FaceSpan

User's Guide

Version 3.0

Digital Technology INTERNATIONAL®

Digital Technology International
500 West 1200 South
Orem, UT 84058
801-226-2984
801-221-9254 Fax

Visit the FaceSpan Web site at www.facespan.com for the most current information on FaceSpan.

Table of Contents

Appendix

Index

Preface

Table of Contents Index

About This Guide

The *FaceSpan User Guide* is a complete guide to the FaceSpan application package. FaceSpan™ the Interface Designer & Application Builder, is a visual interface design and scripting tool that helps you create professional-looking Macintosh applications.

You should read this section if you want to know:

➤ how this guide is organized

➤ what each chapter includes

➤ how to interpret words that have special formatting, including scripts

➤ what references were used in writing this guide

Organization of the Guide

The *FaceSpan User Guide* describes how to use FaceSpan, defines its terminology, and provides a reference for the properties of FaceSpan objects and the values those properties can have.

This guide is intended for experienced Macintosh users who know how a Macintosh application should look and behave, and who may have some previous experience in scripting applications.

The guide contains an introduction and three main parts.

➤ Part I: “FaceSpan Usage Guide” describes how to use FaceSpan’s development and testing environments.

➤ Part II: “Application Development” discusses the structure of applications, how to use scripts to control your application, how to use scripting additions, and how to script other applications—including the Scriptable Finder.

➤ Part III: “FaceSpan Object and Language Reference” tells about each of the FaceSpan object classes, their properties and values, and the commands used to control them.

The guide’s appendices include a menu reference, a reference for commands and shortcuts, information about sizes and limits, information on scripting resources, a reserved word list, and where to find more information about how to write forms.

This guide also includes an index.

Introducing FaceSpan

The introduction contains three sections:

➤ "Overview of FaceSpan" gives a brief overview of FaceSpan, its interrelationship with AppleScript, its development and testing environments, and some suggested ways you can use FaceSpan.

➤ "How to Get More Information" tells you where to find demonstration projects and sample applications that are included with FaceSpan, and describes FaceSpan's on-line reference tools.

➤ "New Features in Version 2.1" lists many of the new features added since version 1.0, and tells you where to find additional information about them.

Part I: FaceSpan Usage Guide

Part I contains eight chapters:

Chapter 1

"Overview of the Environment," describes FaceSpan and its associated files, its interface objects, and basic development tasks.

Chapter 2

"Project Management," describes FaceSpan's Project Window and gives you instructions for using it to manage your project's resources.

Chapter 3

"The Window Editor," describes the tools used to create a user interface, and gives instructions about how to use them.

Chapter 4

"The Menu Editor," describes the environment used to create and edit run-time menus, and gives instructions about how to create menus.

Chapter 5

"The Storage Item Editor," defines storage items and gives instructions for making and using them in projects.

Chapter 6

"The Script Editor," describes how to use FaceSpan's scripting environment to create, edit, and compile scripts.

Table of Contents Index

Chapter 7

“Other Scripting Tools” describes how to use the Message Windoid and the Dictionary Windoid.

Chapter 8

“The Testing Environment,” defines various testing modes and tells how to use them.

Part II: Application Development

Part II contains two chapters:

Chapter 9

“The Structure of Applications,” describes the components and hierarchy of applications and how FaceSpan works.

Chapter 10

“Scripting Your Application,” discusses the structure of scripts, how to use them to build applications, and how to use them to control other applications. It also suggests a step-by-step process for application development.

Part III: FaceSpan Object and Language Reference

Part III contains six chapters:

Chapter 11

“Applications,” describes the properties of the application object as well as the commands and messages understood by applications.

Chapter 12

“Windows,” defines the properties of windows as well as the commands and messages understood by windows.

Chapter 13

“Window Items,” describes the properties common to all window items, and the commands understood by all window items. It also provides information about including form definition resources and key filters in your application. Each window item—push buttons, radio buttons, checkboxes, labels, textboxes, pictures, icons, movies, popups, listboxes, boxes, graphic lines, gauges and tables—is defined by its properties and the commands it understands.

Chapter 14

"Menus and Menu Items," describes the properties of menus and menu items.

Chapter 15

"Special Artwork and Text Style Classes," describes the properties of the Resource Info and Text Style Info classes.

Chapter 16

"Storage Objects," describes the properties of storage objects.

Appendices

This guide contains six appendices:

Appendix A

"FaceSpan Menu Reference," explains the commands available from FaceSpan's menus.

Appendix B

"Commands and Shortcuts," lists the keyboard commands, keyboard equivalents to FaceSpan's menu commands, and the mouse shortcuts supported by FaceSpan.

Appendix C

"Sizes and Limits," tells about the maximum sizes or counts you may expect while using FaceSpan.

Appendix D

"Scripting Resources," lists reference materials and on-line services that can help you learn more about FaceSpan and AppleScript.

Appendix E

"Reserved Word List," gives a list of words that are used for objects, classes, properties, messages, and values by FaceSpan.

Appendix F

"How to Write Forms," tells you where to find additional information about writing forms.

Appendix G

"Speed Enhancement Tips," gives you tips on how to make FaceSpan run faster.

Table of Contents Index

Language Conventions

There are instances where words in this guide use a particular text style to set them apart as having special meaning.

➤In examples of script syntax, those items that are to be replaced by the scripter are italicized. For example:

tell window item *itemName* to draw
where *itemName* is to be replaced by an actual name.

➤ Optional parameters are enclosed in square brackets to distinguish them from required parameters. For example:

“The parameters of the keystroke message include [key], [option down], [command down], …”

➤ Words defined by FaceSpan appear in a special font. For example:

“...You can remove any of the control boxes by adjusting the window template’s **`closeable`**, **`resizable`**, and **`titled`** properties.”

or

“...The project script is not run, so open **`window`** commands are not executed.”

References

AppleScript Language Guide English Dialect. Cupertino: Apple Computer, Inc., 1993.

AppleScript Scripting Additions Guide English Dialect. Cupertino: Apple Computer, Inc., 1994.

Macintosh Human Interface Guidelines. Cupertino: Apple Computer, Inc., 1993.

Overview of FaceSpan

You should read this chapter if you want to know:

- ➤ how FaceSpan and AppleScript are interrelated
- ➤ what you can do with FaceSpan
- ➤ what you need to know in order to use FaceSpan

About FaceSpan

FaceSpan is an application package that allows you to create sophisticated graphic user interfaces for AppleScript applications. It includes editors for scripting your application, constructing and scripting windows and menus, as well as a management environment through which you can inspect, edit, duplicate, and delete each application's interface components and resources.

You can script essential elements of FaceSpan itself in order to automate some reporting or to perform wholesale editing of windows and window items. New properties return lists of all resources within a project, and the **`open`** and **`print`** commands accept these resources. For example, the window resources command returns a list of items of type resource info; each item refers to a window in the project. Any of these items may then be used in an open or print command.

FaceSpan and AppleScript

Used alone, most scripting languages can provide only simple dialogs for communication with the user. But using FaceSpan and AppleScript—or any Open Scripting Architecture ("OSA") scripting language—you can create and control windows, dialog boxes, windoids, and menus that give your application the look and functionality of professionally-engineered Macintosh software.

FaceSpan's objects automatically behave in accordance with Macintosh Human Interface Guidelines, and you can change their properties to extend or modify functionality. FaceSpan provides even greater functionality by allowing you add form definition resources to projects, then use them to control window items. Using forms, you can customize the behavior and appearance of almost every window item.

Table of Contents   Index

Completed FaceSpan projects can be saved as applications, and run on any Macintosh computer where AppleScript (version 1.1 or later) has been installed. Using FaceSpan, you can communicate with and control the Finder, "borrow" capabilities from other AppleEvents-aware applications, and integrate those functions to perform specialized tasks. The applications you create with FaceSpan can be as flexible and powerful as you want them to be.

FaceSpan and Frontier

Enhanced Frontier support is now available in FaceSpan! Here are some of the changes:

➤ UserTalk can be executed right from the Message Window, as well as from any object's script.

➤ The Script Editor popups support UserTalk.

➤ The titles of the Script Editor's windows reflect the selected language.

➤ Newly created scripts default to the same language as the project script; you don't have to keep choosing UserTalk for each new script.

➤ You can now record UserTalk scripts.

➤ Errors are now properly reported, and their locations are hilited.

FaceSpan's Development and Testing Environments

A completed FaceSpan project is made up of windows, menus, scripts, artwork, and other resources, all of which function as a unified application. Using FaceSpan's Window Editor and Project Window, you can test-run your project's scripts—as well as test the way objects will respond to user input—while the project is still under development.

FaceSpan's Project Window is the command center from which you can manage all your project's component resources, as well as test-run the project script. The project script—the main script for the application you are building—is a project's central control, and can coordinate a variety of run-time functions including menus and windows.

Menus are created using FaceSpan's Menu Editor. Inside the Menu Editor, you can create, design, and edit menu templates. Once the template is created, FaceSpan then saves a description of it, and uses that description to create the actual menu at runtime. The menus added to the application's menu bar at runtime can include menus associated with a particular window, or with the project itself.

Windows are created in much the same way as menus. FaceSpan’s Window Editor lets you to design, create, and edit window templates. Once window items—such as push buttons, radio buttons, icons, tables, QuickTime movies, and more—are added to the window template, you can use the Window Editor to change the state of the window template from Edit Mode to Play Mode. Play Mode allows you to interact with a window and its window items—interaction very similar to that of an application user during runtime.

FaceSpan’s Message Windoid is a powerful tool in both the development and testing environment. You can open the Message Windoid during Edit or Play Mode to get and set properties, as well as to write and run scripts that send test messages to window items. During runtime, you can use the Message Windoid to track Apple Events.

What Can Be Done with FaceSpan

The projects you make with FaceSpan can be as simple or as powerful as you can imagine. Listed below are a few of the many ways in which it can be used.

Create friendly interfaces

Use FaceSpan to put user-friendly interfaces on otherwise faceless scripts, or to make simplified “command centers” for complicated scriptable applications.

Make standalone applications

FaceSpan has a friendly, object-oriented language, and framework which you can combine with any OSA compatible scripting language to make a complete environment for designing and developing real applications.

Create quick prototypes

Any kind of application can be prototyped with FaceSpan, and at any level of detail, from simple presentations of windows to fully functional applications.

Develop integrated software

You can create FaceSpan applications that pull together the services of other scriptable applications, including other FaceSpan applications, and the Scriptable Finder, too.

Table of Contents Index

Develop scriptable applications

Every application developed with FaceSpan is itself scriptable—even to the point of creating new windows, window items, and scripts at run time.

Make tools for FaceSpan

Because FaceSpan is itself a scriptable application, you can use FaceSpan to develop “palette” applications that help you work with FaceSpan: augment the editing environment with custom tools and shortcuts.

Learn to program

With FaceSpan’s supportive and satisfying environment, you can learn all about programming, including object-oriented programming.

What You Need to Know

This guide is intended for experienced Macintosh users. It assumes that you have at least an informal grasp of Macintosh standards for human interface and application design—that is, that you know how a Macintosh application should look and feel. Previous experience writing scripts is preferred, but not required.

How to Get More Information

Visit the FaceSpan Web site at www.facespan.com for the most current information on FaceSpan.

Feature Highlights

The FaceSpan disks include many simple projects designed to illustrate FaceSpan’s features, demonstrate coding techniques, and show typical uses of the product. You will find them in the “Feature Highlights” folder.

The best way to learn from these projects is to open each one in FaceSpan, run each project script, view the project’s run-time behavior, then look at the scripts and properties used to create the project.

FaceSpan’s Built-in Reference Tools

FaceSpan’s Script Editor has popups (pop-up menus) you can use to automatically insert references to windows, objects, properties, and messages into scripts.

FaceSpan’s Dictionary Windoid allows you to locate a scriptable application or scripting addition, then open its dictionary (aete resource) to inspect definitions of that application’s objects, commands, or other words it understands.

For more information about the Script Editor, see Chapter 6. The Dictionary Windoid is discussed in Chapter 7: “Other Scripting Tools.”

New Features in FaceSpan

New Features in FaceSpan Version 3.0

➤ The Menu Editor has been improved to make it more user-friendly. See *Chapter 4:The Menu Editor*.

➤ You can now assign contextual menus to objects in the Window Editor. When a user presses the Control key and clicks the object, the menu appears. See *Assigning Contextual Menus to an Object* on page 113.

➤ You can now assign a sub menu directly to a menu item. Before you can assign a sub menu, you must create it in the Menu Editor. A menu item with a sub menu cannot have a command key. See *Assigning a Sub Menu to a Menu Item* on page 111.

➤ You can now more easily assign command key modifiers, such as Option or Shift, to a keyboard shortcut. See *The Command Key Pop-up Menu* on page 107.

➤ FaceSpan 3.0 makes it easier to use event logging to assist you in debugging projects saved as applications. See *Logging Events to a File* on page 143.

➤ You can now save applications as windows that float on top of other applications. See *Saving a Project as a Floating Window* on page 60.

➤ New tools in FaceSpan 3.0 include bevel buttons, disclosure triangles, progress bars, sliders and tab panels. See *Bevel Buttons* on page 306, *Disclosure Triangles* on page 323, *Gauges, Sliders, Progress Bars, Spinning Arrows and Up/Down Arrows* on page 326, and *Tab Panels* on page 389.

➤ You can now use scripts to test and set the collapsed state of a window. See *collapsed* on page 235 or *collapsable* on page 235.

➤ You can now link items in a window, such as buttons, checkboxes and so forth, to radio buttons, checkboxes or disclosure triangles. You can choose to enable, disable, hide or show the linked items. See *links (Bevel Buttons)* on page 309, *links (checkboxes)* on page 321, *links (disclosure triangles)* on page 323 and *links (radio buttons)* on page 387.

➤ You can now use ⌘-R as a shortcut for the Run Project command.

➤ You can now use ⌘-K to add a new item to a project.

New features in FaceSpan Version 2.1

➤ When saving miniature and complete applications, you now can choose whether to include the dictionary. See *Saving a project as an editable application* on page 59.

➤ When saving as run-only, you can now choose to extract the visible textual properties of items (e.g. title, contents) in the windows into TEXT and STR resources. See *Saving a project as a non-editable, run-only application* on page 61.

➤ A window now has a **`title`** property distinct from its name property. See *name* on page 245 and *title* on page 252.

➤ A pictbox now has a **`scale`** property, which controls the magnification or reduction of the image in the pictbox. See *scale* on page 367.

➤ A pictbox now has a **`justification`** property. See *justification* on page 366.

➤ An application now has a **`ticks`** property, which returns the number of ticks (60ths of a second) since the machine was turned on. This can be helpful in implementing timed behaviors. See *ticks* on page 217.

➤ An application now has **`heap space`** and **`stack space`** properties. These report the amount of free memory available to the application. They help you better monitor and respond to low-memory situations. See *heap space* on page 211 and *stack space* on page 216.

➤ Enhanced Frontier support is now available in FaceSpan! See *FaceSpan and Frontier* on page 23.

➤ Two new commands let your scripts mimic the actions of a real user within other applications. Using these commands, you can automate the operations of applications that have no built-in support for scripting. See *click as user* on page 219 and *type as user* on page 225.

➤ While FaceSpan itself has been “fat” since version 2.0, miniature and complete applications can now take advantage of the increased performance of native code with a “fat” version of the FaceSpan Extension.

➤ Windows can now be printed under script control. The printing of multiple pages with sophisticated layouts is now supported. A print setup command provides control over paper margins and printing dialogs. See *print / print setup* on page 223.

➤ You can script essential elements of FaceSpan itself, to automate some reporting or to perform wholesale editing of windows and window items. New properties return lists of all resources within a project, and the **`open`** and **`print`** commands accept these resources. For example, the window resources command returns a list of items of type resource info; each item refers to a window in the project. See *About FaceSpan* on page 22.

➤ You can now play and record sounds as either "snd" resources or as "AIFF" files. You can even make the computer speak with Text-To-Speech. You can also import sounds into your project and play them back. See *sound* on page 269 and *text to speech* on page 251.

New features in FaceSpan version 2.0

Enhanced Interface Development Environment

➤ Support for Drag and Drop in editing

See Part I, Chapter 3: "The Window Editor," and Part I, Chapter 6: "The Script Editor."

➤ Control over each window item's resizing in response to window resizing

See Part III, Chapter 13: "Window Items," Properties Common to all Window Items.

➤ User-configured default windows and projects

See Part I, Chapter 2: "Project Management," and Chapter 3: "The Window Editor."

➤ A format property for configuring popups, listboxes, textboxes, buttons, and tables

See Part III, Chapter 13: "Window Items."

➤ Text alignment for labels and titled boxes

See Part III, Chapter 13: "Window Items," Properties of Boxes, Properties of Labels.

➤ Direct import of forms, key filters, and scripting additions from other files

See Part I, Chapter 2: "Project Management" and Part II, Chapter 10: "Scripting Your Application," Using Scripting Additions.

➤ Direct import of cursors and pixel patterns

See Part I, Chapter 2: “Project Management” and Part III, Chapter 11: “Applications,” as well as Chapter 13: “Window Items,” Properties of Boxes.

- On-line form definition resource documentation

  See Part I, Chapter 2: “Project Management” and Part III, Chapter 13: “Window Items,” Form Definition Resources and Key Filters.

Enhanced Scripting and Testing

- OSA support

  See Part I, Chapter 6: “The Script Editor.”
- Event logging

  See Part I, Chapter 8: “The Testing Environment.”
- A storage object that can contain any value, including script objects

  See Part I, Chapter 5: “The Storage Item Editor” and Part III, Chapter 16: “Storage Items.”
- Script-controlled Show Balloon event for objects

  See Part III, Chapter 13: “Window Items,” Commands and Event Messages
- Eleven new application properties, including a clipboard property

  See Part III, Chapter 11: “Applications,” Properties of Applications.
- AppleScript formatting

  See Part I, Chapter 6: “The Script Editor.”
- On-line dictionary reference

  See Part I, Chapter 7: “Other Scripting Tools.”
- Global persistent variables accessible by all scripts

  See Part II: “Application Development” and Part III, Chapter 16: “Storage.”
- “Do Script” event support

  See Part III, Chapter 11: “Applications” and Chapter 12: “Windows.”
- A Script menu with additional editing commands

  See Appendix A: “FaceSpan Menu Reference.”

Table of Contents Index

➤ Full Find and Replace support in the Script Editor

See Part I, Chapter 6: “The Script Editor” and Appendix A: “FaceSpan Menu Reference.”

➤ A Script Editor popup that displays handlers for pre-defined messages and user-defined handlers, and can find or navigate to them

See Part I, Chapter 6: “The Script Editor.”

➤ Script-initiated idle event

See Part III, Chapter 11: “Applications” and Chapter 12: “Windows.”

Enhanced Functionality for Your Applications

➤ A table object

See Part III, Chapter 13: “Window Items,” Properties of Tables.

➤ Support for Drag and Drop at runtime

See Part III, Chapter 13: “Window Items.”

➤ Enhanced key filters for textboxes including DisplayDates, DisplayAlphas, DisplayNumbers, DisplayBooleans, DisplayChoices

See Part III, Chapter 13: “Window Items,” Properties of Textboxes.

➤ Direct embedding of scripting additions (OSAXs) into projects

See Part I, Chapter 2: “Project Management,” and Part II, Chapter 10: “Scripting Your Application.”

➤ Text Suite support for textboxes

See Part III, Chapter 13: “Window Items,” Properties of Textboxes.

➤ Color patterns for filling boxes

See Part III, Chapter 13: “Window Items,” Properties of Boxes.

➤ Animated buttons

See Part III, Chapter 13: “Window Items,” Properties of Push Buttons, Properties of Pictboxes.

Part I: FaceSpan Usage Guide

Chapter 1:

Overview of the Environment

Contents:

Table of Contents Index

Overview of the Environment

You should read this chapter if you want to know:

- ➤ what types of files FaceSpan can create
- ➤ what interface objects FaceSpan can create or support
- ➤ what basic tasks are required to develop an application using FaceSpan

FaceSpan and its Associated Files

FaceSpan
application
desktop icon

Using FaceSpan you can create Project documents, Complete Applications, and Miniature Applications; each has its own desktop icon.

Projects

Project
document
icon

Projects are editable documents created with FaceSpan. A completed FaceSpan project is made up of windows, menus, scripts, artwork, form definition resources, and other resources, all of which can function as a unified application.

You can make an application from a project by saving the project as an application. Any FaceSpan application can be saved in a non-editable run-only form (which you may prefer for distribution copies), or in an editable form that can be opened by FaceSpan for additional modification and testing.

Applications

Application icon

“Droppable”
Application icon

Table of Contents Index

FaceSpan creates two types of applications, Complete Applications and Miniature Applications. Both Complete and Miniature Applications can be made “droppable.” A droppable application’s scripts run when another document’s desktop icon is dropped onto the application’s desktop icon. You can make a FaceSpan application “droppable” by including a handler for the Open command in its project script. Droppable FaceSpan applications are distinguished from others by their icons, which have a downward arrow.

Complete Applications

A project that is saved as a Complete Application can run on any Macintosh computer on which AppleScript 1.1 or later has been installed. All Complete Applications contain a copy of the FaceSpan Extension.

Miniature Applications

A project that is saved as a Miniature Application requires that the FaceSpan Extension, as well as AppleScript (version 1.1 or later), have been installed on the Macintosh computer where it is to be run. Miniature Applications can have all the functionality of Complete Applications; they are called “Miniature” only because the applications themselves require less disk space.

About the FaceSpan Extension

FaceSpan Extension icon

When you purchase FaceSpan, your license agreement allows you to distribute the FaceSpan Extension, along with your applications, royalty-free. The FaceSpan Extension is a System Extension file that acts as a run-time assistant for projects saved as Miniature Applications. The FaceSpan Extension file must be in the Extensions folder (within the System Folder) of any Macintosh computer on which Miniature Applications are to be run. Be certain to include a copy of the FaceSpan Extension (along with installation instructions) when distributing Miniature Applications to users who do not own a copy of FaceSpan.

While FaceSpan itself has been “fat” since version 2.0, miniature and complete applications can now take advantage of the increased performance of native code with a “fat” version of the Facespan Extension.

FaceSpan’s Interface objects

A project’s user interface is made up of menus, document windows, modal dialogs, and windoids that you create using FaceSpan’s editors, and control through scripts.

Menus

The customized menus you create using FaceSpan’s Menu Editor give your applications complete control of the menu bar at runtime. After menus are constructed, you can easily “attach” them to a project, or to any of its windows. At runtime, menus then display when their project or window is active.

Windows

Using FaceSpan, you can create window templates belonging to three classes of windows: document windows, modal dialogs, and floating windoids; all controlled through scripts. Although a project’s windows perform as a coordinated user interface when the application is running, you create and edit each window individually as a template, or model of the run-time window.

Each time you create a new window template or open an existing template, FaceSpan opens a Window Editor for it. Using the Window Editor, you can add window items and arrange them within the template, adjust the properties of the window and its window items (to define their forms and functions), and write scripts to control their behavior.

Window items

Windows contain smaller parts generically called window items. Window items include familiar Macintosh interface objects such as push buttons, radio buttons, checkboxes, icons, pictures, movies, listboxes, popups (pop-up menus), tables, and more. They all belong to the class, window item, because they share common characteristics—such as properties and commands—that they all understand. In turn, each of these objects is its own object class, with a distinct appearance, an automatic response to user input, and properties through which its appearance and function can be modified.

Table of Contents Index

Window Item Properties, Commands, and Messages

Every window item has properties. Each property has a name and a value. For example, a property named **`visible`**, when set to the value **`true`** or **`false`**, determines whether or not that object can be seen in its window. The **`visible`** property (and many others) are common to all window items, but some properties apply only to specific object classes of window items.

At runtime, window items receive messages which are generated automatically in FaceSpan when the user interacts with its interface objects. Message handlers can be added to the scripts of projects, windows, and window items to intercept and respond to these messages.

Artwork

You can add your own artwork to projects. The window item classes **`picture`** and **`icon`** act as containers for the display and manipulation of artwork. In addition to pictures (PICT) and icons (ICON, cicn, and ICN#), you can customize the look of your project by using “color patterns” (ppat), as well as black and white cursors (CURS), which are treated as artwork. FaceSpan allows you to import, copy, paste, delete, and rename these resources.

Form resources

The default standard form definition resource, or form, used by each window item is built into FaceSpan. The standard listbox form, for example, lets a listbox display only text. In addition to providing default standard forms for its objects, FaceSpan lets you add form definition resources to a project, then use them to control window items. For example, you might add a listbox form that allows the listbox to display icons in addition to text. The FaceSpan Additions project (included in your software package) contains additional form resources as well as key filters that control the entry of characters into textboxes. Using FaceSpan, you can import, delete, copy, and paste these resources.

Scripts

Scripts can be attached to any window or window item, or to the project itself. You can use scripts to change the way windows and window items look or behave (in edit mode or during runtime) by assigning new values to their properties. Scripts contain handlers and other subroutines. Handlers are

Table of Contents Index

subroutines that are activated by the messages that are sent in response to user interaction with the application. A project script—the main script for the application you are developing—is a set of handlers and properties that defines the project’s behavior.

Basic Development Tasks

Managing projects

Once you’ve created a project, its Project Window displays a constantly updated overview of the project’s resources: window templates, menu templates, artwork, forms—including key filters and scripting additions—and storage items. You can use the Project Window to access the project script, add new resources, open the appropriate editors for editable project resources, or choose items to be cut, copied, pasted, deleted or renamed. In addition, the Project Window helps you to copy and paste resources between projects.

Editing menus

FaceSpan’s Menu Editor makes creating custom menus for your project as simple as clicking and typing. After you type names for each menu template and the items it contains, you can assign optional mark characters and Command-key equivalent characters to the menu items. Once menu templates are constructed, you can easily “attach” them to a project (or any of its windows), and add handlers to define the actions of each menu’s commands. You don’t have to worry about making menus open and highlight when users click them; FaceSpan automatically does that for you.

Editing windows

Inside FaceSpan’s Window Editor, you can create and fine-tune window templates and the window items they contain. You add window items by selecting them from a palette of objects, then dragging and dropping them onto a window template, or click-dragging them to size in a window template.

Most of the window items automatically behave like similar objects in standard Macintosh applications. Once you’ve created a window item, FaceSpan displays its properties and their current values. You can define and adjust each item’s appearance and behavior by using the Window Editor’s tools, or by using scripts to assign values to the window item’s properties.

Scripting

After you have created the windows and menus for your project's user interface, you write scripts to control the run-time behavior of the project, its windows and menus, and their items. You can attach a script to any project, window, or window item.

The Script Editor not only provides a standard text-entry environment for creating and editing scripts, but on-line reference pop-up menus and a script recorder as well. Using the popups, you can choose references to windows, objects, properties, and handlers; FaceSpan then automatically inserts the text of the reference into your script for you. Using the Script Editor's script recorder, you can generate new scripts from sequences of actions you perform in recordable applications.

Testing

FaceSpan supports testing at all stages of application development. You can test windows while constructing them, and check scripts for compilation errors at any time while scripting. You can use the Message Windoid to send sample messages to any object, and to log AppleEvents and replies. Once you've written the project script you can test your project's run-time behavior by simply clicking the Run button in the Project Window.

Table of Contents Index

Chapter 2:

Project Management

Contents:

Table of Contents Index

Project Management

You should read this chapter if you want to know:

➤ how to create and save projects

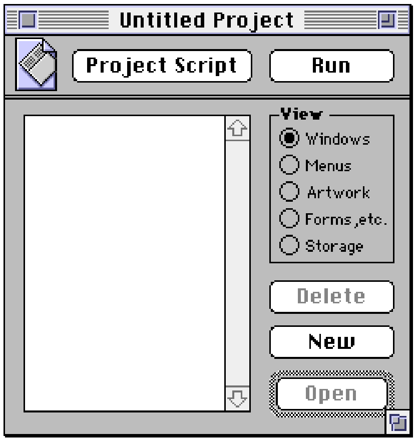
➤ how to import and manage project resources: Windows, Menus, Artwork, Forms, and Storage

A completed FaceSpan project is made up of windows, menus, scripts, artwork, and other resources, all of which function as a unified application. To assist you in creating, editing, and testing interface components, FaceSpan displays a Project Window for each open project document and application being edited.

All project-level editing centers around the Project Window. While the Project Window is active, all of the commands in the File menu and the Edit menu pertain to the project. You can use the File menu to create and save projects, to revert an edited project to the state in which it was last saved, to convert a project to an application, and to print the project's contents. You can use the Edit menu to cut, copy, paste, clear, and duplicate the resources listed in the Project Window.

Design-time editing done directly from the Project Window can include the modification of the project script, and the addition or modification of project resources (window templates, menu templates, artwork, form definition resources, and storage items). In addition to providing a global project management environment, the Project Window allows you to test-run your project or application.

The Project Window



Using the Project Window, you can…

➤ Edit the script of the project

➤ Test projects under construction and run completed projects

➤ Save projects as applications

➤ Cut, copy, and paste any resource among projects

➤ Open editors for editable project resources

➤ Create, inspect, and delete window and menu templates

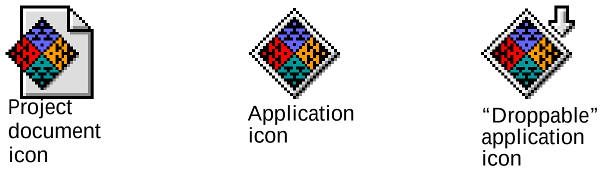
➤ Import artwork from any Macintosh document or application

➤ Create, define, or delete storage items

The Project Window is divided into two sections. The top part contains the Project icon, along with the project script controls. The lower part of the Project Window displays categorized lists of the project's component resources, as well as controls for editing them.

Project icon

The Project icon, located near the top left corner of the Project Window, identifies the form in which the open document was last saved: as a project document, an application, or a droppable application.



Project Script controls

Project Script controls include the Project Script button and the Run button. You can use the project script controls to inspect, edit, and run the project script—the main script of the application you are developing.

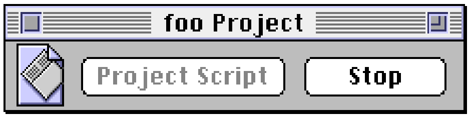
Project Script button

Click the Project Script button to open the Script Editor that contains the project script.

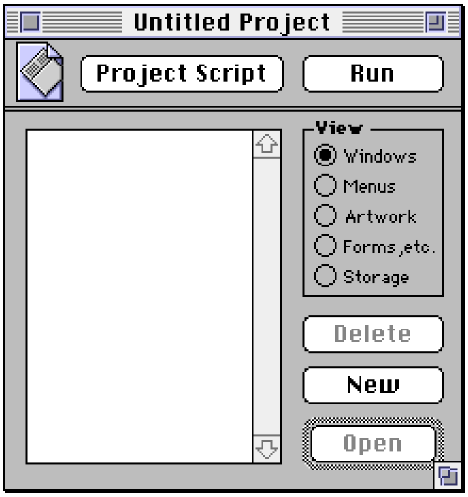
Run button

When you click the Run button, FaceSpan attempts to compile any open scripts, hides any open window templates, and runs the project script.

While the project script is running, the Project Window temporarily shrinks (as shown below) and the Run button becomes a Stop button. You can then click the Stop button to halt the running project script.



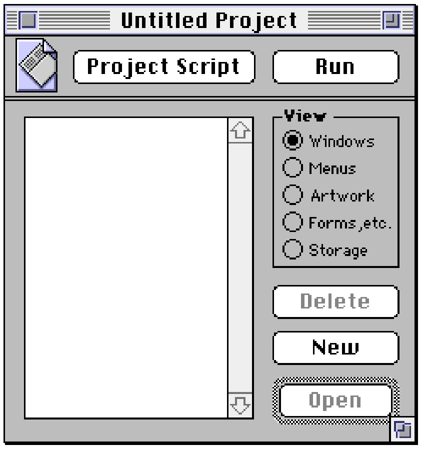
Project Window listbox and View radio buttons



The resources displayed in the Project Window listbox are determined by which View radio button you select. Depending on the view selected, the Project Window listbox displays the names of the window templates, menu templates, artwork, form definition resources, or storage items contained in the project.

Windows View

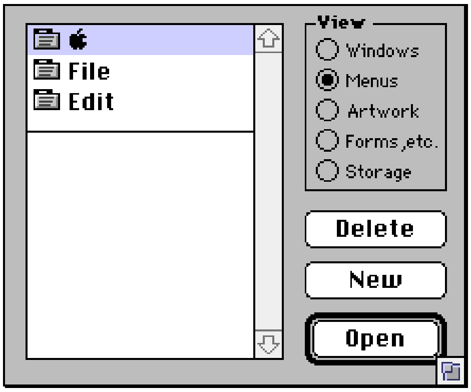
The window templates associated with a project are created within FaceSpan and can be any of the three types of window classes: document windows, modal dialogs, and floating windoids. For additional information about creating window templates, see to Chapter 3: “The Window Editor.”



Menus View

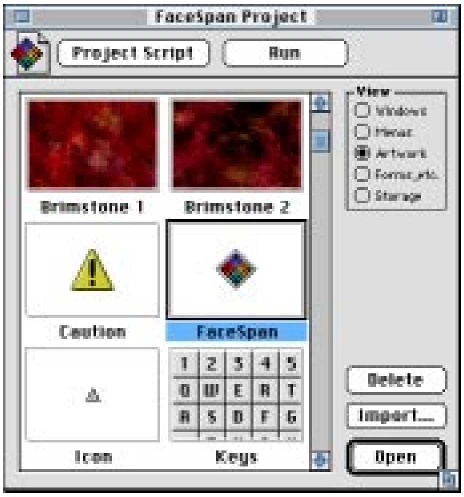
Menu templates are created within FaceSpan, and then associated with particular windows of the project or with the project itself. Menus are displayed in the menu bar during runtime only.

In Menus View, the Project Window listbox contains a horizontal divider; menu templates listed above the divider are associated with the project, while menu templates below the divider are associated with specific windows. You can relocate a menu template in the list by click-dragging its name to a new position in the list. For additional information about creating menu templates, see Chapter 4: “The Menu Editor.”



Artwork View

Artwork resources can include pictures (PICT), icons (cicn, ICN#, and ICON), color patterns (ppat), and black-and-white cursors (CURS). Artwork is not created in FaceSpan, but can be imported into a project from any Macintosh document or application, then renamed and managed using the Project Window. You can find more information about artwork and their properties in Part III, Chapter 12: “Window Items” and Chapter 15: “Special Artwork and Text Style Classes.”



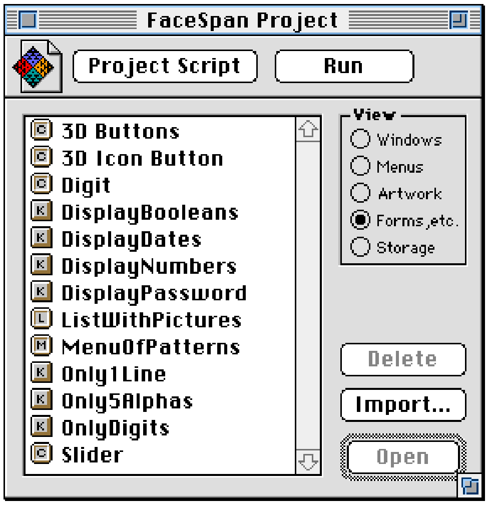
Forms, etc. View

Items listed in the Forms, etc. View can include form definition resources—such as control definitions (CDEFs), menu definitions (MDEFs), list definitions (LDEFs), and key filters (Key*ƒ*)—as well as scripting additions. When forms are listed in the Project Window, FaceSpan places a small icon in front of each name to indicate the kind of form it is: a “C” (as in CDEF) represents forms for buttons and gauges, an “L” (as in LDEF) marks the forms for listboxes, an “M” (as in MDEF) denotes forms for menus and popups, and a “K” represents key filters for textboxes. Scripting additions are marked with the standard scripting addition icon. You can view documentation about each form by double-clicking its name in the list; a dialog box containing an explanation of the form and its use then displays.

Forms and scripting additions are not created in FaceSpan, but can be imported into a project and deleted from a project using the Project Window. To assign a particular form to a window item, you must set the item's **`form`** property or **`key filter`** property to the name of that form or key filter. You can find more information about forms in Part III, Chapter 13: "Window Items." Scripting additions are discussed in Part II, Chapter 10: "Scripting Your Application."

Note

➤ While a project that uses a scripting addition is under development, the scripting addition must be in the Scripting Additions folder (located in the Extensions folder of your System folder). When you are ready to prepare the project for distribution, you must then import the scripting addition into the project so that a copy of it is made a integrated permanent part of the project.

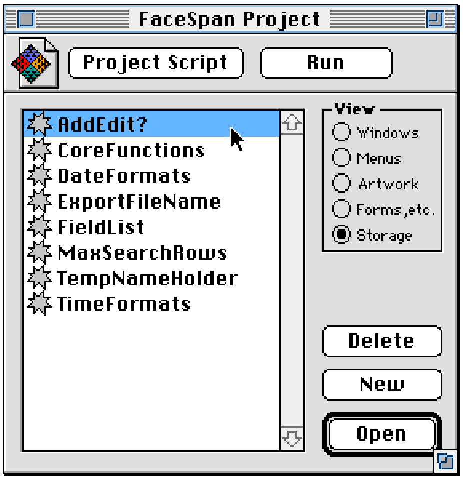


View: ( ) Windows ( ) Menus ( ) Artwork (● Forms,etc.) ( ) Storage

3D Buttons
3D Icon Button
Digit
DisplayBooleans
DisplayDates
DisplayNumbers
DisplayPassword
ListWithPictures
MenuOfPatterns
Only1Line
Only5Alphas
OnlyDigits
Slider

Storage View

A storage item is a piece of data kept in persistent storage within a project. Storage items are created and edited in FaceSpan. You can find out more about them by reading Chapter 5: “The Storage Item Editor.”



Project Window Buttons

The Delete, New, Open, and Import buttons— in combination with Edit menu commands—permit you to modify any editable project resources. The operations you can perform depend on the resource type selected with the View radio buttons. Clicking an item once selects it. Double-clicking an editable item opens its editor.

➤ Window templates an menu templates can be:
Created by using the New button
Opened for editing by using the Open button
Deleted by using the Delete button

➤ Artwork resources can be:
Imported from any other Macintosh file by using the Import button
Renamed by using the Open button
Deleted by using the Delete button

➤ Form resources and key filters can be:
Imported from other FaceSpan projects using the Import button
Deleted by using the Delete button
and, their documentation viewed by using the Open button

➤ Storage items can be:
Created by using the New button (or by script)
Opened for editing by using the Open button
Deleted by using the Delete button

About Copyrights

Remember that many pictures, movies, desktop patterns, cursors, and other resources are not yours to distribute. Some may be distributed with permission, others require that you acknowledge the author, pay a fee, and so on. Do not include a resource in your project unless you know that it is permissible to do so. Even without a copyright attribution, the author of a resource still holds the copyright.

Instructions

Using File menu and Edit menu commands

While the Project Window is active, all of the commands in the File menu and Edit menu pertain to the project. You can use the File menu to create and save projects, to revert an edited project to the state in which it was last saved, to convert a project to an application, and to print the project's contents. You can use the Edit menu to cut, copy, paste, clear, and duplicate the resources listed in the Project Window listbox.

Creating a new project

Choose New Project from the File menu

FaceSpan creates a new "Untitled" project document and displays its Project Window.

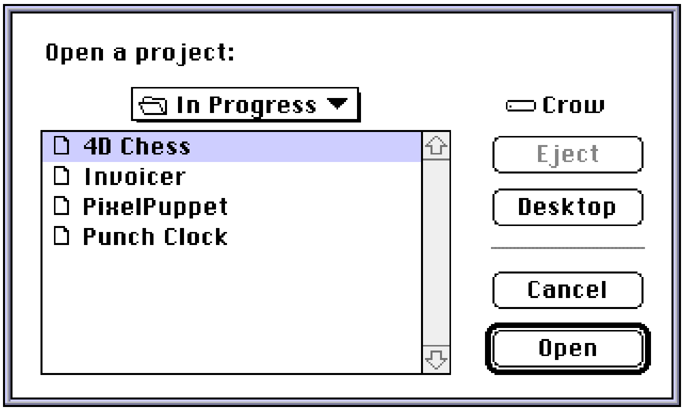
Note

➤ If you have made a customized default project, FaceSpan uses it as a template for each new "Untitled" project document.• FaceSpan also creates a new project each time you run it by double-clicking its Finder icon.

Making a default project

You can create your own customized default project by saving a project document with the name, "Default Project" and placing it in the same folder as FaceSpan. When FaceSpan is asked to make a new "Untitled" project, it will look for a customized "Default Project" in its folder. If the project is found, FaceSpan will use a copy of this project as the new project. If such a project is not found, FaceSpan will create and display a very slimmed down default project.

Opening an existing project or editable application



While FaceSpan is open, choose Open Project… from the File menu, then locate the project to be opened in the Open File dialog.

FaceSpan displays the Project Window of the selected project.

or

If FaceSpan is not open, double-click your project document's desktop icon.

FaceSpan opens and displays the Project Window of the selected project.

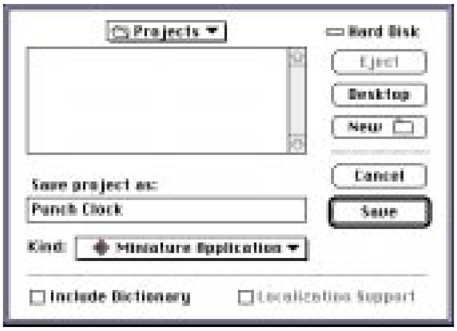
Saving a project

Choose Save Project from the File menu.

FaceSpan saves the project under its current name. If the currently active project has not yet been saved, FaceSpan displays the Save Project As dialog, with which you can save the project in editable form under a new name, optionally as a Miniature or Complete Application.

Table of Contents Index

Saving a project as an editable application



Save project as: Punch Clock
Kind: Miniature Application

1 Choose Save Project As… from the File menu.

FaceSpan displays the Save Project As dialog.

2 Enter the name under which the document will be saved.

3 From the Kind pop-up menu, choose the format in which you want to save the document:

Miniature Application	The document will be saved as an application that can be run on any Macintosh on which the FaceSpan extension and AppleScript (version 1.1 or later) have been installed.
Complete Application	The document will be saved as a stand-alone application that can be run on any Macintosh on which AppleScript (version 1.1 or later) has been installed.

4 If you chose to save the document as a Miniature or Complete application, you can click the Include Dictionary checkbox to include the dictionary.

Excluding the dictionary saves approximately 16K of disk space, but hinders the scripting of that application by other applications.

5 Click the Save button.

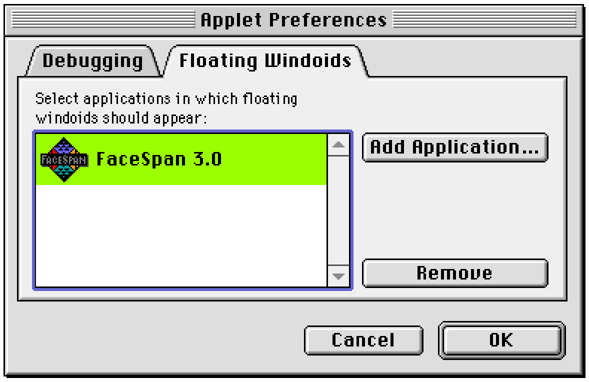
Saving a Project as a Floating Window

You can save applications as windows that float on top of other applications. When an application is active, the window appears. When the application is not active, the window disappears.

➤ To save a project as a floating window:

1 Save the project as an application.

2 With the Project window active, choose **Applet Settings** from the Edit menu to open the Applet Preferences dialog box.



Select applications in which floating windoids should appear:

FaceSpan 3.0

3 Click **Add Application** to display a directory dialog box.

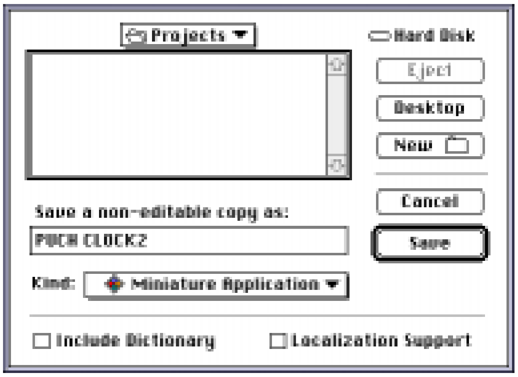
4 Locate and select the application on which you want your project to float.

Note Your application can float on more than one application. Repeat steps 3 and 4 to select additional applications.

5 Click **OK**.

When you launch the application, it will float on top of the application you selected.

Saving a project as a non-editable, run-only application



Save a non-editable copy as: PUCH CLOCK2

Kind: Miniature Application

1 Choose Save As Run Only… from the File menu.

FaceSpan displays the Save As Run Only dialog.

2 Enter the name under which the document will be saved; this name cannot be the same as the current name of the project.

3 From the Kind pop-up menu, choose the format in which you want to save the document:

Miniature Application	The document will be saved as an application that can be run on any Macintosh on which the FaceSpan extension and AppleScript (version 1.1 or later) have been installed.
Complete Application	The document will be saved as a stand-alone application that can be run on any Macintosh on which AppleScript (version 1.1 or later) has been installed.

4 If you want to include the dictionary, click the Include Dictionary checkbox.

Excluding the dictionary saves approximately 16K of disk space, but hinders the scripting of that application by other applications.

5 If you want Localization Support, click the Localization Support checkbox.

When Localization is selected, and the application is saved, all names of menus, textboxes, titlebars, buttons, etc. are stored into the resource fork of your application. This is to accommodate the use of AppleGlot for Language translation.

6 Click the Save button.

Notes

➤ The run-only version of an application can no longer be edited.

➤ When an application is saved as run-only, it does not contain the dictionary. The effect is that the application appears to not be scriptable, when it actually *is* scriptable.

Reverting to the last saved version of a project

Choose Revert Project from the File menu.

FaceSpan discards all changes made since the last time the project was saved.

Note

Changes made to a project become permanent when the project is saved. A project cannot be reverted once it has been saved.

Editing a project's script

Click the Project Script button.

The Script Editor opens to display the project script.

Note

➤ See Chapter 6 for more information about FaceSpan's script editor➤.

Running a project

Click the Run button.

FaceSpan hides all open window templates, compiles any uncompiled scripts, and runs the project's script.

The Run button changes into a Stop button with which the run can be halted.

Stopping a running project

1 Click the running project's Project Window to make it active.

2 Click the Stop button.

FaceSpan immediately halts the running script, closes all windows opened during the run of the project, and re-displays all open window templates.

The previously Stop button changes back into the Run button.

Finding an existing project resource

1 Click the View radio button for the appropriate type of resource.

2 Locate the resource in the list.

(For example, if you’re searching for a particular window template, click the Windows View radio button, then locate the window template's name in the list.)

Cutting a resource from another project

1 Click the appropriate View radio button in the Project Window listbox.

2 Select the name of a window template, menu template, storage item, artwork, or form resource you want to cut.

3 Choose Cut from the Edit menu.

The selected item is copied to the clipboard and deleted from the list.

Copying a resource from another project

1 Click the appropriate View radio button in the Project Window.

2 Select the name of a window template, menu template, storage item, artwork, or form definition resource you want to copy.

3 Choose Copy from the Edit menu.

The selected item is copied to the clipboard.

Note

➤ Artwork can be copied from other sources such as another application or a scrapbook.

Pasting a resource from another project

1 Copy or cut a window template, menu template, storage item, artwork, or form resource from the source project as described above.

2 Click the Project Window of the destination project to make it active.

3 Choose Paste from the Edit menu.

4 If a resource with the same name already exists, a dialog will ask you to give the pasted resource a unique name; type a unique name, then click OK.

The item on the clipboard is pasted into the active project document. FaceSpan automatically adjusts the View radio buttons to the resource type of the pasted item.

Notes

➤ When copying windows or window items, do not close the project copied from, before pasting into the destination project, or you will lose dependent resources.

➤ Form definition resources should be copied and pasted only from other FaceSpan projects.

Duplicating a resource

1 You can select the resource using the Project Window listbox, copy it, then paste it back into the same project.

or

Select the resource, then choose the Duplicate command from the Edit menu.

A dialog asks you to give the new resource a unique name.

2 Type a unique name for the resource into the dialog box.

The name of the new resource is displayed in the Project Window listbox.

Deleting a resource

1 Click the View radio button for the type of resource to be deleted.

2 Select the name of the resource to be deleted.

Table of Contents Index

3 Click the Delete button.

or

Choose the Clear command from the Edit menu.

FaceSpan displays a confirmation dialog.

4 Click the Delete button.

The selected item is now deleted from the project.

Creating a new window or menu template

1 Select the View radio button for the type of resource to be created.

2 Click the New button.

FaceSpan opens an editor for a new, “Untitled” item of the selected type.

Editing an existing window or menu template

1 Click the View radio button for the type of resource to be edited.

A list naming existing resources of the select type is displayed.

2 Double-click the name of a window or menu template…

or

Select the name of a window or menu template, then click the Open button.

The appropriate editor opens.

Importing artwork: pictures, icons, color patterns, and cursors

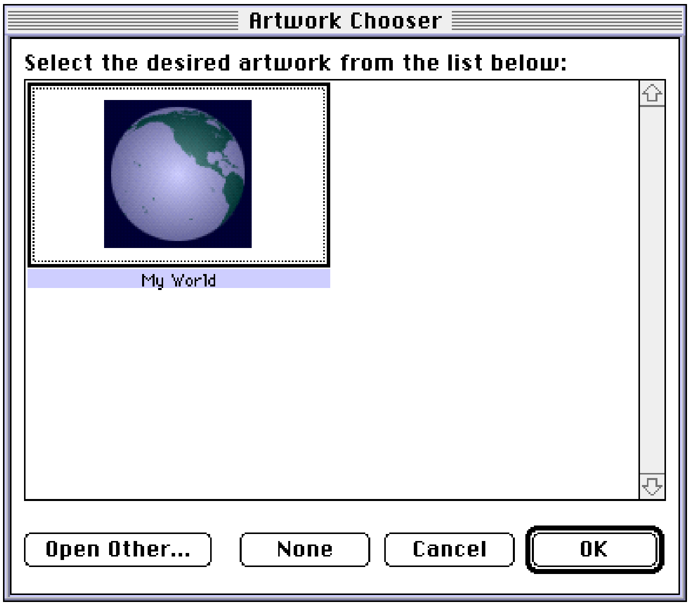
1 Click the Artwork View radio button.

2 Click the Import… button.

FaceSpan displays an Open File dialog.

3 Locate and select a project, scrapbook, application file, or document containing artwork to be imported, then click the OK button.

FaceSpan displays the Artwork Chooser dialog, which contains a scrolling list of the artwork belonging to the opened file.



4 Optionally, click the Artwork Chooser's Open Other… button to open a different file.

5 Select artwork in the Artwork Chooser's scrolling list, then click the OK button.

The selected resources are copied into your project.

Renaming existing artwork

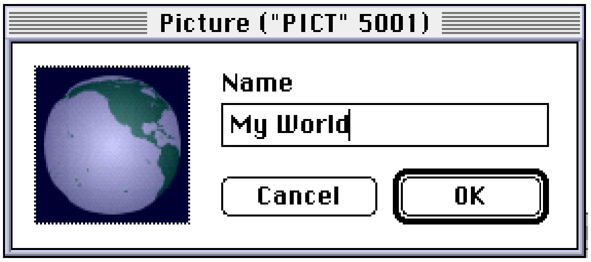
1 Click the Artwork View radio button.

2 Double-click the artwork in the list…

or

Select the artwork in the list and click the Open button.

The Resource Name dialog opens.

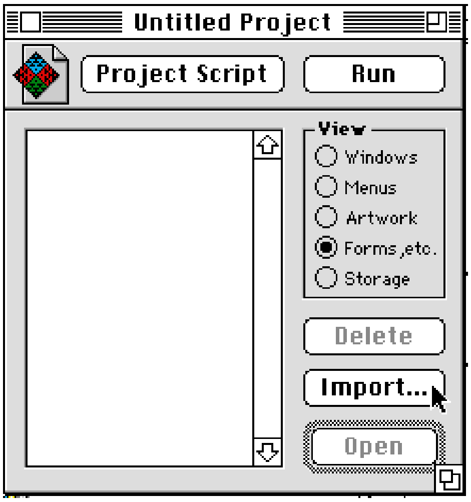


3 Enter the new name for the artwork in the dialog's Name textbox.

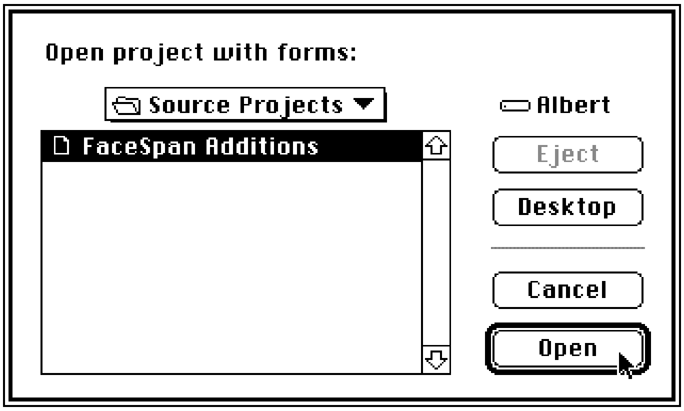
4 Click the dialog's OK button.

Importing form definition resources and scripting additions:

1 While the Forms, etc. View is selected in the Project Window, click the Import button.

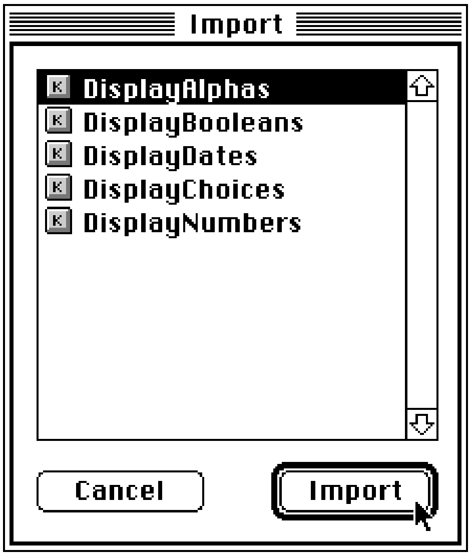


A standard Open dialog displays.



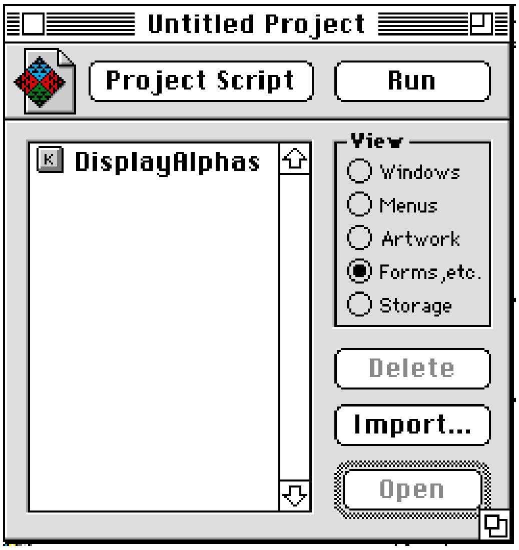
2 Select the name of the project or the name of the scripting addition you want to import, then click OK.

An Import dialog displays.



3 Select the name of the form or scripting addition you want to import, then click the Import button.

The name of the selected form definition resource or scripting addition is added to the Forms, etc. View of the Project Window listbox.



Viewing documentation of form resource

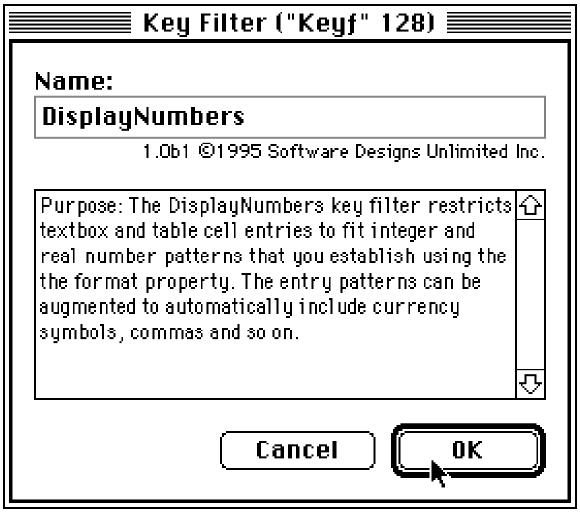
1 Click the View radio button for “Forms, etc.”

2 Double-click the form resource in the list.

or

Select the form resource in the list and click the Open button.

The Form Documentation dialog opens, displaying documentation for the selected type of form.



Purpose: The DisplayNumbers key filter restricts textbox and table cell entries to fit integer and real number patterns that you establish using the the format property. The entry patterns can be augmented to automatically include currency symbols, commas and so on.

3 Click the OK or Cancel button to close the dialog.

Creating a new storage item

1 Select the Storage View radio button.

2 Click the New button.

FaceSpan opens an editor for a new, untitled storage item.

Editing an existing storage item

1 Click the Storage View radio button.

A list naming existing storage items is displayed.

2 Double-click the name of a storage item in the list.

or

Select a name, then click the Open button.

The Storage Item Editor opens for the selected item.

Tips: On-screen position of the Project Window

FaceSpan keeps track, from one session to the next, of information such as the last on-screen position of window templates, the Property Bar, the Tool Palette, and the Project Window. This information is stored in a preference file inside the Preferences folder of the System folder, so that settings can be restored when FaceSpan is next opened.

If you want FaceSpan to return to its default settings, you can drag its preferences file to the trash, or you can hold both the Option Key and Command Key while FaceSpan is in the process of launching.

Chapter 3:

The Window Editor

Contents:

Table of Contents Index

The Window Editor

You should read this chapter if you want to know:

➤ how to use FaceSpan's tools to create user interface windows and window items

Inside the Window Editor, you can create, design, and edit window templates. A project's user interface is made up of three classes of windows: document windows, modal dialogs, and floating windoids; each can be controlled through scripts. Though a project's windows perform as a coordinated user interface while the project is running, each window is first created as a template—a model of the run-time window—and is then edited as an individual unit in the Window Editor. Once a window template is created, FaceSpan then saves a description of it, and uses that description to create the actual window object at runtime.

When you create a new window, FaceSpan displays a blank, modifiable window template, and two windoids—the Property Bar and the Tool Palette, collectively referred to as the Window Editor. You can use the Tool Palette to create, enter text into, select, and arrange window items. Using the Property Bar, you can adjust the properties of the window and its items.

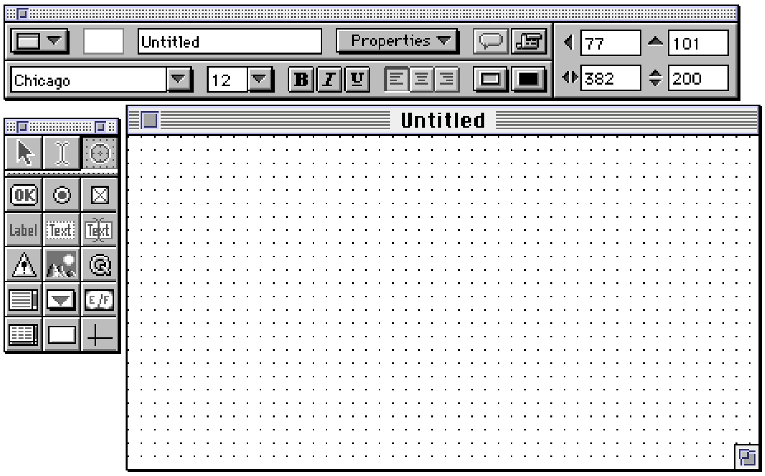
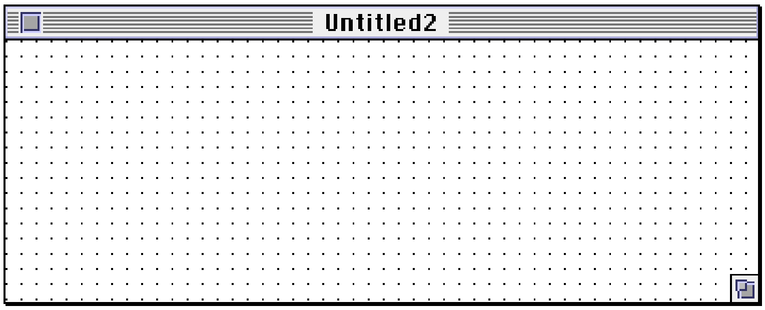


Table of Contents
Index

While the Window Editor is active, you can use the File menu's commands to close a window template, to revert and save an edited window template to the state in which it was last saved, and to print reports of the window template's contents. Design-time editing done directly from the Window Editor can include accessing the active window template's script, creating or modifying window items, and accessing a selected window item's scripts. In addition to providing a window design environment, the Window Editor allows you to test the way the window and its window items will respond to user input during runtime.

The Window Template



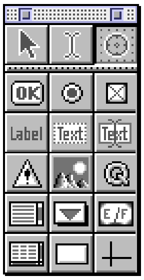
Once a window template has been created, it acts as a container for its window items and for scripts. When a window template is copied from one project and pasted into another, its window items and scripts are copied and pasted with it.

The **`class`** property of the window template is set by default to document window, so that it initially looks like a standard Macintosh document window containing a title bar, a close box, and a resize box.

At any point in the editing process, you can change the appearance and behavior of a window template by defining it as a document window, modal dialog, or floating windoid, and you can remove any of the control boxes by adjusting the window's **`closeable`**, **`resizable`**, **`zoomable`** and **`titled`** properties.

The dotted portion of the window template is the area in which window items can be created and arranged. The dots mark an 8-pixel-by-8-pixel grid. If the Snap To Grid command in the Object menu is turned on during editing, the corners of window items being drawn, moved, or resized automatically snap to points on the grid.

The Tool Palette



The Tool Palette provides three Cursor tools for manipulating objects in the window template and fifteen Object Maker tools for creating new window items.

Arrow tool

Selecting the Arrow tool puts the active window template into Play Mode. During Play Mode, the window template’s grid disappears and the template behaves much as it will while the project is running. You can use the Arrow tool to perform initial testing of the way the window and window items respond to user input. For example, during Play Mode you can click push buttons, select listbox items, choose from popups, and so on, just like the application user would during runtime.

In addition, you can use Play Mode to test how items look or behave using different properties. If you click a selectable window item (or choose any item from the Selected Item popup in the Property Bar), the Property Bar’s display is set to that item. You can then change the item’s properties and observe the results of those changes, all without leaving Play Mode. Changes made to the window template and its items during Play Mode are kept when you return the window template to Edit Mode.

Note

The project script is not run in Play Mode, so **`open window`** commands are not executed, and some application and window script properties may not be initialized. See Chapter 8: “The Testing Environment” for more information about testing during Play and Run Mode.

I-beam tool

3

Using the I-beam tool, you can enter a title or text content in window items that display text (such as listboxes, radio buttons, checkboxes, and labels and so on). The Tab key moves the I-beam tool from window item to window item in the window template in lowest to highest index order (the **`index`** property indicates the layer that each window item occupies in the window template). You can edit the **`title`** or **`contents`** text of a window item by selecting it with the I-beam tool, then typing text at the appropriate location.

Object Mover tool

With the Object Mover tool, you can select, move, and resize window items in the window template. When you select a window item with the Object Mover tool, the window item is surrounded by a selection marquee, and its properties are displayed in the Property Bar. You can select multiple window items with the Object Mover tool by holding down the Shift key while clicking them, or you can drag a selection rectangle around them. While the Object Mover tool is chosen, the Tab key moves the selection from window item to window item in lowest to highest index order.

Viewfinder Crosshair Resize arrow

The Object Mover tool changes to a viewfinder when moved over a window item. You can click to select the window item, or drag to move it.

The Object Mover tool changes to a crosshair when moved over an area of the window template that contains no window items. You can click to select the window itself while deselecting all window items, or you can drag a rectangle around a group of window items to select the group.

The Object Mover tool changes to a resize arrow when moved over the corner of a window item. You can click-drag the arrow to resize the window item.

Note

➤ The Object Mover tool also changes to a crosshair when moved over a window item whose position has been locked (by selecting the item, then choosing the Lock Position command from the Object menu. You can select a window item whose position is locked by clicking it, but you cannot move or resize it until its position is unlocked.

Object Maker Tools

There are fifteen Object Maker tools you can use to make window items including: push button, radio button, checkbox, label, textbox, editable textbox, icon, pictbox, movie, listbox, popup, gauge, table, box and graphic line.

push button	radio button	checkbox
label	textbox	editable textbox
icon	pictbox	movie
listbox	popup	gauge
table	box	graphic line

You can make a window item by selecting the appropriate Object Maker tool, then either clicking and dragging to create the new window item in its window, or—if Drag and Drop is installed on your Macintosh—dragging and dropping the Object Maker tool's icon onto the window template. If the Snap To Size and Snap To Grid commands in the Object menu are turned on,

Table of Contents Index

FaceSpan automatically aligns the window item's position to the grid and adjusts its width and height to "natural" dimensions (in accordance with *Macintosh Human Interface Guidelines*).

Once you've created a new window item, the Object Maker tool changes to the Object Mover tool so that you can continue editing.

Hint To retain the same Object Maker tool after creating a window item—if, for example, you wish to create several window items of the same class—hold down the Command key while creating the item. (You can also hold down Option and drag to clone a window item.)

Drag and Drop support in the Window Editor

If you have Drag and Drop support installed on your Macintosh computer, you can create an object by selecting the Object Maker tool, then dragging out a rectangle in the window template. This works several ways; you can Drag and Drop from the Tool Palette, or Drag and Drop from the desktop, or Drag and Drop from another file.

Drag and Drop from the Tool Palette

You can drag an Object Maker Tool's icon from the Tool Palette, then drop it onto a window template to create the selected class of object.

Drag and Drop from the desktop

You can Drag and Drop selected text and picture clippings from the desktop onto a window template being edited. When you do, the appropriate object is automatically created and filled with the selected content. (In the case of pictures, the picture is automatically added to the project's artwork.)

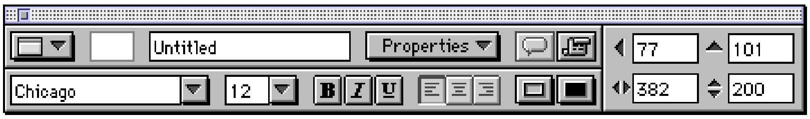
Note

➤ If a picture is dragged and dropped onto a window during run-time, an image of the picture is shown in the window, but the actual picture is not saved into the project.

Drag and Drop from another file

You can select and drag text from any other source and drop it onto the window template being edited. A textbox is automatically created and filled with the selected content.

The Property Bar

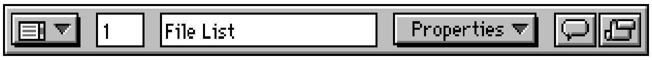


The Property Bar displays the values of common properties of a selected window item (or, of the window template if no window item is selected) and permits you to change the values as needed.

Note

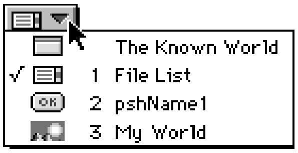
➤ While multiple items are selected, certain areas of the Property Bar—such as those used to open a script editor, or adjust properties—become blank or disabled, but the controls that remain enabled can be used to simultaneously change the applicable properties of all of the selected window items.

General Controls



The Property Bar's general controls include (from left to right): Selected Item popup, Window Item Index textbox, the Item Name textbox, Properties popup, Balloon Help button, and Object Script button. You can use these general controls to identify the item that is currently selected, as well as access its script, Balloon Help text, and class-specific properties.

Selected Item popup



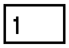
The Selected Item popup (pop-up menu) lists the **name** of the active window template and its window items. Beside each name in the list is an icon—indicating the class of each window item, and a number—indicating the number of the layer that the window item occupies in the window template. A check mark next to an icon indicates the currently selected item. You can use the popup to select any window item or the window template itself.

Hint This is an easy way to select invisible items or items off the screen.

Note

➤You can use the Tab key to move the selection from window item to window item in ascending index order, or use the Select… command from the Object menu.

Window Item Index textbox



This textbox displays the **`index`** of the selected window item. The window item's **`index`** is the number of the layer that the item occupies in the window template. You can change the **`index`** of the selected window item by typing a different number in the index textbox.

Since index numbers are sequential, changing the **`index`** of a window item always changes the index numbers of other window items.

Item Name textbox

This textbox displays the **`name`** property of the selected window item (or of the window itself, if no window item is selected). You can change the **`name`** of the selected window item by typing a new name in the textbox.

Note

➤If you change the name of a window template, the Windows View of Project Window listbox updates when the window is closed or the project is saved.

Properties popup

When clicked, this popup displays a menu of the properties specific to the class of the selected window item (or of the window itself, if no window item is selected). You can change the value of a property by choosing the property's name in the menu. For some properties, a submenu displays so you can then choose the value you want.

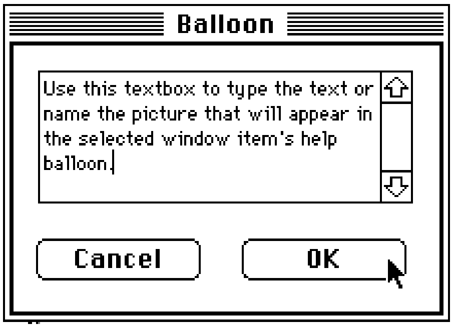
Properties can also be set using the window template or window item's Object Information dialog, and by script. Object Information dialogs are discussed later in this chapter.

Note

➤You can learn more about scripting by reading Part II of this guide, titled "Application Development." For complete information about classes and properties, refer to Part III: "FaceSpan Object and Language Reference."

Balloon Help button

When clicked, the Balloon Help button opens the Balloon Help Editor, and displays the text that will appear in the selected window item's help balloon.



Use this textbox to type the text or name the picture that will appear in the selected window item's help balloon.

You can change the content of a window item's help balloon by typing or pasting in different text, then clicking OK. When there is content in the selected object's balloon help, the Balloon Help button's icon appears to have text in it.

Hint In addition to balloons containing text, you can also create picture balloons by typing the name of a picture—listed in the project's artwork resources—into the Balloon Help Editor.

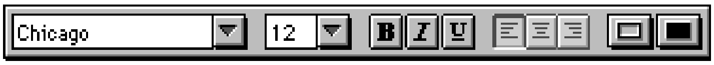
Object Script button

When clicked, the Object Script button opens an editor containing the script of the selected window item (or of the window itself, if no window item is selected). If a selected item has a script, the Object Script button’s icon appears to have text in it.

Note

➤ To allow the message hierarchy to pass through to the project script, window templates are always considered to have scripts, and so, always display an Object Script button icon having text.

Text Property controls



The Property Bar’s text property controls include (from left to right): Font textbox and popup, Size textbox and popup, Style buttons (bold, italic, and underline), Justification buttons (left, center, and right), Pen Color popup, and Fill Color popup.

Note

➤ The **`font`**, **`size`**, and **`style`** properties of each newly created window item default to the corresponding property of the window template itself; they will match this setting until you change them.

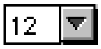
Font textbox and popup

The Font textbox and popup display the **`font`** property of the selected window item. You can change its **`font`** by typing a different text font name in the textbox, or by choosing a different font from the popup, which lists all the fonts installed on your Macintosh.

Note

➤ The **`font`** property of each newly-created window item defaults to the **`font`** property of the window template itself, and will continue to match this setting unless it is explicitly changed.

Size textbox and popup

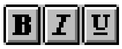


The Size textbox and popup display the **`size`** property of the selected window item. You can change its **`size`** by typing a different number in the textbox, or clicking the popup and dragging to choose a different size.

Note

➤ The **`size`** property of each newly-created window item defaults to the **`size`** property of the window template itself, and will continue to match this setting unless it is explicitly changed.

Style buttons (bold, italic, and underline)



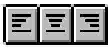
The Style buttons display the styles (if any) of the selected window item. You can click any Style button to apply or remove a style.

Notes

➤ FaceSpan also supports outline, shadow, condensed, extended, and group styles. They can be applied using commands from the Style menu.

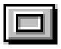
➤ The **`style`** property of each newly-created window item defaults to the **`style`** property of the window template itself, and will continue to match this setting unless it is explicitly changed.

Justification buttons (left, center, and right)



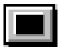
The Justification buttons display the **`justification`** property of a selected window item that has a **`justification`** property. You can change the **`justification`** by clicking a different button. Only one justification button can be used for each window item.

Pen Color popup



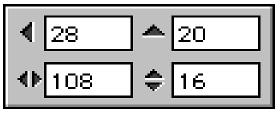
You can use this popup to set the **`pen color`** property of the selected window item (or of the window itself, if no window item is selected). **`Pen color`** determines the color in which the object's foreground (outlines and title) will be drawn. You do not have to assign a pen color; the default is black.

Fill Color popup



You can use this popup to set the **`fill color`** property of the selected window item (or of the window itself, if no window item is selected). **`Fill color`** determines the color with which the object's background will be filled. You do not have to assign a fill color; the default is white.

Position, Width, and Height Property controls

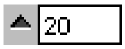


These four textboxes on the Property Bar indicate the **`position`** and **`size`** of the selected window item.

Left Position textbox

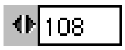
The Left Position textbox displays the distance, in pixels, from the left edge of the window template (0) to the left edge of the selected window item. You can change the distance by typing a different value into the textbox.

Top Offset textbox


20

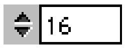
The Top Position textbox displays the distance, in pixels, from the top edge of the window template (0) to the top edge of the selected window item. You can change the distance by typing a different value into the textbox.

Width textbox


108

The Width textbox displays the width, in pixels, of the selected window item. You can change the width property of the window item by typing a different value into the textbox.

Height textbox

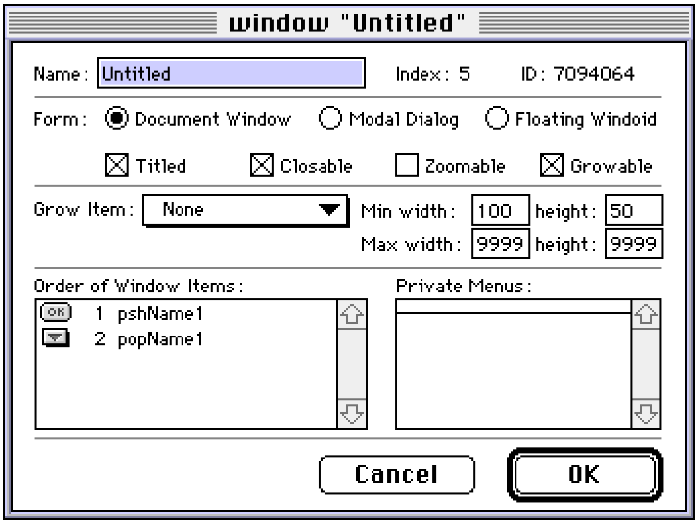

16

The Height textbox displays the **`height`**, in pixels, of the selected window item. You can change the **`height`** property of the window item by typing a different value into the textbox.

Table of Contents   Index

Object Information dialogs

Object Information dialogs allow you to inspect and modify many of the same window or window item properties as the Property Bar, but provide larger text-editing areas and more informative displays of current property values. Some Object Information dialogs display properties not shown in the Property Bar. The layout and number of editable fields contained in each dialog depends on the type of object selected. Once the Object Information dialog is open, you can use the Tab key or the mouse to move from one field to another.



window "Untitled"

Name: Untitled   Index: 5   ID: 7094064

Form: [✓ Document Window] Modal Dialog Floating Windoid

[✓ Titled] [✓ Closable] Zoomable [✓ Growable]

Grow Item: None   Min width: 100 height: 50
Max width: 9999 height: 9999

Order of Window Items:
1 pshName1
2 popName1

Private Menus:

You can display the Object Information dialog of a window template or window item by selecting the object, then choosing the Object Info command from the Object menu. You can also display an Object Information dialog by double-clicking a window item in an active window template, or selecting an item, then using the keyboard shortcut (Command) key - I.

Hint Some objects—windows, textboxes, listboxes, and popups—have expandable Object Information dialogs. To automatically display an Object Information dialog in expanded form, hold the Option key while double-clicking the object.

Instructions

Using File menu commands

All window editing centers around the Window Editor. While the Window Editor is active, you can use the File menu's commands to close a window, to save a window template, to revert an edited window template to the state in which it was last saved, and to print reports of the window template's contents.

Creating a new window template

1 Highlight the Windows View radio button in the Project Window.

2 Click the New button.

FaceSpan creates a new, "Untitled" window template and displays the Property Bar and Tool Palette.

Making a default window template

You can create a pre-configured "Untitled" window template by creating a window template, then naming it "<New>." Once the <New> window is added to the project's list of windows, it will serve as a template whenever you click the New button while the Project Window is in Windows View.

Opening an existing window template

1 Click the Windows View radio button in the Project Window.

2 Double-click the name of the window template in the list.

   or

   Click the name of the window template in the list to select it, then click the Open… button.

FaceSpan opens the Window Editor for the selected window template.

Saving a window template

Choose Close Window from the File menu while the Window Editor is active.

FaceSpan closes the Window Editor, and saves all changes to the edited window template.

Note

➤ Changes made to a window template can be reverted with the File menu's Revert Project command, any time before the project is saved.

Reverting to the last saved version of a window template

While the Window Editor is still open, choose Revert Window from the File menu.

FaceSpan undoes all changes made to the window template since it was opened for editing.

Note

➤ Changes made to a window template can be reverted at any time before the project has been saved.

Selecting the window itself

Click an open space in the window template with the Object Mover tool.

or

Select the window name in the Property Bar’s Window Item Name textbox.

Any previously selected window items are deselected, the window itself is selected, and its property values are displayed in the Property Bar.

Creating a new window item in the window template

1 Click the appropriate Object Maker tool.

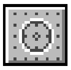
2 Click at the location in the window template where you wish to place the new window item.

3 Drag to draw a rectangle the size of the window item.

If the Object menu's Snap To Size and Snap To Grid commands are check marked true, FaceSpan will automatically align the window item's position to the grid and adjust its width and height to “natural” dimensions —which vary according to the class of the selected window item.

Creating a new window item in the window template using Drag and Drop

1 Click and hold the appropriate Object Maker tool.


Object mover tool

2 Drag the outline of the Object Maker tool to the window template, then drop it (release the mouse button).

Notes

➤ When a new item is created using Drag and Drop, FaceSpan adjust its width and height to “natural” dimensions —which vary according to the class of the selected window item.

➤ If the Object menu's Snap To Grid command is check marked true, FaceSpan will automatically align the window item's position to the grid.

Selecting a window item using the mouse

1 Choose the Object Mover tool from the Tool Palette.

2 Click the window item in the window template with the Object Mover tool.

A selection marquee appears around the selected window item, and the window item's property values are displayed in the Property Bar.

Selecting multiple items

1 Choose the Object Mover tool from the Tool Palette.

2 While holding the Command key down, drag the items you want to select in the window template.

A selection marquee appears around the selected window items.

Selecting a window item using the Property Bar

1 Click the downward arrow of the Selected Item popup in the Property Bar.

A list naming the window template and its items displays.

Table of Contents Index

2 Choose the window item’s name from the list.

A selection marquee appears around the selected window item, and the window item's property values are displayed in the Property Bar.

Selecting multiple window items

1 Choose the Object Mover tool from the Tool Palette.

2 Click the window items in the window template while holding down the Shift key.

or

Drag a rectangle completely around the window items to be selected.

A selection marquee appears around the selected window items; the Property Bar displays blank values to indicate a multiple selection.

Deselecting the selected window items

Click a selected window item while holding down the Shift key.

or

Click an unselected window item.

or

Choose an unselected window item from the Property Bar’s Selected Item popup.

or

Click a blank area of the window template.

Cutting a window item

1 Select the window item to be cut by clicking it with the Object Mover tool or by choosing it in the Selected Item popup.

2 Choose Cut Items from the Edit menu.

The selected window item is copied to the clipboard and deleted from the window template.

Copying a window item

1 Select the window item to be copied by clicking it with the Object Mover tool.

2 Choose Copy Items from the Edit menu.

The selected window item is copied to the clipboard.

Pasting a window item

1 Copy or cut a window item.

2 Activate the window template into which you want to paste the copied item by clicking it with the Object Mover tool.

3 Choose Paste Items from the Edit menu.

The window item is pasted from the clipboard into the window template.

Duplicating a window item

1 Select the window item(s) to be duplicated.

2 Choose Duplicate Items from the Edit menu.

The selected window item(s) are duplicated and deselected, and the duplicates become the new selection.

Cloning a window item

1 Select the window item(s) to be cloned.

2 Click and drag the selected window item(s) while holding down the Option key.

Clones of the selected window item(s) appear under the cursor, the original window items are deselected, and the clones are selected. The clones may now be dragged away from the originals.

Deleting a window item

1 Select the window item to be deleted by clicking it with the Object Mover tool, or choosing it from the Selected Item popup.

2 Press the Delete key.

or

Choose Clear Items from the Edit menu.

The selected window item is deleted from the window template.

Setting a window item's properties using the Property Bar

1 Select the window item to be edited by clicking it with the Object Mover tool or choose it from the Selected Item popup.

The selected window item's properties are displayed in the Property Bar.

2 Use the Property Bar’s controls to adjust the window item's property values

Opening a window item’s Object Information

1 Select the window item to be edited by clicking it with the Object Mover tool or by choosing it from the Selected Item popup.

2 Double-click the window item or choose Object Info from the Object menu.

The selected item’s Object Information dialog opens.

Entering text into a window item

1 Choose the I-Beam tool from the Tool Palette.

2 Click the window item into which text is to be entered.

3 Type the text.

Note

➤ You can also use the item’s Object Information Dialog to enter text.

Testing window items and their scripts

Choose the Arrow tool from the Tool Palette.

The window template's grid disappears to indicate that it is in Play Mode. You may now click buttons, enter sample text in editable textboxes, play movies, and so on.

Note

➤ You can also use the item’s Object Information Dialog to enter text.

Returning the Window Editor to its default position

FaceSpan keeps track, from one session to the next, of information such as the last on-screen position of window templates, the Property Bar, the Tool Palette, and the Project Window. This information is stored in a preference file inside the Preferences folder of the System folder, so that settings can be restored when FaceSpan is next opened.

If you want FaceSpan to return to its default settings, you can drag its preferences file to the trash, or you can hold both the Option Key and Command Key while FaceSpan is in the process of launching.

Table of Contents Index

Tips for Designing Windows

Windows are the focal point for interaction with the application user. They should present your application's capabilities as clearly and consistently as possible. Here are a few points to consider when creating windows.

Be conservative

Doing things differently can be innovative, but sometimes it can also be disorienting or confusing. Remember that most users want software that is easy to use.

Adapt ideas that work

As a software user, you have probably encountered well-designed windows, as well as some that fall short in their appearance or their functionality. You may find it useful to evaluate the windows of applications you frequently use, noting what works and why.

Keep the design simple

Keep windows simple and purposeful by allowing adequate space around window elements, and grouping interface objects according to their function.

Be task-oriented

Draw the user's attention to the task at hand. Plan contrasts of color and line thickness to call attention to the most important items in your windows.

Order window items logically

In most Western languages, people read form left to right, top to bottom. Window items in the top left corner will probably be noticed first, so it makes sense to put frequently used interface elements there. Likewise, lesser or later used elements (like an OK button), should be placed toward the bottom right corner.

Make tasks simple

Try to choose combinations of window items that make the user's job easier. For example, if space permits, don't use a popup menu when a group of radio buttons would present the same choice more conveniently.

Be consistent

When creating applications with multiple windows, you can help your users to learn how various windows work by using similar sets of objects in similar arrangements to perform similar tasks.

Test your work

During the design process, looking at a window constantly can inhibit really “seeing” it. Have someone who has never seen the window before write down their first impression (without interaction from you) about what works, what could be improved, and why.

Table of Contents Index

Chapter 4:

The Menu Editor

Contents:

Table of Contents   Index

The Menu Editor

You should read this chapter if you want to know:

➤ how to create and edit run-time menus for your application

➤ how to create submenus for your application

➤ how to create contextual menus for your application

As with windows, you first create menus as templates. Inside FaceSpan's Menu Editor, you can create and edit menu templates. After each template, or model, of a menu is made, FaceSpan saves a description of the template and uses the description to create the actual menu at run time.

Each menu template has a **`name`**—the text that will appear on the menu bar, and **`menu items`**—the commands that appear in the menu when the user opens it by clicking the menu name. In turn, each menu item has a **`name`**, an optional **`mark`** character, an optional **`command key`** character, and an optional **submenu**. If you add a submenu to a menu item, the menu item cannot have a command key character.

You can also create contextual menus. The contextual menu templates are created with the Menu Editor, but they must be assigned to a button or other object in the Window Editor. Creating contextual windows is explained in this section. For more information on assigning contextual windows to an object, see *Assigning Contextual Menus to an Object* on page 113.

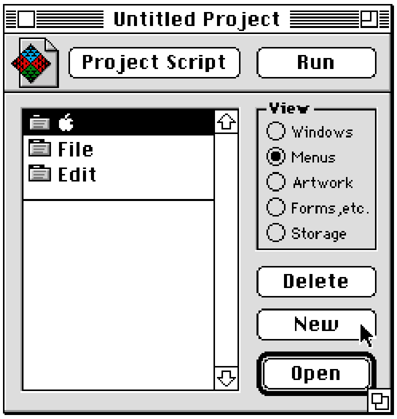
The menus added to the application's menu bar at run time can include menus associated with a particular window or with the project itself. Then, each time the user chooses an item from a menu, messages are sent to the **`frontmost`** window, and the handlers you add to the script of the window or project interpret the messages and take the appropriate actions.

FaceSpan's Menu Editor is a creation and editing environment in which you can construct menu templates by typing and clicking. You can display the menus in a project by clicking the Menus radio button in the Project Window.

The Menu Template

You can open an editor for an existing menu template by clicking the Menus radio button in the Project Window, then double-clicking one of the names displayed in the list.

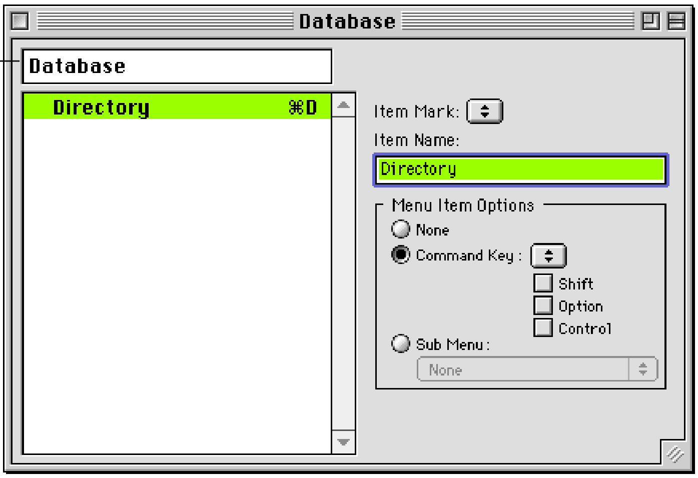
FaceSpan creates a new untitled menu template when you click New while the Project Window is in Menus View.



The Menu Name Textbox

You can use the Menu Name textbox to name or rename a menu template. The **name** you enter in the text box will be displayed on the menu bar of the Menu Editor.

Menu Name textbox



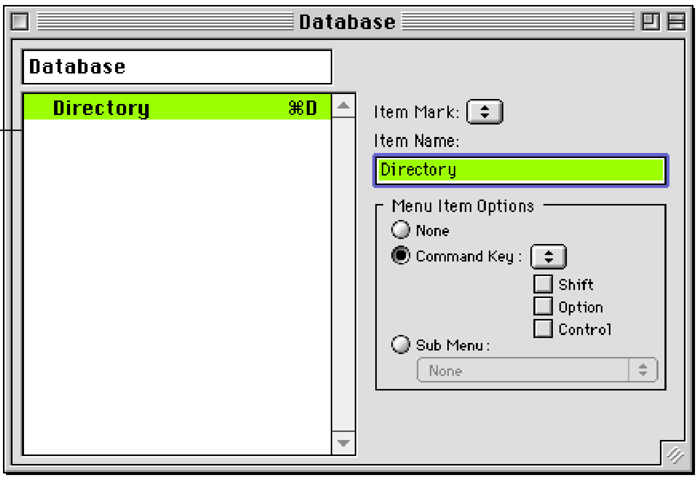
Note

➤You can press the Tab key to select the next existing menu item, or press the Return key to create a new menu item.

The Menu Item Name List Box

You can use the Menu Item Name list box to enter the **name** of each menu item to be displayed in the menu.

Menu Item Name list box

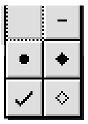


Note

➤Pressing Return creates another menu item after the current one. Any unnamed menu items are deleted when you close the Menu Editor.

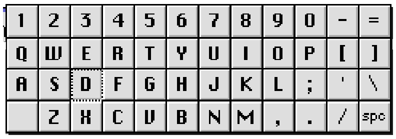
The Item Mark Pop-up Menu

You can assign a **mark** character to a menu item by selecting the menu item, then choosing a character from the Item Mark pop-up menu.



The Command Key Pop-up Menu

You can assign a **`command key`** equivalent to a menu item by selecting it, then clicking the Command Key radio button. Click the Shift, Option or Control checkboxes, if you want to use these in the command key. Click the Command Key pop-up menu to open the Keyboard dialog box. Select the quick key you want. The command key sequence appears to the right of the menu item in the Menu Item list box.



The Sub Menu Pop-up Menu

You can assign a sub menu to a menu item. Before you can assign a sub menu, you must create the sub menu in the Menu Editor. A menu item with a sub menu cannot have a command key. For more information on creating sub menus, see *Assigning a Sub Menu to a Menu Item* on page 111.

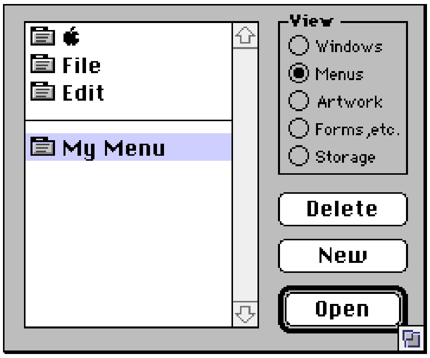
The Contextual Menu

You can create a contextual menu and assign it to an object in the Window Editor. You create a contextual menu as you would any other menu. The contextual menu must stay below the menu line in the Project Window so that it will not appear at the top of your program with all the other menus.

Once you've created the contextual menu, you can assign it to an object in the Window Editor. When a user presses the Control key and clicks the object, the menu appears. For more information on assigning contextual menus to objects, see *Assigning Contextual Menus to an Object* on page 113.

Changing the Menu Sequence

You can change menu sequence—the order in which menus appear on the menu bar—using the Project Window. When you click the Menus View radio button in the Project Window, the Project Window list box displays the names of all menu templates stored in the project. They appear in the order they will appear on the menu bar.



The Menus view of the Project Window list box contains a horizontal divider. Menu templates listed above the divider are associated with the project itself and are displayed on the menu bar whenever the project script is run. Menu templates listed below the divider are associated with specific windows in the project, and are displayed on the menu bar only while those windows are open (private menus); alternatively, the menu templates listed below the divider can be used as submenus assigned to other menu templates or as contextual menus assigned to display objects within windows.

Instructions

Creating a New Menu Template

1 Click the Menus View radio button in the Project Window.

2 Click the New button.

Opening an Editor for an Existing Menu Template

1 Click the Menus View radio button in the Project Window.

2 Double-click one of the menu names in the list.

The Menu Editor for that menu template is displayed.

Selecting a Menu Item

While the Menu Editor is open, click a menu item to select it.

or

Press the Tab key to move from menu item to menu item.

Inserting a New Menu Item Between Existing Menu Items

1 While the Menu Editor is open, select the menu item that you want the new menu item to appear after.

2 Press the Return or Enter key.

or

Choose Insert Menu Item from the Edit menu.

The new menu item now appears in the selected position.

Inserting a New Menu Item at the Top of a Menu

1 While the Menu Editor is open, select the Menu Name text box.

2 Press the Return or Enter key.

or

Choose Insert Menu Item from the Edit menu.

Moving a Menu Item to a Different Position

1 While the Menu Editor is open, select the menu item to be moved.

2 Drag the menu item to the desired position within the menu.

Creating a Divider Bar to Separate Two Groups of Menu Items

Create a menu item whose name is a single hyphen (dash) character.

FaceSpan will insert a divider bar in the menu item's position.

Note

➤ Divider bars can be selected, moved, copied, and pasted just like other menu items.

Deleting an Existing Menu Item

1 While the Menu Editor is open, select the menu item that you want to delete.

2 Choose Clear Menu Item from the Edit menu.

or

Delete the menu item's text.

Note

➤ When a Menu editor is closed, FaceSpan searches the menu for blank items and removes them.

Copying a Menu Item

1 While the Menu Editor is open, select the menu item to be copied.

2 Choose Copy Menu Item from the Edit menu or type ⌘–C.

Pasting a Menu Item

While the Menu Editor is open…

1 Select the menu item that the pasted menu item should appear before.

2 Choose Paste Menu Item from the Edit menu or enter ⌘–V.

Table of Contents Index

Moving the Insertion Mark Within the Menu Item Name Text Box

1 Use the Left Arrow and Right Arrow keys to move the insertion mark to the left or right one character at a time within the active Menu Item Name text box.

2 Use the Up Arrow key to move the insertion mark to the beginning of the Menu Item Name text box.

3 Use the Down Arrow key to move the insertion mark to the end of the Menu Item Name text box.

Associating a Menu Template with a Window

1 Choose the Menus View radio button in the Project Window.

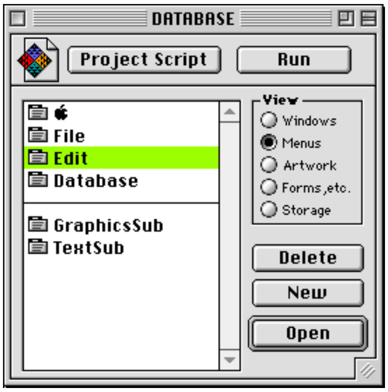
2 Drag the menu template's name to a position below the divider.

3 Add the menu template's name to the value of the window's private menus property (for example, by choosing Object Info from the Object menu, then dragging the menu template's name above the divider in the Object Info dialog's Private Menus listbox).

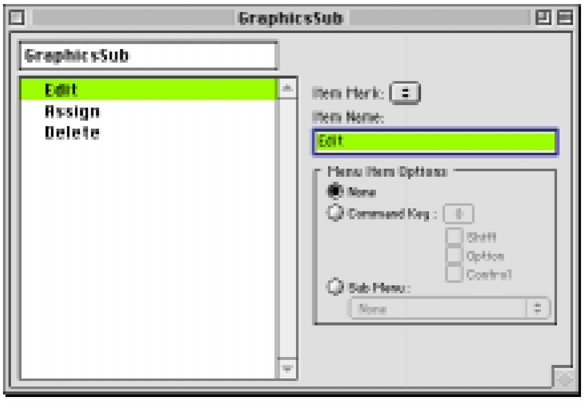
Assigning a Sub Menu to a Menu Item

You can assign a sub menu to a menu item. Before you can assign a sub menu, you must create it in the Menu Editor.

1 Open the Project window



2 Click New to open the Menu Editor.

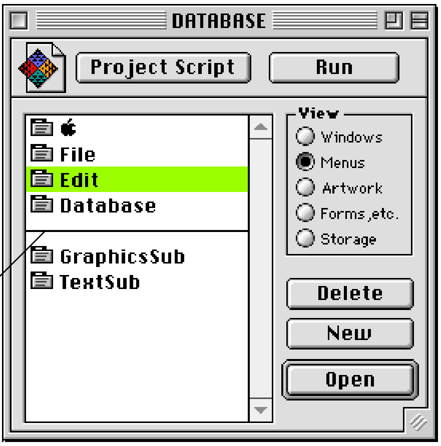


Item Mark:
Item Name: Edit
Menu Item Options: [✓ None] Command Key: Sub Menu: None

3 Name the sub menu and create the sub menu items.

4 Close the sub menu.

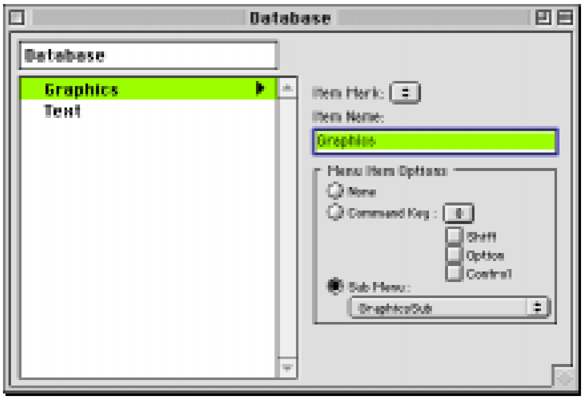
Important The sub menu you created must stay below the menu line in the Project Window so that it will not appear at the top of your program with all the other menus.



View: Windows [✓ Menus] Artwork Forms,etc. Storage

5 Click New to open the Menu Editor and create the menu with the menu item to which you want to assign the sub menu.

Table of Contents Index



6 Name the menu.

7 Create the menu item.

8 Click the Sub Menu radio button, then choose the sub menu from the Sub Menu pop-up menu.

The sub menu now appears when you select the menu item in the program.

Assigning Contextual Menus to an Object

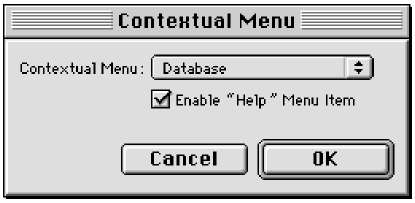
You can create a contextual menu and assign it to an object in the Window Editor. When a user presses the Control key and clicks the object, the menu appears.

1 Create a contextual menu template as you would any other menu template.

Important The contextual menu must stay below the menu line in the Project Window so that it will not appear at the top of your program with all the other menus.

2 In the Window Editor, select the object to which you want to assign a contextual menu.

3 Choose **Object Contextual Menu** from the Object menu to open the Contextual Menu dialog box.



4 From the Contextual Menu pop-up menu, choose the menu you want to assign to the object.

5 If you want to enable the Help menu, click the Enable “Help” checkbox.

Important To assign a script to the Help menu item of a contextual menu, add a help handler to the object script. To assign scripts to other menu items in a contextual menu, do so in the project script as you would any other menu.

6 Click **OK**.

The menu is now assigned to the object. When a user presses the Control key and clicks the object, the menu appears.

Tips for Naming Menus

Since new users probably won’t understand everything about your projects at first glance, the menus should explain your projects’ capabilities as clearly and consistently as possible. Here are a few points to consider when creating names for menus and menu items.

Syntax

In general, a menu’s name should be a noun that explicitly or collectively describes the items on which the menu’s commands operate, or a verb that collectively describes the operations that the menu’s commands perform. Menu item names should be verbs describing processes to be performed on the menu name’s noun, or phrases in the form:

verb + (optional adjective) + direct object.

Brevity

Try to keep the names of menus and menu items short; long menu names crowd and fragment the menu bar, and long menu item names result in wide menu pages that block large parts of the screen when they’re opened. If possible, menu names should be single words rather than phrases.

Clarity

Menu and menu item names should be self-explanatory. Avoid technical terms or jargon, and choose language that users can understand from everyday experience.

Consistency

Names of menus and menu items should be consistent with each other and with the terminology of the Finder’s menus. If several menu items apply the same process to different types of items, always refer to the process by the same term. If several menu items apply different processes to the same type of item, always refer to the item by the same term. When referring to processes or items mentioned in the Finder’s menus, use the Finder’s terms for them. Conversely, try to avoid using the Finder’s terminology in reference to different items or processes in your own menus.

Using Ellipsis

An ellipsis (Option key-“;”) should be appended to any menu item that summons a dialog that obtains more information from the user before a change.

Assigning Command Key Characters

When choosing command key characters for menu items, attempt to select characters that relate memorably or logically to their menu item names. However, avoid using characters from the Finder’s File and Edit menus, particularly C, V, X, S, O, W, P, and Z, except as they are used in the Finder.

Organizing the Menu Bar

Standard Finder menus (File, Edit…) should always appear in the positions they occupy in the Finder’s menu bar. Other menus should be organized hierarchically in the menu bar, with menus that influence larger items (such as documents) to the left and menus that influence smaller items (such as text selections) to the right.

Organizing Menus

Careful arrangement of menu items in your menus makes them understandable, safe, and easy to use. If a menu’s items can’t be arranged to satisfy the following guidelines, consider moving some items to a new or different menu.

Related Items

Group related items together by a common process that they perform, or a common object on which they operate.

Consecutive Items

If several menu items initiate related processes that should be performed in a particular order, arrange the items in that order.

Convenience

Position frequently-used items near the tops of menus, so users can use them easily.

Table of Contents Index

Safety

Items that permanently change data or terminate processes (such as Delete, Clear, Reformat, Close, or Quit) should be located near the bottom of their menus to prevent users from selecting them accidentally while dragging to less dangerous menu items.

Chapter 5:

The Storage Item Editor

Contents:

Table of Contents Index

Understanding the Storage Item Editor

You should read this chapter if you want to know:

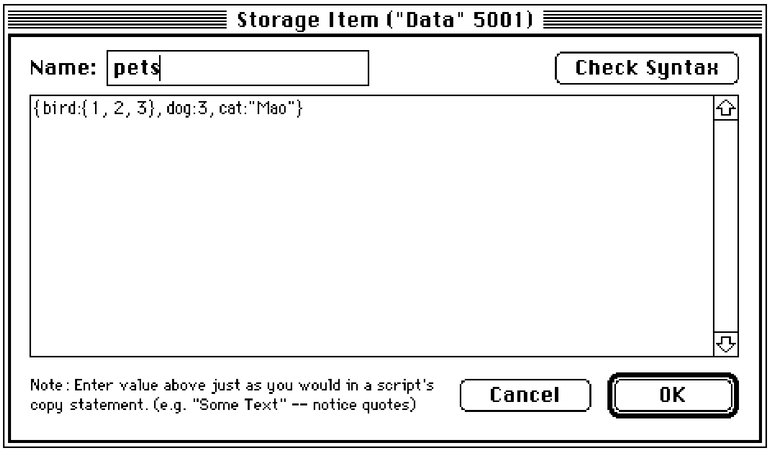
- what storage items are
- how to create storage items
- how to use storage items in your projects

A storage item is a piece of data kept in permanent storage within a project. In addition to values, storage items can be scripts or script objects—named scripts. One storage item can contain several named scripts.

The Storage Editor helps you create, name, define, store, and edit global information for a project. A project may contain any number of storage items.

If you prefer, you can use a script to **`make`**, **`delete`**, or assign values to storage items. Any project storage item—defined by its name and value—can be accessed, by **`name`** or **`id`**, from any script in the project.

The Storage Item Editor



Name: pets

```
{bird:{1, 2, 3}, dog:3, cat:"Mao"}
```

Note : Enter value above just as you would in a script's copy statement. (e.g. "Some Text" -- notice quotes)

The Storage Item Editor is used to create, name, assign a value, and edit a project’s storage items. You can open an editor for an existing storage item by clicking the Storage View radio button in the Project Window, then double-clicking one of the names displayed in the list of project storage items.

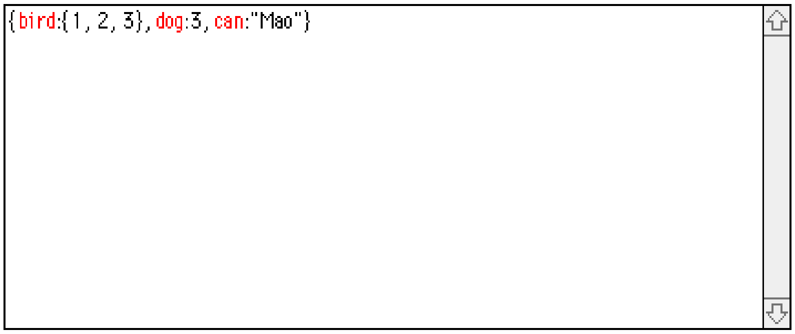
Storage Item Name textbox

By placing the insertion point inside the Storage Item Name textbox, you can name a new storage item or edit the name of an existing storage item. The **`name`** you type will be displayed in the Project Window listbox when Storage is the selected view. If a storage item is unnamed when the editor closes, it is identified in the Project Window listbox by its **`id`**.

Check Syntax button

You can click the Check Syntax button to check the syntax of the value you have defined. If the syntax is incorrect, an error dialog displays. If you do not click the syntax button, FaceSpan checks the syntax when you close the editor, and reports any compilation errors.

Storage Item Value textbox



```
{bird:{1, 2, 3}, dog:3, can:"Mao"}
```

By placing the insertion point inside the Storage Item Value textbox, you can assign a value to a storage item. You may assign a value of any type. The value must appear just as it would in a script's copy statement—that is, string values must be in quotes, and so on.

Instructions

Creating a new storage item

1 Click the Storage View radio button in the Project Window.

2 Click the New button.

A Storage Item Editor displays.

Opening the editor of an existing storage item

1 Click the Storage radio button in the Project Window.

2 Double-click one of the names in the displayed list of storage items.

or

Click once to select an item, then click Open.

The Storage Item Editor displays. You may now rename the item, or replace its existing value.

Cutting a Storage item

1 Select the name of the storage item to be cut.

2 Choose Cut Storage Item from the Edit menu.

Copying a storage item

1 Select the name of the storage item to be copied.

2 Choose Copy Storage Item from the Edit menu.

Pasting a storage item

1 Click the Project Window of the destination project to make it active.

2 Choose Paste Storage Item from the Edit menu.

A Storage Item Name dialog opens.

3 Give the pasted Storage Item a unique name.

4 Click OK.

Duplicating a storage item

1. Select the name of the storage item to be duplicated.
2. Choose Duplicate Storage Item from the Edit menu.

   A Storage Item Name dialog opens.
3. Give the duplicated Storage Item a unique name.
4. Click OK.

Deleting an existing storage item

1. Click to select the name of the storage item that you want to delete.
2. Choose Clear Storage Item from the Edit menu.

   or

   Click the Delete button.

The selected item is now deleted from the project.

Chapter 6:

The Script Editor

Contents:

Table of Contents Index

Understanding the Script Editor

You should read this chapter if you want to know:

➤ how to create and edit scripts using FaceSpan's Script Editor

➤ how to use the Script Editor's on-line reference tools

➤ how to record scripts by example

➤ how to check script syntax

FaceSpan's Script Editor helps you create scripts in an OSA scripting language by providing a standard text-entry environment for creating and editing scripts, as well as on-line reference popups that facilitate scripting, a recorder to record scripts by example, and the ability to test scripts before they are run.

Using the Script Editor, you can…

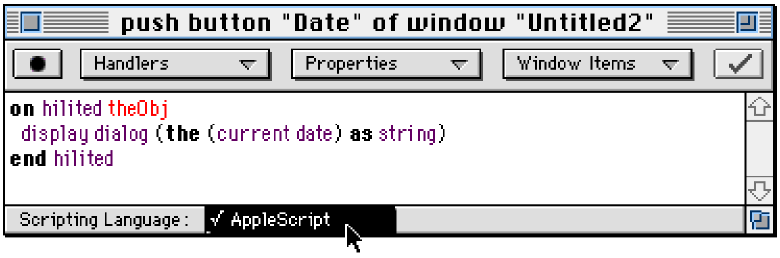
➤ Generate editable scripts by activating the script recorder, then interacting with recordable applications

➤ Facilitate scripting by choosing from popup that automatically insert correctly-phrased statements and references into your scripts

➤ Edit scripts using Script menu commands such as Find, Replace, and Enter Selection to make editing easy

➤ Check script syntax for errors that would prevent scripts from compiling or running

The Script Editor

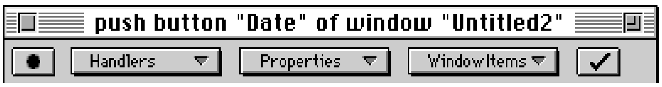


```
on hilited theObj
  display dialog (the (current date) as string)
end hilited
```

Scripting Language : ✓ AppleScript

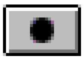
The Script Editor is divided into three sections. The top section contains a title bar—which identifies the object whose script you are editing, controls for script recording and error checking, and three on-line reference popups. The middle part contains a script textbox—which contains the script being edited, while the lowest part contains the Scripting Language popup.

Script textbox controls



Script textbox controls include buttons you can use to record scripts as well as check for syntax errors in scripts, and popups you can use to insert handlers and references into scripts.

Record Script button

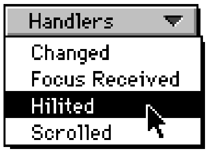


When you click the Record Script button, located near the top left corner of the Script Editor, FaceSpan’s script recorder activates. While recording is in progress, any interactions you make with a recordable application are generated into an editable script. When you turn the recorder off—by clicking the Record Script button a second time—the script you have recorded is automatically compiled and placed in the Script textbox.

If you prefer to control the script recorder from the menu bar, you can choose the Recording command from the Script menu.

Hint Recorded scripts may require additional editing before they can be used effectively in FaceSpan applications. You should check for instructions and references that may be unnecessary, too general, or too specific.

Handlers popup

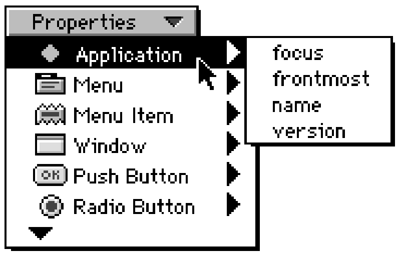


The Handlers popup is a pop-up menu. Its menu items are the messages recognized by the object being edited.

You can script a handler by choosing a message name from the popup. When you do, a correctly-phrased starting and ending statement for the handler is automatically pasted into the Script textbox at the insertion point. Where applicable, FaceSpan includes placeholder variables in the handler—to contain a reference to the window item receiving the message and other parameters.

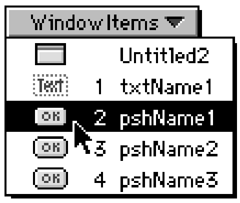
Any custom handlers you have defined for the object being edited are also automatically listed in the Handlers popup. If you select the name of a custom handler listed in the popup, FaceSpan locates and highlights it.

Properties popup



The Properties popup is a hierarchical menu. Its menu items are the classes of FaceSpan objects; a submenu for each object lists all of its properties. You can add a property reference within a handler by placing the insertion point inside the handler, and choosing an object class, then a property from the popup. A correctly phrased property reference is then automatically pasted into the Script textbox at the insertion point.

Window Items popup



The Window Items popup is a pop-up menu. Its menu items include the name of the active window template and it's window items. You can add an object reference within a handler by placing the insertion point inside the handler, then choosing an object's name from the popup. A correctly-phrased object reference is automatically pasted into the Script textbox at the insertion point.

Hint While using this popup, if you choose the name of the window or window item whose script you are editing, a reference to **`theObj`** is inserted. In FaceSpan, **`theObj`** is a variable containing a reference to the object receiving a message. Using **`theObj`** in a script (rather than, say, the actual name of a particular object) can be an advantage; you can copy a script that uses **`theObj`** and paste it into the Script Editor of a different object without having to change literal references.

Check Syntax button

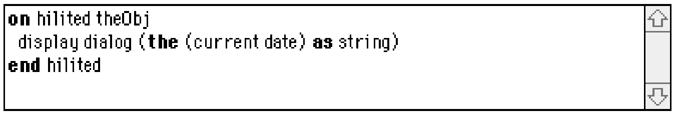
The Check Syntax button is identified by a check mark icon and is located near the top right corner of the Script Editor. You can click the Check Syntax button to compile an uncompiled script in the Script textbox. If reference and syntax errors are found, the script error is highlighted, and an error dialog box displays an explanation.

Hint FaceSpan also attempts to compile a script when you press the Enter key while a Script Editor is open.

Note

➤ For more information about run-time script errors and testing, see Chapter 8: "The Testing Environment."

Script textbox



```
on hilited theObj
 display dialog (the (current date) as string)
end hilited
```

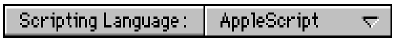
Table of Contents Index

The Script textbox contains the script of the object being edited. It supports standard Macintosh text entry, Edit menu commands (Cut, Copy, Paste, Clear, and Undo), and Script menu commands (Check Syntax, Recording, Enter Selection, Find, Find Again, Find in Next, Replace, Replace Again, AppleScript Formatting).

Note

➤ Appendix A: “FaceSpan Menu Reference” gives an explanation of these Script menu commands, and all other FaceSpan menu commands.

Scripting Language popup



The Scripting Language popup is located near the bottom of the Script Editor. When you click the Scripting Language popup FaceSpan displays the names of scripting languages (Open Scripting Architecture “OSA”systems) installed on your Macintosh computer. If there is more than one language listed, the scripting language currently selected has a check mark beside its name. You can tell FaceSpan to access a different scripting language by using this popup. FaceSpan lets you use any one or all the OSA languages you have available. However, all the scripts for a particular object must be written in the same scripting language.

“Drag and Drop” support in the Script Editor

If you have Drag & Drop installed on your Macintosh computer, in addition to recording scripts and typing scripts, you can “drag and drop” text from the Message Windoid or from the Dictionary Windoid into the Script textbox.

Instructions

Using Edit menu commands

While a Script Editor is active, Edit menu commands—including Cut, Copy, Paste, Clear, and Undo—pertain to the active editor.

Opening a script editor for the project script

While the Project Window is active, click the Project Script button in the Project Window.

A Script Editor containing the project script opens.

Opening a script editor for a window template or window item

1 Click the Windows View radio button in the Project Window.

2 Double-click the name of a window template.

The selected window template opens.

3 Click the window template itself or a window item to select it.

4 Click the Object Script button in the Property Bar.

or

Use the keyboard Equivalent ⌘-E.

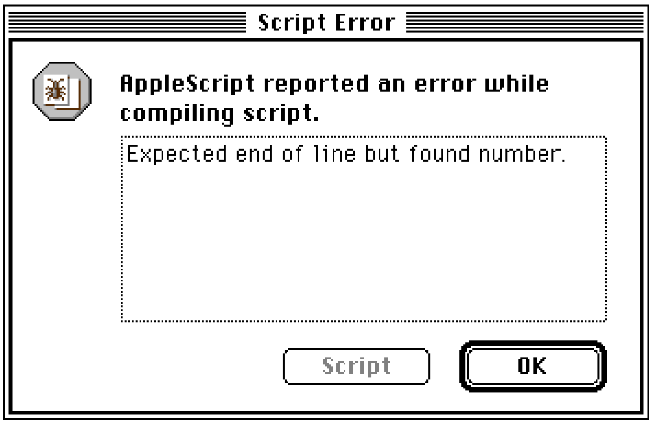
A script editor containing the script of the selected window template, or window item opens.

Checking for errors

1 Click the Check Syntax button.

FaceSpan attempts to compile the script.

If any reference and syntax errors are found a Script Error dialog box displays an explanation.



AppleScript reported an error while compiling script.

Expected end of line but found number.

2 Click OK to dismiss the dialog.

Scripting a message handler

1 Place the insertion point where a handler is to be entered.

2 Choose a handler from the Handlers popup.

FaceSpan pastes an “on” and “end” statement for the chosen handler at the insertion point. The insertion point repositions itself between the “on” and “end” statement. Where applicable, FaceSpan includes placeholder variables to contain a reference to the window item receiving the message.

3 Type the instructions to be performed each time the message is received.

FaceSpan attempts to compile the script if you close the Script Editor or click the Check Syntax button.

Adding a property reference to a handler

1 Place the insertion point where the reference is to be inserted.

2 Choose a property name from the Properties popup.

FaceSpan pastes the appropriate property reference into the Script textbox at the insertion point.

Adding an object reference to a handler

1 Place the insertion point where the reference is to be inserted.

2 Choose an object’s name from the Window Items popup.

FaceSpan pastes the appropriate object reference into the Script textbox at the insertion point.

Recording a script

1 Place the insertion mark at the location where the script recording is to be inserted.

Note

➤ A recorded script must be inside a handler, or it cannot be run later.

2 Click the Record Script button to start recording.

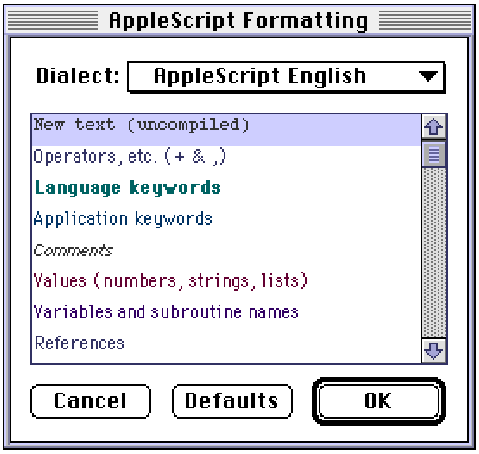
3 Perform procedures in a recordable application.

4 Click the Record Script button again to stop recording, and to compile the scripts in the open editor.

FaceSpan attempts to compile the recorded script and places it in the Script textbox at the insertion point.

AppleScript Formatting

You can use the AppleScript Formatting command in the Script menu to set global preferences for formatting the text of all AppleScript scripts. Changing the format of a script’s text changes the appearance of the text, but does not affect the meaning of the script.



New text (uncompiled)
Operators, etc. (+ & ,)
Language keywords
Application keywords
Comments
Values (numbers, strings, lists)
Variables and subroutine names
References

The AppleScript Formatting dialog displays when you choose AppleScript Formatting from the Script menu.

You can use this dialog to:

➤ Select the dialect in which scripts will display, by using the Dialect popup.

➤ Restore AppleScript's default settings for formatting script text, by clicking the dialog's Defaults button.

➤ Create global formatting preferences for the text of all AppleScript scripts.

To customize the text formatting of all AppleScript scripts:

1 Select the script element you want to format, by using the listbox in the AppleScript Formatting dialog.

2 Choose the format for that script element, by using the Font and/or Style menu.

3 Click OK.

Preferences for formatting script text are applied when the dialog is closed. All AppleScript scripts now follow the preferences designated in the dialog.

Chapter 7:

Other Scripting Tools

Contents:

Table of Contents Index

Other Scripting Tools

You should read this chapter if you want to know:

➤ how to display the dictionary of any scriptable application

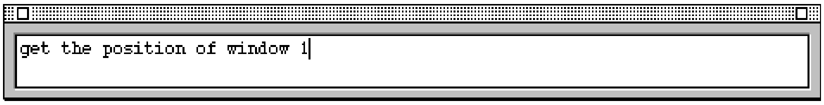
➤ how to use the Message Windoid to send messages to objects or directly to scripts

➤ how to use the Message Windoid to log AppleEvents as they are generated

Message Windoid

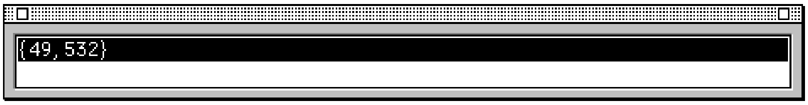
Interactive Debugging

The Message Windoid allows you to get and set properties, as well as to send test messages to window items. Like the Script Editor, the Message Windoid is a standard text-entry environment.



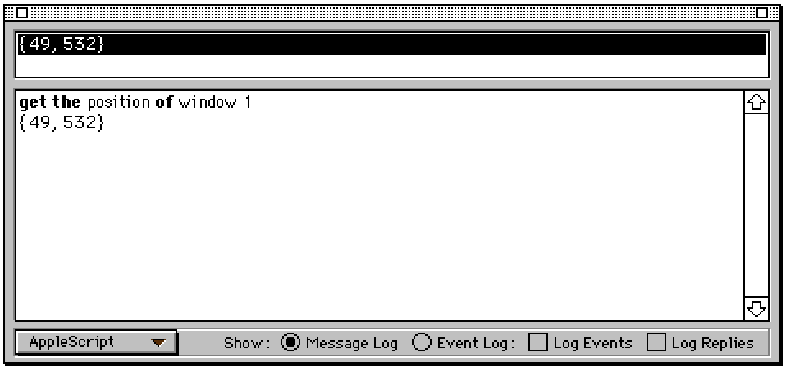
```
get the position of window 1
```

You can use the Message Windoid in either a collapsed or expanded state. If you use the Message Windoid in its collapsed state, as shown here, the instructions you type and the results they return are all displayed in a single textbox. While the cursor is in the textbox, instructions are executed when you press the Return key.



```
{49, 532}
```

If you expand the windoid (by clicking its zoom box) a second textbox—or log—and windoid controls display. The log portion of the expanded windoid is scrollable and can serve as either a Message Log or an Event Log.

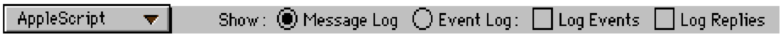


```
{49, 532}

get the position of window 1
{49, 532}
```

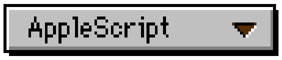
Table of Contents Index

Message Windoid controls



Message Windoid controls include the Scripting Language popup, Message Log and Event Log radio buttons, and the Log Events and Log Replies checkboxes. You can use these controls to designate your choice of scripting language, the current view of the windoid, and the functionality of the Event log when it is active.

Scripting Language popup



Click and hold the Scripting Language popup to display the names of scripting languages (Open Scripting Architecture “OSA” systems) installed on your Macintosh computer.

Message Log radio button

Click the Message Log radio button to display the Message Log View of the Message Windoid.

Event Log radio button

Click the Event Log radio button to display the Event Log View of the Message Windoid.

Log Events checkbox

Click the Log Events checkbox to log AppleEvents while the Events Log is active.

Note

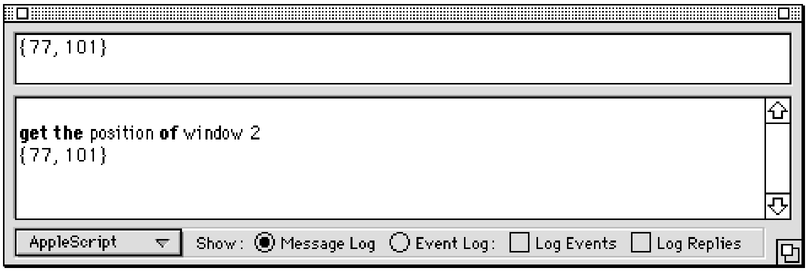
➤Turning off Log Events also turns off Log Replies, if it was on.

Log Replies checkbox

Click the Log Replies checkbox to log replies to AppleEvents while the Events Log is active.

Message Log View

When the lower textbox in the expanded Message Windoid is in Message Log View, a record of all the instructions sent and the results they returned, is displayed. The Message Log is an extremely handy tool when creating projects; it allows you to write scripts such as loops and other multi-statement sequences for immediate execution. Just write the script, select it, then press the Enter key to execute the script.



```
{77, 101}
```

```
get the position of window 2
{77, 101}
```

AppleScript
Show: (● Message Log) ○ Event Log: ☐ Log Events ☐ Log Replies

The Message Log can be used while a window template is in Edit Mode or Play Mode, as well as when the project script is running. You can use the Message Windoid to query things like global variables and access properties of the current window during Run Mode.

Notes

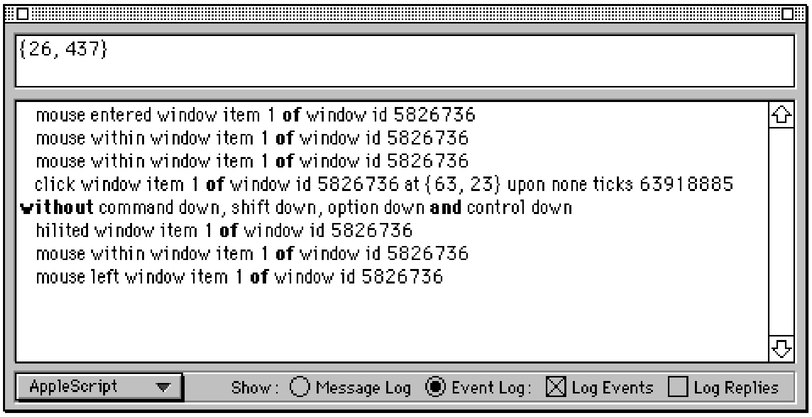
➤ When using the Message Windoid, the window or window template being referenced must be frontmost (immediately behind the Message Windoid). If an open Script Editor is frontmost, an error will result.

Table of Contents Index

➤ You can use the Message Windoid during runtime—to check and change variables, use the Event Log, and so on—by opening it first, then running the project script. Remember that while a project script is running, it controls the menu bar. You must open the Message Windoid first, while FaceSpan's menu commands are available.

Event Log View

When the lower textbox is in Event Log View and the active window template is in Play Mode, or the project script is running, FaceSpan tracks AppleEvents for that window and can display a log of events, replies, or both.



```
{26, 437}
```

```
  mouse entered window item 1 of window id 5826736
  mouse within window item 1 of window id 5826736
  mouse within window item 1 of window id 5826736
  click window item 1 of window id 5826736 at {63, 23} upon none ticks 63918885
without command down, shift down, option down and control down
  hilited window item 1 of window id 5826736
  mouse within window item 1 of window id 5826736
  mouse left window item 1 of window id 5826736
```

You can also leave Log Events and/or Log Replies "on" while you change to Message Log view; FaceSpan will continue to track and log, according to your selection.

Hint Remember that to use the Message Windoid during runtime—to check and change variables, use the Event Log, and so on—you must open it before clicking the Run button.

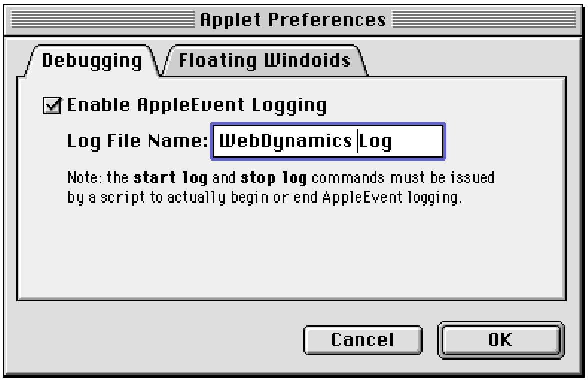
Logging Events to a File

In addition to testing a window in Play Mode with Event Log turned on, you can also use event logging to assist you in debugging projects saved as applications.

➤ To use event logging:

1 Save the project as an application.

2 With the Project window active, choose **Applet Settings** from the Edit menu to open the Applet Preferences dialog box.



☑ Enable AppleEvent Logging

Log File Name: WebDynamics Log

Note: the **start log** and **stop log** commands must be issued by a script to actually begin or end AppleEvent logging.

3 Click the checkbox to enable logging.

4 In the text box, enter a name for the log file.

5 Click **OK**.

When you launch the application, it will create a log file in the application's folder. The log file is named what you entered in step 4. Debugging information is saved to this log file.

Examples

```
on run
   start log
      open window "my window"
   -- more scripting commands here
   stop log
end run
```

Drag and Drop support in the Message Windoid

If you have Drag and Drop installed on your Macintosh computer, you can:

➤ Drag and Drop text from the Message Windoid into the Script Editor.

➤ Drag text from the log area (displayed in the windoid's expanded view) and drop it into the upper textbox of the windoid for editing and/or execution.

➤ Drag text from the Dictionary Windoid and drop it into the Message Windoid.

Instructions for Using the Message Windoid

Displaying the Message Windoid

Choose the Message command from the Window menu or type ⌘-M.

The Message Windoid displays.

Sending a message from the uppermost textbox of the Message Windoid

1 Type a statement in the uppermost textbox of the Message Windoid.

2 Press the Return Key or the Enter key.

The message is sent.

Sending a message from the lower textbox of the Message Windoid

1 If the windoid is not in Message Log View, click the Message Log radio button.

You can:
➤ Type a statement, then select it.
➤ Edit existing text, then select it.
➤ Select existing text.

2 Press the Enter key.

The selected message is sent.

Getting the value of a property of a window item

1 The Message Windoid should be in Message Log View.

2 Use a **`get`** statement.

or

Type a reference to the property and window item.

3 Press the Return key.

The value of the property of the window item is displayed in the Message Windoid.

For example, to **`get`** the value of the text **`font`** property of the lowest **`index`** button in the window template, type:

```
get font of push button 1
```

Now, press the Return key.

The value of the requested property of the window item is displayed in the Message Windoid.

For example, to use a reference to the same property and window item, type:

```
font of push button 1
```

Now, press the Return key.

The value of the requested property of the window item is displayed in the Message Windoid.

Note

➤ To reference an item not in the frontmost window or window template, specify the container of the item. For example, you can reference the container of a window item by its window's name or number: window "name" or, "window 1." If there is more than one open window having the same name, the window closest to the front is referenced.

Setting the value of a property of a window item

1 The Message Windoid should be in Message Log view.

2 Use a **`set`** statement

3 Press the Return key.

The value of the property of the window item is set.

For example, to **`set`** the value of the text **`font`** property of the lowest **`index`** button in the window template, type:

```
set font of push button 1 to "Geneva"
```

Now, press the Return key.

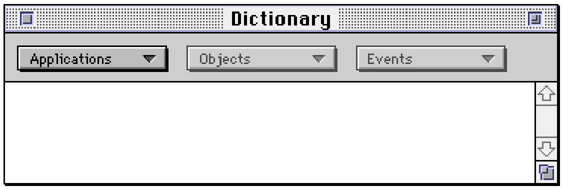
The value of the requested property (**`font`**) of the window item (push button 1) is set (to "Geneva" in this case).

Note

➤ To reference an item not in the frontmost window or window template, specify the container of the item. For example, you can reference the container of a window item by its window’s name or number: window “name” or, “window 1”. If there is more than one open window having the same name, the window closest to the front is referenced.

Understanding the Dictionary Windoid

A scriptable application's dictionary (aete resource) contains definitions for words—objects, commands, or other words—which are understood by that application. You can use FaceSpan's Dictionary Windoid to view FaceSpan's own dictionary, dictionaries of other scriptable applications, or of scripting additions.

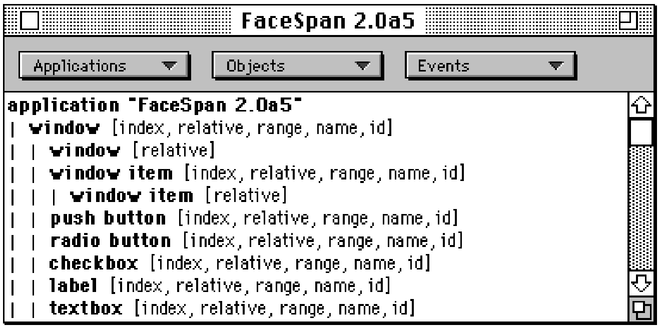


When you choose the Dictionary command from the Window menu, the Dictionary Windoid opens. Using the Dictionary Windoid you can display the dictionary of a selected scriptable application, and can change the view of an active dictionary.

Applications popup

You display a scriptable application's dictionary using the Applications popup. When you first open the Dictionary Windoid, the Applications popup has an Open Other menu item, a FaceSpan menu item, and—if you have System 7.5 installed on your Macintosh computer—a Finder Scripting Extension menu item. You can choose the Open Other menu item to display a standard Open dialog, from which you can open the dictionary of any other scriptable application, or the FaceSpan menu item to open FaceSpan's dictionary, or the Finder Scripting Extension menu item to open the Scriptable Finder's dictionary.

As a convenience, FaceSpan automatically adds the name of any previously opened dictionary to the Applications popup, so that in the future you can simply choose the dictionary's name to open it. The entry is persistent and displays each time you use FaceSpan, unless you choose to delete it by holding down the Command key while selecting the application name you want to delete.



```
application "FaceSpan 2.0a5"
| window [index, relative, range, name, id]
| | window [relative]
| | window item [index, relative, range, name, id]
| | | window item [relative]
| | push button [index, relative, range, name, id]
| | radio button [index, relative, range, name, id]
| | checkbox [index, relative, range, name, id]
| | label [index, relative, range, name, id]
| | textbox [index, relative, range, name, id]
```

When you first open a scriptable application's dictionary, a hierarchical list of all its objects is presented. You can display a dictionary entry for a specific object by clicking its name in the list. While the dictionary is open, you can return to its hierarchical list by selecting the current application's name from the Applications popup.

Notes

➤ If a previously opened application is not found—usually because it is on an unavailable volume, or possibly because you have removed it from your hard disk—a notification dialog displays. You can cancel the dialog, or you can click the Remove button to delete the application's name from the Applications popup.

➤ The Scriptable Finder's dictionary is in the Finder Scripting Extension; located in the Extensions folder of the 7.5 System Folder.

Hint If you prefer to open an application dictionary without displaying its hierarchical list of objects, you can hold the Option key while choosing an application name from the popup; the hierarchical listing is not shown, but the Objects and Events popups are still available.

Objects and Events popups

You can use the Objects and Events popups to view specific object or event definitions in the active dictionary.

Drag and Drop support in the Dictionary Windoid

If you have Drag and Drop installed on your Macintosh computer, you can select text from the Dictionary Windoid, then drag and drop the text into an object's Script Editor, or into the Message Windoid.

Instructions for Using the Dictionary Windoid

Opening FaceSpan's dictionary

Choose FaceSpan from the Applications popup.

FaceSpan's application dictionary displays in the Dictionary Windoid.

Note

➤ Some FaceSpan objects have properties that can have more than one type of value; the dictionary does not display all possible types. You will find comprehensive discussions of objects, their properties and value types, in Part III: "FaceSpan Object and Language Reference."

Opening a different scriptable application's dictionary

1 Choose the Open Other menu item from the Applications popup.

A standard directory dialog displays.

2 Locate and select the name of the application whose dictionary you wish to view.

3 Click the Open button.

If the selected application is a scriptable application, its dictionary displays in the Dictionary Windoid.

Adding an item to the Applications popup

Once a dictionary has been opened, the name of its application is automatically listed in the Applications popup, and remains until removed.

Removing an item from the Applications popup

1 Select the item to be deleted while holding down the Command key.

A dialog displays asking you to confirm the deletion.

2 Click OK.

The selected item is deleted.

Chapter 8:

The Testing Environment

Contents:

Table of Contents

Index

The Testing Environment

You should read this chapter if you want to know:

➤ how to test the way interface objects will respond to user input

➤ how to test-run your project's scripts

➤ how compilation and run-time errors are handled

Testing is an integral part of developing any application. With FaceSpan, you can test-run the project script of your application, as well as test the way objects will respond to user input, while the project is still under development.

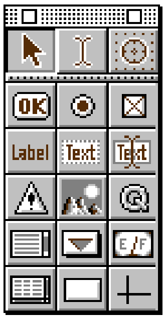
FaceSpan's built-in testing environment includes Play Mode and Run Mode. Play Mode is available while the Window Editor is active, while Run Mode is initiated from the Project Window. In addition to using Play and Run Mode, you can check scripts for compilation errors at any time while scripting, and you can use the Message Windoid as both a scripting tool and a testing tool throughout development.

The Message Windoid and Testing

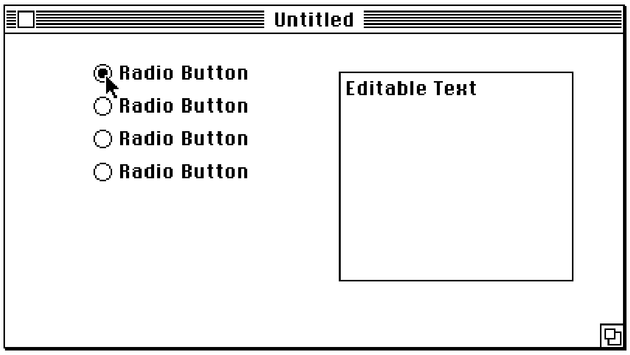
The Message Windoid can be used for testing in Edit Mode, Play Mode, or Run Mode. You can find a detailed discussion of how to use it in Chapter 6: "The Script Editor."

Play Mode

As you create new interface objects, you can test their behaviors by changing the Window Editor from Edit Mode to Play Mode. You change an active window template to Play Mode when you choose the Arrow tool from the Tool Palette. While in Play Mode, you can test the way a window template and its window items will respond to user input during runtime.



When Play Mode is initiated, the window template's object grid disappears. The cursor becomes an I-beam when placed over an editable text field—to enable text entry—but remains an arrow when interacting with other interface objects—so that buttons can be clicked, movies played, and so on. The Property Bar and the Tool Palette remain on-screen unless you use a keyboard command (Option key-Tab) to hide them.



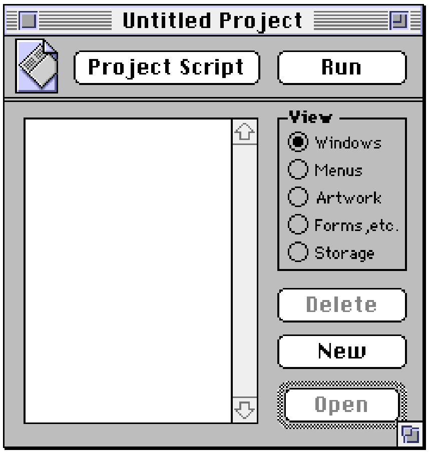
You return a window template and it’s objects to Edit Mode by clicking the Object Mover tool, or any of the Object Maker tools in the Tool Palette.

Note

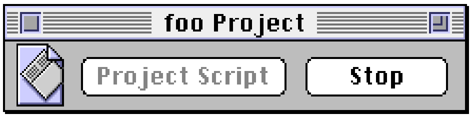
➤ During Play Mode the project script is not run, so **`open window`** commands are not executed and some application and window script properties may not be initialized.

Run Mode

Run Mode allows you to test-run your project's script and any run-time menus you have created. The project script—the main script for the application you are building—can control a variety of functions including menus. If you have created menu templates and associated them with a particular window, or the project itself, they are added to the application's menu bar at runtime.



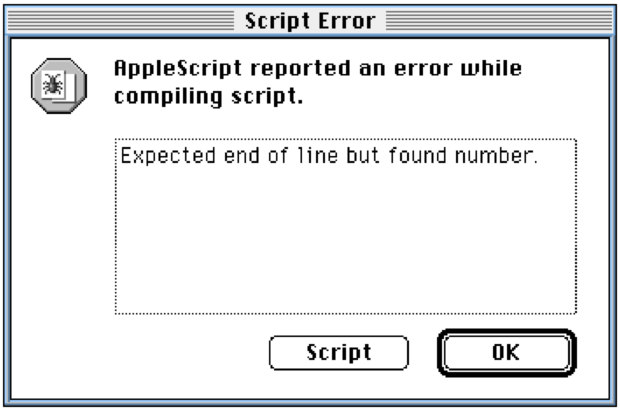
You initiate Run Mode by clicking the Run button in the Project Window. When you click the Run button, FaceSpan attempts to compile any open scripts, hides any open window templates, and runs the project script.



While the project script is running, the Project Window temporarily shrinks. The Run button becomes a Stop button, which you can click to halt the running project script.

Project Script Errors during Runtime

If an error is generated during runtime, and the Project Script contains the script in error, an Error Message dialog box displays.



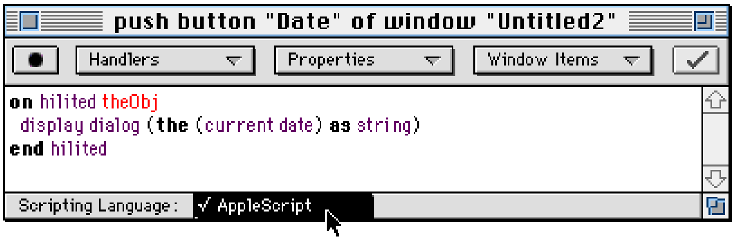
AppleScript reported an error while compiling script.

```
Expected end of line but found number.
```

If you click the Script button in the dialog, FaceSpan automatically opens the Script Editor for the Project Script, locates script in error, and highlights it so you can then make the correction.

Window and Window Item Script Errors during Runtime

If a window or window item script generates an error during runtime, an Error Message dialog box displays. When you click the Script button in the dialog, FaceSpan displays a non-editable version of the Script Editor containing the error.



```
on hilited theObj
 display dialog (the (current date) as string)
end hilited
```

Scripting Language: ✓ AppleScript

FaceSpan displays the error message at the top of the non-editable script, and highlights the script in error—so that you can see where the error occurred.

Because the error is in the script of the window template (or an item it contains), in order to correct the script you need to open the window template—which is available in Edit Mode. Once you halt the run, you can open the appropriate window template or window item's Script Editor to correct the error.

Part II: The Structure of Applications

Chapter 9:

The Structure of Applications

Contents:

Table of Contents Index

Understanding the Structure of Applications

You should read this chapter if you want to learn about:

- ➤ the components and structure of applications you can develop,
- ➤ the components and structure of FaceSpan itself, and
- ➤ ways to approach application development.

Application Components

A FaceSpan application consists of objects and their scripts. *Objects* are programming or scripting entities that contain information and operations upon that information. We call the information “properties” and the operations “handlers.”

Interface Objects

Most of the objects in a FaceSpan application are interface objects, such as windows, menus, and window items. All interface objects have, in addition to properties and handlers, visible manifestations—that is, images on the screen. Many of those images depict things with which we can interact, such as buttons, menus, and scroll bars. The appearances and behaviors of interface objects—often called their “look and feel”—are controlled by their properties and handlers.

Properties

Each property of an object has a name and a value; for example, the `title` property of a label might have as its value the text, “Type your name.” It is the `title` property that defines the text of the label as it appears in a window.

Every object already has several properties defined by FaceSpan, and each property has a default value. For example, a push button already has a **`title`** property; its default value is “Button.” These pre-defined properties thus determine the object’s default appearance.

All the pre-defined properties of all the objects are listed in Part III, “FaceSpan Object and Language Reference.”

Handlers

Interface objects can be sent *messages*. These messages are sent in response to interactions between the object’s image and the application user. For example, when you select an item from a listbox on the screen, a message is sent to that listbox object to tell it that the selection event occurred.

Each kind of interaction causes a message with a unique name to be sent to the object. For example, the message sent when a listbox item is selected is called the `selection made` message. There is a pre-defined set of messages for each object in a FaceSpan application.

Table of Contents Index

An object has pre-defined ways to respond to interactions with the user. For example, when a listbox item is selected, it is hilited. These pre-defined actions determine the object's default behaviors.

In addition, you can write handlers (subroutines) to respond to the messages that are sent to the object. Your handlers, which are named after the messages, define what you want to happen when the object receives the messages.

Objects can respond not only to messages caused by interactions, but also to messages sent by commands in your scripts. Once again, there are several pre-defined command messages and behaviors, and corresponding handlers you can write for each object.

The pre-defined command and event messages for all objects are listed in Part III of this guide.

By the way, the technical term for these messages is "Apple Events."

Scripts

Every object can have an associated script. The script is displayed when you open a Script Editor for the object. In the object's script you can define new properties, new handlers and subroutines.

You can also give new values to the default properties and augment or override the object's default handlers with new actions. In fact, most of the default handlers for pre-defined command and event messages do very little. The life of a FaceSpan application is in its scripts—the scripts you write.

Table of Contents Index

Properties and handlers are expressed in AppleScript or in another “OSA” scripting language. Here is an example script, written in AppleScript, that declares a new property, defines a new message (by defining a handler for it), and defines a handler for a pre-defined message:

```
property num: 7 --a new property

--Here is a new handler, which defines a new message:
on boogaloo(n)
    repeat with i from 1 to n
        beep 1
    end repeat
end boogaloo

--Here we give a handler for the pre-defined hilited message.
--The image on the screen highlights before this is called:
on hilited theObj
--This gives a new value to a pre-defined property of a label:
    set the title of label “sayWhat” to “Get down!”

    --This command sends a pre-defined message to the label:
    tell label “sayWhat” to adjust size

    --This command sends a message to the new handler defined above:
    boogaloo(num)
end hilited
```

Application Object

The application, too, is an object, but it is an organizational entity, not an interface object. Its purpose is to serve as a vehicle or container for all the interface objects. It, too, has a script, called the “project script,” and a set of properties and handlers.

Application Structure

An application that you create with FaceSpan has a logical structure and a physical structure.

The logical structure is your view, as a developer, of how the application components are organized and how they interact.

The physical structure is the real arrangement of program components that implement the logical structure.

Logical Structure

Logically, the application object and all the interface objects, along with their scripts, *are* the application. They are organized hierarchically.

Object hierarchy

The objects in a FaceSpan application are organized into a hierarchy based upon the physical appearance of the application while it is executing. The application contains the windows and the windows contain window items—such as labels and buttons. There can be no window without an application, and there can be no window items without a window.

So the window “contains” the window items, while the application “contains” the windows. The application contains the menus, too.

Message hierarchy

The scripts of all the objects follow the same organization as the objects themselves. The application object’s script contains the window scripts, and each window’s script contains its window items’ scripts.

A message caused by interaction with a physical object on the screen is sent to the script of that object. If the script does not handle it, the message goes up the hierarchy to the script of the containing object, and so forth.

Physical Structure

There is considerably more that goes into an application than what you need to consider while creating it.

The FaceSpan Extension

The FaceSpan Extension is a collection of routines for drawing interface elements and handling interactions with those interface elements.

A project or application developed with FaceSpan does not draw its own interface. Instead, routines in the extension draw the interface. Descriptions of the interface elements are stored within the application or project file. The descriptions are lists of properties.

Similarly, all the default behaviors of the interface objects are defined in the FaceSpan Extension. Your scripts are compiled and stored in the application or project file.

Kinds of applications

A FaceSpan project can be saved as an application in one of several forms.

A Complete Application is one that contains the FaceSpan extension as well as the descriptions of interface objects and their scripts. This makes the application self-contained; the extension need not be present in the Extensions folder.

A Miniature Application is one that is composed of only the descriptions of interface objects and their scripts. To execute a Miniature Application, the FaceSpan extension must be present in the system’s Extensions folder.

You would save a projects as a Miniature Application when small size is important, or as a Complete Application when convenience is most important.

By the way, an application that can be “drop launched” is a Miniature or Complete Application that has a handler for the `open` message in its project script. FaceSpan detects the `open` handler and takes care of the desktop icon and other issues.

How FaceSpan works

FaceSpan itself is an editing and testing environment for creating applications. FaceSpan is itself written in FaceSpan. That is, the entire editing and testing interface that you use when developing projects is created by calls to the FaceSpan extension, a copy of which is imbedded (for convenience) in FaceSpan itself.

Table of Contents Index

Chapter 10:

Scripting Your Application

Contents:

Table of Contents   Index

Scripting Your Application

You should read this chapter if you want to know:

- ➤ how to intercept and respond to messages with handlers,
- ➤ how scripts control applications, windows, and menus,
- ➤ how scripts control other applications,
- ➤ how to use scripting additions in your applications, and
- ➤ how you might approach application development.

Messages and Handlers

When a user interacts with a running FaceSpan application, the interaction sends *messages* to the application, its windows, and window items. Although FaceSpan objects respond automatically to user inputs, message handlers can be added to the scripts of the objects to intercept messages and augment the application's responses.

For example, when someone clicks a push button on the screen, the button image highlights in response to the click, and the button's script is sent a **`hilited`** message. If the script contains a handler for the **`hilited`** message, the instructions contained in the handler will be performed.

The **`hilited`** message handler below sets the **`title`** property of its push button alternately to "Ouch" or "Yeow" when the button is clicked:

```
on hilited theObj
    if the title of theObj is “Ouch” then
        set the title of theObj to “Yeow”
    else
        set the title of theObj to “Ouch”
    end if
end hilited
```

The variable **`theObj`** contains what is called in AppleScript the "direct parameter" of the message. In the case of message handlers, the direct parameter is a reference to the object that is the target of the message. In our example, it is a reference to the button that the user clicked.

Actually, the direct parameter is not necessary. An object’s script is somewhat like a **`tell`** statement: the object is the default object of the statement. That means that all unqualified references to properties are assumed to refer to the object itself. Thus, the example handler above could be written this way:

```
on hilited theObj
   if the title is “Ouch” then
      set the title to “Yeow”
   else
      set the title to “Ouch”
   end if
end hilited
```

The parameter theObj is provided as a convenience so that you can make your property references explicit (like **`title of theObj`**, instead of just **`title`**) or pass the reference along to another handler. Although we will always show the variable name **`theObj`** as the direct parameter of handlers, you can substitute any non-reserved word or omit the variable as needed.

Partial References

When a handler refers to another element within the same container as the handler's default object, it is not necessary to include a reference to the container in the object reference. A handler in push button 1 of window “Same” does not require a reference to window “Same” when referring to another window item in the same window:

```
on hilited theObj
   set the enabled of textbox 2 to false
end hilited
```

However, when a handler refers to an element of a different container, the reference must specify the container:

```
on hilited theObj
   set the enabled of textbox 2 of window “Different” to false
end hilited
```

Finding An Object’s Container

There are times when you do not know the container of an object that is passed as a parameter.

To obtain a reference to the container of any FaceSpan object, use the form, *container class* **`of`** *object reference*. For example, in the **`hilited`** handler above, the term **`window of theObj`** would return a reference to the window containing the push button.

To get a reference to the application there is a special term: **`current application`**

Sending Messages to Other Objects

Handlers in object scripts can also send messages to handlers in other objects’ scripts. Any of FaceSpan's pre-defined messages can be sent using an explicit **`tell`**:

```
tell object reference to  message name
```

The message can be sent using an implicit **`tell`**, too:

```
message name    object reference
```

For example, here are two instructions that send **`hilited`** messages to other objects:

```
tell push button “pshScram” of window “Reactor” to hilited
hilited push button “pshScream”
```

If the script of the indicated push button contains a **`hilited`** handler, the handler will execute when it receives the **`hilited`** message from another handler, just as it will if a user clicks the push button with the mouse. In either case, the direct parameter of the **`hilited`** handler is a reference to the push button that was the target object of the message. Therefore, the effect is the same, whether it received the message from another object, or as the result of a mouse click. (But note that the button on the screen highlights only when actually clicked by the application user.)

Table of Contents Index

Containers Intercept Messages

When a message is sent to an object whose script doesn't handle it, the message doesn't stop there. Instead, the message is automatically continued to the target object's containers in search of a handler to intercept it.

For example, if the script of a push button receiving a **`hilited`** message does not have a **`hilited`** handler, the message is continued to the window containing the push button. If the script of the window contains a **`hilited`** handler, the message is handled; if the window does not contain a **`hilited`** handler, the message is continued to the application containing the window.

When a message is handled by an object's container, as when a window handles a button's **`hilited`** message, the direct parameter of the handler contains a reference to the target object, not to the container. Thus the container, the window in our example, can know the button to which the message was originally sent.

Unhandled Messages

If one of FaceSpan's pre-defined messages is not handled by the target object or any of the target object's containers, all the default behaviors occur, then it is ignored.

However, if a message that you defined is not handled by the target object or by any of the target object's containers, a script error occurs.

Continuing Messages

Even when an object handles a message, you can force the message to continue to the object's containers by including a **`continue`** statement in the handler. The following handler in a push button continues the **`hilited`** message to the window containing the button if the value of the button's **`title`** property is "Yeow":

```
on hilited theObj
    if the title of theObj is “Yeow” then
        continue hilited theObj
    else
        set the title of theObj to “Yeow”
    end if
end hilited
```

Since the **`continue`** instruction includes the object reference parameter for the message, you could even substitute a reference to any object you wish. This would tell the container that the message was originally intended for a different object.

Necessary Continuations

There are a few messages for which you would usually provide a **`continue`** statement. You will note that some message names are in the present tense, while others are in the past tense.

If a message name is in the past tense, then the default behavior of that message has already been completed. If the message name is in the present tense, your handler receives the message before the default behavior has occurred. You can block the default action if it has not yet occurred, but you must take responsibility for that decision.

The **`close`** message, for example, is sent to a window as it is about to close. Unless you continue the **`close`** message, the window will not close. That is why all the examples of **`close`** handlers in this chapter have **`continue`** statements.

Handling Intercepted Messages

The default continuation of messages to the containers of their default objects makes it possible for a container to control a group of its elements with a single handler.

For example, imagine that a window named “Colors” contains three push buttons, named “Red,” “Blue,” and “Yellow.” When the application user clicks one of the push buttons, the button’s name should be displayed as the title of label “whatColor” in the same window. Instead of redundant **`hilited`** handlers in the scripts of each of the three push buttons, the script of window “Colors” can contain the following handler, which sets the **`title`** property of the label to the **`name`** property of the push button that was the original target object of the **`hilited`** message:

```
on hilited theObj
   copy the name of theObj to theObjName
   set the title of label “whatColor” to theObjName
end hilited
```

Even though the window handles the message, not the push button that received it, the handler can refer to the element that originally received the message because the direct parameter of the **`hilited`** message, **`theObj`**, still contains a reference to the push button that was clicked.

The direct parameter reference is useful because the handler above will be activated every time an unhandled **`hilited`** message reaches window “Colors.” The script can find the container of **`theObj`** and can even find out what kind of object it is.

Custom Messages with Positional Parameters

The same effect can be achieved by sending and handling messages that you define. Let us say that the message is to be called **`showColorOf`**. Simple **`hilited`** handlers in the scripts of push buttons “Red,” “Blue,” and “Yellow” could then send **`showColorOf`** messages to label “whatColor” this way:

```
on hilited theObj
   tell label “whatColor” to showColorOf(the name of theObj)
end hilited
```

Then the script of label “whatColor” would need to handle the **`showColorOf`** messages:

```
on showColorOf(clickedButtonName)
   set title to clickedButtonName
end showColorOf
```

The messages you define can pass positional parameters, as needed, within parentheses. In this example, each button’s **`hilited`** handler passes the **`name`** property of **`theObj`**, a reference to itself, as the single parameter of the **`showColorOf`** message. The label’s **`showColorOf`** handler receives the parameter in the variable **`clickedButtonName`**, and sets the **`title`** property of the label to it.

Since the label is the default object of the **`showColorOf`** handler, **`title`** is evaluated as the **`title`** property of label “whatColor.”

Messages with Labeled Parameters

Your handlers can also accept messages with labeled parameters, which are sent using this statement format:

```
tell object reference to message name label name value
```

The script of each push button in our example might therefore contain this **`hilited`** handler:

```
on hilited theObj
    copy name of theObj to myName
    tell label “whatColor” to showColorOf whatName myName
end hilited
```

The script of label “whatColor” handles the **`showColorOf`** messages in this way:

```
on showColorOf whatName theName --note the labeled parameter
    set title to theName
end showColorOf
```

In fact, it is the format of this handler’s first line that *requires* that messages be sent to it using the named parameter.

Properties and the “my” Reference

Let us say that a property is declared in the script of an object, and that the object contains other objects. Then scripts in the containing object can refer directly to the property, while scripts in the contained objects must use the **`my`** prefix.

For example, if the property **`stat`** is declared in the script of a window, it can be referenced as **`stat`** in the window’s script, but the scripts of the window items contained in that window must refer to the property as **`my stat`**.

If **`stat`** is instead declared in the script of the application, it can be used in the application’s script as **`stat`**, in the window scripts as **`my stat`**, and in their window items’ scripts as **`my stat`**.

A script in an object outside the object in which a property is defined can still access the property, but it must fully qualify the reference. For example, if the property **`stat`** is defined in window “Statistics”, then a script in another window can refer to it as **`stat of window “Statistics”`**.

Declare a property at the top of the script of any container this way:

```
property *property name*: *propertyvalue*
```

Note that properties are defined with initial values.

Global Variables

FaceSpan scripts can also use global variables. A global variable must be declared within each script that uses it. A global variable is known throughout the project—that is, anywhere it is declared.

Here are some handlers (possibly in different object’s scripts) that use the same global variable:

```
on hilited theObj
    global gMine
    ...
    set gMine to whatever
end hilited

on MyHandler(something)
    global gMine
    ...
    copy gMine & something to whatelse
end MyHandler
```

Unlike properties, global variables are not initalized in their declarations; you have to assign values to them somewhere in your scripts.

FaceSpan’s **`storage item`** objects are global, can be used anywhere without declarations, and keep their values from run to run of your projects.

Controlling Windows

Opening Windows

When a FaceSpan application opens a window at run time, the window is constructed from the window template resource named in the **`open window`** statement. For example:

```
open window “User Info”
```

This statement looks for a window template resource named “User Info,” and creates a window based on the resource if it is found. The window resource created in the Window Editor is used as a model for a new window opened at run time.

Since each window opened in a FaceSpan application is identical to the window resource as it was saved earlier in the Window Editor, project scripts must often retrieve and set window and window item properties in order to save and restore user changes, as discussed next.

Setting Properties in an open window Statement

You’ll often need to adjust some properties of a window as it is opened. To do this, include a **`with properties`** parameter in the **`open window`** statement, in which you supply a record of the property names and values to be set:

```
open window “User Info” with properties {position:{100,120}}
```

When window “User Info” opens, it will be at position {100, 120} on the screen.

The record in the example above refers only to the window itself, but **`open window`** statements may also set the properties of window items. Each window item record of the **`with properties`** assignment must begin with a **`re`** property that tells the **`index`**, **`id`**, or **`name`** of the window item to be set:

{re: *window item index*, *property name*: *property value*, …}

For example, the following **`open window`** statement sets the **`visible`** and **`enabled`** properties of window item 12 as the window is opened:

```
open window “User Info” with properties {re:12, visible:true, enabled:false}
```

You may list as many records as needed in the **`with properties`** parameter. If you omit the **`re`** property from a record, FaceSpan assumes that the properties to be set belong to the window object. The following example first sets the **`height`** and **`position`** properties of the window, then the **`visible`** and **`enabled`** properties of window item 12:

```
open window “User Info” with properties ¬
{{height:100, position:{90,70}},{re:12, visible:true, enabled:false}}
```

Note the extra brackets. They are necessary when you refer to more than one object in the **`with properties`** parameter.

Retrieving Properties from a Modal Dialog

You can retrieve values set during user interaction with a modal dialog by including a **`returning properties`** parameter in the **`open window`** statement. You supply “model” records of property names (and place-holder values, which are ignored) to receive values when the window closes:

```
open window “User Info” returning properties ¬
   {height:0, position:0, closing item:0}
```

In the example above, a record containing the **`height`**, **`position`**, and **`closing item`** properties of the modal dialog “User Info” will be placed into **`the result`** when the user closes the window. A subsequent script instruction can extract the value of one of the properties from **`the result`** like this:

```
position of the result
```

The place-holder values were simply the digit 0. Actually, the value of **`position`** is a *list* of two numbers, and the value of **`closing item`** is returned as a *record* that describes the item that caused the window to close. For example, the value of **`closing item`** might be returned as:

```
{class:push button, bounds:{38, 110, 118, 131}, id:7003,
name:”pshName2”, title:”Button”, auto close:true}
```

Thus, if you wish to find the name of the button that closed the window, you would extract the name this way:

```
name of closing item of the result
```

Like the **`with properties`** parameter, the **`returning properties`** parameter can also retrieve properties of window items when the window is closed. As before, include the **`re`** property in each record that does not refer to the window object. The following example retrieves the **`height`** and **`position`** properties of the window object, and the **`visible`** and **`enabled`** properties of window item 12:

```
open window “User Info” returning properties ¬
   {{height:0, position:0} ,{re:12, visible:0, enabled:0}}
```

When properties of several items are retrieved in this manner, they are returned to **`the result`** as a list. Individual properties must then be extracted from **`the result`** using the indices of items in the list:

```
set thePos to position of item 1 of the result
set isVis to visible of item 2 of the result
```

An Alternative Method for Retrieving Properties

The **`returning`** parameter, used for multiple assignment in standard AppleScript, can be used instead of the **`returning properties`** parameter to retrieve properties of windows and window items.

The **`returning`** parameter is a “pattern matching” parameter that can be used either explicitly or implicitly, as in these two equivalent examples of assignment statements:

```
get the bounds of x returning {l, t, r, b}
set {l, t, r, b} to the bounds of x
```

Here we use the **`returning`** parameter to retrieve window properties from a modal dialog:

```
open window “untitled”  ¬
   returning {height:ht, position:{h, v}, closing item:{name:nm}}
```

The retrieved values are returned directly to the variables that are given (instead of values) as place holders. The key idea is to mimic the structure of each property's value class. In the example, the **`position`** property is a list of two integers, so we mimic that list structure. We need not include every element, however; the **`closing item`** property is a record of six fields, but we include a reference to only one of them.

Note that the word "properties" is not a part of this example.

Setting and Retrieving Properties in One Statement

The **`with properties`** and **`returning properties`** parameters may be combined in the same **`open window`** statement in this manner:

```
open window “Stuff” with properties {list of records} ¬
   returning properties {list of records}
```

Or you could change the order:

```
open window “Stuff” returning properties {list of records} ¬
   with properties {list of records}
```

You can use **`returning`** instead of **`returning properties`**, if you wish.

Preserving and Restoring the Changes of a Window

As an application window is closed, its **`changes`** property contains a list of the values of all window and window item properties that might have been changed by the user during run time. An application can preserve the changes that have been made within a window by storing the value of the **`changes`** property in a global variable or application property as the window is closed, and restoring the changes each time the window is opened.

To preserve and restore the changes of a window, we would first define a script property called, for example, **`userEdits`** in the project script of the application:

```
property userEdits:{}
```

Then create a handler that stores the **`changes`** property of the window in the **`userEdits`** property. It could be a **`close`** handler as in this example in the window script:

```
on close theObj
      set my userEdits to changes of theObj
   continue close theObj--let the window close!
end close
```

Once saved in this manner, the changes can be restored to the window using the **`with properties`** parameter of an **`open window`** statement in the project script:

```
open window “Total Recall” with properties userEdits
```

Saving a User-Edited Window

One way to save the state of a window that was edited by the application user is to save the entire window. Windows can also be saved to their applications with the **`save`** statement:

```
on close theObj
   save theObj
   continue close theObj--let the window close!
end close
```

Saving a window in this manner replaces the window template resource from which it was created with a resource describing the current state of the window.

To avoid replacing the original window template resource with the one you are saving, give the window a different name before saving it. This example saves a copy of the window template under a new name:

```
on close theObj
   copy name of theObj to windowName
   set name of theObj to (windowName & “User”)
   save theObj
   continue close theObj--let the window close!!
end close
```

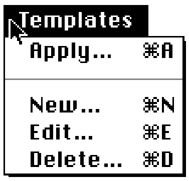
Controlling Menus

Application and Private Menus

In FaceSpan applications, a menu can belong to the application as a whole, or can be one of the private menus of a particular window. A window's private menus are added to the menu bar when the window is opened or activated, and removed when the window is closed or otherwise inactivated. Menus belonging to the application are added to the menu bar when the application runs and removed when it quits or is stopped.

Handling Chosen Messages

Each time the user chooses a menu item from a menu, a **chosen** message is sent to the frontmost document window, so that the window can act upon it. The **chosen** message's parameter will refer to a menu item of an application menu or of a private menu of the active window. Let us assume that a menu resembling the "Templates" menu in the picture belongs to a window:



In the window's script there will be a handler for the **chosen** message. This handler will receive all **chosen** messages from all user interactions with all menus. The handler must first find out if the **chosen** message concerns the window's own menu, the "Templates" menu:

```
on chosen theObj
    Get useful information about theObj, which is a reference
    --to the menu item that the user chose:
    copy the name of theObj to theMenuItem
    copy the name of the menu of theObj to theMenu

    if theMenu is "Template" then
        -- Handle messages from menu "Templates"
        if theMenuItem is "New…" then
        --handle New command here
        else if theMenuItem is "Apply…" then
        --handle Apply command here
        else if theMenuItem is "Edit…" then
        --handle Edit command here
        else
        --handle Delete command here
        end if
    else
        -- Let the application handle messages from the other menus:
        continue chosen theObj
    end if
end chosen
```

This handler intercepts **`chosen`** messages sent to the window and calls appropriate subroutines only when the user chooses items of menu "Templates," the only private menu of the window. The **`continue`** statement near the handler's end continues messages concerning other menus to another **`chosen`** handler in the script of the application. The project script would handle **`chosen`** messages from the project's other menus.

Only document window scripts and the project script can handle the **`chosen`** message, since floating windoids are not sent the message, and the menus are inactive when a modal dialog is open.

By the way, FaceSpan automatically handles the Cut, Copy, Paste and Clear commands in the Edit menu.

Controlling Other Applications

The primary purpose of AppleScript is to control other applications by way of scripts. This section tells how FaceSpan promotes that purpose and supports you in your efforts to control scriptable applications.

Since we are discussing various kinds of applications, we need to distinguish among them. We will call an application that we wish to control with scripts a "target application," and an application we are developing with FaceSpan a "FaceSpan application," both of which are distinct from "FaceSpan itself."

Scriptable Applications

A scriptable application is one that has been specifically written to respond to AppleScript and to share its data and operations with other applications by way of scripts. Every application has scriptability in some sense, since all can be told to open, run, print and quit. But if an application has been written to respond to AppleScript, you can control the details of its operation as well as send and receive data to and from it. So, to control an application with AppleScript and FaceSpan, it must be scriptable.

Terminology

AppleScript provides a general language for describing what you want a target application to do, but every scriptable application uses special terminology to describe the its components, operations, and data. This terminology is stored in a dictionary, which is a component of the application and accessible to AppleScript and FaceSpan. An example of this application-specific terminology is FaceSpan's own terminology—object names such as **`checkbox`**, **`window`** and **`push button`**, and properties such as **`hilite`**, **`position`** and **`pen color`** that are unknown in AppleScript proper.

You have to learn the terminology of a scriptable application if you wish to write scripts to control it. Fortunately, life is simpler than it appears, in this case, because many applications have similar components and handle similar data, and so their developers have defined similar terms.

Better still, Apple and a consortium of application developers have defined standard "suites" of terms. The "text suite," for example, is a set of standard terms and meanings to be used by all applications that have text-processing components. Because developers have been good about adopting these terminology suites, you can apply what you learn about one target application to another of the same category.

Scripting Support

While you create scripts to control a target application, FaceSpan supports you in several ways. First, FaceSpan’s Dictionary Windoid is a ready reference to the terminology of the scriptable application. Second, you can use the Message Windoid to try out commands or entire scripts before you put them into your project. Third, the structure of your project lets you partition your scripts in convenient ways. For example, different buttons in your project’s windows can tell an application to do different things.

Keeping Ideas in Order

When you develop a FaceSpan application to control a target application, you are in triple jeopardy: you use AppleScript terminology, FaceSpan terminology and the target application’s terminology. Here, as it might appear in a Script Editor window, is a sample script that illustrates the problem:

```
tell application “FileMaker”
    --Get a list of all the names from the database.
    set numRecords to Count of Record of Layout 1
    set nameList to {}
    repeat with i from 1 to numRecords
        --Get each name, save in list:
        set LastName to (cellValue of Cell “Last Name” of Record i)
        copy nameList & {LastName} to nameList
    end repeat
end tell
set listbox “lstNames” to nameList
```

To keep the terminology straight, note that:

➤ AppleScript is the overall language that provides structure to your scripts. The keywords (usually bold in script listings), connecting words, a few verbs like “set” and “copy” and the fundamental data types—lists, integers, records, strings—are provided by AppleScript. All the non-bold, non-underlined words, numbers and symbols in the example script are typical of any script.

➤ FaceSpan’s terminology refers mainly to the set of interface elements behind which you put scripts. Most of the nouns and verbs in scripts will be unique to FaceSpan or common to FaceSpan and the target application. The underlined word in the last line of this example is FaceSpan terminology.

➤ A target application’s own terminology alone controls that application. All the underlined words, except the last, are FileMaker terminology.

Table of Contents Index

By the way, you can control the formatting of scripts in your applications; use the AppleScript Formatting command in the Script menu.

See “An Approach to Application Development,” later in this chapter, for a development strategy that helps you keep control of terminology.

Scripting Target Applications

The purpose of AppleScript, and a major purpose of FaceSpan, is to script other applications, especially third-party applications. (Here we use “script an application” to mean “write scripts to control an application.”)

Guidelines

As described above, a **`tell`** statement is used to direct a script at the target application you wish to control. Here is a simple example:

```
tell application “Excel”
    activate
    --Set up the row-column range string:
    copy “R1C1:R” & (count of theInfo) & “C2” to chartRange
    --Now open a spreadsheet and insert the data (assumed to be
    --a list of 2-item lists in theInfo):
    make Document
    set Range chartRange of first Document to theInfo
    set selection of first Document to Range chartRange
    --Set chart parameters, then make the chart:
    set charttitle to “Sales”
    set charttype to bar
    set chartlegend to false
    make Chart
end tell
```

When AppleScript attempts to compile or decompile this script, it looks for the target application called “Excel” so that it can open the application’s terminology dictionary.

If you had just entered the text of this script, you would be asked to locate “Excel.” AppleScript uses the target application’s dictionary to determine how each term gets translated into its compiled form. If you later make changes to the script (other than the name of the application), AppleScript remembers where “Excel” is, and so does not ask.

On the other hand, if your client tries to open a project with this script compiled into it (on a different Macintosh, of course), AppleScript will ask your client to locate “Excel” again—this time because it needs to find out how to decompile the compiled form of the script back into readable text.

One way to avoid the problem is to make sure that the distributed application is put into a folder along with aliases to the target applications. However, these aliases must have the same names as the target applications had when the application was last compiled.

No variables for targets

It is important to understand why the following example will not work. It asks for the name of a target application to control, then tries to send commands to it:

```
--Get the name of an application:
set theName to (choose application with prompt “Open an app:”) as string
--Tell it to do some work:
tell application theName
    activate
    --and so on
    make Chart --for example
end tell
```

The application must be known *when the script is compiled;* otherwise, AppleScript cannot know what the non-AppleScript, non-FaceSpan terms mean.

Similarly, if you try to open a script for editing, and you respond to AppleScript’s request for the target application with the wrong application, the script will be displayed, but it might be full of cryptic sequences like «event Rtsj» and «class Tmqz». These are the compiled forms of terms for which AppleScript could not find terms in the dictionary of the target application you gave it.

Organizing your scripts

AppleScript tries to find a terminology dictionary each time it encounters a **`tell`** statement for an additional target application, or when it encounters a **`tell`** statement in the script of another FaceSpan object. This becomes a nuisance only when you place scripts for the target application throughout

Table of Contents Index

your project. When you distribute such an application to your clients, each client will be asked several times for the same target application. This can be confusing, as well as a nuisance.

The solution to the problem rests upon an appropriate organization of your FaceSpan project: put all references to a target application in the script of just one FaceSpan object.

Perhaps the best solution is to use a script object to hold the subroutines (handlers) and all the variables (properties) that those subroutines need:

```
script SpreadSheet
    --Operations on a spreadsheet application--here we use Excel.
    --Put properties used only in this script object here.

    on doLaunch()
        tell application “Excel”
          --Put commands to start the application here.
        end tell
    end dolaunch

    on doOpen(fileName)
        tell application “Excel”
          --Put commands to open a file here.
        end tell
    end doOpen

    on doGetData(theData)
        tell application “Excel”
          --Put commands to get the data here.
        end tell
    end doGetData

    --...etc.
end script
```

The script object would be in the project script if several windows must call its handlers, or in a window script otherwise. A typical call would look like this:

```
set fileName to ((choose file of type “FMPR”) as string) --get the data file

tell my SpreadSheet to doOpen(fileName) --call a spreadsheet subroutine to
--tell Excel to open the file
```

Another solution is to try to use just one `tell` statement for each target application. This can be done by writing a single subroutine to handle all commands to the target application. Here is an outline for such a subroutine:

```
--Operation “selectors” to use when calling the Spreadsheet subroutine:
property doLaunch: 1 --launch the database or spreadsheet app
property doOpen: 2 --open the database or spreadsheet file
property doGetData: 3 --get data from user's database file
...etc.
on SpreadSheet(theOp, theInfo)
    --Operations on a spreadsheet application--here we use Excel.
    tell application “Excel”
        --Use theOp to select the commands to execute:
        if theOp is doLaunch then
          --Put commands to start the application here.
          --theInfo might simply be ignored.

        else if theOp is doOpen then
          --Put commands to open a file here.
          --theInfo could be the name of the file.

        else if theOp is doGetData then
          --Put commands to get the data here.
          --theInfo might be a list of data values.

        --...etc.

        end if
    end tell
end SpreadSheet
```

Table of Contents Index

This subroutine would be located centrally, most likely in the project script. A typical call might look like this:

```
set fileName to ((choose file of type “FMPR”) as string) --get the data file

SpreadSheet(doOpen, fileName) --call a spreadsheet subroutine to
--tell Excel to open the file
```

Points of view

You can script the target application from one of two points of view. One view is to consider the target application as the central concern. Your FaceSpan application would then be subordinate to it, perhaps acting as a palette or control panel or tool bar. A FaceSpan application built from this point of view is likely to take up little screen area, and its labeling will be derived from that of the target application.

The other view is to make the FaceSpan application the center of attention, using the target application simply as a tool for adding functionality to the FaceSpan application. In this case, the interface will be what makes the most sense, not what is forced upon you by the target application. In particular, you probably will want to hide the target application’s windows so that they do not distract attention from your interface.

Scripting the Finder

The Finder, the application that displays the desktop and controls all its operations, is a scriptable application, so it can be the target of a FaceSpan application. Its terminology dictionary is in a file called “Finder Scripting Extension” in the Extensions folder of the System Folder. (The Finder’s dictionary is always available in the Dictionary Windoid.)

Here is a very simple sample script:

```
tell application “Finder”
    activate
    close every window
end tell
```

The scriptable Finder is also recordable. You can open a FaceSpan Script Editor window, click the Record button, then go to the desktop and go through the steps you want to get done; the script will write itself.

Scripting Other FaceSpan Applications

The easiest kind of application you can script is one developed with FaceSpan. There are two reasons for this. The first is that applications developed with FaceSpan understand all the FaceSpan terminology, so any command you can use *within* a FaceSpan application can be sent *to* a FaceSpan application. For example, application “TestApp” might have this statement in one of its scripts:

```
set the pen color of textbox “txtTitle” of window “Flashy” to black
```

Because there is a textbox “txtTitle” and a window “Flashy,” another application can script “TestApp” the same way:

```
tell application “TestApp”
    set the pen color of textbox “txtTitle” of window “Flashy” to black
end tell
```

The other reason that a FaceSpan application can be easy to script is that you can define your own handlers or other subroutines within that application. For example, you could put the statement of the previous example into a subroutine called “SetPen” in the project script of “TestApp”:

```
on SetPen()
    set the pen color of textbox “txtTitle” of window “Flashy” to black
end SetPen
```

Then another application can script “TestApp” more briefly, this way:

```
tell application “TestApp” to SetPen()
```

In other words, you have the opportunity to make your FaceSpan applications as easily scriptable as you wish.

Scripting FaceSpan Itself

FaceSpan itself can be the target of your scripting efforts. FaceSpan is, of course, an application development environment, so your efforts will be devoted to controlling how applications are built.

There are two ways in which you might script FaceSpan itself. In the first case, you can enter, edit and execute scripts in the log area of the expanded Message Windoid. You might use simple commands to set properties that cannot be set another way, but you might also write **`repeat`** loops or more complex scripts to create or set the properties of a group of window items.

The other way to script FaceSpan itself is to create FaceSpan applications that serve as additional tool palettes. For example, let's say that FaceSpan is open, that there is a window open for editing, and that a window item is selected. Then an application can refer directly to the selected item:

```
tell application “FaceSpan”
    set theSel to the selection of window 1 --get list of selected items
    set theItem to item 1 of theSel --first (or only) item selected in window
    set the pen color of window item theItem to newColor
    --and so on
end tell
```

Your scripts can find out how many items there are, **`get`** and **`set`** any property of any item or of the window itself, and—using the **`make`** command—create new window items in the window. So it is possible, for example, to create a palette that has a button labeled “Dialog” which, when pressed, turns the current window into a standard modal dialog by resizing it and making the button, text and icon items.

Using Scripting Additions

AppleScript Language Extensions

Scripting additions are files that provide additional commands you can use in scripts. They extend the AppleScript language, giving it new features, and new terminology for those features. There are, for example, scripting additions to sort lists alphabetically, to play sounds and to access data stored in popular database systems.

Because they are written in Pascal, C or assembly language, scripting additions can execute complex algorithms faster than AppleScript or other scripting languages, and they can access external resources, such as databases, in ways not directly available to scripts.

Scripting additions have dictionaries that you can view using FaceSpan's Dictionary Windoid. The dictionary gives the syntax, terminology, and parameter classes for a scripting addition.

A scripting addition is often called an "OSAX," which stands for Open Scripting Architecture eXternal commands.

Writing Scripting Additions

Scripting additions are written in Pascal, C, or assembly language. They usually are distributed as extension files that must be dragged into the Scripting Additions folder (located in the Extensions folder of the System Folder).

Using Scripting Additions in Applications

If it becomes necessary or useful to employ scripting additions in your applications, you need to think about distribution. You will find that FaceSpan makes it easy: you can include copies of scripting additions right in your project. People who use your applications will not even have to know that scripting additions are used. To include a copy of a scripting addition in a project:

1 Select the Forms, etc. View radio button in the Project Window.

2 Click the Import button.

The standard Open dialog appears.

3 Locate and select the scripting addition, then click the Open button.

Table of Contents Index

A copy of the scripting addition is brought into the project, and its name is listed among the other form names.

Please note, however, that while you are developing the project, the original scripting addition must remain in the Scripting Additions folder, where the AppleScript compiler expects to find it.

It also is important to know that a single scripting addition file might contain several scripting additions. Use FaceSpan’s Dictionary Windoid to be sure that the names do not conflict with the names in other scripting additions that you import, or with terms used in your application.

Copyrights

Remember that many scripting additions are not yours to distribute. Others may be distributed with permission, some require that you pay a shareware fee, and so on. Do not include a scripting addition in a project unless you know that it is permissible to do so. Even without a copyright attribution, the author of a scripting addition still holds the copyright.

An Approach to Application Development

This section offers suggestions about how to start a project, how to structure it for best results and how to finish it.

Incremental Development

The FaceSpan development and testing environment is so flexible and immediately responsive that you can develop applications “incrementally”—that is, you can design, develop or test the application one feature at a time. This is an especially handy approach to testing, since you must be sure one script works before you test another that depends upon it. Incremental development is similarly useful for interfaces, since they are tightly integrated with the scripts.

Interface First

There are several reasons why you should develop the interface of an application as the first step:

➤ Creating the interface first makes you consider all the information that you want to show to the application’s user.

➤ Controlling the interface might require more scripting than controlling the target application (the application that your FaceSpan application is to control, if any).

➤ The interface often is most critical to a project’s success, and often is the feature upon which projects are judged.

➤ Since the interface is the “public” part of the project, it is the part most likely to be changed to satisfy others.

A good way to avoid confusion between the terminology of FaceSpan and the terminology of the target application is to write all the scripts necessary to control FaceSpan’s interface elements before you write scripts to control the target application. To script the interface as thoroughly as possible before scripting the target application, you might have to make up example data that ultimately will come from the target application.

Finally, write the scripts to control the target application, and to pass information back and forth between the interface and the target application.

Table of Contents Index

Scripting and Code Structure

There are two general ways to structure a FaceSpan project. One is to put all the important routines into the project's script, then have the interface elements call those routines to get the work done; this would be called a "centralized" scheme. The other strategy is to put the routines right into the scripts of the interface elements that call them; this would be a "decentralized" scheme.

In fact, a good way to organize a project is to distribute the routines that control the interface elements, and to centralize the routines that control the target application. Interface-control scripts are thus kept close to the objects they control, while the closely interrelated routines that control the target application are found in one area, where they can be viewed and maintained at the same time.

This organization then requires that you include both centralized and decentralized routines to exchange data with some of the interface elements. For example, a textbox will have to call a central routine to get what it needs to display, and a central routine will call a textbox's routine to get text that was typed into it.

Refinement

There always is more work to do after an application is "finished"—that is, after it looks good and behaves properly. Since it is likely that you will be maintaining your project for some time, you will want to make maintenance as easy as you can.

The first thing to do is to delete all the unused project resources that accumulated during development, such as extra windows, menus, artwork and forms.

In your scripts, remove unused properties and variables and the statements that you commented out when you changed your code.

Make another pass through all the scripts and make sure that you use the same names for the same entities throughout. When this is done, see if your handlers and subroutines can share data by passing it as parameters, rather than by putting it into global variables that all can access.

Be sure that you have commented your scripts adequately and correctly. Comments really should be written before or during scripting. They should include descriptions of the assumptions you made about the data, decisions you made in the design, and the meanings of the data structures. (Comments in AppleScript begin with two dashes, “--”, and continue to the end of the line.)

Finally, do not forget to set up the “About...” dialog to identify yourself and your program and to display your copyright.

Part III: FaceSpan Object and Language Reference

Chapter 11: Applications

Contents:

Table of Contents Index

Applications

Application icon

“Droppable”
Application icon

The application object is the overall container or parent of all the objects in your application. It has properties that pertain to the overall behavior of the application **`clipboard`**, **`mouse position`**, **`cursor`**, the positions of the modifier keys, and so on.

There are several command and event messages sent specifically to application objects, such as **`open`** (for drop-launching), **`quit run`**, and others.

Since the application contains all other objects, it can intercept and handle messages sent by commands and interface events to any object. It might handle such messages if an object does not handle them, or if the object continues them.

Applications can be launched from the desktop when double-clicked. They can also be made to “drop-launch,” which means that they run when document or folder icons are dropped onto their desktop icons. An application is made to drop-launch simply by including a handler for the open message in the project script.

Reference Forms

There are two ways to refer to the application object in a script:

```
current application
application “application name”
```

The **`cursor`** property of the current application would, for example, be referenced as:

Most application properties are unique, however, so they can be referenced without qualification—unless you are referring to properties of another application, one that is running at the same time. Only the **`focus, idle delay, name`** and **`script`** properties must be fully qualified to avoid ambiguity.

Table of Contents   Index

Application Properties

clipboard

The data contained on the Clipboard.

Value Class

any (see notes)

Examples

```
copy clipboard to theClip
if class of theClip is string then
   display dialog "It's a string."
end if
```

Notes

➤ The value class of the **`clipboard`** property depends upon the class of the information it contains. This can be **`string, list, record, integer`** or **`real`**.

➤ To find the value class of the **`clipboard`**, it must be copied to a variable, then the variable tested for the value class.

command down

Is the Command key pressed?

Value Class

boolean

Examples

```
if command down then
   DoOneThing(x)
else
   DoAnother(x)
end if
```

Note

➤ **`Command down`** is a read-only property.

control down

Is the Control key pressed?

Value Class

boolean

Examples

```
if control down then
   DoOneThing(x)
else
   DoAnother(x)
end if
```

Note

➤ **`Control down`** is a read-only property.

cursor

The identity of a cursor resource.

Value Class

integer	cursor (“CURS”) resource id number
string	cursor (“CURS”) resource name
constant	standard / none

Examples

```
 --Get the current cursor:
set saveCursor to the cursor
--Set the cursor to a custom shape:
set the cursor to “Special”
...etc.
--Restore the cursor:
set the cursor to saveCursor
```

Notes

➤ **`Cursor`** can represent the name or id of a cursor that has been imported into the project from the Artwork View of the Project Window.

➤ **`Cursor`** can be set to the constant **`standar`**`d` to restore it to its standard shape. That shape depends upon the context.

➤ If **`cursor`** is set to **`none`**, it becomes invisible until moved.

➤ You can get or set the **`cursor`** property; this lets you save and restore the current cursor.

focus

The window item that is receiving keystrokes (or would, if the application were active).

Value Class

reference

Examples

```
set theObj to focus of application “Data Viewer”
copy window of focus of application “My Editor” to theFocalWindow
```

Notes

➤ The **`focus`** of an application remains set even while the application is suspended.

➤ The **`focus`** of an application is a read-only property.

frontmost

Is the application active?

Value Class

boolean

Examples

```
copy frontmost of application “Document Shredder” to isAtFront
if frontmost of application “MyScriptEditor” then open window “Debugger”
```

Note

➤ **`Frontmost`** of an application is a read-only property.

heap space

Reports the amount of free memory available to the application.

Value Class

integer

Example

```
set label “lblheapValue” to heap space
```

Notes

➤ **`Heap space`** and **`stack space`** help you to better monitor and respond to low-memory situations.

idle delay

The frequency with which the application receives idle messages.

Value Class

integer

Examples

```
copy idle delay of application “Simulator” to saveDelay
set idle delay of application “Simulator” to 1
```

Notes

➤ The **`idle delay`** is given in seconds; the default is 2 seconds.

➤ If a window and its application have different **`idle delay`** values, the **`idle delay`** of the window is ignored unless it is greater than that of the application.

➤ An **`idle delay`** of 0 allows the application to receive idle messages as often as possible.

interruptible

Can Command-period cancel scripts?

Value Class

boolean

Examples

```
if interruptible then
   set textbox “txtMessage” to  “Press Command-period to cancel.”
end if
```

Note

➤ **`Interruptible`** is a read-only property.

mouse down

Is the mouse button pressed?

Value Class

boolean

Examples

```
if mouse down then
   set the fill color of box “boxButton” to black
else
   set the fill color of box “boxButton” to white
```

Note

➤ **`Mouse down`** is a read-only property.

mouse position

The position of the mouse in global coordinates.

Value Class

point	{horizontalOffset, verticalOffset}

Examples

```
set {mouseH, mouseV} to mouse position
set {wdwLeft, wdwTop, wdwRight, wdwBottom} to bounds of my window
--Find and use the local (window) coordinates of the mouse:
set localH to mouseH - wdwLeft
set localV to mouseV - wdwTop
set the position of box “boxMover” to {localH, localV}
```

Notes

➤ The position {0, 0} is the upper-left corner of the main screen.

➤ **`Mouse position`** is a read-only property.

name

The name of the application.

Value Class

string

Examples

```
copy the name of the application of theObj to appName
copy the name of the current application to myAppName
```

Note

➤ **`Name`** is a read-only property.

option down

Is the Option key pressed?

Value Class

boolean

Examples

```
if option down then
   DoOneThing(x)
else
   DoAnother(x)
end if
```

Note

➤ **`Option down`** is a read-only property.

screen bounds

The bounding rectangles of all attached displays.

Value Class

a list of bounding rectangles

Examples

```
copy the screen bounds to boundsList
--Get the coordinates of the main screen:
copy item 1 of boundsList to {myLeft, myTop, myRight, myBottom}
```

Notes

➤ The first rectangle in screen bounds is the bounds of the main screen.

➤ **`Screen bounds`** is a read-only property.

screen depths

Bits per pixel of all attached displays.

Value Class

a list of small integer

Examples

```
set theDepths to screen depths
--Get the depth of the main screen:
copy item 1 of theDepths to mainDepth
```

Notes

➤ The first number in **`screen depths`** is the color depth of the main screen.

➤ **`Screen depths`** is a read-only property.

script

The compiled script of the application.

Value Class

script

Examples

```
set the script of application “Test” to contents of textbox “txtTester”
set the contents of textbox “txtViewer” to (script of current application)
```

Notes

➤ When a string containing a script is assigned to the **`script`** property, it is automatically compiled; you must handle (with a **`try`** statement) any error encountered during compilation.

➤ Coercing the **`scrip`**t property to string or **`text`** decompiles it.

shift down

Is the shift key pressed?

Value Class

boolean

Examples

```
if shift down then
   DoOneThing(x)
else
   DoAnother(x)
```

Note

➤ **`Shift down`** is a read-only property.

stack space

Reports the amount of free memory available to the application.

Value Class

integer

Example

```
set label “lblStackValue” to stack space
```

Notes

➤ Heap space and stack space help you to better monitor and respond to low-memory situations.

ticks

The number of ticks (60ths of a second) since the machine was last turned on. This can be helpful in implementing timed behaviors; the difference between the values returned by two references to the `ticks` property is a precise measure of elapsed time.

Value Class

integer

Example

```
property timedelay : 3600 -- this property = 1 minute. ticks are counted in
60th's of a second.
on run
    repeat
        set startticks to ticks
        repeat
          if (ticks) > startticks + timedelay then
            exit repeat
          end if
        end repeat
        beep 5 -- this will beep 5 times every minute with the value 3600 for
        the timedelay variable
end
```

version

The version of FaceSpan that was used to create the application.

Value Class

string

Examples

```
copy version of application “Article Accelerator” to theVers
```

Notes

➤ The **`version`** property is given as a string, not a number, since it might have two decimal points.

➤ **`Version`** is a read-only property.

Application Command and Event Messages

The application object is the overall container or parent of all other objects; it contains all windows, which contain all window items. Thus, the project script (the script of the application object) can intercept and handle messages sent by commands and events to any object. This could happen if the object does not handle the messages itself, or if the object continues the messages.

See the window and window item references for command and event messages sent specifically to those objects.

Listed here are a few additional command and event messages that are sent directly to applications. The listing tells the source of the message—either a command issued from a script, or an event from the system or from user interaction.

Any message sent by an event can also be sent by a command; the name of the message is the name of the command.

click as user

Lets you script "click" the mouse anywhere on the screen or within a specified window. You can even make it click on a button with a particular name.

Example

```
click as user on button titled "OK"
```

Note

➤ This command lets your scripts mimic the actions of a real user within other applications. Using this command, you can automate the operations of applications that have no built-in support for scripting. See also *type as user* on page 225.

Table of Contents Index

do script

Command to execute a script.

Parameters

(direct)	string	a script

Examples

```
do script “repeat with i in 1 to 5" & return  ¬
   & “set enabled of checkbox i of window 1 to false” ¬
      & return & “end repeat”
tell application “FileMaker” to do script fmScript
```

Note

➤ The **`do script`** command is in FaceSpan mainly for completeness, since most scriptable applications accept it.

idle

Idle event messages sent by the system.

Parameters

(direct)	reference	object to which idle was sent

Notes

➤ **`Idle`** events are received from the system once every two seconds (or at the interval specified by the **`idle delay`** property) when no other events are occurring.

➤ If a window and its application have different **`idle delay`** values, the idle delay of the window is ignored unless it is greater than that of the application.

➤ A window can continue the **`idle`** message to its application; hence, the direct parameter (reference) might not be the application itself.

➤ The application will not send idle messages if a window is not open. You can create a window and make it invisible or move it off the screen.

make

Command to create a new object.

Parameters

(direct)	class	class of intended item
at	integer	position in container beginning or end
of	reference	intended container

[with record description of object properties]

Examples

```
make menu item with properties {name:”Overview”, mark:”•”} ¬
   at beginning of menu “Views”
make textbox with properties {class:textbox, ¬
   bounds:{0,0,100,100}, contents:”Voila!”, ¬
   editable:true, position:{99,8}) at end of window 1
make new window with properties ¬
   {{bounds:{77, 101, 475, 301}, form:modal dialog, ¬
   titled:true, zoomable:false, private menus:{}, ¬
   name:”Bob”}, {class:push button, ¬
   bounds:{195, 151, 275, 171}, name:”cancel”, ¬
```

Notes

➤ A new window can be made at any time, from any script; this does not create a new window template. If a script saves the window with the **`save`** command, a new template will be saved into the application; it can later be edited in edit mode.

➤ A window cannot make a new window item within itself, nor can a window item make a new item within its own window.

➤ When making a window item, the at parameter, if used, specifies the position of the new item in the layering; **`beginning`** means the bottom layer, while end means the top layer.

➤ When making a window item, the **`with properties`** parameter is required, and must include the name and bounds properties.

➤ New listbox items can be made in any listbox.

➤ New menu items can be made in any menu currently displayed; the menu template is not changed.

➤ New menu items can be made in any popup; some menu item properties do not apply to menu items in popups.

➤ New storage items can be made at any time from any script; they are persistent from run to run.

open

Drop-launch event message from the system.

Parameters

(direct)	list of alias	documents and folders dropped on the icon

Example

```
on open theFiles
   repeat with i from 1 to count of theFiles
      copy item i of theFiles to nextFile
      --Do something with this file alias:
      set itsPath to (nextFile as string)
      ProcessFile(itsPath)
   end repeat
end open
```

Notes

➤ An application is drop-launched by dropping desktop icons onto the application’s icon.

➤ The **open** message takes a single parameter whose value is a list of aliases. These are the aliases of all the items whose icons were dropped on the application’s icon. You can convert the aliases to path names by using as string.

➤ The **open** message is sent even if the application is running when the items are dropped.

➤ An **`open`** message is not sent when the application is opened by double-clicking its icon or by a **`tell`** statement in a script. See the discussion of the run event message.

➤ The **`open`** message is not sent when a window opens; instead, a **`prepare`** message is sent. Use a **`prepare`** handler to adjust the window or its window items as the window opens.

print / print setup

Provides control over paper margins and printing dialogs.

Example

```
print setup paper margins {72,72,72,72} with setup dialog and job dialog

print window 1
```

Notes

➤ The first command in the example above would create 1-inch margins and present the user with setup and job dialogs.

➤ The second command in the example above would print window 1 by expanding its size (and its items according to their **`growth`** properties) to conform to the specified paper dimensions (72 points from each edge, in this case).

➤ Items without scrollbars appear on every page. Items with scrollbars are automatically paged until their contents run out. For convenience, these paged items do not print either the images of their scrollbars nor their enclosing frames.

quit

Command to quit execution.

Parameters

(none)

Example

```
on quit
   try
      display dialog “Are you sure it’s OK to quit now?”
      -- (user OK’d)
      continue quit
   on error
      -- (user canceled)
   end try
end quit
```

Note

➤ As shown in the example, a **`quit`** message that is handled must be continued, or the application will not quit.

run

Event message, from the system, to run the application.

Parameters

(direct)	reference	the application

Examples

```
on run theApp
   --Put “loose” executable statements in here.
end run

tell application “Example”
   --Open the application and send a run message:
   activate
   …etc.
end tell

tell application “Example”
   --Open, but do not send a run message:
   launch
   …etc.
end tell
```

Table of Contents   Index

Notes

➤ The **run** message is sent when the user double-clicks the application icon to open the application.

➤ The **run** message is not sent when the user drop-launches the application. See the discussion of the **open** event message.

➤ The **run** message can be sent (or not) when an application is opened by a tell statement in a script, as shown in the examples.

➤ All the "loose" statements in the project script are treated as a default **run** handler, where "loose" means they are not explicitly contained in a handler or subroutine.

➤ You can put an actual **run** handler into the project script; the application can have either a default **run** handler or an actual **run** handler, but not both.

save

Command to save the current configuration of a window.

Parameters

(direct)	reference	the window to save

Examples

```
save window “Preferences”
```

Notes

➤ If you **save** a window, close and reopen it, it will reopen in the configuration (including the **position**) in which it was saved.

➤ The window's saved configuration persists from one execution of the application to the next.

type as user

This command lets you type a sequence of characters into a text area of the target applications. It even simulates holding down the various modifier keys, such as the Command or Option key.

Example

```
type as user “p” with command down
```

Note

➤ This command lets your scripts mimic the actions of a real user within other applications. Using this command, you can automate the operations of applications that have no built-in support for scripting. See also *click as user* on page 219.

Chapter 12:

Windows

Contents:

Table of Contents Index

Windows

A window in a running application is composed of a window object—which forms its foundation—and window items, such as buttons, icons, and textboxes. Windows templates constructed with FaceSpan's Window Editor and saved with a project's resources are used as models for the windows opened while the application is running.

This chapter describes the three classes of windows, their reference forms and properties, as well as the standard command and event messages they can handle. For detailed information about window items, refer to Chapter 13: "Window Items".

Classes of Windows

Like the window items they contain, window objects are defined by properties. They can be scripted to respond to messages received from the user interface and from the scripts of other objects.

Each window object can be set to any one of the three classes of windows: document window, modal dialog, and floating windoid. Each class looks and behaves somewhat differently at run time.

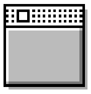
Document windows are generally used to display data that is editable by the user. A document window can remain open in a suspended state while other windows or applications are active, and can then be reactivated by clicking. It can contain a title bar with which it can be dragged around the screen. If a standard window contains a title bar, it may also contain a close box, a zoom box, a resize box, or any combination of these.

Table of Contents Index

Modal dialogs are typically used to obtain information from, and to give instructions to, application users so that processes can be concluded. Modal dialogs cannot be suspended or covered by documents or windoids—only by other modal dialogs.

Once a modal dialog has opened on screen, the user must dismiss it before activating another window in the same project or application.

A modal dialog is surrounded by a four-pixel-thick frame. It can have a title bar, which makes it movable. A modal dialog cannot contain a close box, zoom box or resize box; it is usually closed using a button.



Floating windoids often are used to display control objects and utility information. On the desktop, windoids cannot be covered by documents; only by modal dialogs and other windoids. When an application is suspended, its windoids become invisible until the parent application is reactivated. Each floating windoid has a black-and-white drag bar at its top. The drag bar cannot be removed, but it may contain an optional title, close box, and zoom box. A windoid can contain a resize box.

Reference Forms

Windows can be referenced by name, by index, by id number or as the window of a given window item, any of which can be the value of a variable:

➤window “Preferences”

➤window 3

➤window id 20973248

➤window of theObj

➤window of menu “Sales”

Note that the **`id`** number is not fixed; it may differ each time the window is opened.

Properties of windows

bounds

The global coordinates of the content area of the window.

Value Class

list of integer	bounding rectangle {left, top, right, bottom}

Examples

```
copy the bounds of window 1 to {wLeft, wTop, wRight, wBottom}
--Make the window twice as tall:
set wBottom to wBottom + (wBottom-wTop)
copy {wLeft, wTop, wRight, wBottom} to the bounds of window 1
```

Note

➤ The **`bounds`** of a window are expressed as offsets, in pixels, from the top-left corner of the main screen. That corner is at **`position`** {0, 0}.

changes

The changeable properties of the window and its window items.

Value Class

list of records	{{propertyName:value,…}, {re:7001,propertyName:value,…}, {re:7002}, {re:7003,propertyName:value,…},…}

Examples

```
set allChanges to changes of window “Source Document”
--Extract one of the changed properties:
set wdwChanges to item 1 of allChanges
set wdwBounds to bounds of wdwChanges
--Reopen the window with the original changes:
open window “Source Document” with properties wdwChanges
```

Notes

➤ The **`changes`** property is a list of records; each record gives the identification of an object and its user-changeable properties.

➤ The first record returned contains the changes to the window itself. Each subsequent record refers to a window item, with the **`re`** property indicating the **`id`** of the window item.

➤ **`Changes`** is not a complete list of properties (compare the description property), but rather a list of only the user-editable properties of the window and of its window items.

➤ All window items of the window are represented in the list; the records for those without changeable properties contain only the **`re`** property.

➤ When you close a modal dialog whose **`open window`** statement did not include the **`returning`** parameter, **`changes`** is placed by default into the **`result`**.

class

The object class of the window.

Value Class

constant	Modal Dialog/Non-Modal Dialog/Document Window/ Alert Dialog/Floating Windoid

Examples

```
if class of anObj is dialog then close window anObj
```

Notes

➤ See the **`form`** property; it is the property that establishes the class and the basic appearance of the window.

➤ See also the **`modal`** and **`floating`** properties.

➤ **`Class`** is a read-only property; it cannot be set.

closeable

Does the window have a close box for closing it?

Value Class

boolean

Examples

```
set canBeClosed to the closeable of window whichWindow
```

Notes

➤ **`Closeable`** is always **`false`** if **`titled`** of the window is **`false`** or if the **`form`** of the window is **`modal dialog`**.

closing item

Indicates the window item that caused a window to be closed.

Value Class

record	
constant	close box or none

Examples

```
open window “My Dialog” returning {closing item:closer}
--Execution does not continue here until the modal dialog is closed:
if closer is close box then
    display dialog “closed by a close command without a PER parameter.”
else if closer is none then
    --This is not possible with a modal dialog.
else
    display dialog “closed per “ & name of closer & “.”
end if

open window “My Document” returning {closing item:closer}
--Execution continues while the window is open:
if closer is none then
    --This is the only possible value with a document or floating windoid.
end if
```

Notes

➤ The **`closing item`** of a currently open window is **`none`**.

➤ If a modal dialog is closed by the **`close`** command without a **`per`** parameter (which normally specifies the **`closing item`**), the **`closing item`** defaults to the constant **`close box`**.

➤ If the **`auto close`** property of a button is **`true`** and that button is used to close the window, **`closing item`** contains a description of the button. (For more information, see the **`description`** property common to all window items.)

➤ **`Closing item`** normally is used by the **`returning`** parameter of the **`open window`** statement that opens a modal dialog.

➤ See the discussion of the **`close`** message.

➤ **`Closing item`** is a read-only property.

Table of Contents Index

collapsed

Is the window collapsed?

Value Class

boolean

Examples

```
Set the collapsed of window “myWindow” to true
```

Notes

➤ The collapsable of the window must be true or setting the collapsed of the window will fail.

➤ This feature requires the Mac OS 8 Appearance Manager.

collapsable

Is the window collapsable?

Value Class

boolean

Examples

```
Get the collapsable of window “myWindow”
```

Note

➤ This feature requires the Mac OS 8 Appearance Manager.

contents

The value of the changes property of the window.

Value Class

list of record

Examples

```
copy the contents of window “Wow” to wdwContents
```

Note

➤ The value of the **`contents`** property of a window is the same as the value of the **`changes`** property. That value is also returned by the **`get data`** command.

contextual menu

This will allow you to set contextual menu information via a script.

Value Class

resource info

Examples

```
set contextual menu of window “myWindow” to {class:resource info, type:
“Menu”, name: Edit”, id:5002}
```

contextual help

This will allow you to enable and disable the help menu item of a contextual menu.

Value Class

boolean

Examples

```
set contextual help of window “myWindow” to true
```

description

Complete record of the names and values of the properties of the window and its window items.

Value Class

list of records	{{propertyName:value,…},
	{re:7001,propertyName:value,…},
	{re:7002,propertyName:value,…},…}

Table of Contents Index

Examples

```
copy the description of window “Untitled” to tempDescription
--Get the description of the window only:
set wdwDescription to item 1 of tempDescription
```

Notes

➤ The first record in the list describes the window itself. Each subsequent record refers to a window item, with the **`re`** property indicating the **`id`** of the window item.

➤Compare the **`changes`** property.

➤ **`Description`** is a read-only property.

droppable

Can the window have things dropped on it?

Value Class

boolean

Examples

```
if the droppable of window “My Editor” then
    display dialog “You can use drag & drop.” buttons {“OK”}
end if
```

Notes

➤ For the **`droppable`** property to be **`true`**, the system software must support drag and drop.

enabled

Is the window enabled?

Value Class

boolean

Examples

```
copy the enabled of myWindow to itsEditable
```

Notes

➤ The **`enabled`** property is **`true`** if the window is active (normal in appearance and responsive to user input) and **`false`** if it is inactive (dimmed and unresponsive to user input).

➤ **`Enabled`** is a read-only property.

fill color

The color of the window's background.

Value Class

RGB color	{redValue, greenValue, blueValue}
integer	index to color in System color lookup table
constant	black / white

Examples

```
copy the fill color of window “Paint box” to {theRed, theGreen, theBlue}
set the fill color of window “Food” to white
set the fill color of window “ColorByRGB” to {48059,48059,48059}
set the fill color of window “ColorByIndex” to 23
```

Notes

➤ The **`fill color`** is always returned as an RGB value, a list of three long integers, from 0 to 65535, representing red, green and blue intensities.

floating

Is the window a floating windoid (palette)?

Value Class

boolean

Examples

```
copy the floating of window “Soap” to itFloats
set the floating of window “Ivory Soap” to true --from Message Windoid
```

Table of Contents Index

Notes

➤ The **`form`**, **`floating`** and modal properties of a window can be changed only while using the Window Editor. They are read-only when the application is running.

➤ **`Floating`** and **`modal`** cannot both be **`true`** at the same time.

➤ If **`floating`** and **`modal`** are both **`false`**, the window is a document window.

➤ A script (in the Message Windoid) can set **`floating`** or **`modal`** to **`true`**, but not to **`false`**.

➤ If the **`form`**, **`floating`**, or **`modal`** property is changed, the others change to correspond.

focus

The window item (in this window) that is receiving keystrokes.

Value Class

reference

Examples

```
copy the focus of window “WhereIsTheSelection” to theFocusObj
--Get text from the object if it is a textbox:
if class of theFocusObj is textbox then
    copy contents of theFocusObj to myText
    copy the contents of the selection of theFocusObj to mySelectedText
end if
--Give the focus to a specific window item:
set the focus of window “Edit Me” to textbox 2 of window “Edit Me”
```

Notes

➤The **`focus`** of a window remains set even while the window is inactive.

➤ If you set the **`focus`** to a window item, that window item will get a **`focus received`** message. If the **`focus`** is taken from a textbox that has been edited, that textbox gets a **`changed`** message.

➤ Selected content in inactive windows can be located and manipulated using the **`focus`**, **`contents`**, and **`selection`** properties, as shown in the examples above.

➤ If the window contains no **`editable`** textboxes or **`key scrollable`** listboxes, the value of the **`focus`** property is **`none`**.

font

The default font of window items in the window.

Value Class

string	the name of the font

Examples

```
copy the font of window “Source Document” to itsFont
set the font of window of theObj to “Geneva”
set the font of window “Cubs” to “Chicago”
```

Notes

➤ If the **`font`** of a window item, such as a button, is not set, that item’s text is displayed in the **`font`** of the window.

➤ Changing the **`font`** property of a window changes the font of every window item for which the **`font`** property has not been explicitly set.

➤ Once the **`font`** of a window item is set, the window item can no longer inherit the window’s font.

form

The form of the window as defined by a form definition resource.

Value Class

constant	document window / modal dialog / floating windoid

Examples

```
copy the form of window “Mystery” to itsForm
if form of window 2   modal dialog then close window 2
```

Notes

➤ The **`form`** property of a window can be changed only while using the Window Editor. It is read-only when the application is running.

➤ The **`form`** of a window determines how the window looks and behaves.

➤ The **`form`** definition resources for windows (as opposed to those for window items) are predefined and cannot be removed.

➤ **`Document window`** is the default form.

➤ If the **`form`**, **`floating`**, or **`modal`** property is changed, the others change to correspond.

grow item

A single window item that resizes when the window is resized

Value Class

reference

Examples

```
copy the grow item of window “Document1” to theGrower
set the grow item of window “Food” to window item “Grow Me!”
```

Notes

➤ The **`grow item`** property is included for backward compatibility with FaceSpan 1.0. Any or all window items now can be made to grow (or move) using the **`growth`** property for window items.

➤ When a window is resized, window items above or to the left of the **`grow item`** remain fixed in their original positions, while window items below or to the right of the **`grow item`** move in order to maintain their positions relative to the bottom and right edges of the **`grow item`**.

➤ Window items aligned with the bottom or right edges of the **`grow item`** are resized along with the grow item.

➤ When constructing a window that is to contain a **`grow item`**, make the window the smallest size that the user will be permitted to shrink it, set the window’s **`min size`** property to those dimensions, then create the **`grow item`** and the rest of the window’s items.

height

The height of the content area of the window, measured in pixels.

Value Class

integer

Examples

```
copy the height of window “AmazingColossalDialog” to itsHeight
set the height of window “AmazingColossalDialog” to itsHeight + 100
```

Notes

➤ Window **`height`** measurements do not include title bars or frames.

➤ When the **`height`** changes, the **`bounds`** property also changes.

id

The current identification number of the window.

Value Class

integer

Examples

```
copy the id of window “Source Document” to itsID
set the name of window “Source Document” to somethingFrench
--Use itsID to reference the window in the rest of the script.
```

Notes

➤ Each window receives an **`id`** when it is opened, and that **`id`** does not change while the window remains open; however, it may have a different **`id`** each time it is opened.

➤ A window must be opened by name, since its **`id`** does not exist until it is opened.

➤ **`Id`** is a read-only property.

idle delay

The frequency with which the window receives **`idle`** messages.

Value Class

integer

Examples

```
copy the idle delay of window “wired” to itsDelay
set the idle delay of window “toasted” to 0
```

Notes

➤ The **`idle delay`** value is expressed in whole seconds.

➤ If a window and its application have different **`idle delay`** values, the **`idle delay`** of the window is ignored unless it is greater than that of the application.

➤ If the **`idle delay`** is 0, the window receives **`idle`** messages as often as possible.

➤ Your scripts can send **`idle`** messages; this will allow user interaction and window updates to occur during time-consuming processing. See the description of the **`idle`** event later in this chapter.

index

The position of an open window in the front-to-back ordering of all open windows.

Value Class

integer

Examples

```
copy the index of window “WhichWindowIsThis” to itsIndex
```

Notes

➤ The front most window has an **`index`** of 1.

➤ The commands **`send to back`** and **`bring to front`**, as well as opening and closing other windows, can change the value of the **`index`**.

➤ The **`index`** property is read-only.

max size

The largest size to which the window can be resized or zoomed.

Value Class

point	{maximumWidth, maximumHeight}

Examples

```
copy the max size of window 1 to {maxWidth, maxHeight}
set the max size of window 1 to {maxWidth, maxHeight+100}
```

Notes

➤ The **`max size`** property is given in pixels.

min size

The smallest size to which the window can be resized or zoomed.

Value Class

point	{minimumWidth, minimumHeight}

Examples

```
copy the min size of window “Prolog” to {minWidth, minHeight}
set the min size of window “Prolog” to {minWidth, minHeight+100}
```

Notes

➤ The **`min size`** property is given in pixels.

➤ Be sure to set the **`min size`** of any resizable window to dimensions that maintain its visual integrity.

modal

Is the window a modal dialog?

Value Class

boolean

Examples

```
copy the modal of window “Mystery Window” to itsModal
```

Notes

➤ The **`form`**, **`floating`** and **`modal`** properties of a window can be changed only while using the Window Editor. They are read-only when the application is running.

➤ **`Floating`** and **`modal`** cannot both be **`true`** at the same time.

➤ If both **`floating`** and **`modal`** are **`false`**, the window is a document window.

➤ A script (in the Message Windoid during editing) can set **`floating`** or **`modal`** to **`true`**, but not to **`false`**.

➤ If the **`form`**, **`floating`**, or **`modal`** property is changed, the others change to correspond.

name

The name of the window. A window now has a **`name`** property that is distinct from its **`title`** property. The **`name`** property is the name by which you may refer to the window. The **`title`** is what appears in the title bar of the window.

Value Class

string

Examples

```
copy the name of window of theObj to objWindow
--Keep the id for future reference, then rename the window:
copy the id of window objWindow to objWdwID
set the name of window id objWdwID to “To Paráthuro”
```

Notes

➤ To change a window's **`name`** without losing your reference to the window, obtain its **`id`** when you open it, then use the **`id`** as the reference until it is closed.

➤ If a script changes the **`name`** of a window in a running project or application, then issues the **`save`** command, a new template by that name will be saved into the application.

➤ For compatibility, the name property stored for windows in existing projects will be used as both the **`name`** and the **`title`**. Please note, however, that this change may cause some projects to "break."

➤ If a script had been changing a window's title by setting its **`name`** property, the title will not change as expected. Conversely, if a script had been changing a window's name by setting its **`title`** property, subsequent references to the window by its new name will fail, because only its title will have changed.

pen color

The color of the window's foreground, usually its outline and title.

Value Class

RGB color	{redValue, greenValue, blueValue}
integer	index to color in System color lookup table
constant	black / white

Examples

```
copy the pen color of window "Paint box" to itsPen
set the pen color of window "Paint box" to black
set the pen color of window "ColorByRGB" to {48059,48059,48059}
set the pen color of window "ColorByIndex" to 48
```

Note

➤ The **`pen color`** property is always returned as an RGB value, a list of three long integers, from 0 to 65535, representing red, green and blue intensities.

position

The position of the content area of the window on the screen.

Value Class

point	{leftOffset, topOffset}
constant	centered in main
	centered in current
	centered in deepest
	staggered in main
	staggered in current
	staggered in deepest
	offscreen

Examples

```
copy the position of window “WhereIsIt” to {wdwH, wdwV}
set the position of window “WhereIsIt” to {wdwH+100, wdwV+100}
set the position of window “Stealth” to offscreen
set the position of window “centerme” to centered in main
```

Notes

➤ The **`position`** property is always returned as a point; note the order of the coordinates.

➤ When expressed as a point, the **`position`** is the location of the top-left corner of the content area of the window as measured from the top-left corner of the screen.

➤ When expressed as a constant:

- ♦ **`main`** means the screen that contains the menu bar.
- ♦ **`current`** means the screen that has the focus.
- ♦ **`deepest`** means the screen with the largest color depth.
- ♦ **`staggered`** arranges windows in a cascade from the upper left corner of the monitor to the lower right corner.

Each **`staggered`** window is positioned 16 pixels below and 16 pixels to the right of the window behind it. It is useful when opening several windows in succession, from a single script.

private menus

Menus displayed on the menu bar while the window is active.

Value Class

list of strings	{“menuName”, “menuName”, …}

Examples

```
copy the private menus of window “Restaurant” to itsMenus
if “Wines” is in itsMenus then display dialog “Wine?” buttons {“Never”}
```

Notes

➤ A window’s **`private menus`** property does not include the names of menus used by the project as a whole—that is, the menus above the dividing line in the Menus View of the Project Window.

➤ Any menu can be among the **`private menus`** of more than one window, since only one window can be active at a time.

re

A reference to a window item in a description record.

Value Class

integer	index or id
string	name

Examples

```
open window “Info” with properties {re:3, height:20}
open window “Tom” with properties {re:7890, width:30}
open window “Analysis” with properties {re:”balance”, contents:”0”}
set theItem to 7001 --could also be index or name
open window “Stealth” with properties {re:theItem, title:”Wow”}
open window “User Info” returning ¬
   {{height:h, position:{x,y}}, {re:12, visible:isVisible, enabled:isEnabled}}
```

Notes

➤ The **`re`** property is a component of records returned by the **`changes`**, **`description`** and **`closing item`** properties.

➤ The **`re`** property is a component of records accepted by the **`with properties`** and **`returning properties`** parameters of the **`open window`** command, and of the **`with properties`** parameter of the **`make`** command.

➤ When you give a value for the **`re`** property, you may use the value of the **`index`**, **`id`**, or **`name`** property of a window item.

➤ **`Re`** is always returned with the value of the **`id`** property (an integer greater than 7000), since the **`id`** of a window item never changes.

➤ For more information, see the **`closing item`**, **`description`**, and **`changes`** properties of windows, and the **`make`** command.

➤ If you give an **`re`** value for a window item that does not exist, no error results; the record containing that **`re`** is ignored.

resizable

Does the window have a size box for changing its size?

Value Class

boolean

Examples

```
copy the resizable of window “MyDocument” to canResize
set the resizable of window “Keep Same Size” to false
```

Note

➤ **`Resizable`** is always **`false`** if the **`titled`** property of the window is **`false`** or if the **`modal`** property of the window is **`true`**.

script

The compiled script of the window.

Value Class

script

Examples

```
copy (the script of window “Edit Text”) as text to textbox “txtScript”
set the script of window “Test” to contents of textbox “txtScriptTester”
```

Notes

➤ When the text of a script is assigned to the **`script`** property, it is automatically compiled.

➤ An error occurs if the text of the assigned script cannot be compiled. You must handle such an error with a **`try`** statement.

➤ As soon as a **`script`** has been successfully assigned, it can be executed.

size

The default point size of text in window items in the window.

Value Class

integer

Examples

```
copy the size of window “Source Document” to textHeight
set the size of window of theObj to 12
```

Notes

➤ If the **`size`** property of a window item, such as a button, was not explicitly set, its text is displayed in the font size of the window.

➤ Changing the **`size`** property of a window changes the font size of every window item for which the **`size`** property has not been explicitly set.

➤ Once the **`size`** property of a window item has been set, it no longer is affected by changes to the window's **`size`** property.

style

The default style of text in window items in the window.

Value Class

text style info

Examples

```
copy the style of window “Source Document” to wdwStyle
set the style of window “Watcher” to ¬
   {on styles:{underline}, off styles:{bold, italic, outline, shadow, group}}
```

Notes

➤ If the **`style`** property of a window item, such as a button, was not explicitly set, its text is displayed in the style of the window.

➤ Changing the **`style`** property of a window changes the style of every window item for which the **`style`** property has not been explicitly set.

➤ Once the **`style`** property of a window item has been set, it no longer is affected by changes to the window's **`style`** property.

➤ For information about the **`text style info`** class, see Chapter 15: “Special Artwork and Text Style Classes.”

text to speech

Text to Speech works with text boxes.

Value Class

String

Example

```
On hilited theObi
   say “this is a test” using voice “Kathy”
end hilited

“set foo” to every file of folder “Voices” of extension folder
```

Notes

➤ If the system has the appropriate “Text to Speech” software installed, the above command will use the voice “Kathy” to speak the text.

➤ You can store a list of voices by setting a variable to every file of folder “Voices” of the extension folder. See the second command in the above example.

title

A window now has a **`title`** property that is distinct from its **`name`** property. The **`title`** property is what appears in the title bar of the window. The **`name`** property is the name by which you may refer to the window. The title property now appears in the Object Info dialog for windows in the Property Bar’s “Properties” pop-up menu.

Value Class

string

Notes

➤ For compatibility, the name property stored for windows in existing projects will be used as both the **`name`** and the **`title`**. Please note, however, that this change may cause some projects to “break.”

➤ If a script had been changing a window’s title by setting its **`name`** property, the title will not change as expected. Conversely, if a script had been changing a window’s name by setting its **`title`** property, subsequent references to the window by its new name will fail, because only its title will have changed.

Table of Contents Index

titled

Does the window have a title bar?

Value Class

boolean

Examples

```
copy the titled of window “MyDocument” to isTitled
set titled of window “Drag Me Around” to true
```

Notes

➤ A document window cannot be **`closeable`**, **`resizable`**, or **`zoomable`** unless its **`titled`** property is **`true`**.

➤ A modal dialog is not movable unless its **`titled`** property is **`true`**.

uniform styles

The default text styles that apply to the window items in the window.

Value Class

text style info

Examples

```
copy the uniform styles of window “Source Document” to itsUStyles
set the uniform styles of window 1 to ¬
   {on styles:{underline}, off styles:{bold, italic, outline, shadow, group}}
set the uniform styles of window “Chameleon” to itsUStyles
```

Notes

➤ If the **`uniform styles`** property of a window item, such as a button, was not explicitly set, its text is displayed in the style of the window.

➤ Changing the **`uniform styles`** property of a window changes the uniform styles of every window item for which the **`uniform styles`** property has not been explicitly set.

➤ Once the **`uniform styles`** property of a window item has been set, it no longer is affected by changes to the window's **`uniform styles`** property.

➤ For information about the **`text style info`** class, see Chapter 15: "Special Artwork and Text Style Classes."

visible

Is the window visible?

Value Class

boolean

Examples

```
copy the visible of window "May Be Hidden" to canSeeIt
set the visible of window "Stealth" to false
```

Notes

➤ Even an invisible window has a valid **`index`** and all other window properties; it is still open.

➤ Another way to make a window "invisible" is simply to set its **`position`** to **`offscreen`**.

➤ To find out whether or not a window is open, use **`if exists window`** "X".

width

The width, in pixels, of the content area of the window.

Value Class

integer

Examples

```
copy the width of window "Girth of Earth" to itsWidth
set the width of window "He's not here" to the width of movie "Elvis"
```

Notes

➤ The **`width`** of a modal dialog does not include its frame.

➤ Changing the **`width`** of a window changes its **`bounds`**, too.

zoomable

Does the window have a zoom box?

Value Class

boolean

Examples

```
copy the zoomable of window “MyDocument” to itsCanZoom
set the zoomable of window “No Zooming” to false
```

Note

➤ **`Zoomable`** is always **`false`** if **`titled`** of the window is **`false`**.

zoomed

Is the window zoomed to its maximum size?

Value Class

boolean

Examples

```
copy the zoomed of window “Lens” to itIsZoomed
set zoomed of window “Lookout!” to true
```

Note

➤ When the **`zoomed`** property is set by a script, the window receives a **`resized`** message and a **`moved`** message, but not a **`zoom out`** message. The window also zooms out.

Window Command and Event Messages

Because windows contain the window items, any message sent to a window item can be intercepted and handled by its window if the window item chooses to ignore or to continue the message.

Listed here are a few additional command and event messages that are sent directly to windows. The listing tells the source of the message—either a command issued from a script, or an event from the system or from user interaction.

Any message sent by an event can also be sent by a command; the name of the message is the name of the command.

activated

Event message sent when the window has become active.

Parameters

(direct)	reference	the activated window

Examples

```
--Redraw a normally hidden item after a modal dialog closes (see notes):
on activated theObj
   tell picture “picLastView” to draw
end activated
```

Notes

➤ A window becomes active when it is brought to the front of other windows in the application, when it is opened, and when a modal dialog in front of it is closed.

➤ A window becomes active also when it is the front window of the application, and the application becomes front most.

➤ The active window is distinguished by the active appearance of its title bar, scroll bars and buttons.

➤ If you are using the **`draw`** command to selectively display invisible window items (such as when animating a series of pictures), the image is lost when another window comes to the front, then closes. Use the **`activated`** message to detect when you need to redraw the image.

bring to front

Command to bring the window to the front of the layering.

Parameters

direct)	reference	the window to be brought to the front

Examples

```
bring to front window “Hieroglyphic”
```

Notes

➤ The **`index`** of the window becomes 1; the indices of all open windows that were in front of this one also change.

➤ Any handler for this message must continue the message or the window will not be brought to the front.

chosen

Event message sent when the user chooses a menu item from a menu.

Parameter

(direct)	reference	the chosen menu item

Examples

```
--In any script, we can simulate choosing a menu item:
chosen menu item “Open” of menu “File”

--Outline of a typical chosen handler:
on chosen theObj
   --Get the title of the menu, and the index of the item::
   copy the title of menu of theObj to theMenu
   copy the index of theObj to theMenuItem

if theMenu is myPrivateMenu then --see if it should be handled here
   -- Handle messages from menu “Templates”
   if theMenuItem is 1 then
      --handle first menu command here
   else if theMenuItem is 2 then
      --handle second menu command here
      ...etc.
      else
        --handle last menu command here
      end if
   else
      continue chosen theObj -- Let the application handle it
   end if
end chosen
```

Notes

➤ The *chosen* message is sent to the front window if any windows are open; otherwise, it is sent to the application.

➤ The window should handle the **`chosen`** message if it has private menus. If the message is not for its own menu, **`continue`** the **`chosen`** message so that the project script can handle it.

➤ See Chapter 10: “Scripting Your Application,” for more discussion of the **`chosen`** message.

Table of Contents Index

click

Event message sent when the application user clicks anywhere in the window.

Parameters

(direct)	reference	the window
at	point	coordinates of the mouse
upon	constant	part of window
[option down]	boolean	Option key is down
[shift down]	boolean	Shift key is down
[command down]	boolean	Command key is down
[control down]	boolean	Control key is down
[ticks]	long integer	time at which click occurred

Examples

```
--A click handler that disallows clicks in the content area:
on click theObj at thePos upon thePart
   if thePart is none then
      beep 1
   else
      continue click theObj at thePos upon thePart
   end if
end click
```

Notes

➤ The coordinates of the mouse are relative to the window; the top-left corner of the content area is at {0, 0}.

➤ The constants for the **`upon`** parameter specify the various parts of a window; they are: **`close box`**, **`zoom box`**, **`size box`**, **`title bar`**, **`menu bar`** and **`none`**. The content area is given as **`none`**.

➤ The **`ticks`** parameter is given as 60ths of a second since system startup; it is used to calculate elapsed time, not actual clock time.

➤ Any handler for the **`click`** message must **`continue`** the message or normal interaction with the window will not occur.

➤ If the **`upon`** parameter is not accepted and continued, normal window operations, such as closing and resizing, will be inhibited.

➤ If a **`click`** handler in the window needs a certain parameter, then any **`click`** handlers in the window *items* must accept and continue that same parameter.

close

Event message sent as the window is closed.

Parameters

(direct)	reference	the window to be closed
per	constant	close box
	or reference	the object that closed the window, usually a button

Examples

```
on close theWindow per theClosingItem
   if theClosingItem is close box then
      display dialog “Closed with the close box.”
   else
      display dialog “Closed per “ & (name of theClosingItem)
   end if
   continue close theWindow per theClosingItem
end close
```

Notes

➤ If a window of a running application is closed as the result of a system shut-down, the **`per`** parameter is a reference to the window itself.

➤ If a window of a running project is closed by clicking the Stop button or by quitting FaceSpan, the **`per`** parameter is a reference to the window itself.

➤ If a window of a running project is closed by a **`close`** command in a script, that command specifies the object reference passed in the **`per`** parameter.

➤ Any handler for this message must continue the message or the window will not close; that might, in fact, be what you wish.

deactivated

Event message sent to the active window when it becomes inactive.

Parameters

(direct)	reference	the window that was deactivated

Notes

➤ The front window is deactivated when another window is brought to the front, when a modal dialog opens, or when another application is brought to the front.

➤ The **`deactivated`** message is not sent to a window when it is closed.

delete

Command to delete the specified window item, menu item or listbox item.

Parameters

(direct)	reference	the window item to delete

Examples

```
tell window “Whoops” to delete window item 2
delete menu item 1 of menu “Templates” of window “Other”
delete listbox item n of listbox “Registry”
```

Notes

➤ The **`delete`** command cannot delete a window item from the window that contains the handler issuing the command.

➤ **`Delete`** is the inverse of the **`make`** command.

draw

Command to redraw the window and all window items.

Parameters

(direct)	reference	the window

Examples

```
draw window “Preferences”
tell window item 3 to draw
```

Notes

➤ When a handler makes visible changes to a window or its items, the changes are normally not shown until the handler concludes. To display the changes as soon as they are made, send **`draw`** commands from the handler to the window or window items that were changed.

➤ If you tell the window to **`draw`**, all the window items will also be drawn, in index order (back to front).

➤ See the **`idle`** message discussion for an example of the **`idle`** command, a general way to get a window to redraw itself.

get data

Command to get some data from the window.

Parameters

(direct)	reference	the window
[as]	type class	the type class wanted

Examples

```
get data of window “Preferences”
```

Notes

➤ The value returned by **`get data`** is identical to the value of the **`changes`** property of the window.

➤ Use the **`as`** parameter to coerce the default value into another type class.

➤ The **`get data`** command is included in FaceSpan for the sake of completeness; it is seldom necessary in actual practice.

idle

Event message sent by the application when no other events are occurring.

Parameters

(direct)	reference	the window

Examples

```
--Example loop for a time-consuming process.
repeat with i from 1 to theCount
   --(Put the next step in the process here. See the notes.)
   idle
end repeat

--Idle message handler:
on idle theObj
   --Beep every 10 seconds
   global lastTicks
   set ticksNow --using the Ticks scripting addition
   if ticksNow - lastTicks > 600 then
      beep 1
      set lastTicks to ticksNow
   end if
end repeat
```

Notes

➤ The **`idle`** message is sent to the window by the application once every two seconds, by default, or at the interval specified by the **`idle delay`** property.

➤ If a window and its application have different **`idle delay`** values, the **`idle delay`** of the window is ignored unless it is greater than that of the application.

➤ As shown in the first example, the **`idle`** message can be sent as a command from within time-consuming loops to allow window updates and user interaction. *Important:* Since the **`idle`** command allows user interaction, make sure that the user cannot restart this script while it is running! For example, if the script is in the **`hilited`** handler of a button, the script should disable the button before entering the loop.

make

Command to create a new object.

Parameters

[new]		
(direct)	Objectclass	object to make
at	reference	container or location in a container
[with properties]	record or list of record	properties to be assigned

Examples

```
make menu item with properties {name:”Overview”, mark:”•”} ¬
   at beginning of menu “Views”
make textbox with properties {class:textbox, ¬
   bounds:{0,0,100,100}, contents:”Voila!”, ¬
   editable:true, position:{99,8}) at end of window 1
make new window with properties ¬
   {{bounds:{77, 101, 475, 301}, form:modal dialog, ¬
   titled:true, zoomable:false, private menus:{}, ¬
   name:”Bob”}, {class:push button, ¬
   bounds:{195, 151, 275, 171}, name:”cancel”, ¬
   title:”Cancel”}, {class:push button, ¬
   bounds:{282, 151, 362, 171}, name:”ok”, title:”OK”}}
make storage item with properties {name:”TJ”, contents:”Thomas Jefferson”}
```

Notes

➤ A new window can be made at any time, from any script; this does not create a new window template. If a script saves the window with the **`save`** command, a new template will be saved into the application; it can later be edited with the Window Editor.

➤ A window cannot make a new window item within itself, nor can a window item make a new item within its own window.

➤ When making a window item, the **`at`** parameter, if used, specifies the position of the new item in the layering; **`beginning`** means the bottom layer, while **`end`** means the top layer.

➤ When making a window item, the **`with properties`** parameter is required, and must include the **`name`** and **`bounds`** properties.

➤ New listbox items can be made in any listbox.

➤ New menu items can be made in any menu currently displayed; the menu template is not changed.

➤ New menu items can be made in any popup; some menu item properties do not apply to menu items in popups.

➤ New storage items can be made at any time from any script; they are persistent from run to run.

moved

Event message sent when the window is moved by use of its title bar or by zooming.

Parameters

(direct)	reference	the moved window

Examples

```
on moved theObj
    --Make the window snap to an imaginary 16-pixel grid:
    copy the position to {theLeft, theTop}
    set theLeft to (theLeft div 16) * 16
    set theTop to (theTop div 16) * 16
    copy {theLeft, theTop} to the position
end moved
```

Notes

➤ Changing a window’s **`position`** property does not cause a **`moved`** message.

open

Command to open a window.

Parameters

(direct)	reference	the window to open
[with properties]	record or list of record	values to override
[returning properties]	record or list of record	record(s) to return
[returning]	record or list of record	values to assign

Examples

```
open window “W” returning properties {closing item: 0} --see notes
open window “W” returning {closing item:theCloser} --see notes

open window theDialog ¬
   with properties {re:”txtMessage”, contents:”Enter your name:”} ¬
      returning {re: “txtName”, contents: theName}
display dialog “Your name is “ & theName
```

Notes

➤ The optional **`returning properties`** parameter or returning parameter is used with modal dialogs to retrieve information changed by the application user.

➤ An **`open window`** command with a **`returning properties`** parameter returns (in **`the result`**) a record structured just like the parameter's record, except that the real values are filled in. The values you give are merely place holders.

➤ An **`open window`** command with a **`returning`** parameter returns values of specified items directly in the variable names you give in the parameter.

➤ See the discussions of the **`changes`** and **`closing item`** properties and especially the **`re`** property for more examples of the **`open`** command.

➤ Use a single record to set or get the properties of the window or of a single window item. Use a list of records to set or get the properties of two or more objects.

➤ To find out whether or not a window is open, use **`if exists window`** "X".

➤ See Chapter 10: "Scripting Your Application," for a detailed discussion of the **`open`** command and its parameters.

prepare

Event message sent just before a window is opened.

Parameters

(direct)	reference	the window to be prepared

Examples

```
on prepare theObj
   --Add this window's name to the Windows menu:
   copy name of theObj to windowName
   copy (((count menu items of menu "Windows") + 1) ¬
      as string) to menuIndex
   make new menu item at end of menu "Windows" with ¬
      properties {name:windowName, ¬
      command key:(menuIndex as string), ¬
      enabled:true, checked:false}
end prepare
```

Note

➤ The **`prepare`** message is sent to a window after any changes caused by the **`with properties`** parameter of the **`open`** statement have been applied. Thus, everything is in place, but the window is not yet visible.

resized

Event message sent when the window is resized or zoomed by the user.

Parameters

(direct)	reference	the resized window

Examples

```
on resized theObj
   --Make the window's width & height snap to an imaginary 16-pixel grid:
   copy the bounds to {theLeft, theTop, theRight, theBottom}
   set theRight to (theRight div 16) * 16
   set theBottom to (theBottom div 16) * 16
   copy {theLeft, theTop, theRight, theBottom} to the bounds
end resized
```

Notes

➤ A **`resized`** message is not sent when the **`bounds`** property of the window is changed by a script.

send to back

Command to send the window to the back of the layering.

Parameters

(direct)	reference	the window to be moved

Examples

```
send to back window “Hieroglyphic”
```

Notes

➤ The indices of all open windows behind this one are incremented when this one is sent to the back.

➤ Any handler for this command must continue the command or the window will not be sent to the back.

sound

You can play and record sounds as either “snd” resources or as “AIFF” files. You can even make the computer speak with Text to Speech. See *text to speech* on page 251.

Example

```
record audio “new snd name” -- record a ’snd’ resource
record audio into file “Macintosh HD:New AIFF” -- record an AIFF file

play audio “new snd name” -- play a ’snd’ resource
play audio from file “Macintosh HD:New AIFF” -- play an AIFF file
```

Notes

➤ The new play commands simply initiate the sounds and return immediately. Each command returns a reference to the ongoing sound so that the sound may be paused, resumed or stopped. The existence (using **`if exists`** ...) of the reference may be used to determine if the sound is still playing.

➤ To import a “snd” file, select the “Forms, etc.” view in the project window. Click the Import button, navigate to the “snd” file and select it. To get information on the sound or to test the sound from the project window, double-click or click the open button when the sound is selected. Click the speaker icon in the upper-right corner of the resulting dialog window to hear the sound. To play the imported sound in a script, use the syntax: play audio “The snd Name.”

zoom in

Event message sent when the window is to zoom in.

Parameters

(direct)	reference	the window to be zoomed

Notes

➤ The default response to the **`zoom in`** message is to zoom the window back its minimum allowable size, or to the position and size last set by the user.

➤ The zoomed-in size of a window cannot be less than the value of the **`min size`** property.

➤ Any handler for this message must continue the message or the window will not zoom in. In addition, the window will not be prepared to zoom out.

zoom out

Event message sent when the window is to zoom out.

Parameters

(direct)	reference	the window to be zoomed

Notes

➤ The default response to the **`zoom out`** message is to zoom the window out to its maximum allowable size.

➤ The zoomed-out size of a window cannot be greater than the value of the **`max size`** property.

➤ Any handler for this message must continue the message or the window will not zoom out. In addition, the window will not be prepared to zoom in.

Table of Contents Index

Special Considerations

There are several other features of windows that you can use to your advantage.

Scripts to Edit Windows

When a window template is open for editing in FaceSpan, an application can send messages to FaceSpan itself to augment the construction of the window template.

First, a script can get information about the window and window items being edited:

➤ If a window is being edited, it is **`window 1`**.

➤ The number of window items in the widow can be obtained as the **`count of window items of window 1`**.

➤ The indices of the currently-selected window items are in the **`selection of window 1`**.

Given this information, you can get and set any property of the window or of any window item.

In addition, you can use the make and delete commands to add window items or to remove them. So, for example, you could write a script that sets the bounds of the window to a standard modal dialog size, makes a textbox for a message, and makes an OK button, thereby turning the window into a standard modal dialog.

Every, Whose and Where

You can use the standard AppleScript selection terms—the **`every`**, **`where`** and **`whose`** clauses—to refer to window items. This statement could be in the script of a running application, or it could be executed from the Message Windoid while editing:

```
set the enabled of every push button of window 1 to false
```

Window items can be selected by property values as well, using a **`where`** or **`whose`** clause:

```
set enabled of (every window item of window 1 whose enabled is false) to true
```

Restrictions on the Quit Command

If a handler in a window executes an unqualified **`quit`** command, the default target of the command is the window itself; this is not permissible.

The window's script should have **`quit current application`** instead, or a **`quit`** command could be sent to the window from the project or from another window.

Opening Several Copies of a Window

You can open several windows based upon the same window template resource. If you do, either change the **`name`** of each copy, or refer to the windows by their **`index`** or **`id`** properties.

Chapter 13:

Window Items

Contents:

Table of Contents Index

Window Items

Window items are interface objects—buttons, labels, textboxes, pictboxes and others—that form the working parts of windows. Each window can contain many window items.

A window item is defined and controlled by a set of properties, and is responsive to various messages sent by commands and interface events.

Some properties and messages are common to all window items; these are described first. The properties and messages that are unique to each class of window item are described under the headings for the window items.

Reference Forms

A window item can be referenced by its **`name`**, **`id`** or **`index`** property, or by index relative to its peers. For example:

➤ push button “pshOK”

➤ push button 1

➤ push button id 7003

➤ window item 3

➤ window item “pshOK”

Note that when you refer to a window item by index, there are two ways you can do so. You can refer to the object’s **`index`** property, which is its index relative to all window items. An example is **`window item 3`**. Or you can refer to the index of the object in relation to other objects of the same class. An example is **`push button 1`**, which refers to the push button having the lowest **`index`** number of all push buttons in the window.

To obtain a reference to the window that contains the window item, use:

    window of theObj

where **`theObj`** is a reference to the window item.

A reference to a window item is implemented in terms of its index. Although this implementation seldom has surprising consequences, you should be aware of the possibilities.

Consider these three statements:

```
on hilited theObj --a push button, window item 3
   set the index of theObj to 4
   set the enabled of theObj to false
end hilited
```

The reference **`theObj`** really is referring to window item 3. The first **`set`** statement makes the button become window item 4. The next **`set`** statement sets a property of the *new* window item 3. This consequence is easy to avoid: do not change an item’s index in one of its own handlers.

Common Properties

These properties are common to most or all window items; any exceptions are listed. All window items have additional, unique properties, which are discussed in the sections for the individual items.

balloon

Text of a message or name of a picture resource to be displayed in a window item’s help balloon.

Value Class

string

Examples

```
set the balloon of theObj to “What are YOU looking at?”
set the balloon of listbox “Poisons” to “Skull and Crossbones Picture”
```

Notes

➤ The balloon can be made to display a picture instead of text. Simply enter the name of a picture that you have imported into your project.

➤ If you want the balloon to display text that is the same as the name of a picture, prefix the text with a space.

bounds

Offsets of the four sides of a window item.

Value Class

list of integer	bounding rectangle	{left,top,right,bottom}

Examples

```
copy the bounds of theObj to itsBounds
copy the bounds of textbox 3 to {boxLeft, boxTop, boxRight, boxBottom}
set the bounds of textbox 3 to {boxLeft, boxTop, boxRight+16, boxBottom+12}
```

Notes

➤ The offsets are measured in pixels from the top-left corner of the window containing the window item.

➤ Changing a window item's **`bounds`** also changes its **`height`**, **`width`** and **`position`**.

changes

The properties of the window item that can be changed by an application user.

Value Class

record	{re:objectId,propertyName:propertyValue…}

Examples

```
copy the changes of textbox 1 to whatsChanged
```

Notes

➤ In the example shown, the **`changes`** property would be a record similar to: {re:7004, contents:"I typed this in the textbox."}.

➤ The **`changes`** record in no way implies that the **`contents`** have actually been changed.

➤ If a window item has no properties editable by the application user, its **`changes`** property is a record containing only the **`re`** property.

➤ The **`changes`** property exists simply by analogy to the window's **`changes`** property; it can be used to restore values to a window item when the window is reopened. See the discussion of the **`open`** command in Chapter 12: "Windows".

➤ **`Changes`** is a read-only property.

class

Object class of the window item.

Value Class

class	push button / radio button / checkbox / table / textbox / pictbox / icon / movie / listbox / popup / gauge / table / box / graphic line/ bevel button/ disclosure triangle/ tab panels/ slider/ progress bar/ up-down arrows/ spinning arrows

Examples

```
copy class of theObj to theClass
if theClass is push button then copy name of theObj to theName
```

Notes

➤ The standard data types in AppleScript have classes as well; they are: **`integer`**, **`real`**, **`boolean`**, **`string (`**or **`text)`**, **`list`**, **`record`** and **`alias`**. Note in particular that **`list`** and **`listbox`** are distinct, as are **`text`** and **`textbox`**.

➤ **`Class`** is a read-only property.

contents

The value of the “typical” property of the window item.

Value Class

boolean	hilite (false) of a push button
	hilite of a radio button
	hilite of a checkbox
integer	setting of a gauge
resource info	artwork of an icon
	artwork of a pictbox
alias	artwork of a movie
	contents of a textbox
	artwork of a pictbox

Table of Contents Index

string	title of a label
	contents of a textbox
	contents of selection of a textbox
	title of a box
	title (“") of a graphic line
	contents or name of a listbox item
	contents or name of a menu item
list of string	contents of a listbox
	contents of selection of a listbox
	contents of a popup
	contents of selection of a popup
	contents of a menu
	contents of a table row
	contents of a table column
list of list of	contents of a table
	contents of selection of a table

Examples

```
copy the contents of theObj to objContents
```

Notes

➤ In general, if you wish to obtain and operate upon the **contents**, you must know its value class, since each is limited to certain operations.

➤ See the discussions of **contents** in the detailed property descriptions of the various window items later in this chapter.

➤ When the **form** of a listbox, popup or menu is not standard, the entries that must be typed into it might be textual descriptions or references to the actual displayed **contents**. See the section, “Resources and Key Filters”, below.

➤ When a textbox or table cell has a **key filter** assigned to it, the **key filter** can determine the class of values that can be assigned to or retrieved from the textbox or cell.

➤ If the window item displays its **contents**, the default **contents** is the item’s class name.

➤ The **contents** of a window is the same as its **changes** property.

description

A record of properties describing the window item.

Value Class

record	{propertyName:value,propertyName:value,…}

Examples

```
set itsDescr to the description of theObj
copy description of push button 3 of window “Originals” to mold
make push button “Clone” at end of window “different” with properties mold
```

Notes

➤ The **`description`** does not include some properties if they have the default values. Therefore, its main use, as in the example, is to duplicate a window item.

➤ **`Description`** is a read-only property.

drag locked

Can the window item be dragged during editing?

Value Class

boolean

Notes

➤ If the **`drag locked`** property is **`true`**, the window item cannot be moved while editing the window.

➤ The **`drag locked`** property applies while using the Window Editor, not while running the application.

➤ When the cross-hair cursor passes over a **`drag locked`** item, it does not change to the mover cursor.

➤ **`Drag locked`** is **`false`** when a window item is created.

draggable

Can the item be dragged?

Value Class

boolean

Examples

```
set the draggable of theObj to true
```

Notes

➤ See the discussions of the **`drag`** and **`drop`** messages for information about using this feature.

➤ **`Draggable`** is **`false`** by default.

droppable

Can the item have things dropped onto it?

Value Class

boolean

Examples

```
set the droppable of theObj to false
```

Notes

➤ To be **`droppable`**, a window item must be visible.

➤ A box must have a fill, or else only its border can receive a **`drop`**.

➤ See the discussions of the **`drag`** and **`drop`** messages for information about using this feature.

➤ **`Droppable`** is **`false`** by default.

enabled

Is the window item responsive to user interactions?

Value Class

boolean

Examples

```
get enabled of theObj
set enabled of push button “Save” of window “Untitled” to false
```

Notes

➤ **`Enabled`** window items are normal in appearance and responsive to user interactions, while inactive items are dimmed and unresponsive to user interactions.

➤ If **`enabled`** of a textbox is **`false`**, its **`contents`** cannot be selected and it is omitted from the tabbing order.

➤ By default, **`enabled`** is **`true`**.

fill color

Color of the window item’s background.

Value Class

RGB color	{redValue, greenValue, blueValue}
integer	index to color in System color lookup table
constant	black / white

Examples

```
copy the fill color of theObj to itsFill
set fill color of push button “BlackedOut” to black
set fill color of box “grayFilled” to {32639,32639,32639}
set fill color of box “grayFilled” to 87
```

Notes

➤ The **`fill color`** property is always returned as an RGB value, a list of three long integers, from 0 to 65535, representing red, green and blue intensities.

➤ The specified **`fill color`** replaces all the white areas of pen and fill patterns, as shown in the **`pen pattern`** and **`fill pattern`** palettes.

➤ The **`fill color`** property defaults to **`white`**.

font

The font in which a window item's title or contents text is displayed.

Value Class

string

Examples

```
set font of box 2 of window “Fonts” to “Chicago”
copy the font of theObj to itsFont
set boxFont to font of box “boxLabeled” of window 2
set the font of selection of textbox “txtQuery” to “Geneva”
copy the font of the selection of textbox 3 to itsFont
```

Notes

➤ The value of the **`font`** property is the font's name.

➤ If the **`font`** of a window item, such as a button, was not explicitly set, its text is displayed in the **`font`** of the window.

➤ Changing the **`font`** property of a window changes the **`font`** of every window item for which the **`font`** property has not been explicitly set.

➤ To refer to the **`font`** of a selection within a textbox, pop-up menu, or listbox, use a **`selection`** reference, as shown in the examples.

➤ The **`font`** defaults to “Chicago.”

growth

How the window item's bounds respond to resizing its window.

Value Class

constant	none / adjust height / adjust width / adjust both / move across / move down / move both
list of integer	see notes

Examples

```
copy the growth of theObj to itsGrowth --a constant, or list (if custom)
copy (the growth of textbox “txtMessage”) as list to {l, t, r, b} --a list
```

Notes

➤ The **`growth`** property is a list of four integers, similar to a bounding rectangle, telling how each side of a window item should move in relation to the two moving sides of the window as it is being resized. The integers are in the order: left, top, right and bottom sides. Each integer gives the percentage of that side's movement relative to the growth of the window's side. For example, {0, 0, 0, 0} means that no sides move—the item does not grow or move. The list {100, 100, 100, 100} means that all sides move; the item will slide over and down, following exactly the movement of the window's sides, but does not grow. The list {0, 0, 100, 100} anchors the left and top sides, but the right and bottom sides follow exactly the movement of the window's sides—that is, the item grows.

➤ All the important **`growth`** properties are represented by the constants listed above, under the heading “Value Class”.

➤ Any **`growth`** property value that is normally returned as a constant can be coerced to a list of four integers.

➤ The **`growth`** of a window item defaults to **`none`**.

height

Height in pixels of a window item.

Value Class

integer

Examples

```
set ht to the height of theObj
set the height of push button “Big Button” of window 2 to 64
set the height of theObj to height of box 2 of window 3
```

Note

➤ Setting the value of the **`height`** changes the value of the **`bounds`** property, but the value of the **`position`** property is unchanged.

id

Unique identification number of a window item.

Value Class

integer

Examples

```
copy the id of theObj to idNum
copy the id of push button 3 of window 5 to idOfButton
set the fill color of window item id idNum to black
```

Notes

➤ Each window item receives an **`id`** when it is created, and that **`id`** never changes.

➤ The **`id`** property is assigned starting with 7001. The maximum **`id`** is 32767 (the largest number that is an integer). Integers less than 7000 are used for the **`index`** property.

➤ **`Id`** is a read-only property.

index

Order of a window item in the back-to-front layering of items in the window.

Value Class

integer

Examples

```
copy the index of theObj to itsPosition
set the index of textbox “MoveToBottom” to 1 --move the item to the back
set index of icon “RaiseByOne” to ((index of icon “RaiseByOne”) + 1)
```

Notes

➤ The window items in each window are indexed sequentially from the rearmost item (**`index`** 1) to the frontmost item.

➤ Increasing the **`index`** of a window item moves it forward in the layering, obscuring the objects behind it (objects with lower indices).

➤ Changing the **`index`** of any window item other than the frontmost one always changes the indices of other window items.

➤ An object reference—that is, **`a reference to`** the object—is implemented in terms of the **`index`**.

name

Name of a window item.

Value Class

string

Examples

```
copy the name of theObj to itsName
set myButtonName to name of push button 2 of window “AvailableNames”
set the name of icon 2 of window “Bots” to “Tom Servo”
```

Notes

➤ Window items do not display the values of their **`name`** properties; names are used only for referencing window items in scripts. For displayed values, see the descriptions of the **`contents`** property or the **`title`** property of specific window items.

➤ Window item names default to a three-letter prefix giving the class, concatenated to the word “Name,” which is concatenated to a digit or two giving its order of creation. For example, the first push button you create is called “pshName1.”

pen color

Color of the window item’s drawn areas, usually its text and borders.

Value Class

RGB color	{redValue, greenValue, blueValue}
integer	index to color in System color lookup table
constant	black / white

Examples

```
copy the pen color of theObj to itsPen
set pen color of push button 2 to black
set pen color of box “grayFrame” to {32639,32639,32639}
set pen color of box “grayFrame” to 87
```

Notes

➤ The **`pen color`** property is always returned as an RGB value, a list of three long integers, from 0 to 65535, representing red, green and blue intensities.

➤ The specified **`pen color`** replaces all the black areas of pen and fill patterns, as shown in the **`pen pattern`** and **`fill pattern`** palettes.

➤ By default, **`pen color`** is **`black`**.

Table of Contents Index

position

Position of a window item within its window.

Value Class

list of integer	point	{left, top}

Examples

```
copy the position of theObj to {itsLeft, itsTop}
set the position of push button 3 of window “Satellite” to buttonPos
set the position of box “MoveMeToTopLeftOfWindow” of window 3 to {0,0}
```

Notes

➤ The **`position`** of a window item consists of the coordinates of its upper-left corner relative to the upper-left corner of the window’s content area, in pixels.

➤ Changing a window item’s **`position`** also changes its **`bounds`**, but not its **`height`** or **`width`**.

script

Compiled script of a window item.

Value Class

script

Examples

```
copy (the script of theObj) as string to itsScript --decompiles
set testScript to “on hilited theObj” & return & “beep 1” ¬
   & return & “end hilited”
set the script of push button “pshTester” to testScript
--A script can be put into a variable and executed:
copy the script of push button 1 to presto
tell presto to hilited
```

Notes

➤ When the text of a script is assigned to the **`script`** property, it is automatically compiled.

➤ An error occurs if the text of the assigned script cannot be compiled. You must handle such an error, probably with a **`try`** statement.

➤ As soon as the **`script`** property has been successfully assigned, it can be executed.

➤ A **`script`** can be decompiled into a string simply by using **`as string`** to coerce it.

➤ After a **`script`** has been copied to a variable (as script), the handlers in that script can be told to execute, is in the examples.

➤ Window items by default have *no* script, not simply an empty one. (A window has an empty script by default.)

size

Size in points of a window item’s displayed text.

Value Class

integer

Examples

```
copy the size of theObj to itsFontSize
set the size of textbox “txt8PointText” of window “TextSizes” to 8
set the size of the selection of textbox “txtMessage” to 18
```

Notes

➤ If the **`size`** of a window item, such as a button, was not explicitly set, its text is displayed in the font size of the window.

➤ Changing the **`size`** property of a window changes the font size of every window item for which the **`size`** property has not been explicitly set.

➤ To adjust the line spacing of text contained in textboxes, use the **`line height`** property.

➤ To refer to the **`size`** of a selection within a textbox, use a **`selection`** reference as shown in the example.

➤ The default **`size`** is 12 points.

Table of Contents Index

style

Text style of the first character of the contents or title of a window item.

Value Class

text style info

Examples

```
set itsStyle to the style of theObj
set the style of textBox “txtName” of window “Preferences” to itsStyle
set the style of the selection of textbox “txtMessage” to itsStyle
set the style of textbox 3 of window “Notes” to ¬
   {on styles:{italic},¬
   off styles:{bold,underline,outline,shadow,group}}
```

Notes

➤ If the **`style`** of a window item, such as a button, was not explicitly set, its text is displayed in the font style of the window.

➤ Changing the **`style`** property of a window changes the font style of every window item for which the **`style`** property has not been explicitly set.

➤ To refer to the **`style`** of a selection within a textbox, use a **`selection`** reference, as shown in the example.

➤ The default **`style`** is simply plain text.

➤ For information about the **`text style info`** class, see Chapter 15: “Special Artwork and Textstyle Classes.”

uniform styles

Text styles that are uniformly on and off for all text contained by a window item.

Value Class

text style info

Examples

```
copy the uniform styles of theObj to itsUniStyles
set the uniform styles of the selection of textbox 2 of window 1 to itsUniStyles
set the uniform styles of textbox 3 of window “Notes” to ¬
    {on styles:{italic},¬
    off styles:{bold,underline,outline,shadow,group}}
```

Notes

➤ If the **`uniform styles`** of a window item, such as a button, was not explicitly set, its text is displayed in the uniform styles of the window.

➤ Changing the **`uniform styles`** property of a window changes the uniform styles of every window item for which the **`uniform styles`** property has not been explicitly set.

➤ To refer to the **`uniform styles`** of a selection within a textbox, use a **`selection`** reference, as shown in the example.

➤ For information about the **`text style info`** class, see Chapter 15: “Special Artwork and Text Style Classes.”

visible

Is the window item visible in its window?

Value Class

boolean

Examples

```
set itsVisibility to the visible of theObj
set the visible of textbox “txtMessage” to true
```

Notes

➤ If **`visible`** is **`false`**, the window item cannot receive messages from interactions with the application user.

➤ Invisible window items can be forced to draw, using the **`draw`** command.

➤ A window item can lie beyond the bounds of the window, and thus be unseen, yet still have a **`visible`** property of **`true`**.

➤ **`Visible`** defaults to **`true`**.

width

Width in pixels of a window item.

Value Class

integer

Examples

```
copy the width of theObj to itsWid
set the width of push button “pshBigButton” of window 2 to 64
```

Note

➤ Changing the **`width`** of a window item changes its **`bounds`** but not its **`position`**.

Window Item Command and Event Messages

Certain command and event messages can be sent to any window item. These messages are listed here; any exceptions are noted.

Any event message can be sent by a command; the command name is the name of the message.

adjust size

Command to adjust the size of the window item to fit its contents.

Parameters

(direct)	reference	the window item to be adjusted

Examples

```
adjust size theObj

set the contents of label “lblMessage” to “Please try again.”
tell label “lblMessage” to adjust size
```

Notes

➤ Labels, textboxes and listboxes adjust their heights to accommodate the nearest number of whole lines of text, based upon the font size.

➤ Popups adjust their heights to accommodate the font size of an item, and adjust their lengths to accommodate the longest item.

➤ Gauges (scrollbars) adjust their widths (the narrow dimension) to the standard size, 16 pixels.

➤ Tables adjust their widths and heights to accommodate the nearest whole row and column.

➤ Pictboxes, icons and movies grow or shrink to fit the image exactly.

➤ The **`position`** of a window item remains stationary during the size adjustment, but the **`bounds`** may change.

click

Event message sent when the application user clicks anywhere in the window item.

Parameters

(direct)	reference	the window item
at	point	coordinates of the mouse
upon	constant	window part (see notes)
[option down]	boolean	Option key is down
[shift down]	boolean	Shift key is down
[command down]	boolean	Command key is down
[control down]	boolean	Control key is down
[ticks]	long integer	time at which click occurred

Examples

```
--Make a box act like a push button:
on click theObj at thePos upon thePart
    set fill color to black
    DoAction()
    set fill color to white
    continue click theObj at thePos upon thePart
end click

--Make a box act like a checkbox:
property nowOn:false
on click theObj at thePos upon thePart
    if nowOn then
        set fill color to white
        DoOtherAction()
    else
        set fill color to black
        DoAction()
    end if
    set nowOn to not nowOn
    continue click theObj at thePos upon thePart
end click
```

Notes

➤ The coordinates of the mouse are relative to the window; the top-left corner of the content area of a window is at {0, 0}.

➤ The constants for the **`upon`** parameter specify the various parts of a window: **`close box`**, **`zoom box`**, **`size box`**, **`title bar`**, **`menu bar`** and **`none`**. The content area of a window is given as **`none`**.

➤ The **`ticks`** parameter is given as 60ths of a second since system startup; it is used to calculate elapsed time, not actual clock time.

➤ Any handler for the **`click`** message must **`continue`** the message or normal interaction with the window item and window will not occur.

➤ If the **`upon`** parameter is not accepted and continued, normal window operations, such as closing and resizing, will be inhibited.

➤ If a **`click`** handler in the window needs a certain parameter, then any **`click`** handlers in the window items must accept and **`continue`** that parameter.

drag

Event message sent when a window item is about to be dragged.

Parameters

(direct)	reference	the window item to be dragged
data	list	the data to be dragged
initial outline	rectangle	the bounds of the item
initial mouse position	point	location where drag began

Examples

```
   on drag theObj data theData initial outline theBox initial mouse position
   thePt
   --Alter theData here, or do nothing...
   try
      continue drag theObj data theData initial outline theBox  ¬
       initial mouse position thePt
   on error
      --Script gets here if the mouse is released over a non-droppable item:
      display dialog “I saw you littering!” buttons {“Sorry”}
   end try
end drag
```

Notes

➤ The **`data`** parameter is a list whose values describe the information to be dragged. For a textbox, the **`data`** is a list of three values, the **`contents`** of the **`selection`**, the **`styles`** of the **`selection`**, and the **`text`** of the **`selection`**. For a listbox, the **`data`** is a list of two values, the **`contents`** of the **`selection`** (itself a list) and the same information coerced to a return-delimited string. For a movie, the **`data`** is a list of one element, the displayed frame (a picture).

➤ You can change the contents of the **`data`** parameter to control the format of the information that is dragged. It can be changed to whatever suits yours needs, but the **`drop`** message handlers must expect the same format.

➤ The handler for the **`drag`** message must be continued if you wish the drag to be completed. You can inhibit the drag simply by not continuing the message.

➤ The **`continue`** statement fails if the application user “drops” the data over an object that does not accept drops. You can use the **`try`** statement to handle these cases.

draw

Command to redraw the window item.

Parameters

(direct)	reference	the window item to be drawn

Examples

```
draw theObj
draw window item 2 of window 1
draw push button “AllMessedUp” of window “Pandemonium”
tell window item 3 to draw
```

Notes

➤ When a handler makes visible changes to a window or its items, the changes are normally not shown until the handler concludes. To display particular changes as soon as they are made, you can send **`draw`** commands from the handler to the window or window items that were changed.

➤ To draw all the window items, just **`tell`** the window to **`draw`**.

➤ Icons and pictboxes will draw even if their **`visible`** properties are **`false`**. A simple form of animation entails putting all the images in an invisible pile, then telling them to **`draw`** in succession.

➤ You often can avoid the necessity of sending **`draw`** commands by using the **`idle`** command, which allows windows to receive normal update events. See the discussion of the **`idle`** message in Chapter 12: “Windows.”

drop

Event message sent when dragged data is about to be dropped.

Parameters

(direct)	reference	the window item to be dropped into
data	list	the data that is to be dropped
upon	integer	the part of the item that will receive the drop
final outline	rectangle	the place where the drag outline ended

Examples

```
on drop theObj data theData upon endItem final outline endBox
    --This handler accepts only values that can be coerced to string:
    copy (item 1 of theData) to x --get the “contents” value
    if class of x is string then
        --Here the default behavior is accepted:
        continue drop theObj data theData upon endItem final outline endBox
    else if class of x is list then
        --Here the handler uses the data the way it wishes:
        set contents to x
    else
        beep 1 --it is not what I need
        --Here the data is not used and the default behavior is rejected.
    end if
end drop
```

Notes

➤ The **`data`** parameter is a list whose values describe the information to be dragged. For a textbox, the data is a list of three values, the **`contents`** of the **`selection`**, the **`styles`** of the **`selection`**, and the text of the **`selection`**. For a listbox, the data is a list of two values, the **`contents`** of the **`selection`** (itself a list) and the same information coerced to a return-delimited string. For a movie, the data is a list of one element, the displayed frame (a picture).

➤ If you changed the format of the **`data`** parameter in the **`drag`** handler, then that is the format that the **`drop`** handler will receive.

Table of Contents
Index

➤ **`Continue`** the **`drop`** message only if you wish to accept the default behavior. In a textbox, dropped text is inserted at the insertion point. In a listbox, dropped text or listbox items are inserted at the (horizontal) insertion point.

➤ To be droppable a window item must be **`visible`**. A box must have a **`fill pattern`** or **`fill color`**, or else only its border can receive a **`drop`**.

get data

Command to get some data from the window item.

Parameters

(direct)	reference	the window item
[as]	type class	the type class wanted

Examples

```
set x to get data of theObj
```

Notes

➤ The value returned by **`get data`** is identical to the value of the **`contents`** property of the object. See the discussion, above, of the **`contents`** property.

➤ Use the **`as`** parameter to coerce the default value into another type class.

➤ The **`get data`** command is not normally used, since properties are accessed by name; **`get data`** is provided for completeness only.

mouse entered

Event message sent when the mouse becomes positioned over the window item.

Parameters

(direct)	reference	the window item

Examples

```
--Make a box act as if it has the focus:
on mouse entered theObj
   set the pen size to {4, 4}
end mouse entered
```

Notes

➤ This message is paired with the **`mouse left`** message; that is, receiving **`mouse entered`** means that the window item will eventually receive **`mouse left`**.

➤ The **`mouse entered`** message handler often is used to initialize conditions for the **`mouse within`** message handler.

mouse left

Event message sent when the mouse becomes no longer positioned over the window item.

Parameters

(direct)	reference	the window item

Examples

```
--Make a box (see previous example) act as if it lost the focus:
on mouse left theObj
   set the pen size to {1, 1}
end mouse left
```

Notes

➤ This message is paired with the **`mouse entered`** message.

➤ The **`mouse left`** message handler often is used to finish up after the **`mouse within`** message handler.

mouse within

Event message sent repeatedly while the mouse remains positioned over the window item.

Parameters

(direct)	reference	the window item

Examples

```
--Animate a progress bar while mouse is over this window item:
on mouse within theObj
   set myCount to myCount + 1
   set the scroll of gauge “gagProgress” to myCount
end mouse within
```

Notes

➤ The **`mouse within`** message handler often is used to create animated effects. It often is combined with handlers for the **`mouse entered`** and **`mouse left`** messages.

➤ Changes made to the appearances of other window items or of the window become visible when the handler terminates, which normally is immediately.

➤ The **`mouse within`** message is sent as often as possible, when no other events are pending.

set data

Command to assign data to a window item.

Parameters

(direct)	reference	the window item
to	anything	the data to be assigned

Examples

```
set data of theObj to someData
```

Notes

➤ The class of the value assigned by **`set data`** should be identical to the class of the value of the **`contents`** property of the object. See the discussion, above, of the **`contents`** property.

➤ The **`set data`** command is not normally used, since properties are accessed by name; **`set data`** is provided for completeness only.

show balloon

Command to show the help balloon for the item.

Parameters

(direct)	reference	the window item
item	integer	the part of the item that the cursor is over
message	string	the message to appear in the balloon
within	rectangle	the bounds within which to display the balloon

Examples

```
tell window item 7 to show balloon item 1 message “Click here to quit.” ¬
   within bounds of window item 7
```

Forms and Filters

Forms and Key Filters

The appearance and behavior of every button, gauge, listbox, menu, popup, table cell and textbox is controlled by a form resource. A listbox form, for example, might allow the listbox to display icons in addition to text, while a button form might give the button a three-dimensional appearance.

The default standard form used by each window item is built into FaceSpan. The standard listbox form, for example, lets a listbox display only text. The default button form produces standard Macintosh buttons.

A form for a textbox, and the corresponding property, is called a key filter. A key filter is a form that determines what can be typed into a textbox, and how the contents is formatted.

You can add form definition resources to a project, then use them to control window items. You import forms using the Forms View of the Project Window, where the forms you have imported will be listed by name. The little icon in front of each name has a letter indicating the kind of form it is: a “C” represents forms for buttons and gauges, an “L” marks the forms for listboxes, an “M” denotes forms for menus and popups, and a “K” represents key filters for textboxes.

You assign a particular form to a window item by setting the item’s **`form`** (or **`key filter`**) property, a string, to the name of that form (or key filter).

Formats

Some forms are so simple that they need no additional information to work properly. For example, the OnlyDigits key filter allows only digits to be typed into a textbox. Other forms can perform any of a number of functions and so must be told which function to perform. These instructions are given to a form by way of a format property.

Every window item that has a form property also has a format property. When you assign a form to a window item, the format—if one is needed at all—must be filled in appropriately. For example, the DisplayDates key filter can make a textbox display a date in several ways; to make it display a date as “mm/dd/yy”, the format property must be set to the words “short date”.

By the way, menus can use forms, but they do *not* use formats.

Form and Format Documentation

In the Forms View of the Project Window, double-clicking the name of a form displays a modal dialog containing general information about the form.

When you are editing a window template, you can assign a form to a window item. When you then choose Format from the Properties popup of the Property Bar, a modal dialog appears; it tells the purpose of the form and how it is used, and gives examples. You can copy the examples and use them for your formats.

Key Filters

The forms for textboxes and table cells are called key filters. Key filters have special behaviors that are quite powerful, and which thus warrant special attention.

A key filter can determine the value class of data assigned to, and retrieved from, a textbox or table cell. For example, assume that the DisplayNumbers key filter is assigned to a textbox. You can assign a number to the contents of the textbox, and when you get the contents, it is a number, not text. Of course, you can still use **`as text`** to coerce it to text.

Furthermore, using the **`format`** property of the textbox can give results that are even more useful. For example, assume that the string “'$'###,###.00” is used as the format with the DisplayNumbers key filter. Now, if you set the contents of the textbox to 1234.5, it automatically displays “$1,234.50”. If you get the contents, it is still 1234.5, but if you get it **`as text`**, it is “1,234.50”.

Examples and Sources of Forms

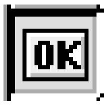
The “•Feature Highlights•” folder contains many projects that demonstrate the features of several forms, and how to write scripts for window items that use forms. In addition, the “FaceSpan Additions” project document contains a variety of forms. Forms are also available from third parties. You can import the forms from any of these sources directly into your projects, by way of the Forms View of the Project Window.

Table of Contents Index

Some Technical Notes

If you are a programmer, you probably recognize that forms for buttons and gauges are code resources of type “CDEF,” forms for listboxes are “LDEF” resources, menu and popup forms are “MDEF”resources, and key filters are “Keyƒ” resources. Information about writing your own forms, in C or Pascal, is available in the FaceSpan SDK (software developers kit).

Bevel Buttons



Bevel buttons permit users to start and end processes in Macintosh windows. They are typically used to summon windows from an existing window and are always used as the method of closure of modal dialogs.

FaceSpan automates the highlighting that occurs when bevel buttons are clicked, and provides a set of properties for extended control of bevel buttons.

➤ The **`command key`** property permits a push button to be activated by a user-specified Command-key combination.

Properties of Bevel Buttons

Bevel buttons have the properties shown here in addition to several properties they have in common with most or all window items. See *Common Properties* on page 277.

Artwork

Example

```
set the artwork of bevel button 1 to {class:resourse info, type: “cicn”, name:
“Caution,” id:5003}
```

Notes

➤ The default value of Artwork is none.

➤ Art Alignment can be set to many different positions on the button. It should be noted that it is possible to cover up the title of the button with the artwork; if this happens you will need to adjust the “Text Placement” property to reveal the title.

auto close

Does clicking the button close its window?

Value Class

boolean

Examples

```
set the auto close of push button “pshScram” of window “Reactor” to true
```

Notes

➤ The **`auto close`** property usually is set in edit mode.

➤ When a button whose **`auto close`** property is **`true`** is used to close a window, the **`closing item`** property of the window contains a reference to the button.

➤ The **`auto close`** property may be **`true`** for any number of buttons in a window.

➤ **`Auto close`** is, by default, **`false`**.

Behavior

Value Class

constant	as push button	one bevel button can be momentarily selected
	as radio button	one bevel button can be persistently selected; other bevel buttons with this selection rule are deselected
	as checkbox	several bevel buttons with this selection rule can be persistently selected

Examples

```
set the selection rule of bevel button 1 to as checkbox.
```

➤ Bevel buttons can act as radio buttons or checkboxes.

command key

Defines the optional Command-key equivalent that activates the button.

Value Class

string	an alphanumeric character

Examples

```
set command key of push button “Error” to “E”
```

Notes

➤ The **`command key`** property is usually set in edit mode.

➤ Do not set the **`command key`** to a character that is already used as the command key of a menu item; the menu item takes precedence, so the button would be ignored.

➤ The default value of **`command key`** is the null string, meaning no command key is in effect.

form

The name of a form definition resource in the project.

Value Class

constant	standard	kSmallBevelButton, KLargeBevelButton
string	see notes	

Examples

```
set the form of bevel button 1 to standard
```

Notes

➤ The **`form`** property usually is set in edit mode.

➤ The default standard form for a bevel button (a resource of type CDEF) is built into FaceSpan. It has the value **`standard`**.

➤ Optional button forms can be imported into a project. These can support the display of icons and pictures and have a variety of other features.

➤ A form might require the use of the **`format`** property.

➤ To see basic documentation for a form, select its name in the Forms View of the Project Window, then click the Open button.

➤ For a detailed discussion of forms and formats, see the section entitled “Form Definition Resources and Formats” in this chapter.

Table of Contents Index

format

A string of parameters for use by a form definition resource.

Value Class

string

Examples

```
set the format of window item 3 to theFormatString
```

Notes

➤ To see basic documentation for using the **`format`** property with a given form, first assign the form to the window item, then choose Format from the Properties pop-up menu in the Property Bar.

➤ Some forms do not require the use of the **`format`** property.

➤ For a detailed discussion of forms and formats, see the section entitled "Form Definition Resources and Formats" in this chapter.

hilite

Is the bevel button hilited?

Value Class

boolean

Notes

➤ The **`hilite`** property of a bevel button is always **`false`**, because it immediately reverts to **`false`** after the **`hilited`** message is sent.

➤ The bevel button actually highlights when it is clicked, and stays highlighted as long as the mouse button is down and the cursor remains over the button. Only when the mouse button is released is the button sent the **`hilited`** message.

links (Bevel Buttons)

➤ When a bevel button is acting as a radio button or checkbox, you can link items in the window, such as buttons, checkboxes and so forth, to the bevel button. You can choose to enable, disable, hide or show the linked items.

To link items, double-click the bevel button in the Window Editor to open its Object Information dialog box. From the Behavior pop-up menu, choose the radio button or checkbox. Click the Links button to open the Links dialog box. Select the items you want to link from the list box and click the radio buttons to enable, disable, hide or show the linked items.

title

The text displayed by the bevel button.

Value Class

string

Examples

```
copy the title of theObj to its Title
set the title of bevel button 3 of window “Instructions” to “Click me”
```

Note

➤ If a script sets the **`title`** of a bevel button, it might also need to tell the bevel button to **`adjust size`** to fit the title.

➤ It should be noted that it is possible to cover up the title of the button with the artwork; if this happens you will need to adjust the “Text Placement” property to reveal the title.

Bevel Button Command and Event Messages

This section describes only the event messages that are sent specifically to bevel buttons.

Bevel buttons can also receive and handle several messages that are sent to most or all window items. See *Window Item Command and Event Messages* on page 293.

hilited

Event message sent when the bevel button is clicked.

Parameters

(direct)	reference	the push button

Examples

```
on hilited of theObj
    set title of theObj to “I was clicked”
end hilited
```

Notes

➤ A bevel button receives the **`hilited`** message only once per click, when the mouse button is released.

➤ Since the bevel button’s **`hilite`** property changes only very briefly, it is always **`false`** when the **`hilited`** message is sent.

➤ A bevel button can receive a **`hilited`** message in response to certain keystrokes, as well as when clicked. Refer to the **`default item`**, **`cancel item`**, and **`command key`** properties for more information.

➤ A bevel button can be designated as the **`doubleclick item`** of a list box or table. When an entry in the list box or table is double-clicked, the button highlights and is sent the **`hilited`** message.

Boxes

Boxes are rectangular graphic objects used in windows principally to enclose groups of related objects or for artistic effects.

The **`pen pattern`**, **`pen size`**, and **`pen color`** properties determine the pattern, thickness, and color of a box's outline. Although the background of each newly-created box is transparent, its **`fill pattern`** and **`fill color`** properties may be set to specify a pattern and color for its background.

The corners of a box can be rounded by setting the **`corners`** property.

If the **`title`** property of a box is set to a non-empty string, the title text is displayed within the upper side of the box. When there is not enough space to display the entire text of a title, the text is truncated and an ellipsis added.

A box can be made into a *scrolling pane* by setting its **`scrollable`** property to **`true`**. A scrolling pane is like the scrolling area of a textbox, except that the area can contain other window items.

Properties of Boxes

Boxes have the properties shown here in addition to several properties they have in common with most or all window items; see the section, "Properties Common to All Window Items," at the beginning of this chapter.

corners

The diameters of the ovals that define the corners of the box.

Value Class

list of integer	point	{horizontal, vertical}

Table of Contents Index

Examples

```
copy the corners of box 3 to {hDiam, vDiam}
set the corners of box “boxSlightlyRounded” to {8,8}
set the corners of box “boxModeratelyRounded” to {16,16}
set the corners of box “boxCircular” to {999,999}
```

Notes

➤ The **`corners`** property is expressed as horizontal and vertical diameters of ovals that would fit into the corners of the box.

➤ If the corner diameters are set to large numbers, a rectangular box becomes oval, while a square box becomes circular.

➤ The **`corners`** of a standard push button are {8, 8}.

➤ The **`corners`** property defaults to {0, 0}—square corners.

fill pattern

Pattern with which a box’s background is filled.

Value Class

constant	white / black / gray / dark gray / light gray / none
integer	index to a pattern in the pattern table, or id of a color pattern (ppat) in the project

Examples

```
copy the fill pattern of theObj to itsFill
set the fill pattern of box “boxSomber” to dark gray
```

Notes

➤ The Fill Pattern popup of the Properties popup in the Property Bar displays the entire pattern table.

➤ Color pattern (ppat) resources are imported into a project in the Artwork View of the Project Window. A segment of each is displayed in the Fill Pattern popup.

➤ The default **`fill pattern`** is **none**.

➤ If the **`fill pattern`** of a box is **none** and you set the **`fill color`** to a color other than **`white`**, **`fill pattern`** is automatically changed to **`white`** so that the fill color becomes visible.

justification

Alignment of the title within the top line of the box.

Value Class

constant	left / right / center

Examples

```
set the justification of box “boxPrefs” to right
```

Notes

➤ The **`justification`** defaults to **`left`** (but there is no default **`title`**).

pen pattern

Pattern with which a box’s outline will be drawn.

Value Class

constant	white / black / gray / dark gray / light gray / raised / inset
integer	index to a pattern in the pattern table, or id of a color pattern (ppat) in the project

Examples

```
copy the pen pattern of theObj to itsPenPat
set the pen pattern of box 3 of window 5 to dark gray
set the pen pattern of box “boxFrame” to inset
```

Notes

➤ The Pen Pattern popup of the Properties popup in the Property Bar displays the entire pattern table.

Table of Contents Index

➤ Color pattern (ppat) resources are imported into a project in the Artwork View of the Project Window. A segment of each is displayed in the Pen Pattern popup.

➤ The **`raised`** and **`inset`** pen patterns produce 3-D effects of shadow and highlight. The current **`pen color`** is used for the shadow, while the highlight is **`white`**.

➤ By default, **`pen pattern`** is **`black`**.

pen size

Line thickness of the box's border.

Value Class

list of integer	point	{penWidth, penHeight}

Examples

```
copy the pen size of theObj to {penWid, penHt}
copy the pen size of box “Nib” of window “Styles” to currentpen
set the pen size of box 3 of window 7 to {2,2}
```

Notes

➤ The Pen Size popup of the Properties popup in the Property Bar offers pen sizes up to {4, 4}, but other values can be set by a script.

➤ The default **`pen size`** is {1, 1}.

scroll

Distance that the content of a scrollable box has been scrolled.

Value Class

list of integer	point	{horizAmount, vertAmount}

Examples

```
copy the scroll of theObj to {hPos, vPos}
set the scroll of box “boxPrefs” to page2Position
```

Notes

➤ The **`scroll`** property is measured in pixels. It is {0,0} if the **`scrollable`** property is **`false`**.

➤ Setting the **`scroll`** property causes the box to **`scroll`**, and the scroll bars to adjust accordingly.

➤ Setting the **`scroll`** property does not cause a **`scrolled`** message to be sent to the box; a script can do that explicitly if necessary.

➤ A box whose **`scrollable`** property is **`true`** behaves as a *scrolling pane*. See the discussion of scrolling panes, below.

scrollable

Does the box have scroll bars?

Value Class

boolean

Examples

```
set itemScrollable to scrollable of theObj
set scrollable of box “boxDescription” of window “Order Entry” to false
```

Notes

➤ A box whose **`scrollable`** property is **`true`** behaves as a *scrolling pane*. See the discussion of scrolling panes, below.

➤ When the **`scrollable`** property is set to **`true`**, the box becomes 15 pixels wider and taller; it becomes 15 pixels narrower and shorter when the **`scrollable`** property is set to **`false`**.

➤ When the value of **`scrollable`** is **`false`**, the box has no scroll bars, and the contents cannot be scrolled by any means. The value of its **`scroll`** property is {0, 0}.

➤ **`Scrollable`** defaults to **`false`**.

title

The text displayed in the top line of a box.

Value Class

string

Examples

```
copy the title of theObj itsTitle
set title of box “boxPrefs” of window “Instructions” to “Current Preferences”
```

Note

➤ The default **`title`** of a box is the null string, which does not display.

Box Command and Event Messages

This section describes the only event message that is sent specifically to boxes.

Boxes can also receive and handle several messages that are sent to most or all window items; see the section, “Window Item Command and Event Messages,” above.

scrolled

Event message sent when the box is scrolled with a scrollbar.

Parameters

(direct)	reference	the box

Example

```
on scrolled theObj
     copy scroll of theObj to newscroll
    set scroll of box “boxToys” to newscroll
end scrolled
```

Notes

➤ A box whose **`scrollable`** property is **`true`** behaves as a *scrolling pane*. See the discussion of scrolling panes, below.

Table of Contents   Index

➤ Setting the **`scroll`** from a script does not cause the **`scrolled`** message to be sent; send a **`scrolled`** message, if needed.

➤ Two or more boxes can be made to scroll in parallel by copying the **`scroll`** property of each one to the others when the **`scrolled`** message is received.

Scrolling Panes

A scrolling pane is a box whose **`scrollable`** property is set to **`true`**. (See the **`scroll`** and **`scrollable`** properties and the **`scrolled`** message for boxes.)

A scrolling pane "encloses" all the window items whose indices are less than that of the box itself. These window items will disappear if they lie outside the bounds of the box, but will become visible within the bounds of the box when the box is scrolled in their original directions.

Scrolling a scrolling pane actually changes the coordinates of the window items it contains; this is true both when playing or running the window. If a scrolling pane is scrolled while playing a window, and the window is then put into edit mode, the enclosed window items will be moved from their original positions.

There is a scrolling pane example among the Feature Highlights that come with FaceSpan.

Checkboxes

Checkboxes are used to present options to the application user. Each option is either in effect (the checkbox is checked) or not (the checkbox is not checked). When a checkbox is clicked, FaceSpan automatically changes its **`hilite`** property to the opposite of its previous value, then sends it a **`hilited`** message.

Properties of Checkboxes

Checkboxes have the properties shown here in addition to several properties they have in common with most or all window items; see the section, “Properties Common to All Window Items,” at the beginning of this chapter.

form

The name of a form definition resource.

Value Class

constant	standard	standard checkbox
string	see notes	

Examples

```
set the form of theObj to “latch”
```

Notes

➤ The **`form`** of a checkbox usually is set in edit mode.

➤ The default standard form for a checkbox (a resource of type CDEF) is built into FaceSpan, and has the value **`standard`**.

➤ Optional button forms can be imported into a project. These can support the display of icons and pictures, and have a variety of other features.

➤ A form might require the use of the **`format`** property.

Table of Contents   Index

➤ To see basic documentation for a form, select its name in the Forms View of the Project Window, then click the Open button.

➤ For a detailed discussion of forms and formats, see the section entitled “Form Definition Resources and Formats” in this chapter.

format

A string of parameters for use by a form definition resource.

Value Class

string

Examples

```
set the format of window item 3 to theFormatString
```

Notes

➤ To see basic documentation for using the **`format`** property with a given form, first assign the **`form`** to the window item, then choose the **`format`** property from the Properties popup in the Property Bar.

➤ Some forms do not require the use of the **`format`** property.

➤ For a detailed discussion of forms and formats, see the section entitled “Form Definition Resources and Formats” in this chapter.

highlight

Same as the **`hilite`** property of a checkbox.

hilite

Is the checkbox hilited—that is, does it display a check?

Value Class

boolean

Examples

```
copy the hilite of window item 2 to isChecked
set the hilite of checkbox “chkSelfRunning” to true
```

Notes

➤ The **`hilite`** of a checkbox is set before the **`hilited`** message is sent to it.

➤ Setting the **`hilite`** property of a checkbox does not send a **`hilited`** message; a script can send the **`hilited`** message if necessary.

links (checkboxes)

➤ You can link items in the window, such as buttons, checkboxes and so forth, to checkboxes. You can choose to enable, disable, hide or show the linked items.

To link items, double-click the checkbox in the Window Editor to open its Object Information dialog box. Click the Links button to open the Links dialog box. From the list box, select the items you want to link and click the radio buttons to enable, disable, hide or show the linked items.

title

The text displayed by the checkbox.

Value Class

string

Examples

```
copy the title of theObj to itsTitle
set the title of checkbox 3 of window “Instructions” to “Section 3"
```

Notes

➤ If a script sets the **`title`** of a checkbox, it might also need to tell the checkbox to **`adjust size`** to fit the title.

➤ The **`title`** of a checkbox defaults to “Checkbox.”

Checkbox Command and Event Messages

This section describes the only event message that is sent specifically to checkboxes.

Checkboxes can also receive and handle several messages that are sent to most or all window items; see the section, “Window Item Command and Event Messages,” above.

hilited

Event message sent when the checkbox is clicked.

Parameters

(direct)	reference	the checkbox

Examples

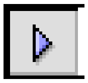
```
property footnotes:false
on hilited of theObj
   set footnotes to highlight of theObj
end hilited
```

Notes

➤ A checkbox receives a **`hilited`** message when it becomes checked and when it becomes unchecked. The **`hilite`** property value is set before the **`hilited`** message is sent.

➤ Setting the **`hilite`** property of a checkbox does not send a **`hilited`** message; a script can send the **`hilited`** message if necessary.

Disclosure Triangles



Disclosure triangles allow you to hide parts of a dialog box or window. The user can choose to display the hidden parts by clicking the disclosure triangle. Hidden parts of a dialog box can include buttons, list boxes, checkboxes and so forth.

Properties of Disclosure Triangles

Disclosure triangles have the properties shown here in addition to several properties they have in common with most or all window items. See *Common Properties* on page 277.

form

The name of a form definition resource.

Value Class

Constant	standard	kLeftDisclousureTriangle

Examples

```
Set the form of disclosure 1 to standard
```

links (disclosure triangles)

➤ You can link items in the window, such as buttons, checkboxes and so forth, to disclosure triangles. You can choose to enable, disable, hide or show the linked items.

To link items, double-click the disclosure triangle in the Window Editor to open its Object Information dialog box. Click the Links button to open the Links dialog box. From the list box, select the items you want to link and click the radio buttons to enable, disable, hide or show the linked items.

window expansion

You can set the window to expand when a disclosure triangle is selected.

Examples

```
set delta width of disclosure to 50
set delta height of disclosure 1 to 150
```

Disclosure Triangle Command and Event Messages

This section describes only the event message that is sent specifically to disclosure triangles.

Disclosure triangles can also receive and handle several messages that are sent to most or all window items; see *Window Item Command and Event Messages* on page 293.

hilited

Event message sent when the disclosure triangle is clicked.

Parameters

(direct)	reference	the disclosure

Examples

```
on hilited the Obj
    set TriangleState to hilited of theObj
    if TriangleState=true then
        set the contents of textbox 1 to “The Disclosure Triangle is Opened!”
    else
        set the contents of textbox 1 to “The Disclosure Triangle is Closed!”
    end if
end hilited
```

Notes

➤ A disclosure triangle receives a hilited message only when it becomes hilited, not when its hilite becomes false.

➤ A disclosure triangle's hilite is set before the hilited message is sent to it.

Gauges, Sliders, Progress Bars, Spinning Arrows and Up/Down Arrows

Gauges Sliders Progress bar

Gauges are objects such as scroll bars and progress indicators.

The **`form`** property determines the type of gauge object that will be displayed. There are now several forms built into FaceSpan. Other custom forms may be imported.

The **`step`** and **`leap`** properties determine the line and page values by which a gauge can be adjusted.

The **`minimum`** and **`maximum`** properties determine the extremes to which a gauge can be adjusted.

Properties of Gauges

Gauges have the properties shown here in addition to several properties they have in common with most or all window items. See *Common Properties* on page 277.

form

The form of the gauge as defined by a form definition resource.

Value Class

constant	standard	a standard scroll bar
constant	kSlider	slider
constant	kForwardSlider	forward slider
constant	kForwardSliderWithTicks	forward slider & tick marks
constant	kReverseSlider	reverse slider

Table of Contents Index

constant	kReverseSliderWithTicks	reverse slider & tick marks
constant	kArrows	up/down arrows
constant	kChasingArrows	spinning arrows
constant	kProgressBar	progress bar
string	see notes	

Examples

```
copy the form of theObj to formName
set the form of gauge “gagHowGoesIt” to “Progress”
```

Notes

➤ The default standard **`form`** for a gauge (a resource of type CDEF) is built into FaceSpan; it is a standard scrollbar, and has the value **`standard`**. Other custom forms may be imported.

➤ Optional gauge forms can be imported into a project. These can support the display of icons and pictures and can have a variety of other features.

➤ A form might require the use of the **`format`** property.

➤ To see basic documentation for an imported form, select its name in the Forms View of the Project Window, then click the Open button.

➤ For a detailed discussion of forms and formats, see the section entitled “Form Definition Resources and Formats” in this chapter.

format

A string of parameters for use by a form definition resource.

Value Class

string

Examples

```
set the format of window item 3 to theFormatString
```

Notes

➤ To see basic documentation for using the **`format`** property with a given form, first assign the **`form`** to the window item, then choose Format Property from the Properties pop-up menu in the Property Bar.

➤ Some forms do not require the use of the **`format`** property.

➤ For a detailed discussion of forms and formats, see the section entitled “Form Definition Resources and Formats” in this chapter.

leap

Amount by which the scroll changes when the gauge is scrolled by a “page.”

Value Class

integer

Examples

```
copy the leap of theObj to leapAmount
set the leap of gauge “gagMycroller” to 25
```

Notes

➤ A standard scrollbar gauge is scrolled by a “page” by clicking in the gray area above or below the slider.

➤ If you set the **`leap`** property, you should also set the **`maximum`**, **`minimum`**, and **`step`** properties, which default to 10, 100, 0 and 1, respectively.

➤ The default **`leap`** for the standard scrollbar gauge is 10.

maximum

The maximum possible value of a gauge’s scroll (or setting) property.

Value Class

integer

Examples

```
copy the maximum of theObj to scrollMax
set the maximum of gauge “gogMyScroll” to 500
set the scroll of gauge “listscroll” to maximum of gauge “listscroll”
```

Notes

➤ If you set the `maximum` property, you should also set the `leap`, `minimum`, and `step` properties.

➤ The default `maximum` for the standard scrollbar gauge is 100.

minimum

The minimum possible value of a gauge’s scroll property.

Value Class

integer

Examples

```
copy the minimum of theObj to scrollMin
set the minimum of gauge “gogMyScroll” to 100
set the scroll of gauge “listscroll” to the minimum of gauge “listscroll”
```

Notes

➤ If you set the `minimum` property, you should also set the `leap`, `maximum`, and `step` properties.

➤ The default `minimum` of the standard scrollbar gauge is 0.

scroll

The amount that the gauge has been scrolled; its setting.

Value Class

integer

Examples

```
copy the scroll of theObj to scrollAmt
set the scroll of gauge “listscroll” to 27
```

Notes

➤ The `scroll` property is the current setting of the gauge; it can be any integral value between the values of the `minimum` and `maximum` properties.

➤ If a script sets the `scroll`, the gauge automatically moves to display the new value.

setting

Same as the **`scroll`** property of a gauge.

step

Amount by which the gauge's scroll (or setting) property will change when the gauge is incremented or decremented by a "line."

Value Class

integer

Examples

```
copy the step of theObj to scrollStep
set the step of gauge “listscroll”to 5
```

Notes

➤ A standard scrollbar gauge is scrolled by a "line" by clicking the Up Arrow or Down Arrow.

➤ The default **`step`** of the standard scrollbar gauge is 1.

➤ If you set the **`step`** property, you should also set the **`leap`**, **`maximum`**, and **`minimum`** properties.

title

The displayed text of the gauge.

Value Class

string

Examples

```
copy the title of theObj to gagTitle
set the title of gauge “gagNewProgress” to “Sorting…”
```

Note

➤ A standard scrollbar gauge does not display its title; other forms might do so.

Gauge Command and Event Messages

This section describes only the event message that is sent specifically to gauges.

Gauges can also receive and handle several messages that are sent to most or all window items. See *Window Item Command and Event Messages* on page 293.

scrolled

Event message sent when a gauge is scrolled interactively.

Parameters

(direct)	reference	the listbox

Example

```
on scrolled theObj
    copy the scroll of theObj to newscroll
    set the scroll of gauges 2 thru 5 to newscroll --scroll all in tandem
end scrolled
```

Note

➤ The **`scrolled`** message is not sent to the gauge when its **`scroll`** is changed by a script; a script can send the **`scrolled`** message if necessary.

Graphic Lines

Graphic lines are graphic objects used in windows principally to create divisions between objects or object groups.

The **`pen pattern`**, **`pen color`**, and **`pen size`** properties determine the pattern, color, and thickness of a graphic line.

Properties of Graphic Lines

Graphic lines have the properties shown here in addition to several properties they have in common with most or all window items; see the section, "Properties Common to All Window Items," at the beginning of this chapter.

pen pattern

The pattern with which a graphic line is drawn.

Value Class

constant	white / black / gray / dark gray / light gray / raised / inset
integer	index to a pattern in the pattern table, or id of a color pattern (ppat) in the project

Examples

```
copy the pen pattern of theObj to itsPenPat
set the pen pattern of graphic line 3 of window 5 to dark gray
```

Notes

➤ The Pen Pattern popup of the Properties popup in the Property Bar displays the entire pattern table.

➤ Color pattern (ppat) resources are imported into a project in the Artwork View of the Project Window.

➤ By default, **`pen pattern`** is **`black`**.

Table of Contents Index

pen size

The graphic line thickness in pixels.

Value Class

list of integer	point	{penWidth, penHeight}

Examples

```
copy the penSize of theObj to {penWid, penHt}
set penSize of graphic line 3 of window 7 to {2,2}
```

Notes

➤ The **`pen size`** of a graphic line needs only a height value if the graphic line is horizontal, and only a width value otherwise.

➤ The Pen Size popup of the Properties popup in the Property Bar offers pen sizes up to 4 pixels, but other values can be set by a script.

➤ The default **`pen size`** is {0, 1}, for a horizontal graphic line, or {1, 0}, for a vertical graphic line.

Graphic Line Command and Event Messages

There are no command or event messages sent specifically to graphic lines. However, graphic lines can receive and handle several messages that are sent to most or all window items; see the section, “Window Item Command and Event Messages,” above.

Icons

Icons are containers in which ICON (black and white icon), ICN# (Finder icon family), and cicn (color icon) resources from a project’s artwork resources can be displayed.

The **`hilite rule`** property permits icons to be used as push buttons, radio buttons, or checkboxes when clicked with the mouse.

The **`hilite style`** property permits icons to be hilited in a variety of styles.

Icons can be scaled to any size.

Properties of Icons

Icons have the properties shown here in addition to several properties they have in common with most or all window items; see the section, “Properties Common to All Window Items,” at the beginning of this chapter.

artwork

The project artwork resource displayed by an icon.

Value Class

resource info	
constant	none

Examples

```
copy the artwork of theObj to {type:itsType, name:itsName, id:itsID}
set the artwork of icon “icnArtist” to {type:”cicn”,id:5003,name:”Tom”}
```

Table of Contents Index

Notes

➤ Although the **`artwork`** property can be set under most circumstances by specifying only a **`name`** or **`id`**, it is more reliable to cite all three properties of the resource info class to avoid any confusion caused by two or more art resources having the same name.

➤ If the **`artwork`** property is set to **`none`**, the icon becomes transparent and may be positioned over another window item for use as a transparent button with user-definable highlighting behavior. For example, if you use such an object to cover a textbox that has several lines of text, when the user clicks it, the whole textbox can appear to highlight according to the icon's **`highlight style`** and **`selection rule`**.

➤ For information about the **`resource info`** class, see Chapter 15: "Special Artwork and Text Style Classes."

highlight

The term **`highlight`** can be used interchangeably with the term **`hilite`**; in fact, every use of **`highlight`** is changed to **`hilite`** when a script is compiled.

hilite

Is the icon highlighted?

Value Class

boolean

Examples

```
copy the hilite of theObj to itsHilite
set the hilite of icon "icnFlag" of window "Salutes Itself" to true
```

Notes

➤ The **`hilite`** property of an icon is always **`false`** if its **`hilite rule`** is **`none`**.

➤ The highlighting sequence of an icon, and the effects of **`hilited`** messages to it, are just like the behavior of the button it is set to imitate. See the **`hilite rule`** property.

➤ By default, **`hilite`** is **`false`**, since the default **`hilite rule`** is **`none`**.

hilite artwork

The artwork used when highlighting a button having the by exchange hilite style.

Value Class

resource info	
constant	none

Example

```
set the hilite artwork of icon “icnLogo” to {name:”FaceSpan”, id:5000}
```

Notes

➤ The **`hilite artwork`** applies only to icons with the **`by exchange`** value of the **`hilite style`**.

➤ Instead of changing the appearance of the normal icon, the **`hilite artwork`** is displayed; that is, the two icons are exchanged while the icon’s **`hilite`** is **`true`**.

➤ Since the default **`hilite style`** is **none**, the default **`hilite artwork`** is **none**.

hilite rule

The kind of button the icon imitates.

Value Class

constant	none / as push button / as radio button / as checkbox

Examples

```
set the hilite rule of icon “icnTaxOption” to as checkbox
```

Notes

➤ The **`hilite rule`** of an icon determines the kind of button it imitates.

➤ The highlighting sequence of an icon, and the use of **`hilited`** messages to it, is just like the behavior of the button it is set to imitate.

➤ By default, **`hilite rule`** is **none**.

hilite style

The visual change in the icon when it is clicked.

Value Class

constant	none	icon does not highlight when clicked
	by hilite	white areas of clicked icon are overlain with the System highlight color (from the Color control panel)
	by invert	colors of clicked icon are inverted
	by lasso	colors of clicked icon are inverted within contours that exclude the icon’s fill color
	by frame	clicked icon is surrounded by a frame
	by sink	clicked icon is surrounded by a frame indented by one column of pixels on the left and one row of pixels on the top
	by exchange	a different icon resource is used for the highlighted icon
	Standard	icon is darkened in a manner similar to the Finder’s icon hilting

Examples

```
copy the hilite style of theObj to itsStyle
set hilite style of icon 5 of window “Bugs” to by invert
```

Notes

➤ You can make all the areas of a given color highlight, if you wish: set the **`fill color`** property of the icon to that color.

➤ If the **`selection rule`** or **`hilite style`** is set to **none**, the icon does not highlight or receive a **`hilited`** message when clicked.

➤ If the **`hilite rule`** is not **none**, but the **`hilite style`** is **none**, the icon still acts like the designated button even though its appearance does not change.

➤ By default, **`hilite style`** is **none**.

Icon Command and Event Messages

This section describes the only event message that is sent specifically to icons.

Icons can also receive and handle several messages that are sent to most or all window items; see the section, "Window Item Command and Event Messages," above.

hilited

Event message sent when an icon with a hilite rule is clicked.

Parameters

(direct)	reference	the icon

Examples

```
on hilited theObj
    copy name of theObj to thechoice
    display dialog "You clicked icon " & thechoice
end hilited
```

Note

➤ If the **`hilite rule`** is set to **none**, the icon does not receive **`hilited`** messages.

Labels

Labels are non-editable, one-line text objects used as titles for other window items, headings for sections of windows, and general static text display.

The rectangular area bounding a label is transparent. When there is not enough space to display the entire text of a label, the text is truncated and an ellipsis added.

Labels lack many of the properties of textboxes, but are often more convenient.

Properties of Labels

Labels have the properties shown here in addition to several properties they have in common with most or all window items; see the section, “Properties Common to All Window Items,” at the beginning of this chapter.

justification

Alignment of the title within the bounds of the label.

Value Class

constant	left / right / center

Examples

```
set the justification of label “lblHeader” to right
```

Notes

➤ The **`justification`** property makes sense only if the bounds of the label are wider than the title it displays.

➤ The default **`justification`** is **`left`**.

title

Text that is displayed as the label.

Value Class

string

Examples

```
copy the title of theObj to itsTitle
set the title of label “lblSection” of window “Instructions” to “Section 3"
```

Note

➤ The default **`title`** of a label is “Label.”

Label Command and Event Messages

There are no command and event messages sent specifically to labels. Labels can, however, receive and handle several messages that are sent to most or all window items; see the section, “Window Item Command and Event Messages,” above.

Listboxes

A listbox displays non-editable text and/or artwork resources in one or more columns.

When there is not enough space to display the entire text of a listbox item, the text is truncated and an ellipsis added.

One or several items can be selected in various ways.

The **`contents`** property of a listbox describes the items that are to be displayed.

The optional **`form`** property of a listbox can customize the data and display formats of the listbox.

Individual items of the contents of listboxes may be accessed by a script using the **`listbox item`** class reference described after this section.

Properties of Listboxes

Listboxes have the properties shown here in addition to several properties they have in common with most or all window items; see the section, "Properties Common to All Window Items," at the beginning of this chapter.

column count

The number of columns of entries displayed in the listbox.

Value Class

integer

Examples

```
copy the column count of theObj to numCols
set the column count of listbox “lstCheese” of window “Dairy Goods to 3
```

Notes

➤ Listbox items (entries) are assigned as a single list, not as a list of lists. The given entries are simply displayed across and down. For example, if the listbox has a **`column count`** of 3, the first 3 listbox items are displayed in the first row, and so on.

➤ All listbox columns are of equal width.

➤ If you resize a multiple-column list, its width is resized in increments of the number of columns. For example, the width of a three-column list will change in increments of three pixels.

➤ By default, **`column count`** is 1, and the **`contents`** is “Listbox.”

doubleclick item

The button that receives a hilited message when any item of the listbox is double-clicked.

Value Class

reference	
constant	none

Examples

```
copy the doubleclick item of theObj to theDblItem
set the doubleclick item of listbox “lstFormats” to push button “pshOK”
```

Notes

➤ The **`doubleclick item`** must be a push button.

➤ The push button’s **`visible`** property must be **`true`**, but it can be located outside the window if you do not want it to be seen.

➤ If a listbox has a **`doubleclick item`**, double-clicking on the listbox itself is just like clicking the button.

➤ By default, **`doubleclick item`** is **none**.

form

The form of the listbox as defined by a form definition resource.

Value Class

constant	standard	(see the notes)
string	see notes	

Examples

```
copy the form of theObj to formName
set the form of listbox “lstGallery” to “ListOfNamedArt”
set form of listbox 12 to standard
```

Notes

➤ The default standard **`form`** for listboxes is built into FaceSpan. It supports only the display of text.

➤ Optional listbox forms (resources of type LDEF) can be imported into a project in the Forms View of the Project Window. These can support the display of icons and pictures, and have a variety of other features.

➤ A form might require the use of the **`format`** property.

➤ Many listbox forms require that listbox items themselves have special formats.

➤ To see basic documentation for a form, select its name in the Forms View of the Project Window, then click the Open button.

➤ For a detailed discussion of forms and formats, see the section entitled “Form Definition Resources and Formats” in this chapter.

format

A string of parameters for use by a form definition resource.

Value Class

string

Examples

```
set the format of window item 3 to theFormatString
```

Notes

➤ To see basic documentation for using the **`format`** property with a given form, first assign the form to the window item, then choose Format from the Properties popup in the Property Bar.

➤ Some forms do not require the use of the **`format`** property.

➤ For a detailed discussion of forms and formats, see the section entitled "Form Definition Resources and Formats" in this chapter.

key scrollable

Can the listbox be scrolled from the keyboard?

Value Class

boolean

Examples

```
copy the key scrollable of theObj to canScroll
set the key scrollable of listbox "lstToDo" of window "To Do Lists" to true
```

➤ If the key **`scrollable`** property of a listbox is true, the listbox can receive the focus just as an editable textbox does. Whenever the listbox has the focus, it responds to scrolling commands from the keyboard.

➤ You can scroll the list with the arrow keys.

➤ If the list is in alphabetical order, you can scroll the list by typing letters. The list scrolls to the entry that starts with (or is alphabetically nearest) the prefix you type.

➤ To suspend the normal time limit for scrolling by typing letters, hold down the Shift key while typing. (This is useful for those who are unable to type quickly.)

➤ When the listbox has the focus, it has a bold outline around it. This can be suppressed; see the **`margin`** property.

➤ The default value of **`key scrollable`** is **`false`**.

margin

The space between the border of a scrollable listbox and its bold focus outline.

Value Class

integer

Examples

```
set the margin of listbox “lstPartners” to 0
```

Notes

➤ A scrollable listbox acquires a bold selection outline when it has the focus.

➤ The default **`margin`** is 3.

➤ The bold selection outline can be suppressed by setting the **`margin`** to 0.

row count

Number of rows of entries contained in the listbox.

Value Class

integer

Examples

```
get row count of theObj
set totalRows to row count of listbox 3 of window 6
set row count of listbox 3 of window “Categories” to 4
```

Notes

➤ The value of **`row count`** is automatically updated as you create or delete listbox items.

➤ The number of rows is not the same as the number of entries. It is affected by the **`column count`**, and is the number of rows that can actually be counted as you scroll through the list.

➤ If you set the **`row count`** property of a listbox to a number less than the current number of rows in the listbox, the contents of the listbox is truncated after the specified number of rows; that is, the remaining listbox items are lost.

scroll

The number of rows that the content of a listbox has been scrolled.

Value Class

integer

Examples

```
copy the scroll of theObj to numRowsDown
set the scroll of listbox “lstGroceries” of window “To Do” to 12
```

Notes

➤ When the value of **`scrollable`** is **`false`**, the listbox still can be scrolled by clicking an entry and dragging up or down, so the **`scroll`** property still is meaningful.

➤ The **`scroll`** is 0 when the listbox is displaying the first item.

scrollable

Does the listbox have a scrollbar?

Value Class

boolean

Examples

```
copy the scrollable of theObj to itCanScroll
set scrollable of listbox “lstLanthanides” of window “Periodic Table” to false
```

Notes

➤ When the value of **`scrollable`** is **`false`**, the listbox still can be scrolled by clicking an entry and dragging up or down.

➤ When the **`scrollable`** property is set to **`true`**, the **`width`** of the listbox increases by 15 pixels; when set to **`false`**, the **`width`** decreases by 15 pixels.

➤ **`Scrollable`** is **`true`** by default.

selection

The indices of the listbox entries that are selected.

Value Class

list of integer	{selectedItem, selectedItem,…}

Examples

```
copy the selection of listbox “lstBase Metals” of window “Alchemy” to selList
set the selection of listbox “lstBase Metals” of window “Alchemy” to {2,3}
set the selection of listbox “lstLanthanides” to {}
set the selection of listbox “lstLetters” to {“beta”, “zeta”, “kappa”}
```

Notes

➤ If no listbox items are selected, the **`selection`** is the empty list, {}. You can deselect all the listbox items by setting the selection to {}.

➤ To select a certain item with a script, set the **`selection`** to a list containing the item’s index or its title.

➤ If the listbox is not full, the **`selection`** can be {}, since the listbox gets a **`selection made`** message even when the application user’s click deselects all entries.

➤ To get or replace the text of a selection within a listbox, use this reference form: **`contents of the selection of listbox`** “whatever.” This refers to a list of strings, each string being the contents of a selected listbox item.

➤ If you replace the **`contents of the selection`**, you can use a string in which the text of each entry is delimited with the return character. Thus, you can insert one or more entries at once.

➤ You can set the **`selection`** to one or more items with a list of their titles, as shown in the last example.

selection rule

The number of listbox items that can be selected, and how selection is done.

Value Class

constant	allow single	only one item can be selected
	allow any	a discontiguous group of items can be selected by holding down the Command key while clicking
	allow group	a contiguous group of items can be selected by dragging
	allow dragging	items can be dragged to reorganize a listbox
	none	no selection can be made

Examples

```
copy the selection rule of theObj to itsRule
set the selection rule of listbox “lstTitles” of window “CDs” to allow any
```

Note

➤ The default value of **`selection rule`** is **`allow any`**. This means that the selected items can be discontiguous.

Listbox Command and Event Messages

This section describes command and event messages that are sent specifically to listboxes.

Listboxes can also receive and handle several messages that are sent to most or all window items; see the section, “Window Item Command and Event Messages,” above.

clear

Edit menu command: Clears the contents of the selection of a listbox.

Parameters

(direct)	reference	the listbox

Examples

```
clear listbox “lstUtensils” of window “Morgue”
```

copy

Edit menu command: Copies the contents of the selection of a listbox to the Clipboard.

Parameters

(direct)	reference	the listbox

Examples

```
copy listbox “lstUtensils” of window “Morgue”
```

focus received

Event message sent when a listbox gets the focus.

Parameters

(direct)	reference	the listbox

Example

```
on focus received theObj
    copy name of theObj to thename
    set title of label “lblActiveItem” to thename
end focus received
```

Notes

➤ A listbox receives the focus when the application user clicks the listbox or tabs to it, or when a script sets the **`focus`** property of the window to the listbox.

➤ The **`focus received`** message is sent only to listboxes that are **`key scrollable`**.

keystroke

Event message sent when a key is pressed while the listbox has the focus.

Parameters

(direct)	reference	the listbox
key	long integer	the key code/character
[option down]	boolean	is Option key down?
[shift down]	boolean	is Shift key down?
[command down]	boolean	is Command key down?
[control down]	boolean	is Control key down?
[ticks]	integer	time (see notes)

Notes

➤ The **`keystroke`** message is received by a listbox only if it is **`key scrollable`**.

➤ The **`keystroke`** message may also be sent by a script to simulate pressing a key.

➤ For more information and an example of a **`keystroke`** handler, see the **`keystroke`** message discussion for textboxes later in this chapter.

scrolled

Event message sent when a listbox is scrolled.

Parameters

(direct)	reference	the listbox

Example

```
on scrolled theObj
    copy scroll of theObj to newscroll
    set scroll of window items 2 thru 5 to newscroll
end scrolled
```

Notes

➤ A listbox can be scrolled with its scrollbar (its **`scrollable`** property is **`true`**), or by clicking and dragging downward or upward.

➤ The **`scroll`** property tells how far down, in rows, the listbox has been scrolled.

selection made

Event message sent when the user selects a listbox item.

Parameters

(direct)	reference	the listbox

Example

```
on selection made theObj
   copy the contents of the selection of theObj to theSelection --a list
   copy item 1 of theSelection to theSelection --a string
   set the contents of label “lblSelection” to theSelection
end selection made
```

Notes

➤ The **`selection`** property tells which listbox items have been selected.

➤ If the listbox is not full, the **`selection`** can be {}, since the listbox gets a **`selection made`** message even when the application user’s click deselects all entries.

Listbox items

Individual items of the contents of listboxes may be specified using the listbox item class reference. When there is not enough space to display the entire text of a listbox item, the text is truncated and an ellipsis added.

Properties of Listbox Items

Listbox items have a very limited collection of properties, just those needed to refer to them and to get or set their contents. They are not actually “window items.”

contents

The text of a listbox item.

Value Class

string

Examples

```
copy the contents of listbox item i of listbox “lstToDo” to whatNow
set the contents of listbox item i of listbox “lstToDo” to whatNow & “done”
```

Notes

➤ The **`contents`** is the text of the entry only if the listbox uses the standard form. The **`contents`** can be a description of something to display, such as artwork, if the form is not **`standard`**.

➤ The **`contents`** property of a listbox item is the same as its **`name`** property.

index

The index of the listbox item within its listbox.

Value Class

integer

Table of Contents Index

Examples

```
copy the index of listbox item “Vegetables” of listbox “lstFoods” to veggieNum
--Can change the contents, and from now on use the index:
set the name of listbox item veggieNum of listbox “lstFoods” to vegInFrench
```

Note

➤ Listbox items are indexed sequentially from top to bottom within their listboxes. (Left to right, and top to bottom in listboxes with two or more columns.)

name

Same as the **`contents`** property of a listbox item.

Movies

Movies are containers in which QuickTime™ movie files can be displayed.

FaceSpan displays movies with or without a standard QuickTime™ controller, and provides various properties for the control of playing speed and volume.

The `editable` property determines whether a movie is selectable and responsive to Edit menu commands.

Properties of Movies

Movies have the properties shown here in addition to several properties they have in common with most or all window items; see the section, “Properties Common to All Window Items,” at the beginning of this chapter.

artwork

The QuickTime movie file displayed by a movie.

Value Class

alias	reference to file containing the movie
constant	none

Examples

```
copy (the artwork of theObj) as string to fullPathName
set artwork of movie 1 to alias “Centris HD:Bijou Folder:Circling Apple”
```

Notes

➤ Movie artwork is not copied into the project, so a movie object will not display any artwork if the artwork cannot be found when the project is run.

➤ If the application is distributed without movies, set the **`artwork`** of the movie to **`none`**. This will prevent prompts for movie files or disk volumes when the application is run on another computer.

editable

Are the frames of the movie selectable and editable?

Value Class

boolean

Examples

```
copy the editable of theObj to canDo
set editable of movie 3 of window “Locked” to false
```

Notes

➤ If **`editable`** is **`true`**, the movie can receive the focus. It receives the focus when a user tabs to it or clicks it, or when a script sets the window’s focus to the movie.

➤ A movie with the focus has a bold outline around it. This can be suppressed; see the **`margin`** property.

➤ **`Editable`** is **`false`** by default.

elapsed time

The position to which a movie has been advanced.

Value Class

integer

Examples

```
copy the elapsed time of movie “movMonsters” to timeSoFar
set the elapsed time of movie “movOverRover” to 2000
```

Notes

➤ The **`elapsed time`** property is expressed in units of the movie’s **`time scale`**, which typically is 600.

➤ The total time contributed by each movie frame is not necessarily the same for all frames. Therefore, setting the **`elapsed time`** to half the total does not necessarily display the middle frame.

➤ The **`elapsed time`** property is the same as the **`scroll`** property.

locked

The inverse of the **`editable`** property.

margin

The space between the border of an editable movie and its bold focus outline.

Value Class

integer

Examples

```
set the margin of movie “movTheWorld” to 0
```

Notes

➤ An **`editable`** movie acquires a bold selection outline when it has the focus.

➤ The bold selection outline can be suppressed by setting the **`margin`** to 0.

➤ The default **`margin`** is 3.

repeating

Will the movie automatically replay when it gets to the end?

Value Class

boolean

Examples

```
copy the repeating of theObj to itRepeats
set the repeating of movie “movLoopingTiresomely” to true
```

Note

➤ By default, **`repeating`** is **`false`**.

scroll

The **`scroll`** property is the same as the **`elapsed time`** property.

scrollable

Does the movie have a standard QuickTime controller?

Value Class

boolean

Examples

```
copy the scrollable of theObj to itScrolls
set the scrollable of movie “Porcupine” of window “Balloon Factory” to false
```

Notes

➤ The standard QuickTime controller lets the application user play, stop and scroll the movie.

➤ When the value of **`scrollable`** is **`false`**, the controller is missing, but the movie still can be played and otherwise controlled by scripts.

➤ The default value of **`scrollable`** is **`true`**.

selection

The portion of a movie that is selected.

Value Class

list of integer	{startingScroll, endingScroll}

Examples

```
copy the selection of theObj to {startTime, endTime}
set the selection of movie “Vertigo” of window “Alfred” to {1270, 1830}
```

Note

➤ The two values in the **`selection`** represent starting and ending elapsed times. See the discussion of the **`elapsed time`** property.

speed

Rate at which a movie is played.

Value Class

integer

Examples

```
copy the speed of theObj to itsSpeed
play movie “Sleep” speed 25
```

Notes

➤ The **`speed`** is expressed as a percentage of the movie’s normal speed.

➤ The **`speed`** property is read-only, but it is set indirectly as a parameter of the **`play`** command.

➤ **`Speed`** defaults to 100 (“”) of the normal speed).

time scale

The standard playing rate of a movie.

Value Class

integer

Examples

```
get time scale of theObj
copy time scale of movie 2 of window “TimeLapse” to movietimescale
set title of label “playspeed” to movietimescale
```

Notes

➤ The standard **`time scale`** is 600 (meaning 600 units per second).

➤ Both the **`selection`** property and the **`elapsed time`** property are expressed in terms of this scale.

➤ **`Time scale`** is a read-only property.

volume

Loudness level at which a movie will be played.

Value Class

integer

Examples

```
copy the volume of movie “Mumbling” to itsVol
set itsVol to itsVol/2
set the volume of movie “Mumbling” to itsVol
```

Notes

➤ The **`volume`** is expressed as a percentage of the movie’s normal volume. The normal volume was set by the person who created the movie.

➤ The default **`volume`** is 100 (% of the normal volume).

Movie Command and Event Messages

This section describes command and event messages that are sent specifically to movies.

Movies can also receive and handle several messages that are sent to most or all window items; see the section, “Window Item Command and Event Messages,” above.

clear

Edit menu command: Deletes the current selection of the movie’s content.

Parameters

(direct)	reference	the movie to be edited

Examples

```
clear movie “movBijou”
clear movie 1 of window “Cutting Room Floor”
```

Note

➤ If the movie is the window item that currently has the focus, the **`clear`** command without a direct parameter applies to that movie.

copy

Edit menu command: Copies the current selection of the movie’s content to the Clipboard.

Parameters

(direct)	reference	the movie to be edited

Examples

```
copy movie “movBijou”
copy movie 1 of window “Homage”
```

Note

➤ If the movie is the window item that currently has the focus, the **`copy`** command without a direct parameter applies to that movie.

cut

Edit menu command: Copies the current selection of the movie’s content to the Clipboard, then deletes it from the movie’s content.

Parameters

(direct)	reference	the movie to be edited

Examples

```
cut movie “movBijou”
cut movie 1 of window “Rushes”
```

Note

➤ If the movie is the window item that currently has the focus, the **`cut`** command without a direct parameter applies to that movie.

focus received

Event message sent when a movie gets the focus.

Parameters

(direct)	reference	the movie

Example

```
on focus received theObj
    global revertFrames
    copy contents of theObj to revertFrames
end focus received
```

Notes

➤ A movie receives the focus when the application user clicks the movie or tabs to it, or when a script sets the **`focus`** of the window to the movie.

➤ The **`focus received`** message is sent only to movies that are **`editable`**.

paste

Edit menu command: Pastes the current contents of the Clipboard into the movie’s content at the insertion point.

Parameters

(direct)	reference	the movie to be edited

Examples

```
paste theObj
paste movie 1 of window “Montage”
```

Notes

➤ If the movie is the window item that currently has the focus, the **`paste`** command without a direct parameter applies to that movie.

➤ If there is no movie in the Clipboard, nothing is pasted.

pause

Command to pause the movie.

Parameters

(direct)	reference	the movie

Examples

```
pause movie 1 of window “VisitOurSnackbar”
--Resume the movie from its current scroll position:
play movie 1 of window “VisitOurSnackbar”
```

play

Command to start or resume a movie.

Parameters

(direct)	reference	the movie
[speed]	integer	percent of normal speed

Examples

```
set the selection of movie “Tae Kwon Tofu” to {1200,1200} --food fight scene
play movie “Tae Kwon Tofu” speed 50 --slow the action
play movie “Two-Handed Ty Ping” --plays at normal speed
```

Notes

➤ To restart a movie from the beginning, set its **`scroll`**(**`elapsed time)`** property to 0.

➤ The **`speed`** is expressed as a percentage of the movie’s normal speed.

➤ Including the **`speed`** parameter in the **`play`** command is the only way to set a movie’s **`speed`** property.

Pictboxes

Pictboxes are containers in which pictures (PICTs) from the project's artwork resources, or the contents of PICT files, can be displayed.

A pictbox can simulate a button or an array of buttons. The number of buttons it simulates is determined by its **`selection grid`** property, the kind of button depends upon the **`selection rule`**, and the **`selection style`** determines the appearance of each simulated button when it is clicked.

A pictbox that simulates a button gets a **`hilited`** message like a button, while a pictbox divided into an array of buttons gets a **`selection made`** message like a listbox.

Pictboxes can be horizontally and vertically **`scrollable`**, and receive **`scrolled`** messages like a scrolling pane or textbox.

Pictboxes cannot be simultaneously scrollable and simulate buttons.

Properties of Pictboxes

Pictboxes have the properties shown here in addition to several properties they have in common with most or all window items; see the section, "Properties Common to All Window Items," at the beginning of this chapter.

artwork

The picture resource or file displayed by the pictbox.

Value Class

resource info	an artwork resource in the project
alias	a PICT file
constant	none

Table of Contents Index

Examples

```
copy the artwork of theObj to {type:itsType, id:itsID, name:itsName}
set the artwork of pictbox 2 to alias “Macintosh HD:Desktop Folder:Logo”
set artwork of pictbox “picPortrait” to {type:”PICT”,id:5003,name:”El Cid”}
```

Notes

➤ If the actual picture is smaller than the **`bounds`** of the pictbox item, it is centered. If the actual picture is too big, it is scaled to fit. If the pictbox is **`scrollable`**, and the actual picture is larger than the **`bounds`**, only a portion of the image is visible, but other portions can be scrolled into view.

➤ You can set **`artwork`** by specifying only the **`name`** or the **`id`**. It is best, however, to give values for all three properties of the **`resource info`** class to make the specification entirely unambiguous.

➤ If you set the **`artwork`** to an alias to a PICT file, the picture will be displayed immediately, and will remain as long as the window is open. The alias is not persistent.

➤ If the **`artwork`** property is set to **`none`**, the pictbox becomes transparent and may be positioned over another window item for use as a transparent button with user-definable highlighting behavior. For example, if you use such an object to cover a textbox that has several lines of text, when the user clicks it, the whole textbox can appear to highlight according to the pictbox’s highlight style and selection rule.

➤ For information about the **`resource info`** class, see Chapter 15: “Special Artwork and Text Style Classes.”

highlight

Same as the **`hilite`** property of a pictbox.

hilite

Is the pictbox hilited (has it been clicked)?

Value Class

boolean

Examples

```
copy the highlight of theObj to itsHilite
set the highlight of pictbox "picNick" of window "Lights On" to true
```

Note

➤ The **`hilite`** property is meaningful only for ungridded pictboxs (pictboxs whose **`selection grid`** is {1, 1}) that have a **`selection rule`** other than **none**. Other wise, the **`hilite`** property is always **`false`**.

hilite rule

Same as the **`selection rule`** property of a pictbox.

hilite style

Same as the **`selection style`** property of a pictbox.

justification

Determines how the pictbox is aligned.

Value Class

constant	**`left`**	aligns the picture to the top-left of the pictbox.
	`center`	aligns the picture to the horizontal and vertical center of the pictbox.
	`right`	aligns the picture to the bottom-right of the pictbox.

Example

```
on hilited theObj
   set justification of pictbox "picScaling" to left
end hilited
```

scale

Controls the magnification or reduction of the image in the pictbox.

Value Class

integer

Example

```
set scale of pictbox “picNamel” to 50
```

Notes

➤ The scale is expressed as a magnification of the image’s original size. For example, 100 is full-size, 200 is double size, etc.

➤ The constant standard, or value of 0, provides automatic scaling similar to FaceSpan’s 2.0 version. It differs from the 2.0 version in that the image’s aspect ratio (ratio of height to width) is now always respected so that images will no longer be distorted. Please note that this will cause some pictures to be displayed differently in 2.1 than in 2.0.

scroll

The distances that the content of a scrollable pictbox has been scrolled.

Value Class

list of integer	point	{horizScroll, vertScroll}

Examples

```
copy the scroll of theObj to {theH, theV}
set the scroll of pictbox “Nude Descending a Staircase” to thebottom
set scroll of theObj to {100,150}
```

Notes

➤ When the value of the **`scrollable`** property is **`false`**, the **`scroll`** property is always {0, 0}.

➤ Setting the **`scroll`** property does not send the pictbox a **`scrolled`** message; the message can be sent by a script, if necessary.

scrollable

Can the pictbox be scrolled (does it have scroll bars)?

Value Class

boolean

Examples

```
copy the scrollable of theObj to itIsScrolling
set the scrollable of pictbox “Jumbo” of  ¬
   window “Periodic Table of Elephants” to false
```

Notes

➤ When the value of **`scrollable`** is **`false`**, the value of the **`scroll`** property is always {0, 0}.

➤ When the **`scrollable`** property is set to **`true`** (usually in edit mode), the pictbox becomes 17 pixels wider and 17 pixels higher to accommodate the scroll bars. When the **`scrollable`** property is set to **`false`**, the pictbox becomes 17 pixels narrower and shorter. (The value 17 includes a border that is drawn around the pictbox.)

➤ **`Scrollable`** is **`false`** by default.

selection

A list of the indices of selected cells in a gridded pictbox.

Value Class

list of integer	{cellIndex, cellIndex, …}

Examples

```
copy the selection of theObj to selList
set the selection of pictbox “picGold” of window “Alchemy” to {2,3}
set the selection of theObj to {} --deselect all cells
```

Notes

➤ Pictbox cells are indexed sequentially, in reading order (left to right and top to bottom).

➤ The **`selection`** is the empty list when no cells are selected.

➤ The **`selection`** is the empty list when the pictbox is not gridded.

➤ If the **`selection rule`** is **`as push button`** or **`as radio button`**, then the **`selection`** is a list of one cell index.

➤ If the **`selection rule`** is **`as checkbox`**, then the selection can be the empty list (none selected), a list of one, or a list of many indices.

selection grid

The numbers of rows and columns that divide the pictbox into selectable cells.

Value Class

list of integer	point	{#OfColumns, #OfRows}

Examples

```
copy the selection grid of theObj to {numCols, numRows}
set the selection grid of pictbox "picOlympia" to {3,3}
```

Notes

➤ The default value of the **`selection grid`** property (an "ungridded pictbox") is {1,1}.

➤ A pictbox whose **`selection grid`** property is {1, 1} can act as a single button, and thus can receive a **`hilited`** message.

➤ Several pictboxes, each acting as a single **`radio button`**, will automatically act in tandem if they have consecutive indices.

➤ A pictbox whose **`selection grid`** property is greater than {1, 1} can act as an array of buttons, and thus can receive a **`selection made`** message.

➤ If **`selection rule`** set to **`none`**, the pictbox will not receive a **`hilited`** or **`selection made`** message when clicked.

➤ If **`selection style`** is set to **`none`**, the pictbox will not highlight when clicked.

➤ The Selection Grid popup of the Properties menu of the Property Bar shows the number of rows, then columns, even though the property is stored as columns and rows.

selection rule

Determines which button is imitated by an ungridded pictbox or by the cells of a gridded pictbox.

Value Class

constant	none	no selection can be made
	as push button	one pictbox or cell can be momentarily selected
	as radio button	one pictbox or cell can be persistently selected; other pictboxes or cells are deselected
	as checkbox	several pictboxes or cells can be persistently selected

Examples

```
copy the selection rule of pictbox “picAnyCard” to itsRule
set selection rule of pictbox 5 of window “Choices” to as push button
```

Notes

➤ If **`selection rule`** is set to **`none`**, the pictbox will not receive a **`hilited`** message or a **`selection made`** message when clicked.

➤ By default, **`selection rule`** is **`none`**.

➤ Several pictboxes, each acting as a single **`radio button`**, will automatically act in tandem if they have consecutive indexes.

selection style

Determines the visual transformation that a pictbox or gridded pictbox cell undergoes when clicked.

Value Class

constant	none	pictbox or cell does not highlight
	by hilite	white areas of pictbox or cell are overlain with the System highlight color (from the Color control panel)
	by invert	colors of pictbox or cell are inverted
	by lasso	colors of pictbox or cell are inverted within contours that exclude thepictbox's fill color
	by frame	pictbox or cell is surrounded by a frame
	by sink	pictbox or cell is surrounded by a frame that is indented by one column of pixels on the left and one row of pixels on the top
	by exchange	a different artwork resource is used for the highlighted pictbox

Examples

```
copy the highlight style of theObj to itsStyle
set highlight style of pictbox “Dogie” of window “Range” to by lasso
```

Notes

➤ You can specify that areas of a given color are highlighted. Set the **`fill color`** property of the pictbox to the color of that area.

➤ If **`selection style`** is set to **`none`**, the pictbox will not highlight or receive a **`hilited`** or **`selection made`** message when clicked.

➤ By default, **`selection style`** is **`none`**.

Pictbox Command and Event Messages

This section describes command and event messages that are sent specifically to pictboxes. Pictboxes can also receive and handle several messages that are sent to most or all window items; see the section, “Window Item Command and Event Messages,” above.

hilited

Event message sent when an ungridded pictbox with a selection rule is clicked.

Parameters

(direct)	reference	the pictbox

Examples

```
on hilited theObj
    copy index of theObj to thechoice
    display dialog “You clicked pictbox “ & (thechoice as string)
end hilited
```

Notes

➤ If an ungridded pictbox’s **`selection rule`** is set to none, or if its **`selection grid`** is greater than {1, 1}, it will not receive **`hilited`** messages.

➤ The **`hilited`** message is for ungridded pictboxes only.

➤ The **`hilited`** and **`selection made`** messages are mutually exclusive.

selection made

Event message sent when a gridded pictbox is clicked.

Parameters

(direct)	reference	the pictbox

Examples

```
on selection made theObj
    --Display a list of indices of the selected cells:
    copy selection of theObj to theSelect
    copy “none” to selectText
    repeat with i in theSelect
        if selectText = “none” then
          copy contents of i to selectText
        else
          copy selectText &”, “ & contents of i to selectText
        end if
    end repeat
    display dialog “Selected cells: “ & selectText
end selection made
```

Notes

➤ If a pictbox’s **`selection grid`** is {1, 1}, or if its **`selection rule`** property is set to **`none`**, it does not receive **`selection made`** messages.

➤ The **`selection made`** message is for gridded pictboxes only.

➤ The **`hilited`** and **`selection made`** messages are mutually exclusive.

➤ If the **`selection rule`** is **`as push button`** or **`as radio button`**, then the **`selection`** is a list of one cell index.

➤ If the **`selection rule`** is **`as checkbox`**, then the **`selection`** can be the empty list (none selected), a list of one, or a list of many indices.

Popups (Pop-up Menus)

A popup expands, when clicked, to display a menu of items in which the user drags to choose an item. Like radio button groups, popups permit the user to choose only one item from a group. Unlike radio buttons, only the user's choice is displayed after the application user releases the mouse button, and the user must click to expand the popup to review the other menu items available.

The **`form`** property of popups permits a variety of data and display formats.

Individual items of the contents of a popup can be specified using the **`menu item`** class reference. When there is not enough space to display the entire text of a popup item, the text is truncated and an ellipsis added.

Properties of Popups

Popups have the properties shown here in addition to several properties they have in common with most or all window items; see the section, "Properties Common to All Window Items", at the beginning of this chapter.

form

The form of the popup as defined by a form definition resource.

Value Class

constant	standard	(see notes)
string	see notes	

Examples

```
copy the form of theObj to itsFormName
set the form of popup “popColors” to “MenuOfColors”
set the form of popup 12 to standard
```

Table of Contents   Index

Notes

➤ The default **`form`** for a popup (a resource of type MDEF) is built into FaceSpan, it has the value **`standard`**.

➤ Optional popup forms can be imported into a project. These can support the display of icons and pictboxes, and have a variety of other features.

➤ A form might require the use of the **`format`** property.

➤ Many popup forms require that menu items themselves have special formats.

➤ To see basic documentation for a form, select its name in the Forms View of the Project Window, then click the Open button.

➤ For a detailed discussion of forms and formats, see the section entitled "Form Definition Resources and Formats" in this chapter.

format

A string of parameters for use by a form definition resource.

Value Class

string

Examples

```
set the format of window item 3 to theFormatString
```

Notes

➤ To see basic documentation for using the **`format`** property with a given form, first assign the form to the window item, then choose Format from the Properties popup in the Property Bar.

➤ Some forms do not require the use of the **`format`** property.

➤ Formats for popups are limited to at most 255 characters.

➤ For a detailed discussion of forms and formats, see the section entitled "Form Definition Resources and Formats" in this chapter.

popup item or menu item properties

Properties of the class **`menu item`** also apply to items of pop-up menus (popups) when you use this reference form: **`menu item`** i **`of popup`** "popWhatever." See Chapter 14: "Menus and Menu Items," for complete descriptions of the properties of the **`menu item`** class.

selection

Index of the item that is selected in the popup.

Value Class

list of integer	
string	contents of an item in the popup

Examples

```
copy the selection of theObj to theSel
set the selection of popup “popMetals” to “Lead”
set the selection of popup “pop”HelpTopics” to {2}
```

Notes

➤ The standard form for popups allows only one menu item to be selected. Its value is **`standard`**.

➤ The **`selection`** property is a list of one item. You can set the **`selection`** with a list containing an index or an item name (item contents).

➤ The **`contents of the selection`** is a string, not a list of one string.

➤ To get or set the contents of a specific menu item in the popup, use this reference form: **`menu item`** i **`of popup`** "popWhatever."

title item

The window item that will be highlighted when the popup is clicked.

Value Class

reference	window item to be highlighted
constant	none

Examples

```
get title item of window item 3 of window of theObj
set title item of popup 3 of window 4 to 12
copy selection of popup “Choices” to userchose
set title of title item of popup “Choices” to userchose
```

Notes

➤ The **`title item`** is hilited by inverting its colors.

➤ If the **`title item`** of a popup is set to the popup itself, the popup becomes invisible except while clicked. This allows you to overlay a popup on a pictbox or any window item to simulate popups with different shapes.

➤ The **`title item`** is **none** by default.

Popup Command and Event Messages

This section describes the only event message that is sent specifically to popups.

Popups can also receive and handle several messages that are sent to most or all window items; see the section, “Window Item Command and Event Messages,” above.

selection made

Event message sent when a menu item in the popup is selected.

Parameters

(direct)	reference	the popup

Example

```
on selection made theObj
    copy contents of selection of theObj to theSelection
    set title of label “lblSelection” to theSelection
end selection made
```

Notes

➤ The standard form for popups allows only one menu item to be selected.

➤ The `selection` property is a list of one item.

➤ You can set the `selection` with a list containing an index or an item name (item contents).

Push Buttons

Push buttons permit users to start and end processes in Macintosh windows. They are typically used to summon windows from an existing window, and are always used as the method of closure of modal dialogs.

FaceSpan automates the highlighting that occurs when push buttons are clicked, and provides a set of properties for extended control of push buttons:

➤ The **`auto close`** property permits a push button to close the window it occupies.

➤ The **`command key`** property permits a push button to be activated by a user-specified Command-key combination.

➤ The **`default item`** property draws a double outline around a push button, and permits its activation by the Return and Enter keys.

➤ The **`cancel item`** property permits a push button to be activated by the Escape and Command+Period keys.

Properties of Push Buttons

Push buttons have the properties shown here in addition to several properties they have in common with most or all window items; see the section, “Properties Common to All Window Items,” at the beginning of this chapter.

auto close

Does clicking the button close its window?

Value Class

boolean

Examples

```
set the auto close of push button “pshScram” of window “Reactor” to true
```

Table of Contents Index

Notes

➤ The **`auto close`** property usually is set in edit mode.

➤ When a button whose **`auto close`** property is **`true`** is used to close a window, the **`closing item`** property of the window contains a reference to the button.

➤ The **`auto close`** property may be **`true`** for any number of buttons in a window.

➤ **`Auto close`** is, by default, **`false`**.

cancel item

Does the button act like a standard Cancel button?

Value Class

boolean

Examples

```
set the cancel item of push button 3 of window “Document1” to true
```

Notes

➤ The button is automatically “clicked” when the application user presses Escape or Command-Period.

➤ The **`cancel item`** property usually is set in edit mode.

➤ The **`cancel item`** property can be **`true`** of only one button in each window.

command key

Defines the optional Command-key equivalent that activates the button.

Value Class

string	an alphanumeric character

Examples

```
set command key of push button “Error” to “E”
```

Notes

➤ The **`command key`** property usually is set in edit mode.

➤ Do not set the **`command key`** to a character that is already used as the command key of a menu item; the menu item takes precedence, so the button would be ignored.

➤ The default value of **`command key`** is the null string, meaning no command key is in effect.

default item

Does the button look and act like a standard OK button?

Value Class

boolean

Examples

```
set the default item of push button 3 of window “Document1” to true
```

Notes

➤ The button is automatically “clicked” when the application user presses Return or Enter.

➤ The **`default item`** property usually is set in edit mode.

➤ The **`default item`** property can be **`true`** of only one button in each window.

➤ The **`default item`** has a bold outline around it.

➤ The default value of **`default item`** is **`false`**.

form

The name of a form definition resource in the project.

Value Class

constant	standard	standard push button
string	see notes	

Examples

```
copy the form of theObj to itsForm
set the form of theObj to "3-D Push Button"
```

Notes

➤ The **`form`** property usually is set in edit mode.

➤ The default standard form for a push button (a resource of type CDEF) is built into FaceSpan. It has the value **`standard`**.

➤ Optional button forms can be imported into a project. These can support the display of icons and pictures, and have a variety of other features.

➤ A form might require the use of the **`format`** property.

➤ To see basic documentation for a form, select its name in the Forms View of the Project Window, then click the Open button.

➤ For a detailed discussion of forms and formats, see the section entitled "Form Definition Resources and Formats" in this chapter.

format

A string of parameters for use by a form definition resource.

Value Class

string

Examples

```
set the format of window item 3 to theFormatString
```

Notes

➤ To see basic documentation for using the **`format`** property with a given form, first assign the form to the window item, then choose Format from the Properties popup in the Property Bar.

➤ Some forms do not require the use of the **`format`** property.

➤ For a detailed discussion of forms and formats, see the section entitled "Form Definition Resources and Formats" in this chapter.

hilite

Is the push button hilited?

Value Class

boolean

Notes

➤ The **`hilite`** property of a push button is always **`false`**, because it immediately reverts to **`false`** after the **`hilited`** message is sent.

➤ The push button actually highlights when it is clicked, and stays highlighted as long as the mouse button is down and the cursor remains over the button. Only when the mouse button is released is the button sent the **`hilited`** message.

title

The text displayed by the push button.

Value Class

string

Examples

```
copy the title of theObj to itsTitle
set the title of push button 3 of window “Instructions” to “Click me”
```

Note

➤ If a script sets the **`title`** of a push button, it might also need to tell the push button to **`adjust size`** to fit the title.

Push Button Command and Event Messages

This section describes the only event messages that is sent specifically to push buttons.

Push buttons can also receive and handle several messages that are sent to most or all window items; see the section, “Window Item Command and Event Messages,” above.

hilited

Event message sent when the push button is clicked.

Parameters

(direct)	reference	the push button

Examples

```
on hilited of theObj
   set title of theObj to “I was clicked”
end hilited
```

Notes

➤ A push button receives the **`hilited`** message only once per click, when the mouse button is released.

➤ Since the push button’s **`hilite`** property changes only very briefly, it is always **`false`** when the **`hilited`** message is sent.

➤ A push button can receive a **`hilited`** message in response to certain keystrokes, as well as when clicked. Refer to the **`default item`**, **`cancel item`**, and **`command key`** properties for more information.

➤ A push button can be designated as the **`doubleclick item`** of a listbox or table. When an entry in the listbox or table is double-clicked, the button highlights and is sent the **`hilited`** message.

Radio Buttons

Radio buttons are displayed in groups of two or more to permit the user to select one choice from a group of choices.

When a radio button becomes hilited, FaceSpan automatically removes the hilite from all other buttons in the group.

FaceSpan considers any group of radio buttons having consecutive index numbers to be a group, and automatically coordinates their hilites.

Properties of Radio Buttons

Radio buttons have the properties shown here in addition to several properties they have in common with most or all window items; see the section, “Properties Common to All Window Items,” at the beginning of this chapter.

form

The name of a form definition resource.

Value Class

constant	standard	standard radio button
string	see notes	

Examples

```
copy the form of theObj itsForm
set the form of theObj to “Fancy Radio”
set the form of radio button 12 to standard
```

Notes

➤ The `form` of a radio button usually is set in edit mode.

➤ The default standard form for a radio button (a resource of type CDEF) is built into FaceSpan. It has the value `standard`.

Table of Contents   Index

➤ Optional button forms can be imported into a project. These can support the display of icons and pictures, and have a variety of other features.

➤ A form might require the use of the **`format`** property.

➤ To see basic documentation for a form, select its name in the Forms View of the Project Window, then click the Open button.

➤ For a detailed discussion of forms and formats, see the section entitled "Form Definition Resources and Formats" in this chapter.

format

A string of parameters for use by a form definition resource.

Value Class

string

Examples

```
set the format of window item 3 to theFormatString
```

Notes

➤ To see basic documentation for using the **`format`** property with a given form, first assign the form to the window item, then choose Format from the Properties popup in the Property Bar.

➤ Some forms do not require the use of the **`format`** property.

➤ For a detailed discussion of forms and formats, see the section entitled "Form Definition Resources and Formats" in this chapter.

highlight

Same as the **`hilite`** property of a radio button.

hilite

Does the radio button appear highlighted?

Value Class

boolean

Examples

```
copy the hilite of radio button “radMicrophone” to theMikeOption
set the hilite of radio button 3 of window “Runs Itself” to true
```

Notes

➤ The effect of clicking a radio button is to set its **`hilite`** to **`true`** and to set the **`hilite`** of others in the same group to **`false`**.

➤ A radio button does not receive a **`hilited`** message when its **`hilite`** is set to **`false`**.

➤ A radio button’s **`hilite`** is set before the **`hilited`** message is sent to it.

➤ The **`hilite`** of a radio button can be set from a script; doing so does not send it a **`hilited`** message.

links (radio buttons)

➤ You can link items in the window, such as buttons, checkboxes and so forth, to radio buttons. You can choose to enable, disable, hide or show the linked items.

To link items, double-click the radio button in the Window Editor to open its Object Information dialog box. Click the Links button to open the Links dialog box. Select the items you want to link from the list box and click the radio buttons to enable, disable, hide or show the linked items.

title

Text displayed by the radio button.

Value Class

string

Examples

```
copy the title of theObj to itsTitle
set the title of radio button “radSection” of window “Help” to “Section 3"
```

Note

➤ If a script sets the **`title`** of a radio button, it might also need to tell the radio button to **`adjust size`** to fit the title.

Radio Button Command and Event Messages

This section describes the only event message that is sent specifically to radio buttons.

Radio buttons can also receive and handle several messages that are sent to most or all window items; see the section, “Window Item Command and Event Messages,” above.

hilited

Event message sent when the radio button becomes hilited by a click.

Parameters

(direct)	reference	the radio button

Examples

```
property chosenRadio:”chosenRadio:””
on hilited of theObj
    set chosenRadio to name of theObj
end hilited
```

Notes

➤ A radio button receives a **`hilited`** message only when it becomes hilited, not when its **`hilite`** becomes **`false`**.

➤ A radio button’s **`hilite`** is set before the **`hilited`** message is sent to it.

Tab Panels

Tab panels are used to organize several different dialog boxes into one window. The user can click on the tabs to open the different dialog boxes.

Properties of Tab Panels

Tab panels have the properties shown here in addition to several properties they have in common with most or all window items. See *Common Properties* on page 277.

form

The name of a form definition resource.

Value Class

Constant	standard	kSmallTabs

Examples

```
set the form of tab panel 1 to standard
```

Scroll

The index of the tab panel

Value Class

integer

Examples

```
set the scroll of tab panel 1 to 3
```

Notes

Table of Contents   Index

➤ The scroll property is the current setting of the tab panel; it can be any integral value between the values of 1 and the number of tabs in the panel.

Tab Name

Value Class

list of string

Examples

```
set tab names of tab panel 1 to {“Name1,” “Name2,” “Name3”}
```

Tab Links

Value Class

list of list

Notes

➤ Tab panel links are automatic.

Tab Panels Command and Event Messages

This section describes only the event message that is sent specifically to Tab panels.

Tab panels can also receive and handle several messages that are sent to most or all window items. See *Window Item Command and Event Messages* on page 293.

scrolled

Event message sent when a tab panel is scrolled interactively.

Parameters

(direct)	reference	the listbox

Example

```
on scroll the Obj
    copy the scroll of theObj to tabSelected
    display dialog “The index of the selected Tab is: " & tabSelected
end scrolled
```

Note

➤ The **`scrolled`** message is not sent to the tab panel when its **`scroll`** is changed by a script; a script can send the **`scrolled`** message if necessary.

Tables

Table Objects are two-dimensional lists of textboxes called cells. Tables can have **`column titles`** and **`row titles`**, and these can be the standard lettering and numbering, or they can be any strings of your choosing.

When a table cell is editable, the **`cut`**, **`copy`**, **`paste`** and **`clear`** editing commands automatically apply to it.

You can make a table **`scrollable`** or not, in either direction.

You can have a table drawn with or without lines between the rows and columns.

The rows and columns can be resizable or not.

A table can be arbitrarily selectable, not selectable, or selectable only by rows or columns.

Each cell of a table can have a **`key filter`** and a corresponding **`format`** to control the entry and display of text.

Finally, every cell and title can have its own **`font`**, font **`size`**, **`style`**, **`fill color`** and **`pen color`**.

Reference Forms

Like menu items and listbox items, the elements of tables are themselves objects with properties. A table is a collection of row or column objects, and a row or column is a collection of cell objects. A table may also be considered a collection of cell objects. Thus, reference forms include:

➤ table 3

➤ rows of table 3

➤ columns of table 3

➤ cells of table 3

Properties of Tables

Tables have the properties shown here in addition to several properties they have in common with most or all window items; see “Properties Common to All Window Items,” at the beginning of this chapter.

Most cell and title properties (all the properties dealing with appearance) can be assigned to all cells at once by ascribing those properties to the whole table. Here are examples of all the cell and title properties that can be assigned collectively:

```
--Each of these sets the indicated property of every cell and title:
set the fill color of table 1 to {0, 65535, 65535}
set the font of table 1 to “Geneva”
set the justification of table 1 to right
set the pen color of table 1 to {65535, 0, 0}
set the size of table 1 to 10
set the style of table 1 to {on styles:italic}
set the uniform styles of table 1 to {on styles:italic}
```

The **`editable`** property can similarly be assigned to all cells at once (but not to titles); see the discussion of **`editable`**, below.

The rest of the properties listed here apply a table *as a single object*; the properties of rows, columns and cells are presented after the discussion of tables.

changing

Is a cell being edited?

Value Class

boolean

Examples

```
copy the changed of table “tblSales” to wasEdited
```

Notes

➤ If the **`changing`** property is **`true`**, then the **`changed`** message will be sent to the table when the application user tries to move the focus elsewhere.

➤ The default value of **`changing`** is false.

column count

The number of columns in the table.

Value Class

integer

Examples

```
set the column count of table “tblSales” to 8
set numCols to count of columns of table 1
set numCols to column count of table 1 --same as above
```

Notes

➤ The maximum **`column count`** of a table is 32,767.

➤ A newly-created table has a **`column count`** of 3.

column lines

Should a line separate each column?

Value Class

boolean

Examples

```
set the column lines of table “tblSales” to true
```

Notes

➤ The lines separating rows and columns are dotted lines drawn with the table’s pen color.

➤ By default, **`column lines`** is **`true`**.

Table of Contents Index

column titles

The titles of all the columns in the table.

Value Class

constant	none / standard / custom
list of string	

Examples

```
set the column titles of table'tblFlightList” to none
copy the column titles of table “tblSales” to theKind --constant
copy (the column titles of table “tblSales”) as list to theTitles --list of strings
copy theKind as list to theTitles --error: cannot coerce a constant
set the column titles of table 2 to {“Jan”, “Feb”, “Mar”, “Apr”}
```

Notes

➤ The usual value of **`column titles`** is simply a constant telling what kind of titles the columns have.

➤ The default value of **`column titles`**, **`standard`**, means that the column are labeled alphabetically.

➤ To hide the column titles, set **`column titles`** to **`none`**.

➤ If the table’s column titles are not displayed, the **`column titles`** property is **`none`**; otherwise, it is **`standard`** or **`custom`**.

➤ If you alter any column title, the **`column titles`** property will be **`custom`**.

➤ You can retrieve the actual title strings by coercing the value to a list while retrieving it, as shown in the examples. The constants, when taken alone, will not coerce.

➤ If the **`column titles`** value is **`none`**, asking for the strings returns a list of empty strings.

column widths

The widths of all the columns in the table.

Value Class

list of integer

Examples

```
set widLists to the column widths of table 1
set widLists to width of columns of table 1 --same as above
width of title of row 1 of table 1 --the column containing row titles
```

Notes

➤ **`Column widths`** includes the width of column zero, which contains the row titles.

➤ Get or set the width of a single column by way of the column's **`width`** property.

➤ The **`column width`** is 64 pixels by default. (The row titles are 32 pixels wide.)

contents

The text or other values of the cells of the table.

Value Class

list of list of string	cells hold strings by default
list of list of any	key filters can return any
string	(see notes)

Examples

```
copy the contents of table “tblSales” to myListOfLists
copy the contents of table 1 to {r1, r2, r3, r4, r5, r6} --assuming 6 rows

set theData to the contents of table 1
set theData to the contents of rows of table 1 --same as above
set theData to the contents of cells of table 1--same as above
set theData to the contents of cells of rows of table 1--same as above

set the contents of table 1 to {{“a”, “b”, “c”}, {“d”, “e”, “f”}, {“g”, “h”, “i”}, ¬
   {“j”, “k”, “l”}, {“m”, “n”, “o”}, {“p”, “q”, “r”}}
set the contents of table 1 to ¬
   “a\tb\tc\rd\te\tf\rg\th\ti\rj\tk\tl\rm\tn\to\rp\tq\tr” --same as above

set the contents of the selection of table 1 to ¬
   {{“1”, “2”}, {“3”, “4”}, {“5”, “6”}}
```

Notes

➤ The **`contents`** of the table or of a selection of the table can be set to a string in which cell values within rows are delimited by tab characters, and rows are delimited by return characters.

➤ The **`contents`** of the table or of a selection of the table can be retrieved as a tab-delimited and return-delimited string by coercing the value when it is retrieved, as shown in the examples.

➤ The **`contents`** of a cell can be set to a value of class **`resource info`** of a picture that has been imported into the project; it will display that picture.

➤ To get or set the **`contents`** of only the selected cells, use the reference form: **`contents of the selection`**.

➤ You can also get and set the **`contents`** of individual rows, columns and cells.

➤ When a cell has a **`key filter`** assigned to it, the key filter can determine the class of values that can be assigned to or retrieved from the cell. See “Form Definition Resources and Key Filters,” in this chapter.

➤ If you assign to a table more column or row values than it has columns or rows, new columns or rows will be created to hold the extra values.

➤ If you assign to a row, column or **`contents of the selection`** more values than will fit, the extra values will be ignored.

➤ If you assign to a table fewer column or row values than there are columns or rows, the unassigned cells are set to empty.

➤ If you assign to a row, column or contents of the selection fewer values than will fit, the unassigned cells are unaffected.

doubleclick item

The push button to be clicked when a cell is double-clicked.

Value Class

reference	a push button
constant	none

Examples

```
set the doubleclick item of table “tblSales” to push button “pshTotals”
```

Notes

➤ The **`doubleclick item`** must be a push button whose **`visible`** property is **`true`**, although the button may reside outside the visible area of the window.

➤ If the table is **`editable`**, double-clicking in a cell selects text; it does not click the **`doubleclick item`**.

➤ By default, **`doubleclick item`** is **`none`**.

editable

May the table’s cells be edited?

Value Class

boolean

Examples

```
set the editable of table “tblSales” to true
```

Notes

➤ **`Editable`** is really a property of cells. Setting the **`editable`** property of a table sets the **`editable`** properties of all the table's cells to the same value.

➤ When **`editable`** is **`true`**, the cursor is an I-beam over cells, and a plus-sign over column and row titles. When **`editable`** is **`false`**, the cursor is a plus-sign always.

➤ When **`editable`** is **`true`**, the text within cells can be selected for editing, but individual cells themselves cannot be selected.

➤ When **`editable`** is **`false`**, then the cell-selection method established by the **`selection rule`** property holds. When **`editable`** is **`true`**, cell selection can be accomplished only by clicking row or column titles.

➤ The **`editable`** property of a new table is **`false`**.

key scrollable

Does the table respond to keystrokes?

Value Class

boolean

Examples

```
set the key scrollable of table “tblSales” to true
```

Notes

➤ If **`key scrollable`** is **`true`**, the arrow keys can be used to navigate among the cells. The keys or combinations Tab, Shift-Tab, Return and Shift-Return also move right, left, down and up, respectively.

➤ If **`key scrollable`** is **`true`** but text (rather than a cell) is selected, then the navigation keys listed in the previous note do not navigate among cells. In fact the arrow keys assume their normal text-editing functionality.

➤ The **`key scrollable`** property of a new table is **`true`**.

resizable columns

May the application user resize the columns?

Value Class

boolean

Examples

```
set the resizable columns of table “tblSales” to false
```

Notes

➤ You can get and set the **`column widths`** property to save and restore the current column widths, if needed.

➤ **`Resizable columns`** defaults to **`true`**.

resizable rows

May the application user resize the rows?

Value Class

boolean

Examples

```
set the resizable rows of table “tblSales”
```

Notes

➤ You can get and set the **`row widths`** property to save and restore the current row widths, if needed.

➤ **`Resizable rows`** defaults to **`false`**.

row count

The number of rows in the table.

Value Class

integer

Table of Contents Index

Examples

```
set the row count of table “tblSales” to 1024
set numRows to the row count of table “tblExpenses”
set numRows to the count of rows of table “tblExpenses” --same as above
```

Notes

- ➤ The maximum row count is 32,767.
- ➤ The default row count is 6.

row heights

The heights of all the rows in the table.

Value Class

list of integer

Examples

```
copy the row heights of table “tblSales” to theHeights
copy the height of the rows of table “tblSales” to theHeights --same as above
```

Notes

- ➤ **`Row heights`** includes the height of the column titles.
- ➤ Get or set the height of a single row by way of the row’s **`height`** property.
- ➤ The default heights of all rows are 17 pixels.

row lines

Should a line separate each row?

Value Class

boolean

Examples

```
set the row lines of table “tblSales” to false
```

Notes

➤ The lines separating the rows are dotted lines drawn in the table's pen color.

➤ By default, the **`row lines`** property is **`true`**.

row titles

The titles of all the rows in the table.

Value Class

constant	none / standard / custom
list of string	

Examples

```
set the row titles of table 'tblFlightList" to none
copy the row titles of table "tblSales" to theKind --constant
copy (the row titles of table "tblSales") as list to theTitle --list of strings
copy theKind as list to theTitle --error
set the row titles of table 2 to {"Angie", "Betty", "Doris", "Grace"}
```

Notes

➤ The usual value of **`row titles`** is simply a constant telling what kind of titles the rows have.

➤ The default value of **`row titles`**, **`standard`**, means that the rows are numbered.

➤ To hide the row titles, set **`row titles`** to **`none`**.

➤ If the table's row titles are not displayed, the **`row titles`** property is **`none`**; otherwise, it is **`standard`** or **`custom`**.

➤ If you alter any row title, the **`row titles`** property will be **`custom`**.

➤ You can retrieve the actual title strings by coercing the value to a list while retrieving it, as shown in the examples. The constants, when taken alone, will not coerce.

➤ If the **`row titles`** value is **`none`**, asking for the strings returns a list of empty strings.

Table of Contents Index

scroll

The row and column positions of the contents of the table.

Value Class

point	{leftmostColumn, topmostRow}

Examples

```
copy the scroll of table “tblSales” to {theRow, theCol}
set the scroll of table 2 to {6, 20}
```

Notes

➤ The **`scroll`** of a table can be defined as the row and column coordinates of the cell that occupies the upper-left corner of the content area (the area not including the titles).

➤ The **`scroll`** of an unscrolled table is {1,1}.

scrollable across

Does the table have a horizontal scrollbar?

Value Class

boolean

Examples

```
set the scrollable across of table “tblPhoneList” to false
```

Notes

➤ When **`scrollable across`** is set to **`true`**, the **`height`** of the table increases 15 pixels to accommodate the horizontal scrollbar; when set to false the **`height`** decreases 15 pixels.

➤ **`Scrollable across`** is **`true`** by default.

scrollable down

Does the table have a vertical scrollbar?

Value Class

boolean

Examples

```
set the scrollable down of table “tblQuarterlies” to false
```

Notes

➤ When **`scrollable down`** is set to **`true`**, the **`width`** of the table increases 15 pixels to accommodate the vertical scrollbar; when set to **`false`** the **`width`** decreases 15 pixels.

➤ **`Scrollable across`** defaults to **`true`**.

selection

The selection of the table.

Value Class

list of integer	{startCol, startRow, endCol, endRow}

Examples

```
copy the selection of table “tblSales” to {sC, sR, eC, eR}
set the selection of table “tblSales” to {sC, sR, eC, eR+1} --1 more row
```

Notes

➤ The **`selection`** property defines a rectangle bounding the selected cells.

➤ To get or set the actual values that are selected, use the reference form: **`contents of the selection`**.

➤ If no cells are selected, the **`selection`** is {0, 0, 0, 0}.

➤ You can deselect all cells by setting the **`selection`** to {0, 0, 0, 0}.

selection rule

The manner in which cells can be selected.

Value Class

constant	allow single	one cell only
	allow group	any rectangular group
	allow rows	one or more rows only
	allow columns	one or more columns only
	allow one row	one row only
	allow one column	only column only
	none	no cells at all

Examples

```
set the selection rule of table “tblPhoneList” to allow rows
```

Notes

➤ The selection rules **`allow rows`**, **`allow columns`**, **`allow one row`** and **`allow one column`** force an entire row or column to become selected when the application user clicks a single cell.

➤ The **`selection rule`** defaults to **`allow group`**.

Table Command and Event Messages

This section describes command and event messages that are sent specifically to tables.

Note that the elements of a table—cells, rows and columns—do not receive messages; they do not have scripts, so all messages are handled in the table’s script.

Tables can also receive and handle several messages that are sent to most or all window items; see “Window Item Command and Event Messages,” above.

changed

Event sent after a cell has been edited, and is about to lose the focus.

Parameters

(direct)	reference	the table

Examples

```
on changed theObj
    if not (valid of cell phoneLoc of theObj) then
        display dialog “Sorry, that value is not correct. Please retry.”
        return invalid
    end if
end changed
```

Notes

➤ A cell of a table loses the focus when the interactive user tabs away from it, clicks another editable window item, or closes the window. That is when the **`changed`** message is sent, if the cell has been changed.

➤ If any cell has a **`key filter`** assigned to it, the **`valid`** property of that cell should be checked in the **`changed`** message handler. See the **`valid`** property.

➤ You can prevent the focus from moving away from a changed cell by issuing the **`return invalid`** command.

focus received

Event sent after the window’s **`focus`** is set to refer to the table.

Parameters

(direct)	reference	the table

Table of Contents Index

Examples

```
on focus received theObj
    --Scroll the table back to show the first cell:
    set the scroll of theObj to {1, 1}
end focus received
```

Notes

➤ A table can receive the focus only when its **`editable`** or **`key scrollable`** property is **`true`**.

➤ It receives the focus when the application user clicks it or tabs to it from another window item. If it is the only window item that can get the focus, it does so when the window is opened.

➤ A script can set the **`focus`** of the window to a table.

scrolled

Event sent after the table is scrolled by the application user.

Parameters

(direct)	reference	the table

Examples

```
on scrolled theObj
    --Keep the upper-left cell of the table selected:
    set {colNum, rowNum} to the scroll of theObj
    set the selection of theObj to {colNum, rowNum, colNum, rowNum}
end scrolled
```

Notes

➤ A table can be scrolled with the scroll bars, or by clicking a cell and dragging to make a selection.

➤ A table cannot be scrolled by dragging if it has no scroll bars.

➤ Setting the **`scroll`** property from a script does not send a **`scrolled`** message. The script can send a **`scrolled`** message if needed.

selection made

Event sent after one or more cells of the table have been selected.

Parameters

(direct)	reference	the table

Examples

```
on selection made theObj
   set the font of the selection of table 1 to “Geneva”
end selection made
```

Notes

➤ The **`selection made`** message is sent only when the application user selects cells.

➤ The **`selection made`** message is not sent when a script sets the selection. The message can be sent from the script if needed.

Rows of Tables

Rows, like columns, are not window items; they are elements of tables. Rows are themselves objects with properties. In addition, rows are containers; they contain cells.

Reference Forms

You can refer to a row by its index or name, or as a collection of cells. Here are examples of row references:

➤ row 1 of table 4

➤ row “Totals” of table “tblSales”

➤ cells of row 3

Properties of Rows

Almost all the properties affecting the appearance or contents of the cells of a row are, in fact, cell properties or properties of the row’s title.

You can get or set many cell properties as if they were properties of rows. In most cases, ascribing a cell property to a row means all the cells in that row. Thus, this statement:

```
set the editable of row “Totals” of table “tblSales” to false
```

sets the **`editable`** property of every cell in the row to false. But when you retrieve the property, the result is a list of values:

```
set editableList to the editable of row “Totals” of table “tblSales”
-->{false, false, false, false}
```

This statement acknowledges that the **`editable`** property actually belongs to the cells, and assigns each its own value:

```
set the editable of row 4 of table 1 to {false, true, false, true, true}
```

The foregoing statements apply to every cell property except **`contents`**. If you set the **`contents`** of a row to a single value, only the **`contents`** of the first cell is changed. See the discussion of the **`contents`** property for examples.

Rows do have five properties of their own: **`index`**, **`name`**, **`title`**, **`visible`** and **`width`**. Of these, **`title`** behaves as if it were an object with its own properties; see the discussion of **`title`** for more information.

contents

The values of the cells of the row.

Value Class

list of string	(the default)
list of any	(depending upon key filters)

Examples

```
set theC to contents of row “McMullen” of table “tblSales”
set theC to cells of row “McMullen” of table “tblSales” --same effect as above

set the contents of row 3 of table “tblQuarterlies” to ¬
   {“12,500”, “14,000”, “13,750”, “16,550”}
set the contents of row 3 of table “tblQuarterlies” to ¬
   “12,500\t14,000\t13,750\t16,550” --same effect as above

set the contents of row “Totals” to 27 --single value assigned to first cell
```

Notes

➤ **`Contents`** is a cell property, not a row or column property. This example shows that rows act as if they have that property.

➤ In the case of the **`contents`** property, a list of values must be assigned if you wish to assign to all cells in the row; assigning a single value merely sets the first cell.

➤ The default value class of the **`contents`** of a table cell is **`string`**, so a list of strings will fill a row of cells.

➤ If all cells accept strings, a single string, with cell values delimited by tab characters, can be assigned instead of a list.

Table of Contents Index

➤ When a cell has a **`key filter`** assigned to it, the key filter can determine the class of values that can be assigned to, or retrieved from, the cell. Thus, the list assigned to, or retrieved from, a row of cells might include a mixture of value classes. See “Form Definition Resources and Key Filters,” in this chapter.

height

The height in pixels of the row (and all its cells).

Value Class

list of integer

Examples

```
copy the height of row “Totals” to Ht
set the height of row “Totals” to ht*2 --double its height
```

Note

➤ The default **`height`** of a row is 17 pixels.

index

The index of the row within the table.

Value Class

integer

Examples

```
set the Ind to the index of row “Totals” of table “Taxes”
```

Notes

➤ The **`index`** of the top row of cells is 1.

➤ The **`index`** is a read-only property.

name

The name of the row.

Value Class

string

Examples

```
set the name of row 10 of table “Taxes” to “Deductions”
copy the name of row 5 of table 3 to itsName
```

Notes

➤ The **`name`** property is shorthand for **`contents of the title`**.

title

The title of the row.

Value Class

string	
object	(see notes)

Examples

```
set the contents of the title of row 4 of table 1 to “1”
set the fill color of the title of row 4 of table 1 to {56797, 56797, 56797}
set the font of the title of row 4 of table 1 to “Chicago”
set the justification of the title of row 4 of table 1 to center
set the pen color of the title of row 4 of table 1 to black
set the size of the title of row 4 of table 1 to 12
set the style of the title of row 4 of table 1 to {on styles:bold}
set the uniform styles of the title of row 4 of table 1 to {on styles:bold}
set the width of the title of row 1 of table 1 to 32

set the justification of title of rows of table 1 to left --sets all row titles
```

Notes

➤ The **`title`** property acts as if it is an object with its own properties, as seen in the examples. All the possible properties are shown, as if assigning their default values.

➤ The term **`title of row`** is shorthand for **`contents of title of row`**.

➤ Setting the **`width`** of any row title sets the width of all.

➤ The last example shows the use of **`title`** in a collective assignment.

➤ If you assign a new **`contents`** to a row **`title`**, it will not display until you have set the table's **`row titles`** property to **`custom`**. Setting **`row titles`** back to **`standard`** hides them, but they still exist.

➤ If you assign a new **`contents`** to a row **`title`**, then you need to assign all the row titles, or they will be blank.

visible

Is the row (its title and cells) visible?

Value Class

boolean

Examples

```
set the visible of row “Deductions” of table “Taxes” to false
```

Note

➤ When the **`visible`** of a row is **`false`**, the table closes up as if the row did not exist. It does exist, and so operations upon its title and cells can continue.

Columns

Columns, like rows, are not window items; they are elements of tables. Columns are themselves objects with properties. In addition, they are containers; they contain cells.

Reference Forms

You can refer to a column by its index or name, or as a collection of cells. Here are examples of column references:

- column 7 of table “Mesa”
- column “Expenses” of table “Taxes”
- cells of column 3

Properties of Columns

Almost all the properties affecting the appearance or contents of the cells of a column are, in fact, cell properties or properties of the column’s title.

You can get or set many cell properties as if they were properties of columns. In most cases, ascribing a cell property to a column means all the cells in that column. Thus, this statement:

```
set the editable of column “Tax” of table “tblSales” to false
```

sets the **`editable`** property of every cell in the column to false. But when you retrieve the property, the result is a list of values:

```
set editableList to the editable of column “Tax” of table “tblSales”
   -->{false, false, false, false}
```

This statement acknowledges that the **`editable`** property actually belongs to the cells, and assigns each its own value:

```
set the editable of column 7 of table 1 to {false, true, false, true, true}
```

The foregoing statements apply to every cell property except **`contents`**. If you set the **`contents`** of a column to a single value, only the **`contents`** of the first cell is changed. See the discussion of the **`contents`** property for examples.

Columns do have five properties of their own: **`index`**, **`name`**, **`title`**, **`visible`** and **`width`**. Of these, **`title`** behaves as if it were an object with its own properties; see the discussion of **`title`** for more information.

contents

The values of the cells of the column.

Value Class

list of string	(the default)
list of any	(depending upon key filters)

Examples

```
set theC to contents of column “January” of table “tblSales”
set theC to cells of column “January” of table “tblSales” --same as above

set the contents of column 3 of table “tblAttendance” to ¬
   {“12,500”, “14,000”, “13,750”, “16,550”}
set the contents of column 3 of table “tblAttendance” to ¬
   “12,500\t14,000\t13,750\t16,550”  --same effect as above

set the contents of column “Late” to 0 --single value assigned to first cell
```

Notes

➤ **`Contents`** is a cell property, not a row or column property. This example shows that columns act as if they have that property.

➤ In the case of the **`contents`** property, a list of values must be assigned if you wish to assign to all cells in the column; assigning a single value merely sets the first cell.

➤ The default value class of the **`contents`** of a table cell is **`string`**, so a list of strings will fill a column of cells.

➤ If all cells accept strings, a single string, with cell values delimited by return characters, can be assigned instead of a list.

➤ When a cell has a **`key filter`** assigned to it, the key filter can determine the class of values that can be assigned to, or retrieved from, the cell. Thus, the list assigned to, or retrieved from, a column of cells might include a mixture of value classes. See “Form Definition Resources and Key Filters,” in this chapter.

index

The index of the column.

Value Class

integer

Examples

```
set theInd to the index of column “Excise” of table “Taxes”
```

Notes

➤ The **`index`** of the left column of cells is 1.

➤ The **`index`** is a read-only property.

name

The name of the column.

Value Class

string

Examples

```
set the name of column 10 of table “Taxes” to “End of Year”
copy the name of column 7 of table 3 to itsName
```

Note

➤ The **`name`** property is shorthand for **`contents of the title`**.

Table of Contents Index

title

The title of the column.

Value Class

string	
object	(see notes)

Examples

```
set the contents of the title of column 4 of table 1 to “A”
set the fill color of the title of column 4 of table 1 to {56797, 56797,
56797}
set the font of the title of column 4 of table 1 to “Chicago”
set the height of the title of column 1 of table 1 to 17
set the justification of the title of column 4 of table 1 to center
set the pen color of the title of column 4 of table 1 to black
set the size of the title of column 4 of table 1 to 12
set the style of the title of column 4 of table 1 to {on styles:bold}
set the uniform styles of the title of column 4 of table 1 to {on styles:bold}

set the justification of title of columns of table 1 to right --sets all col titles
```

Notes

➤ The **`title`** property acts as if it is an object with its own properties, as seen in the examples. All the possible properties are shown, as if assigning their default values.

➤ The term **`title of column`** is shorthand for **`contents of title of column`**.

➤ Setting the **`height`** of any column title sets the height of all.

➤ The last example shows the use of **`title`** in a collective assignment.

➤ If you assign a new **`contents`** to a column **`title`**, it will not display until you have set the table’s **`column titles`** property to **`custom`**. Setting **`column titles`** back to **`standard`** hides them, but they still exist.

➤ If you assign a new **`contents`** to a column **`title`**, then you need to assign all the column titles, or they will be blank.

visible

Is the column (its title and cells) visible?

Value Class

boolean

Examples

```
set the visible of column “Benefits” of table “Taxes” to false
```

Note

➤ When the **`visible`** of a column is **`false`**, the table closes up as if the column did not exist. It does exist, and so operations upon its title and cells can continue.

width

The width in pixels of the column (and all its cells).

Value Class

list of integer

Examples

```
set the width of column “Debts” of table “Taxes” to 128
```

Note

➤ The default **`width`** of a column is 64.

Cells

Cells, like rows and columns, are not window items; they are elements of tables. Cells are themselves objects with properties.

Reference Forms

You can refer to a cell by its index or name, or by its position within a row or column. Here are examples of cell references; in fact, all refer to the same cell:

- ➤ cell {3, 6} of table “Mesa”
- ➤ cell 3 of row 6 of table “Mesa”
- ➤ cell 6 of column 3 of table “Taxes”
- ➤ cell “C6” of table “Mesa” --if standard titles

Properties of Cells

Almost all the properties affecting the appearance or contents of a table are, in fact, cell properties (or properties of the column and row titles).

You can get or set cell properties as if they were properties of columns, rows or whole tables. In most cases, ascribing a cell property to a column, row or table means all the cells in that column, row or table. See the discussion in “Properties of Columns,” above.

Conversely, you can get and set three column and row properties as if they were properties of cells. For example, the statement:

```
set the width of cell {2, 4} of table 1 to 64
```

sets the **`width`** of the cell’s column (column 2) to 64 pixels. A similar effect occurs with the **`height`** property (of rows), while setting the **`visible`** of a cell to **`false`** hides both its row and column.

Almost all cell properties must be set using scripts, either from the Message Winded or at run time.

contents

The value of the cell.

Value Class

text	the default contents
any	returned by a key filter

Examples

```
set the key filter of cell 4 of column 5 of table 1 to “DisplayDates
set the format of cell 4 of column 5 of table 1 to “Both Date and Time”
set the contents of cell 4 of column 5 of table 1 to current date

copy the contents of cell 4 of column 5 of table 1 to myData --a date value
copy (the contents of cell 4 of column 5 of table 1) as text to dateStr --string
```

Notes

➤ The default value class of the **`contents`** of a cell is text.

➤ When a cell has a **`key filter`** assigned to it, the key filter can determine the class of values that can be assigned to or retrieved from the cell. See “Form Definition Resources and Key Filters,” in this chapter.

editable

May this cell be edited?

Value Class

boolean

Examples

```
set the editable of cell “B3” to false
set the editable of row “Totals” of table “tblSales” to false --all cells in row
set the editable of row 4 of table 1 to {false, true, false, true, true}
set the editable of table “Taxes” to false
```

Notes

➤ The **`editable`** property of a single cell or of all cells in a row, column or table can be set at once. Examples of setting by cell, row and table are shown.

➤ By default the **`editable`** of every cell in a table is **`false`**.

fill color

The fill color of the cell.

Value Class

RGB color	{redValue, greenValue, blueValue}
integer	index to color in System color lookup table
constant	black / white

Examples

```
set myFill to {26214, 65535, 65535}

set the fill color of cell “C6” to myFill
set the fill color of row “Totals” of table “tblSales” to myFill --all cells
set the fill color of row 7 of table 4 to {myFill, white, myFill} --each cell
```

Notes

➤ The **`fill color`** property of a single cell or of all cells in a row, column or table can be set at once. Examples of setting by cell and by row are shown.

➤ By default the **`fill color`** of every cell in a table is **`white`**.

font

The font of the cell’s contents.

Value Class

string

Table of Contents   Index

Examples

```
set the font of cell {3, 6} to “Palatino”
set the font of row “Totals” of table “tblSales” to “Palatino” --all cells
set the font of row 7 of table 4 to {“Geneva”, “Chicago”, “Geneva”} --each cell
```

Notes

➤ The **`font`** property of a single cell or of all cells in a row, column or table can be set at once. Examples of setting by cell and by row are shown.

➤ By default the **`font`** of every cell in a table is “Chicago.”

format

Parameters for use by a key filter (form) definition resource.

Value Class

string

Examples

```
set the format of cell 4 of column 5 of table 1 to “DisplayDates”
```

Notes

➤ To see basic documentation for using the **`format`** property with a given **`key filter`**, first assign the key filter to the cell, then choose Cell’s Format from the Properties popup in the Property Bar.

➤ Some key filters do not require the use of the **`format`** property. See “Form Definition Resources and Key Filters,” in this chapter.

➤ By default the **`format`** property is the empty string.

index

The index of the cell itself.

Value Class

list of integer	point	{column#, row#}

Examples

```
copy the index of cell “JanuaryTotals” to cellLocus
```

Notes

➤ In the example, the cell resides in row “Totals” and column “January.” See the **`name`** property.

➤ A cell’s **`index`** is persistent, while its **`name`** can change.

➤ **`Index`** is a read-only property.

justification

The justification of the cell’s contents.

Value Class

constant	left / right / center

Examples

```
set the justification of cell {3, 6} to right
set the justification of row “Totals” of table “tblSales” to right --all cells
set the justification of row 7 of table 4 to {right, left, center} --each cell
```

Notes

➤ The **`justification`** property of a single cell or of all cells in a row, column or table can be set at once. Examples of setting by cell and by row are shown.

➤ By default the **`justification`** of every cell in a table is **`left`**.

key filter

A form that controls the entry of characters into the cell.

Value Class

constant	none	no key filter is used
string		key filter name

Examples

```
set the key filter of cell 4 of column 5 of table 1 to “onlyDigits”
```

Notes

➤ The **`key filter`** property normally is set in edit mode, during the design of the window.

➤ A key filter might require the use of the **`format`** property.

➤ The key filter, if any, determines the value of the **`valid`** property.

➤ You can find out how to use a key filter by “opening” it in the Forms View of the Project Window.

➤ When a cell has a **`key filter`** assigned to it, the key filter can determine the class of values that can be assigned to or retrieved from the cell.

➤ Key filters are not built into FaceSpan, but can be imported into a project. See “Form Definition Resources and Key Filters,” in this chapter.

name

The cell’s column and row titles, concatenated.

Value Class

string

Examples

```
copy the name of cell {3,6} to itsName
```

Notes

➤ The **`name`** can be used to refer to the cell.

➤ If the **`row titles`** or **`column titles`** of the table are changed, the **`name`** of each cell changes. (The **`index`** is persistent.)

➤ The **`name`** property is read-only.

pen color

The pen color of the cell.

Value Class

RGB color	{redValue, greenValue, blueValue}
integer	index to color in System color lookup table
constant	black / white

Examples

```
set myPen to {26214, 65535, 65535}

set the pen color of cell “C6” to myPen
set the pen color of row “Totals” of table “tblSales” to myPen --all cells
set the pen color of row 7 of table 4 to {myPen, black, myPen} --each cell
```

Notes

➤ The **`pen color`** property of a single cell or of all cells in a row, column or table can be set at once. Examples of setting by cell and by row are shown.

➤ By default the **`pen color`** of every cell in a table is **`black`**.

size

The text size—in points—of the cell’s contents.

Value Class

integer

Examples

```
set the size of cell “C6” to 18
set the size of row “Totals” of table “tblSales” to 18 --all cells
set the size of row 7 of table 4 to {18, 12, 18} --each cell
```

Notes

➤ The **`size`** property of a single cell or of all cells in a row, column or table can be set at once. Examples of setting by cell and by row are shown.

➤ By default the **`size`** of every cell in a table is 12.

style

The text style of the cell's contents.

Value Class

text style info

Examples

```
set myStyle to {on styles: bold}

set the style of cell “C6” to myStyle
set the style of row “Totals” of table “tblSales” to myStyle --all cells
```

Notes

➤ The **`style`** property of a single cell or of all cells in a row, column or table can be set at once. Examples of setting by cell and by row are shown.

➤ By default the **`style`** of every cell in a table is **`plain`**.

uniform styles

The text styles that are uniform.

Value Class

text style info

Examples

```
set myUS to {on styles: bold}

set the uniform styles of cell “C6” to myUS
set the uniform styles of row “Totals” of table “tblSales” to myUS --all cells
```

Notes

➤ The **`uniform styles`** property of a single cell or of all cells in a row, column or table can be set at once. Examples of setting by cell and by row are shown.

➤ By default the **`uniform styles`** of every cell in a table is **`plain`**.

valid

Is the text in the cell correct according to the key filter?

Value Class

boolean	
list of integer	(see notes)

Examples

```
copy the valid of cell 4 of column 5 of table 1 to itsOK
copy (the valid of cell 4 of column 5 of table 1) as list to {startBad, endBad}
```

Notes

➤ Asking for the value of the **`valid`** property causes a call to the key filter of the cell; the key filter checks the validity of the cell at that moment.

➤ If the **`valid`** property is **`false`**, asking for it as a list (as shown in the examples) returns the selection of the first group of invalid characters in the textbox. If **`valid`** was true, the list returned is {0, 0}.

➤ The **`valid`** property is read-only.

Textboxes

Textboxes are containers for variable amounts of text. They may be editable and scrollable.

The text styles of any portion of the text in a textbox can be precisely controlled, as can the fonts sizes, line heights and colors.

Text in textboxes can be manipulated using the standard Text Suite of objects and properties. See the discussion of the Text Suite in this chapter.

Key filters can be applied to editable textboxes to control the entry of characters. (Key filters are form definition resources for textboxes. They are not built into FaceSpan, but can be added to projects.)

Properties of Textboxes

Textboxes have the properties shown here in addition to several properties they have in common with most or all window items; see the section, “Properties Common to All Window Items,” at the beginning of this chapter.

changing

Has the textbox been changed?

Value Class

boolean

Examples

```
on changed theObj
    copy contents of theObj to theValue
    if theValue < 1 or theValue > 9 then
        beep
        display dialog “Please enter a value between 1 and 9." ¬
           buttons “OK” default button “OK”
        return invalid -- prevents the focus from leaving the textbox
    end if
end changed
```

Notes

➤ The value of the **`changing`** property is set to **`true`** once the user has altered the **`contents`** of a textbox

➤ If the **`changing`** property of a textbox is **`true`**, it receives a **`changed`** message when it loses the focus, as when the user clicks another window item or presses Tab.

➤ If a handler changes the **`changing`** property of a textbox from **`true`** to **`false`**, the textbox receives a **`changed`** message.

➤ Use the **`return invalid`** command to prevent the user from removing the focus from the textbox until an invalid entry is corrected.

contents

The text or key-filtered value of the textbox.

Value Class

text	the default contents
alias	a reference to a text file
any	returned by a key filter

Examples

```
set the key filter of textbox “txtDate” to “DisplayDates
set the format of textbox “txtDate” to “Both Date and Time”
set the contents of textbox “txtDate” to current date
copy the contents of textbox “txtDate” to myData --a date value
copy (the contents of textbox “txtDate”) as text to dateStr --a string value
```

Notes

➤ When a textbox has a **`key filter`** assigned to it, the key filter can determine the class of values that can be assigned to or retrieved from the textbox.

➤ See the **`key filter`** property description.

➤ If the **`contents`** is set to the alias of a text file (a file of type TEXT), the text is immediately loaded into the textbox. In edit mode, this is the same as typing text into the textbox. Otherwise, the text is not persistent; it will be lost when the window closes.

editable

Can the text be selected and edited by the user?

Value Class

boolean

Examples

```
copy the editable of theObj to isItEditable
set editable of textbox 3 of window “Editable” to false
set editable of theObj to not (editable of theObj)
```

Notes

➤ The standard editing commands—**`cut`**, **`copy`**, **`paste`** and **`clear`**—are automatically available in editable textboxes.

➤ When the Command key is down, the **`editable`** property of all enabled textboxes of a window is temporarily set to **`false`**. So, an editable textbox then behaves like an uneditable one.

➤ You can create “hot text”—words and phrases that act like push buttons—in editable and non-editable textboxes. See the discussion of the **`hilited`** message and the **`selection rule`** property for more information.

➤ The **`locked`** property is the inverse of the **`editable`** property.

form

See the description of the **`key filter`** property.

format

A string of parameters for use by a key filter (form) definition resource.

Value Class

string

Examples

```
set the format of window item 3 to theFormatString
```

Notes

➤ To see basic documentation for using the **`format`** property with a given key filter, first assign the key filter to the textbox, then choose Format from the Properties popup in the Property Bar.

➤ Some key filters do not require the use of the **`format`** property.

➤ For a detailed discussion of key filters, forms and formats, see the section entitled “Form Definition Resources and Formats” in this chapter.

justification

Alignment of the text within the bounds of the textbox.

Value Class

constant	left / right / center

Examples

```
set theJust to the justification of theObj
set the justification of textbox 3 of window “Documents” to center
```

key filter

A form that controls the entry of characters into the textbox.

Value Class

constant	none	no filter is used; any characters can be entered
string	key filter name	

Examples

```
set the key filter of textbox “txtRate” of window “Payables” to “onlyDigits”
```

Notes

➤ Key filters normally are set in edit mode, during the design of the window.

➤ A key filter might require the use of the **`format`** property.

➤ The key filter, if any, determines the value of the **`valid`** property.

➤ You can find out how to use a key filter by “opening” it in the Forms View of the Project Window.

➤ When a textbox has a **`key filter`** assigned to it, the key filter can determine the class of values that can be assigned to or retrieved from the textbox.

➤ For a detailed discussion of key filters, forms and formats, see the section entitled “Form Definition Resources and Formats” in this chapter.

line height

Line spacing in pixels between the baselines of text in a textbox.

Value Class

integer	
constant	single space

Examples

```
copy the line height of theObj to itsLH
set the line height of textbox “Table” of window “Report” to 16
set line height of textbox 3 to single space
```

Notes

➤ If the text contains fonts of various sizes, then setting **`line height`** to a specific value that is too small can make the words overlap. The **`line height`** should be **`single space`** for text of mixed sizes.

➤ When **`line height`** is set to an integer, the calculation (line height - size) yields the amount of blank space in pixels between lines of text.

➤ If not set, the value of the **`line height`** property defaults to **`single space`**.

locked

The **`locked`** property is the inverse of the **`editable`** property.

margin

Margin between the border of the textbox and its text.

Value Class

integer

Examples

```
copy the margin of theObj to itsMargin
set the margin of textbox “WideMargins” to 12
```

Notes

➤ The **`margin`** is measured in pixels. It applies to all four sides.

➤ If not set, the value of **`margin`** defaults to 3.

➤ Setting the **`margin`** of a textbox to 0 makes its border invisible.

➤ When the **`editable`** property of a textbox is set to **`false`**, its **`margin`** is automatically set to 0, hiding its border. You can then change it if you wish, showing the border.

➤ The **`margin`** of a textbox is *not* the same entity as the **`margin`** of a listbox's or movie's bold focus outline.

mixed styles

Can the textbox contain text with mixed styles?

Value Class

boolean

Examples

```
set the mixed styles of textbox "txtCustomer" to false
```

Notes

➤ The **`mixed styles`** property normally is set in edit mode, when designing the window.

➤ If the **`mixed styles`** property is set to **`false`** while mixed-style text is in the textbox, all the text assumes the same **`style`**, the **`style`** that was assigned to the textbox as a whole.

scroll

Distance that the content of a scrollable textbox has been scrolled.

Value Class

integer

Examples

```
copy the scroll of theObj to itsScroll
set the scroll of textbox "txtGroceryList" of window "To Do" to 72
```

Table of Contents Index

Notes

➤ The **`scroll`** property is measured in pixels.

➤ Setting the **`scroll`** property causes the textbox to scroll, and the scrollbar to adjust accordingly.

➤ Setting the **`scroll`** property does not cause a **`scrolled`** message to be sent to the textbox; a script can do that explicitly if needed.

➤ When the value of **`scrollable`** is **`false`**, the **`scroll`** property still has a value. The text can be scrolled by clicking and dragging in the textbox, and by setting its **`scroll`**.

scrollable

Does the textbox have a scrollbar?

Value Class

boolean

Examples

```
set itemScrollable to scrollable of theObj
set scrollable of textbox “txtDescription” of window “Order Entry” to false
```

Notes

➤ When the **`scrollable`** property is set to **`true`**, the textbox becomes 15 pixels wider; it becomes 15 pixels narrower when the **`scrollable`** property is set to **`false`**.

➤ When the value of **`scrollable`** is **`false`**, the textbox has no scrollbar, but the contents (if editable) can be scrolled by clicking in the text and dragging up or down.

➤ A textbox has a **`scroll`** value regardless of the value of the **`scrollable`** property.

➤ Textboxes whose **`wrapped property`** is **`true`** will accept return characters types into them. Therefore, if a window’s **`focus`** is the textbox, pressing Return will not actuate the default button.

➤ Non-wrapped textboxes do not accept return characters, and thus do not interfere with the default push button.

selection

Portion of the contents of a textbox that is selected.

Value Class

list of integer	{beginningOffset, endingOffset}

Examples

```
copy the selection of theObj to {selStart, selEnd}
set selection of theObj to {0,32767}
copy the contents of selection of textbox 3 to theSelectedText
set the contents of selection of textbox 3 to “This replaces the selected
text.”
```

Notes

➤ Setting the **`selection`** causes the indicated text to be hilited.

➤ Setting the **`selection`** to a range larger than the amount of text actually in the textbox highlights all of it.

➤ If the first and second items of the **`selection`** are equal, no text is selected; this is how an insertion point is represented.

➤ The selected text within a textbox, and its **`size`**, **`style`** and **`pen color`**, can be accessed using these reference forms:

```
contents of the selection of textbox “txtABC”
size of the selection of textbox “txtABC”
style of the selection of textbox “txtABC”
pen color of the selection of textbox “txtABC”
```

➤ The **`style`**, **`size`**, **`font`**, **`pen color`** and **`contents`** of the **`selection`** can be set independently of the rest of the text in the textbox.

Table of Contents Index

selection rule

Makes every word of a non-editable textbox into “hot text.”

Value Class

constant	as push button enables “hot text”

Examples

```
set the selection rule of textbox “txtInstructions” of to as push button
```

Notes

➤ The **`selection rule`** property, although not normally associated with textboxes, has been extended to provide a way to make every word of a non-editable textbox act like “hot text” without the use of the **`group`** text style.

➤ If the **`selection rule`** property is **`as push button`** (any other value is ignored), then clicking any word will select that word and send a **`hilited`** message to the textbox. You can use this feature to implement “hypertext” behaviors.

➤ When this technique is applied to an **`editable`** textbox, words act like “hot text” only when the Command key is held down.

valid

Is the text in the textbox correct according to the key filter?

Value Class

boolean	
list of integer	(see notes)

Examples

```
copy the valid of theObj to itsOK
copy (the valid of textbox 3) as list to {startBad, endBad}
```

Notes

➤ If the **`valid`** property is **`false`**, asking for it as a list (as shown in the examples) returns the **`selection`** of the first group of invalid characters in the textbox. If **`valid`** was **`true`**, the list returned is {0, 0}.

➤ The **`valid`** property is read-only.

wrapped

Is the text in the textbox automatically wrapped?

Value Class

boolean

Examples

```
set wrapFormat to wrapped of theObj
set wrapped of textbox 3 of window “Papers” to wrapFormat
set wrapped of textbox “Unwrap” to false
```

Notes

➤ Wrapping means that when a line of text will not fit within the bounds of a textbox, it is stopped after the last word that fits on that line, then continued on the next line.

➤ Textboxes whose **`wrapped`** property is **`true`** will accept return characters typed into them. Therefore, if a window’s **`focus`** is the textbox, pressing Return will not actuate the default button.

➤ Non-wrapped textboxes do not accept return characters, and thus do not interfere with the default push button.

Textbox Command and Event Messages

This section describes command and event messages that are sent specifically to textboxes.

Textboxes can also receive and handle several messages that are sent to most or all window items; see the section, “Window Item Command and Event Messages,” above.

changed

Event message sent when a newly-edited textbox loses the focus.

Parameters

(direct)	reference	the textbox whose contents have been changed

Example

```
property revertText:”revertText:””
on changed theObj
   copy contents of theObj to theContents
   if theContents < “2” or theContents > “7” then
      beep 1
      display dialog “Entry not between 2 & 7; try again.” buttons “OK”
      set contents of the window item to revertText
      return invalid -- prevents focus from leaving the textbox
   else
      set revertText to theContents
   end if
   end changed
```

Notes

➤ A textbox loses the focus—and is sent a **`changed`** message—when the application user clicks or tabs to another window item, or when the window is closed.

➤ Textboxes in modal dialogs do not receive the **`changed`** message if the application user clicks a button whose **`cancel item`** property is **`true`**.

➤ The example script shows how the application user can be forced to enter a valid value into a textbox. Note the use of the **`return invalid`** command to inhibit the loss of focus from the textbox.

➤ See the **`changing`** property of textboxes for more information.

clear

Edit menu command: Deletes the contents of the selection of a textbox.

Parameters

(none)		textbox with the focus
(direct)	reference	the textbox

Examples

```
clear
clear textbox “txtDumbo”
```

Notes

➤ Without a direct parameter, the **`clear`** command clears the **`contents of the selection`** of the textbox, if any, that has the focus.

➤ With a direct parameter that is a reference to a textbox (with or without the focus), the **`clear`** command clears the **`contents of the selection`** of that textbox.

➤ See the **`selection`** property of textboxes for more information.

copy

Edit menu command: Copies the contents of the selection of the specified textbox to the Clipboard.

Parameters

(none)		textbox with the focus
(direct)	reference	the textbox

Examples

```
copy textbox “txtBoilerPlate” of window “MyText”
```

Notes

➤ Without a direct parameter, the **`copy`** command copies the **`contents of the selection`** of the textbox, if any, that has the focus.

Table of Contents Index

➤ With a direct parameter that is a reference to a textbox (with or without the focus), the **`copy`** command copies the **`contents of the selection`** of that textbox.

➤ See the **`selection`** property of textboxes for more information.

cut

Edit menu command: Copies to the Clipboard the contents of the selection of a textbox, and deletes that text from the textbox.

Parameters

(none)		textbox with the focus
(direct)	reference	the textbox

Examples

```
cut
cut textbox “txtDocument1” of window “DocumentEditor”
```

Notes

➤ Without a direct parameter, the **`cut`** command cuts the **`contents of the selection`** of the textbox, if any, that has the focus.

➤ With a direct parameter that is a reference to a textbox (with or without the focus), the **`cut`** command cuts the **`contents of the selection`** of that textbox.

➤ See the **`selection`** property of textboxes for more information.

focus received

Event message sent when a textbox gains the focus.

Parameters

(direct)	reference	the textbox

Table of Contents Index

Example

```
property revertText:"revertText:""
on focus received theObj
   set revertText to contents of theObj
end focus received
```

Notes

➤ A textbox receives the focus when the application user clicks the textbox or tabs to it, or when a script sets the **`focus`** of the window to the textbox.

➤ The **`focus received`** message is sent only to textboxes whose **`editable`** property is **`true`**.

hilited

Event message sent when any group-styled text in a locked textbox is clicked.

Parameters

(direct)	reference	the textbox

Example

```
on hilited theObj
   set contents of textbox "holdSelection" to contents of selection of theObj
end hilited
```

Notes

➤ In a non-editable textbox, text with the **`group`** style is hilited when the application user clicks it; a **`hilited`** message is then sent to the textbox. See the discussion of the **`editable`** property.

➤ In a non-editable textbox, any word of a textbox whose **`selection rule`** property is set to **`as push button`** is hilited when the application user clicks it; a **`hilited`** message is then sent to the textbox. See the discussion of the **`selection rule`** property.

➤ At the moment the textbox gets the **`hilited`** message, the selected text is available as the **`contents of the selection`**.

➤ You can use this "hot text" feature of **`locked`** textboxes to implement hypertext.

➤ Holding down the Command key temporarily locks the contents of **`editable`** textboxes, allowing them, too, to behave in this way.

➤ Words and phrases can be set to the **`group`** style from the Style menu while editing the window. To group text from a script, use the **`on styles`** property described in Chapter 15: “Special Artwork and Text Style Classes.”

keystroke

Event message sent to a textbox with the focus when a key is pressed.

Parameters

(direct)	reference	the textbox
key	long integer	the key code/character
[option down]	boolean	is Option key down?
[shift down]	boolean	is Shift key down?
[command down]	boolean	is Command key down?
[control down]	boolean	is Control key down?
[ticks]	integer	time (see notes)

Examples

```
-- Convert lower case text typed by the application user to UPPERCASE:
on keystroke theObj key keychar --ignoring other parameters
-- Extract the ASCIIvalue of typed character into variable theChar:
   copy keychar mod 256 to theChar
-- If theChar is lower case, promote the keychar to upper case:
if theChar > 96 and theChar < 123 then ¬
      copy keychar - 32 to keychar
-- Continue keystroke message using the updated value:
   continue keystroke theObj key keychar
end keystroke
```

Notes

➤ The **`key`** parameter is a composite value containing both the internal Macintosh key code and the ASCII value of the character actually typed by that key.

➤ To extract the key code, divide the **`key`** parameter by 256. To extract the ASCII code of the character that was typed, use the modulo 256 of the **`key`** parameter.

➤ A script can send the **`keystroke`** message as a command to simulate typing directly into a textbox (at the insertion point or selection) just as an application user does. The characters appear one at a time.

➤ If the **`keystroke`** handler is not continued, then the character is not typed into the textbox.

➤ The standard String Commands scripting addition includes functions for converting between characters and their ASCII values.

➤ **`Ticks`** indicates 60ths of a second since the last system startup and can be used to determine elapsed time between two events. For example, if you want to distinguish a double-click from two discrete clicks, you can compare the value of ticks at each click and conclude that the user intended a double-click if the difference between the two values is less than, for example, 30.

paste

Edit menu command: Pastes the contents of the Clipboard over the contents of the selection of a textbox.

Parameters

(none)		textbox with the focus
(direct)	reference	the textbox

Examples

```
paste
paste textbox “txtHeaders” of window “Outline”
```

Notes

➤ Without a direct parameter, the **`paste`** command pastes the contents of the Clipboard in place of the **`contents of the selection`** of the textbox, if any, that has the focus.

➤ With a direct parameter that is a reference to a textbox (with or without the focus), the **`paste`** command pastes the contents of the Clipboard in place of the **`contents of the selection`** of that textbox.

➤ See the **`selection`** property of textboxes for more information.

scrolled

Event message sent when the textbox is scrolled interactively.

Parameters

(direct)	reference	the textbox

Example

```
on scrolled theObj
   copy scroll of theObj to newscroll
   set scroll of textbox 'txtFirstName" to newscroll
end scrolled
```

Notes

➤ A textbox gets a **`scrolled`** message when it is scrolled with its scrollbar (its **`scrollable`** property is **`true`**), or by clicking in it and dragging upward or downward.

➤ Setting the **`scroll`** from a script does not cause the **`scrolled`** message to be sent; send a **`scrolled`** message, if needed.

➤ Two or more textboxes can be made to scroll in parallel by copying the **`scroll`** property of each one to the others when the **`scrolled`** message is received.

Text Suite

The Text Suite is a standard set of objects and properties for manipulating text *in textboxes*. This suite has been adopted by developers whose applications include text editing.

As defined by the Text Suite, the contents of a textbox can be viewed as a whole text, or as a collection of paragraphs, lines words or characters. Each of these elements can be referenced directly, using its index within the container.

FaceSpan implements all the important objects and properties of the Text Suite; missing are **`best type`**, **`class`** and **`default type`** properties, and the **`text flow`** object.

Reference Forms

Here are examples of the various ways that you can reference text in your applications:

➤ characters of textbox 3

➤ character 2 of textbox 3

➤ words of textbox 3

➤ word 7 of textbox 3

➤ lines of textbox 3

➤ line 5 of textbox 3

➤ paragraphs of textbox 3

➤ paragraph 2 of textbox 3

➤ character 2 of word 7 of paragraph 2 of textbox 3

In addition, you can refer to sequences of characters, words, lines and paragraphs. Here are some examples:

```
characters 11 thru 17 of textbox 1 --a list of those 7 characters
words 5 thru 10 of paragraph 3 of textbox 1 --a list of those 6 words
lines 1 thru 8 of textbox “txtRaven”
```

If you refer to words this way, note that the words returned have no punctuation; the result is just the words, delimited by spaces.

Often you will want the text that falls within a range of words, characters, and so on, as a single string, rather than a list of words or of characters. To obtain the text as one string, use statements that follow these forms:

```
text from word 1 to word 6 of paragraph 1 of textbox 1
text from character 40 to character 50 of paragraph 1 of textbox 1
text from character 20 to character 36 of textbox 1
```

When you obtain a range of text this way, it includes all the punctuation that lies in that range.

Characters

A character is a single letter, digit or other symbol in a text.

Properties of Characters

It is characters that display in a textbox, and so it is characters that have the properties that govern appearance.

color

Color of the character.

Value Class

RGB color	{redValue, greenValue, blueValue}
integer	index to color in System color lookup table
constant	black / white

Examples

```
copy the color of character 27 of textbox “txtMessage” to theColor
set the color of character 27 of textbox “txtMessage” to black
set the color of characters 1 thru 20 of textbox 1 to {{65535, 0, 0}}
```

Notes

➤ The **`color`** property is always returned as an RGB value, a list of three long integers, from 0 to 65535, representing red, green and blue intensities.

➤ The integer values for indexing color are treated as a list within a list.

➤ By default, **`color`** is **`black`**.

font

The name of the font.

Value Class

string

Examples

```
copy the font of character 27 of textbox “txtMessage” to theFont
set the font of character 27 of textbox “txtMessage” to “New York”
```

Note

➤ The default **`font`** of a textbox depends upon the window’s **`font`** property.

size

The size of the font in pixels.

Value Class

fixed

Examples

```
copy the size of character 27 of textbox “txtMessage” to theSize
set the size of characters 10 thru 27 of textbox 1 to 18
```

Note

➤ The default **`size`** of a textbox depends upon the window’s **`size`** property.

style

The text style.

Value Class

text style info

Examples

```
copy the style of character 27 of textbox “txtMessage” to theStyle
set the style of characters 10 thru 27 of textbox 1 to {on styles: bold}
```

Notes

➤ For more information about the **`text style info`** class, see Chapter 15: “Special Artwork and Text Style Classes.”

➤ The default **`style`** of a textbox depends upon the window’s **`style`** property.

uniform styles

The text styles that are uniform throughout the text.

Value Class

text style info

Examples

```
copy the uniform styles of character 27 of textbox “txtMessage” to theUS
```

Notes

➤ Although a single character does have a **`uniform styles`** property, it is of no real use except when dealing with groups of characters.

➤ For more information about the **`text style info`** class, see Chapter 15: “Special Artwork and Text Style Classes.”

Table of Contents Index

Lines, Paragraphs, Words

Lines contain words and characters. All the properties of characters apply to lines as well. However, the values of these properties are the values associated with the first character of the line in question.

Properties of Lines

Lines have one additional property—**`justification`**.

justification

Justification of the text.

Value Class

constant	left / right / center

Examples

```
copy the justification of line 1 of textbox “txtMessage” to theJust
```

Note

➤ **`Justification`** is a read-only property.

Paragraphs

Paragraphs contain words and characters. All the properties of characters apply to paragraphs as well. However, the values of these properties are the values associated with the first character of the paragraph in question.

Words

Words contain characters. All the properties of characters apply to words as well. However, the values of these properties are the values associated with the first character of the word in question.

Chapter 14:

Menus and Menu Items

Contents:

Table of Contents Index

Menus and Menu Items

Like window items, menus and menu items are interface objects with adjustable properties. However, menus and menu items do not have scripts, so the **`chosen`** message sent when a user chooses an item from a menu must be handled in the script of the active window or its application.

See the discussion of the chosen message in Chapter 12: “Windows.”

Menus

Menus constructed with FaceSpan’s Menu Editor and saved with a project’s resources are used as templates for the menus opened while a project application is running. Menu resources can be attached to a window as its **`private menus`** or to the application.

Table of Contents Index

Properties of Menus

The properties of menus and menu items normally are applied to a menu template, using the Menu Editor, but all can be set at run time as well. There are two important points to consider: first, run-time changes to menus can be made only to the displayed copy of the menu, not to its template, and second, the menu must be displayed at the time its property is to be set.

contents

A list of the names of the menu items contained by the menu.

Value Class

list of strings	{menuItamName”,”menuItemName”...}

Examples

```
copy the contents of menu “Loop De Jour” to menuList
set the contents of menu 2 to {“Show Totals”,”Hide Totals”}
set the contents of menu 2 to “Show Totals\rHide Totals “--same as above
```

Note

➤ You can use a return-delimited string, instead of a list of strings, to set the `contents`. The string should contain the menu items separated by return characters.

enabled

Is the menu active or inactive?

Value Class

boolean

Examples

```
copy the enabled of menu “Documents” to itsEnabled
set the enabled of menu “Formats” to false
```

Notes

➤ An active menu is normal in appearance and responsive to user input; an inactive menu is dimmed and unresponsive to user input.

➤ If you need to disable all the items in a menu, disable the menu itself, so that it is apparent from the menu title that there are no enabled items.

➤ By default, **`enabled`** is **`true`**.

form

The form of the menu as defined by a form definition resource.

Value Class

constant	standard	no custom form
string	see note	

Examples

```
copy the form of theObj to itsFormName
set the form of popup “popColors “ to “MenuOfColors”
set the form of popup 12 to standard
```

Notes

➤ The default standard form for a menu (a resource of type MDEF) is built into FaceSpan.

➤ Menu forms can be assigned only at runtime; the value is not persistent from run to run.

➤ Optional menu forms can be imported into a project. These can support the display of icons and pictures, and have a variety of other features.

➤ Menus neither have nor use the **`format`** property.

➤ Many menu forms require that menu items themselves have special formats. These often are descriptions of what is to be displayed in place of the item text.

➤ To see basic documentation for a form, select its name in the Forms View of the Project Window, then click the Open button.

➤ For a discussion of forms, see Chapter 13: “Window Items.”

index

The index number of the menu within the menu bar.

Value Class

integer

Examples

```
copy the index of menu "View" to itsIndex
set the enabled of menu 3 to false
```

Notes

➤ Menus are indexed sequentially from left to right, starting with the apple menu at 1.

➤ The **`index`** of a menu is a read-only property.

name

The displayed name of the menu.

Value Class

string

Examples

```
copy the name of theObj to itsName --theObj is a parameter of the chosen msg
copy name of menu 2 to menu2name
```

Notes

➤ The **`name`** of a menu is a read-only property.

➤ The **`name`** property is the same as the **`title`** property of a menu.

title

The **`title`** property is the same as the **`name`** property.

Table of Contents Index

Menu Items

Menu items are the individually-choosable items contained within a menu or a popup. Each menu item can be assigned an optional Command-key equivalent, and can be enabled or disabled, checked or not checked.

Properties of Menu Items

The properties of menus and menu items normally are applied to a menu template, using the Menu Editor, but all can be set at run time as well. There are two important points to consider: first, run-time changes to menus can be made only to the displayed copy of the menu, not to its template, and second, the menu must be displayed at the time its property is to be set.

checked

Does a mark character appear alongside the menu item?

Value Class

boolean

Examples

```
copy the checked of menu item 2 of menu 3 to itIsChecked
set the checked of menu item “Grid” of menu “Alignment Tools” to true
```

Notes

➤ The character used to mark a menu item is defined by the **`mark`** property.

➤ If you set the **`mark`** of a menu item to a value other than a null string (“”), **`checked`** is set to **`true`**. If you set mark to a null string, **`checked`** is set to **`false`**.

➤ By **`default`**, checked is **`false`**.

command key

Command-key equivalent that activates a menu item.

Value Class

string	an alphanumeric character

Examples

```
copy the command key of theObj to cmdChar
set the command key of menu item “Find...” of menu “Tools” to “F”
```

Notes

➤ The **`command key`** property does not apply to popup menu items.

➤ If you set **`command key`** to a null string (“”), then no Command-key equivalent is in effect.

➤ Do not set **`command key`** to a space.

➤ If you use the same Command-key equivalent for more than one item in a menu, the uppermost item with that equivalent will be chosen when the Command key is pressed.

➤ If you use the same Command-key equivalent for items in more than one menu, the item in the leftmost menu with that equivalent will be executed when the Command-key is pressed.

➤ Command keys for menu items also can conflict with the command keys assigned to push buttons.

➤ The **`command key`** is the null string by default.

contents

The **`contents`** property is the same as the name property.

enabled

Is the menu item enabled (active)?

Value Class

boolean

Examples

```
set the enabled of menu item “Apply...” of menu “Formats” to false
copy the enabled of menu item 2 of menu 1 to itsEnabled
```

Notes

➤ When a menu item is enabled, it is normal in appearance and responsive to user input; when disabled, it is dimmed and unresponsive to user input.

➤ **`Enabled`** is **`true`** by default.

index

The index number of a menu item within its menu.

Value Class

integer

Examples

```
copy the index of menu item “Side” of menu “View” to itsIndex
set enabled of menu item “Editing Tools” of menu “Tools” to false
```

Notes

➤ Menu items are indexed sequentially from top to bottom within their menus. The menu name itself is not counted.

➤ The **`index`** of a menu item is a read-only property.

mark

The character (if any) that marks a checked menu item.

Value Class

string	a single alphanumeric character

Examples:

```
copy the mark of theObj to itsMark
set the mark of menu item “Editing Tools” of menu “Tools” to “•”
```

Notes

➤ If you set the **mark** of a menu item to a value other than a null string (“”), checked is set to **true**. If you set the mark to a null string, **checked** is set to **false**.

➤ If **checked** is **true** and the mark is not specified, mark defaults to the standard check mark symbol.

name

The name or text of the menu item.

Value Class

string

Examples

```
copy the name of theObj itemName
set the name of menu item 4 of menu “Edit Tools” to “Sorter”
```

Note

➤ The **name** property is the same as the **contents** property of a menu item.

style

The text style of the first character of the name of a menu item.

Value Class

text style info

Examples

```
copy the style of theObj itsStyle
set the style of menu item 3 of menu “Notes” to ¬
    {on styles:{italic},¬
    off styles:{bold, underline, outline, shadow}}
```

Notes

➤ The **`style`** of a menu item is expressed as lists of **`on styles`** and **`off styles`**. See Chapter 15: “Special Artwork and Text Style Classes.”

uniform styles

The text styles that are uniform to the contents of a menu item.

Value Class

text style info

Examples

```
copy the uniform styles of theObj to itsUStyles
set the uniform styles of menu item 3 of menu “Notes” to {on styles:{plain}}
```

Note

➤ The **`uniform styles`** property of a menu item is expressed as lists of **`on styles`** and **`off styles`**. See Chapter 15: “Special Artwork and Text Style Classes.”

Menu Command and Event Messages

There is only one event and message associated with menus, and it is handled in the script of the menu’s window or in the project script.

chosen

When the application user chooses a command from a menu, FaceSpan sends a **chosen** message to the active window, if any are open, or to the project itself.

The direct parameter of a **chosen** message is a reference to the menu item currently chosen. You can use this reference to query AppleScript for references to all the containers of the menu item—its menu, window, and application.

The **chosen** message can be sent as a command, but the menu must be visible in the menus bar or an error will occur.

Menus and menu items do not have their own scripts. See the discussion of the **chosen** message in Chapter 12: “Windows.”

Chapter 15:
Special Artwork and Text Style Classes

Contents:

Table of Contents Index

Resource Info

The `resource info` object class is used to specify artwork displayed by icon and picture window items, and by table cells.

See the discussions of the artwork property of icons and pictures in Chapter 13: “Window Items,” for more information.

Properties of Resource Info

The resource info object is implemented as a record. The object’s properties are the fields of the record. Here is an example resource info record:

```
{class:resource info, typesetting,” name:”FaceSpan,” id:5000}
```

The meanings of these four properties, or fields, are discussed in this section.

class

The defining class of a resource info record.

Value Class

resource info

Note

➤ It is not always necessary to include the `class` when assigning a resource info value. See the various examples in this section.

id

Unique identification number of a resource.

Value Class

integer

Examples

```
copy the artwork of icon 1 to {type:theType, name:theName, id:theID}

copy the artwork of icon “icnSignal” to theArt
set the id of theArt to the (id of theArt) + 3
set the name of theArt to “Raised Flag”
copy theArt to the artwork of icon “icnSignal”

on hilited theObj
   copy artwork of icon “icnCard” to cardArt
   if id of cardArt is 5101 then display dialog “You chose the Ace of spades.”
end hilited
```

Note

➤ The **`id`** must be unique only among resources of the same type. So, a PICT resource could have the same id as an ICON resource.

name

Name of an artwork resource.

Value Class

string

Examples

```
set the artwork of icon 1 to ¬
   {class:resource info, type:”cicn,” name:”Blue,” id:5102}
on hilited theObj
   copy the artwork of theObj to theArt
   display dialog (name of theArt)
end hilited
```

Note

➤ The **name** must be unique only among resources of the same type. So, a PICT resource could have the same **name** as an ICON resource.

type

The resource type of an artwork resource.

Value Class

string	PICT / ICON / ICN# / cicn

Examples

```
set artwork of picture “picBearPortrait” to {type:”PICT,” name:”Smokey”}

on MyTellType(theObj)
    copy the artwork of theObj to {type:itsType}
    if itsType is “cicn” then
        display dialog “The type is cicn.”
    else if itsType is “ICN#” then
        display dialog “The type is ICN#.”
    else if itsType is “ICON “ then
        display dialog “The type is ICON.”
    else if itsType is “PICT “ then
        display dialog “The type is PICT.”
    else
        display dialog “This type is...Gee, I don’t know.”
end MyTellType
```

Note

➤ See also the examples in the discussions of **`name`** and **`id`**.

Text Style Info

The `text style` info object class lets you specify which styles are on and off for the text of FaceSpan objects.

See the descriptions of the style and uniform styles properties of textbox items in Chapter 13: “Window Items,” for more information.

Properties of Text Style Info

off styles

The text styles that are off in the style or uniform styles property of an object or the selected text of an object.

Value Class

list	plain/bold/italic/outline/shadow/underline/ condensed/extended/group

Examples

```
copy style of window item 3 to thestyle
copy off styles of thestyle to theoffstyles
set uniform styles of textbox 3 to {on styles:{plain}, off styles:{bold, italic, un-
derline, outline, shadow, condensed, expanded, group}}
```

on styles

The text styles that are on in the style or uniform styles property of an object or selected text of an object.

Value Class

list	plain / bold / italic / outline / shadow / underline / condensed / extended / group

Examples

```
copy style of window item 3 to thestyle
copy on styles of thestyle to theonstyles
set style of selection of textbox 3 to {on styles:{bold,italic}, off styles:{under-
line, outline, shadow, condensed, expanded, group}}
```

Note

➤ For menu items only, the last style in the list is **`gray`** rather than **`group`**. **`Gray`** is not technically a text style, but using it sets to **`false`** the **`enabled`** property of a menu item in a menu being edited.

Chapter 16:

Storage Items

Contents:

Table of Contents Index

Storage

A storage item is a piece of data kept in permanent storage within a project. Any project storage item—defined by its **`name`** and **`contents`**—can be accessed directly, by name or id, from any script in the project.

Storage items can be created using the storage item editor (as explained in Chapter 5, “The Storage Item Editor”) or created at run time—by script—using the **`make`** command.

Reference Forms

There are only two reference forms for storage items, by **`name`** and by **`id`**:

storage item “Current Name”

storage item id 5003

Properties of Storage items

contents

The information kept in the storage item.

Value Class

anything

Examples

```
set newCount to currentCount + storage item “Saved Count” --a number
set theMessage to “Please call “ & storage item “Who to call” --a string

--A list, copied for local use:
copy storage item “My List” to localList
set x to item 3 of localList

--A list, coerced and used in place:
set x to item 3 of ((storage item “My List”) as list)

--How to use a script object from a storage item when you want the object
--to retain the values of its properties while in use:
copy Munger of ((storage item “Script Lib”) as script) to myMunger
tell myMunger to Catalog(theBook) --call handler in “Munger” object copy
--How to use a script object from a storage item when you will not
--be setting any of its properties:
tell Munger of ((storage item “Script Lib”) as script) to Catalog(theBook)
```

Notes

➤ You can **`get`** and **`set`** the **`contents`** of a storage item.

➤ Storage item values persist until changed by a script or in the storage item editor. They are automatically saved when a project is saved and when an application quits.

➤ Some **`contents`** property values cannot be used directly, but must be copied out of the storage item into a variable, or coerced in place. In general, these are the multi-valued items such as lists, records, text style info, resource info, scripts and script object.

Table of Contents Index

➤ The use of a storage item that contains a script object is shown in the examples. Use of a storage item that contains a single script is similar. In addition, you can copy the script out, use it, then copy it back in to preserve its properties.

id

The unique id of the storage item.

Value Class

integer

Examples

```
set storeID to the id of storage item “Francis”
```

Notes

➤ The **`id`** is a read-only property.

➤ The **`id`** values start at 5001.

name

The name of the storage item.

Value Class

string

Examples

```
copy the name of storage item 1 to oldStoreName
set the name of storage item 1 to “Ralph”
make new storage item ¬
   with properties {name:”wow”, contents:”Now what?”}
```

Notes

➤ The **`name`** property values are limited to strings no longer than 255 characters.

➤ The **`name`** property can be changed from a script, but doing so obviously invalidates any references that use the original name.

➤ Storage items can be created, named and set to a given value at run time, using the **`make`** command.

Special Considerations

Because they are both permanent and global, storage items can be used—with moderation—as associative memory or as a very simple database, using the **`name`** property as the key.

One of the more interesting uses of a storage item is to hold a library of script objects. Such script objects are available from any script in the project, yet do not get in the way and do not use up space in the application, window and window item scripts. You might even create a default project with your most-used script objects in storage items. See the discussion of the `contents` property for examples of the use of script objects.

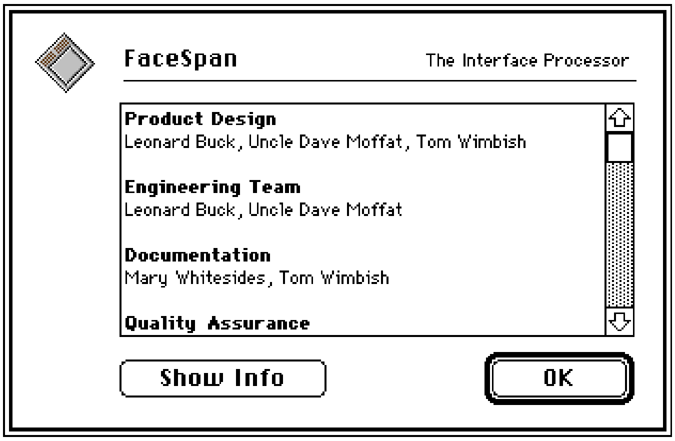
Appendix

Contents:

Table of Contents Index

Appendix A: FaceSpan Menu Reference

Apple menu



Product Design
Leonard Buck, Uncle Dave Moffat, Tom Wimbish

Engineering Team
Leonard Buck, Uncle Dave Moffat

Documentation
Mary Whitesides, Tom Wimbish

Quality Assurance

When FaceSpan is the active application, choose the About FaceSpan command from the Apple menu to display ordering and support services information, as well as a list of persons who contributed to FaceSpan's development. You can click the Show Info button to alternately display registration, software configuration, and memory usage information.

File menu

While a Project Window is active, all of the File menu's commands pertain to the project. Use the File menu to create and save projects, to revert an edited project to the state in which it was last saved, to convert a project into an application, and to print reports of the project's contents.

New Project

When you choose New Project, FaceSpan creates a new “Untitled” Project, and opens its Project Window. FaceSpan also creates a new project each time you launch it by double-clicking the FaceSpan desktop icon, or by opening FaceSpan using the Finder.

If you have created a customized “Default Project,” and placed it in the same folder as FaceSpan, each new “Untitled” project will be created from that template. If not, FaceSpan will provide its own standard default project. For instructions about how to create a customized default project, see Chapter 2: “Project Management.”

Open Project

Choose Open Project to display a dialog box from which you can open an existing project or editable application.

Close Project

Choose Close Project to close the active project. A Save Project dialog box displays if the project contains unsaved changes.

While a Window Editor, Menu Editor, or Script Editor is active, the Close command pertains to the active editor.

Revert Project

Choose Revert Project, to revert an active project to the state in which it was last saved.

While a Window Editor, Menu Editor, or Script Editor is active, the Revert command pertains to the active editor.

Save Project

Choose Save Project to save the active project under its current name. If the project has not yet been saved, the Save Project As dialog box displays so you can save the project in editable form under a new name, or optionally as a Miniature or Complete Application.

Table of Contents Index

Save Project As

Choose Save Project As to display the Save Project As dialog box, which allows you to save the active project in editable form, optionally as a Miniature or Complete Application.

Save As Run Only

Choose Save As Run Only to display the Save As Run Only dialog box, which allows you to save the active project in non-editable form (under a new name) as a Miniature or Complete Application.

Page Setup

Choose Page Setup to display the standard Macintosh printer Page Setup dialog box.

Print Project

Choose Print Project to print a report about the window templates, menu templates, artwork, form definition resources, and storage items of the active project.

While a Window Editor, Menu Editor, or Script Editor is active, the Print command pertains to the active editor.

Quit

Choose Quit to close any currently open projects and quit FaceSpan. If an open project contains unsaved changes, a Save Project dialog box displays.

Edit menu

While a Project Window is active, Edit menu commands can be used to Cut, Copy, Paste, Clear, and Duplicate window, menu, artwork, storage, and form resources shown in the listbox.

While a Window Editor, Menu Editor, or Script Editor is active, the Edit menu commands pertain to the active editor.

Window menu

Use the Window menu to display or hide FaceSpan windoids, and to move between open projects and their window templates.

Tools

Use the Tools command to hide or display the Window Editor’s Property Bar and Tool Palette.

Dictionary

You can use FaceSpan’s Dictionary Windoid to inspect FaceSpan’s own dictionary, or dictionaries of other scriptable applications. When you choose the Dictionary command from the Window menu, the Dictionary Windoid displays.

Detailed instructions about how to use the Dictionary Windoid are in Chapter 7: “Other Scripting Tools.”

Message

Use the Message command to hide or display the Message Windoid. While editing window templates, you can use the Message Windoid to get and set properties, and to send test commands to window items. Chapter 7: “Other Scripting Tools” contains detailed instructions about how to use the Message Windoid.

Window List

The name of each open project and its open window templates is listed in the Window List—a section of the Window menu set off by a horizontal divider. Beside each name is a sequentially numbered Command-key equivalent. You can choose a name (or use the keyboard equivalent) to make an open project or window frontmost.

Next Window

Use the Next Window command to bring the window template beneath the active window template to the front, making it active.

Table of Contents Index

Script menu

The Script menu is enabled whenever a Script Editor is active. Script menu commands provide additional control over scripts during editing.

Check Syntax

When you choose Check Syntax from the Script menu, FaceSpan attempts to compile the script, and reports any compilation errors. If errors in syntax are found, a Script Error dialog displays an explanation.

Recording

FaceSpan’s script recorder begins when you choose Recording from the Script menu. While recording is in progress, an editable script is generated from interactions with any recordable application. When the recorder is turned off—by choosing the Recording command a second time, scripts are compiled automatically and placed in the Script textbox of the open Script Editor.

Enter Selection

To automatically enter the highlighted text into the “Find” field of the Find dialog and Replace dialog, you select text within a script or the Message windoid, then choose Enter Selection from the Script menu. The selected string value is entered as the object of a search; you can now find or replace the string by using Find Again, Find in Next, or Replace Again commands from the Script menu.

Find

Use the Find command to locate a particular word or phrase occurring in a script.

First, choose the Find command from the Script menu to display the Find dialog. Next, enter the string value to be searched for in the “Find” field. Then, click the Find button. If the specified string exists in the Script Editor, FaceSpan finds and selects it.

Find Again

To search the active Script Editor for the next occurrence of a string value entered in the Find dialog, choose the Find Again command from the Script menu.

Find in Next

Choose Find in Next to search for the next occurrence of a string value—entered in the Find dialog—in the scripts of other window items, or other windows in the project. Window item scripts are searched in index order. Windows are searched in alphabetical order. If the specified string is found in the script of another window item or window, its Script Editor is opened, and the string is selected in the script.

Replace

To replace a particular word or phrase occurring in a script, you can use the Replace command.

First, choose Replace from the Script menu to display the Replace dialog. Next, enter a string value to be searched for—in the "Find" field of the dialog. Then, enter a string value with which to replace it. Finally, click the Replace button. If the specified string exists in the Script Editor, FaceSpan finds and replaces it with the new string value.

Replace Again

Choose Replace Again from the Script menu to search and replace the next occurrence of a string value—entered in the "Find" field of the Replace dialog—in the active Script Editor.

AppleScript Formatting

Choose the AppleScript Formatting command from the Script menu to display the AppleScript Formatting dialog box. Use the dialog box to set global preferences for formatting the text of all AppleScript scripts. You can find detailed instructions in Chapter 6: "The Script Editor."

Object menu

The Object menu commands provide additional control of the window template and its window items during editing.

Table of Contents Index

Object Info

To display an Object Information dialog, select a window template or window item, then choose the Object Info command. An Object Info dialog box allows you to inspect and modify many of the same window item properties as the Property Bar, but provides larger text-editing areas and more informative displays of current property values. You may also display a window item's Object Info dialog by double-clicking the item.

Object Script

To display the Script Editor for the selected object, select a window template or window item, then choose the Object Script command.

Snap to Grid

Snap to Grid causes a window item to align with the window template's 8-pixel-by-8-pixel grid if the item is moved or resized. When is Snap to Grid is "turned on" a check mark appears next to its name in the menu.

Snap to Size

When Snap to Size is turned on (check marked in the menu), the size of the window item is automatically adjusted to its standardized dimensions if you move or resize it.

Lock Position

When Lock Position is turned on (check marked in the menu) the **`drag locked`** property of a selected window item is set to **`true`**. When the **`drag locked`** property is **`true`**, selected window items cannot be moved from their current positions until you set the value to **`false`**.

Unlock Position

When Unlock Position is turned on (check marked in the menu) the **`drag locked`** property of a selected window item is set to **`false`**. When the **`drag locked`** property is **`false`**, selected window items can be moved from their current positions.

Alignment

To display a hierarchical menu of commands you can use to align the positions of window items, choose the Alignment command.

Align Lefts

Aligns the left edges of all selected window items with the left edge of the leftmost selected item.

Align Centers

Aligns the centers of selected window items along an imaginary vertical line that passes through the centerpoint of the items' horizontal span.

Align Rights

Aligns the right edges of selected window items with the right edge of the rightmost selected item.

Distribute Across

Distributes the centers of selected window items evenly across the breadth of the items' horizontal span.

Align Tops

Aligns the top edges of selected window items with the top edge of the topmost selected item.

Align Centers

Aligns the centers of selected window items along an imaginary horizontal line that passes through the centerpoint of the items' vertical span.

Align Bottoms

Aligns the bottom edges of selected window items with the bottom edge of the bottommost selected item.

Distribute Down

Distributes the centers of selected window items evenly down the height of the items' vertical span.

Table of Contents Index

Bring to Front

Brings the selected window items to the front layers in the window template by increasing their index numbers. This has no effect if a selected window item already has the highest **`index`**.

Bring Forward

Brings each selected window item forward by one layer, increasing its **`index`** by one. This has no effect if a selected window item already has the highest **`index`**.

Send Backward

Sends each selected window item backward by one layer, decreasing its **`index`** by one. This has no effect if the selected window item already has the lowest **`index`**.

Send to Back

Sends the selected window items to the back layers in the window template by decreasing their **`index`** numbers. This has no effect if the selected window item already has the lowest **`index`**.

Select

You can use the Select command to automatically select a window item in an active window editor. When the Select dialog box displays, enter the **`name`**, **`index`**, or **`id`** of the window item to be selected, then click the Select button.

Font menu

Choose the Font menu to display a list of fonts installed on your Macintosh computer. You can make a choice from the menu to change the font of a selected window item.

Style menu

Styles

Select a text **`style`** for any selected character(s) contained in a window item.

Sizes

Select a point **`size`** for any selected character(s) contained in a window item.

Pen Color

You can use this popup to set the **`pen color`** property of the selected window template or window item. Pen color determines the color in which the foreground (outlines and text) will be drawn.

Fill Color

You can use this popup to set the **`fill color`** property of the selected window template or window item. Fill color determines the color in which the object's background will be drawn.

Table of Contents Index

Appendix B: Commands and Shortcuts

Keyboard commands

The Project Window, Window Editor (window template, Property Bar, Tool Palette), and Script Editor can all be partially controlled from the keyboard. The results of some keyboard commands differ depending on which of these is active.

Project Window commands

Option	Allows the deletion of project resources without confirmation.
⌘-R	Use as a shortcut for the Run Project command.
⌘-K	Adds a new item to a project.

Window Template keyboard commands

Tab	Moves the selection from window item to window item in ascending index order.
Shift-Tab	Moves the selection in descending index order.
Arrow keys	Move the selected window item in the indicated direction by 1 pixel if the Snap To Grid command in the Object menu is turned off, or 8 pixels if Snap To Grid is turned on.
⌘-Arrow keys	Changes the width or height of the selected window item by 1 pixel if Snap To Grid in the Object menu is turned off, or 8 pixels if Snap To Grid is turned on. (Adjustments to width are always made to the right edge of an object. Adjustments to height are always made to the bottom edge of an object.)
Control	Temporarily reverses the current Snap to Grid setting.

Property Bar keyboard commands

Return or Enter	Brings the focus back to the window template from the Property Bar.
Tab	Moves the selection from textbox to textbox in the Property Bar.
Shift-Tab	Moves the selection from textbox to textbox in reverse order.
Arrow keys	Move the insertion point within the selected textbox of the Property Bar.

Tool Palette keyboard commands

Option-Tab	Shows or hides the Tool Palette and Property Bar.
⌘-Tab	Chooses the Arrow tool.
⌘-Tab+Tab	Chooses the I-beam tool.
⌘-Tab+Tab+Tab	Chooses the Object Mover Tool.

Script Editor keyboard commands

Enter	Checks Syntax.

Keyboard equivalents for menu commands

Window menu keyboard shortcuts

Show/Hide Dictionary	⌘,
Show/Hide Message	⌘–M
Make Project Active	⌘-(number of the projects)
Next Window	⌘-L

Table of Contents Index

Object menu keyboard shortcuts

Object Info	⌘-I
Expanded Object Info (if available)	⌘-Option-M
Object Script	⌘-E
Snap To Grid	⌘-Y
Snap To Size	⌘-V
Bring to Front	⌘ =
Send Back	⌘ -
Select	⌘-F

Script menu keyboard shortcuts

Check Syntax	⌘-K
Enter Selection	⌘-E
Find	⌘-F
Find Again	⌘-G
Find In Next	⌘-J
Replace	⌘-R
Replace Again	⌘-T

Mouse Shortcuts

Opening the Script Editor

Hold the Command Key while double-clicking a window item

Selecting a line of code in the Script Editor

Triple-click the line of code.

Selecting enclosed text in the Script Editor

Double-click one of these characters to select all the text up to its mate:
“[{ « (' “ ”

Selecting multiple items in a template window

Hold the Command key while dragging over window items.

Opening the Object Information dialog

Double-click a window item

Opening an expanded Object Information dialog

Hold the Option Key while double-clicking a window template, textbox, listbox, or popup.

Displaying Artwork Information dialogs

Double-click the artwork while the Artwork Chooser is open.

Keeping the same Object Maker tool

Hold the Command Key while the tool is selected, then drag to make several of the same object.

Duplicating a window item

Hold the Option Key while click-dragging the object to be cloned.

Aligning window items

Selected window items can be aligned by using combinations of modifier keys and mouse clicks to the alignment tool icons in the Property Bar.

Align Lefts	◀	Option-click
Align Rights	◀	Option Shift-click
Align Tops	▲	Option-click
Align Bottoms	▲	Option Shift-click
Distribute Across	◀▶	Option-click
Distribute Down	▲▼	Option-click

Table of Contents
Index

Appendix C: Sizes and Limits

This appendix tells about the maximum sizes and counts you can expect while using FaceSpan.

Sizes and counts:

➤ Fewer than 330 window items are allowed in an open window template.

➤ Up to 9 projects can be open at once.

➤ Textboxes and listboxes can hold up to 32K (32,767) characters of text.

➤ Scripts can contain up to 32K of text each.

➤ The Message Windoid's log area can contain up to 32K of text.

➤ Each cell of a table can contain 32K of text.

➤ The effective length of a listbox item is 100 characters.

➤ Each item stored in the storage items area can contain up to 32K of text.

Larger than effective sizes:

➤ Labels and checkbox, radio button, push button, table row, table, column, and box titles can contain up to 250 characters.

➤ Balloon text can contain up to 32K of characters.

➤ Window, menu, menu item, artwork, form, and storage item names can contain up to 250 characters.

Note

➤ Drag & drop works by default under System 7.5, and under System 7.1 in which it has been purposely installed (however any FaceSpan application can be made to "drop-launch").

Table of Contents Index

Appendix D: Scripting Resources

To use FaceSpan effectively, you will want to develop a good understanding of AppleScript in addition to whatever other OSA compatible scripting language you use. You may find the following books and resources helpful.

Books About AppleScript

➤ *AppleScript Applications: Building Applications with FaceSpan and AppleScript.* Peppermill, MA, August 1996.

AppleScript Applications: Building Applications with FaceSpan and AppleScript (ISBN 0-12-6233957-6) shows the reader how to create complete Macintosh applications using AppleScript and the FaceSpan interface builder. It includes detailed examples that are developed over the course of the book. It includes sections on designing Graphical User Interface (GUI) and on debugging applications using FaceSpan. A CD Rom includes AppleScript 1.1, a demonstration version of FaceSpan 2.1, source code for all example applications, and numerous AppleScript shareware and demonstration programs. The book will be released in August 1996.

➤*AppleScript Finder Guide*. Cupertino: Apple Computer, Inc., 1994.

AppleScript Finder Guide (ISBN 0-201-409-10-0) describes the commands and object classes defined by the Finder for use with the English dialect of the AppleScript language. This book is for those who want to record scripts, write new scripts, or modify existing scripts that control actions in the Finder.

➤ *AppleScript for Dummies* (ISBN# 1-56884-975-3). Indianapolis: IDG Books Worldwide, Inc.

➤ *AppleScript Language Guide English Dialect*. Cupertino: Apple Computer, Inc., 1993.

AppleScript Language Guide English Dialect (ISBN 0-201-40735-3) gives information about the commands and other terms provided by the English dialect of the AppleScript scripting language and by the Scriptable Text Editor. This book is bundled with the AppleScript software in the AppleScript Development Kit from APDA and AppleScript 1.1 Scripting Kit from Apple resellers. Addison-Wesley Publishing Company, Inc., also sells it separately.

➤ *AppleScript Scripting Additions Guide English Dialect*. Cupertino: Apple Computer, Inc., 1994.

AppleScript Scripting Additions Guide (ISBN 0-201-40736-1) describes the scripting additions that accompany the AppleScript English Dialect of the AppleScript language. This book is for anyone who wants to write or modify scripts, as well as for developers who want to write scripting additions. It is included in the AppleScript Development Kit from APDA and AppleScript 1.1 Scripting Kit. You can also order it separately from Addison-Wesley.

➤ Goodman, Danny. *Danny Goodman's AppleScript Handbook*, 2d ed. New York: Random House Electronic Publishing, 1994.

This second edition of *AppleScript Handbook* (ISBN 0-679-75806-2) includes in-depth coverage of scripting the Finder and the top scriptable applications (e.g., Excel 5, Word 6, QuarkXPress, FileMaker Pro). A complete FaceSpan application (a "reminder alarm agent") is described in the book and included on the book's companion disk.

➤ Michel, Steve. *Scripting the Scriptable Finder*. Pleasant Hill: Heizer Publishing, 1995.

This book presents a in-depth discussion about how to use AppleScript to script the Scriptable Finder. It's companion disk contains some scripting utilities, a number of scripting additions, and over 100 scripts. *Scripting the Scriptable Finder* is for novice scripters and Mac Managers alike.

➤ Schneider, Derrick, with Hans Hansen and Tim Holmes. *The Tao of AppleScript*. 2d ed. Indianapolis: Hayden Books, 1994.

The Tao of AppleScript (ISBN 1-56830-075-1) from the Berkeley Macintosh User Group (BMUG), features expanded coverage of AppleScript basics and new information on the Scriptable Finder. This is a book for "all users." The Tao's two companion disks include AppleScript 1.1, some scriptable applications, scripting additions, and many example scripts.

➤ Trinko, Tom. *Applied Mac Scripting*. New York: Henry Holt & Company, 1994.

Applied Mac Scripting (ISBN 1-55828-330-7) teaches you a process for developing scripts—from initial idea to final delivery. More than fifty pages and dozens of screenshots are devoted to how to design and implement a FaceSpan application. The book's companion CD-ROM contains AppleScript, a run-time version of Frontier UserTalk, and demo versions of several other automation tools.

Table of Contents Index

Other Helpful References

➤ *Electronic Guide to Macintosh Human Interface Design.*

This CD-ROM disc contains full electronic text of *Macintosh Human Interface Guidelines* as well as its CD-ROM companion, *Making It Macintosh*, and is available through APDA.

➤ *Inside Macintosh: Interapplication Communication.* Cupertino: Apple Computer, Inc., 1993.

Inside Macintosh: Interapplication Communication (ISBN 0-201-62200-9) is a source for more information about OSA, Apple Events, and AppleScript. To use this book, you should be familiar with the Macintosh Toolbox and how to respond to user events. It is available through APDA, technical bookstores, and Addison-Wesley.

➤ *Macintosh Human Interface Guidelines*. Cupertino: Apple Computer, Inc., 1993.

Macintosh Human Interface Guidelines (ISBN 0-201-62216-5) can help you make your application look and act as much as possible like a standard Macintosh application. It explains the reasoning behind the design of the Macintosh interface and tells what gives the Mac it's special "look and feel." The book is published by Addison-Wesley, and is available through APDA and in bookstores.

➤ Tognazzini, Bruce. *TOG on Interface*. Reading: Addison-Wesley, 1992.

TOG on Interface (ISBN 0-201-60842-1) is for "all those concerned about the relationship between people and computers," and discusses the underlying principles of graphic user interface design…as well as Information Theory, Jungian philosophy, and more.

Sources

➤ Addison-Wesley Publishing Company, Inc.
Reading, Massachusetts
1-800-822-6339

➤ APDA
Apple Computers, Inc.
P.O. Box 319
Buffalo, NY 14207-0319
1-800-282-2732 (Unites States)
1-800-637-0029 (Canada)
1-800-871-6555 (International)
e-mail (AppleLink): APDA
e-mail (Internet): APDA@applelink.apple.com

➤ Apple Computer, Inc.
20525 Mariani Avenue
Cupertino, CA 95014
408-996-1010

➤ Berkeley Macintosh User Group (BMUG)
1442A Walnut Street, #62
Berkeley, CA 94709-1496
1-800-776-BMUG
e-mail (Internet): bmug@aol.com

➤ Heizer Publishing
P.O. Box 232019
Pleasant Hill, CA 94523
510-943-7667
e-mail (Internet): heizersw@aol.com

➤ Henry Holt and Company
115 West 18th Street
New York, New York 10011
1-800-488-5233

Other Scripting Tools

The AppleScript Development Kit from APDA and AppleScript 1.1 Scripting Kit from Apple resellers include Apple's Script Editor and the Scriptable Text Editor.

The Script Editor is an application you can use to open and run scripts as well as record, write, and save new scripts. The Scriptable Text Editor is a sample scriptable application.

Script Debugger, a product of Late Night Software Ltd. Voice: (604) 929-5578, has great enhancements for writing and debugging your scripts.

AppleScript Support on-line

AppleLink

Developer Support: AppleScript Talk

(Note: The “Interface Builder” folder is for discussion of FaceSpan)

Internet

MacScripting Digest Mailing List

To subscribe, send a mail message to:
LISTSERV@dartmouth.edu

include the following in the body of the message:
SUBSCRIBE MACSCRPT firstname lastname

AOL

Computing: Utilities/Desk Accessories: AppleScripting

eWorld

Computer Center:Software Center from Ziffnet/Mac:Software Central:Scripting & Programming:AppleScript & Frontier

Computer Center:Straight to the Source:Ground Zero

Computer Center:Forums:Macintosh Development Forum:Software Library:Scripting Samples

Compuserve

MACDEV/Scripting Month

MACDEV/Tools/Debuggers

MACDEV/Other Languages

MACSYS/Utilities

DTPFORUM/Mac DTP Utilities

APPHYPER/XCMDs & XFCNs

APPHYPER/Xpert Alley

DTPFORUM/Program Demos

INETRESOURCE/Mac Internet S/W

MACAP/Databases

MACDEV/C and Pascal

MACAP/Misc. Applications

Appendix E: Reserved Words List

There are many words that are used for objects, classes, properties, messages and values by FaceSpan. These so-called “reserved words” may be used only for their intended purposes. For example, you cannot have a variable called “hilited”, since **`hilited`** is a message name and, therefore, a reserved word.

There are also words reserved for use in AppleScript, in the Text Suite and in each scriptable application.

FaceSpan’s reserved words are:

activated
application
artwork
at
balloon
black
bold
bounds
box
boxes
cell
cells
center
changed
changes
changing
character
characters
checkbox
checkboxes
checked
chosen
class
clear
click
clipboard
close
closeable
color
column
columns
condensed
contents
copy
corners
count
cursor
cut
deactivated
delete
description
dialog
down
drag
draggable
draw
drop
droppable
editable
enabled
extended
floating
focus
font
form
format
front
frontmost
gauge
gauges
gray
group
growth

height
hilite
hilited
icon
icons
id
idle
index
inset
interruptible
italic
item
justification
key
keystroke
label
labels
leap
left
line
lines
listbox
listboxes
locked
make
margin
mark
maximum
menu
menus
minimum
modal
move
moved
movie
movies
name
none
open
outline
paste
pause
picture
pictures
plain
play
popup
popups
position
prepare
quit
raised
repeating
resizable
resized
right
row
rows
run
save
script
scroll
scrollable
scrolled
selection
setting
shadow
size
speed
standard
step
style
table
tables
textbox
textboxes
time
title
titled
type
underline
valid
version
visible
volume
white
width
window
windows
wrapped
zoomable
zoomed

Appendix F: How to Write Forms

A folder called “How to Write Forms” in included on your FaceSpan disks. It contains a “Read Me” text file and source code for several example forms.

Appendix G:
Speed Enhancement Tips

To speed up the opening of projects, try the following tips:

➤ Within the info dialog of Editable or Lockable text boxes, deselect the Mixed Styles checkbox. This only works if you are not mixing the stylization of text within these boxes.

➤ If you have many windows that will be accessed by a user, try opening them all with the **`visible`** set to **`false`**. Then, when a window needs to be displayed, instead of opening the window, set its **`visible`** property to **`true`**, and instead of closing the window, set its **`visible`** property to **`false`**.

To speed up the running of a FaceSpan project application, try the following tips:

➤ Unless needed, avoid placing scripts behind every item of a window. You can place as much of the script as possible at the window or project script level.

➤ Try opening your application in ResEdit and setting “preload” of specific resources in the “get resource info.” This requires a larger memory allocation.

Index

Index