project itself.

Namespaces

A project can not have two objects with the same name. Objects in a project must have unique names, i.e., **namespaces**.

Also, since a project can inherit all objects and their names from a superproject, a project can not have an object with the same name as an object in the superproject. For example, suppose you are developing a project called Foo. Foo is a subproject of SK8. Because there is an object in SK8 called `Rect`, Foo can not have an object named `Rect`.

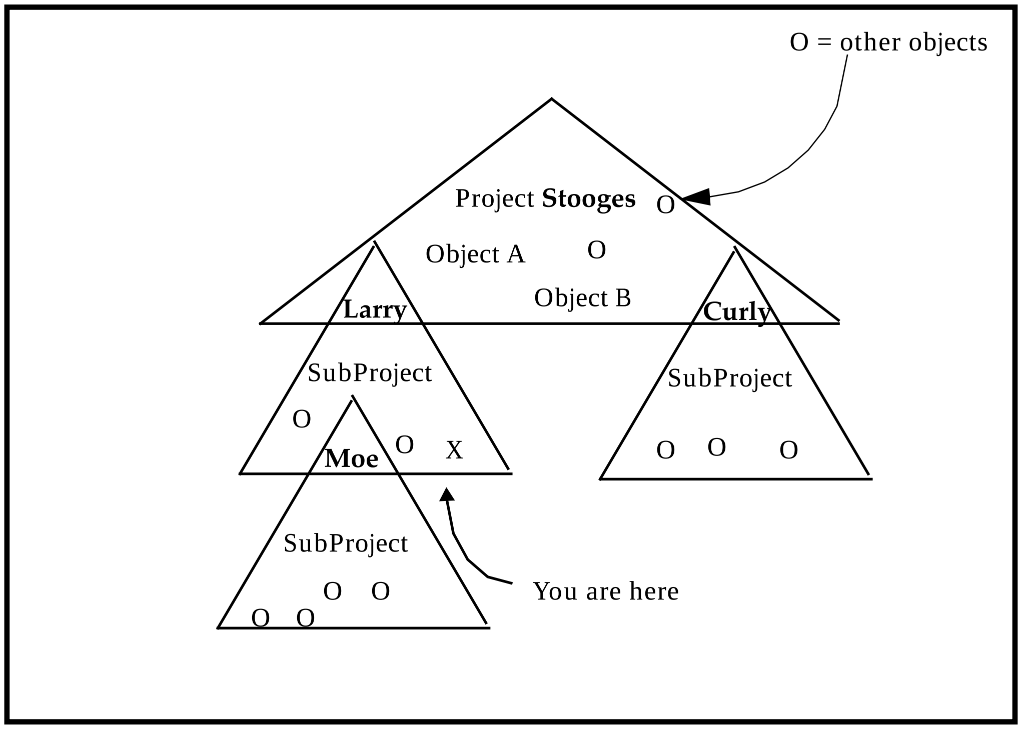
The above example is an over simplification, to explain the concept of namespaces. In real life, things are not this simple. The `publishSymbol` handler, described below, discusses in more detail how object names or symbols are passed to (published) or not passed to (unpublished) subprojects.

SubProjects

A project can be comprised of subProjects. In the following diagram, Projects Larry and Curly are subprojects of Project Stooges. Project Moe is a subproject of Project Larry. Project Stooges and each of the subprojects have their respective objects.

Subprojects are useful to limit the scope of object names, to manage logical groups of objects, etc.

Figure 2-1 Subprojects



Some features of subprojects are best explained with an example:

The first feature is: a project object is **not** in its own project. (Remember that a project is not only a collection of objects but is also an object itself.) In the example above, subprojects Larry and Curly are project objects to Project Stooges and subproject Moe is a project object to subproject Larry. Looking at it from another angle, project object Larry is not a project object of subproject Larry (i.e. itself), project object Curly is not a project object of subproject Curly, and so on.

The second feature is: if you (or an object) are at location X, you can see all the objects in subproject Larry and superproject Stooges, but you can not see or get to the objects in subproject Curly or subproject Moe. In other words, a subproject is not able to see all the objects within a sibling subproject.

Project Handlers

openProject

The `openProject` handler opens a project from a project file.

```
openProject **file** with **copyright**
```

`file`	A file object specifying the file from which the project will be loaded.
`copyright` keyword	A boolean specifying if you want a copyright announcement to be generated at the moment the project completes loading. The default is `false`.

opened Event

The `opened` event is sent to a project immediately after it has been successfully opened by `openProject`. It is possible to specialize an `opened` handler for a project to perform initialization of the project workspace.

openedProject

A system level event sent to the objectSystem. This event can be forwarded to a specific object by inserting that object into the eventListeners property of the objectSystem.

saveProject

The `saveProject` handler saves a project into a file. All objects and imported media are stored/saved in a file.

```
saveProject ourproject with **documentation** ¬
   with **copyright**
```

`documentation keyword`	Provides a documentation string—a description of the project.
`copyright keyword`	Provides for a copyright announcement (encoded as a string).

Supplying a new `documentation` or `copyright` string overrides the previously saved one.

writeSources (Saving Project Source Text Files)

`writeSources` saves the text (SK8Script) sources of a project.

```
writeSources of Project with file ¬
   with objects ¬
   with handlers ¬
   with globals ¬
   with functions
```

`Project`	The project whose sources are to be saved.
`file`	A file object representing the file into which the text will be written. If the file exists, it will be overwritten.
`Objects`	Boolean. If true, objects' sources are stored to a file (default = false).
`Handlers`	Boolean. If true, the handlers for all objects are stored (default = true).
`Globals`	Boolean. If true, variables and constants are stored (default = true).
`Functions`	Boolean. If true, project functions are stored (default = true).

▲ **WARNING**
This feature is supported for the handlers, globals, and functions options. The objects option will provide the best possible written record of the objects in the project. However, it is not guaranteed that evaluating the scripts for these objects will necessarily and satisfactorily reconstitute these objects in memory. You can, however, use the objects option to get a close idea of the script representation of the objects in your project. ◆

compactProject

The `compactProject` handler compacts a project file resulting in a dramatic reduction of disk space needed for storing the project. Project files should be compacted regularly.

```
compactProject project with filename ¬
   with maxVersions ¬
   with media
```

`filename` keyword	The file pathname of the file in which a compacted version of your project is saved. By providing the `filename` keyword, the original project file will not be compacted. Using the `filename` keyword is like using the "Save As..." capability. The compacted file is separate from the original. If the `filename` keyword is not provided, then the file from which the project was loaded is the one compacted. (To check the original file from which a project was loaded, use `get the file of myproject`.)
`maxVersions` keyword	Determines the number of available versions of the source that are preserved in the compacted file. The valid options are `true` or a positive integer. If `true`, all available versions of the source for the project are preserved. An integer indicates the maximum number of versions preserved. The default is 1 (only the currently active version of the source will be preserved).
`media` keyword	Determines if you want the media objects compacted as well. Compacting your media, in the present context, means that if no objects in your project reference that media, then the media will be deleted from the file. For example, you may have imported an image into the project but, after some experimentation, may not be actually referencing that media. Media compaction ensures that the space in disk occupied by the media is reclaimed. The default is `true`.

Publishing Symbols

Publishing symbols is a functionality in SK8 that allows you to publish a symbol to a project's subprojects. What exactly does this mean? First, think of symbol as a name of an object or a reference to an object. (Refer to the SK8 Overview chapter for a more complete explanation of symbol.)

Publish means if an object in a superproject is "published", the object now becomes available to all the subprojects. Referring back to Figure 10-1, "Subprojects", if Object A and Object B in the Project Stooges is published, Object A and B are now available to all subprojects of Stooges. This also means the subprojects **can not have** any local objects named Object A and Object B. (Refer to namespaces at the beginning of this chapter.)

If Object A and B have not been published, the other subprojects can not get to Object A and B. This also means the subprojects **can have** local objects named Object A and Object B.

Publish Handlers

The `Publish` handlers are: `publishSymbol, unpublishSymbols,` and `publishedSymbols.`

publishSymbol

The `publishSymbol` handler publishes a symbol for your subprojects. The only current reason to publish a symbol is when you want to provide handlers or functions that accept a set of symbols as their arguments.

```
publishSymbol Project, Symbol
```

`Project`	The project in which the symbol is published.
`Symbol`	The symbol to be published.

When testing for equality between your symbol and those provided by your subprojects, it is important that the symbols be the same symbol. Even though your subproject may use a symbol with the same name as yours, this does not necessarily mean that the symbols are equal. Since the symbols exist in two different projects, they could be different.

An analogy to illustrate the concept of equality is that of identical twins. Even though they are identical (look the same, same characteristics, etc.), they are two different people.

To ensure that your subprojects use the appropriate symbol, all you need to do is publish the appropriate symbols. For example, suppose you have an object `Foo` in your project and you write a handler called `location` which is written as follows:

```
on set location of me (a Foo) to value
   if value = ‘center’ then beep
end set location
```

In this case, you want to make sure that your subproject’s ‘center’ symbol is the same as the one you’ve used in the IF test above. You can ensure this by executing the following statement:

```
publish ‘center’
```

All subprojects will “subscribe” to this symbol, so that a beep will sound when a statement in your subproject does this:

```
set foo’s location to ‘center’
```

unpublishSymbols

The `unpublishSymbols` handler removes a previously published symbol.

```
unpublishSymbol Project, Symbol
```

publishedSymbols

The `publishedSymbols` handler returns all of the symbols that have been published by a project.

```
publishedSymbols Project
```

Object Store (Project Store)

Object Store or Project Store, in the generic sense, is the mechanism SK8 provides for the saving and re-opening of a SK8 project undergoing the throes of development.

To save a project use the Save command under the File Menu of Project Builder.

When a project is stored or saved, all the objects, handlers, properties, variables, functions, etc. are saved with it. The disk file that contains all the project “stuff” is called the Object Store. For reasons that are obvious, the user is not able to do the usual file manipulation with the Object Store file. An Object Store file contains only one project. Each project has a file object that corresponds to the file that is the project’s Object Store.

When a project is saved everything is written out to the Object Store file. When the project is opened, everything in the Object Store is read into memory and the project’s contents are created.

The Object Store file is in binary format. The negative side of this is that you can not look at the file using an ASCII text editor. At this point in time, the only functionality provided by SK8 in terms of this file is Open and Save.

Objects in a project are a very inter-related, tangled network of connections, pointers, etc., and it is very difficult to store these relationships in a text file. The positive side of the binary format is that it is able to store these relationships and references to objects, internal pointers, etc. Also binary format loads into memory much faster than other file formats.

The Object Store is also used to store handler scripts as temporary files (also in a binary form for all particular purposes). The importance of this is that the handler files can be read from and written to without opening the project.

The only part of a project that is not kept in memory at all times is the SK8Script handlers. When a project is retrieved from Object Store, the entire project is brought into memory, except handlers. Handlers only need to be in memory when executing. This is why handlers are kept in a temporary file until they are read into memory. They are read into memory from the temporary file when they are accessed via the Script Editor. When the Script Editor window is closed, the handler is written back to the temporary file and not kept in memory.

When you open an existing project, the object file is copied to a temporary file in a temporary folder. This is the file used for working storage. When you save the project, all the objects, etc. currently in memory, are written to the temporary file. The old project file is deleted and the temporary file becomes the new project file.

When SK8 aborts, you will see lots of temporary files in your folder. If SK8 is not running, it is okay to get rid of those files because the only way these temporary files can exist (when SK8 is not running) is through an abrupt error that prevented SK8 from performing cleanup and file deletion. If SK8 is running, leave the files alone.

Project files tend to be very small except when using media. It is a developer design decision whether to store media directly in the project or reference the media stored outside the project.

Only objects that are in the project proper are stored in the Object Store file. Any objects outside the project, that are referenced by the project, are stored as a reference. Since references are resolved at load time, the reference must be a valid reference when the project is loaded.

There may be some cases when the user must provide an initialization phase upon opening a project to add specific behavior when restored from store. For example, if the parent project contains an array but the elements of the array are in a child or subproject, some specific initialization needs to occur to link subproject elements to the array in the master/super project.

Media

Media are the objects which connect you with the real world of data. Each media object provides you with properties and handlers that shield you from having to directly manipulate the data. Media objects are typically created by the `Translator` object, which is responsible for exporting SK8 objects as external platform-specific media and visa versa.

For example, the `SoundToSndTranslator` is used to import Macintosh ‘snd’ resources into SK8 sound objects. It is also used to export Sound objects as ‘snd’ resources.

Media

Media includes all objects that represent platform-specific media. The term media refers to formatted data which is the basis, for example, for images and sampled sounds. These include objects which represent resources in the Macintosh, QuickTime™ movie files, PICT and Window DIB graphics files, and so on.

Resource (a Media)

This is the object from which all Macintosh resource objects inherit.

All Macintosh files consist of two main segments: the resource fork and the data fork. The resource fork is the collection of objects available to the data fork. Icons, sounds, cursor styles, PICT images, and dialog box items are just some of the resources that can be added to any Macintosh file. ResEdit, for example, is designed for adding, modifying, and deleting a file’s resources.

In some types of files, the resource fork is empty. A text file, for example, is made up only of a data fork that stores all the text. By contrast, a word processing file that displays several PICT graphics will contain the file’s text in its data fork and the PICTs in its resource fork.

In SK8, the `resource` fork is contained by the `Resource` object, which is a child of Media. The Resource object is the parent of the following built-in SK8 objects:

QDPicture	A resource whose type is ‘PICT’ and is used for Macintosh pictures. Renderers that use pictures require a child of this resource in their frame property. QDPicture resources have a `boundsRect` and a `size` property.
CursorRSRC	A resource whose type is ‘CRSR’ and is used to display the image of the mouse pointer. Your can set the `cursor` of the `Stage` to one of these objects. SK8 comes with over a dozen built-in cursors. (To see the names of each, type `the child of Cursor` into the Message Box.) Cursors have a `boundsRect` and a `size` property.
IconRSRC	A resource whose type is ‘CICN’ that is used for Macintosh icons. Renderers and actors that use icons will expect a child of this resource. SK8 includes icon resources for its built-in interface objects. Examples: arrow icons for Scrollers, button icons for its dialog boxes, and so forth.
SoundRSRC	A resource whose type is ‘SND’ that stores all information necessary to play a given sound. These objects have a play handler that plays the sound. Up to four different sounds can be played at once. Sound objects also have a `duration` property.
BWPattern	A resource whose type is ‘PAT’ that is used for Macintosh black and white patterns. Patterns are 8x8 images that are used to fill in graphic objects, and were widely used through the control panel to create desktop patterns in older black and white McIntoshes. SK8 includes five built-in BWPattern objects: BlackPatternRSRC, DarkGrayPatternRSRC, GrayPatternRSRC, LightGrayPatternRSRC, and WhitePatternRSRC. The SK8 user interface defines 45 others. For example, BWPattern objects can be passed to renderers that want a penPattern. BWPatterns have a `boundsRect` and a `size` property, but they are always 8x8.
ColorPattern	A resource whose type is ‘PPAT’ and is used to create color patterns. Like BWPatterns, ColorPatterns can be passed to renderers that require a penPattern. ColorPattern objects have a `boundsRect` and a `size` property, but they are always 8x8.

To add one of the above resources, you make a child of the appropriate Resource object.

Note

Resource is not documented, but it is available for use. Use the SK8 User Interface (Project Builder) to get more information on the descendants of Resource. ◆

Menus

This section describes how menus are created, installed and used via SK8Script.

Note

The Project Builder also provides the capability of menu creation and installation by direct manipulation. Refer to the Menu Editor section in the Project Builder chapter for more information. ◆

Setting up a Menubar

Creating a Menubar

To create a menubar with three menus, enter the following script:

```
new menubar with objectname “ourMenubar”
```

Adding Menus

```
new menu with objectname “firstMenu” with text “first”
new menu with objectname “secondMenu” with text “second”
new menu with objectname “thirdMenu” with text “third”
```

To install the menus into the menubar, you have two options:

- setting the menubar property of each menu, or
- setting the menus property of the menubar.

Using the second method, we can write the following script:

```
set ourMenubar’s menus to {firstMenu, secondMenu, thirdMenu}
```

Adding MenuItems

MenuItems are installed into the menu by setting their menu property or by setting the menuItems property of the menu.

Enter this script:

```
new menuItem with objectName “mit1” with text “Menu Item 1”
new menuitem with objectName “mit2” with text “Menu Item 2”
new menuItem with objectName “mit3” with text “Menu Item 3”
set mit1’s menu to firstMenu
set mit2’s menu to firstMenu
set mit3’s menu to firstMenu
```

Now we’re ready to test our menubar!

Installing Menus and Menubars

A SK8 menubar can be placed in one of two locations:

- on the Stage, in which case it becomes the Macintosh menubar, or
- on another actor.

To install the menubar on the Stage, you can use the `currentMenubar` property of the Stage as in the following line.

```
set the currentMenubar of the stage to ourMenubar
```

Place a rectangle on the Stage and simply call it `theRect`.

To install the menubar into it, set the menubar’s container to the actor, as shown in the following example. Incidentally, this is how the SK8 User Interface Project gets menubars placed on actors such as the SK8 Object Browser.

```
set the container of ourMenubar to theRect
```

A menubar can’t be in two places at the same time, so the command we have just executed removes the menubar from the Stage. (The same result is achieved by setting the `currentMenubar` of the Stage to `False`.) Similarly, setting the `currentMenubar` of the Stage to `ourmenubar` would remove `ourMenubar` from `theRect`.

In the previous examples, menus installed in a menubar behave as pull down menus.

A popup menu, by contrast, is a menu that is not installed in a menubar. Instead it is contained by any SK8 actor. To make the first menu in `ourMenubar` (`firstMenu`) a pop up menu, set its container to an actor (`theRect`, for example), as follows:

```
set the container of firstMenu to theRect
```

Connecting a Menu to a Handler

Each time an item is selected from a menu item a handler called `menuSelect` is called.

```
on menuSelect of me (a mit1)
   beep
end menuSelect
```

Be sure to call `do inherited` from your `menuSelect` handler to get it to perform the behavior that you want.

Object

This chapter is under development. It will appear in the finished manual.

Pickers

This chapter is under development. It will appear in the finished manual.

Ports

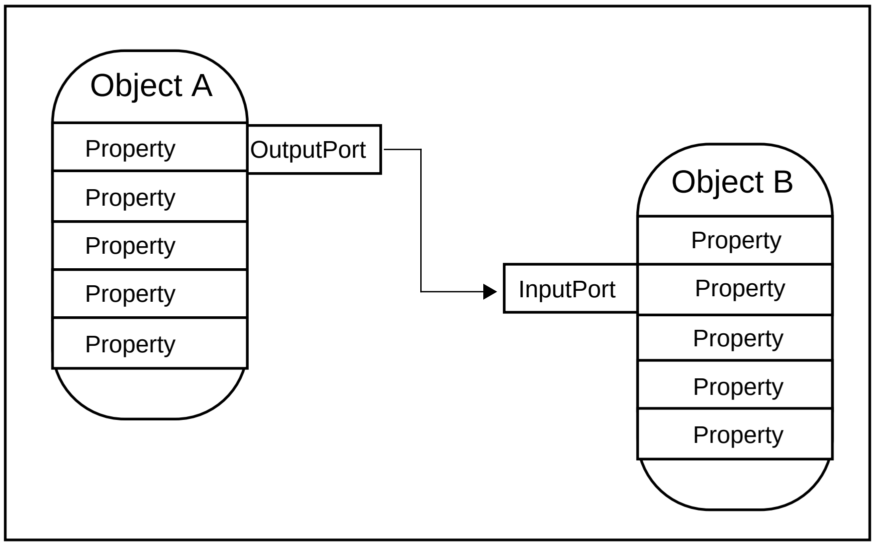
Ports provide a covenient and dynamic to allow properties to be interconnected. They are described in detail below.

Introduction

It is often desirable that changing a property’s value causes side effects. This could be done in SK8 by overriding the property’s setter handler. But if those side effects are complex, or cannot be predetermined, it can be difficult to design and maintain the side effect code.

Ports formalize side effects, in a convenient and dynamic way. Ports “attach” to the properties of objects. Two or more ports can be “wired” together. When the value of the property changes, the attached port broadcasts a message to all the ports wired to the receiving ports and affect a change to their attached property. Ports can easily be re-wired (i.e. disconnected and connected) dynamically, during runtime

.An illustration of Ports



(For nomenclature: “attach” refers to associating a port to a property; “wired” refers to establishing a communication link between ports.)

Ports are attached to the properties of objects. There is no limit to the number of ports attached to any particular property.

Ports are attached to the properties of individual instances of an object. They are not inherited. If you need all the children of an object to have similar ports, you will need to override the object’s initialize routine to add the port when the child is instantiated.

The Port Object

Types of Ports

Three types of `port` objects exist:

- output,
- input, and
- inputOutput.

An output port can be connected to any number of input ports. An input port can be connected to any number of output ports. An InputOutput port can be connected to any number of InputOutput ports. InputOutput ports cannot be connected to input ports nor output ports.

Note the “Port Wiring Mode” a Project Builder graphic interface for creating ports. It uses Connector objects (lines) to graphically simulate ports, allowing you to “draw connections between actors. To use this mode, use the “Enter Port Wiring Mode” command in Project Builder’s Project’s Workspace menu

If you need more elaborate side-effects that just copying the value of a property from one object to the property of another, you should override the `ActiveOutputPort`, or `ActivateInputPort`, handler for your port. These handlers are used for “filtering” the data flowing from the output to the input.

Port Handlers

Below are a few examples of the handlers of the Port object. Examine a port object with the SK8 Object Editor and refer to the SK8 Object Reference Manual for more detailed descriptions.

AddOutputPort

To create and attach an Output port to an object’s property you can use the `AddOutputPort` handler of Object.

Example:

```
new rectangle with objectName “source” with container stage ¬
   with boundsrect {100,50,200,150}
addOuputPort source, ‘fillcolor’ with objectname “colorOut”
```

AddInputPort

To create and attach an input port to an object’s property you can use the `AddInputPort` handler of Object.

Example:

```
new rectangle with objectName “target” with container stage ¬
   with boundsrect {250,50,350,150}
addInputPort source, ‘fillcolor’ with objectname “colorIn”
```

AddInputOutPort

To create and attach an inputOutput port to an object’s property you can use the `AddInputOutPort` handler.

Example:

```
new Oval with objectName “Ying” with container stage ¬
   with boundsrect {100,150,200,250}
addInputOutPort Ying, ‘fillcolor’ with objectname “colorYing”
new Oval with objectName “Yang” with container stage ¬
   with boundsrect {250,150,350,250}
addInputOutPort Yang, ‘fillcolor’ with objectname “colorYang”
```

attachPort

If a port object already exists, you can attach it to an object's property with the `attachPort` handler.

Example:

```
new port with objectname “frmKolor”
attachPort frmKolor with obj target with property ‘framecolor’
```

wirePorts

Ports are wired together using the `wirePorts` handler of the OutputPort and InputOutputPort objects.

Example:

```
wirePorts colorOut,colorIn
wirePorts colorYing,colorYang
```

unwirePorts

The communication link between any two ports can be removed by using the `unwirePorts` handler of the OutputPort and InputOutputPort objects.

Example:

```
unwirePorts colorOut,colorIn
wirePorts colorYing,colorYang
```

unwirePort

All communication links associated with a port can be severed with the `unwirePort` handler of the OutputPort and InputOutputPort objects.

Example:

```
unwirePorts colorIn
wirePorts colorYang
```

wiredTo

`wiredTo` returns a list of all the ports this port is wired to.

Example:

```
wiredtTo colorIn
wiredTo colorYang
```

ActivateOutputPort

Every time the value of a property changes, and an output port, or inputOuputPort is attached to that property, ActivateOutputPort gets called. To filter the value your port broadcasts to its wired ports you must override the ActivateOutputPort handler. The parameters passed by this are “oldValue” representing the attached properties original value, and “newValue” representing the attached property’s new value.

If you need `ActivateOutputPort` to be called before the attached property’s value is changed, set the “triggerBefore” property of the port to `True`.

Example:

```
on ActivateOutputPort of me (a colorOut), oldValue, newValue
   -- do not allow color to be set to same color
   repeat while oldValue = newValue
      set newValue to any of the knownchildren of RGBColor
   end repeat
   do inherited(me, oldValue, newValue)  -- very important!
      -- if you do not call do inherited with parameters
      -- it will send original values of parameters
end ActivateOutputPort
```

ActivateInputPort

Every time the property associated with the port changes, `ActivateInputPort` is called, assuming an inputPort or inputOutputPort is attached to property. To filter the value your port receives, you must override the `ActivateOutputPort` handler. The parameters passed to this handler are “oldValue” and “newValue” representing the value change that has taken place on the output (source) side of the connection.

Example:

```
on ActivateInputPort of me (a colorIn), oldValue, newValue
   if newValue = Green
      set newValue to Blue
   if oldValue = Red
      set newValue to Green
   do inherited(me, oldValue, newValue)  -- very important!
      -- if you do not call do inherited with parameters
      -- it will send original values of parameters
end ActivateInputPort
```

C H A P T E R 2 7

QuickTime^{TM}

QuickTime^{TM} is a major application of multimedia technology. It allows realtime playback of composed video on a computer screen. SK8 allows the playing of QuickTime^{TM} movies and provides the ability to create and modify movies.

This chapter describes SK8 support of QuickTime^{TM}.

QuickTime Objects in SK8

QuickTime™ support in SK8 starts with the `QuickTimeMovie, QuickTimeRenderer, and MovieRectangle` objects.

The `QuickTimeMovie` object provides an abstract object-oriented interface into QuickTime™ movies.

The `QuickTimeRenderer` object provides an extended set of capabilities, including playing a movie.

The `MovieRectangle` object is a Black Rectangle with an `initialize` handler that sets each new child's `fillColor` to a new child of `QuickTimeRenderer`. `MovieRectangle` is on the standard palette. After you make a new `MovieRectangle`, set its `fillColor's` media to the movie you want or to true (which will bring up a QuickTime open file dialog).

One other important way to use QuickTime in SK8 is to start with an actor and call its `moviefy` handler. This sets the actor's `fillColor` to a new child of `QuickTimeRenderer` with its media set to the movie you choose from a QuickTime open file dialog which will come up. The movie will render into the shape of the actor, whatever it is.

A `QuickTimeRenderer` can only render one actor at a time. If a `QuickTimeRenderer` is already rendering actor A, and you set actor B's `fillColor` to the same `QuickTimeRenderer`, the renderer will be yanked away from actor A and given to actor B. In this case, the renderer for the region of actor A that was being rendered reverts to the current setting of the `QuickTimeRenderer’s backgroundColor` property.

How to Play a QuickTime Movie

This is one example of how to play a QuickTime™ movie in SK8.

1. Create an Actor to play the movie in:

```
new Rectangle ¬
   with objectName "screen" ¬
   with container Stage
```

2. Create a renderer that provides handlers for manipulating a movie and set your actor’s `fillColor` to this renderer:

```
new QuickTimeRenderer ¬
   with objectName "player"
set the screen’s fillColor to the player
```

3. Create an object representing a movie file:

```
new QuickTimeMovie ¬
   with resourceFile true ¬
   with objectName "movie1"
```

Selecting `resourceFile true` displays an open dialog. Canceling out of the dialog causes an error and aborts the creation of the new object.

4. Tell the renderer which movie you want to manipulate:

```
set player’s media to movie1
```

This has a useful side effect in that it adds `movie1` to the parents of player. This means that the movie can be manipulated through the player object, using properties and handlers of `QuickTimeMovie` as well as `QuickTimeRenderer.` This means there is more functionality available to you for manipulating the movie.

While a movie is associated with a renderer in this way, you can access it by referencing either the renderer or the movie. The right thing happens whether you do something to the renderer that should affect the movie, or you do something to the movie that should affect the renderer.

Moviefy of Actor

Examples:

```
moviefy Actor

moviefy Actor ¬
   with movie (a QuickTimeMovie) -- or true (the default)

moviefy Actor ¬
   with file (a File)

moviefy Actor ¬
   with logicalName (a string)

moviefy Actor ¬
   without movie
```

A new `QuickTimeRenderer` is created and the `fillColor` of the actor is set to the new renderer. All of the examples, except the last, set the media of the new renderer to a QuickTimeMovie. The renderer is immediately usable for playing movies when its media is set.

If `with movie true` or `with file true` is specified, a QuickTime open file dialog is displayed. The media can be set via the dialog box.

It is an error to specify more than one of `movie, file,` or `logicalName`.

Once you have "moviefied" the actor, you can change or set the movie to be played by doing either one of the following to the actor's `fillColor`:

```
set funMovie's media to true
```

This will bring up an open file dialog for specifiying the movie file.

```
set funMovie's media to someMovie
```

<where>

`someMovie` is a QuickTimeMovie

When the `QuickTimeRenderer` is stopped, the previous setting of the actor's `fillColor` will be rendered. You can change color this by setting the actor's `fillColor's backgroundColor`.

Creating QuickTimeMovie Objects

Examples of SK8Script code for creating a new `QuickTimeMovie` Object follows:

Create a new QuickTimeMovie Object referencing an existing move resource file

```
new QuickTimeMovie
   with resourceFile foo
   with resourceID (defaulting to 0)
   -- 0 means use the ID of the first usable
   -- resource found in the file
   [ with resourceName ]
   -- find the resource by name rather than by number
   -- false means we don't care about
   -- the resource name for the movie
```

Note
See *Inside Macintosh QuickTime™, "*"NewMovieFromFile". ◆

Create a new QuickTimeMovie Object from a file with the movie data stored in the data fork

This alternative is useful for movie files not created on a Macintosh.

```
new QuickTimeMovie ¬
   with dataFile foo ¬
   with fileOffset (defaulting to 0)
```

Note
See *Inside Macintosh QuickTime,* "NewMovieFromDataFork". ◆

Creating a new QuickTimeMovie Object from a file that may have the movie data stored in either a movie resource or in the data fork

If there is a movie resource in the file, it is used in preference to a movie in the data fork.

```
new QuickTimeMovie with file foo ¬
```

Creating a new QuickTimeMovie Object from the clipboard

```
new QuickTimeMovie ¬
   with clipBoard ¬
```

Note
See *Inside Macintosh QuickTime,* “NewMovieFromScrap”. ◆

Creating a new QuickTimeMovie Object from scratch, in memory

This will be more useful when QuickTime movie editing is fully supported.

```
new QuickTimeMovie
```

Note
See *Inside Macintosh QuickTime,* “NewMovie”. ◆

QuickTime Commands

This section provides a quick reference to QuickTime commands. For some commands, a reference is make to the Inside Macintosh QuickTime™ (IMQT) manual. Text stating “Consult the IMQT” is referring to this manual.

`cutSelection` of QuickTimeMovie

Returns a new QuickTimeMovie each time it is called, allowing one to create a **cut** from the current selection of the movie.

`copySelection` of QuickTimeMovie

Returns a new QuickTimeMovie each time it is called, allowing one to create a **copy** from the current selection of the movie.

`gotoBeginning` of QuickTimeRenderer

Rewinds movie to the beginning of the active segment.

`gotoEnd` of QuickTimeRenderer

Forwards movie to the end of the active segment.

`loadIntoRam` of QuickTimeMovie -- with options

Allows loading of a movie (or as much as wanted) into RAM. Returns `true` if entire movie fits in memory, otherwise `false.`

`nextInterestingTime` of QuickTimeMovie -- with options

Allows forward or backward navigation in a movie by finding the "interesting times". Sets `timeValue` to the new time value and returns it. Consult the *Inside Macintosh QuickTime* manual.

Supported keywords are:

with startTime

Defaults to the movie's `timeValue.`

with rate

Defaults to the movie's `preferredRate.`

with mediaSample

Defaults to `false.`

with mediaEdit

Defaults to `false.`

with trackEdit

Defaults to `false.`

with syncSample

Defaults to `false.`

with edgeOK

Defaults to `false.`

with ignoreActiveSegment

Defaults to `false.`

with mediaTypes

Defaults to `{"eyes"}.`

with interestingDurationWanted

Defaults to `false.`

`pasteSelection` of QuickTimeMovie

Allows the pasting of the current selection of a movie into another movie.

Example:

```
pasteSelection toMovie, fromMovie
```

`pause` QuickTimeRenderer

Leaves current frame visible.

`play` QuickTimeRenderer

Plays from current position.

`play` QuickTimeRenderer with noOtherActivity

Plays without any other event processing.

`preroll` of QuickTimeMovie

Allows pre-rolling of movie before it actually plays. The supported keywords are:

startTime	Defaults to movie's current `timeValue.`
rate	Defaults to movie's `preferredRate.`

`start` QuickTimeRenderer

Plays from beginning of the active segment.

`start` QuickTimeRendererwith noOtherActivity

Plays without any other event processing.

`step of` QuickTimeRenderer

Steps through video samples and returns the number of steps accomplished. Returns `negative` if it went backwards. See `nextInterestingTime` of QuickTimeMovie.

`step player`

Goes to next sample.

`step player by: 2`

Steps 2 samples.

`step player by:-3`

Steps backwards 3 samples.

`stop` QuickTimeRenderer

Redraws the actor using the `backgroundRenderer` of the `QuickTimeRenderer.`

QuickTime™ Properties

QuickTimeRenderer Object

`actorOwner` of QuickTimeRenderer

The `actorOwner` is the actor whose `fillColor`, `frameColor`, or `textColor` we are rendering. If this property is `false`, you can still play the movie and only hear the sound tracks.

`displaying` of QuickTimeRenderer

false	Makes the renderer invisible (sound will still be heard when playing).
true	Make it visible.

`media` of QuickTimeRenderer

The `media` property specifies the movie being played. Change the movie by changing the media of the renderer. Set the `media` of the renderer to `false` if you want to have no movie associated with the renderer (the `backgroundColor` of the renderer will be rendered in the renderer's `actorOwner`).

`newStateWhenDone` of QuickTimeRenderer

The `newStateWhenDone` property controls the state of the renderer after it is finished playing. Legal values are:

'previous' 'inactive' 'poster' 'stopped' 'paused'

The 'previous' option refers to the state that existed before the movie played. Refer to `state` property of `QuickTimeRenderer` for more information.

`playWithNoOtherActivity` of QuickTimeRenderer

The `playWithNoOtherActivity` property, if set to `true,` allows the renderer to play its movies without any other event processing in the system. The default is `false`. This may improve the playback performance of your movie, at the expense other system activity. You will not be able to control your movie when in this mode. Do not use this mode when your movie's `repeating` property is set to 'loop' or 'palindrome.'

`rate` of QuickTimeRenderer

Changes the rate of the renderer. You can set the rate to a positive or negative number, in which case the `state` of the renderer is set to 'playing'. Setting it to 1 causes it to play at normal speed. Setting it to 0 is equivalent to pausing the renderer.

`resizingStyle` of QuickTimeRenderer

The `resizingStyle` of the renderer determines what happens when the renderer gets a new media or is set to render one of the regions of an actor. The valid `resizingStyle` values are:

'resizeMovieFromRenderedRegion'

'resizeMovieFromFill'

'resizeMovieFromFrame'

'resizeRenderedRegionFromMovie'

'resizeFillFromMovie'

'resizeFrameFromMovie'

The default is `false`.

`state` of QuickTimeRenderer

A renderer's state can be changed by setting its `state` property to one of the following states:

'playing'	Playing.
'pause'	Current timeValue is displayed.
'poster'	Movie's poster is displayed.
'stopped'	Stopped, backgroundColor rendered.
'inactive'	Active of movie is false.

QuickTimeMovie Object

`active` of QuickTimeMovie

A virtual property stored in the child of QuickTimeMovie. This property is rather esoteric, as it is set automatically when you pause or play the movie. See *Inside Macintosh QuickTime*.

`activeSegment` of QuickTimeMovie

A virtual property stored in the child of QuickTimeMovie. Allows specification of the active segment of the movie. The segment is specified as an interval, which is a list of two integers `{timeValue, duration}`. This will be the part of the movie that actually plays. For example:

```
set the activeSegment of m to {10, 60}
```

If the interval list is `false` or `{false, false}`, then the selection is the entire movie. If `timeValue` is `false` `{false, 120}`, then `timeValue` is assumed to be the current `timeValue`. If the duration is `false {60, false}` or `not present {60}`, then the rest of the movie (forward or reverse as indicated by `preferredRate`) is assumed.

`couldNotResolveDataReference` of QuickTimeMovie

This property is `true` if the Movie Toolbox was unable to resolve all the data references when creating the SK8 object. It does not have a setter. It is set when the `QuickTimeMovie` object is initialized.

`dataReferenceWasChanged` of QuickTimeMovie

This property is `true` if the Movie Toolbox had to change any data references while resolving them when creating the SK8 object. It does not have a setter. It is set when the `QuickTimeMovie` object is initialized.

`datasize` of QuickTimeMovie

A virtual property stored in the child of `QuickTimeMovie`. See *Inside Macintosh QuickTime*. It does not have a setter. Its value is retrieved from the media.

`duration` of QuickTimeMovie

A virtual property is stored in the child of `QuickTimeMovie`. A non-negative integer specifying the duration of the movie. Duration is a number of time units. Time units are specified by the movie's time scale. (See *Inside Macintosh QuickTime.*) It does not have a setter. Its value is retrieved from the media and can be changed by editing the movie.

`file` of QuickTimeMovie

Stored in the child of `QuickTimeMovie`. Initially, this is the file the movie was created from, if any. This is a `File` object. This property does not have a setter. It is set when the `QuickTimeMovie` object is initialized.

`fileOffset` of QuickTimeMovie

Contains a number if the movie was created from a data fork-only movie file (such as one would get from a Window's machine movie authoring application). It does not have a setter. It is set when the `QuickTimeMovie` object is initialized.

`moviePictGCHandle` of QuickTimeMovie

A high-quality PICT of the current frame (interpolated). This property does not have a setter because it is derived from the movie data.

`preferredRate` of QuickTimeMovie

A virtual property stored in the child of `QuickTimeMovie`. Allows changing the rate of the movie the next time it plays.

`preferredVolume` of QuickTimeMovie

A virtual property stored in the child of `QuickTimeMovie`. It allows changing the volume for the movie the first time it plays after being loaded. This should be a number between 0 (no sound) and 1 (loudest). You can set this to a larger value to amplify a quiet movie, but then loud sounds will distort by "clipping". To temporarily turn sound off, but still remember the normal value, you can negate the normal value.

`rate` of QuickTimeMovie

A virtual property stored in the child of `QuickTimeMovie`. A number one (1) means normal speed forward, minus one (-1) means normal speed backwards. It does not have a setter.

`renderer` of QuickTimeMovie

Specifies the `QuickTimeRenderer` this movie is currently bound to, if any. A `QuickTimeMovie` can only be bound to one `QuickTimeRenderer` at a time. If a QuickTimeMovie is already being rendered by QuickTimeRenderer A, and you set QuickTimeRenderer B's media to the same QuickTimeMovie, the movie will be yanked away from its renderer A and given to renderer B.This property does not have a setter.

`repeating` of QuickTimeMovie

The `repeating` property is a virtual property stored in the movie object. The `repeating` property can take the following values:

'loop'	Again and again.
'palindrome'	Back and forth.
'false'	No looping.

`resourceID` of QuickTimeMovie

Contains an integer if the movie was created from a resource file. It does not have a setter. It is set when the `QuickTimeMovie` object is initialized.

`resourceName` of QuickTimeMovie

Sets to a string if the movie was created from a resource file by specifying the resource name. It does not have a setter. It is set when the `QuickTimeMovie` object is initialized.

`selection` of QuickTimeMovie

A virtual property stored in the child of `QuickTimeMovie`. Determines what part of the movie is selected for editing operations. (See `copySelection, cutSelection` and `pasteSelection`.) A selection is specified as an interval, which is a list of two integers `{timeValue, duration}` (see `activeSegment` of `QuickTimeMovie`). For example,

```
set the selection of m to {60,120}
```

`timeScale` of QuickTimeMovie

A virtual property stored in the child of `QuickTimeMovie`. Allows changing the time scale of the movie's time base. See *Inside Macintosh QuickTime*.

`timeValue` of QuickTimeMovie

A virtual property stored in the child of `QuickTimeMovie`. Allows changing of the current position in the movie. It functions even while the movie is playing. The following example rewinds the movie to its beginning.

```
set the timeValue of m to 0
```

`volume` of QuickTimeMovie

A virtual property stored in the child of `QuickTimeMovie`. It allows you to change the volume of the movie. This should be a number between 0 (no sound) and 1 (loudest). You can set this to a larger value to amplify a quiet movie, but loud sounds will distort by "clipping". To temporarily turn sound off, but still remember the normal value, you can negate the normal value.

Renderers

Introduction

Renderers simplify the SK8 programmer's color model. Instead of having to assign many options as to how an actor should be rendered (e.g., how much red, blue and green should go into the forecolor and backcolor, what patterns and copying behavior to use), the programmer need only assign a Renderer object to one of the three areas of an actor (i.e., the fill, frame and text areas).

A descendant of `Renderer` may be assigned to any of the three subareas of an actor (e.g., the fillColor) to render that area. The job of `Renderer` is to accept an actor and a mask (`Mask` is described below) and to draw into the area designated by that mask. The SK8 graphics engine takes care of all the bookkeeping so that the renderer's only concern is to paint the designated mask.

There is a wide variety of renderers supplied with SK8. In addition, since renderers are objects, you can extend their functionality or implement primitives.

A renderer may be used as many times as you wish to render any number of actors in a project. If you wish to have a slightly modified version of an already defined renderer, you can create a child of that renderer and change the appropriate properties to create the desired effect. When you create your own renderers (or children of existing renderers), these renderers will be saved with your project.

CHAPTER 29

Shapes and Lines

This chapter is under development. It will appear in the finished manual.

System and Devices

SK8 provides a set of objects which represent system components in an abstract, platform-independent way.

Devices

`Device` supports all kinds of devices usually attached to the computer in which SK8 is currently running. SK8 monitors the operating environment as it loads and dynamically as it runs for changes in device configurations. Many types of devices in your environment are represented by SK8 as descendants of `Device`.

The following devices are currently supported by SK8:

Monitors	All monitor screens
Storage devices	All storage devices, including CD players and cassette recorders
Keyboards	Example: an Extended keyboard on a Macintosh
Printers	
Pointers	Example: a mouse
Modems	
Audio Channels	Example: synthesizers

Monitors

Inquiries regarding which monitors are currently installed in your operating environment may occur in two ways:

- ask the `System` object for its monitors:

```
get the monitors of the System
```

- ask the children of `Monitor` for information:

```
get the children of Monitor
set myMonitor to Item1 in the monitor of system
if myMonitor is mainMonitor then ...
```

`mainMonitor`	Returns `true` if the monitor is the main monitor. In the Macintosh, this is the monitor in which the menubar always appears.
`active`	Returns `true` whenever the monitor is the active monitor. `if myMonitor is active then ...`
`colorDepth`	Returns the maximum color depth supported by the monitor (e.g., 32 bits). The `colorDepth` is a number. This should not be confused with the `depth` of a window actor. The `colorDepth` of a monitor refers to the depth of a physical device. `get the colorDepth of myMonitor`
`color`	Returns `true` if the monitor supports color. `if myMonitor is color then ....`
`location`	Returns a list with the `h` and `v` location of the top-left corner of the monitor with respect to the Stage. In the Macintosh, the Stage corresponds to the Desktop. `get the location of myMonitor`
`size`	Returns a list with the width and height of the monitor in pixels. `get the size of myMonitor`
`boundsRect`	Returns a list with the left, top, right and bottom points of the monitor in the Stage. `get the boundsRect of myMonitor`

Storage Device

The `storageDevice` object's children are all secondary storage devices attached to the system. Generally, these are disks, although they may also be CD-ROM players and other input/output devices.

You may get all children of `storageDevice` by asking for the children of `storageDevice`:

```
get the storage devices of the System
```

The following capabilities are supported:

`name`	Returns a string representing the name of the storage device. In the Macintosh, this is the name of your volume as shown in the Finder. `get the name of myStorageDevice`
`number`	Returns the volume number of the storage device. `get the number of myStorageDevice`
`drive`	Returns the drive number of the storage device `get the drive of myStorageDevice`
`ejected`	Returns true if the storage device has been ejected. `if myStorageDevice is ejected then ....`
`flush`	Flushes all buffers to the storage device. `flush myStorageDevice`
`unmount`	Unmounts the storage device. `unmount myStorageDevice`

Keyboard

You can query the children of `Keyboard` or ask for the keyboards of the System to get all keyboards installed in your system environment. The following capabilities are supported:

`keysDown`	Returns false or a list of keys which are currently depressed. This does not return any modifier keys. `get the keysDown of myKeyboard`
`getKeys`	Returns a list of all the key characters current depressed in the main keyboard. `set x to getKeys()`
`shiftKeyDown`	Returns true if the shift key is currently down. `if shiftKeyDown() then ...`
`optionKeyDown`	Returns true if the option key is currently down `if optionKeyDown() then ...`
`commandKeyDown`	Returns true if the command key is currently down.

```
if commandKeyDown() then ...
```

`controlKeyDown`	Returns true if the control key is currently down. `if controlKeyDown() then ...`
`capsKeyDown`	Returns true if the caps key is currently down. `if capsKeyDown() then ...`

Printer

This object represents the currently selected printer.

Pointer

Children of this object are the pointers currently connected to your system. Pointer devices include the mouse or pen tablets.

Modem

Children of this object are the modems (e.g., fax) currently connected to your system.

Cursor

`Cursor` is the object used to render the visual views corresponding to pointing devices.

Color Cursors

A type of resource which behaves in a transparent way. You can set the cursor of the Stage to any type of cursor.

Animated Cursors

These are clocks with a property that contains a list of cursors. You use it by setting the cursor of the stage to the `animatedCursor`. The cursor will be updated by the `animatedCursor` as required (provided `tickEventClock` is called by the system).

CHAPTER 31

Types

This chapter is under development. It will appear in the finished manual.

Widgets

This chapter is under development. It will appear in the finished manual.

Glossary

A

actor
actor (with a small “a”) is a collective, generic term referring to an Actor Object(s).

Actor
Actor (with a capital “A”) refers specifically to the Actor Object. Actor provides most of the graphical and drawing capabilities within SK8 and is a descendent of the Graphic Object.

Ancestors
Objects from which child objects are derived.

Anonymous Object
An object whose objectName is False.

Arguments
Data that is passed as a parameter with a message within the handler. If the called object does not have a feature of the same name, then an exception occurs.

B

C

Calling
The action of sending a message object to another object.

Child
A child object is one that has been derived from an existing ancestor or parent object.

Containers
Connections between the components in the containment hierarchy. A container, in SK8, is either an actor or the Stage. A visible actor is either attached to the Stage or another actor that is attached, ultimately, to the Stage.

Containment
An object inside another object.

Containment Hierarchy
The organization of objects to represent the order in which objects are contained by other objects. For example, a visible actor is contained by Stage (or contained by another object that is contained by Stage).

D

Deep Copy
Made by copying the object and every object it references and every object referenced by those objects and so on down the line.

Descendants
The “off-spring” (children) of objects.

E

Error System
A pre-defined, but extensible, set of objects and handlers that trap particular system errors. See Appendix for a list of supported error types.

Event
A user-initiated (mouse, keyboard) or system action. SK8 identifies the event and invokes the appropriate event handler (if defined by SK8 or the developer).

Event Mode
Event modes are predefined objects in SK8 that allow the circumvention of normal event processing.

F

Focus
The area of the screen that is the current target for mouse or keyboard events.

G

Garbage Collection

Any object not being referenced by any other object is dynamically purged from memory. This feature is built into the language and frees the programmer from worrying about memory management. Most common compiled languages do not contain built in garbage collection.

H

Handler

A named piece of code that is executed in response to a message (the mouseDown handler is invoked when a mouseDown event happens.) In other Object Oriented systems, handlers are called “methods”.

Handling

Handling a message is equivalent to executing a handler.

Heterarchy

A model in which an object can have more than one parent and therefore can inherit properties and handlers from more than one parent.

Hierarchy

A model in which an object an have only one parent. Also, a method for organizing objects. Example, containment hierarchy.

I

Inheritance

An OOP term. The ability of an object to inherit the characteristics (handlers, properties, etc.) of another object.

Instantiate

An OOP term. Creating an object as a data type.

Interlude

An interlude is designed to “distract” the user and fill up the time required for a large media file to load.

M

Message

An object used to tell another object to perform its function. In a broader sense, an OOP term.

Multimedia

The current hot buzzword in the computer industry. A “vanilla” definition: the embracing of video, sound, still images, animation, or hypertext with computer hardware and software technology. SK8 is an example of the software technology used to generate multimedia applications.

Multiple Inheritance

An OOP term. The ability to inherit features, properties, etc. from multiple parents.

O

object

In SK8, an object is an instance of another object. Each object contains properties and handlers and each object has parents or ancestors.

In “traditional” OOP, an object is an instance of a single class and does not have parents or ancestors -- the object’s class has ancestors.

Object

Object (with capital “O”) refers to the Object object. Object is the parent of all objects. All objects are derived from Object.

Object Heterarchy

Represents an object’s inheritance of properties, features, etc. from multiple parents.

Object Hierarchy

Represents the inheritance of objects: parents, children, ancestors, and descendants from a single parent.

OOP

Object Oriented Programming

Overload

An OOP term. The same operation can be used consistently with objects of different classes (types). Not only can the same operator be used for characters, integers and Booleans, but the operator can be overloaded for other classes by declaring an operator method for that class.

Override

If a particular object wants to do things differently than its ancestors, it can override any handler that it inherited from any of its ancestors by declaring a handler of the same name.

P

Parameter

Data passed with a handler or message.

Parents

Objects from which an object inherits properties and handlers.

Polymorphism

Dynamic binding of messages to specific features. The same message can be sent to several objects which may have different object derivations but have handlers to handle that message. This allows the object framework to be abstracted to a higher level.

The ability to exhibit different behavior for the same message (depending on the object type).

Property

A property is storage space in an object. This storage has a name and can hold exactly one value. In other Object Oriented systems, properties are known as “instance variables” or “slots”.

Protocol

A formal set of “rules” or conventions that govern how two entities interface, communicate, or react to each other. Within SK8 there are protocols between objects and collections of objects.

Prototype-based

In a prototype-based model, no distinction is made between the object and the “template” (class in conventional OOP). The object is the “template” for other objects. Classes are not involved.

In SK8, everything is an object. Every object is a prototype. Every prototype is a type. Therefore, every objects is a type. See Type.

Q

QuickTime™

An Apple product/technology developed for the Macintosh that SK8 supports for the playing of movies. Many libraries of QuickTime movies are available. SK8 can be used to create, modify, and play QuickTime movies.

R

Receive, Receiving a message

When a message is received by an object, it executes the named handler or returns the value of the named field.

S

Screen Object

An object that is currently displayed on the screen.

Scripter

A person who writes scripts -- such as SK8 Script, AppleScript, and HyperTalk.

Sending a message

A message is an object that can be sent to another object.

Shadow Copy

Made by only copying the object and just referencing the objects it references.

Stream

A classification of data where the data characters exist in an input/output stream.

T

Title, Title Application

A term used to refer to a multimedia project.

Type

The object type is the values of the object’s properties (variables) and handler behavior. The object type is determined by inheritance from parent objects. Every object is a type. See Prototype-based.