JAVASCRIPT TOOLS GUIDE

Adobe

Adobe® Creative Suite 3 JavaScript Tools Guide for Windows® and Macintosh®.

Adobe Systems Incorporated, 345 Park Avenue, San Jose, California 95110, USA.

Contents

1 Introduction

JavaScript is a platform-independent scripting language that you can use to control many features and automate many tasks in Adobe® applications. Scripting is easier to learn and use than many other kinds of programming, and provides a convenient way of automating repetetive tasks or extending applications to provide additional tools for other users.

- If you are new to scripting, see *Adobe Creative Suite: Introduction to Scripting*, which introduces basic scripting concepts and describes different scripting languages that are available, including JavaScript. JavaScript and other scripting languages are object-oriented, and this book also describes the basic concepts of object-oriented programming and document object models.
- Each application that supports JavaScript also provides an application-specific Scripting Guide that introduces the object model for that application, and reference material for the objects. This document provides information about the JavaScript features, tools, and objects that are common to all Adobe applications that support JavaScript.
- This document does not teach JavaScript. If you are familiar with scripting or programming in general, but unfamiliar with JavaScript, see publicly available Web resources and documents, such as:
  - The public JavaScript standards organization web site: www.ecma-international.org
  - "JavaScript: The Definitive Guide," David Flanagan, O'Reily Media Inc, 2002. ISBN 0-596-00048-0
  - "JavaScript Bible," Danny Goodman, Hungry Minds Inc, 2001. ISBN 0-7645-4718-6
  - "Adobe Scripting," Chandler McWilliams, Wiley Publishing, Inc., 2003. ISBN 0-7645-2455-0

Note: Complete details of some new features in Adobe® Creative Suite 3 are not available at time of publication. Check for updated versions of this document at http://partners.adobe.com.

ExtendScript Overview

Adobe provides an extended implementation of JavaScript, called ExtendScript, that is used by all Adobe applications that provide a scripting interface. In addition to implementing the JavaScript language according to the ECMA JavaScript specification, ExtendScript provides certain additional features and utilities.

This document describes JavaScript modules, tools, utilities, and features that are available to all JavaScript-enabled Adobe applications.

Note: Some modules, and features of some modules, are optional. Check the product documentation for each application for details of which modules and features are implemented.

➤ Example code

The Adobe Bridge SDK, which contains this document, also contains a set of code samples ("Snippets"), that demonstrate how to use the features of ScriptUI and interapplication communication. This book refers to these samples by name for illustration of concepts and techniques. You can download the SDK from http://partners.adobe.com.

Development and debugging tools

For help in developing, debugging, and testing scripts, Adobe provides the ExtendScript Toolkit, an interactive development and testing environment for ExtendScript., which is installed with all JavaScript-enabled applications. For complete details, see Chapter 2, "The ExtendScript Toolkit."

ExtendScript also provides global objects that support development and debugging:

- A global debugging object, the Dollar ($) object.
- A reporting utility for ExtendScript elements, the ExtendScript Reflection Interface.

For complete details, see Chapter 8, "ExtendScript Tools and Features."

Cross-platform file-system access

Adobe ExtendScript defines `File` and `Folder` classes that simplify cross-platform file-system access. These classes are available to all applications that support a JavaScript interface.

For complete details, see Chapter 3, "File System Access."

User-interface development tools

Adobe provides the *ScriptUI* module, which works with the ExtendScript JavaScript interpreter to provide JavaScript scripts with the ability to create and interact with user interface elements. It provides an object model for windows and UI control elements within an Adobe application.

For complete details, see Chapter 4, "User Interface Tools."

In addition, ExtendScript provides:

- Global functions for localization of display strings; see 'Localizing ExtendScript Strings' on page 204
- Global functions for displaying short messages in dialog boxes; see 'User Notification Dialogs' on page 208.
- An object type for specifying measurement values together with their units; see 'Specifying Measurement Values' on page 211.

Interapplication communication and messaging

ExtendScript provides a common scripting environment for all Adobe JavaScript-enabled applications, and allows interapplication communication through scripts.

Different levels of communication are provided through the *cross-DOM* and the *messaging framework*.

- Cross-DOM Functions are a limited set of basic functions common across all message-enabled applications, which allow your script to, for example, open or print files in other applications, simply by calling the `open` or `print` function for that application.

  In addition to the basic set of common functions, some applications provide more extensive sets of exported JavaScript functions to other applications.
- The interapplication message framework is an application programming interface (API) that allows extensive control over communication between applications. The API allows you to send messages to other applications and receive results, and to receive messages sent by other applications and return results. Typically the data passed between applications are JavaScript scripts. However, the messaging

framework is extensible. It allows you to define different types of data to send between applications, and to specify how they are handled.

For complete details, see Chapter 5, "Interapplication Communication with Scripts."

External communication

ExtendScript offers tools for communicating with other computers or the internet using standard protocols. These objects support external communication:

- The Web Access library defines the FtpConnection Object, which supports FTP and SFTP communication protocols, and the HttpConnection Object, which supports HTTP and HTTPS communication protocols.
- The Socket Object supports low-level TCP connections.

For complete details, see Chapter 6, "External Communication Tools."

External shared-library integration

You can extend the JavaScript DOM for an application by writing a C or C++ shared library, compiling it for the platform you are using, and loading it into JavaScript as an `ExternalObject` instance. A shared library is implemented by a DLL in Windows, a bundle or framework in Mac OS, or a SharedObject in Unix.

For complete details, see Chapter 7, "Integrating External Libraries."

Additional utilities and features

ExtendScript provides these utilities and features:

- JavaScript language enhancements:
  - Tools for combining scripts, such as a `#include` directive, and `import` and `export` statements. See 'Modular Programming Support' on page 215.
  - Support for extending or overriding math and logical operator behavior on a class-by-class basis. See 'Operator Overloading' on page 218.

  For complete details, see Chapter 8, "ExtendScript Tools and Features."
- JavaScript compilation, through the ExtendScript Toolkit. See 'The ExtendScript Toolkit' on page 12.
- XML integration: ExtendScript defines the `XML` object, which allows you to process XML with your JavaScript scripts. For complete details, see Chapter 9, "Integrating XML into JavaScript."

Scripting for Specific Applications

On startup, all Adobe JavaScript-enabled applications execute JSX files that they find in their startup directories; some of these are installed by applications, and some can be installed by scripters. The policies of different applications vary as to the locations, write access, and loading order.

In addition, individual applications may look for application-specific scripts in particular directories, which may be configurable. Some applications allow access to scripts from menus; all of them allow you to load and run scripts using the ExtendScript Toolkit.

For details of how to load and run scripts for any individual application, see the *JavaScript Scripting Guide* for that application.

Startup scripts

A script in a startup directory might be executed on startup by multiple applications. If you place a script in such a directory, it must contain code to check whether it is being run by the intended application. You can do this using the `appName` static property of the `BridgeTalk` class. For example:

```
if( BridgeTalk.appName == "bridge" ) {
      //continue executing script
   }
```

If a script that is run by one application will communicate with another application, or add functionality that depends on another application, it must first check whether that application and version is installed. You can do this using the `BridgeTalk.getSpecifier()` static function. For example:

```
if( BridgeTalk.appName == "bridge-2.0" ) {
// Check to see that Photoshop is installed.
   if( BridgeTalk.getSpecifier("photoshop",10)){
      // Add the Photoshop automate menu to the Adobe Bridge UI.
   }
}
```

For details of interapplication communication, see Chapter 5, "Interapplication Communication with Scripts."

JavaScript variables

Scripting shares a global environment, so any script executed at startup can define variables and functions that are available to all scripts. In all cases, variables and functions, once defined by running a script that contains them, persist in subsequent scripts during a given application session. Once the application is quit, all such globally defined variables and functions are cleared. Scripters should be careful about giving variables in scripts unique names, so that a script does not inadvertently reassign global variables intended to persist throughout a session.

2 The ExtendScript Toolkit

The ExtendScript Toolkit provides an interactive development and testing environment for ExtendScript in all JavaScript-enabled Adobe applications. It includes a full-featured, syntax-highlighting text editor with Unicode capabilities and multiple undo/redo support. The Toolkit is the default editor for ExtendScript files, which use the extension `.jsx`.

The Toolkit includes a JavaScript debugger that allows you to:

- Single-step through JavaScript scripts (JS or JSX files) inside an application.
- Inspect all data for a running script.
- Set and execute breakpoints.

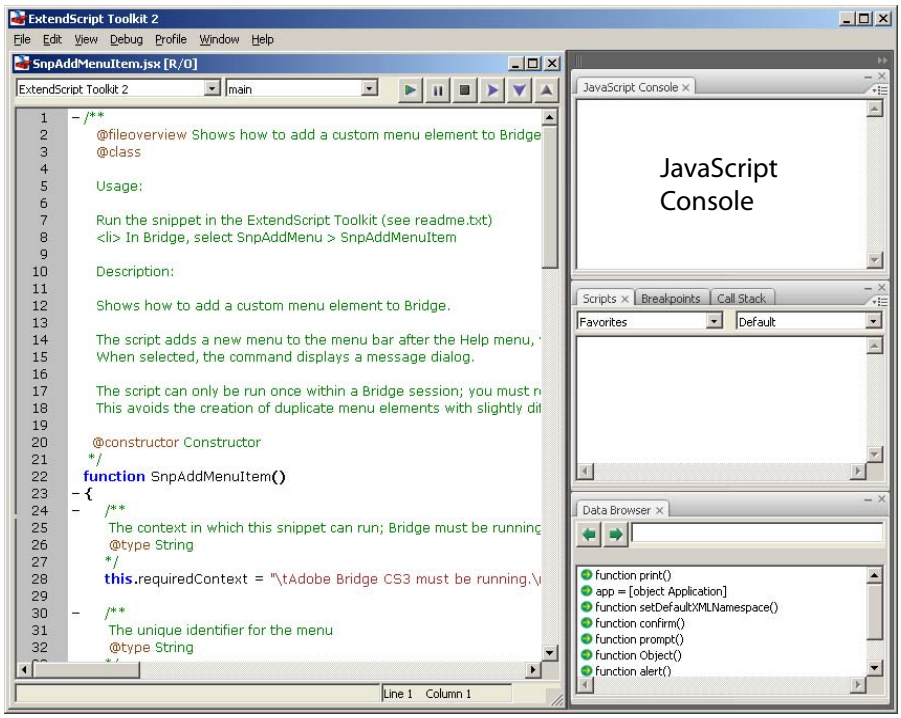
When you double click a JSX file in the platform's windowing environment, the script runs in the Toolkit, unless it specifies a particular target application using the `#target` directive. For more information, see 'Selecting a Debugging Target' on page 24 and 'Preprocessor directives' on page 215.

Tip: When you have completed editing and debugging your JavaScript script, you can compile it, using **File > Export to Binary**.

Configuring the Toolkit Window

The ExtendScript Toolkit initially appears with a default workspace arrangement, containing a default configuration of tabbed panels and Script Editor document windows. The arrangement is highly configurable, through the Window menu, the context menus of individual panels and panel groups, or directly using drag and drop.

Document window



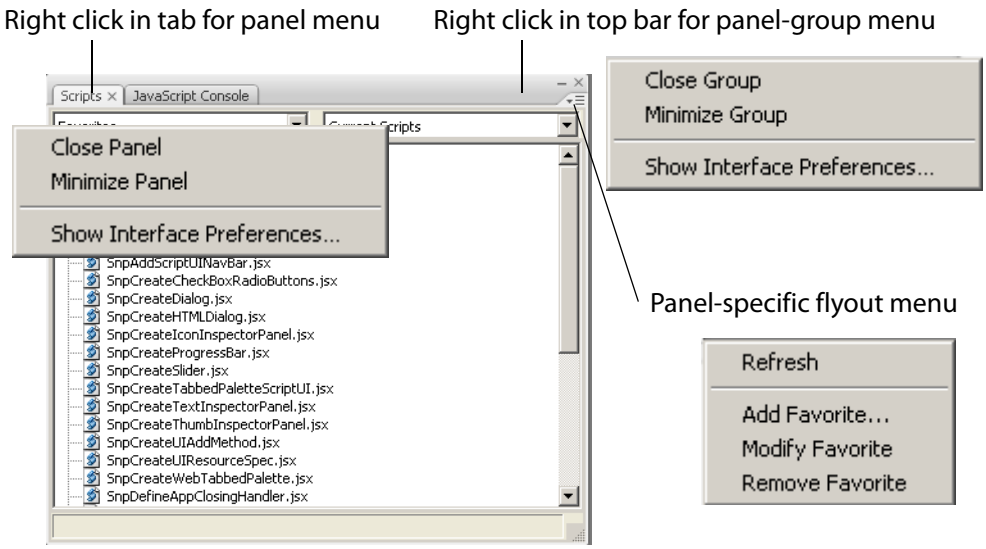
Panels

You can, for example, adjust the relative sizes of the panels by dragging the separators up or down, or right or left, and can rearrange the groupings. To move a tabbed panel, drag the tab into another pane.

If you drag a tab so that the entire destination group is highlighted, it becomes another stacked panel in that group. If you drag a tab to the top or bottom of a group (so that only the top or bottom bar of the destination group is highlighted), that group splits to show the panels in a tiled format.

- You can dock the entire panel group to different edges of the Toolkit window.
- You can collapse the entire panel group, then expose individual panels.
- You can open and close, or collapse and expand individual panels, regardless of the dock state.
- You can undock individual tabs, making them floating panels.

Panel menus



Both individual panels and panel groups have context menus, which you invoke with a right click in the tab or on the background of the title bar. These menus have panel-control commands, including **Close Panel** and **Close Group** to hide the individual panel or entire group.

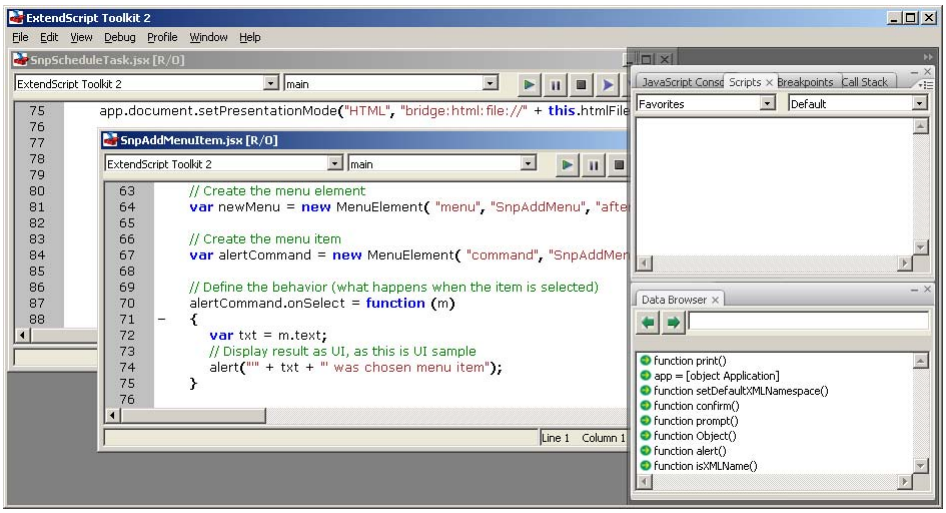
- You can also show or hide specific panels by toggling them on or off in the **Window** menu. Use the **Window** menu to show a hidden panel, or to bring a floating panel to the front.
- Use **Window > Hide panels** to close all of the panels.

Some panels also have a flyout menu, specific to that panel, which you access through the menu icon in the upper right corner.

The individual panels are discussed in detail in the following sections.

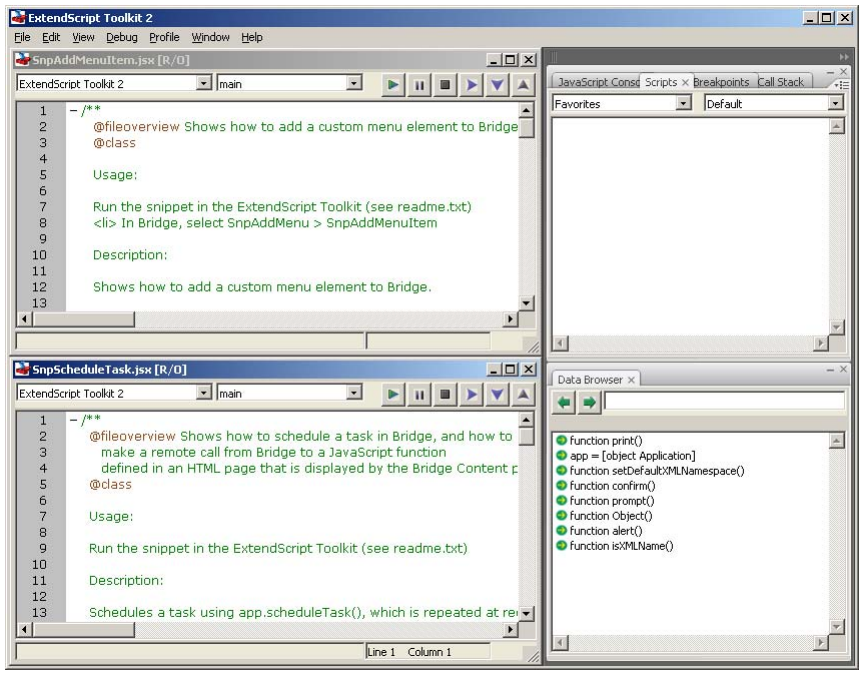
Document windows

When you open scripts or text files, each file appears in its own Script Editor document window. You can use the Window menu to arrange multiple document windows in the default cascade display style (overlapping windows) or tiled display style (non-overlapping windows). In the cascade style, a window that you move into the panel area goes behind any visible panels.

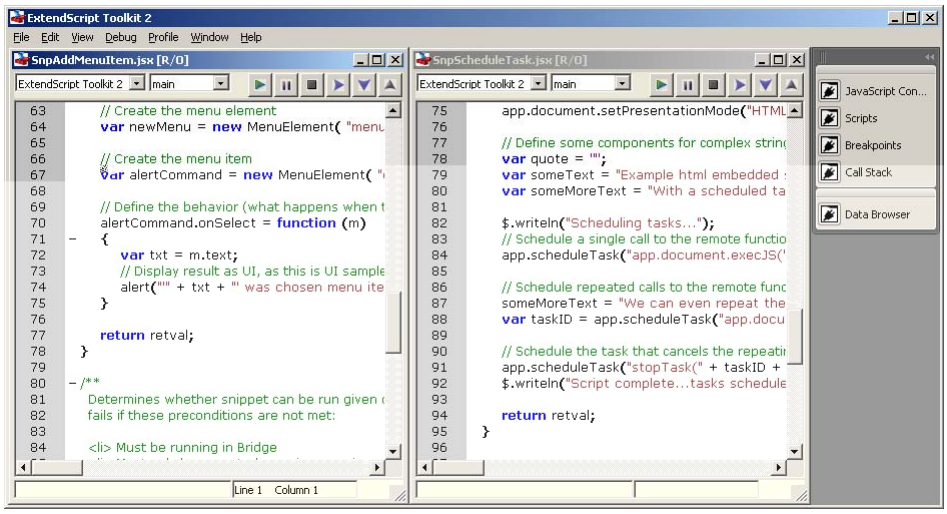


Document windows, cascade style Docked panels

If you choose a tiled display style, the open document windows are automatically arranged into non-overlapping tiles, either vertical or horizontal, that fill the space between the edge of the window and the edge of the current panel configuration.



Document windows, horizontal tile style Docked panels



Document windows, vertical tile style Minimized docked panels

You can edit or run scripts in multiple document windows simultaneously. The current document window is highlighted and has the input focus. You can select another document window by clicking in it, or you can switch between them with the commands **Window > Next document** and **Window > Previous document**. The default keyboard shortcuts for these commands are F6 and SHIFT-F6; you can change these using the Keyboard Shortcuts page in the Preferences dialog (**Edit > Preferences**).

For more information about the document windows and the Script Editor, see 'The Script Editor' on page 17.

Workspaces

The Toolkit saves the current layout when you exit, and restores it at the next startup. It also saves and restores the open documents, the current positions within the documents, and any breakpoints that have been set.

- If you do not want to restore all settings on startup, hold SHIFT while the Toolkit loads to restore default settings.
- The Startup page in the Preferences dialog (**Edit > Preferences**) offers a choice of whether to open a blank document window, no document window, or display a previously opened document on startup.
- The Keyboard Shortcuts page in the Preferences dialog (**Edit > Preferences**) allows you to set or modify keyboard shortcuts for all menu commands. If you do this, you are responsible for avoiding collisions among key combinations.

Selecting Scripts

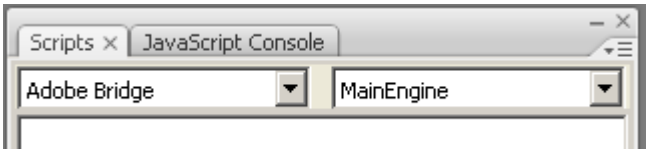
You can open multiple scripts (or text files, including programs in other languages) You can find and open script in a number of ways:

- Use **File > Open** to bring up the platform-specific file browser.
- Choose from recently opened files using **File > Recent files**
- Create a new script using **File > New JavaScript**.
- Drop files from the Explorer or the Finder onto the Toolkit to open them in a document window.
- For JavaScript scripts in trusted locations (the user-script folders of installed Adobe applications), a double-click on the file runs it in the target application or in the Toolkit. For script files in other locations, you must confirm that you want to run the script.
- Search for scripts containing particular text using **Edit > Find/replace**. You can search in a particular document window, among all scripts open in document windows, or among scripts associated with an application, or kept in favorite locations. See 'Searching in text' on page 21.
- Use the Script panel to display and open scripts made available by loaded Adobe applications, or those kept in favorite locations.

The Scripts panel and favorite script locations

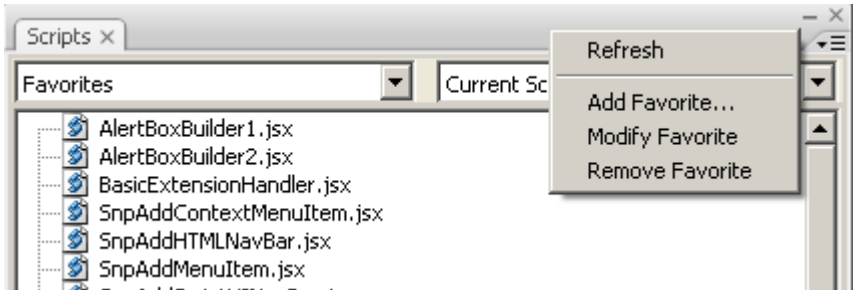
The Scripts panel offers a list of debuggable scripts, which can be JS or JSX files or (for some applications) HTML files that contain embedded scripts.

You can display a list of scripts made available by a particular target application. Select the target application in the leftmost drop-down list; the available JavaScript engines for that application become available in the right-hand list.

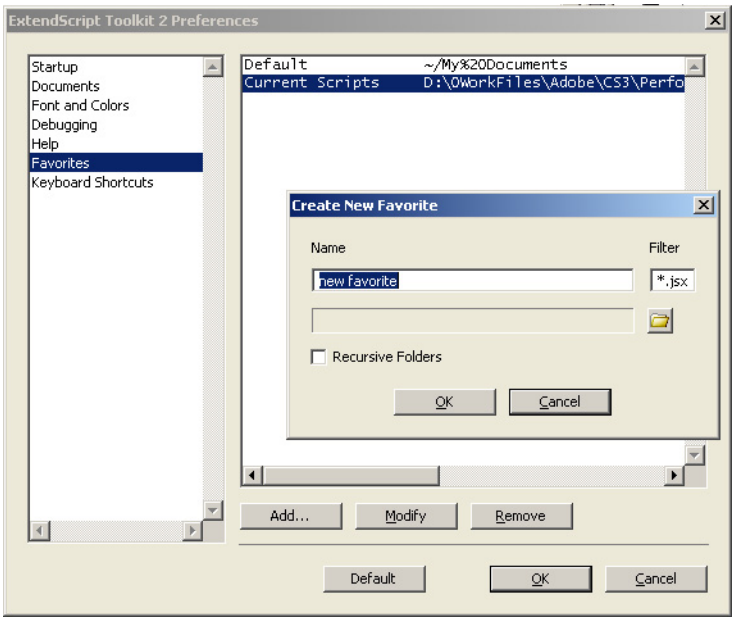


When you select a target application, the Toolkit offers to open that application if it is not running, then displays the scripts which that application makes public. Select a script in this panel to load it and display its contents in a new document window, where you can modify it, save it, or run it within the target application.

When you choose the target Favorites, the right-hand list shows the default favorite script location, and any other favorite locations that have been defined. You can create your own list of favorite script locations using the flyout menu.



You can also examine and set favorite locations using the Favorites page of the Preferences dialog (**Edit > Preferences**). Use the **Add**, **Modify**, and **Remove** buttons to edit the list of folders.



The favorite script locations that you define are also available to the Find and Replace dialog; see 'Searching in text' on page 21.

The Script Editor

The Script Editor is a full-featured source code editor for JavaScript. You can open any number of Script Editor document windows; each displays one Unicode source code document.

The Script Editor offers many useful and powerful text editing and navigation features. Some are intended specifically for use with JavaScript, while others are useful for all kinds of text editing. Features include:

- Navigation aids and options applicable to any kind of text, and specific code navigation for JavaScript; see 'Navigation aids' on page 18.
- General editing and coding support such as undo-redo, and specific JavaScript coding support such as syntax checking; see 'Coding aids' on page 20.
- A full-featured text search tool that can search in multiple files; see 'Searching in text' on page 21.
- Syntax marking (color and font styles for specific syntactic structures) for JavaScript and for many other computer languages. The marking styles are configurable; see 'Syntax marking' on page 23.

Navigation aids

You can configure the Script Editor to display text with various feature that help you track the structure of your code, or that help you move around in the file. It also offers mouse and keyboard shortcuts for specific types of cursor movement and text selection.

View options

The Script Editor offers a number of viewing options that aid in code navigation, including the following:

- Automatic line numbering. **View > Line Numbers** toggles numbering on and off.
- A collapsible tree view of code, where you can open or close logical units of the structure, such as comments or function definitions. **View > Code Collapse** toggles the tree view on and off.
- A line-wrapping mode, where there is no horizontal scroll bar, and lines are wrapped at word breaks. **View > Word Wrap** toggles line-wrapping on and off.
- Syntax marking, which uses color and font styles to highlight specific syntactic structures. **View > Syntax Highlighting** allows you to turn syntax marking off, or set it to mark a particular language, JavaScript or many other computer languages. The marking styles are configurable; see 'Syntax marking' on page 23.

You can set the default values for any of these states using the Documents page of the Preferences dialog (**Edit > Preferences**).

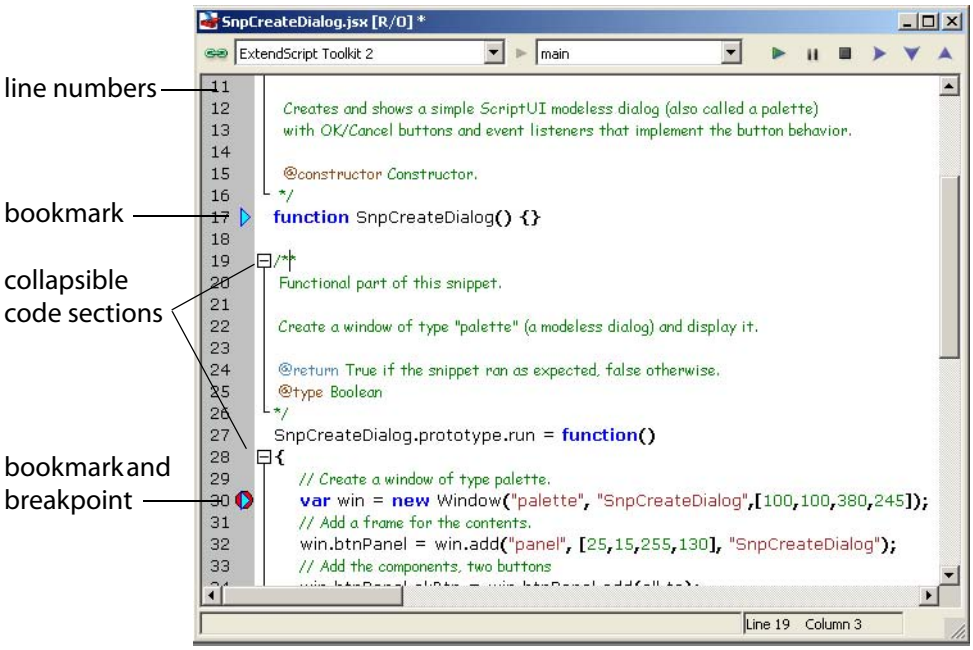
Bookmarks

The **Edit > Bookmarks** menu allows you to set and clear navigation points in your text. The F2 function key is the default shortcut key for the bookmark commands:

- Toggle the bookmark for the current line using CTRL-F2.
- Move the cursor to the next bookmark with F2, or to the previous one with SHIFT-F2. The bookmarks wrap, so that the first follows the last.
- Use SHIFT-CTRL-F2 to clear all bookmarks in the current text.

When you navigate to a bookmark in a collapsed section of code, that section automatically opens.

Bookmarks are marked with a blue, right-pointing arrow at the left of the line (to the right of the line number if it is shown). This is the same place where a breakpoint is marked with a dot (see 'Setting breakpoints' on page 27). If you have both a breakpoint and a bookmark set in the same line, the blue arrow is superimposed on the breakpoint dot.



Mouse navigation and selection

You can use the mouse or special keyboard shortcuts to move the insertion point or to select text in the document window. Click the left mouse button in the document window to move the position caret.

To select text with the mouse, click in unselected text, then drag over the text to be selected. If you drag above or below the currently displayed text, the text scrolls, continuing to select while scrolling. You can also double-click to select a word, or triple-click to select a line.

To initiate a drag-and-drop of selected text, click in the block of selected text, then drag to the destination. You can drag text from one document window to another. You can also drag text out of the Toolkit into another application that accepts dragged text, and drag text from another application into a Toolkit document window.

You can drop files from the Explorer or the Finder onto the Toolkit to open them in a document window.

Keyboard navigation and selection

The Keyboard Shortcuts page in the Preferences dialog (**Edit > Preferences**) allows you to set or modify keyboard shortcuts for all menu commands. If you do this, you are responsible for avoiding collisions among key combinations.

In addition to the keyboard shortcuts specfied for menu commands, and the usual keyboard input, the document window accepts these special movement keys. You can also select text by using a movement key while pressing SHIFT.

Enter	Insert a Line Feed character
Backspace	Delete character to the left
Delete	Delete character to the right

Left arrow	Move insertion point left one character
Right arrow	Move insertion point right one character
Up arrow	Move insertion point up one line; stay in column if possible
Down arrow	Move insertion point down one line; stay in column if possible
Page up	Move insertion point one page up
Page down	Move insertion point one page down
CTRL + Up arrow	Scroll up one line without moving the insertion point
CTRL + Down arrow	Scroll down one line without moving the insertion point
CTRL + Page up	Scroll one page up without moving the insertion point
CTRL + page down	Scroll one page down without moving the insertion point
CTRL + Left arrow	Move insertion point one word to the left
CTRL + right arrow	Move insertion point one word to the right
Home	Move insertion point to start of line
End	Move insertion point to end of line
CTRL + Home	Move insertion point to start of text
CTRL + End	Move insertion point to end of text

The Script Editor supports extended keyboard input via IME (Windows) or TMS (Mac OS). This is especially important for Far Eastern characters.

Coding aids

The Script Editor offers a number of visual and editing features that help you navigate in and maintain the syntactic structure of your JavaScript code, including:

➤ **Brace matching**

The **Edit** menu offers two kinds of brace-matching selection, that operate when the cursor is placed immediate after an opening brace character, or immediately before a closing brace:

- **Edit > Select to Brace**: Moves the cursor to the matching bracing, but does not select any text. The default keyboard shortcut is CTRL 0 (zero).
- **Edit > Select Including Brace**: Selects all text between the braces. The default keyboard shortcut is SHIFT CTRL 0 (zero).

Brace characters include parantheses, curly braces, and square brackets.

➤ **Block indentation**

To indent a block of text, select some or all of the text on the line or lines, and press TAB. To outdent, press SHIFT TAB.

➤ **Comment and uncomment commands**

Use **Edit > Comment or Uncomment Selection** to temporarily remove parts of a JavaScript program from the path of execution. This command is toggle. When you first issue the command, it places the special comment sequence `//~` at the front of any line that is wholly or partially selected. When you next issue the command with such a line selected, it removes that comment marker.

The command affects only the comment markers it places in the text; it ignores any comment markers that were already in the selected lines. This allows you to temporarily remove and replace blocks of text that include both code and comments.

➤ **Version comments**

A special comment format is reserved for a code versioning statement, which is used internally by Adobe scripts, but is available to all scripters. Use **Edit > Insert Version Tag** to insert a comment containing the file name and current date-time, in this format:

```
/**
* @@@BUILDINFO@@@ SnpCreateDialog.jsx !Version! Tue Dec 05 2006 08:03:38 GMT-0800
*/
```

You are responsible for manually updating the `!Version!` portion with your own version information.

➤ **Undo and redo**

Choose **Undo** or **Redo** from the **Edit** menu or from the document window's right-click context menu to revoke and reinstate multiple editing changes sequentially.

➤ **Syntax checking**

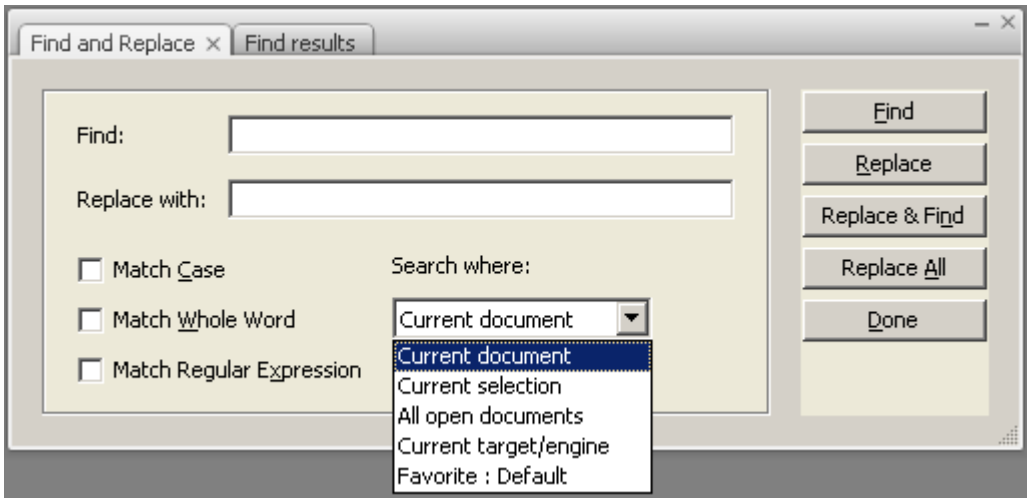
Before running the new script or saving the text as a script file, use **Edit > Check Syntax** to check whether the text contains JavaScript syntax errors. The default keyboard shortcut is F7.

- If the script is syntactically correct, the status line shows "No syntax errors".
- If the Toolkit finds a syntax error, such as a missing quote, it highlights the affected text, plays a sound, and shows the error message in the status line so you can fix the error.

Searching in text

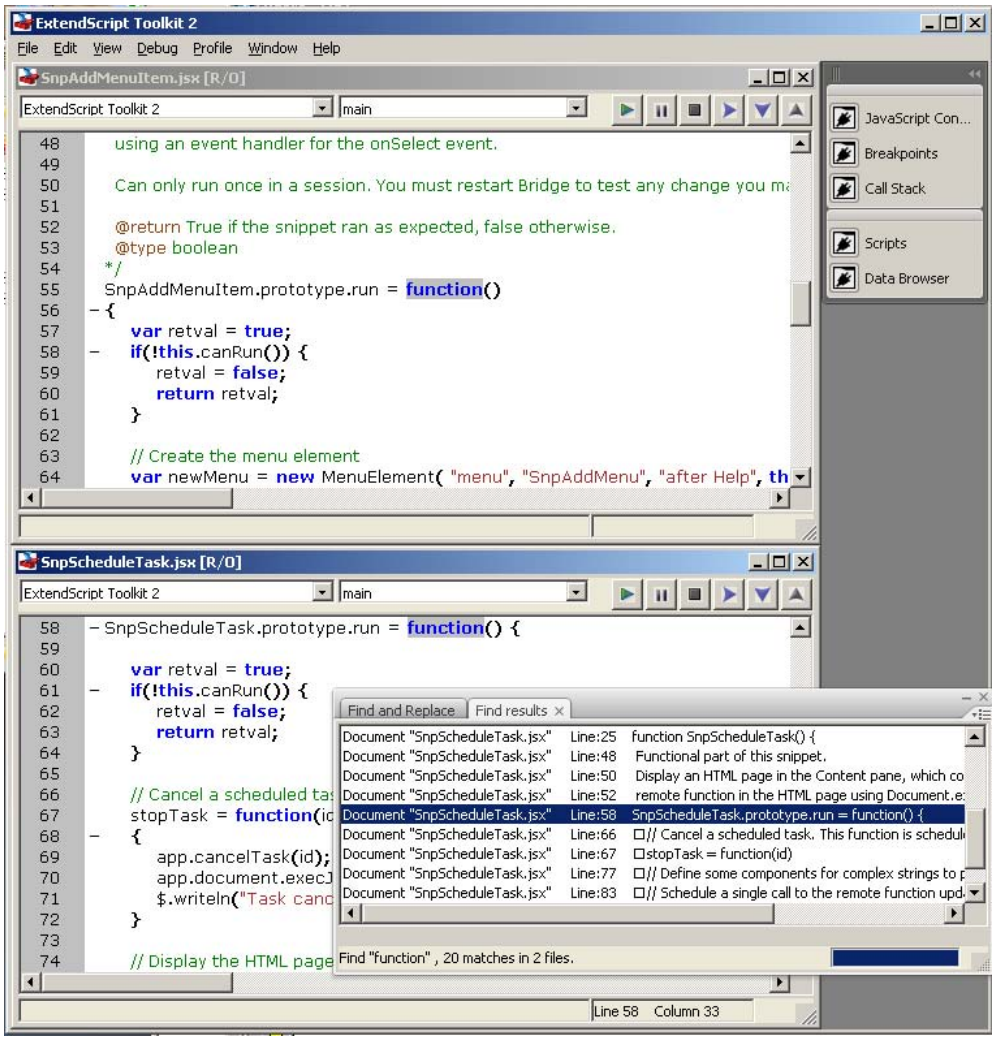
The Toolkit offers a search utility through the **Edit > Find/replace** command. This command brings up the Find and Replace panel, which allows you to search through multiple documents for text that matches a regular expression. You can choose to search in:

- The current document, or the current selection in the current document
- All open documents
- All scripts made public by the current target application
- Folders that you have defined as favorite locations; see 'The Scripts panel and favorite script locations' on page 16.



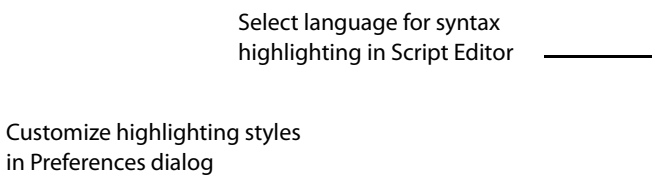
The results of a search are listed in the Find results panel; by default, this is stacked with the Find and Replace panel, but you can drag it to another stack, or display it as an independant floating panel.

Double-click a result line in the Find results panel to jump directly to the document and line where the text was found.



Syntax marking

The Script Editor offers language-based syntax highlighting to aid in editing code. Although the debugging features (including syntax checking) are only available for JavaScript, you can choose to edit other kinds of code, and the syntax is highlighted according to the language. The style of syntax marking is automatically set to match the file extension, or you can choose the language from the **View > Syntax Highlighting** menu.



The style of highlighting is configurable, using the Fonts and Colors page of the Preferences dialog.

Debugging in the Toolkit

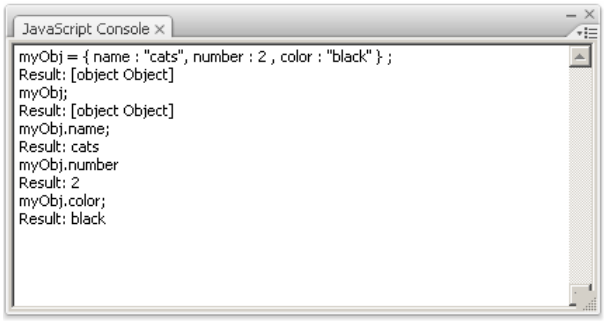
You can debug the code in the currently active document window. Select one of the debugging commands to either run or to single-step through the program.

When you run code from the document window, it runs in the current target application's selected

first checks for syntax errors in the script. If any are found, the Toolkit displays the error in a message box and quits silently, rather than launching the target application. For example:

The JavaScript console

The JavaScript console is a command shell and output window for the currently selected JavaScript engine. It connects you to the global namespace of that engine.



The console is a JavaScript listener, that expects input text to be JavaScript code.

You can use the console to evaluate expressions or call functions. Enter any JavaScript statement and execute it by pressing ENTER. The statement executes within the stack scope of the line highlighted in the Call Stack panel, and the result appears in the next line.

- The right-click context menu provides the same editing commands as that of the document window. You can copy, cut, and paste text, and undo and redo previous actions.
- You can select text with the mouse, and use the normal copy and paste shortcuts.
- Use the up- and down-arrow keys to scroll through previous entries, or place the cursor with the mouse. Pressing ENTER executes the line that contains the cursor.
- The flyout menu allows you to clear the current content.

Commands entered in the console execute with a timeout of one second. If a command takes longer than one second to execute, the Toolkit generates a timeout error and terminates the attempt.

The console is the standard output location for JavaScript execution. If any script generates a syntax error, the error is displayed here along with the file name and the line number. The Toolkit displays errors here during its own startup phase.

Controlling code execution

The debugging commands are available from the **Debug** menu, from the document window's right-click context menu, through keyboard shortcuts, and from the toolbar buttons. Use these menu commands and buttons to control the execution of code when the JavaScript Debugger is active.

	**Run** **Continue**	**F5** (Windows) **Ctrl R** (Mac OS)	Starts or resumes execution of a script. Disabled when script is executing.
	**Break**	**Ctrl F5** (Windows) **Cmd .** (Mac OS)	Halts the currently executing script temporarily and reactivates the JavaScript Debugger. Enabled when a script is executing.
	**Stop**	**Shift F5** (Windows) **Ctrl K** (Mac OS)	Stops execution of the script and generates a runtime error. Enabled when a script is executing.
	**Step Over**	**F10** (Windows) **Ctrl S** (Mac OS)	Halts after executing a single JavaScript line in the script. If the statement calls a JavaScript function, executes the function in its entirety before stopping (do not step into the function).
	**Step Into**	**F11** (Windows) **Ctrl T** (Mac OS)	Halts after executing a single JavaScript line statement in the script or after executing a single statement in any JavaScript function that the script calls.
	**Step Out**	**Shift F11** (Windows) **Ctrl U** (Mac OS)	When paused within the body of a JavaScript function, resumes script execution until the function returns. When paused outside the body of a function, resumes script execution until the script terminates.

Visual indication of execution states

While the engine is running, an icon  in the lower right corner of the Toolkit window indicates that the script is active.

When the execution of a script halts because the script reached a breakpoint, or when the script reaches the next line when stepping line by line, the document window displays the current script with the current line highlighted in yellow.

If the script encounters a runtime error, the Toolkit halts the execution of the script, displays the current script with the current line highlighted in red, displays the error message in the status line, and plays a sound.

Scripts often use a `try`/`catch` clause to execute code that may cause a runtime error, in order to catch the error programmatically rather than have the script terminate. You can choose to allow regular processing of such errors using the `catch` clause, rather than breaking into the debugger. To set this behavior, choose **Debug** > **Don't Break On Guarded Exceptions**. Some runtime errors, such as `Out Of Memory`, always cause the termination of the script, regardless of this setting.

Setting breakpoints

When debugging a script, it is often helpful to make it stop at certain lines so that you can inspect the state of the environment, whether function calls are nested properly, or whether all variables contain the expected data.

- To stop execution of a script at a given line, click to the left of the line number to set a breakpoint. A red dot indicates the breakpoint.
- Click a second time to temporarily disable the breakpoint; the icon changes color.
- Click a third time to delete the breakpoint. The icon is removed.

Some breakpoints need to be conditional. For example, if you set a breakpoint in a loop that is executed several thousand times, you would not want to have the program stop each time through the loop, but only on each 1000th iteration.

You can attach a condition to a breakpoint, in the form of a JavaScript expression. Every time execution reaches the breakpoint, it runs the JavaScript expression. If the expression evaluates to a nonzero number or `true`, execution stops.

To set a conditional breakpoint in a loop, for example, the conditional expression could be `"i >= 1000"`, which means that the program execution halts if the value of the iteration variable `i` is equal to or greater than 1000.

You can set breakpoints on lines that do not contain any code, such as comment lines. When the Toolkit runs the program, it automatically moves such a breakpoint down to the next line that actually contains code.

The Breakpoints panel

The Breakpoints panel displays all breakpoints set in the current document window. You can use the panel's flyout menu to add, change, or remove a breakpoint.

You can edit a breakpoint by double-clicking it, or by selecting it and choosing **Add** or **Modify** from the panel menu. A dialog allows you to change the line number, the breakpoint's enabled state, and the condition statement.

Whenever execution reaches this breakpoint, the debugger evaluates this condition. If it does not evaluate to `true`, the breakpoint is ignored and execution continues. This allows you to break only when certain conditions are met, such as a variable having a particular value.

Breakpoint icons

Each breakpoint is indicated by an icon to the left of the line number in the document window, and an icon and line number in the Breakpoints panel. In the Breakpoints panel, the icon for a conditional breakpoint is a diamond, while the icon for an unconditional breakpoint is round. Disabled breakpoints are indicated by an outline icon, while active ones are filled.

	Unconditional breakpoint. Execution stops here.
	Unconditional breakpoint, disabled. Execution does not stop.
	Conditional breakpoint. Execution stops if the attached JavaScript expression evaluates to `true`.
	Conditional breakpoint, disabled. Execution does not stop.

In the document window, all breakpoint icons are round, and disabled breakpoint icons are black.

Evaluation in help tips

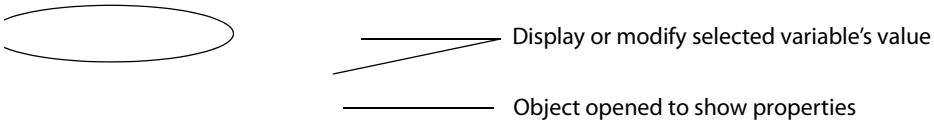
If you let your mouse pointer rest over a variable or function in an document window, the result of evaluating that variable or function is displayed as a help tip. When you are not debugging the program, this is helpful only if the variables and functions are already known to the JavaScript engine. During debugging, however, this is an extremely useful way to display the current value of a variable, along with its current data type.

Tracking data

The Data Browser panel is your window into the JavaScript engine. It displays all live data defined in the current context, as a list of variables with their current values. If execution has stopped at a breakpoint, it shows variables that have been defined using `var` in the current function, and the function arguments. To show variables defined in the global or calling scope, use the Call Stack to change the context (see 'The call stack' on page 29).

You can use the Data Browser to examine and set variable values.

- Click a variable name to show its current value in the edit field at the top of the panel.
- To change the value, enter a new value and press ENTER. If a variable is Read only, the edit field is disabled.



The flyout menu for this panel lets you control the amount of data displayed:

- **Show Global Functions** toggles the display of all global function definitions.
- **Show Object Methods** toggles the display of all functions that are attached to objects. Most often, the interesting data in an object are its callable methods.
- **Show JavaScript Language Elements** toggles the display of all data that is part of the JavaScript language standard, such as the Array constructor or the Math object. An interesting property is the `__proto__` property, which reveals the JavaScript object prototype chain.

Each variable has a small icon that indicates the data type. An invalid object is a reference to an object that has been deleted. If a variable is undefined, it does not have an icon.

	**null**
	**Boolean**
	**Number**
	**String**
	**Object**
	**Invalid object**

You can inspect the contents of an object by clicking its icon. The list expands to show the object's properties (and methods, if **Show Object Methods** is enabled), and the triangle points down to indicate that the object is open.

The call stack

The Call Stack panel is active while debugging a program. When an executing program stops because of a breakpoint or runtime error, the panel displays the sequence of function calls that led to the current

execution point. The Call Stack panel shows the names of the active functions, along with the actual arguments passed in to that function.

For example, this stack pane shows a break occurring at a breakpoint in a function `dayOfWeek`:

The function containing the breakpoint is highlighted in both the Call Stack panel and the document window.

You can click any function in the call hierarchy to inspect it. In the document window, the line containing the function call that led to that point of execution is marked with a green background. In the example, when you select the line `[Top Level]` in the call stack, the Document Window highlights the line where the `dayOfWeek` function was called.

Switching between the functions in the call hierarchy allows you to trace how the current function was called. The Console and Data Browser panels coordinate with the Call Stack pane. When you select a function in the Call Stack:

- The Console pane switches its scope to the execution context of that function, so you can inspect and modify its local variables. These would otherwise be inaccessible to the running JavaScript program from within a called function.
- The Data Browser pane displays all data defined in the selected context.

Code Profiling for Optimization

The Profiling tool helps you to optimize program execution. When you turn profiling on, the JavaScript engine collects information about a program while it is running. It counts how often the program executed a line or function, or how long it took to execute a line or function. You can choose exactly which profiling data to display.

Because profiling significantly slows execution time, the **Profile** menu offers these profiling options.

Off	Profiling turned off. This is the default.
Functions	The profiler counts each function call. At the end of execution, displays the total to the left of the line number where the function header is defined.

Lines	The profiler counts each time each line is executed. At the end of execution, displays the total to the left of the line number. Consumes more execution time, but delivers more detailed information.
Add Timing Info	Instead of counting the functions or lines, records the time taken to execute each function or line. At the end of execution, displays the total number of microseconds spent in the function or line, to the left of the line number. This is the most time-consuming form of profiling.
No Profiler Data	When selected, do not display profiler data.
Show Hit Count	When selected, display hit counts.
Show Timing	When selected, display timing data.
Erase Profiler Data	Clear all profiling data.
Save Data As	Save profiling data as comma-separated values in a CSV file that can be loaded into a spreadsheet program such as Excel.

When execution halts (at termination, at a breakpoint, or due to a runtime error), the Toolkit displays this information in the Document Window, line by line. The profiling data is color coded:

- Green indicates the lowest number of hits, or the fastest execution time.
- Red indicates trouble spots, such as a line that has been executed many times, or which line took the most time to execute.

This example displays timing information for the program, where the fastest line took 4 microseconds to execute, and the slowest line took 29 microseconds. The timing might not be accurate down to the microsecond; it depends on the resolution and accuracy of the hardware timers built into your computer.

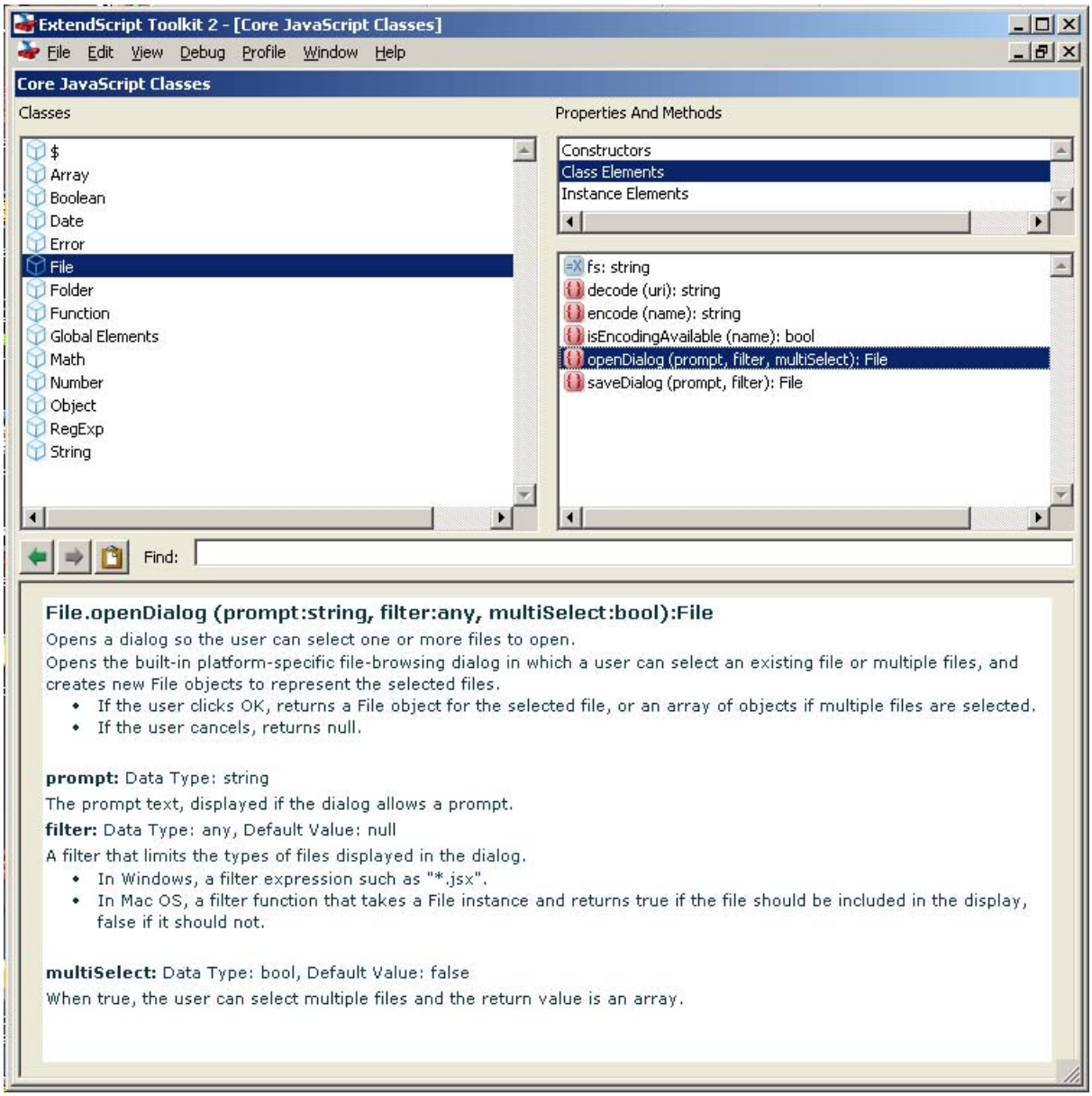
Inspecting Object Models

The ExtendScript Toolkit offers the ability to inspect the object model of any loaded dictionary. The Core JavaScript Classes dictionary includes Adobe tools and utilities such as `File` and `Folder`. Each Adobe application defines a dictionary for that application's Document Object Model (DOM), and the JavaScript ScriptUI user-interface module defines its own dictionary. Loaded dictionaries are listed in the Help menu.

Note: The dictionary for a particular application may not be available until you launch that application, or until you select it as a target in the Toolkit.

To inspect an object model, select the dictionary from the Help menu.

The Object Model Viewer appears, allowing you to browse through the object hierarchy and inspect the type and description of each property, and the description and parameters for each method.



3 File System Access

Adobe ExtendScript defines classes that simplify cross-platform file-system access. These classes are available to all applications that support a JavaScript interface.

- The first part of this chapter, Using File and Folder Objects, describes how to use these classes and provides details of pathname syntax .
- 'File and Folder Object Reference' on page 41 provides reference details of the objects, properties, methods, and creation parameters . You can also choose the Core JavaScript Classes dictionary from the Help menu in the ExtendScript Toolkit to inspect the objects in the Object Model Viewer.

Using File and Folder Objects

Because path name syntax is very different on Windows, Mac OS and UNIX®, Adobe ExendScript defines the `File` and `Folder` objects to provide platform-independent access to the underlying file system. A `File` object represents a disk file, a `Folder` object represents a directory or folder.

- The `Folder` object supports file system functionality such as traversing the hierarchy; creating, renaming or removing files; or resolving file aliases.
- The `File` object supports input/output functions to read or write files.

There are several ways to distinguish between a `File` and a `Folder` object. For example:

```
if (f instanceof File) ...
if (typeof f.open == "undefined") ...// Folders do not open
```

`File` and `Folder` objects can be used anywhere that a path name is required, such as in properties and arguments for files and folders.

Note: When you create two `File` objects that refer to the same disk file, they are treated as distinct objects. If you open one of them for I/O, the operating system may inhibit access from the other object, because the disk file already is open.

Specifying paths

When creating a `File` or `Folder` object, you can specify a platform-specific path name, or an absolute or relative path in a platform-independent format known as *universal resource identifier* (*URI*) notation. The path stored in the object is always an absolute, full path name that points to a fixed location on the disk.

- Use the `toString` method to obtain the name of the file or folder as string containing an absolute path name in URI notation.
- Use the `fsName` property to obtain the platform-specific file name.

Absolute and relative path names

An absolute path name in URI notation describes the full path from a root directory down to a specific file or folder. It starts with one or two slashes (`/`), and a slash separates path elements. For example, the following describes an absolute location for the file `myFile.jsx`:

```
/dir1/dir2/mydir/myFile.jsx
```

A relative path name in URI notation is appended to the path of the current directory, as stored in the globally-available `current` property of the `Folder` class. It starts with a folder or file name, or with one of the special names dot (`.`) for the current directory, or dot dot (`..`) for the parent of the current directory. A slash (`/`) separates path elements. For example, the following paths describe various relative locations for the file `myFile.jsx`:

`myFile.jsx` `./myFile.jsx`	In the current directory.
`../myFile.jsx`	In the parent of the current directory.
`../../myFile.jsx`	In the grandparent of the current directory.
`../dir1/myFile.jsx`	In `dir1`, which is parallel to the current directory.

Relative path names are independent of different volume names on different machines and operating systems, and therefore make your code considerably more portable. You can, for example, use an absolute path for a single operation, to set the current directory in the `Folder.current` property, and use relative paths for all other operations. You would then need only a single code change to update to a new platform or file location.

Character interpretation in paths

There are some platform differences in how pathnames are interpreted:

- On Windows and Mac OS, path names are not case sensitive. In UNIX, paths are case sensitive.
- On Windows, both the slash (`/`) and the backslash (`\`) are valid path element separators. Backslash is the escape character, so you must use a double backslash (`\\`) to indicate the character.
- On Mac OS, both the slash (`/`) and the colon (`:`) are valid path element separators.

If a path name starts with two slashes (or backslashes on Windows), the first element refers to a remote server. For example, `//myhost/mydir/myfile` refers to the path `/mydir/myfile` on the server `myhost`.

URI notation allows special characters in pathnames, but they must specified with an escape character (`%`) followed by a hexadecimal character code. Special characters are those which are not alphanumeric and not one of the characters:

```
/ - — . ! ~ * ' ( )
```

A space, for example, is encoded as `%20`, so the file name `"my file"` is specified as `"my%20file"`. Similarly, the character `ä` is encoded as `%E4`, so the file name `"Bräun"` is specified as `"Br%E4un"`.

This encoding scheme is compatible with the global JavaScript functions `encodeURI` and `decodeURI`.

The home directory

A path name can start with a tilde (`~`) to indicate the user's home directory. It corresponds to the platform's `HOME` environment variable.

UNIX and Mac OS assign the `HOME` environment variable according to the user login. On Mac OS, the default home directory is `/Users/`*`username`*. In UNIX, it is typically `/home/`*`username`* or `/users/`*`username`*. ExtendScript assigns the home directory value directly from the platform value.

On Windows, the HOME environment variable is optional. If it is assigned, its value must be a Windows path name or a path name referring to a remote server (such as \\myhost\mydir). If the HOME environment variable is undefined, the ExtendScript default is the user's home directory, usually the C:\Documents and Settings*username* folder.

Note: A script can access many of the folders that are specified with platform-specific variables through static, globally-available Folder class properties; for instance, appData contains the folder that stores application data for all users.

Volume and drive names

A volume or drive name can be the first part of an absolute path in URI notation. The values are interpreted according to the platform.

Mac OS volumes

When Mac OS X starts, the startup volume is the root directory of the file system. All other volumes, including remote volumes, are part of the /Volumes directory. The File and Folder objects use these rules to interpret the first element of a path name:

- If the name is the name of the startup volume, discard it.
- If the name is a volume name, prepend /Volumes.
- Otherwise, leave the path as is.

Mac OS 9 is not supported as an operating system, but the use of the colon as a path separator is still supported and corresponds to URI and to Mac OS X paths as shown in the following table. These examples assume that the startup volume is MacOSX, and that there is a mounted volume Remote.

URI path name	Mac OS 9 path name	Mac OS X path name
/MacOSX/dir/file	MacOSX:dir:file	/dir/file
/Remote/dir/file	Remote:dir:file	/Volumes/Remote/dir/file
/root/dir/file	Root:dir:file	/root/dir/file
~/dir/file		/Users/jdoe/dir/file

Windows drives

On Windows, volume names correspond to drive letters. The URI path /c/temp/file normally translates to the Windows path C:\temp\file.

If a drive exists with a name matching the first part of the path, that part is always interpreted as that drive. It is possible for there to be a folder in the root that has the same name as the drive; imagine, for example, a folder C:\C on Windows. A path starting with /c always addresses the drive C:, so in this case, to access the folder by name, you must use both the drive name and the folder name, for example /c/c for C:\C.

If the current drive contains a root folder with the same name as another drive letter, that name is considered to be a folder. That is, if there is a folder D:\C, and if the current drive is D:, the URI path /c/temp/file translates to the Windows path D:\c\temp\file. In this case, to access drive C, you would have to use the Windows path name conventions.

To access a remote volume, use a uniform naming convention (UNC) path name of the form //*servername*/*sharename*. These path names are portable, because both Max OS X and UNIX ignore multiple slash characters. Note that on Windows, UNC names do *not* work for local volumes.

These examples assume that the current drive is `D:`

URI path name	Windows path name
`/c/dir/file`	`c:\dir\file`
`/remote/dir/file`	`D:\remote\dir\file`
`/root/dir/file`	`D:\root\dir\file`
`~/dir/file`	`C:\Documents and Settings\jdoe\dir\file`

Aliases

When you access an alias, the operation is transparently forwarded to the real file. The only operations that affect the alias are calls to `rename` and `remove`, and setting properties `readonly` and `hidden`. When a `File` object represents an alias, the `alias` property of the object returns `true`, and the `resolve` method returns the `File` or `Folder` object for the target of the alias.

On Windows, all file system aliases (called *shortcuts*) are actual files whose names end with the extension `.lnk`. Never use this extension directly; the `File` and `Folder` objects work without it.

For example, suppose there is a shortcut to the file `/folder1/some.txt` in the folder `/folder2`. The full Windows file name of the shortcut file is `\folder2\some.txt.lnk`.

To access the shortcut from a `File` object, specify the path `/folder2/some.txt`. Calling that `File` object's `open` method opens the linked file (in `/folder1`). Calling the `File` object's `rename` method renames the shortcut file itself (leaving the `.lnk` extension intact).

However, Windows permits a file and its shortcut to reside in the same folder. In this case, the `File` object always accesses the original file. You cannot create a `File` object to access the shortcut when it is in the same folder as its linked file.

A script can create a file alias by creating a `File` object for a file that does not yet exist on disk, and using its `createAlias` method to specify the target of the alias.

Portability issues

If your application will run on multiple platforms, use relative path names, or try to originate path names from the home directory. If that is not possible, work with Mac OS X and UNIX aliases, and store your files on a machine that is remote to your Windows machine so that you can use UNC names.

As an example, suppose you use the UNIX machine `myServer` for data storage. If you set up an alias share in the root directory of `myServer`, and if you set up a Windows-accessible share at `share` pointing to the same data location, the path name `//myServer/share/file` would work for all three platforms.

Unicode I/O

When doing file I/O, Adobe applications convert 8-bit character encoding to Unicode. By default, this conversion process assumes that the system encoding is used (code page 1252 on Windows or Mac Roman on Mac OS). The `encoding` property of a `File` object returns the current encoding. You can set the `encoding` property to the name of the desired encoding. The `File` object looks for the corresponding encoder in the operating system to use for subsequent I/O. The name is one of the standard Internet names that are used to describe the encoding of HTML files, such as `ASCII`, `X-SJIS`, or `ISO-8859-1`. For a complete list, see File and Folder Supported Encoding Names.

A special encoder, `BINARY`, is provided for binary I/O. This encoder simply extends every 8-bit character it finds to a Unicode character between 0 and 255. When using this encoder to write binary files, the encoder writes the lower 8 bits of the Unicode character. For example, to write the Unicode character `1000`, which is 0x3E8, the encoder actually writes the character 232 (0xE8).

The data of some of the common file formats (UCS-2, UCS-4, UTF-8, UTF-16) starts with a special byte order mark (BOM) character (`\uFEFF`). The `File.open` method reads a few bytes of a file looking for this character. If it is found, the corresponding encoding is set automatically and the character is skipped. If there is no BOM character at the beginning of the file, `open()` reads the first 2 KB of the file and checks whether the data might be valid UTF-8 encoded data, and if so, sets the encoding to UTF-8.

To write 16-bit Unicode files in UTF-16 format, use the encoding UCS-2. This encoding uses whatever byte-order format the host platform supports.

When using UTF-8 encoding or 16-bit Unicode, always write the BOM character "`\uFEFF`" as the first character of the file.

File error handling

Each object has an `error` property. If accessing a property or calling a method causes an error, this property contains a message describing the type of the error. On success, the property contains the empty string. You can set the property, but setting it only causes the error message to be cleared. If a file is open, assigning an arbitrary value to the property also resets its error flag.

For a complete list of supported error messages, see 'File and Folder Error Messages' on page 38.

File and Folder Error Messages

The following messages can be returned in the `error` property.

`File or folder does not exist`	The file or folder does not exist, but the parent folder exists.
`File or folder already exists`	The file or folder already exists.
`I/O device is not open`	An I/O operation was attempted on a file that was closed.
`Read past EOF`	Attempt to read beyond the end of a file.
`Conversion error`	The content of the file cannot be converted to Unicode.
`Partial multibyte character found`	The character encoding of the file data has errors.
`Permission denied`	The OS did not allow the attempted operation.
`Cannot change directory`	Cannot change the current folder.
`Cannot create`	Cannot create a folder.
`Cannot rename`	Cannot rename a file or folder.
`Cannot delete`	Cannot delete a file or folder.
`I/O error`	Unspecified I/O error.
`Cannot set size`	Setting the file size failed.
`Cannot open`	Opening of a file failed.
`Cannot close`	Closing a file failed.
`Read error`	Reading from a file failed.
`Write error`	Writing to a file failed.
`Cannot seek`	Seek failure.
`Cannot execute`	Unable to execute the specified file.

File and Folder Supported Encoding Names

The following list of names is a basic set of encoding names supported by the `File` object. Some of the character encoders are built in, while the operating system is queried for most of the other encoders. Depending on the language packs installed, some of the encodings may not be available. Names that refer to the same encoding are listed in one line. Underlines are replaced with dashes before matching an encoding name.

The `File` object processes an extended Unicode character with a value greater that 65535 as a Unicode surrogate pair (two characters in the range between 0xD700-0xDFFF).

Built-in encodings are:

```
US-ASCII, ASCII,ISO646-US,I SO-646.IRV:1991, ISO-IR-6,
ANSI-X3.4-1968,CP367,IBM367,US,ISO646.1991-IRV
UCS-2,UCS2, ISO-10646-UCS-2
UCS2LE,UCS-2LE,ISO-10646-UCS-2LE
UCS2BE,UCS-2BE,ISO-10646-UCS-2BE
UCS-4,UCS4, ISO-10646-UCS-4
UCS4LE,UCS-4LE,ISO-10646-UCS-4LE
UCS4BE,UCS-4BE,ISO-10646-UCS-4BE
UTF-8,UTF8,UNICODE-1-1-UTF-8,UNICODE-2-0-UTF-8,X-UNICODE-2-0-UTF-8
UTF16,UTF-16,ISO-10646-UTF-16
UTF16LE,UTF-16LE,ISO-10646-UTF-16LE
UTF16BE,UTF-16BE,ISO-10646-UTF-16BE
CP1252,WINDOWS-1252,MS-ANSI
ISO-8859-1,ISO-8859-1,ISO-8859-1:1987,ISO-IR-100,LATIN1
MACINTOSH,X-MAC-ROMAN
BINARY
```

The ASCII encoder raises errors for characters greater than 127, and the BINARY encoder simply converts between bytes and Unicode characters by using the lower 8 bits. The latter encoder is convenient for reading and writing binary data.

Additional encodings

In Windows, all encodings use code pages, which are assigned numeric values. The usual Western character set that Windows uses, for example, is the code page 1252. You can select Windows code pages by prepending the number of the code page with "CP" or "WINDOWS": for example, "CP1252" for the code page 1252. The `File` object has many other built-in encoding names that match predefined code page numbers. If a code page is not present, the encoding cannot be selected.

In Mac OS, you can select encoders by name rather than by code page number. The `File` object queries Mac OS directly for an encoder. As far as Mac OS character sets are identical with Windows code pages, Mac OS also knows the Windows code page numbers.

In UNIX, the number of available encoders depends on the installation of the `iconv` library.

Common encoding names

The following encoding names are implemented both in Windows and in Mac OS:

```
UTF-7,UTF7,UNICODE-1-1-UTF-7,X-UNICODE-2-0-UTF-7
ISO-8859-2,ISO-8859-2,ISO-8859-2:1987,ISO-IR-101,LATIN2
ISO-8859-3,ISO-8859-3,ISO-8859-3:1988,ISO-IR-109,LATIN3
ISO-8859-4,ISO-8859-4,ISO-8859-4:1988,ISO-IR-110,LATIN4,BALTIC
ISO-8859-5,ISO-8859-5,ISO-8859-5:1988,ISO-IR-144,CYRILLIC
ISO-8859-6,ISO-8859-6,ISO-8859-6:1987,ISO-IR-127,ECMA-114,ASMO-708,ARABIC
ISO-8859-7,ISO-8859-7,ISO-8859-7:1987,ISO-IR-126,ECMA-118,ELOT-928,GREEK8,GREEK
ISO-8859-8,ISO-8859-8,ISO-8859-8:1988,ISO-IR-138,HEBREW
```

```
ISO-8859-9,ISO-8859-9,ISO-8859-9:1989,ISO-IR-148,LATIN5,TURKISH
ISO-8859-10,ISO-8859-10,ISO-8859-10:1992,ISO-IR-157,LATIN6
ISO-8859-13,ISO-8859-13,ISO-IR-179,LATIN7
ISO-8859-14,ISO-8859-14,ISO-8859-14,ISO-8859-14:1998,ISO-IR-199,LATIN8
ISO-8859-15,ISO-8859-15,ISO-8859-15:1998,ISO-IR-203
ISO-8859-16,ISO-885,ISO-885,MS-EE
CP850,WINDOWS-850,IBM850
CP866,WINDOWS-866,IBM866
CP932,WINDOWS-932,SJIS,SHIFT-JIS,X-SJIS,X-MS-SJIS,MS-SJIS,MS-KANJI
CP936,WINDOWS-936,GBK,WINDOWS-936,GB2312,GB-2312-80,ISO-IR-58,CHINESE
CP949,WINDOWS-949,UHC,KSC-5601,KS-C-5601-1987,KS-C-5601-1989,ISO-IR-149,KOREAN
CP950,WINDOWS-950,BIG5,BIG-5,BIG-FIVE,BIGFIVE,CN-BIG5,X-X-BIG5
CP1251,WINDOWS-1251,MS-CYRL
CP1252,WINDOWS-1252,MS-ANSI
CP1253,WINDOWS-1253,MS-GREEK
CP1254,WINDOWS-1254,MS-TURK
CP1255,WINDOWS-1255,MS-HEBR
CP1256,WINDOWS-1256,MS-ARAB
CP1257,WINDOWS-1257,WINBALTRIM
CP1258,WINDOWS-1258
CP1361,WINDOWS-1361,JOHAB
EUC-JP,EUCJP,X-EUC-JP
EUC-KR,EUCKR,X-EUC-KR
HZ,HZ-GB-2312
X-MAC-JAPANESE
X-MAC-GREEK
X-MAC-CYRILLIC
X-MAC-LATIN
X-MAC-ICELANDIC
X-MAC-TURKISH
```

Additional Windows encoding names

```
CP437,IBM850,WINDOWS-437
CP709,WINDOWS-709,ASMO-449,BCONV4
EBCDIC
KOI-8R
KOI-8U
ISO-2022-JP
ISO-2022-KR
```

Additional Mac OS encoding names

These names are alias names for encodings that Mac OS might know.

```
TIS-620,TIS620,TIS620-0,TIS620.2529-1,TIS620.2533-0,TIS620.2533-1,ISO-IR-166
CP874,WINDOWS-874
JP,JIS-C6220-1969-RO,ISO646-JP,ISO-IR-14
JIS-X0201,JISX0201-1976,X0201
JIS-X0208,JIS-X0208-1983,JIS-X0208-1990,JIS0208,X0208,ISO-IR-87
JIS-X0212,JIS-X0212.1990-0,JIS-X0212-1990,X0212,ISO-IR-159
CN,GB-1988-80,ISO646-CN,ISO-IR-57
ISO-IR-16,CN-GB-ISOIR165
KSC-5601,KS-C-5601-1987,KS-C-5601-1989,ISO-IR-149
EUC-CN,EUCCN,GB2312,CN-GB
EUC-TW,EUCTW,X-EUC-TW
```

UNIX encodings

In UNIX, the `File` object looks for the presence of the `iconv` library, and uses whatever encoding it finds there. If you need a special encoding in UNIX, make sure that there is an `iconv` encoding module installed that converts between UTF-16 (the internal format that the `File` object uses) and the desired encoding.

File and Folder Object Reference

File Object

Represents a file in the local file system in a platform-independent manner. All properties and methods resolve file system aliases automatically and act on the original file unless otherwise noted.

File object constructors

To create a `File` object, use the `File` function or the `new` operator. The constructor accepts full or partial path names, and returns the new object. The CRLF sequence for the file is preset to the system default, and the encoding is preset to the default system encoding.

```
File ([path]); //can return a Folder object
new File ([path]); //always returns a File object
```

path	Optional. The absolute or relative path to the file associated with this object, specified in platform-specific or URI format; see 'Specifying paths' on page 33. The value stored in the object is the absolute path. The path need not refer to an existing file. If not supplied, a temporary name is generated. If the path refers to an existing folder: • The `File` function returns a `Folder` object instead of a `File` object. • The `new` operator returns a `File` object for a nonexisting file with the same name.

File class properties

This property is available as a static property of the `File` class. It is not necessary to create an instance to access it.

fs	String	The name of the file system. Read only. One of `Windows`, `Macintosh`, or `Unix`.

File class functions

These functions are available as static methods of the `File` class. It is not necessary to create an instance to call them.

decode()

```
File.decode (uri)
```

uri String. The encoded string to decode. All special characters must be encoded in UTF-8 and stored as escaped characters starting with the percent sign followed by two hexadecimal digits. For example, the string "`my%20file`" is decoded as "`my file`".

Special characters are those with a numeric value greater than 127, except the following:

`/ - _ . ! ~ * ' ( )`

Decodes the specified string as required by RFC 2396.

Returns the decoded string.

encode()

```
File.encode (name)
```

name String. The string to encode.

Encodes the specified string as required by RFC 2396. All special characters are encoded in UTF-8 and stored as escaped characters starting with the percent sign followed by two hexadecimal digits. For example, the string "`my file`" is encoded as "`my%20file`".

Special characters are those with a numeric value greater than 127, except the following:

`/ - _ . ! ~ * ' ( )`

Returns the encoded string.

isEncodingAvailable()

```
File.isEncodingAvailable (name)
```

name String. The encoding name. Typical values are "ASCII", "binary", or "UTF-8". See 'File and Folder Supported Encoding Names' on page 39.

Checks whether a given encoding is available.

Returns `true` if your system supports the specified encoding, `false` otherwise.

openDialog()

```
File.openDialog ([prompt, filter, multiSelect])
```

prompt Optional. A string containing the prompt text, if the e9.8r al(lo)1.2we promp.[

File object properties

These properties are available for `File` objects.

encoding	String	Gets or sets the encoding for subsequent read/write operations. One of the encoding constants listed in 'File and Folder Supported Encoding Names' on page 39. If the value is not recognized, uses the system default encoding. A special encoder, `BINARY`, is used to read binary files. It stores each byte of the file as one Unicode character regardless of any encoding. When writing, the lower byte of each Unicode character is treated as a single byte to write.
eof	Boolean	When `true`, a read attempt caused the current position to be at the end of the file, or the file is not open. Read only.
error	String	A message describing the last file system error; see 'File and Folder Error Messages' on page 38. Typically set by the file system, but a script can set it. Setting this value clears any error message and resets the error bit for opened files. Contains the empty string if there is no error.
exists	Boolean	When `true`, this object refers to a file or file-system alias that actually exists in the file system. Read only.
fsName	String	The platform-specific full path name for the referenced file. Read only.
fullName	String	The full path name for the referenced file in URI notation. Read only.
hidden	Boolean	When `true`, the file is not shown in the platform-specific file browser. Read/write. If the object references a file-system alias or shortcut, the flag is altered on the alias, not on the original file.
length	Number	The size of the file in bytes. Can be set only for a file that is not open, in which case it truncates or pads

relativeURI	String	The path name for the referenced file in URI notation, relative to the current folder. Read only.
type	String	The file type as a four-character string. • In Mac OS , the Mac OS file type. • In Windows, "appl" for .EXE files, "shlb" for .DLL files and "TEXT" for any other file. If the file does not exist, the value is "????". Read only.

File object functions

These functions are available for `File` objects.

changePath()

fileObj.changePath (*path*)

path A string containing the new path, absolute or relative to the current folder.

Changes the path specification of the referenced file.

Returns `true` on success.

close()

fileObj.close ()

Closes this open file.

Returns `true` on success, `false` if there are I/O errors.

copy()

fileObj.copy (*target*)

target A string with the URI path to the target location, or a `File` object that references the target location.

Copies this object's referenced file to the specified target location. Resolves any aliases to find the source file. If a file exists at the target location, it is overwritten.

Returns `true` if the copy was successful, `false` otherwise.

createAlias()

fileObj.createAlias (*path*])

path A string containing the path of the target file.

Makes this file a file-system alias or shortcut to the specified file. The referenced file for this object must not yet exist on disk.

Returns `true` if the operation was successful, `false` otherwise.

execute()

fileObj.execute ()

Opens this file using the appropriate application, as if it had been double-clicked in a file browser. You can use this method to run scripts, launch applications, and so on.

Returns `true` immediately if the application launch was successful.

getRelativeURI()

```
fileObj.getRelativeURI ([basePath])
```

basePath Optional. A string containing the base path for the relative URI. Default is the current folder.

Retrieves the URI for this file, relative to the specified base path, in URI notation. If no base path is supplied, the URI is relative to the path of the current folder.

Returns a string containing the relative URI.

open()

```
fileObj.open (mode[,type][,creator])
```

mode A string indicating the read/write mode. One of:

- `r`: (read) Opens for reading. If the file does not exist or cannot be found, the call fails.
- `w`: (write) Opens a file for writing. If the file exists, its contents are destroyed. If the file does not exist, creates a new, empty file.
- `e`: (edit) Opens an existing file for reading and writing.

type Optional. In Mac OS, the type of a newly created file, a 4-character string. Ignored in Windows and UNIX.

creator Optional. In Mac OS, the creator of a newly created file, a 4-character string. Ignored in Windows and UNIX.

Opens the referenced file for subsequent read/write operations. The method resolves any aliases to find the file.

Returns `true` if the file has been opened successfully, `false` otherwise.

The method attempts to detect the encoding of the open file. It reads a few bytes at the current location and tries to detect the Byte Order Mark character `0xFFFE`. If found, the current position is advanced behind the detected character and the encoding property is set to one of the strings `UCS-2BE`, `UCS-2LE`, `UCS4-BE`, `UCS-4LE`, or `UTF-8`. If the marker character is not found, it checks for zero bytes at the current location and makes an assumption about one of the above formats (except UTF-8). If everything fails, the `encoding` property is set to the system encoding.

Note: Be careful about opening a file more than once. The operating system usually permits you to do so, but if you start writing to the file using two different `File` objects, you can destroy your data.

openDlg()

```
fileObj.OpenDlg ([prompt][,filter][,multiSelect])
```

prompt Optional. A string containing the prompt text, if the dialog allows a prompt.

filter Optional. A filter that limits the types of files displayed in the dialog.

- In Windows, a filter expression, such as `"*.jsx"`
- In Mac OS, a filter function that takes a File instance and returns `true` if the file should be included in the display, `false` if it should not.

multiSelect Optional. Boolean. When `true`, the user can select multiple files and the return value is an array. Default is `false`.

Opens the built-in platform-specific file-browsing dialog, in which the user can select an existing file or files, and creates new `File` objects to represent the selected files. Differs from the class method `openDialog()` in that it presets the current folder to this `File` object's parent folder and the current file to this object's associated file.

If the user clicks **OK**, returns a `File` or `Folder` object for the selected file or folder, or an array of objects. If the user cancels, returns `null`.

read()

```
fileObj.read ([chars])
```

chars Optional. An integer specifying the number of characters to read. By default, reads from the current position to the end of the file. If the file is encoded, multiple bytes might be read to create single Unicode characters.

Reads the contents of the file starting at the current position.

Returns a string that contains up to the specified number of characters.

readch()

```
fileObj.readch ()
```

Reads a single text character from the file at the current position. Line feeds are recognized as `CR`, `LF`, `CRLF`, or `LFCR` pairs. If the file is encoded, multiple bytes might be read to create single Unicode characters.

Returns a string that contains the character.

readln()

```
fileObj.readln ()
```

Reads a single line of text from the file at the current position, and returns it in a string. Line feeds are recognized as `CR`, `LF`, `CRLF`, or `LFCR` pairs. If the file is encoded, multiple bytes might be read to create single Unicode characters.

Returns a string that contains the text.

remove()

```
fileObj.remove ()
```

Deletes the file associated with this object from disk, immediately, without moving it to the system trash. Does not resolve aliases; instead, deletes the referenced alias or shortcut file itself.

Note: Cannot be undone. It is recommended that you prompt the user for permission before deleting.

Returns `true` if the file is deleted successfully.

rename()

```
fileObj.rename (newName)
```

newName The new file name, with no path.

Renames the associated file. Does not resolve aliases, but renames the referenced alias or shortcut file itself.

Returns `true` on success.

resolve()

```
fileObj.resolve ()
```

If this object references an alias or shortcut, this method resolves that alias and returns a new `File` object that references the file-system element to which the alias resolves.

Returns the new `File` object, or `null` if this object does not reference an alias, or if the alias cannot be resolved.

saveDlg()

```
fileObj.saveDlg ([prompt][,preset])
```

prompt Optional. A string containing the prompt text, if the dialog allows a prompt.

preset Optional. A filter that limits the types of files displayed in the dialog.

- In Windows, a filter expression, such as `"*.jsx"`
- In Mac OS, a filter function that takes a File instance and returns `true` if the file should be included in the display, `false` if it should not.

Opens the built-in platform-specific file-browsing dialog, in which the user can select an existing file location to which to save information, and creates a new `File` object to represent the selected file.

Differs from the class method `saveDialog()` in that it presets the current folder to this `File` object's parent folder and the file to this object's associated file.

If the user clicks **OK**, returns a `File` object for the selected file. If the user cancels, returns `null`.

seek()

```
fileObj.seek (pos[, mode])
```

pos The new current position in the file as an offset in bytes from the start, current position, or end, depending on the `mode`.

mode Optional. The seek mode, one of:

0: Seek to absolute position, where `pos=0` is the first byte of the file. This is the default.
1: Seek relative to the current position.
2. Seek backward from the end of the file.

Seeks to the specified position in the file. The new position cannot be less than 0 or greater than the current file size.

Returns `true` if the position was changed.

tell()

```
fileObj.tell ()
```

Retrieves the current position as a byte offset from the start of the file.

Returns a number, the position index.

write()

```
fileObj.write (text[, text...]...)
```

text One or more strings to write, which are concatenated to form a single string.

Writes the specified text to the file at the current position. For encoded files, writing a single Unicode character may write multiple bytes.

Note: Be careful not to write to a file that is open in another application or object, as this can overwrite existing data.

Returns `true` on success.

writeln()

```
fileObj.writeln (text[, text...]...)
```

text One or more strings to write, which are concatenated to form a single string.

Writes the specified text to the file at the current position, and appends a Line Feed sequence in the style specified by the `linefeed` property.For encoded files, writing a single Unicode character may write multiple bytes.

Note: Be careful not to write to a file that is open in another application or object, as this can overwrite existing data.

Returns `true` on success.

Folder Object

Represents a file-system folder or directory in a platform-independent manner. All properties and methods resolve file system aliases automatically and act on the original file unless otherwise noted.

Folder object constructors

To create a `Folder` object, use the `Folder` function or the `new` operator. The constructor accepts full or partial path names, and returns the new object.

```
Folder ([path]); //can return a File object
new Folder ([path]); //always returns a Folder object
```

path	Optional. The absolute or relative path to the folder associated with this object, specified in URI format; see 'Specifying paths' on page 33. The value stored in the object is the absolute path. The path need not refer to an existing folder. If not supplied, a temporary name is generated. If the path refers to an existing file: • The `Folder` function returns a `File` object instead of a `Folder` object. • The `new` operator returns a `Folder` object for a nonexisting folder with the same name.

Folder class properties

These properties are available as static properties of the `Folder` class. It is not necessary to create an instance to access them.

appData	Folder	A `Folder` object for the folder that contains application data for all users. Read only. • In Windows, the value of `%APPDATA%` (by default, `C:\Documents and Settings\All Users\Application Data`) • In Mac OS, `/Library/Application Support`
appPackage	String	In Mac OS, the `Folder` object for the folder that contains the bundle of the running application. Read only.
commonFiles	Folder	A `Folder` object for the folder that contains files common to all programs. Read only. • In Windows, the value of `%CommonProgramFiles%` (by default, `C:\Program Files\Common Files`) • In Mac OS,`/Library/Application Support`
current	Folder	A `Folder` object for the current folder. Assign either a `Folder` object or a string containing the new path name to set the current folder.
desktop	Folder	A `Folder` object for the folder that contains the user's desktop. Read only. • In Windows, `C:\Documents and Settings\`*username*`\Desktop` • In Mac OS, `~/Desktop`

fs	String	The name of the file system. Read only. One of `Windows`, `Macintosh`, or `Unix`.
myDocuments	Folder	A `Folder` object for the user's default document folder. Read only. • In Windows, `C:\Documents and Settings\`*username*`\My Documents` • In Mac OS, `~/Documents`
startup	Folder	A `Folder` object for the folder containing the executable image of the running application. Read only.
system	Folder	A `Folder` object for the folder containing the operating system files. Read only. • In Windows, the value of `%windir%` (by default, `C:\Windows`) • In Mac OS, `/System`
temp	Folder	A `Folder` object for the default folder for temporary files. Read only.
trash	Folder	A `Folder` object for the folder containing deleted items. Read only.
userData	Folder	A `Folder` object for the folder that contains application data for the current user. Read only. • In Windows, the value of `%USERDATA%` (by default, `C:\Documents and Settings\`*username*`\Application Data`) • In Mac OS, `~/Library/Application Support`

Folder class functions

These functions are available as a static methods of the `Folder` class. It is not necessary to create an instance in order to call them.

decode()

```
Folder.decode (uri)
```

uri String. The encoded string to decode. All special characters must be encoded in UTF-8 and stored as escaped characters starting with the percent sign followed by two hexadecimal digits. For example, the string "`my%20file`" is decoded as "`my file`".

Special characters are those with a numeric value greater than 127, except the following:

```
/ - _ . ! ~ * ' ( )
```

Decodes the specified string as required by RFC 2396.

Returns the decoded string.

encode()

```
Folder.encode (name)
```

name String. The string to encode.

Encodes the specified string as required by RFC 2396 . All special characters are encoded in UTF-8 and stored as escaped characters starting with the percent sign followed by two hexadecimal digits. For example, the string "`my file`" is encoded as "`my%20file`".

Special characters are those with a numeric value greater than 127, except the following:

`/ - _ . ! ~ * ' ( )`

Returns the encoded string.

isEncodingAvailable()

```
Folder.isEncodingAvailable (name)
```

name String. The encoding name. Typical values are "ASCII", "binary", or "UTF-8". See 'File and Folder Supported Encoding Names' on page 39.

Checks whether a given encoding is available.

Returns `true` if your system supports the specified encoding, `false` otherwise.

selectDialog()

```
Folder.selectDialog ([prompt])
```

prompt Optional. A string containing the prompt text, if the dialog allows a prompt.

Opens the built-in platform-specific file-browsing dialog, and creates a new `File` or `Folder` object for the selected file or folder. Differs from the object method `selectDlg()` in that it does not preselect a folder.

If the user clicks **OK**, returns a `File` or `Folder` object for the selected file or folder. If the user cancels, returns `null`.

Folder object properties

These properties are available for `Folder` objects.

absoluteURI	String	The full path name for the referenced folder in URI notation. Read only.
alias	Boolean	When `true`, the object refers to a file system alias or shortcut. Read only.
created	Date	The creation date of the referenced folder, or `null` if the object does not refer to a folder on disk. Read only.
displayName	String	The localized name of the referenced folder, without the path. Read only.
error	String	A message describing the most recent file system error; see 'File and Folder Error Messages' on page 38. Typically set by the file system, but a script can set it. Setting this value clears any error message and resets the error bit for opened files. Contains the empty string if there is no error.

exists	Boolean	When `true`, this object refers to a folder that currently exists in the file system. Read only.
fsName	String	The platform-specific name of the referenced folder as a full path name. Read only.
fullName	String	The full path name for the referenced folder in URI notation. Read only.
localizedName	String	A localized version of the folder name portion of the absolute URI for the referenced file, without the path specification. Read only.
modified	Date	The date of the referenced folder's last modification, or `null` if the object does not refer to a folder on disk. Read only.
name	String	The folder name portion of the absolute URI for the referenced file, without the path specification. Read only.
parent	Folder	The `Folder` object for the folder that contains this folder, or `null` if this object refers to the root folder of a volume. Read only.
path	String	The path portion of the absolute URI for the referenced folder, without the folder name. Read only.
relativeURI	String	The path name for the referenced folder in URI notation, relative to the current folder. Read only.

Folder object functions

These functions are available for `Folder` objects.

changePath()

```
folderObj.changePath (path)
```

path A string containing the new path, absolute or relative to the current parent folder.

Changes the path specification of the referenced folder.

Returns `true` on success.

create()

```
folderObj.create ()
```

Creates a folder at the location given by this object's path property.

Returns `true` if the folder was created successfully.

execute()

```
folderObj.execute ()
```

Opens this folder in the platform-specific file browser (as if it had been double-clicked in the file browser).

Returns `true` immediately if the folder was opened successfully.

getFiles()

```
folderObj.getFiles ([mask])
```

mask Optional. A search mask for file names. A string that can contain question mark (?) and asterisk (*) wild cards. Default is "*", which matches all file names.

Can also be the name of a function that takes a File or Folder object as its argument. It is called for each file or folder found in the search; if it returns true, the object is added to the return array.

Note: In Windows, all aliases end with the extension .lnk; ExtendScript strips this from the file name when found, in order to preserve compatibility with other operating systems. You can search for all aliases by supplying the search mask "*.lnk", but note that such code is not portable.

Retrieves the contents of this folder, filtered by the supplied mask.

Returns an array of File and Folder objects, or null if this object's referenced folder does not exist.

getRelativeURI()

```
folderObj.getRelativeURI ([basePath])
```

basePath Optional. A string containing the base path for the relative URI. Default is the current folder.

Retrieves the path for this folder relative to the specified base path or the current folder, in URI notation.

Returns a string containing the relative URI.

remove()

```
folderObj.remove ()
```

Deletes the empty folder associated with this object from disk, immediately, without moving it to the system trash.

- Folders must be empty before they can be deleted.
- Does not resolve aliases; instead, deletes the referenced alias or shortcut file itself.

Note: Cannot be undone. It is recommended that you prompt the user for permission before deleting.

Returns true if the folder is deleted successfully.

rename()

```
folderObj.rename (newName)
```

newName The new folder name, with no path.

Renames the associated folder. Does not resolve aliases; instead, renames the referenced alias or shortcut file itself.

Returns true on success.

resolve()

```
folderObj.resolve ()
```

If this object references an alias or shortcut, this method resolves that alias

Returns a new Folder object that references the file-system element to which the alias resolves, or null if this object does not reference an alias, or if the alias cannot be resolved.

selectDlg()

```
folderObj.selectDlg (prompt)
```

prompt A string containing the prompt text, if the dialog allows a prompt.

Opens the built-in platform-specific file-browsing dialog, and creates a new `File` or `Folder` object for the selected file or folder. Differs from the class method `selectDialog()` in that it preselects this folder.

If the user clicks **OK**, returns a `File` or `Folder` object for the selected file or folder. If the user cancels, returns `null`.

4 User Interface Tools

Adobe provides the ScriptUI component, which works with the ExtendScript JavaScript interpreter to provide JavaScript scripts with the ability to create and interact with user interface elements. It provides an object model for windows and UI control elements within an Adobe application.

- The first part of this chapter describes how to use these classes and provides details of pathname syntax .
- 'ScriptUI Object Reference' on page 89 provides reference details of the objects, properties, methods, and creation parameters . You can also choose the ScriptUI Classes dictionary from the Help menu in the ExtendScript Toolkit to inspect the objects in the Object Model Viewer.

Note: Complete details of some new features in Adobe® Creative Suite 3 are not available at time of publication. Check for updated versions of this document at http://partners.adobe.com.

➤ Code examples for ScriptUI

The sample code distributed with the Adobe Bridge SDK includes code examples that specifically demonstrate different ways of building and populating a ScriptUI dialog.

Building ScriptUI dialogs	
`SnpCreateDialog.jsx`	Creates a very simple, modeless dialog (a palette) with OK and Cancel button behavior.
`SnpCreateUIAddMethod.jsx`	Shows how to add controls to a dialog using the `add` method.
`SnpCreateUIResourceSpec.jsx`	Shows how to define a resource string that creates the control hierarchy in a dialog.
`SnpCreateCheckBoxRadioButtons.jsx`	Shows how to create and initialize check box and radio button controls and define their behavior.
`SnpCreateProgressBar.jsx`	Shows how to create, initialize, and update a progress bar.
`SnpCreateSlider.jsx`	Shows how to create and handle input from a slider control.
`SnpCreateFlashControl.jsx`	Shows how to create a Flash® Player, and use it to load a play back a movie defined in an SWF file.
`FlashDemo.jsx`	Shows how to communicate between the Adobe application scripting environment and the ActionScript™ scripting environment of the Flash Player.
`AlertBoxBuilder1.jsx`	Shows a way to use resource specifications. Uses the `add()` method to build a dialog that collects values from the user, and creates a resource string from those values. Saves the string to a file, then uses it to build a new dialog. See 'Using resource strings' on page 70.
`AlertBoxBuilder2.jsx`	Shows another way to use a resource specification, building the same user-input dialog itself from a resource string. See 'Using resource strings' on page 70.

ScriptUI Programming Model

ScriptUI defines `Window` objects that represent platform-specific windows, and various control elements such as `Button` and `StaticText`, that represent user-interface controls. These objects share a common set of properties and methods that allow you to query the type, move the element around, set the title, caption or content, and so on. Many element types also have properties unique to that class of elements.

Creating a window

ScriptUI defines the following types of windows:

- Modal dialog box: Holds focus when shown, does not allow activity in other application windows until dismissed.
- Floating palette: Also called modeless dialog, allows activity in other application windows. (Adobe Photoshop® does not support script creation of palette windows.)
- Main window: Suitable for use as an application's main window. (Main windows are not normally created by script developers for Adobe applications. Photoshop does not support script creation of main windows.)

To create a new window, use the `Window` constructor function. The constructor takes the desired type of the window. The type is `"dialog"` for a modal dialog, or `"palette"` for a modeless dialog or floating palette. You can supply optional arguments to specify an initial window title and bounds.

The following example creates an empty dialog with the variable name `dlg`, which is used in subsequent examples:

```
// Create an empty dialog window near the upper left of the screen
var dlg = new Window('dialog', 'Alert Box Builder', [100,100,480,490]);
```

Newly created windows are initially hidden; the `show` method makes them visible and responsive to user interaction. For example:

```
dlg.show();
```

Container elements

All `Windows` are containers—that is, they contain other elements within their bounds. Within a `Window`, you can create other types of container elements: `Panels` and `Groups`. These can contain control elements, and can also contain other `Panel` and `Group` containers. However, a `Window` cannot be added to any container.

- A `Group` is the simplest container used to visually organize related controls. You would typically define a group and populate it with related elements, for instance an `edittext` box and its descriptive `statictext` label.
- A `Panel` is a frame object, also typically used to visually organize related controls. It has a text property to specify a title, and can have a border to visually separate the collection of elements from other elements of a dialog.

You might create a `Panel` and populate it with several `Groups`, each with their own elements. You can create nested containers, with different layout properties for different containers, in order to define a relatively complex layout without any explicit placement.

You can add elements to any container using the `add` method (see 'Adding elements to containers' on page 59). An element added to a container is considered a child of that container. Certain operations on a container apply to its children; for example, when you hide a container, its children are also hidden.

Window layout

When a script creates a `Window` and adds various UI elements to it, the locations and sizes of elements and spacing between elements is known as the *layout* of the window. Each UI element has properties which define its location and dimensions: `location`, `size`, and `bounds`. These properties are initially undefined, and a script that employs Automatic Layout should leave them undefined for the main window as well as its contained elements, allowing the automatic layout mechanism to set their values.

Your script can access these values, and (if not using auto-layout) set them as follows:

- The `location` of a window is defined by a `Point` object containing a pair of coordinates (`x` and `y`) for the top left corner (the *origin*), specified in the screen coordinate system. The `location` of an element within a window or other container is defined as the origin point specified in the container's coordinate system. That is, the `x` and `y` values are relative to the origin of the container.

  The following examples show equivalent ways of placing the content region of an existing window at screen coordinates [10, 50]:

  ```
  win.location = [10, 50];
  win.location = {x:10, y:50};
  win.location = "x:10, y:50";
  ```

- The `size` of an element's region is defined by a `Dimension` object containing a `width` and `height` in pixels.

  The following examples show equivalent ways of changing an existing window's width and height to 200 and 100:

  ```
  win.size = [200, 100];
  win.size = {width:200, height:100};
  win.size = "width:200, height:100";
  ```

  This example shows how to change a window's height to 100, leaving its location and width unchanged:

  ```
  win.size.height = 100;
  ```

- The `bounds` of an element are defined by a `Bounds` object containing both the origin point (`x`, `y`) and size (`width`, `height`) To define the size and location of windows and controls in one step, use the `bounds` property.

  The value of the `bounds` property can be a string with appropriate contents, an inline JavaScript `Bounds` object, or a four-element array. The following examples show equivalent ways of placing a 380 by 390 pixel window near the upper left corner of the screen:

  ```
  var dlg = new Window('dialog', 'Alert Box Builder', [100,100,480,490]);
  dlg.bounds = [100,100,480,490];
  dlg.bounds = {x:100, y:100, width:380, height:390};
  dlg.bounds = {left:100, top:100, right:480, bottom:490};
  dlg.bounds = "left:100, top:100, right:480, bottom:490";
  ```

The `window` dimensions define the size of the *content region* of the window, or that portion of the window that a script can directly control. The actual window size is typically larger, because the host platform's window system typically adds title bars and borders. The `bounds` property for a `Window` refers only to its content region. To determine the bounds of the frame surrounding the content region of a window, use the `Window.frameBounds` property.

Adding elements to containers

To add elements to a `window`, `panel`, or `group`, use the container's `add` method. This method accepts the `type` of the element to be created and some optional parameters, depending on the element type. It creates and returns an object of the specified type.

In additions to windows, ScriptUI defines the following user-interface elements and controls:

- Panels (frames) and groups, to collect and organize other control types
- Push buttons with text or icons, radio buttons, checkbox buttons
- Static text or images, edit text
- Progress bars, scrollbars, sliders
- Lists, which include list boxes and drop-down (also called popup) lists. Each item in a list is a control of type `item`, and the parent list's `items` property contains an array of child items. You can add list items with the parent list's `add` method.

You can specify the initial size and position of any new element relative to the working area of the parent container, in an optional `bounds` parameter. Different types of elements have different additional parameters. For elements which display text, for example, you can specify the initial text. See the ScriptUI Classes dictionary in the ExtendScript Toolkit's Object Model Viewer for details.

The order of optional parameters must be maintained. Use the value `undefined` for a parameter you do not wish to set. For example, if you want to use automatic layout to determine the bounds, but still set the title and text in a panel and button, the following creates `Panel` and `Button` elements with an initial `text` value, but no `bounds` value:

```
dlg.btnPnl = dlg.add('panel', undefined, 'Build it');
dlg.btnPnl.testBtn = dlg.btnPnl.add('button', undefined, 'Test');
```

Tip: This example creates a dynamic property, `btnPnl`, on the parent window object, which contains the returned reference to the child control object. This is not required, but provides a useful way to access your controls.

A new element is initially set to be visible, but it not shown unless its parent object is shown.

Creation properties

Some element types have attributes that can only be specified when the element is created. These are not normal properties of the element, in that they cannot be changed during the element's lifetime, and they are only needed once. For these element types, you can supply an optional *creation-properties* argument to the `add` method. This argument is an object with one or more properties that control aspects of the element's appearance, or special functions such as whether an edit text element is editable or Read only. See 'Control object constructors' on page 104 for details.

All UI elements have an optional creation property called `name`, which assigns a name for identifying that element. For example, the following creates a new `Button` element with the name 'ok':

```
dlg.btnPnl.buildBtn =
   dlg.btnPnl.add('button', undefined, 'Build', {name:'ok'});
```

Note: In Photoshop CS, panel coordinates were measured from outside the frame (including the title bar), but in Photoshop CS2, panel coordinates are measured from the inside the frame (the content area). This means that if you use the same values to set the vertical positions of child controls in a panel, the positions are slightly different in the two versions. When you add a panel to a window, you can

choose to set a creation property (`su1PanelCoordinates`), which causes that panel to automatically adjust the positions of its children; see the `add` method for `panel`. When automatic adjustment is enabled, you provide position values that were correct for Photoshop CS, and the result is the same in Photoshop CS2 or CS3. You can also set automatic adjustment for a window; in this case, it applies to all child panels of that window unless it is explicitly disabled in the child panel. See `Window` object constructor.

Accessing child elements

A reference to each element added to a container is appended to the container's `children` property. You can access the child elements through this array, using a 0-based index. For controls that are not containers, the `children` collection is empty.

In this example, the `msgPnl` panel was the first element created in `dlg`, so the script can access the panel object at index 0 of the parent's `children` property to set the text for the title:

```
var dlg = new Window('dialog', 'Alert Box Builder');
dlg.msgPnl = dlg.add('panel');
dlg.children[0].text = 'Messages';
```

If you use a creation property to assign a name to a newly created element, you can access that child by its name, either in the `children` array of its parent, or directly as a property of its parent. For example, the `Button` in a previous example was named "`ok`", so it can be referenced as follows:

```
dlg.btnPnl.children['ok'].text = "Build";
dlg.btnPnl.ok.text = "Build";
```

For list controls (type `list` and `dropdown`), you can access the child list-item objects through the `items` array.

Removing elements

To add elements to a `window`, `panel`, or `group`, use the container's `remove` method. This method accepts an object representing the element to be removed, or the name of the element, or the index of the element in the container's `children` collection (see 'Accessing child elements' on page 60).

The specified element is removed from view if it was currently visible, and it is no longer accessible from the container or window. The results of any further references by a script to the object representing the element are undefined.

To remove list items from a list, use the parent list control's `remove` method in the same way. It removes the item from the parent's `items` list, hides it from view, and deletes the item object.

Types of Controls

The following sections introduce the types of controls you can add to a `Window` or other container element (`panel` or `group`). For details of the properties and functions, and of how to create each type of element, see 'Control object constructors' on page 104.

Containers

These are types of `Control` objects which are contained in windows, and which contain and group other controls.

`Panel`	Typically used to visually organize related controls. • Set the `text` property to define a title which appears at the top of the `Panel`. • An optional `borderStyle` creation property controls the appearance of the border drawn around the panel. You can use `Panels` as separators: those with `width = 0` appear as vertical lines and those with `height = 0` appear as horizontal lines. `var dlg = new Window('dialog', 'Alert Box Builder',[100,100,480,245]);` **`dlg.msgPnl = dlg.add('panel', [25,15,355,130], 'Messages');`**
`Group`	Used to visually organize related controls. Unlike `Panels`, `Groups` have no title or visible border. You can use them to create hierarchies of controls, and for fine control over layout attributes of certain groups of controls within a larger panel. For examples, see 'Creating more complex arrangements' on page 82.

User interface controls

These are types of `Control` objects which are contained in windows, panels, and groups, and which provide specific kinds of display and user interaction.

`StaticText`	Typically used to display text strings that are not intended for direct manipulation by a user, such as informative messages or labels.
	This example creates a `Panel` and adds several `StaticText` elements:
	`var dlg = new Window('dialog', 'Alert Box Builder',[100,100,480,245]);`
	`dlg.msgPnl = dlg.add('panel', [25,15,355,130], 'Messages');`
	`dlg.msgPnl.titleSt = dlg.msgPnl.add('statictext', [15,15,105,35],`
	`'Alert box title:');`
	`dlg.msgPnl.msgSt = dlg.msgPnl.add('statictext', [15,65,105,85],`
	`'Alert message:');`
	`dlg.show();`

EditText	Allows users to enter text, which is returned to the script when the dialog is dismissed. Text in EditText elements can be selected, copied, and pasted.
	• Set the text property to assign the initial displayed text in the element, and read it to obtain the current text value, as entered or modified by the user.
	• Set the textselection property to replace the current selection with new text, or to insert text at the cursor (insertion point). Read this property to obtain the current selection, if any.
	This example adds some EditText elements, with initial values that a user can accept or replace:
	`var dlg = new Window('dialog', 'Alert Box Builder',[100,100,480,245]);`
	`dlg.msgPnl = dlg.add('panel', [25,15,355,130], 'Messages');`
	`dlg.msgPnl.titleSt = dlg.msgPnl.add('statictext', [15,15,105,35],`
	`'Alert box title:');`
	`dlg.msgPnl.titleEt = dlg.msgPnl.add('edittext', [115,15,315,35],`
	`'Sample Alert');`
	`dlg.msgPnl.msgSt = dlg.msgPnl.add('statictext', [15,65,105,85],`
	`'Alert message:');`
	`dlg.msgPnl.msgEt = dlg.msgPnl.add('edittext', [115,45,315,105],`
	`'<your message here>', {multiline:true});`
	`dlg.show();`
	Note the creation property on the second EditText field, where multiline:true indicates a field in which a long text string can be entered. The text wraps to appear as multiple lines.

Button	Typically used to initiate some action from a window when a user clicks the button; for example, accepting a dialog's current settings, canceling a dialog, bringing up a new dialog, and so on.
	• Set the text property to assign a label to identify a Button's function.
	• The onClick callback method provides behavior.
	`var dlg = new Window('dialog', 'Alert Box Builder',[100,100,480,245]);`
	`dlg.btnPnl = dlg.add('panel', [15,50,365,95], 'Build it');`
	`dlg.btnPnl.testBtn = dlg.btnPnl.add('button', [15,15,115,35], 'Test');`
	`dlg.btnPnl.buildBtn = dlg.btnPnl.add('button', [125,15,225,35],`
	`'Build', {name:'ok'});`
	`dlg.btnPnl.cancelBtn = dlg.btnPnl.add('button', [235,15,335,35],`
	`'Cancel', {name:'cancel'});`
	`dlg.show();`

IconButton	A button that displays an icon instead of text. Like a text button, typically initiates an action in response to a click. • The icon property identifies the icon image; see'Displaying icons' on page 65. • The onClick callback method provides behavior.
Image	Displays an iconic image. • The icon property identifies the icon image; see 'Displaying icons' on page 65.

Checkbox	Allows the user to set a boolean state.
	• Set the `text` property to assign an identifying text string that appears next to the clickable box.
	• The user can click to select or deselect the box, which shows a checkmark when selected. The `value=true` when it is selected (checked) and `false` when it is not.
	When you create a `Checkbox`, you can set its `value` property to specify its initial state and appearance.
	`// Add a checkbox to control the buttons that dismiss an alert box`
	`dlg.hasBtnsCb = dlg.add('checkbox', [125,145,255,165],`
	`'Should there be alert buttons?');`
	`dlg.hasBtnsCb.value = true;`

RadioButton	Allows the user to select one choice among several.
	• Set the `text` property to assign an identifying text string that appears next to the clickable button.
	• The `value=true` when the button is selected. The button shows the state in a platform-specific manner, with a filled or empty dot, for example.
	You group a related set of radio buttons by creating all the related elements one after another. When any button's value becomes `true`, the value of all other buttons in the group becomes `false`. When you create a group of radio buttons, you should set the state of one of them `true`:
	`var dlg = new Window('dialog', 'Alert Box Builder',[100,100,480,245]);`
	`dlg.alertBtnsPnl = dlg.add('panel', [45,50,335,95], 'Button alignment');`
	`dlg.alertBtnsPnl.alignLeftRb = dlg.alertBtnsPnl.add('radiobutton',`
	`[15,15,95,35], 'Left');`
	`dlg.alertBtnsPnl.alignCenterRb = dlg.alertBtnsPnl.add('radiobutton',`
	`[105,15,185,35], 'Center');`
	`dlg.alertBtnsPnl.alignRightRb = dlg.alertBtnsPnl.add('radiobutton',`
	`[195,15,275,35], 'Right');`
	`dlg.alertBtnsPnl.alignCenterRb.value = true``;`
	`dlg.show();`

Progressbar	Typically used to display the progress of a time-consuming operation. A colored bar covers a percentage of the area of the control, representing the percentage completion of the operation. The `value` property reflects and controls how much of the visible area is colored, relative to the maximum value (`maxvalue`). By default the range is 0 to 100, so the `value=50` when the operation is half done.
Slider	Typically used to select within a range of values. The slider is a horizontal bar with a draggable indicator, and you can click a point on the slider bar to jump the indicator to that location. The `value` property reflects and controls the position of the indicator, within the range determined by `minvalue` and `maxvalue`. By default the range is 0 to 100, so setting `value=50` moves the indicator to the middle of the bar.

Scrollbar	Like a slider, the scrollbar is a bar with a draggable indicator. It also has "stepper" buttons at each end, that you can click to jump the indicator by the amount in the `stepdelta` property. If you click a point on the bar outside the indicator, the indicator jumps by the amount in the `jumpdelta` property.
	You can create scrollbars with horizontal or vertical orientation; if `width` is greater than `height`, it is horizontal, otherwise it is vertical.
	Scrollbars are often created with an associated `EditText` field to display the current value of the scrollbar, and to allow setting the scrollbar's position to a specific value. This example creates a scrollbar with associated `StaticText` and `EditText` elements within a panel:
	`dlg.sizePnl = dlg.add('panel', [60,240,320,315], 'Dimensions');`
	`dlg.sizePnl.widthSt = dlg.sizePnl.add('statictext', [15,15,65,35], 'Width:');`
	`dlg.sizePnl.widthScrl = dlg.sizePnl.add('scrollbar', [75,15,195,35], 300, 300, 800);`
	`dlg.sizePnl.widthEt = dlg.sizePnl.add('edittext', [205,15,245,35]);`
	The last three arguments to the `add` method that creates the scrollbar define the values for the `value`, `minvalue` and `maxvalue` properties.

ListBox **DropDownList** **TreeView**	These controls display lists of items, which are represented by `ListItem` objects in the `items` property. You can access the items in this array using a 0-based index. • A `ListBox` control displays a list of choices. When you create the object, you specify whether it allows the user to select only one or multiple items. If a list contains more items than can be displayed in the available area, a scrollbar may appear that allows the user to scroll through all the list items. • A `DropDownList` control displays a single visible item. When you click the control, a list drops down and allows you to select one of the other items in the list. Drop-down lists can have nonselectable separator items for visually separating groups of related items, as in a menu. You can specify the choice items on creation of the list object, or afterward using the list object's `add()` method. You can remove items programmatically with the list object's `remove()` and `removeAll()` methods.
ListItem	Items added to or inserted into any type of list control are `ListItem` objects, with properties that can be manipulated from a script. `ListItem` elements can be of the following types: • `item`: the typical item in any type of list. It displays text or an icon, and can be selected. To display an icon, set the item object's `icon` property; see 'Displaying icons' on page 65. • `separator`: a separator is a nonselectable visual element in a drop-down list. Although it has a `text` property, the value is ignored, and the item is displayed as a horizontal line.
FlashPlayer	Runs a Flash movie within a ScriptUI window. Its control's methods allow you to load a movie from an SWF file and control the playback. See 'FlashPlayer control functions' on page 119. You can also use the control object to communicate with the Flash application, calling ActionScript methods, and making JavaScript methods defined in your Adobe application script available to the Flash ActionScript code. See 'Calling ActionScript functions from an Adobe script' on page 75.

Displaying icons

You can display icon images in Image or IconButton controls, or in place of strings as the selectable items in a Listbox or DropdownList control. In each case, the image is defined by setting the element's `icon` property, either to a named icon resource, a `File` object, or the pathname of a file containing the iconic image (see 'Specifying paths' on page 33).

The image data for an icon must be in Portable Network Graphics (PNG) format. See `http://www.libpng.org` for detailed information on the PNG format.

You can set or reset the `icon` property at any time to change the image displayed in the element.

The scripting environment can define icon *resources*, which are available to scripts by name. To specify an icon resource, set a control's `icon` property to the resource's JavaScript name, or refer to the resource by name when creating the control. For example, to create a button with an application-defined icon resource:

```
myWin.upBtn = myWin.add ("iconbutton", undefined, "SourceFolderIcon");
```

Photoshop CS3, for example, defines these icon resources:

```
Step1Icon
Step2Icon
Step3Icon
Step4Icon
SourceFolderIcon
DestinationFolderIcon
```

If a script does not explicitly set the `preferredSize` or `size` property of an element that displays a icon image, the value of `preferredSize` is determined by the dimensions of the iconic image. If the size values are explicitly set to dimensions smaller than those of the actual image graphic, the displayed image is clipped. If they are set to dimensions larger than those of the image graphic, the displayed image is centered in the larger space. An image is never scaled to fit the available space.

Prompts and alerts

Static functions on the `Window` class are globally available to display short messages in standard dialogs. The host application controls the appearance of these simple dialogs, so they are consistent with other alert and message boxes displayed by the application. You can often use these standard dialogs for simple interactions with your users, rather than designing special-purpose dialogs of your own.

Use the static functions `alert`, `confirm`, and `prompt` on the `Window` class to invoke these dialogs with your own messages. You do not need to create a `window` object to call these functions.

Modal dialogs

A modal dialog is initially invisible. Your script invokes it using the `show` method, which does not return until the dialog has been dismissed. The user can dismiss it by using a platform-specific window gesture, or by using one of the dialog controls that you supply, typically an **OK** or **Cancel** button. The `onClick` method of such a button must call the `close` or `hide` method to close the dialog. The `close` method allows you to pass a value to be returned by the `show` method.

For an example of how to define such buttons and their behavior, see 'Defining Behavior with Event Callbacks and Listeners' on page 71.

Creating and using modal dialogs

A dialog typically contains some controls that the user must interact with, to make selections or enter values that your script will use. In some cases, the result of the user action is stored in the object, and you can retrieve it after the dialog has been dismissed. For example, if the user changes the state of a `Checkbox` or `RadioButton`, the new state is found in the control's `value` property.

However, if you need to respond to a user action while the dialog is still active, you must assign the control a callback function for the interaction event, either `onClick` or `onChange`. The callback function is the value of the `onClick` or `onChange` property of the control.

For example, if you need to validate a value that the user enters in a `edittext` control, you can do so in an `onChange` callback handler function for that control. The callback can perform the validation, and perhaps display an alert to inform the user of errors.

Sometimes, a modal dialog presents choices to the user that must be correct before your script allows the dialog to be dismissed. If your script needs to validate the state of a dialog after the user clicks **OK**, you can define an `onClose` event handler for the dialog. This callback function is invoked whenever a window is closed. If the function returns `true`, the window is closed, but if it returns `false`, the close operation is cancelled and the window remains open.

Your `onClose` handler can examine the states of any controls in the dialog to determine their correctness, and can show `alert` messages or use other modal dialogs to alert the user to any errors that must be corrected. It can then return `true` to allow the dialog to be dismissed, or `false` to allow the user to correct any errors.

Dismissing a modal dialog

Every modal dialog should have at least one button that the user can click to dismiss the dialog. Typically modal dialogs have an **OK** and a **Cancel** button to close the dialog with or without accepting changes that were made in it.

You can define `onClick` callbacks for the buttons that close the parent dialog by calling its `close` method. You have the option of sending a value to the `close` method, which is in turn passed on to and returned from the `show` method that invoked the dialog. This return value allows your script to distinguish different closing events; for example, clicking **OK** can return 1, clicking **Cancel** can return 2. However, for this typical behavior, you do not need to define these callbacks explicitly; see 'Default and cancel elements' on page 66.

For some dialogs, such as a simple alert with only an **OK** button, you do not need to return any value. For more complex dialogs with several possible user actions, you might need to distinguish more outcomes. If you need to distinguish more than two closing states, you must define your own closing callbacks rather than relying on the default behavior.

If, by mistake, you create a modal dialog with no buttons to dismiss it, or if your dialog does have buttons, but their `onClick` handlers do not function properly, a user can still dismiss the dialog by typing ESC. In this case, the system will execute a call to the dialog's `close` method, passing a value of 2. This is not, of course, a recommended way to design your dialogs, but is provided as an escape hatch to prevent the application from hanging in case of an error in the operations of your dialog.

Default and cancel elements

The user can typically dismiss a modal dialog by clicking an **OK** or **Cancel** button, or by typing certain keyboard shortcuts. By convention, typing ENTER is the same as clicking **OK** or the default button, and

typing ESC is the same as clicking **Cancel**. The keyboard shortcut has the same effect as calling `notify` for the associated `button` control.

To determine which control is notified by which keyboard shortcut, set the `Dialog` object's `defaultElement` and `cancelElement` properties. The value is the control object that should be notified when the user types the associated keyboard shortcut.

- For buttons assigned as the `defaultElement`, if there is no `onClick` handler associated with the button, clicking the button or typing ENTER calls the parent dialog's `close` method, passing a value of 1 to be returned by the `show` call that opened the dialog.
- For buttons assigned as the `cancelElement`, if there is no `onClick` handler associated with the button, clicking the button or typing ESC calls the parent dialog's `close` method, passing a value of 2 to be returned by the `show` call that opened the dialog.

If you do not set the `defaultElement` and `cancelElement` properties explicitly, ScriptUI tries to choose reasonable defaults when the dialog is about to be shown for the first time. For the default element, it looks for a button whose `name` or `text` value is "`ok`" (disregarding case). For the cancel element, it looks for a button whose `name` or `text` value is "`cancel`" (disregarding case). Because it looks at the `name` value first, this works even if the `text` value is localized. If there is no suitable button in the dialog, the property value remains `null`, which means that the keyboard shortcut has no effect in that dialog.

To make this feature most useful, it is recommended that you always provide the `name` creation property for buttons meant to be used in this way.

Size and Location Objects

ScriptUI defines objects to represent the complex values of properties that place and size windows and UI elements. These objects cannot be created directly, but are created when you set the corresponding property. That property then returns that object. For example, the `bounds` property returns a `Bounds` object.

You can set these properties as objects, strings, or arrays.

- `e.`*`prop`* `= Object`: The object must contain the set of properties defined for this type, as shown in the table below. The properties have integer values.
- `e.`*`prop`* `= String`: The string must be an executable JavaScript inline object declaration, conforming to the same object description.
- `e.`*`prop`* `= Array`: The array must have integer coordinate values in the order defined for this type, as shown in the table below. For example:

The following examples show equivalent ways of placing a 380 by 390 pixel window near the upper left corner of the screen:

```
var dlg = new Window('dialog', 'Alert Box Builder');
dlg.bounds = {x:100, y:100, width:380, height:390}; //object
dlg.bounds = {left:100, top:100, right:480, bottom:490}; //object
dlg.bounds = "x:100, y:100, width:380, height:390"; //string
dlg.bounds = "left:100, top:100, right:480, bottom:490"; //string
dlg.bounds = [100,100,480,490]; //array
```

You can access the resulting object as an array with values in the order defined for the type, or as an object with the properties supported for the type.

Size and location object types

The following table shows the property-value object types, the element properties that create and contain them, and their array and object-property formats.

Bounds	Defines the boundaries of a window within the screen's coordinate space, or of a UI element within the container's coordinate space. Contains an array, `[left, top, right, bottom]`, that defines the coordinates of the upper left and lower right corners of the element. A `Bounds` object is created when you set an element's `bounds` property, and this property returns a `Bounds` object. • An object must contain properties named `left`, `top`, `right`, `bottom`, or `x`, `y`, `width`, `height`. • An array must have values in the order `[left, top, right, bottom]`.
Dimension	Defines the size of a `window` or UI element. Contains an array, `[width, height]`, that defines the element's size in pixels. A `Dimension` object is created when you set an element's `size` or `preferredSize` property. • An object must contain properties named `width` and `height`. • An array must have values in the order `[width, height]`.
Margins	Defines the number of pixels between the edges of a container and its outermost child elements. Contains an array `[left, top, right, bottom]` whose elements define the margins between the left edge of a container and its leftmost child element, and so on. A `Margins` object is created when you set an element's `margins` property. • An object must contain properties named `left`, `top`, `right`, and `bottom`. • An array must have values in the order `[left, top, right, bottom]`. You can also set the `margins` property to a number; all of the array values are then set to this number.
Point	Defines the location of a `window` or UI element. Contains an array, `[x, y]`, whose values represent the origin point of the element as horizontal and vertical pixel offsets from the origin of the element's coordinate space. A `Point` object is created when you set an element's `location` property. • An object must contain properties named `x` and `y`. • An array must have values in the order `[x, y]`.

Drawing Objects

ScriptUI allows you to draw directly on controls to customize their appearance. You can do this by calling methods of the ScriptUIGraphics Object in response to the onDraw event; see 'Defining Behavior with Event Callbacks and Listeners' on page 71. These methods take a number of helper object that encapsulate drawing information, including the following:

ScriptUIBrush	Describes the brush used to paint textures in a control.

ScriptUIFont	Describes the font used to write text into a control.
ScriptUIImage	Describes an image to be drawn in a control.
ScriptUIPath	Describes a drawing path for a figure to be drawn into a control.
ScriptUIPen	Describes the pen used to draw lines in a control.

Resource Specifications

You can create one or more UI elements at a time using a *resource specification*. This specially formatted string provides a simple and compact means of creating an element, including any container element and its component elements. The resource-specification string is passed as the `type` parameter to the `Window()` or `add()` constructor function.

The general structure of a resource specification is an element type specification (such as `dialog`), followed by a set of braces enclosing one or more property definitions.

```
var myResource = "dialog{ control_specs }";
var myDialog = new Window ( myResource );
```

Controls are defined as properties within windows and other containers. For each control, give the class name of the control, followed by the properties of the control enclosed in braces. For example, the following specifies a `Button`:

```
testBtn: Button { text: 'Test' }
```

The following resource string specifies a panel that contains several `StaticText` and `EditText` controls:

```
"msgPnl: Panel { text: 'Messages', bounds:[25,15,355,130], \
  titleSt: StaticText { text:'Alert box title:', \
     bounds:[15,15,105,35] }, \
  titleEt: EditText { text:'Sample Alert', bounds:[115,15,315,35] }, \
  msgSt: StaticText { text:'Alert message:', \
     bounds:[15,65,105,85] }, \
  msgEt: EditText { text:'<your message here>', \
     bounds:[115,45,315,105], properties:{multiline:true} } \
}"
```

The property with name `properties` specifies creation properties; see 'Creation properties' on page 59.

A property value can be specified as `null`, `true`, `false`, a string, a number, an inline array, or an object.

- An inline array contains one or more values in the form:

  `[value, value,...]`

- An object can be an inline object, or a named object, in the form:

  `{classname inlineObject}`

- An inline object contains one or more properties, in the form:

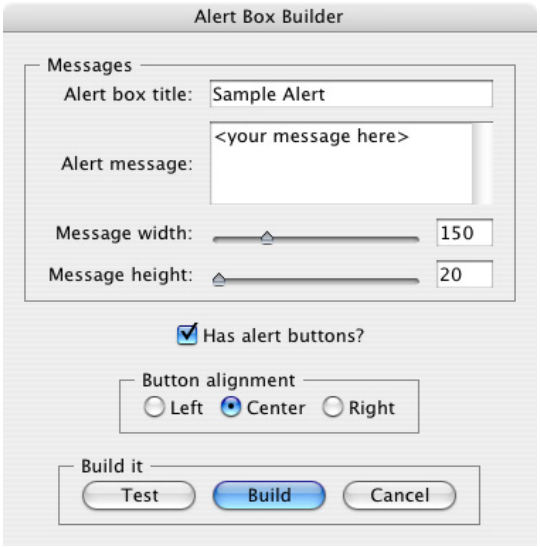
  `{propertyName:propertyValue,propertyName:propertyValue,... }`

Using resource strings

These examples in the Adobe Bridge and JavaScript SDK demonstrate how to use resource specification strings:

`AlertBoxBuilder1.jsx`	Demonstrates one way to use resource strings, creating a dialog that allows the user to enter some values, and then using those values to construct the resource string for a customizable alert dialog.
`AlertBoxBuilder2.jsx`	Constructs the same dialog, using a resource string (rather than the `add()` method) to specify all of the dialog contents for the user-input dialog.

The two Alert Box Builder examples create the same dialog to collect values from the user.



The Build button event handler builds a resource string from the collected values, and returns it from the dialog invocation function; the script then saves the resource string to a file. That resource string can later be used to create and display the user-configured alert box.

The resource specification format can also be used to create a single element or container and its child elements. For instance, if the `alertBuilderResource` in the example did not contain the panel `btnPnlResource`, you could define that resource separately, then add it to the dialog as follows:

```
var btnPnlResource =
   "panel { text: 'Build it', bounds:[15,330,365,375], \
      testBtn: Button { text:'Test', bounds:[15,15,115,35] }, \
      buildBtn: Button { text:'Build', bounds:[125,15,225,35], \
      properties:{name:'ok'} }, \
      cancelBtn: Button { text:'Cancel', bounds:[235,15,335,35], \
      properties:{name:'cancel'} } \
   }";
dlg = new Window(alertBuilderResource);
dlg.btnPnl = dlg.add(btnPnlResource);
dlg.show();
```

Defining Behavior with Event Callbacks and Listeners

You must define the behavior of your controls in order for them to respond to user interaction. You can do this by defining event-handling callback functions as part of the definition of the control or window. To respond to a specific event, define a handler function for it, and assign a reference to that function in the corresponding property of the window or control object. Different types of windows and controls respond to different actions, or events:

- Windows generate events when the user moves or resizes the window. To handle these events, define callback functions for onMove, onMoving, onResize, and onResizing. To respond to the user opening or closing the window, define callback functions for onShow and onClose.
- Button, radiobutton, and checkbox controls generate events when the user clicks within the control bounds. To handle the event, define a callback function for onClick.
- Edittext, scrollbar, and slider controls generate events when the content or value changes—that is, when the user types into an edit field, or moves the scroll or slider indicator. To handle these events, define callback functions for onChange and onChanging.
- Both containers and controls generate events just before they are drawn, that allow you to customize their appearance. To handle these events, define callback functions for onDraw. Your handler can modify or control how the container or control is drawn using the methods defined in the control's associated ScriptUIGraphics Object.
- In Windows only, you can register a key sequence as a shortcutKey for a control. To handle the key sequence, define a callback function for onShortcutKey in that control.

Defining event handler callback functions

Your script can define an event handler as a named function referenced by the callback property, or as an unnamed function defined inline in the callback property.

- If you define a named function, assign its name as the value of the corresponding callback property. For example:

```
function hasBtnsCbOnClick { /* do something interesting */ }
hasBtnsCb.onClick = hasBtnsCbOnClick;
```

- For a simple, unnamed function, set the property value directly to the function definition:

```
UI_element.callback_name = function () { handler_definition};
```

Event-handler functions take no arguments.

For example, the following sets the `onClick` property of the checkbox `hasBtnsCb`, to a function that enables another control in the same dialog:

```
hasBtnsCb.onClick = function ()
   { this.parent.alertBtnsPnl.enabled = this.value; };
```

The following statements set the `onClick` event handlers for buttons that close the containing dialog, returning different values to the `show` method that invoked the dialog, so that the calling script can tell which button was clicked:

```
buildBtn.onClick = function () { this.parent.parent.close(1); };
cancelBtn.onClick = function () { this.parent.parent.close(2); };
```

Simulating user events

You can simulate user actions by sending an event notification directly to a window or control with the `notify` method. A script can use this method to generate events in the controls of a window, as if a user was clicking buttons, entering text, or moving the window. If you have defined an event-handler callback for the element, the `notify` method invokes it.

The `notify` method takes an optional argument that specifies which event it should simulate. If a control can generate only one kind of event, notification generates that event by default.

The following controls generate the `onClick` event:

```
button
checkbox
iconbutton
radiobutton
```

The following controls generate the `onChange` event:

```
dropdownlist
edittext
listbox
scrollbar
slider
```

The following controls generate the `onChanging` event:

```
edittext
scrollbar
slider
```

In `radiobutton` and `checkbox` controls, the boolean `value` property automatically changes when the user clicks the control. If you use `notify()` to simulate a click, the `value` changes just as if the user had clicked. For example, if the `value` of a checkbox `hasBtnsCb` is `true`, this code changes the value to `false`:

```
if (dlg.hasBtnsCb.value == true) dlg.hasBtnsCb.notify();
// dlg.hasBtnsCb.value is now false
```

Registering event listeners for windows or controls

Note: Complete details not available at time of publication. Check for updated versions of this document at http://partners.adobe.com.

Another way to define the behavior of your windows and controls is register a handler function that responds to a specific type of event in that window or control. This technique allows you to respond to the cascading of an event through a hierarchy of containers and controls.

Use *`windowObj`*`.addEventListener()` or *`controlObj`*`.addEventListener()` to register a handler. The function you register receives a UIEvent Object that encapsulates the event information. As an event cascades down through a hierarchy and back up through the hierarchy, your handler can respond at any level, or use the `UIEvent` object's stopPropagation() method to stop the event propagation at some level.

You can register:

- The name of a handler function defined in the extension that takes one argument, the UIEvent Object. For example:

```
myButton.addEventListener( 'click', myFunction );
```

- A locally defined handler function that takes one argument, the UIEvent Object. For example:

```
myButton.addEventListener( 'click', 'function(e){/*handler code*/}');
```

The handler or registered code statement is executed when the specified event occurs in the target. A script can programmatically simulate an event by calling an event target's dispatchEvent() function.

You can remove a handler that has been previously registered by calling the event target's removeEventListener() function. The parameters you pass to this function must be identical to those passed to the addEventListener() call that registered the handler. Typically, a script would register all event handlers during initialization, and unregister them during termination; however, unregistering handlers on termination is not required.

You can register for an event in a parent or ancestor object of the actual target; see the following section.

The predefined types of `UIEvent` correspond to the event callbacks, as follows:

Callback	UIEvent type
onChange	change
onChanging	changing
onClick	click (detail = 1)
onDoubleClick	click (detail = 2)
onEnterKey	emterKey
onMove	move
onMoving	moving
onResize	resize
onResizing	resizing
onShow	show
onActivate	focus
onDeactivate	blur

How registered event-handlers are called

When an event occurs in a target, all handlers that have been registered for that event and target are called. Multiple event handlers can be registered for the same event in different targets, even in targets of the same type. For example, if there is a dialog with two checkboxes, you might want to register a `click` handler for each `checkbox` object. You would do this, for example, if each checkbox reacts differently to the click.

You can also register events for child objects with a parent object. If both checkboxes should react the same way to a mouse click, they require the same handler. In this case, you can register the handler with

the parent window or container instead. When the `click` event occurs in either child control, the handler registered for the parent window is called.

You can combine these two techniques, so that more than one action occurs in response to the event. That is, you can register a general event handler with the parent, and register a different, more specific handler for the same event with the child object that is the actual target.

The rules for how multiple event handlers are called depend on three phases of event propagation, as follows:

- Capture phase

  When an event occurs in an object hierarchy, it is *captured* by the topmost ancestor object at which a handler is registered (the window, for example). If no handler is registered for the topmost ancestor, ScriptUI looks for a handler for the next ancestor (the dialog, for example), on down through the hierarchy to the direct parent of actual target. When ScriptUI finds a handler registered for any ancestor of the target, it executes that handler then proceeds to the next phase.

- At-target phase

  ScriptUI calls any handlers that are rend1et, i1(e)-1.dh the chil14.8(tual)5.7t.1(ch)1.9(et. W)ecb01 Tc[ndleBubb)

Communicating with the Flash Application

ScriptUI supports a Flash Player, which runs the Flash application within a window in an Adobe application. The Flash application runs ActionScript, a different implementation of JavaScript from the ExtendScript version of JavaScript that Adobe applications run.

To open a Flash Player, add a control of type flashplayer to your ScriptUI window. A control object of this type contains functions that allow your script to load SWF files and control movie playback. It also contains functions that allow your Adobe application script to communicate with the ActionScript environment of the Flash application. See 'FlashPlayer control functions' on page 119.

A limited set of data types can be passed between the two scripting environments:

```
Number
String
Boolean
Null
undefined
Object
Array
```

The ActionScript `class` and `date` objects are not supported as parameter values.

In the ActionScript script for your Flash application, you must prepare for two-way communication by providing access to the External API. Do this by importing the `ExternalInterface` class into your Flash application:

```
import flash.external.ExternalInterface;
```

Calling ExtendScript functions from ActionScript

The Flash Player element allows you to call any JavaScript function that has been defined in the Adobe application script, and run it in the ActionScript environment. Any defined function can be called by name as a method of the flashplayer control object:

```
result = flashElement.myJavaScriptFunction ( [arg1, ..., argN] );
```

There are no special requirements for function names, but the function must take and return only data of the supported types.

You do not need to register the JavaScript function in the ActionScript environment. Your ActionScript script can simply call the external function using the `ExternalInterface.call()` method:

```
var res = ExternalInterface.call("myJavaScriptFunction");
```

Calling ActionScript functions from an Adobe script

From the ExtendScript side, use the FlashPlayer method invokePlayerFunction() to call ActionScript methods that have been defined within the Flash application:

```
result = flashElement.invokePlayerFunction ("ActionScript_function_name",
                                             [arg1, ..., argN] );
```

You can use the optional arguments to pass data (of supported types) to the ActionScript method.

Before you can call any ActionScript function from your Adobe application script, your Flash application must register that function with the `ExternalInterface` object, as a callback from the Flash container. To register a function, use the `ExtenalInterface.addCallback()` method:

```
public static addCallback (methodName:String, instance:Object,
                           method:Function)
```

For example, this registers a function defined in your Adobe application script named `getActionScriptArray()`:

```
ExternalInterface.addCallback("getActionScriptArray", this,
                              getActionScriptArray);
```

Flash Examples

These examples in the Adobe Bridge and JavaScript SDK demonstrate how to use the Flash Player:

`SnpCreateFlashControl.jsx`	Shows how to create a Flash Player, and use it to load a play back a movie defined in an SWF file.
`FlashDemo.jsx`	Shows how to communicate between the Adobe application scripting environment and the ActionScript scripting environment of the Flash Player.

Automatic Layout

When a script creates a window and its associated UI elements, it can explicitly control the size and location of each element and of the container elements, or it can take advantage of the automatic layout capability provided by ScriptUI. The automatic layout mechanism uses certain available information about UI elements, along with a set of layout rules, to establish a visually pleasing layout of the controls in a dialog, automatically determining the proper sizes for elements and containers.

Automatic layout is easier to program than explicit layout. It makes a script easier to modify and maintain, and it also makes the script easier to localize for different languages.

The script programmer has considerable control over the automatic layout process. Each container has an associated layout manager object, specified in the `layout` property. The layout manager controls the sizes and positions of the contained elements, and also sizes the container itself.

There is a default layout manager object, or you can create a new one:

```
myWin.layout = new AutoLayoutManager(myWin);
```

Default layout behavior

By default, the `autoLayoutManager` object implements the default layout behavior. A script can modify the properties of the default layout manager object, or create a new, custom layout manager if it needs more specialized layout behavior. See 'Custom layout manager example' on page 85.

Child elements of a container can be organized in a single row or column, or in a stack, where the elements overlap one other in the same region of the container, and only the top element is fully visible. This is controlled by the container's `orientation` property, which can have the value `row`, `column`, or `stack`.

You can nest `Panel` and `Group` containers to create more complex organizations. For example, to display two columns of controls, you can create a panel with a row orientation that in turn contains two groups, each with a column orientation.

Containers have properties to control inter-element spacing and margins within their edges. The layout manager provides defaults if these are not set.

The alignment of child elements within a container is controlled by the `alignChildren` property of the container, and the `alignment` property of the individual controls. The `alignChildren` property determines an overall strategy for the container, which can be overridden by a particular child element's `alignment` value.

A layout manager can determine the best size for a child element through the element's `preferredSize` property. The value defaults to dimensions determined by the UI framework based on characteristics of the control type and variable characteristics such as a displayed text string.

For details of how you can set these property values to affect the automatic layout, see 'Automatic layout properties' on page 78.

Note: ScriptUI does not offer direct control of fonts, and fonts are chosen differently on different platforms, so windows that are created the same way can appear different on different platforms.

Automatic layout properties

Your script establishes rules for the layout manager by setting the values of certain properties, both in the container object and in the child elements. The following examples show the effects of various combinations of values for these properties. The examples are based on a simple window containing a `StaticText`, `Button` and `EditText` element, created (using Resource Specifications) as follows:

```
var w = new Window(
   "window { \
      orientation: 'row', \
      st: StaticText { }, \
      pb: Button { text: 'OK' }, \
      et: EditText { size:[20, 30] } \
   }");
w.show();
```

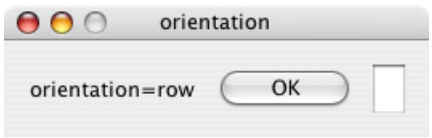
Each example shows the effects of setting particular layout properties in various ways. In each window, `w.text` is set so that the window title shows which property is being varied, and `w.st.text` is set to display the particular property value being demonstrated.

Container orientation

The `orientation` property of a container specifies the organization of child elements within it. It can have these values:

- `row`: Child elements are arranged next to each other, in a single row from left to right across the container. The height of the container is based on the height of the tallest child element in the row, and the width of the container is based on the combined widths of all the child elements.
- `column`: Child elements are arranged above and below each other, in a single column from top to bottom across the container. The height of the container is based on the combined heights of all the child elements, and the width of the container is based on the widest child element in the column.
- `stack`: Child elements are arranged overlapping one another, as in a stack of papers. The elements overlie one another in the same region of the container. Only the top element is fully visible. The height of the container is based on the height of the tallest child element in the stack, and the width of the container is based on the widest child element in the stack.

The following figure shows the results of laying out the sample window with each of these orientations:



Aligning children

The alignment of child elements within a container is controlled by two properties: `alignChildren` in the parent container, and `alignment` in each child. The `alignChildren` value in the parent container controls the alignment of all children within that container, unless it is overridden by the `alignment` value set on an individual child element.

These properties use the same values, which specify alignment along one axis, depending on the orientation of the container. You can specify an array of two of these strings, to specify alignment along both axes. The first string specifies the horizontal value, the second specifies the vertical value. The property values are not case-sensitive; for example, the strings `FILL`, `Fill`, and `fill` are all valid.

Elements in a row can be aligned along the vertical axis, in these ways:

- `top`: The element's top edge is located at the top margin of its container.
- `bottom`: The element's bottom edge is located at the bottom margin of its container.
- `center`: The element is centered within the top and bottom margins of its container.
- `fill`: The element's height is adjusted to fill the height of the container between the top and bottom margins.

Elements in a column can be aligned along the horizontal axis, in these ways:

- `left`: The element's left edge is located at the left margin of its container.
- `right`: The element's right edge is located at the right margin of its container.
- `center`: The element is centered within the right and left margins of its container.
- `fill`: The element's width is adjusted to fill the width of the container between the right and left margins.

Elements in a stack can be aligned along either the vertical or the horizontal axis, in these ways:

- `top`: The element's top edge is located at the top margin of its container, and the element is centered within the right and left margins of its container.
- `bottom`: The element's bottom edge is located at the bottom margin of its container, and the element is centered within the right and left margins of its container.
- `left`: The element's left edge is located at the left margin of its container, and the element is centered within the top and bottom margins of its container.
- `right`: The element's right edge is located at the right margin of its container, and the element is centered within the top and bottom margins of its container.
- `center`: The element is centered within the top, bottom, right and left margins of its container.
- `fill`: The element's height is adjusted to fill the height of the container between the top and bottom margins., and the element's width is adjusted to fill the width of the container between the right and left margins.

The following figure shows the results of creating the sample window with row orientation and the `bottom` and `top` alignment settings in the parent's `alignChildren` property:

The following figure shows the results of creating the sample window with column orientation and the `right`, `left`, and `fill` alignment settings in the parent's `alignChildren` property. Notice how in the `fill` case, each element is made as wide as the widest element in the container:

You can override the container's child alignment, as specified by `alignChildren`, by setting the `alignment` property of a particular child element. The following diagram shows the result of setting `alignment` to `right` for the `EditText` element, when the parent's `alignChildren` value is left:

Setting margins

The `margins` property of a container specifies the number of pixels between the edges of a container and the outermost edges of the child elements. You can set this property to a simple number to specify equal margins, or using a `Margins` object, which allows you to specify different margins for each edge of the container.

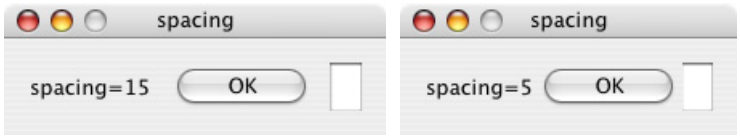
The following figure shows the results of creating the sample window with row orientation and margins of 5 and 15 pixels:

This figure shows the results of creating the sample window with column orientation, a top margin of 0 pixels, a bottom margin of 20 pixels, and left and right margins of 15 pixels:

Spacing between children

The `spacing` property of a container specifies the number of pixels separating one child element from its adjacent sibling element.

This figure shows the results of creating the sample window with row orientation, and spacing of 15 and 5 pixels, respectively:



This figure shows the results of creating the sample window with column orientation, and spacing of 20 pixels:

Determining a preferred size

Each element has a `preferredSize` property, which is initially defined with reasonable default dimensions for the element. The default value is calculated by ScriptUI, and is based on constant characteristics of each type of element, and variable characteristics such as the text string to be displayed in a button or text element.

If an element's `size` property is not defined, the layout manager uses the value of `preferredSize` to determine the dimensions of each element during the layout process. Generally, you should avoid setting the `preferredSize` property explicitly, and let ScriptUI determine the best value based on the state of an element at layout time. This allows you to set the `text` properties of your UI elements using localizable strings (see 'Localization in ScriptUI Objects' on page 87). The width and height of each element are calculated at layout time based on the chosen language-specific text string, rather than relying on the script to specify a fixed size for each element.

However, a script can explicitly set the `preferredSize` property to give hints to the layout manager about the intended sizes of elements for which a reasonable default size is not easily determined, such as an `EditText` element that has no initial text content to measure.

You can also set a maximum and/or minimum size value for a control, that limit how it can be resized. There is a default maximum size that prevents automatic layout from creating elements larger than the screen.

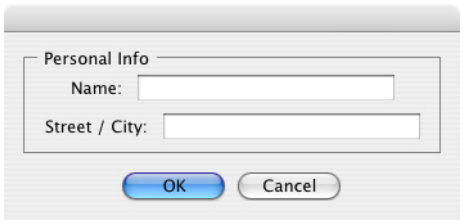
You can explicitly resize the controls in a window to fit the current text contents, or after the window is resized by the user, using the resize() method of the layout object.

Creating more complex arrangements

You can easily create more complex arrangements by nesting `Group` containers within `Panel` containers and other `Group` containers.

Many dialogs consist of rows of information to be filled in, where each row has columns of related types of controls. For instance, an edit field is typically in a row next to a static text label that identifies it, and a series of such rows are arranged in a column. This example (created using Resource Specifications) shows a simple dialog in which a user can enter information into two `EditText` fields, each arranged in a row with its `StaticText` label. To create the layout, a `Panel` with a column orientation contains two `Group` elements with row orientation. These groups contain the control rows. A third `Group`, outside the panel, contains the row of buttons.

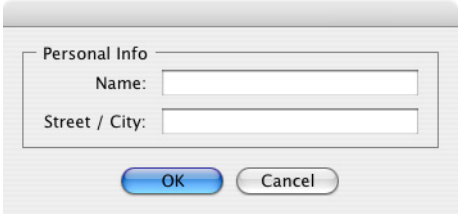
```
res =
"dialog { \
  info: Panel { orientation: 'column', \
     text: 'Personal Info', \
     name: Group { orientation: 'row', \
        s: StaticText { text:'Name:' }, \
        e: EditText { preferredSize: [200, 20] } \
     }, \
     addr: Group { orientation: 'row', \
        s: StaticText { text:'Street / City:' }, \
        e: EditText { preferredSize: [200, 20] } \
     }  \
  }, \
  buttons: Group { orientation: 'row', \
     okBtn: Button { text:'OK', properties:{name:'ok'} }, \
     cancelBtn: Button { text:'Cancel', properties:{name:'cancel'} } \
  } \
}";
win = new Window (res);
win.center();
win.show();
```



In this simplest example, the columns are not vertically aligned. When you are using fixed-width controls in your rows, a simple way to get an attractive alignment of the `StaticText` labels for your `EditText` fields is to align the child rows in the `Panel` to the right of the panel. In the example, add the following to the `Panel` specification:

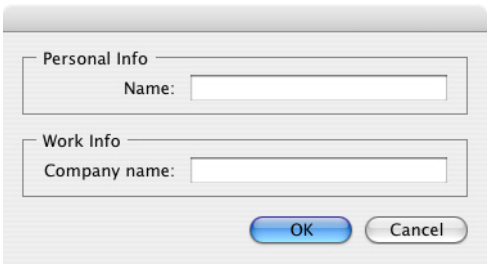
```
info: Panel { orientation: 'column', alignChildren:'right', \
```

This creates the following result:



Suppose now that you need two panels, and want each panel to have the same width in the dialog. You can specify this at the level of the dialog window object, the parent of both panels. Specify `alignChildren='fill'`, which makes each child of the dialog match its width to the widest child.

```
res =
   "dialog { alignChildren: 'fill', \
      info: Panel { orientation: 'column', alignChildren:'right', \
         text: 'Personal Info', \
         name: Group { orientation: 'row', \
            s: StaticText { text:'Name:' }, \
            e: EditText { preferredSize: [200, 20] } \
         } \
      }, \
      workInfo: Panel { orientation: 'column', \
         text: 'Work Info', \
         name: Group { orientation: 'row', \
            s: StaticText { text:'Company name:' }, \
            e: EditText { preferredSize: [200, 20] } \
         } \
      }, \
      buttons: Group { orientation: 'row', alignment: 'right', \
         okBtn: Button { text:'OK', properties:{name:'ok'} }, \
         cancelBtn: Button { text:'Cancel', properties:{name:'cancel'} } \
      } \
   }";
win = new Window (res); win.center(); win.show();
```



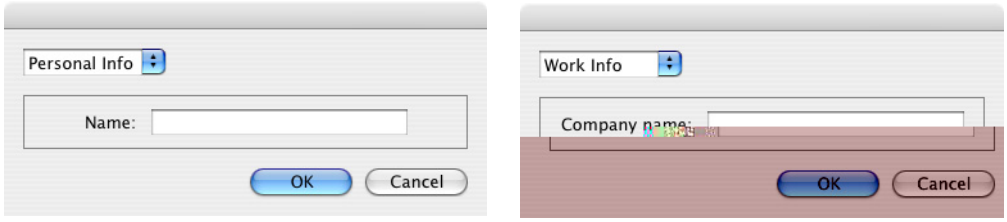
To make the buttons to appear at the right of the dialog, the `buttons` group overrides the `fill` alignment of its parent (the dialog), and specifies `alignment='right'`.

Creating dynamic content

Many dialogs need to present different sets of information based on the user selecting some option within the dialog. You can use the `stack` orientation to present different views in the same region of a dialog.

A `stack` orientation of a container places child elements so they are centered in a space which is wide enough to hold the widest child element, and tall enough to contain the tallest child element. If you arrange groups or panels in such a stack, you can show and hide them in different combinations to display a different set of controls in the same space, depending on other choices in the dialog.

For example, this dialog changes dynamically according to the user's choice in the `DropDownList`.



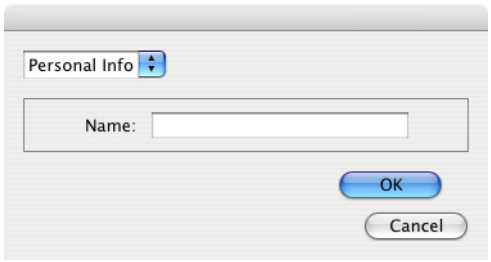
The following script creates this dialog. It compresses the "Personal Info" and "Work Info" panels from the previous example into a single `Panel` that has two `Groups` arranged in a stack. A `DropDownList` allows the user to choose which set of information to view. When the user makes a choice in the list, its `onChange` function shows one group, and hides the other.

```
res =
   "dialog { \
      whichInfo: DropDownList { alignment:'left' }, \
      allGroups: Panel { orientation:'stack', \
         info: Group { orientation: 'column', \
            name: Group { orientation: 'row', \
               s: StaticText { text:'Name:' }, \
               e: EditText { preferredSize: [200, 20] } \
            } \
         }, \
         workInfo: Group { orientation: 'column', \
            name: Group { orientation: 'row', \
               s: StaticText { text:'Company name:' }, \
               e: EditText { preferredSize: [200, 20] } \
            } \
            }, \
      }, \
      buttons: Group { orientation: 'row', alignment: 'right', \
         okBtn: Button { text:'OK', properties:{name:'ok'} }, \
         cancelBtn: Button { text:'Cancel', properties:{name:'cancel'} } \
      } \
   }";
win = new Window (res);
win.whichInfo.onChange = function () {
   if (this.selection != null) {
      for (var g = 0; g < this.items.length; g++)
         this.items[g].group.visible = false; //hide all other groups
      this.selection.group.visible = true;//show this group
   }
}
var item = win.whichInfo.add ('item', 'Personal Info');
item.group = win.allGroups.info;
item = win.whichInfo.add ('item', 'Work Info');
item.group = win.allGroups.workInfo;
win.whichInfo.selection = win.whichInfo.items[0];
win.center();
win.show();
```

Custom layout manager example

This script creates a dialog almost identical to the one in the previous example, except that it defines a layout-manager subclass, and assigns an instance of this class as the `layout` property for the last `Group` in the dialog. (The example also demonstrates the technique for defining a reusable class in JavaScript.)

This script-defined layout manager positions elements in its container in a stair-step fashion, so that the buttons are staggered rather than in a straight line.




```
/* Define a custom layout manager that arranges the children
** of 'container' in a stair-step fashion.*/

function StairStepButtonLayout (container) { this.initSelf(container); }

// Define its 'method' functions
function SSBL_initSelf (container) { this.container = container; }

function SSBL_layout() {
   var top = 0, left = 0;
   var width;
   var vspacing = 10, hspacing = 20;
   for (i = 0; i < this.container.children.length; i++) {
      var child = this.container.children[i];
      if (typeof child.layout != "undefined")
         // If child is a container, call its layout method
         child.layout.layout();
      child.size = child.preferredSize;
      child.location = [left, top];
      width = left + child.size.width;
      top += child.size.height + vspacing;
      left += hspacing;
   }
   this.container.preferredSize = [width, top - vspacing];
}

// Attach methods to Object's prototype
StairStepButtonLayout.prototype.initSelf = SSBL_initSelf;
StairStepButtonLayout.prototype.layout = SSBL_layout;

// Define a string containing the resource specification for the controls
res =
 "dialog { \
   whichInfo: DropDownList { alignment:'left' }, \
   allGroups: Panel { orientation:'stack', \
      info: Group { orientation: 'column', \
         name: Group { orientation: 'row', \
            s: StaticText { text:'Name:' }, \
```

```
            e: EditText { preferredSize: [200, 20] } \
          } \
        }, \
        workInfo: Group { orientation: 'column', \
          name: Group { orientation: 'row', \
            s: StaticText { text:'Company name:' }, \
            e: EditText { preferredSize: [200, 20] } \
          } \
        }, \
      }, \
      buttons: Group { orientation: 'row', alignment: 'right', \
        okBtn: Button { text:'OK', properties:{name:'ok'} }, \
        cancelBtn: Button { text:'Cancel', properties:{name:'cancel'} } \
      } \
    }";
// Create window using resource spec
win = new Window (res);
// Create list items, select first one
win.whichInfo.onChange = function () {
   if (this.selection != null) {
      for (var g = 0; g < this.items.length; g++)
         this.items[g].group.visible = false;
      this.selection.group.visible = true;
   }
}
var item = win.whichInfo.add ('item', 'Personal Info');
item.group = win.allGroups.info;
item = win.whichInfo.add ('item', 'Work Info');
item.group = win.allGroups.workInfo;
win.whichInfo.selection = win.whichInfo.items[0];

// Override the default layout manager for the 'buttons' group
// with custom layout manager
win.buttons.layout = new StairStepButtonLayout(win.buttons);

win.center();
win.show();
```

The AutoLayoutManager algorithm

When a script creates a `window` object and its elements and shows it the first time, the visible UI-platform window and controls are created. At this point, if no explicit placement of controls was specified by the script, all the controls are located at [0, 0] within their containers, and have default dimensions. Before the window is made visible, the layout manager's `layout` method is called to assign locations and sizes for all the elements and their containers.

The default `AutoLayoutManager`'s `layout` method performs these steps when invoked during the initial call to a `window` object's `show` method:

1. Read the `bounds` property for the managed container; if undefined, proceed with auto layout. If defined, assume that the script has explicitly placed the elements in this container, and cancel the layout operation (if both the `location` and `size` property have been set, this is equivalent to setting the `bounds` property, and layout does not proceed).

2. Determine the container's margins and inter-element spacing from its `margins` and `spacing` properties, and the orientation and alignment of its child elements from the container's `orientation` and `alignChildren` properties. If any of these properties are undefined, use default settings obtained from platform and UI framework-specific default values.

3. Enumerate the child elements, and for each child:
   - If the child is a container, call its layout manager (that is, execute this entire algorithm again for the container).
   - Read its `alignment` property; if defined, override the default alignment established by the parent container with its `alignChildren` property.
   - Read its `size` property: if defined, use it to determine the child's dimensions. If undefined, read its `preferredSize` property to get the child's dimensions. Ignore the child's `location` property.

   All the per-child information is collected for later use.

4. Based on the orientation, calculate the trial location of each child in the row or column, using inter-element spacing and the container's margins.

5. Determine the column, row, or stack dimensions, based on the dimensions of the children.

6. Using the desired alignment for each child element, adjust its trial location relative to the edges of its container. For stack orientation, center each child horizontally and vertically in its container.

7. Set the `bounds` property for each child element.

8. Set the container's `preferredSize` property, based on the margins and dimensions of the row or column of child elements.

Automatic layout restrictions

The following restrictions apply to the automatic layout mechanism:

- The default layout manager does not attempt to lay out a container that has a defined `bounds` property. The script programmer can override this behavior by defining a custom layout manager for the container.
- The layout mechanism does not track changes to element sizes after the initial layout has occurred. The script can initiate another layout by calling the layout manager's `layout` method, and can force the manager to recalculate the sizes of all child containers by passing the optional argument as `true`.

Localization in ScriptUI Objects

For portions of your user interface that are displayed on the screen, you may want to localize the displayed text. You can localize the display strings in any ScriptUI object simply and efficiently, using the global `localize` function. This function takes as its argument a *localization object* containing the localized versions of a string.

For complete details of this ExtendScript feature, see 'Localizing ExtendScript Strings' on page 204.

A localization object is a JavaScript object literal whose property names are locale names, and whose property values are the localized text strings. The locale name is an identifier as specified in the ISO 3166 standard. In this example, a `btnText` object contains localized text strings for several locales. This object supplies the text for a `Button` to be added to a window `w`:

```
btnText = { en: "Yes", de: "Ja", fr: "Oui" };
b1 = w.add ("button", undefined, localize (btnText));
```

The `localize` function extracts the proper string for the current locale. It matches the current locale and platform to one of the object's properties and returns the associated string. On a German system, for example, the property `de` provides the string `"Ja"`.

When your script uses localization to provide language-appropriate strings for UI elements, it should also take advantage of the Automatic Layout feature. The layout manager can determine the best size for each UI element based on its localized `text` value, automatically adjusting the layout of your script-defined dialogs to allow for the varying widths of strings for different languages.

Variable values in localized strings

The `localize` function allows you to include variables in the string values. Each variable is replaced with the result of evaluating an additional argument. For example:

```
today = {
   en: "Today is %1/%2.",
   de: "Heute ist der %2.%1."
   };
d = new Date();
Window.alert (localize (today, d.getMonth()+1, d.getDate()));
```

Enabling automatic localization

If you do not need variable replacement, you can use automatic localization. To turn on automatic localization, set the global value:

```
$.localization=true
```

When it is enabled, you can specify a localization object directly as the value of any property that takes a localizable string, without using the `localize` function. For example:

```
btnText = { en: "Yes", de: "Ja", fr: "Oui" };
b1 = w.add ("button", undefined, btnText);
```

The `localize` function always performs its translation, regardless of the setting of the `$.localize` variable. For example:

```
//Only works if the $.localize=true
b1 = w.add ("button", undefined, btnText);
//Always works, regardless of $.localize value
b1 = w.add ("button", undefined, localize (btnText));
```

If you need to include variables in the localized strings, use the `localize` function. .

ScriptUI Object Reference

ScriptUI is a component that works with the ExtendScript JavaScript interpreter to provide JavaScript programs with the ability to create and interact with user interface elements. It provides an object model for windows and UI control elements within an application.

This chapter provides the details of the ScriptUI classes and objects with their properties, methods, and creation parameters.

Global ScriptUI Object

The globally available `ScriptUI` object provides central information about the ScriptUI module. This object is not instantiable.

ScriptUI global properties

`alignment`	String or Array of 2 Strings	The default alignment style for automatic layout. When not specified, all alignments default to `center`.
		This can be a single string, which indicates the alignment for the `orientation` specified in the parent container, or an array of two strings, indicating both the horizontal and vertical alignment (in that order).
		Allowed values depend on the `orientation` value of the parent container. For `orientation=row`:
		`top`
		`bottom`
		`fill`
		For `orientation=column`:
		`left`
		`right`
		`fill`
		For `orientation=stack`:
		`top`
		`bottom`
		`left`
		`right`
		`fill`

`applicationFonts`	Object	The default application fonts.
`compatability`	Object	*Description not available at time of publication*

coreVersion	String	The internal core version number of the ScriptUI components. Read only.
fontStyle	String	The default font style for controls that contain text. One of: REGULAR BOLD ITALIC BOLDITALIC
frameworkName	String	The name of the user-interface framework with which this ScriptUI component is compatible. Read only.
version	String	The main version number of the ScriptUI component framework. Read only.

ScriptUI global functions

getResourceText()

```
ScriptUI.getResourceText (text)
```

text The text to match.

Finds and returns the resource for a given text string from the host application's resource data. If no string resource matches the given text, the text itself is returned.

Returns a String.

newFont()

```
ScriptUI.newFont ( name, style, size );
```

name The font or font family name string.

style The font style string or number.

size The font size in points, a number.

Creates a new global font for use in text controls and titles.

Returns a ScriptUIFont Object.

newImage()

```
ScriptUI.newImage ( normal, disabled, pressed, rollover );
```

normal The resource name or path to the image to use for the normal state.

disabled The resource name or path to the image to use for the disabled state.

pressed The resource name or path to the image to use for the pressed state.

rollover The resource name or path to the image to use for the rollover state.

Creates a new global image object for use in controls that can display images, loading the associated images from the specified resources or image files.

Returns a ScriptUIImage Object.

Global Window Object

The globally available `Window` object defines these static properties and functions. Window instances created with `new Window()` do not have these properties and functions defined.

Window global properties

`frameworkName`	String	The name of the user-interface framework with which this ScriptUI component is compatible. Read only.
`version`	String	The main version number of the ScriptUI component framework. Read only.

Window global functions

`alert()`

```
Window.alert (message[, title, errorIcon]);
```

`message` The string for the displayed message.

`title` Optional. A string to appear as the title of the dialog, if the platform supports a title. Mac OS does not support titles for alert dialogs. The default title string is "Script Alert".

`errorIcon` Optional. When `true`, the platform-standard alert icon is replaced by the platform-standard error icon in the dialog. Default is `false`.

Displays a platform-standard dialog containing a short message and an **OK** button.

Returns `undefined`

`confirm()`

```
Window.confirm (message[,noAsDflt ,title ]);
```

`message` The string for the displayed message.

`noAsDflt` Optional. When `true`, the **No** button is the default choice, selected when the user types ENTER. Default is `false`, meaning that **Yes** is the default choice.

`title` Optional. A string to appear as the title of the dialog, if the platform supports a title. Mac OS does not support titles for confirmation dialogs. The default title string is "Script Alert".

Displays a platform-standard dialog containing a short message and two buttons labeled **Yes** and **No**.

Returns `true` if the user clicked **Yes**, `false` if the user clicked **No**.

find()

```
Window.find (resourceName)
Window.find (type, title)
```

resourceName	The name of a predefined resource available to JavaScript in the current application.
type	Optional. The window type (see 'Window object constructor' on page 92) used if there is more than one window with the same title. Can be `null` or the empty string.
title	The window title.

Use this method to find an existing window. This includes windows already created by a script, and windows created by the application (if the application supports this case).

Note: Not supported in all ScriptUI implementations.

Returns a window object found or generated from the resource, or `null` if no such window or resource exists.

prompt()

```
Window.prompt (message, preset[, title ]);
```

message	The string for the displayed message.
preset	The initial value to be displayed in the text edit field.
title	Optional. A string to appear as the title of the dialog. In Windows, this appears in the window's frame; in Mac OS it appears above the message. The default title string is "Script Prompt".

Displays a modal dialog that returns the user's text input.

Returns the value of the text edit field if the user clicked **OK**, `null` if the user clicked **Cancel**.

Window Object

Window object constructor

The constructor creates and returns a new `window` object, or `null` if window creation failed.

```
new Window (type [, title, bounds, {creation_properties}]);
```

type	The window type. The value is: `dialog`: Creates a modal dialog. `palette`: Creates a modeless dialog, also called a floating palette. (Not supported by Photoshop CS3.) `window`: Creates a simple window that can be used as a main window for an application. (Not supported by Photoshop CS3.) This argument can be a ScriptUI resource specification; in this case, all other arguments are ignored. See 'Resource Specifications' on page 69.
title	Optional. The window title. A localizable string.
bounds	Optional. The window's position and size.

creation_properties	Optional. An object that can contain any of these properties: `resizeable`: When `true`, the window can be resized by the user. Default is `false`. `su1PanelCoordinates`: Photoshop only. When `true`, the child panels of this window automatically adjust the positions of their children for compatability with Photoshop CS (in which the vertical coordinate was measured from outside the frame). Default is `false`. Individual panels can override the parent window's setting. `closeButton`: Bridge only. When `true`, the title bar includes a button to close the window, if the platform and window type allow it. When `false`, it does not. Default is `true`. Not used for dialogs. `maximizeButton`: Bridge only. When `true`, the title bar includes a button to expand the window to its maximum size (typically, the entire screen), if the platform and window type allow it. When `false`, it does not. Default is `false` for type `palette`, `true` for type `window`. Not used for dialogs. `minimizeButton`: Bridge only. When `true`, the title bar includes a button to minimize or iconify the window, if the platform and window type allow it. When `false`, it does not. Default is `false` for type `palette`, `true` for type `window`. Main windows cannot have a minimize button in Mac OS. Not used for dialogs. `independent`: Bridge only. When `true`, a window of type `window` is independent of other application windows, and can be hidden behind them in Windows. In Mac OS, has no effect. Default is `false`.

Common properties

All types of UI elements, including windows, containers, and controls, share many of the same properties, although some have slightly different meanings for different types of objects. The following table summarizes which properties are used in which object types.

Property	Window	Panel	Group	Button	CheckBox	DropDownList	EditText	FlashPlayer	IconButton	Image	ListBox	ListItem	ProgressBar	RadioButton	Scrollbar	Slider	StaticText	TreeView
`active`	x			x	x	x	x	x	x	x	x			x	x	x	x	x
`alignChildren`	x	x	x															
`alignment`	x	x	x	x	x	x	x	x	x	x	x		x	x	x	x	x	x
`bounds`	x	x	x	x	x	x	x	x	x	x	x		x	x	x	x	x	x
`cancelElement`	x																	
`characters`	x	x	x															
`checked`												x						
`children`	x	x	x	x	x	x	x	x	x	x	x		x	x	x	x	x	x

Property	Window	Panel	Group	Button	CheckBox	DropDownList	EditText	FlashPlayer	IconButton	Image	ListBox	ListItem	ProgressBar	RadioButton	Scrollbar	Slider	StaticText	TreeView
closeButton	x																	
defaultElement	x																	
enabled	x	x	x	x	x	x	x	x	x	x	x		x	x	x	x	x	x
expanded												x						
frameBounds	x																	
frameLocation	x																	
frameSize	x																	
graphics	x	x	x	x	x	x	x	x	x	x	x		x	x	x	x	x	x
helpTip	x	x	x	x	x	x	x	x	x	x	x		x	x	x	x	x	x
icon									x	x		x						
image												x						
independent	x																	
index												x						
items							x				x							x
itemSize							x				x							x
jumpdelta															x			
justify	x	x		x	x		x							x			x	
layout	x	x	x															
location	x	x	x	x	x	x	x	x	x	x	x		x	x	x	x	x	x
margins	x	x	x															
maximiaeButton	x																	
maximumSize	x	x	x	x	x	x	x	x	x	x	x		x	x	x	x	x	x
maxvalue													x		x	x		
minimiaeButton	x																	
minimumSize	x	x	x	x	x	x	x	x	x	x	x		x	x	x	x	x	x
minvalue													x		x	x		
orientation	x	x	x															
parent	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x

Property	Window	Panel	Group	Button	CheckBox	DropDownList	EditText	FlashPlayer	IconButton	Image	ListBox	ListItem	ProgressBar	RadioButton	Scrollbar	Slider	StaticText	TreeView
preferredSize	x	x	x	x	x	x	x	x	x	x	x		x	x	x	x	x	x
properties	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
resizeable	x																	
selected												x						
selection						x					x							x
shortcutKey	x			x	x	x	x	x	x		x			x	x	x	x	x
size	x	x	x	x	x	x	x	x	x	x	x		x	x	x	x	x	x
spacing	x	x	x															
stepdelta															x			
text	x	x		x	x		x					x		x			x	
textselection							x											
type	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
value					x								x	x	x	x		
visible	x	x	x	x	x	x	x	x	x	x	x		x	x	x	x	x	x
window	x	x	x	x	x	x	x	x	x	x	x		x	x	x	x	x	x
windowBounds	x	x	x	x	x	x	x	x	x	x	x		x	x	x	x	x	x

Window object properties

The following element properties apply specifically to `Window` elements:

`active`	Boolean	When `true`, the object is active, `false` otherwise. Set to `true` to make a given control or dialog active. • A modal dialog that is visible is by definition the active dialog. • An active palette is the front-most window. • An active control is the one with focus—that is, the one that accepts keystrokes, or in the case of a `Button`, be selected when the user types RETURN or ENTER.
`cancelElement`	Object	For a window of type `dialog`, the control to notify when a user types the ESC key in Windows, or the CMD . (dot) combination in Mac OS. By default, looks for a `button` whose name or text is "`cancel`" (case disregarded).

closeButton	Boolean	When `true`, the title bar should show a platform-specific button for closing the window. Ignored for window types and platforms that do not support the close button.
closeOnKey	String	A keyboard shortcut for closing this window.
defaultElement	Object	For a window of type `dialog`, the control to notify when a user types the ENTER key. By default, looks for a `button` whose name or text is "ok" (case disregarded).
frameBounds	Bounds	A Bounds object for the boundaries of the Window's frame in screen coordinates. The frame consists of the title bar and borders that enclose the content region of a window, depending on the windowing system. Read only.
frameLocation	Point	A Point object for the location of the top left corner of the Window's frame. The same as `[frameBounds.x, frameBounds.y]`. Set this value to move the window frame to the specified location on the screen. The `frameBounds` value changes accordingly.
frameSize	Dimension	A Dimension object for the size and location of the Window's frame in screen coordinates. Read only.
independent	Boolean	In Windows only, for top-level windows created with type `window`. When `true`, this window is independent of all other application windows, and can be hidden behind them. Default is `false`, meaning this window cannot be hidden behind other windows in this application.
maximizeButton	Boolean	When `true`, the title bar should show a platform-specific button for expanding the window to its maximum size (typically the entire screen). Ignored for window types and platforms that do not support the maximize button.
maximized	Boolean	When `true`, the window is expanded.
minimizeButton	Boolean	When `true`, the title bar should show a platform-specific button for minimizing or iconifying the window. Ignored for window types and platforms that do not support the minimize button.
minimized	Boolean	When `true`, the window is minimized or iconified.
resizeable	Boolean	When `true`, the window can be resized in the platform-standard way. Default is `false`.
shortcutKey	String	The key sequence that invokes this window's onShortcutKey callback (in Windows only).
su1PanelCoordinates	Boolean	Photoshop CS2 only. When `true`, panels in this window use the obsolete coordinate system of Photoshop CS1, and require automatic adjustment of the locations of child elements. Default is `false`.

Container properties

The following table shows properties that apply to `Window` objects and container objects (controls of type `panel` and `group`).

`alignChildren`	String	Tells the layout manager how unlike-sized children of a container should be aligned within a column or row. Order of creation determines which children are at the top of a column or the left of a row; the earlier a child is created, the closer it is to the top or left of its column or row.
		If defined, `alignment` for a child element overrides the `alignChildren` setting for the parent container.
		Allowed values depend on the `orientation` value. For `orientation=row`:
		`top`
		`bottom`
		`center (default)`
		`fill`
		For `orientation=column`:
		`left`
		`right`
		`center (default)`
		`fill`
		For `orientation=stack`:
		`top`
		`bottom`
		`left`
		`right`
		`center (default)`
		`fill`
		Values are not case sensitive.

alignment	String	Applies to child elements of a container. If defined, this value overrides the `alignChildren` setting for the parent container.
		
		This can be a single string, which indicates the alignment for the `orientation` specified in the parent container, or an array of two strings, indicating both the horizontal and vertical alignment (in that order).
		
		Allowed values depend on the `orientation` value of the parent container. For `orientation=row`:
		
		`top`
		`bottom`
		`center (default)`
		`fill`
		
		For `orientation=column`:
		
		`left`
		`right`
		`center (default)`
		`fill`
		
		For `orientation=stack`:
		
		`top`
		`bottom`
		`left`
		`right`
		`center (default)`
		`fill`
		
		Values are not case sensitive.

bounds	Bounds	A Bounds object for the boundaries of the window's drawable area in screen coordinates. Compare frameBounds. Read only.
characters	Number	A number of characters for which to reserve space when calculating the preferred size of the window. A positive integer.
children	Array of Object	The collection of UI elements that have been added to this container (`window`, `panel`, `group`). An array indexed by number or by a string containing an element's `name`. The `length` property of this array is the number of child elements for container elements, and is zero for controls. Read only.
graphics	Graphics	A ScriptUIGraphics Object that can be used to customize the window's appearance, in response to the onDraw event.

justify	String	The justification style for text in this window or panel. Does not apply to a container of type `group`. One of:
		
		`left`
		`center`
		`right`

layout	LayoutManager	A LayoutManager Object for a container (`window`, `panel`, `group`). The first time a container object is made visible, ScriptUI invokes this layout manager by calling its `layout` function. Default is an instance of the `LayoutManager` class that is automatically created when the container element is created.
location	Point	A Point object for the location of the top left corner of the Window's drawable area. The same as `[bounds.x, bounds.y]`.
margins	Margins	A Margins object describing the number of pixels between the edges of this container and the outermost child elements. You can specify different margins for each edge of the container. The default value is based on the type of container, and is chosen to match the standard Adobe UI guidelines.
maximumSize	Dimension	A Dimension object for the largest rectangle to which the window can be resized, used in automatic layout and resizing.
minimumSize	Dimension	A Dimension object for the smallest rectangle to which the window can be resized, used in automatic layout and resizing.

orientation	String	How elements are organized within this container. Interpreted by the layout manager for the container. The default LayoutManager Object accepts the (case-insensitive) values:
		`row`
		`column`
		`stack`
		The default orientation depends on the type of container. For `Window` and `Panel`, the default is `column`, and for `Group` the default is `row`.
		The allowed values for the container's `alignChildren` and its children's `alignment` properties depend on the orientation.

parent	Object	The immediate parent object of this element, a `window`, `panel` or `group`. The value is `null` for `Window` objects. Read only.
preferredSize		

spacing	Number	The number of pixels separating one child element from its adjacent sibling element. Because each container holds only a single row or column of children, only a single spacing value is needed for a container. The default value is based on the type of co ntainer, and is chosen to match standard Adobe UI guidelines.
text	String	The title, label, or displayed text. Does not apply to containers of type `group`. This is a localizable string: see 'Localization in ScriptUI Objects' on page 87.
visible	Boolean	When `true`, the element is shown, when `false` it is hidden. When a container is hidden, its children are also hidden, but they retain their own visibility values, and are shown or hidden accordingly when the parent is next shown.
window	Window	The top-level parent window of this container, a Window Object.
windowBounds	Bounds	A Bounds object for the size and location of this container relative to its top-level parent window.

Window object functions

These functions are defined for `Window` instances.

add()

```
windowObj.add (type [, bounds, text, { creation_props> } ]);
```

type — The control type. See 'Control types and creation parameters' on page 105.

bounds — Optional. A bounds specification that describes the size and position of the new control or container, relative to its parent. See Bounds object for specification formats.

If supplied, this value creates a new Bounds object which is assigned to the new object's `bounds` property.

text — Optional. String. Initial text to be displayed in the control as the title, label, or contents, depending on the control type. If supplied, this value is assigned to the new object's `text` property.

creation_props — Optional. Object. The properties of this object specify creation parameters, which are specific to each object type. See 'Control types and creation parameters' on page 105.

Creates and returns a new control or container object and adds it to the children of this window.

Returns the new object, or `null` if unable to create the object.

addEventListener()

```
windowObj.addEventListener (eventName, handler[, capturePhase]);
```

eventName The event name string.

handler The function to register for the specified event in this target. This can be the name of a function defined in the extension, or a locally defined handler function to be executed when the event occurs. A handler function takes one argument, the UIEvent Object. See 'Registering event listeners for windows or controls' on page 72.

capturePhase Optional. When `true`, the handler is called only in the capturing phase of the event propagation. Default is `false`, meaning that the handler is called in the bubbling phase if this object is an ancestor of the target, or in the at-target phase if this object is itself the target.

Registers an event handler for a particular type of event occuring in this window.

Returns `undefined`.

center()

```
windowObj.center ([window])
```

window Optional. A Window Object.

Centers this window on the screen, or with respect to another specified window.

Returns `undefined`.

close()

```
windowObj.close ([result])
```

result Optional. A number to be returned from the `show` method that invoked this window as a modal dialog.

Closes this window. If an onClose callback is defined for the window, calls that function before closing the window.

Returns `undefined`.

dispatchEvent()

```
windowObj.dispatchEvent ( eventObj, data )
```

eventObj An UIEvent Object.

data Optional. Data to pass to the event handler, of the type appropriate to the type of event object.

Simulates the occurrence of an event in this target. A script can create a UIEvent Object for a specific event and pass it to this method to start the event propagation for the event.

Returns the UIEvent Object.

hide()

windowObj.hide()

Hides this window. When a window is hidden, its children are also hidden, but when it is shown again, the children retain their own visibility states.

For a modal dialog, closes the dialog and sets its result to 0.

Returns undefined.

notify()

windowObj.notify([*event*])

event

show()

```
windowObj.show()
```

Shows this window, container, or control. If an onShow callback is defined for a window, calls that function before showing the window.

When a window or container is hidden, its children are also hidden, but when it is shown again, the children retain their own visibility states.

For a modal dialog, opens the dialog and does not return until the dialog is dismissed. If it is dismissed via the close() method, this method returns any *result* value passed to that method. Otherwise, returns 0.

Window event-handling callbacks

The following callback functions can be defined to respond to events in windows. To respond to an event, define a function with the corresponding name in the `Window` instance. These callbacks are not available for other container types (controls of type `panel` or `group`).

Callback	Description
onClose	Called when a request is made to close the window, either by an explicit call to the close() function or by a user action (clicking the OS-specific close icon in the title bar). The function is called before the window actually closes; it can return `false` to cancel the close operation.
onDraw	Called when a container or control is about to be drawn. Allows the script to modify or control the appearance, using the control's associated ScriptUIGraphics Object. Handler takes one argument, a DrawState Object.
onMove	Called when the window has been moved.
onMoving	Called while a window in being moved, each time the position changes. A handler can monitor the move operation.
onResize	Called when the window has been resized.
onResizing	Called while a window is being resized, each time the height or width changes. A handler can monitor the resize operation.
onShortcutKey	(In Windows only) Called when a shortcut-key sequence is typed that matches the shortcutKey value for this window.
onShow	Called when a request is made to open the window using the show() method, before the window is made visible, but after automatic layout is complete. A handler can modify the results of the automatic layout.

Control Objects

Control object constructors

Use the `add` method to create new containers and controls. The `add` method is available on `window` and container (`panel` and `group`) objects. (See also add() for dropdownlist and listbox controls.)

add( )

```
containerObj.(type [, bounds, text, { creation_props> } ]);
```

type	The control type. See 'Control types and creation parameters' on page 105.
bounds	Optional. A bounds specification that describes the size and position of the new control or container, relative to its parent. See Bounds object for specification formats. If supplied, this value creates a new Bounds object which is assigned to the new object's bounds property.
text	Optional. String. Initial text to be displayed in the control as the title, label, or contents, depending on the control type. If supplied, this value is assigned to the new object's `text` property.
creation_props	Optional. Object. The properties of this object specify creation parameters, which are specific to each object type. See 'Control types and creation parameters' on page 105.

Creates and returns a new control or container object and adds it to the children of this window or container.

Returns the new object, or `null` if unable to create the object.

Control types and creation parameters

The following type names can be used in string literals as the `type` specifier for the `add` method, available on `window` and container (`panel` and `group`) objects. The class names can used in resource specifications to define controls within a window or panel.

Type name	Class name	Description
`button`	`Button`	A pushbutton containing a mouse-sensitive text string. Calls the onClick callback if the control is clicked or if its notify() method is called. To add to a window `w`: `w.add ("button" [, bounds, text ]);` *bounds*: Optional. The control's position and size. *text*: Optional. The text displayed in the control.
`checkbox`	`Checkbox`	A dual-state control showing a box with a checkmark when `value=true`, empty when `value=false`. Calls the onClick callback if the control is clicked or if its notify() method is called. To add to a window `w`: `w.add ("checkbox" [, bounds, text ]);` *bounds*: Optional. The control's position and size. *text*: Optional. The text displayed in the control.

`dropdownlist`	`DropDownList`	A drop-down list with zero or more items. Calls the onChange callback if the item selection is changed or if its notify() method is called.
		To add to a window `w`:
		`w.add ("dropdownlist", bounds [, items]`
		`[, {creation_properties}]);`
		bounds: The control's position and size.
		items: Optional. Supply this argument or the *creation_properties* argument, not both. An array of strings for the text of each list item. A `ListItem` object is created for each item. An item with the text string `"-"` creates a separator item.
		creation_properties: Optional. Supply this argument or the *items* argument, not both. This form is most useful for elements defined using Resource Specifications. An object that contains the following property:
		`items`: An array of strings for the text of each list item. A `ListItem` object is created for each item. An item with the text string `"-"` creates a separator item.

Type name	Class name	Description
edittext	EditText	An editable text field that the user can change. Calls the onChange callback if the text is changed and the user types ENTER or the control loses focus, or if its notify() method is called. Calls the onChanging callback when any change is made to the text. The textselection property contains currently selected text. To add to a window w: `w.add ("edittext" [, bounds, text, {creation_properties}]);` *bounds*: Optional. The control's position and size. *text*: Optional. The text displayed in the control. *creation_properties*: Optional. An object that contains any of the following properties: **multiline**: When false (the default), the control accepts a single line of text. When true, the control accepts multiple lines, in which case the text wraps within the width of the control. **readonly**: When false (the default), the control accepts text input. When true, the control does not accept input but only displays the contents of the text property. **noecho**: When false (the default), the control displays input text. When true, the control does not display input text (used for password input fields). **enterKeySignalsOnChange**: When false (the default), the control signals an onChange event when the editable text is changed and the control loses the keyboard focus (that is, the user tabs to another control, clicks outside the control, or types ENTER). When true, the control only signals an onChange event when the editable text is changed and the user types ENTER; other changes to the keyboard focus do not signal the event.
flashplayer	FlashPlayer	A control that contains a Flash Player, which can load and play Flash movies stored in SWF files. The ScriptUI FlashPlayer element runs the Flash application within an Adobe application. The Flash application runs ActionScript, a different implementation of JavaScript from the ExtendScript version of JavaScript that Adobe applications run. A control object of this type contains functions that allow your script to load SWF files, control movie playback, and communicate with the ActionScript environment. See 'FlashPlayer control functions' on page 119. To add to a window w: `w.add ("flashplayer" [, bounds]);` *bounds*: Optional. The control's position and size.

Type name	Class name	Description
group	Group	A container for other controls. Containers have additional properties that control the children; see 'Container properties' on page 97. Hiding a group hides all its children. Making it visible makes visible those children that are not individually hidden. To add to a window w: `w.add ("group" [, bounds]);` *bounds*: Optional. The element's position and size.

iconbutton	IconButton	A mouse-sensitive pushbutton containing an icon. Calls the onClick callback if the control is clicked or if its notify() method is called.
		
		To add to a window w:
		`w.add ("iconbutton" [, bounds, icon,`
		`{creation_properties}]);`
		
		bounds: Optional. The control's position and size.
		icon: Optional. The named resource for the icon or family of icons displayed in the button control, or a pathname or File Object for an image file. Images must be in PNG format.
		creation_properties: Optional. An object that contains the following property:
		style: A string for the visual style, one of:
		button: Has a visible border with a raised or 3D appearance.
		toolbutton: Has a flat appearance, appropriate for inclusion in a toolbar

image	Image	Displays an icon or image. To add to a window w: `w.add ("image" [, bounds, icon]);` *bounds*: Optional. The control's position and size. *icon*: Optional. The named resource for the icon or family of icons displayed in the image control, or a pathname or File Object for an image file. Images must be in PNG format.
item	Array of ListItem	The choice items in a list box or drop-down list. The objects are created when items are specified on creation of the parent list object, or afterward using the list control's add() method. • Items in a drop-down list can be of type separator, in which case they cannot be selected, and are shown as a horizontal line. Item objects have these properties which are not found in other controls: checked expanded image index selected

Type name	Class name	Description
listbox	ListBox	A list box with zero or more items. Calls the onChange callback if the item selection is changed or if its notify() method is called. To add to a window w: `w.add ("listbox", bounds [, items, {creation_properties}]);` *bounds*: Optional. The control's position and size. *items*: Optional. An array of strings for the text of each list item. A ListItem object is created for each item. Supply this argument, or the items property in *creation_properties*, not both. *creation_properties*: Optional. An object that contains any of the following properties: **multiselect**: When false (the default), only one item can be selected. When true, multiple items can be selected. **items**: An array of strings for the text of each list item. A ListItem object is created for each item. An item with the text string "-" creates a separator item. Supply this property, or the items argument, not both. This form is most useful for elements defined using Resource Specifications.
panel	Panel	A container for other types of controls, with an optional frame. Containers have additional properties that control the children; see 'Container properties' on page 97. Hiding a panel hides all its children. Making it visible makes visible those children that are not individually hidden. To add to a window w: `w.add ("panel" [, bounds, text, {creation_properties} ]);` *bounds*: Optional. The element's position and size. A panel whose width is 0 appears as a vertical line. A panel whose height is 0 appears as a horizontal line. *text*: Optional. The text displayed in the border of the panel. *creation_properties*: Optional. An object that contains the following property: **borderStyle**: A string that specifies the appearance of the border drawn around the panel. One of black, etched, gray, raised, sunken. Default is etched. **su1PanelCoordinates**: When true, this panel automatically adjusts the positions of its children for compatability with Photoshop CS. Default is false, meaning that the panel does not adjust the positions of its children, even if the parent window has automatic adjustment enabled.

Type name	Class name	Description
progressbar	Progressbar	A horizontal rectangle that shows progress of an operation. All progressbar controls have a horizontal orientation. The value property contains the current position of the progress indicator; the default is 0. There is a minvalue property, but it is always 0; attempts to set it to a different value are silently ignored. To add to a window w: `w.add ("progressbar" [, bounds, value, maxvalue]);` *bounds*: Optional. The control's position and size. *value*: Optional. The initial position of the progress indicator. Default is 0. *maxvalue*: Optional. The maximum value that the value property can be set to. Default is 100.
radiobutton	RadioButton	A dual-state control, grouped with other radiobuttons, of which only one can be in the selected state. Shows the selected state when value=true, empty when value=false. Calls the onClick callback if the control is clicked or if its notify() method is called. All radiobuttons in a group must be created sequentially, with no intervening creation of other element types. Only one radiobutton in a group can be set at a time; setting a different radiobutton unsets the original one. To add to a window w: `w.add ("radiobutton" [, bounds, text ]);` *bounds*: Optional. The control's position and size. *text*: Optional. The text displayed in the control.

Type name	Class name	Description
scrollbar	Scrollbar	A scrollbar with a draggable scroll indicator and stepper buttons to move the indicator. The scrollbar control has a horizontal orientation if the width is greater than the height at creation time, or vertical if its height is greater than its width. Calls the onChange callback after the position of the indicator is changed or if its notify() method is called. Calls the onChanging callback repeatedly while the user is moving the indicator. The value property contains the current position of the scrollbar's indicator within the scrolling area, within the range of minvalue and maxvalue. The stepdelta property determines the scrolling unit for the up or down arrow; default is 1. The jumpdelta property determines the scrolling unit for a jump (as when the bar is clicked outside the indicator or arrows); default is 20% of the range between minvalue and maxvalue. To add to a window w: `w.add ("scrollbar" [, bounds, value, minvalue, maxvalue]);` *bounds*: Optional. The control's position and size. *value*: Optional. The initial position of the scroll indicator. Default is 0. *minvalue*: Optional. The minimum value that the value property can be set to. Default is 0. Together with *maxvalue*, defines the scrolling range. *maxvalue*: Optional. The maximum value that the value property can be set to. Default is 100. Together with *minvalue*, defines the scrolling range.
slider	Slider	A slider with a moveable position indicator. All slider controls have a horizontal orientation. Calls the onChange callback after the position of the indicator is changed or if its notify() method is called. Calls the onChanging callback repeatedly while the user is moving the indicator. The value property contains the current position of the indicator within the range of minvalue and maxvalue. To add to a window w: `w.add ("slider" [, bounds, value, minvalue, maxvalue]);` *bounds*: Optional. The control's position and size. *value*: Optional. The initial position of the indicator. Default is 0. *minvalue*: Optional. The minimum value that the value property can be set to. Default is 0. Together with *maxvalue*, defines the range. *maxvalue*: Optional. The maximum value that the value property can be set to. Default is 100. Together with *minvalue*, defines the range.

Type name	Class name	Description
statictext	StaticText	A text field that the user cannot change. To add to a window w: `w.add ("statictext" [, bounds, text, {creation_properties}]);` *bounds*: Optional. The control's position and size. *text*: Optional. The text displayed in the control. *creation_properties*: Optional. An object that contains any of the following properties: **multiline**: When false (the default), the control displays a single line of text. When true, the control displays multiple lines, in which case the text wraps within the width of the control. **scrolling**: When false (the default), the displayed text cannot be scrolled. When true, the displayed text can be vertically scrolled using the UP ARROW and DOWN ARROW; this case implies multiline=true.
treeview	TreeView	A hierarchical list whose items can contain child items. Items at any level of the tree can be individually selected. To add to a window w: `w.add ("treeview" [, bounds, items, {creation_properties}])` *bounds*: Optional. The control's position and size. *items*: Optional. An array of strings for the text of each top-level list item. A ListItem object is created for each item. Supply this argument, or the items property in *creation_properties*, not both. *creation_properties*: Optional. An object that contains any of the following properties: **items**: An array of strings for the text of each top-level list item. A ListItem object is created for each item. An item with the text string "-" creates a separator item. Supply this property, or the items argument, not both. This form is most useful for elements defined using Resource Specifications. **Note:** Details of this new feature are not available at time of publication. Check for updated versions of this document at http://partners.adobe.com.

Control object properties

The following table shows the properties of all ScriptUI elements. Some values apply only to controls of particular types, as indicated.

active	Boolean	When `true`, the object is active, `false` otherwise. Set to `true` to make a given control or dialog active. • A modal dialog that is visible is by definition the active dialog. • An active palette is the front-most window. • An active control is the one with focus—that is, the one that accepts keystrokes, or in the case of a `Button`, be selected when the user types a Return.

alignment	String	Applies to child elements of a container. If defined, this value overrides the `alignChildren` setting for the parent container.
		This can be a single string, which indicates the alignment for the `orientation` specified in the parent container, or an array of two strings, indicating both the horizontal and vertical alignment (in that order).
		Allowed values depend on the `orientation` value of the parent container. For `orientation=row`:
		`top`
		`bottom`
		`center (default)`
		`fill`
		For `orientation=column`:
		`left`
		`right`
		`center (default)`
		`fill`
		For `orientation=stack`:
		`top`
		`bottom`
		`left`
		`right`
		`center (default)`
		`fill`
		Values are not case sensitive.

bounds	Bounds	A Bounds object describing the boundaries of the element, in screen coordinates for `window` elements, and parent-relative coordinates for child elements (compare windowBounds). For windows, the bounds refer only to the window's content region. Setting an element's `size` or `location` changes its `bounds` property, and vice-versa.
checked	Boolean	For `ListItem` objects only. When `true`, the item is marked with the platform-appropriate checkmark. When `false`, no checkmark is drawn, but space is reserved for it in the left margin, so that the item lines up with other checkable items. When `undefined`, no space is reserved for a checkmark.

enabled	Boolean	When `true`, the control is enabled, meaning that it accepts input. When `false`, control elements do not accept input, and all types of elements have a grayed-out appearance.
expanded	Boolean	For `ListItem` objects only, applies only to children of `TreeView` list controls. When `true`, the item is in the expanded state and its children are shown, when `false`, it is collapsed and children are hidden.
graphics	Object	A ScriptUIGraphics Object that can be used to customize the control's appearance, in response to the onDraw event.
helpTip	String	A brief help message (also called a *tool tip*) that is displayed in a small floating window when the mouse cursor hovers over a UI control element. Set to an empty string or `null` to remove help text.
icon	String or File	The name of an icon resource or the pathname or File Object for a file that contains a platform-specific icon image in PNG format. • For an `IconButton`, the icon appears as the content of the button. • For a `ListItem`, the icon is displayed to the left of the text. • For an `Image`, the icon is the entire content of the image element.
image	Object	For `ListItem` objects only. A ScriptUIImage Object to use as an display label for the item, drawn to the left of the text label.
indent	Number	A number of pixels by which to indent the element during automatic layout. Applies for `column` orientation and `left` alignment, or `row` orientation and `top` alignment.
index	Number	For `ListItem` objects only. The index of this item in the `items` collection of its parent list control. Read only.
items	Array of Object	For a list object (`listbox`, `dropdown`, or `treeview` list), a collection of `ListItem` objects for the items in the list. Access by 0-based index. To obtain the number of items in the list, use `items.length`. Read only.
itemSize	Dimension	For a list object (`listbox`, `dropdown`, or `treeview` list), a Dimension object describing the width and height in pixels of each item in the list. Used by auto-layout to determine the `preferredSize` of the list, if not otherwise specified. If not set explicitly, the size of each item is set to match the largest height and width among all items in the list
jumpdelta	Number	The amount to increment or decrement a `scrollbar` indicator's position when the user clicks ahead or behind the moveable element. Default is 20% of the range between the `maxvalue` and `minvalue` property values.

justify	String	The justification of text in static text and edit text controls. One of:
		`left (default)`
		`center`
		`right`
		Note: Justification only works if the value is set before the window containing the control is displayed for the first time.

location	Point	A Point object describing the location of the element as an array, `[x, y]`, representing the coordinates of the upper left corner of the element. These are screen coordinates for `window` elements, and parent-relative coordinates for other elements. The `location` is defined as `[bounds.x, bounds.y]`. Setting an element's `location` changes its `bounds` property, and vice-versa. By default, `location` is `undefined`.
maximumSize	Dimension	A Dimension object that specifies the maximum height and width for an element.
minimumSize	Dimension	A Dimension object that specifies the minimum height and width for an element. Default is `[0,0]`.
maxvalue	Number	The maximum value that the `value` property can have. If `maxvalue` is reset less than `value`, `value` is reset to `maxvalue`. If `maxvalue` is reset less than `minvalue`, `minvalue` is reset to `maxvalue`.
minvalue	Number	The minimum value that the `value` property can have. If `minvalue` is reset greater than `value`, `value` is reset to `minvalue`. If `minvalue` is reset greater than `maxvalue`, `maxvalue` is reset to `minvalue`.
parent	Object	The immediate parent object of this element. Read only.
preferredSize	Dimension	A Dimension object used by layout managers to determine the best size for each element. If not explicitly set by a script, value is established by the UI framework in which ScriptUI is employed, and is based on such attributes of the element as its text, font, font size, icon size, and other UI framework-specific attributes. A script can explicitly set `preferredSize` before the layout manager is invoked in order to establish an element size other than the default.
properties	Object	An object that contains one or more creation properties of the element (properties used only when the element is created).
selected	Boolean	For `ListItem` objects only. When `true`, the item is part of the `selection` for its parent list. When `false`, the item is not selected. Set to true to select this item in a single-selection list, or to add it to the to the selection array for a multi-selection list.

selection	ListItem, Array of ListItem	For a list object (`listbox`, `dropdown`, or `treeview` list), the currently selected `ListItem` object for a single-selection list, or an array of `ListItem` objects for current selection in a multi-selection list. Setting this value causes the selected item to be highlighted and to be scrolled into view if necessary. You can set the value using the index of an item or an array of indices, rather than object references. If set to an index value that is out of range, the operation is ignored. When set with index values, the property still returns object references. • If you set the value to an array for a single-selection list, only the first item in the array is selected. • If you set the value to a single item for a multi-selection list, that item is added to the current selection. If no items are selected, the value is `null`. Set to `null` to deselect all items.
shortcutKey	String	The key sequence that invokes the onShortcutKey callback for this element (in Windows only).
size	Dimension	A Dimension object that defines the actual dimensions of an element. Initially `undefined`, and unless explicitly set by a script, it is defined by a `LayoutManager`. A script can explicitly set size before the layout manager is invoked to establish an element size other than the `preferredSize` or the default size. Defined as `[bounds.width, bounds.height]`. Setting an element's `size` changes its `bounds` property, and vice-versa.
spacing	Number	A number of pixels to leave between this element and an adjacent sibling during automatic layout.
stepdelta	Number	The amount by which to increment or decrement a `Scrollbar` element's position when the user clicks a stepper button.
text	String	The title, label, or displayed text. Ignored for containers of type `group`. For controls, the meaning depends on the control type. Buttons use the `text` as a label, for example, while edit fields use the `text` to access the content. This is a localizable string: see 'Localization in ScriptUI Objects' on page 87.
textselection	String	The currently selected text in a control that displays text, or the empty string if there is no text selected. Setting the value replaces the current text selection and modifies the value of the `text` property. If there is no current selection, inserts the new value into the `text` string at the current insertion point. The `textselection` value is reset to an empty string after it modifies the `text` value. **Note:** Setting the `textselection` property before the `edittext` control's parent `Window` exists is an undefined operation.

type	String	Contains the type name of the element, as specified on creation. • For `window` objects, one of the type names `window`, `palette`, or `dialog`. • For controls, the type of the control, as specified in the `add` method that created it. Read only.
value	Boolean	For a checkbox or radiobutton, `true` if the control is in the selected or set state, `false` if it is not.
value	Number	For a scrollbar or slider, the current position of the indicator. If set to a value outside the range specified by `minvalue` and `maxvalue`, it is automatically reset to the closest boundary.
visible	Boolean	When `true`, the element is shown, when `false` it is hidden. When a container is hidden, its children are also hidden, but they retain their own visibility values, and are shown or hidden accordingly when the parent is next shown.
window	Window	The Window Object that contains this control. Read only.
windowBounds	Bounds	A Bounds object that contains the bounds of this control in the containing window's coordinates. Compare bounds, in which coordinates are relative to the immediate parent container. Read only.
function_name	Function	For the Flash player control, a function definition for a callback from the Flash ActionScript environment. There are no special naming requirements, but the function must take and return only the supported data types: Number String Boolean Null undefined Object Array **Note:** The ActionScript `class` and `date` objects are not supported as parameter values.

Control object functions

The following table shows the methods defined for each element type, and for specific control types as indicated.

add()

```
listObj.add (type, text[, index])
```

type	The type of item to add. One of: `item`: A basic, selectable item with a text label. `separator`: A separator. For `dropdownlist` controls only. In this case, the *text* value is ignored, and the method returns `null`.
text	The localizable text label for the item.
index	Optional. The index into the current item list after which this item is inserted. If not supplied, or greater than the current list length, the new item is added at the end.

For list objects (`listbox` , `dropdownlist`, or `treeview`) only. Adds an item to the `items` array at the given index.

Returns the `item` control object for `type=item`, or `null` for `type=separator`.

addEventListener()

```
controlObj.addEventListener (eventName, handler, capturePhase);
```

eventName	The event name string.
handler	The function to register for the specified event in this target. This can be the name of a function defined in the extension, or a locally defined handler function to be executed when the event occurs. A handler function takes one argument, the UIEvent Object. See 'Registering event listeners for windows or controls' on page 72.
capturePhase	Optional. When `true`, the handler is called only in the capturing phase of the event propagation. Default is `false`, meaning that the handler is called in the bubbling phase if this object is an ancestor of the target, or in the at-target phase if this object is itself the target.

Registers an event handler for a particular type of event occuring in this control.

Returns `undefined`.

dispatchEvent()

```
controlObj.dispatchEvent ( eventObj, data )
```

eventObj	An UIEvent Object.
data	Optional. Data to pass to the event handler, of the type appropriate to the type of event object.

Simulates the occurrence of an event in this target. A script can create an UIEvent Object for a specific event and pass it to this method to start the event propagation for the event.

Returns the UIEvent Object.

find()

```
listObj.find(text)
```

text The text of the item to find.

For list objects (`listbox` , `dropdownlist`, or `treeview`) only. Looks in this object's `items` array for an `item` object with the given `text` value.

Returns the `item` object if found; otherwise, returns `null`.

hide()

```
controlObj.hide()
```

Hides this container or control. When a window or container is hidden, its children are also hidden, but when it is shown again, the children retain their own visibility states.

Returns `undefined`.

notify()

```
controlObj.notify([event])
```

event Optional. The name of the control event handler to call. One of:

`onClick`
`onChange`
`onChanging`

By default, simulates the `onChange` event for an `edittext` control, an `onClick` event for controls that support that event.

Sends a notification message, simulating the specified user interaction event.

Returns `undefined`.

remove()

```
containerObj.remove(index)
containerObj.remove(text)
containerObj.remove(child)
```

index
text
child

The item or child to remove, specified by 0-based index, `text` value, or as a `control` object.

For containers (`panel`, `group`), removes the specified child control from the container's `children` array.

For list objects (`listbox`, `dropdown`, or `treeview` list) only, removes the specified item from this object's `items` array. No error results if the item does not exist.

Returns `undefined`.

removeAll()

```
listObj.removeAll()
```

For list objects (`listbox`, `dropdown`, or `treeview` list) only. Removes all items from the object's `items` array.

Returns `undefined`.

removeEventListener()

controlbj.addEventListener (*eventName*, *handler*[, *capturePhase*]);

eventName	The event name string.
handler	The function that was registered to handle the event.
capturePhase	Optional. Whether the handler was to respond only in the capture phase.

Unregisters an event handler for a particular type of event occuring in this control. All arguments must be identical to those that were used to register the event handler.

Returns undefined.

show()

controlObj.show()

Shows this container or control. When a window or container is hidden, its children are also hidden, but when it is shown again, the children retain their own visibility states.

Returns undefined.

toString()

listItemObj.toString()

For item controls only. Retrieves the value of this item's text property as a string.

Returns a String.

valueOf()

listItemObj.valueOf()

For item controls only. Retrieves the index number of this item in the parent list's items array.

Returns a Number.

FlashPlayer control functions

These functions apply only to controls of type flashplayer.

Note: There are limitations on how these functions can be used to control playback of Flash movies:

- Do not use stopMovie() and playMovie() to suspend and subsequently resume or restart an SWF file produced by Flex™.
- The stopMovie() and playMovie() sequence does not make sense for some SWF files produced by Flash Authoring, depending on the exact details of how they were implemented. The sequence may not correctly reset the file to the initial state (when the rewind argument to playMovie() is true) nor suspend then resume the execution of the file (when rewind is false).
- Using stopMovie() from the player's hosting environment has no effect on an SWF file playing in a ScriptUI Flash Player element. It is, however, possible to produce an SWF using Flash Authoring that can stop itself in response to user interaction.
- Do not call playMovie() when an SWF file is already playing.

invokePlayerFunction()

```
flashPlayerObj.invokePlayerFunction(fnName, [arg1[,...argn]] )
```

fnName String. The name of a Flash ActionScript function that has been registered with the `ExternalInterface` object by the currently loaded SWF file; see 'Calling ActionScript functions from an Adobe script' on page 75.

args Optional. One or more arguments to pass through to the function, of these types:

Number
String
Boolean
Null
undefined
Object
Array

Invokes an ActionScript function defined in the Flash application.

Returns the result of the invoked function, which must be one of the allowed types. The ActionScript `class` and `date` objects are not supported as return values.

loadMovie()

```
flashPlayerObj.loadMovie(file)
```

file The File Object for the SWF file.

Loads a movie into the Flash Player, and begins playing it.

Returns `undefined`.

playMovie()

```
flashPlayerObj.playMovie(rewind)
```

rewind When true, restarts the movie from the beginning; otherwise, starts playing from the point where it was stopped.

Restarts a movie that has been stopped.

Note: Do not call when a movie is currently playing.

Returns `undefined`.

stopMovie()

```
flashPlayerObj.stopMovie()
```

Halts playback of the current movie.

Note: Does not work when called from the player's hosting environment.

Returns `undefined`.

Control event-handling callbacks

The following events are signalled in certain types of controls. To handle the event, define a function with the corresponding name in the control object. Handler functions take no arguments and have no expected return values; see 'Defining Behavior with Event Callbacks and Listeners' on page 71.

DrawState Object

A helper object that describes an input state at the time of the triggering onDraw event. Contains properties that report whether the current control has the input focus, and the particular mouse and keypress state. There is no object constructor.

DrawState object properties

The object contains the follo(o)0ainigge-16.1(tor)51.5(.)]TJ□F4 1 Tf□and71 -1.pad-0 0 17lo(o)0aily2 1 1TD□0.5 0□□av(

UIEvent Object

Encapsulates input event information for an event that propagates through a container and control hierarchy.

This object is passed to a function that you register to respond to events of a certain type that occur in a window or control. Use *windowObj*.addEventListener() or *controlObj*.addEventListener() to register a handler function. See 'Registering event listeners for windows or controls' on page 72.

Note: Complete details of this new feature are not available at time of publication. Check for updated versions of this document at http://partners.adobe.com.

UIEvent object constructor

The `UIEvent` object is normally created by ScriptUI and passed to your event handler. However, you can simulate a user action by constructing an event object and sending it to a target object's dispatchEvent() function.

```
new UIEvent (type, canBubble, cancelable, view, detail);
```

type	String	The event type.
canBubble	Boolean	When `true`, the event should be triggered in ancestors of the target object during the bubbling phase.
cancelable	Boolean	When `true`, the event can be cancelled.
view	Object	The container or control object that dispatched the event.
detail	Number	Details of the event, which vary according to the event type. The value is 1 or 2 for the `click` event, indicating a single or double click.

UIEvent object properties

bubbles	Boolean	When `true`, the event supports the bubbling phase.
cancelable	Boolean	When `true`, the handler can call this object's preventDefault() method to cancel the default action of the event.
currentTarget	Object	The element object where the currently executing handler was registered. This could be an ancestor of the target object, if the handler is invoked during the capture or bubbling phase.
detail	Number	Details of the event, which vary according to the event type. The value is 1 or 2 for the `click` event, indicating a single or double click.

eventPhase	Number	Current event propagation phase. One of these constants:
		`Event.NOT_DISPATCHING`
		`Event.CAPTURING_PHASE`
		`Event.AT_TARGET`
		`Event.BUBBLING_PHASE`

target	Object	The element object where the event occurred.
timeStamp	Object	Time the event was initiated. A JavaScript `Date` object.

type	String	The name of the event that occurred. Predefined events types are:
		`change`
		`changing`
		`click (detail = 1)`
		`click (detail = 2)`
		`emterKey`
		`move`
		`moving`
		`resize`
		`resizing`
		`show`
		`focus`
		`blur`

view	Object	The container or control object that dispatched the event.

UIEvent object functions

initEvent()

eventObj.initEvent (*eventName, bubble, isCancelable*)

eventName	The event name string.
bubble	When `true`, the event should be triggered in ancestors of the target object during the bubbling phase.
isCancelable	When `true`, the event can be cancelled.

Reinitializes the object, allowing you to change the event properties after construction.

Returns `undefined`.

initUIEvent()

eventObj.initUIEvent (*eventName, bubble, isCancelable*)

eventName	The event name string.
bubble	When `true`, the event should be triggered in ancestors of the target object during the bubbling phase.
isCancelable	When `true`, the event can be cancelled.
view	The container or control object that dispatched the event.
detail	Details of the event, which vary according to the event type. The value is 1 or 2 for the `click` event, indicating a single or double click.

Modifies an event before it is dispatched to its targets. Takes effect only if `UIEvent.eventPhase` is `Event.NOT_DISPATCHING`. Ignored at all other phases.

Returns `undefined`.

preventDefault()

eventObj.preventDefault ( )

Cancels the default action of this event, if this event is cancelable (that is, cancelable is `true`). For example, the default click action of an OK button is to close the containing dialog; this call prevents that behavior.

Returns `undefined`.

stopPropagation()

```
eventObj.stopPropagation ( )
```

Stops event propagation (bubbling and capturing) after executing the handler or handlers at the current target.

Returns `undefined`.

Graphic Customization Objects

These objects provide the ability to customize the appearance of user-interface controls before they are drawn.

Note: Complete details of this new feature are not available at time of publication. Check for updated versions of this document at http://partners.adobe.com.

ScriptUIGraphics Object

Most types of UI elements have a graphics property which contains an object of this type, which allows you to customize aspects of the element's appearance, such as the color and font. Use an onDraw callback function to set these properties or call the functions.

All measurements are in pixels.

ScriptUIGraphics object properties

The object contains the following properties:

backgroundColor	Array of Number	The background color of a container, or the parent background color for a control element. An array in the form [*R*, *B*, *G*, *A*] specifying the red, green, blue values of the color as numbers in the range [0..255], and the transparency (alpha channel) value as a number in the range [0..100].
BrushType	Object	The brush used to paint color in the element, a ScriptUIBrush Object.
currentPath	Object	The current drawing path for this object, a ScriptUIPath Object.
currentPoint	Object	The current position in the drawing path for this object, a Point object.
disabledBackgroundColor	Array of Number	The background color for the disabled state of a container, or the parent background color for the disabled state of a control element. An array in the form [*R*, *B*, *G*, *A*] specifying the red, green, blue values of the color as numbers in the range [0..255], and the transparency (alpha channel) value as a number in the range [0..100].

disabledForegroundColor	Array of Number	The foreground color for the disabled state of a container, or the parent foreground color for the disabled state of a control element. An array in the form [*R*, *B*, *G*, *A*] specifying the red, green, blue values of the color as numbers in the range [0..255], and the transparency (alpha channel) value as a number in the range [0..100].
font	Object	The font to use in writing text, a ScriptUIFont Object.
foregroundColor	Array of Number	The foreground color for a container, or the parent foreground color of a control element. An array in the form [*R*, *B*, *G*, *A*] specifying the red, green, blue values of the color as numbers in the range [0..255], and the transparency (alpha channel) value as a number in the range [0..100].
PenType	Object	The pen to use in drawing lines, a ScriptUIPen Object.

ScriptUIGraphics object functions

The `Graphics` object can define the following functions to customize the appearance of the control:

closePath()

```
controlObj.graphics.closePath ( )
```

Draws a line from the current postion to the start point of the current path, which closes the current path.

Returns `undefined`.

drawFocusRing()

```
controlObj.graphics.drawFocusRing (left, top, width, height)
```

left, top, width, height	The rectangular area, in the coordinate system of the control that contains this graphics object.

Draws a focus ring within a rectangular area.

Returns `undefined`.

drawImage()

```
controlObj.graphics.drawImage (image, left, top, width, height)
```

image	The ScriptUIImage Object.
left, top, width, height	The rectangular area, in the coordinate system of the control that contains this graphics object.

Draws an image within a given rectangular area, using the image file from the given image object that is appropriate to the control's current state.

Returns `undefined`.

drawOSControl()

```
controlObj.graphics.drawOSControl ( )
```

Draws the platform-specific control associated with this element.

Returns undefined.

drawString()

```
controlObj.graphics.drawString (text, pen, x, y, font)
```

text	The text string.
pen	The ScriptUIPen Object for the drawing pen to use.
x, y	The origin point of the drawn text, in the coordinate system of the control that contains this graphics object.
font	Optional. The ScriptUIFont Object for the font to use. Default is determined by the fontStyle value in the Global ScriptUI Object

Draws a string of text starting at a given point, using the given pen and font.

Returns undefined.

ellipsePath()

```
controlObj.graphics.ellipsePath (left, top, width, height)
```

left, top, width, height	The rectangular area, in the coordinate system of the control that contains this graphics object.

Draws an ellipse within a given rectangular area, using the drawing pen of this object.

Returns a Point object for the upper left corner of the area, which is the new current position.

fillPath()

```
controlObj.graphics.fillPath (brush, path)
```

brush	The ScriptUIBrush Object for the painting brush to use.
path	The ScriptUIPath Object for the path.

Paints a texture into the fill area of a drawing path.

Returns undefined.

lineto()

```
controlObj.graphics.lineto (x, y)
```

x, y	The destination point of the line, in the coordinate system of the control that contains this graphics object.

Draws a line between the current position and a given point, using the drawing pen of this object.

Returns a Point object for the given destination point, which is the new current position.

measureString()

```
controlObj.graphics.measureString (text, font, boundingWidth)
```

text The text string.

font Optional. The ScriptUIFont Object for the font to use. Default is determined by the fontStyle value in the Global ScriptUI Object

boundingWidth A number, the bounding width.

Calculates the size needed to draw a text string in a given font.

Returns a Dimension object containing the height and width of the string in pixels.

moveto()

```
controlObj.graphics.moveto (x, y)
```

x, y The new coordinates, in the coordinate system of the control that contains this graphics object.

Sets the current position of this element.

Returns a Point object for the given destination point, which is the new current position.

newBrush()

```
controlObj.graphics.newBrush( type, color );
```

type The brush type, one of these constants:

```
SOLID_COLOR
THEME_COLOR
```

color The brush color.

- If type is `SOLID_COLOR`, the color expressed as a number If the type is `THEME_COLOR`, the name string of the theme.

Creates a new painting brush.

Returns a ScriptUIBrush Object.

newPath()

```
controlObj.graphics.newPath( );
```

Creates a new drawing path.

Note: Details not available at time of publication. Check for updated versions of this document at http://partners.adobe.com.

Returns a ScriptUIPath Object.

ScriptUIBrush Object

ScriptUIFont Object

A helper object that encapsulates the qualities of a font used to draw text into a control. Create with the newFont() method of the Global ScriptUI Object.

ScriptUIFont object properties

The object contains the following properties:

family	String	The font family name.
name	String	The font name.
size	Number	The font point size.
style		

ScriptUIImage Object

A helper object that encapsulates a set of images that can be drawn into a control. Create with the newImage() method of the Global ScriptUI Object.

Note: Details not available at time of publication. Check for updated versions of this document at http://partners.adobe.com.

ScriptUIImage object properties

The object contains the following properties:

ScriptUIPath Object

A helper object that encapsulates a drawing path for a figure to be drawn into a control. Create with the newPath() method of the ScriptUIGraphics Object.

Note: Details not available at time of publication. Check for updated versions of this document at http://partners.adobe.com.

ScriptUIPen Object

A helper object that encapsulates the qualities of a pen used to draw lines into a control. Create with the newPen() method of the ScriptUIGraphics Object.

ScriptUIPen object properties

The object contains the following properties:

color	Array of Number	The paint color to use when the type is SOLID_COLOR. An array in the form [*R*, *B*, *G*, *A*] specifying the red, green, blue values of the color as numbers in the range [0..255], and the transparency (alpha channel) value as a number in the range [0..100].
lineWidth	Number	The pixel width of the drawing line.
theme	String	The name of a color theme to use for drawing when the type is THEME_COLOR.
type	Number	The pen type, one of these constants: SOLID_COLOR THEME_COLOR

LayoutManager Object

Controls the automatic layout behavior for a window or container. The subclass `AutoLayoutManager` implements the default automatic layout behavior.

AutoLayoutManager object constructor

Create an instance of the `AutoLayoutManager` class with the `new` operator:

```
myWin.layout = new AutoLayoutManager(myWin);
```

An instance is automatically created when you create a `Window` or container (`group` or `panel`) object, and referenced by the container's layout property. This instance implements the default layout behavior unless you override it.

AutoLayoutManager object properties

The default object has no predefined properties, but a script can assign arbitrary properties to an object it creates, to store data needed by the script-defined layout algorithm.

AutoLayoutManager object functions

`layout()`

`windowObj``.layout.layout (`*`recalculate`*`)`

`recalculate` Optional. When `true`, forces the layout manager to recalculate the container size for this and any child containers. Default is `false`.

Invokes the automatic layout behavior for the managed container. Adjusts sizes and positions of the child elements of this window or container according to the placement and alignment property values in the parent and children.

Invoked automatically the first time the window is displayed. Thereafter, the script must invoke it explicitly to change the layout in case of changes in the size or position of the parent or children.

Returns `undefined`

`resize()`

`windowObj``.layout.resize (`*`alignment`*`)`

`alignment` See 'Aligning children' on page 78.

Resizes the child elements of the managed container with a given alignment type, after the window has been resized by the user.

Returns `undefined`.

5 Interapplication Communication with Scripts

The Adobe scripting environment provides an interapplication message framework, a way for to send and receive information and scripts from one Adobe application to another. An application that supports the messaging framework is said to be *message enabled*.

Code samples that demonstrate various techniques are provided with the Adobe Bridge SDK, and referenced by name in the relevant sections.

Communications Overview

Scripts written for any message-enabled application can communicate with other message-enabled applications in two ways; through directly calling functions defined in a remote application, and by sending messages and receiving responses from a remote application. A specific syntax is provided for identifying applications unambiguously.

Remote function calls

A limited set of basic functions (the *cross-DOM*) are common across all message-enabled applications, and allow your script to, for example, open or print files in other applications, simply by calling the `open` or `print` function for that application.

- 'Cross-DOM Functions' on page 134 describes the usage of this feature.
- 'Cross-DOM API reference' on page 135 provides reference details for the functions of the basic cross-DOM.

Each message-enabled application can also export a set of functions to provide a selected set of application-specific functionality; see 'Application-specific exported functions' on page 134. For example, an Adobe Bridge script can request a photo merge in Photoshop by calling `photoshop.photomerge(files)`. The set of functions available for each application varies widely.

Message framework

The interapplication message framework is a JavaScript application programming interface (API) that allows extensive control over communication between applications. The API allows you to send messages to other applications and receive results, and to receive messages sent by other applications and return results. Typically the data passed between applications are JavaScript scripts. However, the messaging framework is extensible. It allows you to define different types of data to send between applications, and to specify how they are handled.

- 'Communicating Through Messages' on page 137 describes the usage of this feature.
- 'Message Framework API Reference' on page 146 provides complete reference details.

Identifying applications

When calling external functions or exchanging messages, you must identify particular applications using *namespace specifiers*. A specifier consists of a specific name string (such as `photoshop`), and optional

additions that identify a particular release or locale version. Application specifiers are used occasionally in other contexts as well. For details of the syntax, see 'Application and Namespace Specifiers' on page 157.

Regardless of which method you use to perform interapplication communication, you must place your script in a location where the application you want to run it can see it. There are different locations for the startup scripts of the applications themselves, and for scripts provided by developers.

Because all JavaScript-enabled applications look in the same locations for scripts to run, the scripts themselves must be explicit about which application they are meant for. A script should check that all applications it needs to communicate with are installed with the correct version, and that any other applications that might be installed do not run the script. For details, see 'Scripting for Specific Applications' on page 10.

Cross-DOM Functions

The cross-DOM is a small application programming interface (API), which provides a set of functions that are common across message-enabled applications. These include functions to open files, execute scripts, and print files. For details of the function set, see the 'Cross-DOM API reference' on page 135.

You can access cross-DOM functions in any script by prefixing the function name with the *namespace specifier* for the target application (see 'Namespace specifiers' on page 159). For example, a Photoshop CS3 script can call `indesign.open(`*`file`*`)` to open a file in Adobe InDesign® CS3.

The cross-DOM functions for each application are implemented in JavaScript. You can see the implementation for each installed application by reading its associated startup script in the Adobe startup folder. For example, Adobe Illustrator® CS3 defines `illustrator.open()` in the `illustrator-13.jsx` startup script (*13* is the version number of the installed application). See 'Startup folder locations' on page 135.

➤ Example code

The sample code distributed with the Adobe Bridge SDK includes these code examples that specifically demonstrate the use of cross-DOM functions:

Cross-DOM calls	
`SnpOpenInPhotoshop.jsx`	Shows how to send a file selected in Adobe Bridge to be opened in Photoshop.

Application-specific exported functions

In addition to the required base cross-DOM functions, each message-enabled application can provide application-specific functionality to all scripts through a simple syntax. You can access exported functions in any script by prefixing the function name with the namespace specifier for the target application (see 'Namespace specifiers' on page 159). For example, Photoshop CS3 exports the `photomerge` function, so an Illustrator CS3 script can directly call `photoshop.photomerge(`*`files`*`)`.

The only difference between cross-DOM functions and the application-specific exported functions is that all applications expose the same set of cross-DOM functions, whereas each application exposes its own set of application-specific functions. Each application determines the extent of its exported functionality. Some applications provide extensive support for exported functions, others less.

For details of additional functions that are exported by individual applications, refer to the startup scripts for those applications. The application startup scripts are named *appname-n*.jsx, where *n* is the version number of the installed application. See 'Startup folder locations' on page 135.

Startup folder locations

For each platform, there is a startup folder shared by all Adobe Creative Suite 3 applications that support JavaScript, and an application-specific startup folder.

- In Windows®, the installation startup folders are:

```
%CommonProgramFiles%\Adobe\Startup Scripts CS3\
%CommonProgramFiles%\Adobe\Startup Scripts CS3\Adobe AppName\
```

- In Mac OS®, the installation startup folders are:

```
/Library/Application Support/Adobe/StartupScripts CS3/
/Library/Application Support/Adobe/StartupScripts CS3/Adobe AppName/
```

Note: This is *not* the location in which to store your own startup scripts; see 'Scripting for Specific Applications' on page 10.

Cross-DOM API reference

All exported functions, including those of the cross-DOM API, are invoked through the exporting application, identified by its *namespace specifier* (see 'Namespace specifiers' on page 159). For example:

```
//execute an Illustrator script in version 12
illustrator12.executeScript(myAIScript);
```

A specifier with no version information invokes the highest installed version of the application. For example:

```
//execute a Photoshop script in the highest available version
photoshop.executeScript (myPSScript);
```

All message-enabled applications implement the following cross-DOM functions:

executeScript()

appspec.executeScript(*script*)

script A string containing the script to be evaluated.

Performs a JavaScript eval on the specified script. The entire document object model (DOM) of the target application is available to the script. Returns undefined.

open()

appspec.open(*files*)

files A File Object or array of File objects. For applications that use compound documents, this should be a project file.

Performs the equivalent of the target application's **File > Open** command on the specified files. Returns undefined.

openAsNew()

appspec.openAsNew([*options*])

options Optional. Application-specific creation options:

Bridge: none
Photoshop: none
InDesign: creation options are:

(Boolean:*showingWindow*, ObjectOrString:*documentPresets*)

See the arguments for documents.add() in the *InDesign CS3 Scripting Reference*.

Illustrator: creation options are:

([DocumentColorSpace:*colorspace*][, Number:*width*, Number:*height*])

See the arguments for documents.add() in the *Illustrator CS3 JavaScript Reference*.

Performs the equivalent of the target application's **File > New** command. Returns true on success.

print()

appspec.print(*files*)

files A File Object or array of File objects. For applications that use compound documents, this should be a project file.

Performs the equivalent of the target application's **File > Print** command on the specified files. Returns undefined.

quit()

appspec.quit()

Performs the equivalent of the target application's **File > Exit** or **File > Close** command. Returns undefined.

Note: This function is available for Adobe Acrobat®, but does nothing. Scripts cannot terminate the application.

reveal()

appspec.reveal(*file*)

file A File Object or string specifying a file that can be opened in the target application.

Gives the target application the operating-system focus, and, if the specified file is open in that application, brings it to the foreground. Returns undefined.

Communicating Through Messages

Adobe Bridge provides an application programming interface (API) that defines a communication protocol between Adobe ExtendScript- and message-enabled applications. This provides the most general mechanism for communication between applications. A messaging-enabled application can launch another messaging-enabled application, and send or receive scripts to effect certain actions. For example, from within Adobe Bridge, a script can launch Photoshop, and then send a script to Photoshop that requests a photomerge operation.

While the exported functions allow specific access to certain capabilities of the application, the script in an interapplication message allows full access to the target application's document object model (DOM), in addition to all cross-DOM and application exported functions.

The messaging API defines the BridgeTalk class, whose globally-available static properties and functions provide access to environmental information relevant for communication between applications. You can instantiate this class to create a BridgeTalk message object, which encapsulates a message and allows you to send it to another application. For details of these objects, see 'Message Framework API Reference' on page 146.

Sending messages

To send a script or other data to another application, you must create and configure a BridgeTalk message object. This object contains the data to be sent (generally a script to be executed in the target application), and also specifies how to handle the response.

This simple example walks through the steps of sending a script from Adobe Bridge CS3 to Photoshop CS3, and receiving a response.

Step 1: Check that the target application is installed

Before you can actually send a message, you must check that the required version of the target application is installed. The function getSpecifier(), available in the global namespace through the BridgeTalk class, provides this information.

For example, this code, which will send a message to Adobe Bridge CS3 as part of a script being executed by Photoshop CS3, checks that the required version of Adobe Bridge is installed:

```
var targetApp = BridgeTalk.getSpecifier( "bridge", "2");
if( targetApp ) {
   // construct and send message
}
```

When you send the message, the messaging framework automatically launches the target application, if it is not already running.

Step 2: Construct a message object

The next step is to construct a message to send

```
bt.target  = "bridge-2.0"; // send this msg to the Adobe Bridge application
// the script to evaluate is contained in a string in the "body" property
bt.body = "new Document('C:\\BridgeScripts');
           app.document.target.children.length;"
```

Step 3: Specify how to handle a response

If you want to handle a response for this message, or use the data that is returned from the script's evaluation, you must set up the response-handling mechanism before you send the message. You do this by defining the onResult callback in the message object.

Note: The message callbacks are optional, and are not implemented by all message-enabled applications.

The response to a message is, by default, the result of evaluation of the script contained in that message's `body` property. The target application might define some different kind of response; see 'Receiving messages' on page 139.

When the target has finished processing this message, it looks for an `onResult` callback in the message object it received. If it is found, the target automatically invokes it, passing it the response. The response is packaged into a string, which is in turn packaged into the `body` property of a new message object. That message object is the argument to your `onResult` callback function.

This handler, for example, processes the returned result using a script-defined `processResult` function.

```
bt.onResult = function(returnBtObj)
   { processResult(returnBtObj.body); }
```

If you want to handle errors that might arise during script processing, you can define an onError callback in the message object. Similarly, you can define a timeout value and onTimeout callback to handle the case where the target cannot process the message within a given time. For more information, see 'Handling responses from the message target' on page 140.

Step 4: Send the message

To send the message, call the message object's `send` method. You do not need to specify where to send the message to, since the target application is set in the message itself.

```
bt.send(0);
```

You can optionally specify a timeout value, which makes the call synchronous; when you do this, the method waits for a response from the target application, or for the timeout value to expire, before returning. When a timeout is not specified, or is 0 as in this example, the call is asynchronous and the `send()` method returns immediately.

A second optional parameter allows you to specify launch parameters, in case the target application is not currently running, and the messaging framework needs to launch it.

The complete script looks like this:

```
// script to be executed in Photoshop CS3
#target "photoshop-10.0"
// check that the target app is installed
var targetApp = BridgeTalk.getSpecifier( "bridge", "2");
if( targetApp ) {
   // construct a message object
   var bt = new BridgeTalk;
   // the message is intended for Adobe Bridge
   bt.target  = "bridge";
```

```
    // the script to evaluate is contained in a string in the "body" property
    bt.body = "new Document('C:\\BridgeScripts');
              app.document.target.children.length;"
    // define result handler callback
    bt.onResult = function(returnBtObj) {
       processResult(returnBtObj.body); } //fn defined elsewhere
    // send the message
    bt.send(0);
}
```

Receiving messages

An application can be the target of a message; that is, it receives an unsolicited message from another application. An unsolicited message is handled by the static `BridgeTalk.onReceive` callback function in the target application. See 'Handling unsolicited messages' on page 139.

An application that sends a message can receive *response messages*; that is, messages that come as the result of requesting a response when a message was sent. These can be:

- The result of an error in processing the message
- The result of a timeout when attempting to process the message
- A notification of receipt of the message
- Intermediate responses
- The final result of processing the message.

All of these response messages are sent automatically by the target application, and are handled by callbacks defined in the sending message object. For details, see 'Handling responses from the message target' on page 140.

Handling unsolicited messages

To specify how the application should handle unsolicited incoming messages, define a callback handler function in the static onReceive property of the `BridgeTalk` class. This function takes a single argument, a BridgeTalk message object.

The default behavior of the `onReceive` handler is to evaluate the `body` of the received message with JavaScript, and return the result of that evaluation. (The result of evaluating a script is the result of the last line of the script.) To return the result, it creates a new message object, encapsulates the result in a string in the `body` property of that object, and passes that object to the onResult callback defined in the original message.

If an error occurs on evaluation, the default `onReceive` handler returns the error information using a similar mechanism. It creates a new message object, encapsulates the error information in a string in the `body` property of that object, and passes that object to the onError callback defined in the original message.

To change the default behavior set the `BridgeTalk.onReceive` property to a function definition in the following form:

```
BridgeTalk.onReceive = function( bridgeTalkObject ) {
   // callback definition here
   };
```

- The `body` property of the received message object contains the received data.
- The function can return any type.

The function that you define does not need to explicitly create and return a `BridgeTalk` message object. The messaging framework creates a new `BridgeTalk` message object, and packages the return value of the `onReceive` handler as a string in the `body` property of that object.

Return values are flattened into a string using the Unicode Transformation Format-8 (UTF-8) encoding. If the function does not specify a return value, the resulting string is the empty string.

The result object is transmitted back to the sender if the sender has implemented an `onResult` callback for the original message.

➤ Message handling examples

This example shows the default mechanism for handling unsolicited messages received from other applications. This simple handler executes the message's data as a script and returns the results of that execution.

```
BridgeTalk.onReceive = function (message) {
   return eval( message.body );
}
```

This example shows how you might extend the receive handler to process a new type of message.

```
BridgeTalk.onReceive = function (message) {
   switch (message.type) {
      case "Data":
         return processData( message );
         break;
      default:  //"ExtendScript"
         return eval( mesage.body );
      }
   }
```

Handling responses from the message target

To handle responses to a message you have sent, you define callback handler functions in the message object itself. The target application cannot send a response message back to the sender unless the message object it received has the appropriate callback defined.

Note: The message callbacks are optional, and are not implemented by all message-enabled applications.

When your message is received by its target, the target application's static `BridgeTalk` object's onReceive method processes that message, and can invoke one of the message object's callbacks to return a response. In each case, the messaging framework packages the response in a new message object, whose target application is the sender. Your callback functions receive this response message object as an argument.

A response message can be:

- The result of an error in processing the message. This is handled by the onError callback.

  If an error occurs in processing the message `body` (as the result of a JavaScript syntax error, for instance), the target application invokes the `onError` callback, passing a response message that contains the error code and error message. If you do not have an `onError` callback defined, the error is completely transparent. It can appear that the message has not been processed, since no result is ever returned to the `onResult` callback.

- A notification of receipt of the message. This is handled by the onReceived callback.

  Message sending is asynchronous. Getting a `true` result from the send method does not guarantee that your message was actually received by the target application. If you want to be notified of the receipt of your message, define the onReceived callback in the message object. The target sends back the original message object to this callback, first replacing the `body` value with an empty string.

- The result of a timeout. This is handled by the onTimeout callback.

  You can specify a number of seconds in a message object's timeout property. If the message is not removed from the input queue for processing before the time elapses, it is discarded. If the sender has defined an onTimeout callback for the message, the target application sends a timeout message back to the sender.

- Intermediate responses. These are handled by the onResult callback.

  The script that you send can send back intermediate responses by invoking the original message object's sendResult() method. It can send data of any type, but that data is packaged into a `body` string in a new message object, which is passed to your callback. See 'Passing values between applications' on page 143.

- The final result of processing the message. This is handled by the onResult callback.

  When it finishes processing your message, the target application can send back a result of any type. If you have sent a script, and the target application is using the default `BridgeTalk.onReceive` callback to process messages, the return value is the final result of evaluating that script. In any case, the return value is packaged into a `body` string in a new message object, which is passed to your callback. See 'Passing values between applications' on page 143.

The following examples demonstrate how to handle simple responses and multiple responses, and how to integrate error handling with response handling.

➤ Example: Receiving a simple response

In this example, an application script asks Adobe Bridge to find out how many files and folders are in a certain folder, which the evaluation of the script returns. (The default `BridgeTalk.onReceive` method processes this correctly.)

The `onResult` method saves that number in `fileCountResult`, a script-defined property of the message, for later use.

```
var bt = new BridgeTalk;
bt.target = "bridge-2.0";
bt.body = "new Document('C:\\BridgeScripts');
            app.document.target.children.length;"
bt.onResult = function( retObj ) {
   processFileCount(retObj.body);
}
bt.send();
```

➤ Example: Handling any error

In this example, the `onError` handler re-throws the error message within the sending application.

```
var bt = new BridgeTalk;
bt.onError = function (btObj) {
   var errorCode = parseInt (btObj.headers ["Error-Code"]);
   throw new Error (errorCode, btObj.body);
}
```

➤ Example: Handling expected errors and responses

This example creates a message that asks Adobe Bridge to return XMP metadata for a specific file. The `onResult` method processes the data using a script-defined `processFileSize` function. Any errors are handled by the `onError` method. For example, if the file requested is not an existing file, the resulting error is returned to the `onError` method.

```
var bt = new BridgeTalk;
bt.target = "bridge-2.0";
bt.body = "var tn = new Thumbnail('C/MyPhotos/temp.gif');
           tn.core.immediate.size;"
bt.onResult = function( resultMsg ) {
   processFileSize(resultMsg.body);
}
bt.onError = function( errorMsg ) {
   var errCode = parseInt (errorMsg.headers ["Error-Code"]);
   throw new Error (errCode, errorMsg.body);
}
bt.send();
```

➤ Example: Setting up a target to send multiple responses

This example integrates the sending of multiple responses with the evaluation of a message body. It sets up a handler for a message such as the one sent in the following example.

The target application (Adobe Bridge) defines a static onReceive method to allow for a new type of message, which it calls an *iterator*. An `iterator` type of message expects the `message.body` to use the iteration variable `i` within the script, so that different results are produced for each pass through the `while` loop. Each result is sent back to the sending application with the sendResult() method. When the `message.body` has finished processing its task, it sets a flag to end the `while` loop.

```
// Code for processing the message and sending intermediate responses
// in the target application (Adobe Bridge)
   BridgeTalk.onReceive = function (message){
      switch (message.type) {
         case "iterator":
            done = false;
            i = 0;
            while (!done) {
               // the message.body uses "i" to produce different results
               // for each execution of the message.
               // when done, the message.body sets "done" to true
               // so this onReceive method breaks out of the loop.
                message.sendResult(eval(message.body));
               i++; }
            break;
         default:  //"ExtendScript"
            return eval( message.body );
      }
   }
```

➤ Example: Setting up a sender to receive multiple responses

This example sends a message of the type `iterator`, to be handled by the onReceive handler in the previous example, and processes the responses received from that target.

The sending application creates a message whose script (contained in the `body` string) iterates through all files in a specific folder (represented by an Adobe Bridge `Thumbnail` object), using the iterator variable `i`. For each file in the folder, it returns file size data. For each contained folder, it returns -1. The last executed line in the script is the final result value for the message.

The onResult method of the message object receives each intermediate result, stores it into an array, `resArr`, and processes it immediately using a script-defined function `processInterResult`.

```
// Code for send message and handling response
// in the sending application (any message-enabled application)
   var idx = 0;
   var resArr = new Array;

   bt = new BridgeTalk;
   bt.target = "bridge";
   bt.type = "iterator";

   bt.body = "
      var fld = new Thumbnail(Folder('C/Junk'));
      if (i == (fld.children.length - 1))
         done = true; //no more files, end loop
      tn = fld.children[i];
      if (tn.spec.constructor.name == 'File')
         md = tn.core.immediate.size;
      else md = -1;
      ";
   // store intermediate results
   bt.onResult = function(rObj) {
      resArr[idx] = rObj.body;
      processInterResult(resArr[idx]);
      idx++;};

   bt.onError = function(eObj) {
      bt.error = eObj.body };

   bt.send();
```

Passing values between applications

The `BridgeTalk.` onReceive static callback function can return values of any type. The messaging framework, however, packages the response into a response message, and passes any returned values in the message `body`, first converting the result to a UTF-8-encoded string.

Passing simple types

When your message object's onResult callback receives a response, it must interpret the string it finds in the `body` of the response message to obtain a result of the correct type. Results of various types can be identified and processed as follows:

Number	JavaScript allows you to access a string that contains a number directly as a number, without doing any type conversion. However, be careful when using the plus operator (+), which works with either strings or numbers. If one of the operands is a string, both operands are converted to strings and concatenated.
String	No conversion is required.
Boolean	The result string is either "true" or "false". You can convert it to a true boolean by evaluating it with the `eval` method.
Date	The result string contains the date in the form: `"dow mmm dd yyyy hh:mm:ss GMT-nnnn".` For example "Wed Jun 23 2004 00:00:00 GMT-0700".
Array	The result string contains a comma delimited list of the elements of the array. For example, If the result array is `[12, "test", 432]`, the message framework flattens this into the string `"12,test,432"`. As an alternative to simply returning the array, the message target can use the `toSource` method to return the code used to create the array. In this case, the sender must reconstitute the array by using the `eval` method on the result string in the response body. See discussion below.

Passing complex types

When returning complex types (arrays and objects), the script that you send must construct a result string, using the `toSource` method to serialize the array or object. In this case, the sender must reconstitute the array or object by using the `eval` method on the result string in the response body.

➤ Passing an array with toSource and eval

For example, the following code sends a script that returns an array in this way. The `onResult` callback that receives the response uses `eval` to reconstruct the array.

```
var bt = new BridgeTalk;
bt.target = "bridge-2.0";
// the script passed to the target application
// needs to return the array using "toSource"
bt.body = "var arr = [10, "this string", 324];
           arr.toSource();"
bt.onResult = function(resObj) {
   // use eval to reconstruct the array
   arr = eval(resObj.body);
   // now you can access the returned array
   for (i=0; i< arr.length(); i++)
   doSomething(arr[i]);
}
// send the message
bt.send();
```

➤ Passing an object with toSource and eval

This technique is the only way to pass objects between applications. For example, this code sends a script that returns an object containing some of the metadata for a specific file, and defines an `onResult` callback that receives the object.

```
var bt = new BridgeTalk;
bt.target = "bridge-2.0";

//the script passed to the target application
// returns the object using "toSource"
bt.body = "var tn = new Thumbnail(File('C:\\Myphotos\\photo1.jpg'));
           var md = {fname:tn.core.immediate.name,
              fsize:tn.core.immediate.size};
           md.toSource();"
//For the result, use eval to reconstruct the object
bt.onResult = function(resObj) {
    md = bt.result = eval(resObj.body);
   // now you can access fname and fsize properties
   doSomething (md.fname, md.fsize);
}
// send the message
bt.send();
```

➤ Passing a DOM object

You can send a script that returns a DOM object, but the resulting object contains only those properties that were accessed within the script. For example, the following script requests the return of the Adobe Bridge DOM `Thumbnail` object. Only the properties `path` and `uri` are accessed by the script, and only those properties are returned:

```
var bt = new BridgeTalk;
bt.target = "bridge";
//set up the script passed to the target application
// to return the array using "toSource"
bt.body = "var tn = new Thumbnail(File('C:\\Myphotos\\photo1.jpg'));
   var p = tn.path; var u = tn.uri;
   tn.toSource();"
//For the result, use eval to reconstruct the object
bt.onResult = function(resObj) {
   // use eval to reconstruct the object
   tn = eval(resObj.body);
   // now the script can access tn.path and tn.uri,
   // but no other properties of the Adobe Bridge DOM Thumbnail object
   doSomething (tn.path, tn.uri);
}
// send the message
bt.send();
```

Message Framework API Reference

This application programming interface (API) defines a communication protocol between message-enabled applications. These objects are available to all ExtendScript scripts when any of the applications is loaded.

The messaging protocol is extensible. Although it is primarily designed to send scripts, you can use it to send other kinds of data.

The messaging API defines the `BridgeTalk` class. Static properties and methods of the class provide access to environmental information relevant for communication between applications. Instantiate the class to create a `BridgeTalk` message object, which encapsulates the message itself. For discussion and examples, see 'Communicating Through Messages' on page 137, and the example code provided with the Adobe Bridge SDK.

➤ Example code

The sample code distributed with the Adobe Bridge SDK includes these code examples that specifically demonstrate the use of interapplication messaging:

Interapplication messaging	
`SnpSendMessage.jsx`	Shows how to send message from Adobe Bridge to Photoshop and receive a response.
`SnpSendArray.jsx`	Sends message from Adobe Bridge to Photoshop that creates an array in the target and passes it back to the sender.
`SnpSendCustomObject.jsx`	Sends message from Adobe Bridge to Photoshop that creates a JavaScript object in the target and passes it back to the sender.
`SnpSendDOMObject.jsx`	Sends message from Adobe Bridge to Photoshop that creates a Photoshop object in the target and passes values from it back to the sender.

BridgeTalk class

Static properties and methods of this class provide a way for your script to determine basic messaging system information before you create any specific message objects. Static methods allow you to check if an application is installed and is already running, and to launch the application. A callback defined on the class determines how the application processes incoming messages.

You can access static properties and methods in the `BridgeTalk` class, which is available in the global namespace. For example:

```
var thisApp = BridgeTalk.appName;
```

Note: You must instantiate the `BridgeTalk` class to create the `BridgeTalk` message object, which is used to send message packets between applications. Dynamic properties and methods can be accessed only in instances.

BridgeTalk class properties

The `BridgeTalk` class provides these static properties, which are available in the global namespace:

`appInstance`	String	The instance identifier of an application launched by the messaging framework, the *instance* portion of an application specifier; see 'Application specifiers' on page 157. Read only. Used only for those applications, such as InDesign, that support launching and running multiple instances. The first instance to be launched in a session is assigned the number 0, and instance numbers are incremented for additional launches throughout the session.
`appLocale`	String	The locale of this application, the *locale* portion of an application specifier; see 'Application specifiers' on page 157. When a message is sent, this is the locale of the sending application. Read only.
`appName`	String	The name of this application, the *appname* portion of an application specifier; see 'Application specifiers' on page 157. When a message is sent, this is the name of the sending application. Read only.
`appSpecifier`	String	A lower-case string containing the complete specifier for this application; see 'Application specifiers' on page 157. Read/write.
`appStatus`	String	The current processing status of this application. Read only. One of: `busy`: The application is currently busy, but not processing messages. This is the case, for example, when a modal dialog is shown. `idle`: The application is currently idle, but processes messages regularly. `not installed`: The application is not installed.
`appVersion`	String	The version number of this application, the *version* portion of an application specifier; see 'Application specifiers' on page 157. When a message is sent, this is the version of the sending application. Read only.

`onReceive`	Function	A callback function that this application applies to unsolicited incoming messages. The default function evaluates the body of the received message and returns the result of evaluation. To change the default behavior, set this to a function definition in the following form:
		`BridgeTalk.onReceive = function( bridgeTalkObject ) {`
		`  // act on received message`
		`};`
		The `body` property of the received message object contains the received data. The function can return any type. See 'Handling unsolicited messages' on page 139.
		Note: This function is *not* applied to a message that is received in response to a message sent from this application. Response messages are processed by the `onResult`, `onReceived`, or `onError` callbacks associated with the sent message.

BridgeTalk class functions

The `BridgeTalk` class provides these static methods, which are available in the global namespace:

bringToFront()

```
BridgeTalk.bringToFront (app)
```

app A specifier for the target application; see 'Application specifiers' on page 157.

Brings all windows of the specified application to the front of the screen.

In Mac OS, an application can be running but have no windows open. In this case, calling this function might or might not open a new window, depending on the application. For Adobe Bridge, it opens a new browser window.

getAppPath()

```
BridgeTalk.getAppPath (app)
```

app A specifier for the target application; see 'Application specifiers' on page 157.

Retrieves the full path of the executable file for a specified application.

Returns a string.

getDisplayName()

```
BridgeTalk.getSpecifier (app)
```

app A specifier for the target application; see 'Application specifiers' on page 157.

Returns a localized display name for an application, or `NULL` if the application is not installed. For example:

```
BridgeTalk.getDisplayName("photoshop-10.0");
=> Adobe Photoshop CS3
```

getSpecifier()

```
BridgeTalk.getSpecifier (appName,[version],[locale])
```

appName The base name of the application to search for.

version Optional. The specific version number to search for. If `0` or not supplied, returns the most recent version. If negative, returns the highest version up to and including the absolute value.

If a major version is specified, returns the highest minor-version variation. For example, if Photoshop CS versions 9, 9.1, and 10 are installed:

```
BridgeTalk.Specifier( "photoshop", "9" )
  => ["photoshop-9.1"]
```

locale Optional. The specific locale to search for.

If not supplied and multiple language versions are installed, prefers the version for the current locale.

Retrieves a complete application specifier.

Returns a complete specifier (see 'Application specifiers' on page 157) for a messaging-enabled application version installed on this computer, or `null` if the requested version of the application is not installed.

For example, assuming installed applications include Photoshop CS3 10.0 en_us, Photoshop CS2 8.5 de_de, Photoshop CS2 9.0 de_de, and Photoshop CS2 9.5 de_de, and that the current locale is en_US:

```
BridgeTalk.getSpecifier ("photoshop");
  => ["photoshop-10.0-en_us"]
BridgeTalk.getSpecifier ("photoshop", 0, "en_us");
  => ["photoshop-10.0-en_us"]
BridgeTalk.getSpecifier ("photoshop", 0, "de_de");
  => ["photoshop-9.5-de_de"]
BridgeTalk.getSpecifier ("photoshop", -9.2, "de_de");
  => ["photoshop-9.0-de_de"]
BridgeTalk.getSpecifier ("photoshop", 8);
  => ["photoshop-8.5-de_de"]
```

getStatus()

```
BridgeTalk.getStatus (targetSpec)
```

targetSpec Optional, a specifier for the target application; see 'Application specifiers' on page 157.

If not supplied, returns the processing status of the current application.

Retrieves the processing status of an application. Returns a string, one of:

BUSY: The application is currently busy, but not processing messages. This is the case, for example, when a modal dialog is shown.

IDLE: The application is currently idle, but processes messages regularly.

PUMPING: The application is currently processing messages.

ISNOTRUNNING: The application is installed but not running.

ISNOTINSTALLED: The application is not installed.

UNDEFINED: The application is running but not responding to ping requests. This can be true of a CS2 application that uses an earlier version of the messaging framework.

getTargets()

```
BridgeTalk.getTargets ([version],[locale])
```

version Optional. The specific version number to search for, or `null` to return the most appropriate version (matching, most recent, or running), with version information.

- Specify only a major version number to return the highest minor-version variation. For example, if Photoshop CS versions 9, 9.5, and 10 are installed:

```
BridgeTalk.getTargets( "9" )
    => [photoshop-9.5]
```

- Specify a negative value to return all versions up to the absolute value of the version number. For example:

```
BridgeTalk.getTargets( "-9.9" )
    => [photoshop-9.0, photoshop-9.5]
```

locale Optional. The specific locale to search for, or `null` to return applications for all locales, with locale information.

If not supplied when *version* is supplied, returns specifiers with version information only.

Retrieves a list of messaging-enabled applications installed on this computer.

Returns an array of 'Application specifiers' on page 157.

- If `version` is supplied, specifiers include the base name plus the version information.
- If `locale` is supplied, specifiers include the full name, with both version and locale information.
- If neither `version` nor `locale` is supplied, returns base specifiers with neither version nor locale information, but tries to find the most appropriate version and locale; see 'Application specifiers' on page 157.

For example, assuming installed applications include Photoshop CS2 9.0 `en_US`, Photoshop CS3 10.0 `en_us`, and Illustrator CS3 13.0 `de_de`:

```
BridgeTalk.getTargets();
  => [photoshop,illustrator]
BridgeTalk.getTargets( "9.0" );
  => [photoshop-9.0]
BridgeTalk.getTargets( null );
  => [photoshop-10.0, illustrator-13.0]
BridgeTalk.getTargets( null, "en_US" );
  => [photoshop-9.0-en_US, photoshop-10.0-en_US]
BridgeTalk.getTargets( null, null );
  => [photoshop-9.0-en_US, photoshop-10.0-en_us, illustrator-13.0-de_de]
```

isRunning()

```
BridgeTalk.isRunning (specifier)
```

specifier A specifier for the target application; see 'Application specifiers' on page 157.

Returns `true` if the given application is running and active on the local computer.

launch()

```
BridgeTalk.launch (specifier [, where])
```

specifier A specifier for the target application; see 'Application specifiers' on page 157.

where Optional. If the value "background" is specified, the application's main window is not brought to the front of the screen.

Launches the given application on the local computer. It is not necessary to launch an application explicitly in order to send it a message; sending a message to an application that is not running automatically launches it.

Returns `true` if the application has already been launched, `false` if it was launched by this call.

loadAppScript()

```
BridgeTalk.loadAppScript (specifier)
```

specifier A specifier for the target application; see 'Application specifiers' on page 157.

Loads the startup script for an application from the common `StartupScripts` folders. Use to implement late loading of startup scripts.

Returns `true` if the script was successfully loaded.

ping()

```
BridgeTalk.ping (specifier, pingRequest)
```

specifier A specifier for the target application; see 'Application specifiers' on page 157.

pingRequest An identifying key string for a specific type of return value. One of:

`STATUS`: Returns the processing status; see `getStatus()`.
`DIAGNOSTICS`: Returns a diagnostic report that includes a list of valid ping keys.
`ECHO_REQUEST`: Returns `ECHO_RESPONSE` for a simple ping request.

Sends a message to another application to determine whether it can be contacted. Returns a string whose meaning is defined by the ping-request key.

pump()

```
BridgeTalk.pump ()
```

Checks all active messaging interfaces for outgoing and incoming messages, and processes them if there are any.

Returns `true` if any messages have been processed, `false` otherwise.

(Most applications have a message processing loop that continually checks the message queues, so use of this method is rarely required.)

BridgeTalk message object

The message object defines the basic communication packet that is sent between applications. Its properties allow you to specify the receiving application (the `target`), the data to send to the target (the `body`), and the `type` of data that is sent. The messaging protocol is extensible; it allows you to define new types of data for the type property, and to send and receive arbitrary additional information with the `headers` property.

BridgeTalk message object constructor

Create a new message object using a simple constructor:

```
var bt = new BridgeTalk;
```

Before you send a message to another application, you must set the `target` property to the receiving application, and the `body` property to the data message (typically a script) you want to send.

BridgeTalk message object properties

body	String	The data payload of the message. Read/write. • If this is an unsolicited message to another application, typically contains a script packaged as a string. The target application's full document object model (DOM) is available within the script. • If this message is a result returned from the static `BridgeTalk` onReceive method of a target application, directed to an onResult callback in this object, contains the return result from that method flattened into a string. See 'Passing values between applications' on page 143. • If this message contains an error notification for the onError callback, contains the error message.

headers	Object	A JavaScript object containing script-defined headers. Read/write.
		Use this property to define custom header data to send supplementary information between applications. You can add any number of new headers. The headers are name/value pairs, and can be accessed with the JavaScript dot notation (*`msgObj`*`.headers.`*`propName`*), or bracket notation (*`msgObj`*`.headers[`*`propName`*`]`). If the header name conforms to JavaScript symbol syntax, use the dot notation. If not, use the bracket notation.
		The predefined header `["Error-Code"]` is used to return error messages to a sender; see 'Messaging error codes' on page 156.
		Examples of setting headers:
		`bt.headers.info = "Additional Information";`
		`bt.headers ["Error-Code"] = 8;`
		Examples of getting header values:
		`var info = bt.headers.info;`
		`var error = bt.headers ["Error-Code"];`

sender	String	The application specifier for the sending application (see 'Application specifiers' on page 157). Read/write.

target	String	The application specifier for the target, or receiving, application (see 'Application specifiers' on page 157). Read/write.
timeout	Number	The number of seconds before the message times out. Read/write. If a message has not been removed from the input queue for processing before this time elapses, the message is discarded. If the sender has defined an onTimeout callback for the message, the target application sends a timeout message back to the sender.
type	String	The message type, which indicates what type of data the body contains. Read/write. Default is `ExtendScript`. You can define a type for script-defined data. If you do so, the target application must have a static `BridgeTalk` onReceive method that checks for and processes that type.

BridgeTalk message object callbacks

Note: The message callbacks are optional, and are not implemented by all message-enabled applications.

onError	Function	A callback function that the target application invokes to return an error response to the sender. It can send JavaScript run-time errors or exceptions, or C++ exceptions.
		To define error-response behavior, set this to a function definition in the following form:
		`bridgeTalkObj.onError = function( errorMsgObject ) {`
		`  // error handler defined here`
		`};`
		The body property of the received message object contains the error message, and the headers property contains the error code in its `Error-Code` property. See 'Messaging error codes' on page 156.
		The function returns undefined.

onReceived	Function	A callback function that the target application invokes to confirm that the message was received. (Note that this is different from the static onReceive method of the `BridgeTalk` class that handles unsolicited messages.)
		To define a response to receipt notification, set this to a function definition in the following form:
		`bridgeTalkObj.onReceived = function( origMsgObject ) {`
		`  // handler defined here`
		`};`
		The target passes back the original message object, with the body property set to the empty string.
		The function returns undefined.

onResult	Function	A callback function that the target application invokes to return a response to the sender. This can be an intermediate response or the final result of processing the message.
		To handle the response, set this to a function definition in the following form:
		`bridgeTalkObj.onResult = function( responseMsgObject ) {`
		`  // handler defined here`
		`};`
		The target passes a new message object, with the `body` property set to the result string. This is the result of the target application's static `BridgeTalk` onReceive method, packaged as a UTF-8-encoded string. See 'Passing values between applications' on page 143.

onTimeout	Function	A callback function that the target application invokes with a timeout message if timeout occurred before the target finished processing another message previously sent by this application. To enable this callback, the message must specify a value for the timeout property.
		To define a response to the timeout event, set this to a function definition in the following form:
		`bridgeTalkObj.onTimeout = function( timeoutMsgObject ) {`
		`  // handler defined here`
		`};`

BridgeTalk message object functions

send()

`bridgeTalkObj.send ([timoutInSecs[, launchParameters]])`

timoutInSecs Optional. A maximum number of seconds to wait for a result before returning from this function. The message is sent synchronously, and the function does not return until the target has processed the message or this number of seconds have passed.

If not supplied or 0, the message is sent asynchronously, and the function returns immediately without waiting for a result.

launchParameters Optional. A string of parameters to append to the name of the target application when launching it, if the application is not already running.

If the target application is already running, this value is ignored.

Sends this message to the `target` application.

Returns `true` if the message could be sent immediately, `false` if it could not be sent or was queued for sending later.

If the target application is not running and the message contains a body, the messaging system automatically launches the target application, passing in any supplied launch parameters. In this case, the message is queued rather than sent immediately, and this method returns `false`. The message is processed once the application is running.

Sending the message does not guarantee that the target actually receives it. You can request notification of receipt by defining an onReceived callback for this message object. (Note that this is different from the static onReceive method of the `BridgeTalk` class that handles unsolicited messages.)

sendResult()

bridgeTalkObj.sendResult (*result*)

result You can send data of any type as the result value. The messaging framework creates a BridgeTalk message object, and flattens this value into a string which it stores in the body of that message. See 'Passing values between applications' on page 143.

When processing an unsolicited message, the static BridgeTalk onReceive method can return an intermediate result to the sender by calling this method in the received message object. It invokes the onResult callback of the original message, passing a new message object containing the specified *result* value.

This allows you to send multiple responses to messages.

Returns true if the received message has an onResult callback defined and the response message can be sent, false otherwise.

Messaging error codes

The interapplication messaging protocol defines the following error codes, which are compatible with ExtendScript error codes. Negative values indicate unrecoverable errors that cause ExtendScript to terminate a running script.

1	General error
8	Syntax error
20	Bad argument list
27	Stack overrun
-28	Out of memory
-29	Uncaught exception
31	Bad URI
32	Cannot perform requested action
-33	Internal error
-36	Not yet implemented
41	Range error
44	Cannot convert
47	Type mismatch
48	File or folder does not exist
49	File of folder already exists
50	I/O device is not open
51	Read past EOF
52	I/O error
53	Permission denied
54	JavaScript execution
56	Cannot connect
57	Cannot resolve reference
58	I/O timeout
59	No response

Application and Namespace Specifiers

All forms of interapplication communication use Application specifiers to identify Adobe applications.

- In all ExtendScript scripts, the `#target` directive can use an specifier to identify the application that should run that script. See 'Preprocessor directives' on page 215.
- In interapplication messages, the specifier is used as the value of the `target` property of the message object, to identify the target application for the message.
- Adobe Bridge (which is integrated with many Adobe applications) uses an application specifier as the value of the `document.owner` property, to identify another application that created or opened an Adobe Bridge browser window. For details, see the *Bridge JavaScript Reference*.

When a script for one application invokes cross-DOM or exported functions, it identifies the exporting application using Namespace specifiers.

Application specifiers

Application specifiers are strings that encode the application name, a version number and a language code. They take the following form:

appname`[_`*instance*`[[-`*version*`[-`*locale*`]]]`

appname	An Adobe application name. One of:
	`acrobat`
	`aftereffects`
	`audition`
	`bridge`
	`contribute`
	`devicecentral`
	`dreamweaver`
	`encore`
	`estoolkit`
	`fireworks`
	`flash`
	`illustrator`
	`incopy`
	`indesign`
	`indesignserver`
	`photoshop`
	`photoshopalbum`
	`photoshopelements`
	`premiere`
	`stockphotos`

instance	Optional. A sequential, 0-based numeric value that distinguishes the instance, for those applications, such as InDesign, that support the launching and running of multiple instances. . For example, `indesign_1` designates the second instance of the application to be launched in the current session.

version	Optional. A number indicating at least a major version. The number can include a minor version separated from the major version number by a dot; for example, `1.5`.
	If not supplied, assumes the same suite version as the sending application, if possible; otherwise, the highest available version number.
	These are the latest version numbers for the Creative Suite 3 release:
	`acrobat-8.0`
	`aftereffects-8.0`
	`audition-3.0`
	`bridge-2.0`
	`contribute-4.5`
	`devicecentral-1.0`
	`dreamweaver-9.0`
	`encore-3.0`
	`estoolkit-2.0`
	`fireworks-9.0`
	`flash-9.0`
	`illustrator-13.0`
	`incopy-5.0`
	`indesign-5.0`
	`indesignserver-5.0`
	`photoshop-10.0`
	`photoshopalbum`
	`photoshopelements`
	`premiere-3.0`
	`stockphotos-1.5`

locale	Optional. An Adobe locale code, consisting of a 2-letter ISO-639 language code and an optional 2-letter ISO 3166 country code separated by an underscore. Case is significant. For example, `en_us`, `en_uk`, `ja_jp`, `de_de`, `fr_fr`. If not supplied, ExtendScript uses the current platform locale. Do not specify a locale for a multilingual application, such as Bridge, that has all locale versions included in a single installation.

The following are examples of legal specifiers:

```
photoshop
bridge-2.0
indesign_1-5.0
illustrator-13.0
illustrator-13.0-de_de
```

If a specifier does not supply specific version and locale information, the framework tries to find the most appropriate available installation. It tries to match to available applications in this order:

1. Peer applications (from the same suite)
2. Applications with the highest available version number
3. Applications that are currently running
4. Applications that match the current locale
5. Applications for any locale

Namespace specifiers

When calling cross-DOM and exported functions from other applications, a namespace specifier qualifies the function call, directing it to the appropriate application.

Namespace specifiers consist of an application name, as used in an application specifier, with an optional major version number. Use it as a prefix to an exported function name, with the JavaScript dot notation.

```
appname[majorVersion].functionName(args)
```

For example:

- To call the cross-DOM function `quit` in Photoshop, use `photoshop.quit()`, and to call it in Illustrator®, use `illustrator.quit()`.
- To call the exported function `place`, defined for Illustrator CS3 version 13, call `illustrator13.place(myFiles)`.

For information about the cross-DOM and exported functions, see 'Remote function calls' on page 133.

6 External Communication Tools

ExtendScript offers tools for communicating with other computers or the internet using standard protocols. These objects support external communication:

- The Web Access library defines the FtpConnection Object, which supports FTP and SFTP communication protocols, and the HttpConnection Object, which supports HTTP and HTTPS communication protocols.

  Your script must load the platform-compiled Web Access library as an `ExternalObject` in order to use these objects. For details, see 'Loading the Web Access Library' on page 160. This library is available in:

  - Adobe Bridge CS3.
- The Socket Object supports low-level TCP connections. It is available in the following applications:
  - Adobe Bridge CS3
  - Adobe InDesign CS3
  - Adobe After Effects® CS3
  - Adobe Photoshop CS3

Loading the Web Access Library

To use the FtpConnection Object or HttpConnection Object, you must dynamically load the Web Access library into Adobe Bridge as an `ExternalObject`. This library is compiled as a shared library; a DLL in Windows, a bundle or framework in Mac OS.

For example, use the following JavaScript code:

```
if( webaccesslib == undefined ) {
      if( Folder.fs == "Windows" ) {
         var pathToLib = Folder.startup.fsName + "/webaccesslib.dll";
      } else {
         var pathToLib = Folder.startup.fsName + "/webaccesslib.bundle";
         // verify that the path is valid
      }
      var libfile = new File( pathToLib );
      var webaccesslib = new ExternalObject("lib:" + pathToLib );
   }
```

The location of the compiled library files is determined by the operating system.

- In Windows, the DLLs reside in the executable directory.
- In Mac OS, bundles and frameworks are loaded from the `@executable/../Frameworks` directory. Use the layout of bundles and Frameworks from the `shellframework` sample application as a template. See 'Mac OS library paths and executables' on page 161.

For more information on loading compiled libraries into JavaScript, see Chapter 7, "Integrating External Libraries."

Mac OS library paths and executables

Mac OS locates dynamic libraries using strings that are part of the executable and dynamic libraries. You can examine those strings with the command:

```
otool -L path-to-executable
```

The path to the executable is typically `..../Bridge.app/Contents/MacOS/Bridge`. You can modify this string with the utility `install_name_tool`. It has two different modes:

```
install_name_tool -change old-string new-string path-to-executable
install_name_tool -id id-for-executable path-to-executable
```

The *id-for-executable* is the first string in the table revealed by `otool -L`.

- To refer to the other libraries from the executable program, use `@executable_path/../Frameworks/...`
- To specify a path relative to the library that requests the load, use `@loader_path`.

FtpConnection Object

Supports the FTP and SFTP protocols for file transfer. The object allows you to send data to or receive data from an FTP server, synchronously or asynchronously.

To use the `FtpConnection` object, you must load the Web Access library (`webaccesslib`) into JavaScript as an `ExternalObject`. See 'Loading the Web Access Library' on page 160.

Using File objects with the FtpConnection object

Typically, you create a File Object for use with your `FtpConnection` object. The get() and put() operations automatically open the file for read and write, respectively, if you have not done so explicitly. The default transfer mode is binary.

- To transfer binary files to the server, use code such as the following:

```
var ftp = new FtpConnection( new File("/c/Photo.jpg")) ;
var b = ftp.put(file,"Photo.jpg") ;
file.close() ; // close the file
```

- Similarly, to transfer binary files from the server:

```
var ftp = new FtpConnection( new File("/c/Photo.jpg")) ;
var b = ftp.get("Photo.jpg",file) ;
file.close() ; // close the file
```

The operations do not automatically close the file. This allows you, for example, to use get() to copy many files to a single file on your local file system. For example:

```
var file = new File("/c/archive.bin") ;
ftp.get("a.txt",file) ;
ftp.get("c.txt",file) ;
file.close() ;
```

Open files are eventually closed by the JavaScript garbage collector when there are no remaining JavaScript references.

ExtendScript supports many file filters; see 'File and Folder Supported Encoding Names' on page 39.

Synchronous and asynchronous operation

Two properties of the `FtpConnection`, sync and async, control whether get() and put() operations are performed synchronously or asynchronously. The property values are tied together, and are mutually exclusive. You can set either one, and the other is automatically toggled to the opposite value.

When the property sync is set to `true` (the default), the connection operation blocks the main thread. All operations must be completed before your script continues.

➤ **Example: synchronous operation (blocking)**

```
var ftp = new FtpConnection("ftp://localhost") ;
var file = new File("here.text") ;
// synchronous mode is the default
ftp.get("remote.txt",file) ;
// the operation simply returns when complete
file.close() ;
ftp.close() ;
```

When the property sync is set to `false` (or async set to `true`), the connection operation occurs in a background thread while your script continues to do other work. The background thread sets the property isComplete to `true` when the current operation has finished. If the operation times out, isComplete is set to `true` and error is set to `FtpConnection.errorTimeout`.

Only a single connection to the FTP server is allowed; you cannot start two operations on the server at the same time. If you do attempt to do so, error is set to `FtpConnection.errorCommandActive` to indicate that the connection is waiting to complete a previous operation.

You can define a callback function in the onCallback property, that checks the completion status of an asynchronous call, and closes the file and connection when it is done. Use the pump() function to call that function periodically from the main thread. Typically, a callback function displays and updates a dialog that shows the progress, and allows the user to cancel an asynchronous operation before its completion; your callback can accomplish this using cancel().

➤ **Example: asynchronous operation (non-blocking)**

```
var file = new File("here.text") ;
ftp.sync = false ; // set asynchronous mode
// define callback to check status and close when complete
ftp.onCallback = function(reason,p_log,total) {
   if ( this.isComplete ) {
      file.close() ;
      this.close() ;
      }
   }
// the operation spawns a new thread and returns
ftp.get("remote.txt",file) ;
// at some time and occasionally
// update progress by calling ftp.onCallback()
ftp.pump() ;
```

FtpConnection Object Reference

This section provides details of the `FtpConnection` object's properties and functions.

FtpConnection object constructor

```
[new] FtpConnection ( [url] );
```

url Optional. The URL to which to connect. The URL specifies the protocol; for example:

```
ftp://localhost
sftp://localhost
```

If not provided, you must set the object's `url` property.

FtpConnection object properties

active	Boolean	When `true`, the connection is active, not passive. Sets `passive` to `false`. See the FTP standard (RFC 959) for details. Read-write.
ascii	Boolean	When `true`, the encoding used to transmit data is ASCII . Default is `false`. When set to `true`, sets `binary` to `false`. Read-write.
async	Boolean	When `true`, the connection is asynchronous. Operations spawn a thread and return immediately to the main thread. The background thread sets `isComplete` to `true` when the current operation has finished. If the operation times out, `isComplete` is set to `true` and `error` is set to `errorTimeout`. Default is `false`. When set to `true`, sets sync to `false`. Read-write.
binary	Boolean	When `true`, the encoding used to transmit data is binary. Default is `true`. When set to `true`, sets `ascii` to `false`. Read-write.
cd	String	Sets the current directory when the connection is open. Default is `undefined`. Read-write. Setting to a directory that does not exist causes a JavaScript error, and sets the `error` and `errorString` properties of the object.
dates	Array of Date	The dates of the files in the current directory. Set by the ls() call. An array corresponding to the files array, where each member is a JavaScript `Date` object. Default is `undefined`. Read only.

error	Number	The most recent error encountered in the course of connecting or executing the operation. All functions set this value before returning. A constant value, one of:
		`FtpConnection.errorNoError`
		`FtpConnection.errorCommandActive`
		`FtpConnection.errorUnknownException`
		`FtpConnection.errorUnknown`
		`FtpConnection.errorOutOfMemory`
		`FtpConnection.errorCancelled`
		`FtpConnection.errorUnknownHost`
		`FtpConnection.errorConnectFailed`
		`FtpConnection.errorTimedOut`
		`FtpConnection.errorLoginFailed`

		FtpConnection.errorProtocolError FtpConnection.errorUnknownProtocol FtpConnection.errorChannelOpen FtpConnection.errorChannelClosed FtpConnection.errorOperationPending FtpConnection.errorBadParameters FtpConnection.errorResourceExists FtpConnection.errorResourceDoesntExist FtpConnection.errorResourceInUse FtpConnection.errorAccessDenied FtpConnection.errorOutOfDisk FtpConnection.errorLocalIoError FtpConnection.errorRemoteIoError FtpConnection.errorNotEmpty FtpConnection.errorNotDirectory FtpConnection.errorNotFile FtpConnection.errorBadPathname FtpConnection.errorNotImplemented FtpConnection.errorNotLocked FtpConnection.errorLocked FtpConnection.errorMethodNotAllowed FtpConnection.errorResourceRedirected Default is errorNoError. Read only.
errorString	String	A description of the most recent error encountered in the course of connecting or executing the operation. Default is "OK". Read only.
files	Array of File	The files in the current directory. Set by the ls() call. Default is undefined. Read only.
flags	Array of Number	The access permissions and types for the files in the current directory. Set by the ls() call. An array corresponding to the files array, where each member is a logical OR of these constant values: FtpConnection.flagOtherExecute FtpConnection.flagOtherWrite FtpConnection.flagOtherRead FtpConnection.flagGroupExecute FtpConnection.flagGroupWrite FtpConnection.flagGroupRead FtpConnection.flagOwnerExecute FtpConnection.flagOwnerWrite FtpConnection.flagOwnerRead FtpConnection.flagDirectory FtpConnection.flagSymLink Default is undefined. Read only.
isComplete	Boolean	When true, the operation is completed. See 'Synchronous and asynchronous operation' on page 162. Default is true. Read only.
isOpen	Boolean	When true, the connection to the FTP server is open. Default is false. Read only.

onCallback	Function	Optional. A callback function to the connection thread for asynchronous mode. The object stores progress messages from operation thread; to check on the progress, call pump() on the main thread. The `pump()` method invokes this function on each stored message, passing the operation status at that point. Within the call, you can use `this.`cancel() to halt the asynchronous operation. Read-write. The function must return `undefined`, and take these arguments: `function(reason,p_log,total) { }` • *reason*: The type of progress message. One of: `FtpConnection.reasonStart`: The transfer started. `FtpConnection.reasonComplete`: The transfer is complete. `FtpConnection.reasonFailed`: The transfer failed. `FtpConnection.reasonProgress`: The transfer is in progress. `FtpConnection.reasonLog`: The operation generated a log message. • *p_log*: Depends on the reason for the message: — For a log message, the message string. — For a progress message, the current number of bytes transferred. — Otherwise, `undefined`. • *total*: Depends on the reason for the message: — For a progress message, the total number of bytes to be transferred. — Otherwise `undefined`.
passive	Boolean	When `true`, the connection is passive, not active. See the FTP standard (RFC 959) for details. When set to `true`, sets `active` to `false`. Default is `false`. Read-write.
password	String	The connection password for the FTP server. Set this to override the password given in the URL. Default is `undefined`. Read-write.
proxy	String	Not used.
renamestyle	String	The rename() function takes a source and destination path and file name, so that it can both move and rename the source object. You can normally specify the source and destination without a path or with a relative path (such as `../myfile.htm`). The function interprets the path as relative to the current working directory. This typical case is handled by the default value for this property, `"style1"`. If you connect to an FTP server that cannot parse the ".." notation, change this value to `"style2"`, and specify both source and destination with absolute paths.
sizes	Array of Number	The sizes of the files in the current directory. Set by the ls() call. An array corresponding to the files array, where each member is a number of bytes. Default is `undefined`. Read only.
sync	Boolean	When `true`, the connection is synchronous. Operations block the main thread and return when complete. Default is `true`. When set to `true`, sets `async` to `false`. Read-write.

timeout	Number	An integer, the number of seconds to continue attempting the operation before completing with the error message `errorTimeout`. Default is 5. Read-write.
url	String	The URL of the FTP server, and optionally the port, to which to connect. This includes the protocol (FTP or SFTP), and can include a login user name and password in this format: `[s]ftp://[[username:]password@]server[:port]` This string must use escape sequences for special characters, such as %20 for space and %40 for @. Default is `undefined`. Read-write.
username	String	The connection user name for the FTP server. Set this to override the user name given in the URL. Default is `undefined`, for anonymous FTP. Read-write.

FtpConnection object functions

All functions set the `error` property to indicate the status of the operation when completed (`errorNoError` on success).

cancel()

```
ftpObj.cancel ();
```

Cancels the current operation, if it is being performed asynchronously. See 'Synchronous and asynchronous operation' on page 162.

Returns `true` on success.

close()

```
ftpObj.close ();
```

Terminates the open connection. Deleting the object also closes the connection, but not until JavaScript garbage-collects the object. The connection might stay open longer than you wish if you do not close it explicitly. There are a limited number of open connections available; failing to close connections can make you unable to open a new one.

Returns `true` if the connection was closed, `false` on I/O errors.

chmod()

```
ftpObj.chmod (remote[, flags]);
```

remote String. The name of the remote file-system object.

flags Optional. The new permissions. A logical OR of the flags constants.

Changes the permissions and/or type of a file-system object on the FTP server.

Returns `true` on success.

cwd()

```
ftpObj.cwd (remote);
```

remote String. The name of the remote directory.

Changes the current directory on the FTP server.

Returns `true` on success.

date()

```
ftpObj.date (remote);
```

remote String. The name of the remote file.

Retrieves the date information for a file-system object on the FTP server.

Returns an array of three JavaScript `Date` objects, for the creation, modification, and most recent access dates.

Returns `false` if dates are unavailable; as when the file-system object does not exist, is a directory, or is a link that cannot be resolved. A dates value of `undefined` indicates that this is the case.

del()

```
ftpObj.del (remote);
```

remote String. The name of the remote file-system object.

Deletes a file-system object on the FTP server.

Returns `true` on success.

exists()

```
ftpObj.exists (remote);
```

remote String. The name of the remote file-system object.

Reports whether a file-system object exists on the FTP server.

Returns `true` if the object exists on the server, `false` if it does not exist or is a link that cannot be resolved.

get()

```
ftpObj.get (remote, file);
```

remote String. The name of the remote file containing data to transfer.

file A File Object, the local file in which to receive the data.

Transfers date from a file on the FTP server to a local file.

Returns `true` on success.

isDir()

```
ftpObj.isDir (remote);
```

remote String. The name of the remote file-system object.

Reports whether a file-system object on the FTP server is a directory.

Returns `true` if the file is a directory on the server, `false` otherwise.

ls()

```
ftpObj.ls ();
```

Retrieves information about the current directory, and returns it in the files, dates, sizes, and flags properties of this object.

Returns `true` on success, `false` on I/O errors.

mkdir()

```
ftpObj.mkdir (remote);
```

remote String. The name of the new remote directory.

Creates a directory on the FTP server.

Returns `true` on success.

open()

```
ftpObj.open ();
```

Opens the FTP connection explicitly. This in not typically needed; calling a function to perform an operation opens the connection if necessary.

Returns `true` if the connection was successfully opened, `false` on I/O errors.

pump()

```
ftpObj.pump ();
```

Executes the callback procedure defined in onCallback on all progress messages that have been received since the last call to this function.

Use this function in the main thread to invoke the callback, in order to check on the progress of an asynchronous operation. It is not required, however; the asynchronous operation continues to progress on the spawned thread, whether or not you make this call.

Returns `true` on success, `false` on I/O errors.

put()

```
ftpObj.get (file, remote[, putMode]);
```

file A File Object, the local file containing data to transfer.

remote String. The name of the remote file in which to receive the data.

putMode Optional. The style of transfer, one of these constants:

- `FtpConnection.putModeTruncateOrCreate` (default): Allows creation of the target file, and truncates an existing file to the size of data written. Does not lock the target file.
- `FtpConnection.putModeExclusive`: Locks the target file during the write operation.

Transfers data from a local file to a file on the FTP server. Overwrites the target file, if it already exists.

Returns `true` on success.

rename()

```
ftpObj.rename (from, to);
```

from String. The path and file name of the source object in the remote file system.

to String. The path and file name of the destination object in the remote file system.

Moves and changes the name of a file-system object on the FTP server. The path can be absolute, or (in most cases) relative to the current working directory; see renamestyle.

Returns `true` on success.

rmdir()

```
ftpObj.rmdir (remote);
```

remote String. The name of the remote directory.

Deletes a directory on the FTP server.

Returns `true` on success.

size()

```
ftpObj.size (remote);
```

remote String. The name of the remote file-system object.

Retrieves the size of a file-system object on the FTP server.

Returns the number of bytes in the file, or -1 if there is no such file, or if the object is a directory or a link that cannot be resolved.

HttpConnection Object

Supports the HTTP and HTTPS protocols for Internet communication. The object allows your script to open a connection to a remote computer that acts as an HTTP server, send an HTTP request, and receive the response.

To use the `HttpConnection` object, you must load the Web Access library (`webaccesslib`) into JavaScript as an `ExternalObject`. See 'Loading the Web Access Library' on page 160.

The `HttpConnection` object can open only one connection to the internet. If you call execute() before the current operation is complete (status is `HttpConnection.statusCompleted`), the current operation is terminated.

Requests and responses

The method property of the `HttpConnection` object determines the type of operation: GET, PUT, POST, HEAD, or DELETE. The GET operation is the default.

The request and response properties can contain File Objects or strings.

- Request and response files

  The default encoding for both request and response files is BINARY; you can specify another encoding in the File Object; see File and Folder Supported Encoding Names. (The `HttpConnection` properties requestencoding and responseencoding affect only string values, not files.)

  If the file is not open, it will be opened for reading (for a request) or for writing (for a response). Request and response files are not closed automatically; when there are no remaining JavaScript references to a file, it is eventually closed by the garbage collector.

- Request and response strings

  When the request is a string, it is converted to binary as specified by the requestencoding value. The default encoding is UTF-8.

  When the server response is anything other than a file, it is converted to a string using the responseencoding value; the default is ASCII.

➤ Getting a file

```
var http = new HttpConnection("http://www.clanmills.com/robin.shtml") ;
http.response = new File("/c/temp/robin.shtml") ;
// Get is the default method
http.execute() ;
http.response.close() ;
```

➤ Posting a string

```
var http = new HttpConnection("http://localhost/perlasp/wform.asp" ) ;
http.request = "Yourname=Fred Smith" ;
http.method = "POST"
http.execute() ;
```

➤ Adding request headers and printing response headers

```
var http = new HttpConnection("http://localhost/perlasp/httpvar.asp") ;
http.requestheaders = ["MyHeader" , "MyValue"] ;
http.execute() ;
```

```
http.response = new File("/c/temp/dumpvars.txt") ;
var a = http.responseheaders ;
for ( i = 0 ; i < a.length/2 ; i++ ) {
   alert(a[i*2] + " => " + a[i*2+1]) ;
   }
```

Asynchronous operations

The `HttpConnection` object can operate asynchronously; when you set async to true (or sync to false) the operation is performed in the background, while your script continues to do other work. However, the asynchronous behavior is not automatic. You must execute the pump() method periodically to increment the progress of the operation, and periodically test the status and lastread properties. After the status is `HttpConnection.statusCompleted`, you must continue to call pump() to transfer all bytes from the server to your object, until lastread is negative.

➤ **Blocking (synchronous use)**

```
var http = new HttpConnection("http://www.clanmills.com/robin.shtml") ;
http.response = new File("/c/newfile.htm") ;
http.execute() ;
```

➤ **Not blocking (asynchronous call)**

```
var http = new HttpConnection("http://some.website/file.html") ;
http.async = true ; // or http.sync = false ;
http.onCallback = function() {
   with ( this ) {
      if ( status == HttpConnection.statusComplete && http.lastread < 0 ) {
         alert("done") ;
         this.close() ;
         }
      }
   return HttpConnection.actionContinue ;
}
http.execute() ; // returns immediately
//
// . . . Somewhere and occasionally
if ( http.status <= HttpConnection.statusComplete && http.lastread >= 0 )
   http.pump() ;
```

Authentication

You can specify a user login name and password in the URL using the standard syntax:

`http://[`*username*`:][`*password*`@]`*server*`[:`*port*`]/`*path*`?`*querystring*

Use an escape sequence for special characters, such as `%20` for space and `%40` for @.

You can override the user name and password specified in the URL by setting the username and password on the `HttpConnection` object.

If the connection is challenged by the server and authentication is required, the operation invokes your onAuthentication callback function. You can use this set the username and password properties; you cannot use it to change the URL.

➤ **Authentication callback**

```
var http = new HttpConnection("http://www.website.com") ;
http.onAuthentication= function (host,realm,isProxy,retries,
   currentUser,currentPassword) {
      alert ("onHttpAuthentication CALLED" + \n +
         "host = " + host + \n +
         "realm = " + realm + \n +
         "isProxy = " + isProxy + \n +
         "retries = " + retrie + \n +
         "currentUser = " + currentUser + \n +
         "currentPassword = " + currentPassword ) ;
      this.username = "therealusername" ;
      this.password = "thepassword" ;
      return HttpConnection.actionContinue ;
      }
http.execute() ;
```

HttpConnection Object Reference

This section provides details of the `HttpConnection` object's properties and functions.

HttpConnection object constructor

```
[new] HttpConnection ( [url] );
```

url Optional. The URL to which to connect. The URL specifies the protocol; for example:

```
http://localhost
https://localhost
```

If not provided, you must set the object's `url` property.

HttpConnection object properties

async	Boolean	When `true`, the connection is asynchronous. Operations spawn a thread and return immediately to the main thread. The background thread sets `isComplete` to `true` when the current operation has finished. If the operation times out, `isComplete` is set to `true` and `error` is set to `errorTimeout`. Default is `false`. When set to `true`, sets `sync` to `false`. Read-write.
chunked	Boolean	When `true`, send the response using chunked encoding. Default is `true`. Read-write.
bytesReceived	Number	The number of bytes received from the HTTP server. -1 when there is no connection.
bytesSent	Number	The number of bytes transmitted to the HTTP server. -1 when there is no connection.

fault	Number	The error status of the connection. Read only. A constant value, one of:
		`HttpConnection.faultNone`
		`HttpConnection.faultUserCancelled`
		`HttpConnection.faultNoConnection`
		`HttpConnection.faultHostNotFound`
		`HttpConnection.faultNetTimeout`
		`HttpConnection.faultClientTimeout`
		`HttpConnection.faultMalformedUrl`
		`HttpConnection.faultInvalidResponse`
		`HttpConnection.faultUnauthorized`
		`HttpConnection.faultRelocated`

isOpen	Boolean	When `true`, the connection to the FTP server is open. Default is `false`. Read only.
lastread	Number	The number of bytes read from the HTTP server during the last call to pump(). Negative when execution is completely finished. Default is 0. Read only.

method	String	The HTTP method. Read-write. One of:
		`GET` (default)
		`PUT`
		`HEAD`
		`POST`
		`DELETE`

mime	String	The MIME type of the request. Default is `text/html`. Read-write.

network	Number	The network status of the connection. Read only. A constant value, one of:
		`HttpConnection.networkIdle`
		`HttpConnection.networkConnecting`
		`HttpConnection.networkSendingRequestHeaders`
		`HttpConnection.networkSendingRequestBody`
		`HttpConnection.networkAwaitingResponse`
		`HttpConnection.networkReceiveingResponseHeaders`
		`HttpConnection.networkReceiveingResponseBody`
		`HttpConnection.networkResponseComplete`
		`HttpConnection.networkProxyIdle`
		`HttpConnection.networkProxyConnecting`
		`HttpConnection.networkProxyConnected`

onAuthentication	Function	Optional. A callback function invoked by the server if authentication fails using the username and password passed with the original URL. Use this method to override the username and password by setting `this.`username and `this.`password. The callback function takes these arguments: *host*: The server name string. *realm*: A string provided by the server. *isProxy*: True if the server is a proxy. *retries*: Always 1 *currentUser*: The user name string already presented to the server. *currentPassword*: The password string already presented to the server. The function should return `HttpConnection.actionContinue`.

onCallback	Function	Optional. A callback function for the operation being executed. It is automatically invoked periodically during synchronous operations. For an asynchronous operation, each call to pump() invokes this function. Read-write. You can use this function to monitor the progress and check the completion status in this object (the value of `this` in the function), in order to provide progress feedback in the user interface and allow cancellation of long operations. Use `this`.close() in this function to halt the operation. The function takes no arguments. It should return `HttpConnection.actionContinue` or `HttpConnection.actionCancel`.
password	String	The connection password for the HTTP server. Set this to override the password given in the URL. Default is `undefined`, for an unsecured or anonymous connection. Read-write.
proxy	String	The HTTP proxy server. A string containing an IP address and port, or the empty string to use the operating-system default, or `undefined` (the default) for no proxy server. Read-write.
redirect	Number	The maximum number of redirection tries for the request. If the server redirects the request to another server (returning a response status of 301 or 302), this connection resends the request to that server. If it redirects this number of times without success, it returns an error. Default is 5. Read-write.
response	String or File	The response to the request, received from the HTTP server. Read only.
responseencoding	String	The encoding to use in converting the request to a string. Default is `ascii`. Read-write.
responseheaders	Array of String	The response headers, an array of key-value pairs. Read only.
responseStatus	Number	The response status, an HTTP Response code (such as 200 for OK, or 404 for "file not found") or -1 if no status has been received. Read only.
request	String or File	The request to execute on the HTTP server. Read-write.
requestencoding	String	The encoding to use in converting the request string to binary. Default is `utf8`. Read-write.
requestheaders	Array of String	The request headers, an array of key-value pairs. Read-write.
snooze	Number	A number of milliseconds to wait before checking the completion status of synchronous operations. Default is 10. Read-write.

status	Number	The execution status of the request. Read only. A constant value, one of:
		`HttpConnection.statusIdle`
		`HttpConnection.statusRunning`
		`HttpConnection.statusCompleted`
		`HttpConnection.statusSuspended`
		`HttpConnection.statusFailed`

sync	Boolean	When `true`, the connection is synchronous. Operations block the main thread and return when complete. Default is `true`. When set to `true`, sets `async` to `false`. Read-write.
timeout	Number	An integer, the number of seconds to continue attempting to make the connection before completing with the message `faultNetTimeout`. Default is 5. Read-write.
url	String	The URL of the HTTP server, and optionally the port, to which to connect. This includes the protocol (HTTP or HTTPS), and can include a login user name and password in this format: `http[s]://[[`*`username`*`:]`*`password`*`@]`*`server`*`[:`*`port`*`]` Default is `undefined`. Read-write.
username	String	The connection user name for the HTTP server. Set this to override the user name given in the URL. Default is `undefined`, for an anonymous connection. Read-write.

HttpConnection object functions

close()

```
httpObj.close ();
```

Terminates the open connection. Deleting the object also closes the connection, but not until JavaScript garbage-collects the object. The connection might stay open longer than you wish if you do not close it explicitly. There are a limited number of open connections available; failing to close connections can make you unable to open a new one.

Returns `true` if the connection was closed, `false` on I/O errors.

execute()

```
httpObj.execute ();
```

Opens a connection if necessary, executes the request on the HTTP server, and receives the response.

Returns `true` on success, `false` on errors. Check fault for the error c ode.

pump()

```
httpObj.pump ();
```

Increments the progress of an asynchronous connection. You must call this function periodically to advance the progress of an asynchronous operation.

Executes the callback procedure defined in onCallback, passing no arguments.

Returns `true` on success, `false` on I/O errors.

Socket Object

TCP connections are the basic transport layer of the Internet. Every time your Web browser connects to a server and requests a new page, it opens a TCP connection to handle the request as well as the server's reply. The JavaScript `Socket` object lets you connect to any server on the Internet and to exchange data with this server.

The `Socket` object provides basic functionality to connect to a remote computer over a TCP/IP network or the Internet. It provides calls like `open()` and `close()` to establish or to terminate a connection, and `read()` or `write()` to transfer data. The `listen()` method establishes a simple Internet server; the server uses the method `poll()` to check for incoming connections.

Many of these connections are based on simple data exchange of ASCII data, while other protocols, like the FTP protocol, are more complex and involve binary data. One of the simplest protocols is the HTTP protocol. The following sample TCP/IP client connects to a WWW server (which listens on port 80); it then sends a very simple HTTP GET request to obtain the home page of the WWW server, and then it reads the reply, which is the home page together with a HTTP response header.

```
reply = "";
conn = new Socket;
// access Adobe's home page
if (conn.open ("www.adobe.com:80")) {
   // send a HTTP GET request
   conn.write ("GET /index.html HTTP/1.0\n\n");
   // and read the server's reply
   reply = conn.read();
   conn.close();
}
```

After executing this code, the variable `reply` contains the contents of the Adobe home page together with an HTTP response header.

Establishing an Internet server is a bit more complicated. A typical server program sits and waits for incoming connections, which it then processes. Usually, you would not want your application to run in an endless loop, waiting for any incoming connection request. Therefore, you can ask a `Socket` object for an incoming connection by calling the `poll()` method of a `Socket` object. This call would just check the incoming connections and then return immediately. If there is a connection request, the call to `poll()` would return another `Socket` object containing the brand new connection. Use this connection object to talk to the calling client; when finished, close the connection and discard the connection object.

Before a `Socket` object is able to check for an incoming connection, it must be told to listen on a specific port, like port 80 for HTTP requests. Do this by calling the `listen()` method instead of the `open()` method.

The following example is a very simple Web server. It listens on port 80, waiting until it detects an incoming request. The HTTP header is discarded, and a dummy HTML page is transmitted to the caller.

```
conn = new Socket;
// listen on port 80
if (conn.listen (80)) {
   // wait forever for a connection
   var incoming;
   do incoming = conn.poll();
   while (incoming == null);
   // discard the request
   conn.read();
```

```
    // Reply with a HTTP header
    incoming.writeln ("HTTP/1.0 200 OK");
    incoming.writeln ("Content-Type: text/html");
    incoming.writeln();
    // Transmit a dummy homepage
    incoming.writeln ("<html><body><h1>Homepage</h1></body></html>");
    // done!
    incoming.close();
    delete incoming;
  }
```

Often, the remote endpoint terminates the connection after transmitting data. Therefore, there is a connected property that contains `true` as long as the connection still exists. If the connected property returns `false`, the connection is closed automatically.

On errors, the `error` property of the `Socket` object contains a short message describing the type of the error.

The `Socket` object lets you easily implement software that talks to each other via the Internet. You could, for example, let two Adobe applications exchange documents and data simply by writing and executing JavaScript programs.

Chat server sample

The following sample code implements a very simple chat server. A chat client may connect to the chat server, who is listening on port number 1234. The server responds with a welcome message and waits for one line of input from the client. The client types some text and transmits it to the server who displays the text and lets the user at the server computer type a line of text, which the client computer again displays. This goes back and forth until either the server or the client computer types the word "bye".

```
// A simple Chat server on port 1234

function chatServer() {
  var tcp = new Socket;
  // listen on port 1234
  writeln ("Chat server listening on port 1234");
  if (tcp.listen (1234)) {
    for (;;) {
      // poll for a new connection
      var connection = tcp.poll();
      if (connection != null) {
        writeln ("Connection from " + connection.host);
        // we have a new connection, so welcome and chat
        // until client terminates the session
        connection.writeln ("Welcome to a little chat!");
        chat (connection);
        connection.writeln ( "*** Goodbye ***");
        connection.close();
        delete connection;
        writeln ("Connection closed");
      }
    }
  }
}
```

```
function chatClient() {
   var connection = new Socket;
   // connect to sample server
   if (connection.open ("remote-pc.corp.adobe.com:1234")) {
      // then chat with server
      chat (connection);
      connection.close();
      delete connection;
   }
}

function chat (c) {
   // select a long timeout
   c.timeout=1000;
   while (true) {
      // get one line and echo it
      writeln (c.read());
      // stop if the connection is broken
      if (!c.connected)
         break;
      // read a line of text
      write ("chat: ");
      var text = readln();
      if (text == "bye")
         // stop conversation if the user entered "bye"
         break;
      else
         // otherwise transmit to server
         c.writeln (text);
   }
}
```

Socket Object Reference

This section provides details of the object's properties and methods.

Socket object constructor

```
[new] Socket ();
```

Creates and returns a new `Socket` object.

Socket object properties

`connected`	Boolean	When `true`, the connection is active. Read only.
`encoding`	String	Sets or retrieves the name of the encoding used to transmit data. Typical values are "ASCII", "BINARY", or "UTF-8".
`eof`	Boolean	When `true`, the receive buffer is empty. Read only.
`error`	String	A message describing the most recent error. Setting this value clears any error message.

host	String	The name of the remote computer when a connection is established. If the connection is shut down or does not exist, the property contains the empty string. Read only.
timeout	Number	The timeout in seconds to be applied to read or write operations. Default is 10.

Socket object functions

close()

```
socketObj.close ();
```

Terminates the open connection. Deleting the object also closes the connection, but not until JavaScript garbage-collects the object. The connection might stay open longer than you wish if you do not close it explicitly.

Returns `true` if the connection was closed, `false` on I/O errors.

listen()

```
socketObj.listen (port [, encoding]);
```

port Number. The TCP/IP port number to listen on. Valid port numbers are 1 to 65535. Typical values are 80 for a Web server, 23 for a Telnet server and so on.

encoding Optional. String. The encoding to be used for the connection. Typical values are "ASCII", "binary", or "UTF-8". Default is "ASCII".

Instructs the object to start listening for an incoming connection.

The call to `open()` and the call to `listen()` are mutually exclusive. Call one function or the other, not both.

Returns `true` on success.

open()

```
socketObj.open (host [, encoding]);
```

host String. The name or IP address of the remote computer, followed by a colon and the port number to connect to. The port number is required. Valid computer names are, for example, "www.adobe.com:80" or "192.150.14.12:80".

encoding Optional. String. The encoding to be used for the connection. Typical values are "ASCII", "binary", or "UTF-8". Default is "ASCII".

Opens the connection for subsequent read/write operations.

The call to `open()` and the call to `listen()` are mutually exclusive. Call one function or the other, not both.

Returns `true` on success.

poll()

```
socketObj.poll ();
```

Checks a listening object for a new incoming connection. If a connection request was detected, the method returns a new `Socket` object that wraps the new connection. Use this connection object to communicate with the remote computer. After use, close the connection and delete the JavaScript object. If no new connection request was detected, the method returns `null`.

Returns a `Socket` object or `null`.

read()

```
socketObj.read ([count]);
```

count Optional. Number. The number of characters to read. If not supplied, the connection attempts to read as many characters as it can, then returns immediately.

Reads up to the specified number of characters from the connection. Ignores CR characters unless encoding is set to BINARY.

Returns a string that contains up to the number of characters that were supposed to be read, or the number of characters read before the connection closed or timed out.

readln()

```
socketObj.readln ();
```

Reads one line of text up to the next line feed. Line feeds are recognized as LF or CRLF pairs. CR characters are ignored.

Returns a string.

write()

```
socketObj.write (text[, text...]);
```

text String. Any number of string values. All arguments are concatenated to form the string to be written.

Concatenates all arguments into a single string and writes that string to the connection. CRLF sequences are converted to LFs unless encoding is set to BINARY.

Returns `true` on success.

writeln()

```
socketObj.write (text[, text...]);
```

text String. Any number of string values. All arguments are concatenated to form the string to be written.

Concatenates all arguments into a single string, appends a Line Feed character, and writes that string to the connection.

Returns `true` on success.

7 Integrating External Libraries

You can extend the JavaScript DOM for an application by writing a C or C++ shared library, compiling it for the platform you are using, and loading it into JavaScript as an ExternalObject Object. A shared library is implemented by a DLL in Windows, a bundle or framework in Mac OS, or a SharedObject in Unix.

You can access the library functions directly through the `ExternalObject` instance, or you can define an interface that allows your C/C++ code to create and access JavaScript classes and objects.

All Creative Suite 3 applications support this feature.

➤ Example code

The sample code distributed with the Adobe Bridge and JavaScript SDK includes an example that demonstrates how write a C/C++ shared library to be integrated with JavaScript. It is in the directory:

sdkInstall/sdksamples/cpp/

The sample shows how to write a plug-in for Adobe Bridge in C/C++, using the `ExternalObject` mechanism, which enables the C/C++ code to be called from the JavaScript context. Project files for Microsoft Visual Studio 2005 and XCode 2.4 are included in subfolders of *sdkInstall*/sdksamples/cpp/build.

Loading and Using Shared Libraries

To load an external shared library into JavaScript, create a new ExternalObject Object. The instance acts as a container and manager for the JavaScript interface to the library. It provides a logging facility that prints status information to the JavaScript Console in the ExtendScript Toolkit, to help you debug your external library usage.

Once the library has been loaded, its exported symbols become available to JavaScript. In your JavaScript code, you can call the functions defined in the library *directly* in the `ExternalObject` instance, or *indirectly* through library-defined object types.

- Direct access to library calls through the `ExternalObject` instance

  Use the direct access style for C-language libraries. For each function defined in the C library, there is a corresponding method in the `ExternalObject` object. You can pass data to these methods and receive the return value directly. For example:

  ```
  mylib = new ExternalObject ("lib:" + samplelib); // load the library
  alert(mylib.version) ;
  // access functions directly from ExternalObject instance
  var a = mylib.method_abc(1,2.0,true,"this is data") ;
  alert(a) ;
  mylib.unload() ;
  ```

  For details of how to define functions for direct access through the ExternalObject object, see 'Defining entry points for direct access' on page 184.

- Indirect access to library calls through JavaScript classes

  Use the indirect style to access classes defined in a C++ library. For each C++ class defined in the library, a corresponding JavaScript class is automatically defined, and you can access the properties and methods through an instance of that class. For example:

```
anotherlib= new ExternalObject ("lib:" + filespec); // load the library
alert(anotherlib.version) ;
// instantiate library-defined class
var myObject = new MyNewClass() ;
// access functions from instance
var a = myObject.method_abc(1,2.0,true,"this is data") ;
alert(a) ;
anotherlib.unload() ;
```

  For details of how to define functions for direct access through the ExternalObject object, see 'Defining entry points for indirect access' on page 186.

ExternalObject Object

You can specify the name of the library in the constructor, or create the object and call its load() method. The loader first looks into the current folder for the library, then in the startup folder (as specified in `Folder.startup`). You can also specify possible search locations in `ExternalObject.searchFolders`.

Before loading the library, the current folder is temporarily switched to the location of the found executable file.

- In Mac OS, the current directory is set to the bundle or framework folder for the library.
- In Windows and UNIX, the current directory is set to the folder that contains the library.

ExternalObject constructor

```
obj = new ExternalObject ("lib:" + filespec, arg1, ...argn);
```

filespec	The specifier `"lib:"` is case sensitive, and serves as the marker for dynamic libraries. Concatenate this to the base name of the shared library, with or without an extension. ExtendScript appends a file extension if necessary, according to the operating system: • `.dll` in Windows • `.bundle` or `.framework` in Mac OS (only Mach-O bundles are supported) • `.so` in Unix (except for HP/UX, where the extension is `.sl`) The name of the library is case sensitive in Unix.
arg1...argn	Optional. Any number of arguments to pass to the library's initialization routine.

For example:

```
var mylib = new ExternalObject( "lib:myApi.dll" );
```

ExternalObject object properties

log	Boolean	`True` to write status information to standard output (the JavaScript Console in the ExtendScript Toolkit). `False` to turn logging off. Default is `false`.
searchFolders	String	A set of alternate paths in which to search for the shared library files, a single string with multiple path specifications delimited by semicolons (;). Paths can be absolute or relative to the `Folder.` startup location. Default value is: • In Windows, `"Plugins;Plug-Ins;."` • In Mac OS, `"Plugins;Plug-Ins;Frameworks;.;../../../Plug-Ins;../../../Frameworks;../../..;"` • In UNIX, `"Plugins;Plug-Ins;plugins;."`
version	Number	The version of the library, as returned by ESGetVersion()

ExternalObject object functions

load()

obj.load (*spec*)

spec String. The file specification for the compiled library, with or without path information.

Loads a compiled C/C++ library into this object.

Returns `true` if the library is loaded successfully, `false` otherwise.

search()

obj.search (*spec*)

spec String. The file specification for the compiled library, with or without path information.

Reports whether a compiled C/C++ library can be found, but does not load it.

Returns `true` if the library is found, `false` otherwise.

unload()

obj.unload ()

Unloads this compiled C/C++ library. Leaves the library object in an invalid state; any further attempts to access properties or methods of that object result in a run-time error. All objects in the library-defined classes are automatically invalidated.

Returns `undefined`.

Defining entry points for direct access

A library to be loaded and accessed directly through an `ExternalObject` instance must publish the following entry points:

`ESInitialize()`

```
char* ESInitialize (TaggedData* argv, long argc);
```

argv, argc	The pointer to and number of arguments passed to the constructor, in the form of TaggedData.

Called when your library is loaded into memory.

Returns a string of function signatures; see 'Library initialization' on page 185.

`ESGetVersion()`

```
long ESGetVersion (void );
```

Takes no arguments, and returns a version number for the library as a long integer. The result is available in JavaScript as `ExternalObject.version`.

`ESFreeMem()`

```
void ESFreeMem (void* p);
```

p	A pointer to the string.

Called to free memory allocated for a `null`-terminated string passed to or from library functions. Returns nothing.

`ESTerminate()`

```
void ESTerminate (void );
```

Called when your library is being unloaded. See 'Library termination' on page 186.

Takes no arguments, and returns nothing.

Additional functions

The shared library can contain any number of additional functions. Each function corresponds to a JavaScript method in the `ExternalObject` instance. If a function is undefined, ExtendScript throws a run-time error.

Each function must accept the following arguments:

- An array of variant data.
- An argument count.
- A variant data structure that takes the return value.

The variant data does not support JavaScript objects. The following data types are allowed:

`Undefined`
`Boolean`
`Double`

`String`	Must be UTF-8 encoded. The library must define an entry point ESFreeMem(), which ExtendScript calls to release a returned string pointer. If this entry point is missing, ExtendScript does not attempt to release any returned string data.
`Script`	A string to be evaluated by ExtendScript. Use to return small JavaScript scripts that define arbitrarily complex data.

If, when a function is invoked, a supplied parameter is undefined, ExtendScript sets the data type to `undefined` and does not attempt to convert the data to the requested type.

Note: The data type of a return value cannot be predefined; JavaScript functions can return any data type. The called function is free to return any of the listed data types.

Library initialization

ExtendScript calls ESInitialize() to initialize the library.

- The function receives an argument vector containing the additional arguments passed in to the `ExternalObject` constructor.
- The function can return a string of function signatures, which are used to support the ExtendScript Reflection Interface, and to cast function arguments to specific types. You do not need to define a signature for a function in order to make it callable in JavaScript.

Function signatures

If you choose to return a string of function signatures, separate multiple signatures with commas:

```
"functionName1_argtypes, functionName2_argtypes, functionName3"
```

Each function-signature substring begins with the function name, followed by an underscore character and a list of argument data types, represented as a single character for each argument. If the function does not have arguments, you can omit the trailing underscore character (unless there is an underscore in the function name).

The characters that indicate data types are:

`a`	Any type. The argument is not converted. This is the default, if no type is supplied or if a type code is unrecognized.
`b`	Boolean
`d`	signed 32 bit integer
`u`	unsigned 32 bit integer
`f`	64 bit floating point
`s`	String

For example, suppose your library defines these two entry points:

```
One (Integer a, String b);
Two ();
```

The signature string for these two functions would be `"One_ds,Two"`.

Note: You cannot define function overloading by returning multiple different signatures for one function. Attempting to do so produces undefined results.

Library termination

Define the entry point ESTerminate() to free any memory you have allocated when your library is unloaded.

Whenever a JavaScript function makes a call to a library function, it increments a reference count for that library. When the reference count for a library reaches 0, the library is automatically unloaded; your termination function is called, and the `ExternalObject` instance is deleted. Note that deleting the `ExternalObject` instance does *not* unload the library if there are remaining references.

To explicitly unload your library, call the `ExternalObject`.unload() method. This calls the termination function and unloads the library. It does not delete the `ExternalObject` instance, but the contents are invalidated.

Defining entry points for indirect access

The C-client object interface for external libraries allows your C or C++ shared-library code to define, create, use, and manage JavaScript objects.

The following entry points are required if you wish to use the object interface:

`ESClientInterface()`

```
int ESClientInterface (SoCClient_e kReason, SoServerInterface* pServer, SoHServer hServer)
```

kReason	The reason for this call, one of these constants: `kSoCClient_init`: The function is being called for initialization upon load. `kSoCClient_term`.: The function is being called for termination upon unload.
pServer	A pointer to an SoServerInterface containing function pointers for the entry points, which enable the shared-library code to call into JavaScript to create and access JavaScript classes and objects. The shared-library code is responsible for storing this structure between the initialization and termination call, and retrieving it to access the functions.
hServer	An SoHServer reference for this shared library. The server is an object factory that creates and manages SoHObject objects. The shared-library code is responsible for storing this structure between the initialization and termination calls. You must pass it to taggedDataInit() and taggedDataFree().

Your library must define this global function in order to use the object interface to JavaScript. The function is called twice in each session, immediately upon loading the library, and again when unloading it.

Returns an error code, `kESErrOK` on success.

ESMallocMem()

```
void * ESMallocMem ( size_t nbytes)
```

nbytes The number of bytes to allocate.

Provides a memory allocation routine to be used by JavaScript for managing memory associated with the library's objects.

Returns a pointer to the allocated block of memory.

Shared-library function API

Your shared-library C/C++ code defines its interface to JavaScript in two sets of functions, collected in SoServerInterface and SoObjectInterface function-pointer structures.

Return values from most functions are integer constants. The error code `kESErrOK == 0` indicates success.

SoServerInterface

`SoServerInterface` is a structure of function pointers which enable the shared-library code to call JavaScript objects. It is passed to the global ESClientInterface() function for initialization when the library is loaded, and again for cleanup when the library is unloaded. Between these calls, your shared-library code must store the structure and use it to access the communication functions.

You can store information for every object and class in your C code. The recommended method is to create a data structure during the initialize() and free it during finalize(). You can then access that data with setClientData() and getClientData().

The `SoServerInterface` structure contains these function pointers:

```
SoServerInterface {
   SoServerMalloc_f          malloc; // allocate memory
   SoServerFree_f            free; // free memory

   SoServerDumpServer_f      dumpServer; //debugging, show server in console
   SoServerDumpObject_f      dumpObject; //debugging, show object in console

   SoServerAddClass_f        addClass; //define a JS class

   SoServerAddMethod_f       addMethod; // define a method
   SoServerAddMethods_f      addMethods; // define a set of methods
   SoServerAddProperty_f     addProperty; // define a property
   SoServerAddProperties_f   addProperties; // define a set of properties

   SoServerGetClass_f        getClass; // get class for an instance
   SoServerGetServer_f       getServer; // get server for an instance

   SoServerSetClientData_f   setClientData; //set data in instance
   SoServerGetClientData_f   getClientData; //get data from instance

   SoServerEval_f            eval; // call JavaScript interpreter
   SoServerTaggedDataInit_f  taggedDataInit; // init tagged data
   SoServerTaggedDataFree_f  taggedDataFree; // free tagged data
}
```

These functions allow your C/C++ shared library code to create, modify, and access JavaScript classes and objects. The functions must conform to the following type definitions.

malloc()

```
void* malloc (SoHServer hServer, size_t nBytes);
```

hServer The SoHServer reference for this shared library, as passed to your global ESClientInterface() function on initialization.

nBytes The number of bytes.

Allocates a block of memory.

Returns a pointer to the block.

free()

```
void free (SoHObject hObject, void* pMem);
```

hObject The SoHObject reference for an instance of this class.

pMem A pointer to the memory block, as returned by malloc().

Frees a block of memory allocated with malloc().

Returns nothing.

dumpServer()

```
ESerror_t dumpServer (SoHServer hServer);
```

hServer The SoHServer reference for this shared library, as passed to your global ESClientInterface() function on initialization.

Prints the contents of this server to the JavaScript Console in the ExtendScript Toolkit, for debugging.

Returns an error code, kESErrOK on success.

dumpObject()

```
ESerror_t dumpObject (SoHObject hObject);
```

hObject The SoHObject reference for an instance of this class.

Prints the contents of this object to the JavaScript Console in the ExtendScript Toolkit, for debugging.

Returns an error code, kESErrOK on success.

addClass()

```
ESerror_t addClass (SoHServer hServer, char* name, SoObjectInterface_p objectInterface);
```

hServer The SoHServer reference for this shared library, as passed to your global ESClientInterface() function on initialization.

name String. The unique name of the new class. The name must begin with an uppercase alphabetic character.

pObjectInterface A pointer to an SoObjectInterface. A structure containing pointers to the object interface methods for instances of this class.

Creates a new JavaScript class.

Returns an error code, kESErrOK on success.

addMethod()

```
EServer_t addMethod (SoHObject hObject, const char* name, int id, char* desc);
```

hObject	The SoHObject reference for an instance of this class.
name	String. The unique name of the new method.
id	Number. The unique identifier for the new method.
desc	String. A descriptive string for the new method.

Adds new method to an instance.

Returns an error code, kESErrOK on success.

addMethods()

```
EServer_t addMethods (SoHObject hObject, SoCClientName_p pNames);
```

hObject	The SoHObject reference for an instance of this class.
pNames[]	SoCClientName. A structure containing the names and identifiers of methods to be added.

Adds a set of new methods to an instance.

Returns an error code, kESErrOK on success.

addProperty()

```
EServer_t addProperty (SoHObject hObject, const char* name, int id, char* desc);
```

hObject	The SoHObject reference for an instance of this class.
name	String. The unique name of the new property.
id	Number. The unique identifier for the new property.
desc	String. Optional. A descriptive string for the new property, or `null`.

Adds new property to an instance.

Returns an error code, kESErrOK on success.

addProperties()

```
EServer_t addProperties (SoHObject hObject, SoCClientName_p pNames);
```

hObject	The SoHObject reference for an instance of this class.
pNames[]	SoCClientName. A structure containing the names and identifiers of properties to be added.

Adds a set of new properties to an instance.

Returns an error code, kESErrOK on success.

getClass()

```
EServer_t getClass (SoHObject hObject, char* name, int name_l);
```

hObject	The SoHObject reference for an instance of the class.
name	String. A buffer in which to return the unique name of the class.
name_l	Number. The size of the name buffer.

Retrieves this object's parent class name.

Returns an error code, kESErrOK on success.

getServer()

```
EServer_t getServer (SoHObject hObject, SoHServer* phServer,
                     SoServerInterface_p* ppServerInterface);
```

hObject	The SoHObject reference for an instance of the class.
phServer	A buffer in which to return theSoHServer reference for this object.
ppServerInterface	A buffer in which to return the SoServerInterface reference for this object.

Retrieves the interface methods for this object, and the server object that manages it.

Returns an error code, kESErrOK on success.

setClientData()

```
EServer_t setClientData (SoHObject hObject, void* pData);
```

hObject	The SoHObject reference for an instance of the class.
pData	A pointer to the library-defined data.

Sets your own data to be stored with an object.

Returns an error code, kESErrOK on success.

getClientData()

```
EServer_t setClientData (SoHObject hObject, void** pData);
```

hObject	The SoHObject reference for an instance of the class.
pData	A buffer in which to return a pointer to the library-defined data.

Retrieves data that was stored with setClientData().

Returns an error code, kESErrOK on success.

eval()

```
EServer_t eval (SohServer hServer, char* string, TaggedData* pTaggedData);
```

hServer	The SoHServer reference for this shared library, as passed to your global ESClientInterface() function on initialization.
string	A string containing the JavaScript expression to evaluate.
pTaggedData	A pointer to a TaggedData object in which to return the result of evaluation.

Calls the JavaScript interpreter to evaluate a JavaScript expression.

Returns an error code, kESErrOK on success.

taggedDataInit()

```
EServer_t taggedDataInit (SoHSever hServer, TaggedData* pTaggedData);
```

hServer The SoHServer reference for this shared library, as passed to your global ESClientInterface() function on initialization.

pTaggedData A pointer to the TaggedData.

Initializes a TaggedData structure.

Returns an error code, kESErrOK on success.

taggedDataFree()

```
EServer_t setClientData (SoHServer hServer, TaggedData* pTaggedData);
```

hServer The SoHServer reference for this shared library, as passed to your global ESClientInterface() function on initialization.

pTaggedData A pointer to the TaggedData.

Frees memory being used by a TaggedData structure.

Returns an error code, kESErrOK on success.

SoObjectInterface

When you add a JavaScript class with `SoServerInterface.addClass()`, you must provide this interface. JavaScript calls the provided functions in order to interact with objects of the new class.

The `SoObjectInterface` is an array of function pointers defined as follows:

```
SoObjectInterface {
   SoObjectInitialize_f initialize;
   SoObjectPut_f        put;
   SoObjectGet_f        get;
   SoObjectCall_f       call;
   SoObjectValueOf_f    valueOf;
   SoObjectToString_f   toString;
   SoObjectFinalize_f   finalize;
}
```

All `SoObjectInterface` members must be valid function pointers, or `NULL`. You must implement `initialize()` and `finalize()`. The functions must conform to the following type definitions.

initialize()

```
EServer_t initialize (SoHObject hObject, int argc, TaggedData* argv);
```

hObject	The SoHObject reference for this instance.
argc, argv	The number of and pointer to arguments passed to the constructor, in the form of TaggedData.

Required. Called when JavaScript code instantiates this class with the `new` operator:

```
var xx = New MyClass(arg1, ...)
```

The initialization function typically adds properties and methods to the object. Objects of the same class can offer different properties and methods, which you can add with the addMethod() and addProperty() functions in the stored SoServerInterface.

Returns an error code, `kESErrOK` on success.

put()

```
EServer_t put (SoHObject hObject, SoCClientName* name, TaggedData* pValue);
```

hObject	The SoHObject reference for this instance.
name	The name of the property, a pointer to an SoCClientName.
pValue	The new value, a pointer to a TaggedData.

Called when JavaScript code sets a property of this class:

```
xx.myproperty = "abc" ;
```

If you provide `NULL` for this function, the JavaScript object is read-only.

Returns an error code, `kESErrOK` on success.

get()

```
EServer_t get (SoHObject hObject, SoCClientName* name, TaggedData* pValue);
```

hObject	The SoHObject reference for this instance.
name	The name of the property, a pointer to an SoCClientName.
pValue	A buffer in which to return the property value, a TaggedData.

Called when JavaScript code accesses a property of this class:

```
alert(xx.myproperty);
```

Returns an error code, `kESErrOK` on success.

call()

```
EServer_t call (SoHObject hObject, SoCClientName* name, int argc, TaggedData* argv,
                TaggedData* pResult);
```

hObject The SoHObject reference for this instance.

name The name of the method, an SoCClientName.

argc, argv The number and pointer to arguments passed to the call, in the form of TaggedDatas.

pResult A buffer in which to return the result of the call, in the form of TaggedDatas.

Called when JavaScript code calls a method of this class:

```
xx.mymethod()
```

Required in order for JavaScript to call any methods of this class.

Returns an error code, kESErrOK on success.

valueOf()

```
EServer_t valueOf (SoHObject hObject, TaggedData* pResult);
```

hObject The SoHObject reference for this instance.

pResult A buffer in which to return the result of the value, in the form of TaggedDatas.

Creates and returns the value of the object, with no type conversion.

Returns an error code, kESErrOK on success.

toString()

```
EServer_t toString (SoHObject hObject, TaggedData* pResult);
```

hObject The SoHObject reference for this instance.

pResult A buffer in which to return the result of the string, in the form of TaggedDatas.

Creates and returns a string representing the value of this object.

Returns an error code, kESErrOK on success.

finalize()

```
EServer_t finalize (SoHObject hObject);
```

hObject The SoHObject reference for this instance.

Required. Called when JavaScript deletes an instance of this class. Use this function to free any memory you have allocated.

Returns an error code, kESErrOK on success.

Support structures

These support structures are passed to functions that you define for your JavaScript interface:

SoHObject	An opaque pointer (`long *`) to the C/C++ representation of a JavaScript object.
SoHServer	An opaque pointer (`long *`) to the server object, which acts as an object factory for the shared library.
SoCClientName	A structure that uniquely identifies methods and properties.
TaggedData	A structure that encapsulates data values with type information, to be passed between C/C++ and JavaScript.

SoCClientName

The `SoCClientName` data structure stores identifying information for methods and properties of JavaScript objects created by shared-library C/C++ code. It is defined as follows:

```
SoCClientName {
   char* name_sig ;
   uint32_t id ;
   char* desc ;
}
```

name_sig	The name of the property or method, unique within the class. Optionally contains a signature following an underscore, which identifies the types of arguments to methods; see Function signatures. When names are passed back to your SoObjectInterface functions, the signature portion is omitted.
id	A unique identifying number for the property or method, or 0 to assign a generated UID. If you assign the UID, your C/C++ code can use it to avoid string comparisons when identifying JavaScript properties and methods. It is recommended that you either assign all UIDs explicitly, or allow them all to be generated.
desc	A descriptive string or `NULL`.

TaggedData

The `TaggedData` structure is used to communicate data values between JavaScript and shared-library C/C++ code. Types are automatically converted as appropriate.

```
typedef struct {
   union {
      long intval;
      double fltval;
      char* string;
      SoHObject* hObject;
   } data;
   long type;
   long filler;
} TaggedData;
```

intval	Integer and boolean data values. Type is `kTypeInteger`, `kTypeUInteger`, or `kTypeBool`.
fltval	Floating-point numeric data values. Type is `kTypeDouble`.
string	String data values. All strings are UTF-8 encoded and `null`-terminated. Type is `kTypeString` or `kTypeScript`. • The library must define an entry point ESFreeMem(), which ExtendScript calls to release a returned string pointer. If this entry point is missing, ExtendScript does not attempt to release any returned string data. • When a function returns a string of type `kTypeScript`, ExtendScript evaluates the script and returns the result of evaluation as the result of the function call.
hObject	A C/C++ representation of a JavaScript object data value. Type is `kTypeLiveObject6` or `kTypeLiveObjectRelease7`. • When a function returns an object of type `kTypeLiveObject6`, ExtendScript does not release the object. • When a function returns an object of type `kTypeLiveObjectRelease7`, ExtendScript releases the object.
type	The data type tag. One of: `kTypeUndefined`: A null value, equivalent of JavaScript `undefined`. The return value for a function is always set to this by default. `kTypeBool`: A boolean value, 0 for false, 1 for true. `kTypeDouble`: A64-bit floating-point number. `kTypeString`: A character string. `kTypeLiveObject6`: A pointer to an internal representation of an object (SoHObject). `kTypeLiveObjectRelease7`: A pointer to an internal representation of an object (SoHObject). `kTypeInteger`: A 32-bit signed integer value. `kTypeUInteger`: A 32-bit unsigned integer value. `kTypeScript`: A string containing an executable JavaScript script.
filler	A 4-byte filler for 8-byte alignment.

8 ExtendScript Tools and Features

In addition to the specific functional modules and development tools, ExtendScript provides these tools and features:

- Global objects that support debugging and object inspection; these include the Dollar ($) object and the ExtendScript Reflection Interface.
- A localization utility for providing user-interface string values in different languages. See Localizing ExtendScript Strings.
- Global functions for displaying short messages in dialog boxes. See User Notification Dialogs.
- An object type for specifying measurement values together with their units. See Specifying Measurement Values.
- Tools for combining scripts, such as a `#include` directive, and `import` and `export` statements. See Modular Programming Support.
- Support for extending or overriding math and logical operator behavior on a class-by-class basis. See Operator Overloading.

ExtendScript also provides a common scripting environment for all Adobe JavaScript-enabled applications, and allows interapplication communication through scripts. For information on these features, see Chapter 5, "Interapplication Communication with Scripts."

Dollar ($) object

This global ExtendScript object provides a number of debugging facilities and informational methods. The properties of the `$` object allow you to get global information such as the most recent run-time error, and set flags that control debugging and localization behavior. The methods allow you to output text to the JavaScript Console during script execution, control execution and other ExtendScript behavior programmatically, and gather statistics on object use.

Dollar ($) object properties

`build`	Number	The ExtendScript build number. Read only.
`buildDate`	Date	The date the current ExtendScript engine was built. Read only.
`engine`	String	The name of the current ExtendScript engine, if set. Read only.
`error`	Error String	The most recent run-time error information, contained in a JavaScript `Error` object. Assigning error text to this property generates a run-time error; however, the preferred way to generate a run-time error is to throw an `Error` object.
`fileName`	String	The file name of the current script. Read only.
`flags`	Number	Gets or sets low-level debug output flags. A logical AND of the following bit flag values: `0x0002` (2): Displays each line with its line number as it is executed. `0x0040` (64): Enables excessive garbage collection. Usually, garbage collection starts when the number of objects has increased by a certain amount since the last garbage collection. This flag causes ExtendScript to garbage collect after almost every statement. This impairs performance severely, but is useful when you suspect that an object gets released too soon. `0x0080` (128): Displays all calls with their arguments and the return value. `0x0100` (256): Enables extended error handling (see strict). `0x0200` (512): Enables the localization feature of the `toString` method. Equivalent to the localize property.
`global`	Global	Provides access to the `Global` object, which contains the JavaScript global namespace.
`includePath`	String	The path for include files for the current script. Read only.
`level`	Number	The current debugging level, which enables or disables the JavaScript debugger. Read only. One of: 0: No debugging 1: Break on runtime errors 2: Full debug mode

locale	String	Gets or sets the current locale. The string contains five characters in the form *LL_RR*, where *LL* is an ISO 639 language specifier, and *RR* is an ISO 3166 region specifier. Initially, this is the value that the application or the platform returns for the current user. You can set it to temporarily change the locale for testing. To return to the application or platform setting, set to `undefined`, `null`, or the empty string.
localize	Boolean	Enable or disable the extended localization features of the built-in `toString` method. See Localizing ExtendScript Strings.
memCache	Number	Gets or sets the ExtendScript memory cache size in bytes.
os	String	The current operating system version information. Read only.
screens	Array	An array of objects containing information about the display screens attached to your computer. • Each object has the properties `left`, `top`, `right`, and `bottom`, which contain the four corners of each screen in global coordinates. • A property `primary` is `true` if that object describes the primary display.
stack	String	The current stack trace.
strict	Boolean	When `true`, any attempt to write to a read-only property causes a runtime error. Some objects do not permit the creation of new properties when `true`.
version	String	The version number of the ExtendScript engine as a three-part number and description; for example: "3.6.5 (debug)" Read only.

Dollar ($) object functions

Function	Return type
about() `$.about ()` Displays the About box for the ExtendScript component, and returns the text of the About box as a string.	String
bp() `$.bp ([condition])` Executes a breakpoint at the current position. *condition:* Optional. A string containing a JavaScript statement to be used as a condition. If the statement evaluates to `true` or nonzero when this point is reached, execution stops. If no condition is needed, it is recommended that you use the JavaScript `debugger` statement in the script, rather than this method.	`undefined`

Function	Return type
colorPicker() `$.colorPicker (name)` Invokes the platform-specific color selection dialog, and returns the selected color as a hexadecimal RGB value: `0x`*RRGGBB*. *name:* The color to be preselected in the dialog, as a hexadecimal RGB value (`0x`*RRGGBB*), or -1 for the platform default.	Number
evalFile() `$.evalFile (`*path*`[, `*timeout*`])` Loads a JavaScript script file from disk, evaluates it, and returns the result of evaluation. *path:* The name and location of the file. *timeout*: Optional. A number of milliseconds to wait before returning `undefined`, if the script cannot be evaluated. Default is 10000 milliseconds.	Any
gc() `$.gc ()` Initiates garbage collection in the ExtendScript engine.	`undefined`
getenv() `$.getenv (`*envname*`)` Retrieves the value of the specified environment variable, or `null` if no such variable is defined. *envname:* The name of the environment variable.	String
setenv() `$.setenv (`*envname*`, `*value*`)` Sets the value of the specified environment variable, if no such variable is defined. *envname:* The name of the environment variable. *value*: The new value, a string.	`undefined`
sleep() `$.sleep (`*milliseconds*`)` Suspends the calling thread for the given number of milliseconds. *milliseconds:* The number of milliseconds to wait. During a sleep period, checks at 100 millisecond intervals to see whether the sleep should be terminated. This can happen if there is a break request, or if the script timeout has expired.	`undefined`

Function	Return type
write() `$.write (text[, text...]...)` Writes the specified text to the JavaScript Console. *text:* One or more strings to write, which are concatenated to form a single string.	undefined
writeln() `$.writeln (text[, text...]...)` Writes the specified text to the JavaScript Console and appends a linefeed sequence. *text:* One or more strings to write, which are concatenated to form a single string.	undefined

ExtendScript Reflection Interface

ExtendScript provides a reflection interface that allows you to find out everything about an object, including its name, a description, the expected data type for properties, the arguments and return value for methods, and any default values or limitations to the input values.

Reflection object

Every object has a `reflect` property that returns a `reflection` object that reports the contents of the object. You can, for example, show the values of all the properties of an object with code like this:

```
var f = new File ("myfile");
var props = f.reflect.properties;
for (var i = 0; i < props.length; i++) {
    $.writeln('this property ' + props[i].name + ' is ' + f[props[i].name]);
}
```

Reflection object properties

All properties are read only.

description	String	Short text describing the reflected object, or `undefined` if no description is available.
help	String	Longer text describing the reflected object more completely, or `undefined` if no description is available.
methods	Array of ReflectionInfo	An Array of ReflectionInfo objects containing all methods of the reflected object, defined in the class or in the specific instance.
name	String	The class name of the reflected object.
properties	Array of ReflectionInfo	An Array of ReflectionInfo objects containing all properties of the reflected object, defined in the class or in the specific instance. For objects with dynamic properties (defined at runtime) the list contains only those dynamic properties that have already been accessed by the script. For example, in an object wrapping an HTML tag, the names of the HTML attributes are determined at run time.

Reflection object functions

find()

reflectionObj.find (*name*)

name The property for which to retrieve information.

Returns the ReflectionInfo object for the named property of the reflected object, or `null` if no such property exists.

Use this method to get information about dynamic properties that have not yet been accessed, but that are known to exist.

➤ Examples

This code determines the class name of an object:

```
obj = new String ("hi");
obj.reflect.name; // => String
```

This code gets a list of all methods:

```
obj = new String ("hi");
obj.reflect.methods; //=> indexOf,slice,...
obj.reflect.find ("indexOf"); // => the method info
```

This code gets a list of properties:

```
Math.reflect.properties; //=> PI,LOG10,...
```

This code gets the data type of a property:

```
Math.reflect.find ("PI").type; // => number
```

ReflectionInfo object

This object contains information about a property, a method, or a method argument.

- You can access `ReflectionInfo` objects in a Reflection object's `properties` and `methods` arrays, by name or index:

```
obj = new String ("hi");
obj.reflect.methods[0];
obj.reflect.methods["indexOf"];
```

- You can access the `ReflectionInfo` objects for the arguments of a method in the `arguments` array of the `ReflectionInfo` object for the method, by index:

```
obj.reflect.methods["indexOf"].arguments[0];
```

ReflectionInfo object properties

`arguments`	Array of ReflectionInfo	For a reflected method, an array of ReflectionInfo objects describing each method argument.

`dataType`	String	The data type of the reflected element. One of:
		`boolean`
		`number`
		`string`
		`Classname`: The class name of an object.
		Note: Class names start with a capital letter. Thus, the value `string` stands for a JavaScript string, while `String` is a JavaScript `String` wrapper object.
		*: Any type. This is the default.
		`null`
		`undefined`: Return data type for a function that does not return any value.
		`unknown`

defaultValue	any	The default value for a reflected property or method argument, or `undefined` if there is no default value, if the property is undefined, or if the element is a method.
description	String	Short text describing the reflected object, or `undefined` if no description is available.
help	String	Longer text describing the reflected object more completely, or `undefined` if no description is available.
isCollection	Boolean	When `true`, the reflected property or method returns a collection; otherwise, `false`.
max	Number	The maximum numeric value for the reflected element, or `undefined` if there is no maximum or if the element is a method.
min	Number	The minimum numeric value for the reflected element, or `undefined` if there is no minimum or if the element is a method.
name	String Number	The name of the reflected element. A string, or a number for an array index.
type	String	The type of the reflected element. One of: `readonly`: A Read only property. `readwrite`: A read-write property. `createonly`: A property that is valid only during creation of an object. `method`: A method.

Localizing ExtendScript Strings

Localization is the process of translating and otherwise manipulating an interface so that it looks as if it had been originally designed for a particular language. ExtendScript gives you the ability to localize the strings in your script's user interface. The language is chosen by the application at startup, according to the current locale provided by the operating system.

For portions of your user interface that are displayed on the screen, you may want to localize the displayed text. You can localize any string explicitly using the Global localize function, which takes as its argument a *localization object* containing the localized versions of a string.

A localization object is a JavaScript object literal whose property names are locale names, and whose property values are the localized text strings. The locale name is a standard language code with an optional region identifier. For details of the syntax, see Locale names.

In this example, a `msg` object contains localized text strings for two locales. This object supplies the text for an alert dialog.

```
msg = { en: "Hello, world", de: "Hallo Welt" };
alert (msg);
```

ExtendScript matches the current locale and platform to one of the object's properties and uses the associated string. On a German system, for example, the property `de: "Hallo Welt"` is converted to the string `"Hallo Welt"`.

Variable values in localized strings

Some localization strings need to contain additional data whose position and order may change according to the language used.

You can include variables in the string values of the localization object, in the form `%n`. The variables are replaced in the returned string with the results of JavaScript expressions, supplied as additional arguments to the `localize` function. The variable `%1` corresponds to the first additional argument, `%2` to the second, and so on.

Because the replacement occurs after the localized string is chosen, the variable values are inserted in the correct position. For example:

```
today = {
   en: "Today is %1/%2.",
   de: "Heute ist der %2.%1."
   };
d = new Date();
alert (localize (today, d.getMonth()+1, d.getDate()));
```

Enabling automatic localization

ExtendScript offers an automatic localization feature. When it is enabled, you can specify a localization object directly as the value of any property that takes a localizable string, without using the `localize` function. For example:

```
msg = { en: "Yes", de: "Ja", fr: "Oui" };
alert (msg);
```

To use automatic translation of localization objects, you must enable localization in your script with this statement:

```
$.localize = true;
```

The `localize` function always performs its translation, regardless of the setting of the `$.localize` variable. For example:

```
msg = { en: "Yes", de: "Ja", fr: "Oui" };
//Only works if the $.localize=true
alert (msg);
//Always works, regardless of $.localize value
alert ( localize (msg));
```

If you need to include variables in the localized strings, use the `localize` function.

Locale names

A locale name is an identifier string in that contains an ISO 639 language specifier, and optionally an ISO 3166 region specifier, separated from the language specifier by an underscore.

- The ISO 639 standard defines a set of two-letter language abbreviations, such as `en` for English and `de` for German.
- The ISO 3166 standard defines a region code, another two-letter identifier, which you can optionally append to the language identifier with an underscore. For example, `en_US` identifies U.S. English, while `en_GB` identifies British English.

This object defines one message for British English, another for all other flavors of English, and another for all flavors of German:

```
message = {
   en_GB: "Please select a colour."
   en: "Please select a colour."
   de: "Bitte wählen Sie eine Farbe."
};
```

If you need to specify different messages for different platforms, you can append another underline character and the name of the platform, one of `Win`, `Mac`, or `Unix`. For example, this objects defines one message in British English to be displayed on Mac OS, one for all other flavors of English on Mac OS, and one for all other flavors of English on all other platforms:

```
pressMsg = {
   en_GB_Mac: "Press Cmd-S to select a colour.",
   en_Mac: "Press Cmd-S to select a color.",
   en: "Press Ctrl-S to select a color."
};
```

All of these identifiers are case sensitive. For example, `EN_US` is not valid.

➤ How locale names are resolved

1. ExtendScript gets the hosting application's locale; for example, `en_US`.
2. It appends the platform identifier; for example, `en_US_Win`.
3. It looks for a matching property, and if found, returns the value string.
4. If not found, it removes the platform identifier (for example, `en_US`) and retries .

5. If not found, it removes the region identifier (for example, `en`) and retries .
6. If not found, it tries the identifier `en` (that is, the default language is English).
7. If not found, it returns the entire localizer object.

Testing localization

ExtendScript stores the current locale in the variable `$.locale`. This variable is updated whenever the locale of the hosting application changes.

To test your localized strings, you can temporarily reset the locale. To restore the original behavior, set the variable to `null`, `false`, 0, or the empty string. An example:

```
$.locale = "ru"; // try your Russian messages
$.locale = null; // restore to the locale of the app
```

Global localize function

The globally available `localize` function can be used to provide localized strings anywhere a displayed text value is specified. The function takes a specially formatted set of localized versions of a display string, and returns the version appropriate to the current locale.

localize()

```
localize (localization_obj[, args])
localize (ZString)
```

localization_obj A JavaScript object literal whose property names are locale names, and whose property values are the localized text strings. The locale name is an identifier as specified in the ISO 3166 standard, a set of two-letter language abbreviations, such as "en" for English and "de" for German.

For example:

```
btnText = { en: "Yes", de: "Ja", fr: "Oui" };
b1 = w.add ("button", undefined, localize (btnText));
```

The string value of each property can contain variables in the form %1, %2, and so on, corresponding to additional arguments. The variable is replaced with the result of evaluating the corresponding argument in the returned string.

args Optional. Additional JavaScript expressions matching variables in the string values supplied in the localization object. The first argument corresponds to the variable %1, the second to %2, and so on.

Each expression is evaluated and the result inserted in the variable's position in the returned string.

ZString **Internal use only.** A ZString is an internal Adobe format for localized strings, which you might see in Adobe scripts. It is a string that begins with $$$ and contains a path to the localized string in an installed ZString dictionary. For example:

```
w = new Window ("dialog", localize ("$$$/UI/title1=Sample"));
```

For example:

```
today = {
   en: "Today is %1/%2",
   de: "Heute ist der %2.%1."
   };
d = new Date();
alert (localize (today, d.getMonth()+1, d.getDate()));
```

User Notification Dialogs

ExtendScript provides a set of globally available functions that allow you to display short messages to the user in platform-standard dialog boxes. There are three types of message dialogs:

- **Alert**: Displays a dialog containing a short message and an **OK** button.
- **Confirm**: Displays a dialog containing a short message and two buttons, **Yes** and **No**, allowing the user to accept or reject an action.
- **Prompt**: Displays a dialog containing a short message, a text entry field, and **OK** and **Cancel** buttons, allowing the user to supply a value to the script.

These dialogs are customizable to a small degree. The appearance is platform specific.

Global alert function

Displays a platform-standard dialog containing a short message and an **OK** button.

alert()

```
alert (message[, title, errorIcon]);
```

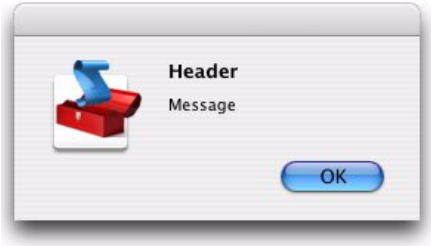
message	The string for the displayed message.
title	Optional. A string to appear as the title of the dialog, if the platform supports a title. Mac OS does not support titles for alert dialogs. The default title string is "Script Alert".
errorIcon	Optional. When `true`, the platform-standard alert icon is replaced by the platform-standard error icon in the dialog. Default is `false`.

Returns: `undefined`

➤ Examples

This figure shows simple alert dialogs on Windows and Mac OS.

This figure shows alert dialogs with error icons.



Global confirm function

Displays a platform-standard dialog containing a short message and two buttons labeled **Yes** and **No**.

confirm()

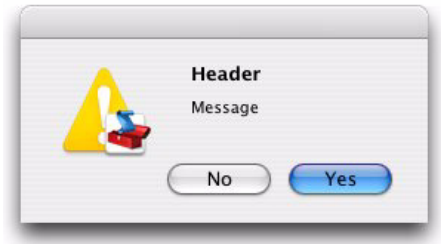
```
confirm (message[,noAsDflt ,title ]);
```

message	The string for the displayed message.
noAsDflt	Optional. When `true`, the **No** button is the default choice, selected when the user types ENTER. Default is `false`, meaning that **Yes** is the default choice.
title	Optional. A string to appear as the title of the dialog, if the platform supports a title. Mac OS does not support titles for confirmation dialogs. The default title string is "Script Alert".

Returns: `true` if the user clicked **Yes**, `false` if the user clicked **No**.

➤ Examples

This figure shows simple confirmation dialogs on Windows and Mac OS.



This figure shows confirmation dialogs with **No** as the default button.

Global prompt function

Displays a platform-standard dialog containing a short message, a text edit field, and two buttons labeled **OK** and **Cancel**.

prompt()

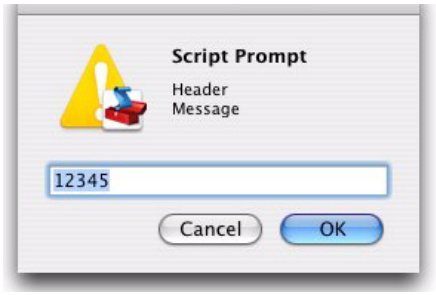
```
prompt (message, preset[, title ]);
```

message	The string for the displayed message.
preset	The initial value to be displayed in the text edit field.
title	Optional. A string to appear as the title of the dialog. On Windows, this appears in the window's frame, while on Mac OS it appears above the message. The default title string is "Script Prompt".

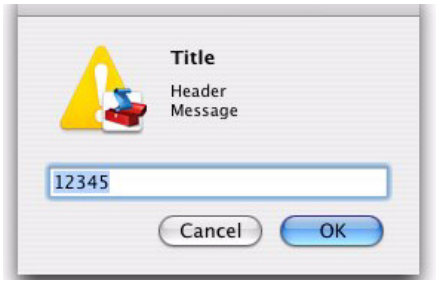
Returns: The value of the text edit field if the user clicked **OK**, `null` if the user clicked **Cancel**.

➤ Examples

This figure shows simple prompt dialogs on Windows and Mac OS.



This figure shows confirmation dialogs with a `title` value specified.



Specifying Measurement Values

ExtendScript provides the UnitValue object to represent measurement values. The properties and methods of the `UnitValue` object make it easy to change the value, the unit, or both, or to perform conversions from one unit to another.

UnitValue object

Represents measurement values that contain both the numeric magnitude and the unit of measurement.

UnitValue object constructor

The `UnitValue` constructor creates a new `UnitValue` object. The keyword `new` is optional:

```
myVal = new UnitValue (value, unit);
myVal = new UnitValue ("value unit");
myVal = new UnitValue (value, "unit");
```

The *value* is a number, and the *unit* is specified with a string in abbreviated, singular, or plural form, as shown in the following table.

Abbreviation	Singular	Plural	Comments
in	inch	inches	2.54 cm
ft	foot	feet	30.48 cm
yd	yard	yards	91.44 cm
mi	mile	miles	1609.344 m
mm	millimeter	millimeters	
cm	centimeter	centimeters	
m	meter	meters	
km	kilometer	kilometers	
pt	point	points	inches / 72
pc	pica	picas	points * 12
tpt	traditional point	traditional points	inches / 72.27
tpc	traditional pica	traditional picas	12 tpt
ci	cicero	ciceros	12.7872 pt
px	pixel	pixels	baseless (see below)
%	percent	percent	baseless (see below)

If an unknown unit type is supplied, the type is set to `"?"`, and the `UnitValue` object prints as "UnitValue 0.00000".

For example, all of the following formats are equivalent:

```
myVal = new UnitValue (12, "cm");
```

```
myVal = new UnitValue ("12 cm");
myVal = UnitValue ("12 centimeters");
```

UnitValue object properties

baseUnit	UnitValue	A UnitValue object that defines the size of one pixel, or a total size to use as a base for percentage values. This is used as the base conversion unit for pixels and percentages; see Converting pixel and percentage values. Default is 0.013889 inches (1/72 in), which is the base conversion unit for pixels at 72 dpi. Set to `null` to restore the default.
type	String	The unit type in abbreviated form; for example, "cm" or "in".
value	Number	The numeric measurement value.

UnitValue object functions

as()

unitValueObj.as (*unit*)

unit The unit type in abbreviated form; for example, "cm" or "in".

Returns the numeric value of this object in the given unit. If the unit is unknown or cannot be computed, generates a run-time error.

convert()

unitValueObj.convert (*unit*)

unit The unit type in abbreviated form; for example, "cm" or "in".

Converts this object to the given unit, resetting the `type` and `value` accordingly.

Returns `true` if the conversion is successful. If the unit is unknown or the object cannot be converted, generates a run-time error and returns `false`.

Converting pixel and percentage values

Converting measurements among different units requires a common base unit. For example, for length, the meter is the base unit. All length units can be converted into meters, which makes it possible to convert any length unit into any other length unit.

Pixels and percentages do not have a standard common base unit. Pixel measurements are relative to display resolution, and percentages are relative to an absolute total size.

- To convert pixels into length units, you must know the size of a single pixel. The size of a pixel depends on the display resolution. A common resolution measurement is 72 dpi, which means that there are 72 pixels to the inch. The conversion base for pixels at 72 dpi is 0.013889 inches (1/72 inch).
- Percentage values are relative to a total measurement. For example, 10% of 100 inches is 10 inches, while 10% of 1 meter is 0.1 meters. The conversion base of a percentage is the unit value corresponding to 100%.

The default `baseUnit` of a `unitValue` object is 0.013889 inches, the base for pixels at 72 dpi. If the `unitValue` is for pixels at any other dpi, or for a percentage value, you must set the `baseUnit` value accordingly. The `baseUnit` value is itself a `unitValue` object, containing both a magnitude and a unit.

For a system using a different dpi, you can change the `baseUnit` value in the `UnitValue` class, thus changing the default for all new `unitValue` objects. For example, to double the resolution of pixels:

```
UnitValue.baseUnit = UnitValue (1/144, "in"); //144 dpi
```

To restore the default, assign `null` to the class property:

```
UnitValue.baseUnit = null; //restore default
```

You can override the default value for any particular `unitValue` object by setting the property in that object. For example, to create a unitValue object for pixels with 96 dpi:

```
pixels = UnitValue (10, "px");
myPixBase = UnitValue (1/96, "in");
pixels.baseUnit = myPixBase;
```

For percentage measurements, set the `baseUnit` property to the measurement value for 100%. For example, to create a `unitValue` object for 40 % of 10 feet:

```
myPctVal = UnitValue (40, "%");
myBase = UnitValue (10, "ft")
myPctVal.baseUnit = myBase;
```

Use the as() method to get to a percentage value as a unit value:

```
myFootVal = myPctVal.as ("ft"); // => 4
myInchVal = myPctVal.as ("in"); // => 36
```

You can convert a `unitValue` from an absolute measurement to pixels or percents in the same way:

```
myMeterVal = UnitValue (10, "m"); // 10 meters
myBase = UnitValue (1, "km");
myMeterVal.baseUnit = myBase; //as a percentage of 1 kilometer
pctOfKm = myMeterVal.as ('%'); // => 1

myVal = UnitValue ("1 in"); // Define measurement in inches
// convert to pixels using default base
myVal.convert ("px"); // => value=72 type=px
```

Computing with unit values

`UnitValue` objects can be used in computational JavaScript expressions. The way the value is used depends on the type of operator.

- Unary operators (`~`, `!`, `+`, `-`)

~unitValue	The numeric value is converted to a 32-bit integer with inverted bits.
!unitValue	Result is `true` if the numeric value is nonzero, `false` if it is not.
+unitValue	Result is the numeric value.
-unitValue	Result is the negated numeric value.

- Binary operators (+, -, *, /, %)

  If one operand is `unitValue` object and the other is a number, the operation is applied to the number and the numeric value of the object. The expression returns a new `unitValue` object with the result as its `value`. For example:

  ```
  val = new UnitValue ("10 cm");
  res = val * 20;
  // res is a UnitValue (200, "cm");
  ```

  If both operands are `unitValue` objects, JavaScript converts the right operand to the same unit as the left operand and applies the operation to the resulting values. The expression returns a new `unitValue` object with the unit of the left operand, and the result `value`. For example:

  ```
  a = new UnitValue ("1 m");
  b = new UnitValue ("10 cm");
  a + b;
  // res is a UnitValue (1.1, "m");
  b + a;
  // res is a UnitValue (110, "cm");
  ```

- Comparisons (=, ==, <, >, <=, >=)

  If one operand is a `unitValue` object and the other is a number, JavaScript compares the number with the `unitValue`'s numeric value.

  If both operands are `unitValue` objects, JavaScript converts both objects to the same unit, and compares the converted numeric values.

  For example:

  ```
  a = new UnitValue ("98 cm");
  b = new UnitValue ("1 m");
  a < b;   // => true
  a < 1;   // => false
  a == 98; // => true
  ```

Modular Programming Support

ExtendScript provides support for a modular approach to scripting by allowing you to include one script in another as a resource, and allowing a script to export definitions that can be imported and used in another script.

Preprocessor directives

ExtendScript provides preprocessor directives for including external scripts, naming scripts, specifying an ExtendScript engine, and setting certain flags. Specify these with a C-style statement starting with the # character:

```
#include "file.jsxinc"
```

When a directive takes one or more arguments, and an argument contains any nonalphanumeric characters, the argument must be enclosed in single or double quotes. This is generally the case with paths and file names, for example, which contain dots and slashes.

#engine *name*	Identifies the ExtendScript engine that runs this script. This allows other engines to refer to the scripts in this engine by importing the exported functions and variables. See Importing and exporting between scripts.
	Use JavaScript identifier syntax for the name. Enclosing quotes are optional. For example:
	`#engine library`
	`#engine "$lib"`

#include *file*	Includes a JavaScript source file from another location. Inserts the contents of the named file into this file at the location of this statement. The *file* argument is an Adobe portable file specification. See Specifying paths. As a convention, use the file extension .jsxinc for JavaScript include files. For example: `#include "../include/lib.jsxinc"` To set one or more paths for the #include statement to scan, use the #includepath preprocessor directive. If the file to be included cannot be found, ExtendScript throws a run-time error. Included source code is not shown in the debugger, so you cannot set breakpoints in it.

#includepath *path*	One or more paths that the #include statement should use to locate the files to be included. The semicolon (;) separates path names.
	
	If a #include file name starts with a slash (/), it is an absolute path name, and the include paths are ignored. Otherwise, ExtendScript attempts to find the file by prefixing the file with each path set by the #includepath statement.
	
	For example:
	
	`#includepath "include;../include"`
	`#include "file.jsxinc"`
	
	Multiple #includepath statements are allowed; the list of paths changes each time an #includepath statement is executed.
	
	As a fallback, ExtendScript also uses the contents of the environment variable JSINCLUDE as a list of include paths.
	
	Some engines can have a predefined set of include paths. If so, the path provided by #includepath is tried before the predefined paths. If, for example, the engine has a predefined path set to predef;predef/include, the preceding example causes the following lookup sequence:
	
	file.jsxinc: literal lookup
	include/file.jsxinc: first #includepath path
	../include/file.jsxinc: second #includepath path
	predef/file.jsxinc: first predefined engine path
	predef/include/file.jsxinc: second predefined engine path

#script *name*	Names a script. Enclosing quotes are optional, but required for names that include spaces or special characters. For example:
	
	`#script SetupPalette`
	`#script "Load image file"`
	
	The *name* value is displayed in the Toolkit Editor tab. An unnamed script is assigned a unique name generated from a number.

#strict on	Turns on strict error checking. See the Dollar ($) object's strict property.
#target *name*	Defines the target application of this JSX file. The *name* value is an application specifier; see Application and Namespace Specifiers. Enclosing quotes are optional. If the Toolkit is registered as the handler for files with the .jsx extension (as it is by default), opening the file opens the target application to run the script. If this directive is not present, the Toolkit loads and displays the script. A user can open a file by double-clicking it in a file browser, and a script can open a file using a File object's execute method.

Importing and exporting between scripts

ExtendScript has been extended to support function calls and variable access across various source code modules and ExtendScript engines. A script can use the export statement to make its definitions available to other scripts, which use the import statement to access those definitions.

To use this feature, the exporting script must name its ExtendScript engine using the #engine preprocessor statement. The name must follow JavaScript naming syntax; it cannot be an expression.

For example, the following script could serve as a library or resource file. It defines and exports a constant and a function:

```
#engine library
export random, libVersion;
const libVersion = "Library 1.0";
function random (max) {
   return Math.floor (Math.random() * max);
}
```

A script running in a different engine can import the exported elements. The import statement identifies the resource script that exported the variables using the engine name:

```
import library.random, library.libVersion;
print (random (100));
```

You can use the asterisk wildcard (`*`) to import all symbols exported by a library:

```
import library.*
```

Objects cannot be transferred between engines. You cannot retrieve or store objects, and you cannot call functions with objects as arguments. However, you can use the JavaScript `toSource` function to serialize objects into strings before passing them. You can then use the JavaScript `eval` function to reconstruct the object from the string.

For example, this function takes as its argument a serialized string and constructs an object from it:

```
function myFn (serialized) {
   var obj = eval (serialized);
   // continue working...
}
```

In calling the function, you deconstruct the object you want to pass into a serialized string:

```
myFn (myObject.toSource()); // pass a serialized object
```

Operator Overloading

ExtendScript allows you to extend or override the behavior of a math or a Boolean operator for a specific class by defining a method in that class with same name as the operator. For example, this code defines the additiona(tor fo-07.8or clssg no this asle (the addt)5.2(tion(oper)685(at)5.1(o siampye adas the)]TJ -196

9 Integrating XML into JavaScript

ExtendScript defines the XML object, which allows you to process XML with your JavaScript scripts. This feature offers a subset of the functionality specified by the ECMA-357 specification (E4X). For more information on this standard, see:

http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-357.pdf

The following applications support this feature:

- Adobe Bridge CS3

The XML Object

The `XML` object represents an XML element node in an XML tree. The topmost `XML` object for an XML file represents the root node. It acts as a list, which contains additional `XML` objects for each element. These in turn contain `XML` objects for their own member elements, and so on.

The child elements of an element tree are available as properties the `XML` object for the parent. The name of the property corresponds to the name of the element. Each property contains an array of `XML` objects, each of which represents one element of the named type.

For example, suppose you have the following, minimal XML code:

```
<rootElement>
   <elementA>
      <elementB></elementB>
   </elementA>
   <elementA>
      <elementB></elementB>
   </elementB>
</rootElement>
```

In a JavaScript script, the `XML` object that you create from this XML code represents the root element:

```
var myRoot = new XML ( "<rootElement> <elementA> <elementB></elementB>
   </elementA> <elementA> <elementB></elementB> </elementB>
   </rootElement>");
```

The object `myRoot` contains a property named `elementA`, which contains two `XML` objects for the two instances of that element. Each of these, in turn, contains an `elementB` property, which contains one empty `XML` object:

```
var elemB1 = myRoot.elementA[0].elementB[0];
```

If an element is empty in the XML, the corresponding property exists and contains an empty `XML` object; it is never `null` or `undefined`.

Accessing XML elements

This sample XML code is used for examples throughout this chapter:

```
<bookstore>
```

```
<book category="COOKING">
   <title lang="en">Everyday Italian</title>
   <author>Giada De Laurentiis</author>
   <year>2005</year>
   <price>30.00</price>
</book>
<book category="CHILDREN">
   <title lang="en">Harry Potter</title>
   <author>J K. Rowling</author>
   <year>2005</year>
   <price>29.99</price>
</book>
<book category="WEB">
   <title lang="en">XQuery Kick Start</title>
   <author>James McGovern</author>
   <author>Per Bothner</author>
   <author>Kurt Cagle</author>
   <author>James Linn</author>
   <author>Vaidyanathan Nagarajan</author>
   <year>2003</year>
   <price>49.99</price>
</book>
<book category="WEB">
   <title lang="en">Learning XML</title>
   <author>Erik T. Ray</author>
   <year>2003</year>
   <price>39.95</price>
</book>
</bookstore>
```

To encapsulate this code in an XML object, serialize it into a string and pass that string to the constructor:

```
var bookXmlStr = "...";
var bookstoreObj = new XML (bookXmlStr);
```

Using this example, the root element `<bookstore>`, is represented by the `XML` object returned from the constructor. Each of the `<book>` elements is available as a member of the `book` property of the XML object.

- The Javascript statement `bookstoreObj.book;` returns the entire list of books.
- The statement `bookstoreObj.book[0];` returns the `XML` object for the first book.
- The statement `bookstoreObj.book[0].author;` returns all authors of the first book.

Accessing XML attributes

Attribute are properties of their parent elements, with the attribute name preceded with an at-sign '@'. An attribute property is a one-element list, which contains an XML object for the value of the attribute. For example:

```
xml.book [0].@category;
```

This returns the `category` attribute of the first book, whose value is the string "COOKING".

To access all `category` attributes of all books, use this statement:

```
xml.book.@category
```

Viewing XML objects

The `XML` object, like all ExtendScript objects, has a toString() method that serializes the contents into a string. This method is called when you evaluate the object in the JavaScript Console of the ExtendScript Toolkit.

In this case, the method recreates the XML text that the object encapsulates. Thus, if you evaluate the object `xml.book[1]` in the Console, you see the XML text for the encapsulated tree, formatted with line feeds and spaces:

```
> xml.book[1];
   <book category="CHILDREN">
      <title lang="en">Harry Potter</title>
      <author>J K. Rowling</author>
      <year>2005</year>
      <price>29.99</price>
   </book>
```

If you evaluate an object with a text value, you see the text value. For example:

```
> xml.book[1].@category;
   CHILDREN
```

If you access multiple values, you see each on a separate line:

```
> xml.book.@category
   COOKING
   CHILDREN
   WEB
   WEB
```

Modifying XML elements and attributes

You can change an element by assigning a value to the corresponding property.

- If the value assigned is an XML element, the element is simply replaced. If there are multiple elements of the same type, the first element is replaced, and all other elements are deleted.
- If the value assigned is not XML, it is converted to astring, and the content of the element is replaced with that string.
- If no element of this type is present, a new element is appended to the XML.

You can change the values of attributes using the same technique.

➤ Modification examples

- In the sample XML, the third book has five `<author>` elements. This statement replaces all of them with a single element, containing a new string:

  ```
  bookstoreObj.book[2].author = "John Warnock";
  ```

  The result is this XML:

  ```
  <book category="WEB">
     <title lang="en">XQuery Kick Start</title>
     <author>John Warnock</author>
     <year>2003</year>
     <price>49.99</price>
  </book>
  ```

- To replace just the first author, leaving all the other authors in place, use this statement:

```
bookstoreObj.book[2].author[0] = "John Warnock";
```

- This statement changes the content of the `<year>` element in the second book, converting the numeric value to a string:

```
bookstoreObj.book[1].year = 2007;
```

- This following statement adds a new `<rating>` element to the second book:

```
bookstoreObj.book[1].rating = "*****";
```

The result is this XML:

```
<book category="CHILDREN">
   <title lang="en">Harry Potter</title>
   <author>J K. Rowling</author>
   <year>2005</year>
   <price>29.99</price>
   <rating>*****</rating>
</book>
```

- This statement changes the value of the `category` attribute of the second book:

```
bookstoreObj.book[1].@category = "LITERATURE, FANTASY"
```

The result is this XML:

```
<book category="LITERATURE, FANTASY">
   <title lang="en">Harry Potter</title>
   ...
```

Deleting elements and attributes

To delete an element or attribute in the XML, use the JavaScript `delete` operator to delete the corresponding element or attribute property. If there are multiple instances of an element, you can delete all, or refer to a single one by its index.

➤ Deletion examples

- This statement deletes all authors from the third book:

```
delete bookstoreObj.book[2].author;
```

- This statement deletes only the second author from the third book:

```
delete bookstoreObj.book[2].author[1];
```

- This statement deletes the `category` attribute from the third book:

```
delete bookstoreObj.book[2].@category;
```

Retrieving contained elements

Several methods in the XML object retrieve the contents, in various ways:

- `XML`.children() gets the direct child elements, including text elements.
- `XML`..elements() gets the direct child elements that are XML tags, but does not get text.
- `XML`.descendants() allows you to match a specific tag, and gets all matching elements at any level of nesting.

For example, consider this XML code loaded into a top-level `XML` object named `x`:

```
<top>
   <one>one text</one>
   <two>
      two text
      <inside>inside text</inside>
   </two>
   top text
</top>
```

Here are the results of the different calls.

- The result of `XML.children()` contains 3 elements, the direct child tags `<one>` and `<two>`, and the directly-contained text of the `<top>` tag:

```
> x.children()
   <one>one text</one>
   <two>
      two text
      <inside>inside text</inside>
   </two>
   top text
> x.children().length()
   3
```

- The result of `XML.elements()` contains 2 elements, the direct child tags `<one>` and `<two>`:

```
> x.elements()
   <one>one text</one>
   <two>
      two text
      <inside>inside text</inside>
   </two>
> x.elements().length()
   2
```

- The result of `XML.descendants()` contains 7 elements, the direct child tags `<one>` and `<two>`, the `<inside>` tag one level down, and the text contents of all the tags:

```
> x.descendants()
   <one>one text</one>
   one text
   <two>
      two text
      <inside>inside text</inside>
   </two>
   two text
   <inside>inside text</inside>
   inside text
   top text
> x.descendants().length()
   7
```

Operations on XML elements

- Use the plus operator, +, to combine XML elements into a list.
- Use the == operator to make an in-depth comparison of two XML trees.

XML Object Reference

This section provides reference details for the properties and methods of the XML object itself, and for the related utility objects and global functions that you use to work with namespaces:

XML object

The XML object provides both static properties and functions, available through the XML class, and dynamic properties and functions available through each instance.

XML object constructor

The constructor returns the XML object representing the root node of an XML tree, which contains additional XML objects for all contained elements.

```
[new] XML (xmlCode);
```

xmlCode	String or XML	A string containing valid XML code, or an existing XML object. • If a valid string is supplied, returns a new XML object encapsulating the XML code. If the XML code cannot be parsed, throws a JavaScript error. • If an existing object is supplied and the new operator is used , returns a copy of the object; otherwise, returns the object itself.

XML class properties

These static properties are available through the XML class. They control how XML is parsed and generated:

ignoreComments	Boolean	When true, comments are stripped from the XML during parsing. Default is false.
ignoreProcessingInstructions	Boolean	When true, processing instructions (<?*xxx*?> elements) are stripped from the XML during parsing. Default is false.
ignoreWhitespace	Boolean	When true, white-space characters are stripped from the XML during parsing. Default is true.
prettyIndent	Number	The number of spaces to use for indenting when pretty-printing. Default is 2.
prettyPrinting	Boolean	When true, toXMLString() uses indenting and line feeds to create the XML string.. Default is true.

XML class functions

These static functions are available through the `XML` class, and provide information about the global settings of the XML parser.

defaultSettings()

```
XML.defaultSettings ();
```

Retrieves the default global option settings that control how XML is parsed and generated.

Returns a JavaScript object containing five properties, which correspond to the five XML class properties.

settings()

```
XML.settings ();
```

Retrieves the current global option settings that control how XML is parsed and generated.

Returns a JavaScript object containing five properties, which correspond to the five XML class properties.

setSettings()

```
XML.setSettings (object);
```

object A JavaScript object containing five properties, which correspond to the five XML class properties.

Sets the global option settings that control how XML is parsed and generated. You can use this to restore settings retrieved with settings() or defaultSettings().

Returns `undefined`.

XML object properties

The properties of the `XML` object are named for and contain the values of the child elements and attributes of the element that the object represents.

childElementName	XML object	Child-element properties are named with the child element name.
@attributeName	XML object	Attribute properties are named with the attribute name prefixed with the at-sign, `@`.

XML object functions

addNamespace()

```
xmlObj.addNamespace (ns);
```

ns A Namespace object.

Adds a namespace declaration to this node.

Returns this XML object.

appendChild()

```
xmlObj.appendChild (child);
```

child An XML object or or any value that can be converted to a String with `toString()`.

Appends a child element to this node, after any existing children. If the argument is not XML, creates a new XML element that contains the string as its text value, using the same element name as the last element currently contained in this object's node.

Returns this XML object.

attributes()

```
xmlObj.attributes (name);
```

name A String, the attribute name.

Retrieves a list of the named attribute elements contained in this node.

Returns an XML object containing all values of the named attribute.

child()

```
xmlObj.child (which);
```

which A String, the element name, or a Number, a 0-based index into this node's child array.

Retrieves a list of all child elements of this node of a given type.

Returns an XML object containing all child elements of the given type.

childIndex()

```
xmlObj.childIndex ();
```

Retrieves the 0-based position index of this node within its parent node.

Returns a Number.

children()

```
xmlObj.children();
```

Retrieves all of the immediate child elements of this node, including text elements.

Returns an XML object containing the child elements.

comments()

```
xmlObj.comments();
```

Retrieves all XML comment elements from this node.

Returns an XML object containing the comments.

contains()

```
xmlObj.contains (element);
```

element An XML object.

Reports whether an element is contained in this node at any level of nesting.

Returns `true` if the element is contained in this XML tree.

copy()

```
xmlObj.copy();
```

Creates a copy of this node.

Returns the new XML object.

descendants()

```
xmlObj.descendants ([name]);
```

name	Optional. A String, the element name to match. If not provided, matches all elements.

Retrieves all descendent elements of this node of a given element type, or all XML-valued descendants, at any level of nesting. Includes text elements.

Returns an XML object containing properties for each descendant element.

elements()

```
xmlObj.elements (name);
```

name	Optional. A String, the element name to match. If not provided, matches all elements.

Retrieves all of the immediate child elements of this node of the given type, or of all types. Does not include text elements.

Returns an XML object containing properties for each child element.

hasComplexContent()

```
xmlObj.hasComplexContent ();
```

Reports whether this node has complex content; that is, whether it contains child elements. Disreclex(apr)129(dns oen)6.6(t)5.5(en)6.6(ts of)-5.7oethnudsc,(incluninga(t)5.5tr)]TJ2-0.3143 0 TD Tc 0.0201

.

Returns;

haShleteons()

insertChildAfter()

```
xmlObj.insertChildAfter (child1, child2);
```

child1 An XML object, the existing child element after which to place the new child, or `null` to insert the new child at the beginning.

child2 An XML object, the new child element, or any value that can be converted to a String with `toString()`.

Inserts a new child element or text node into this node, after another existing child element. If the relative element is not currently in this node, does not insert the new child.

Returns this XML object.

insertChildBefore()

```
xmlObj.insertChildBefore (child1, child2);
```

child1 An XML object, the existing child element before which to place the new child, or `null` to insert the new child at the end.

child2 An XML object, the new child element, or any value that can be converted to a String with `toString()`.

Inserts a new child element or text node into this node, before another existing child element. If the relative element is not currently in this node, does not insert the new child.

Returns this XML object.

length()

```
xmlObj.length ();
```

Reports the number of child elements contained in this node. The minimum number is 1, the element that this object represents.

Returns a Number.

localName()

```
xmlObj.localName ();
```

Retrieves the local name of this element; that is, the element name, without any namespace prefix.

Returns a String.

name()

```
xmlObj.name ();
```

Retrieves the full name of this element, with the namespace information.

Returns a QName object containing the element name and namepace URI.

namespace()

```
xmlObj.namespace ();
```

Retrieves the namespace URI of this element.

Returns a String.

nodeKind()

```
xmlObj.nodeKind ();
```

Reports the type of this node.

Returns a String, one of:

```
element
attribute
comment
processing-instruction
text
```

namespaceDeclarations()

```
xmlObj.namespaceDeclarations ();
```

Retrieves all of the namespace declarations contained in this node.

Returns an Array of Namespace objects .

normalize()

```
xmlObj.normalize ();
```

Puts all text nodes in this and all descendant XML objects into a normal form by merging adjacent text nodes and eliminating empty text nodes.

Returns this XML object.

parent()

```
xmlObj.parent ();
```

Retrieves the parent node of this node.

Returns an XML object, or `null` for the root element.

prependChild()

```
xmlObj.prependChild (child);
```

child	An XML object or string.

Prepends a child element to this node, before any existing children. If you prepend a string to a text element, the result is two text elements; call normalize() to concatenate them into a single text string.

Returns this XML object.

processingInstructions()

```
xmlObj.processingInstructions ([name]);
```

name	A String, the name of a processing instruction, or `null` to get all processing instructions.

Retrieves processing instructions contained in this node.

Returns an XML object containing the children of this 5.6681 Tf2.2 0 TD0.046n50.5 0 0 10.5ng

replace()

```
xmlObj.replace (name, value);
```

name An element or attribute name, with or without the 0-based position index of a specific element, or the wildcard string "*".

- If no position index is supplied, changes the value of all matching elements.
- If the wildcard is supplied, changes the value of all contained elements.

value An XML object or any value that can be converted to a String with `toString()`.

Replaces one or more property values in this node. Acts like the assignment operator; if the named element does not exist, creates and adds one with the given value.

Returns this XML object.

setChildren()

```
xmlObj.setChildren (value);
```

value An XML object or any value that can be converted to a String with `toString()`.

Replaces all of the XML-valued properties in this object with a new value, which can be a simple text element, or can contain another set of XML properties.

Returns this XML object.

setLocalName()

```
xmlObj.setLocalName(name);
```

name A String, the new name.

Replaces the local name of this object; that is, the element name without any namespace prefix.

Returns this XML object.

setName()

```
xmlObj.setName(name);
```

name A String, the new name.

Replaces the full name of this object; that is, the element name and its namespace prefix.

Returns this XML object.

setNamespace()

```
xmlObj.setNamespace(ns);
```

ns A Namespace object for a namespace that has been declared in the tree above this element.

Sets the namespace for this XML element. If the namespace has not been declared in the tree above this element, add a namespace declaration instead.

Returns this XML object.

text()

```
xmlObj.text();
```

Retrieves text nodes from this element.

Returns an XML object

toString()

```
xmlObj.toString();
```

Creates a string representation of this object.

- For text and attribute nodes, this is the textual value of the node.
- For other elements, it is the result of toXMLString().
- If this XML object is a list, concatenates the result of calling the function on each contained element.

Returns a String.

toXMLString()

```
xmlObj.toXMLString();
```

Creates an XML-encoded string representation of this XML object. This result includes the start tag, attributes and end tag of the XML object, regardless of its content. Formats the string as specified by the global settings `XML.prettyPrinting` and `XML.prettyIndent`.

Returns a String.

xpath()

```
xmlObj.xpath (expression[, variables]);
```

expression A String containing an XPath expression.

Note: In this context, include the actual top level element. For example, an expression for the example XML must start with "/bookstore". This is unlike JavaScript property access, where the top level element is implied.

variables Optional. A JavaScript object containing variable definitions. The properties are used to look up XPath variables contained in the expression. For example, if the expression contains the variable `$abc`, the value is in the object's `abc` property.

Evaluates an XPath expression in accordance with the W3C XPath recommendation, using this XML object as the context node. The context position and size are set to 1, and all variables are initially unbound. If this XML object is a list, evaluates all contained XML element nodes (not comments or other node types) and return the results in a list in the order of execution.

If the XPath expression does not evaluate to a node list, throws a JavaScript exception.

Returns an XML object, the result of evaluation.

Global functions

These functions are available in the JavaScript global namespace.

isXMLName()

```
isXMLName (String name)
```

name A string .

Reports whether a string contains a name that conforms to valid XML syntax.

Note: This implementation uses the same rules as for a JavaScript name, except for the '$' character, which is disallowed, and the '-' character, which as added. It does not follow the W3C definition of an XML name, which adds more Unicode characters to the valid set of characters.

Returns `true` if the name is a valid XML name, `false` otherwise.

setDefaultXMLNamespace()

```
setDefaultXMLNamespace (Namespace ns)
```

ns A Namespace object. Any prefix is ignored.

Sets the default namespace for XML objects.

Returns `undefined`.

QName object

This object encapsulates a fully-qualified XML name, the combination of a local XML name and its namespace URI .

QName object constructors

The constructor takes several forms:

```
new QName ()
new QName (name)
new QName (ns)
new QName (uri, name)
```

When no arguments are supplies, creates a `QName` object with an empty local name and no URI.

name	String	Creates a `QName` object with the given local name and the URI of the default namespace. Can be the wildcard character, `"*"`.
name	QName	Creates a copy of an existing QName object.
ns	Namespace	Creates a `QName` object with an empty local name and the URI of the Namespace object.
uri, name	String	Create a `QName` object with the given namespace URI and local name. If the local name is supplied as the wildcard character, `"*"`, the *uri* argument is ignored, and the URI value is that of the default namespace.

QName object properties

name	String	The local element name portion of the XML element's fully-qualified XML name.
uri	String	The namespace prefix of the XML element's fully-qualified XML name.

Namespace object

This object encapsulates the definition of an XML namespace. A namespace associates an XML-name prefix with a complete URI. The prefix is a string that precedes the local name of an XML element or attribute and identifies the namespace, while the URI points to the actual location where the definition of the namespace is found.

For example, this XML definition contains a namespace declaration:

```
<?xml xmlns:adobe=http://www.adobe.com/test?>
```

In the corresponding namespace, the prefix is `adobe`, and the URI is `http://www.adobe.com/test`.

Namespace object constructors

The `Namespace` constructor takes several forms:

```
new Namespace()
new Namespace (String uri)
new Namespace (QName prefix)
new Namespace (Namespace ns)
new Namespace (String prefix, String uri)
```

When no argument is supplied, creates a namespace with an empty prefix and URI.

uri	String	Creates a `Namespace` object with an empty prefix and the given URI.
prefix	QName	Creates a namespace with an empty prefix and the URI set to the URI of the QName object (if the `QName` object contains a URI).
ns	Namespace	Creates a copy of the given Namespace object. If the `Namespace`() function is called without the `new` operator, and the only argument is a `Namespace` object, the function simply returns that object, rather than creating a copy.
prefix, uri	String	Creates a `Namespace` object with the given prefix and the given URI.

Namespace object properties

prefix	String	The element-name prefix associated with the namespace URI. The prefix value can be `undefined`, as when a specified prefix is not a valid XML name. Namespaces with an undefined prefix are completely ignored; they are not added to an XML namespace declaration.
uri	String	The location of the namespace definition, a URI.

10 Porting Guide

This chapter briefly describes changes between this release and the previous release of ExtendScript, to aid you in porting applications to current versions.

New Features in ExtendScript

ExtendScript Toolkit

This release of ExtendScript includes a new look and many new features in the ExtendScript Toolkit; see 'Configuring the Toolkit Window' on page 12. New features include:

- Multiple document windows in which you can edit and run multiple scripts simultaneously, with per-document target selection.
- A powerful search-and-replace facility, with the ability to search for text in multiple files.
- A Scripts palette that provides easy access to frequently-used scripts, configurable with user-defined Favorites folders.
- Configurable keyboard shortcuts for menu commands.

The Toolkit includes an expanded Script Editor with many new features to support a structured view of JavaScript code; see 'The Script Editor' on page 17. New features include:

- A collapsible tree view for JavaScript code files that allows you to collapse and expand structural units
- Find matching brace and select to matching brace
- Line wrapping
- Block indentation
- Comment and uncomment commands
- JavaScript syntax checking
- Configurable syntax highlighting for JavaScript and many other languages

The Toolkit offers expanded debugging features; see 'Debugging in the Toolkit' on page 24. New features include:

- A multi-line JavaScript listener and output console
- The ability to create and navigate to bookmarks in files

The Toolkit now offers an Object Model Viewer, which allows you to inspect the JavaScript object model of an application. See 'Inspecting Object Models' on page 31

ScriptUI

This release includes:

- Support for Flash Players in ScriptUI, through the flashplayer control type.
- New treeview control type, a hierarchical list with collapsible items.

- New Global ScriptUI Object and Global Window Object for access to global settings and built-in dialogs.
- New ScriptUIGraphics Object for low-level control of element appearance.
- Enhancements to resizing behavior and other features of automatic layout.
- Addition of JPEG image format for `IconButton` and `Image` elements.
- Checkmarks for items in list boxes.

Communication and messaging framework

This release includes:

- Timeouts for message receipt and acknowledgement.
- Increased security; see 'Changes in messaging' on page 235.
- Internal improvements for performance and reliability.

External communication tools

This release supports a utility for communicating over sockets, and higher-level support for FTP and HTTP communication protocols.

XML and C/C++ integration

This release provides for the integration of C/C++ shared libraries, and allows you to represent and manipulate XML code directly in JavaScript.

Changes and Deprecations in ExtendScript API

Support objects and features

- The ability to specify script directives within comments using the @ character has been deprecated. Do not continue to use the following syntax:

  ```
  // @include "file.jsxinc"
  ```

  Use the # syntax instead:

  ```
  #include "file.jsxinc"
  ```

- New debugger properties: `$.evalFile()`

Changes in messaging

This release of the messaging framework includes these changes that are incompatible with the previous release:

- Message authentication added

  By default, Creative Suite 3 applications reject messages built with the Creative Suite 2 framework. To accept such messages, set `BridgeTalk.messageAuthentication = false` in the Creative Suite 3 application.

- Changed launch aliases for applications

  The Creative Suite 2 release used XPEP launch aliases for applications. In this release, Creative Suite 3 applications are registered in the Adobe Product Code Database (PCD).

  To use the previous version of messaging with Creative Suite 3 applications, you must either launch them outside the framework, or add XPEP launch aliases for them.