

# Using Help

---

## About online Help

Adobe Systems, Inc. provides complete documentation in the Adobe PDF Help system. The Help system includes information on all the tools, commands, and features for both Windows and Mac OS. The PDF format is designed for easy navigation online, and support for third-party screen readers compatible with Windows. The Help can also be printed as a desktop reference.

## Navigating in Help

The Help will open in an Acrobat window with the bookmark pane open. If the bookmark pane is not open choose Window > Bookmarks. You can also navigate using the navigation bar, the index, or search the document.

At the top and bottom of each page is a navigation bar. Click Using Help to return to this introduction. Clicking Contents, or Index will take you to that section.

The Next Page ▶ and the Previous Page ◀ arrows let you move through the pages sequentially. Click Back to return to the last page you viewed. You can also use the navigation arrows in the Acrobat toolbar.

## Using bookmarks, the table of contents, the index, and Find

The contents of Help are shown as bookmarks in the bookmark pane. To view subtopics, click the plus sign next to a bookmark. Each bookmark is a hyperlink to the associated section of the Help document.

To go to the information, click its bookmark. As the information is displayed in the document pane, its bookmark is highlighted.

You can turn highlighting on or off by selecting the Highlight Current Bookmark option from the bookmark pane menu.

### To find a topic using the table of contents:

- 1 Click Contents in the navigation bar at the top or bottom of any page.
- 2 Click a topic on the Contents page to move to the first page of that topic.
- 3 In the bookmark pane, expand the topic to see its subtopics.

### To find a topic using the index:

- 1 Click Index in the navigation bar at the top or bottom of any page.
- 2 Click the appropriate letter at the top of the page.

You can also expand the Index bookmark, and click the letter in the bookmark pane.

- 3 Locate your entry, and click the page number link to view the information.
- 4 To view multiple entries, click Back to return to the same place in the index.



**To find a topic using the Find command:**

- 1 Choose Edit > Find.
- 2 Enter a word or a phrase in the text box, and click OK.

Acrobat will search the document, starting from the current page, and display the first occurrence of the word or phrase you are searching for.

- 3 To find the next occurrence, choose Edit > Find Again.

**Printing the Help file**

Although the Help has been optimized for on-screen viewing, you can print pages you select, or the entire file.

To print, choose Print from the File menu, or click the printer icon in the Acrobat toolbar.

# Contents

---

[Overview 4](#)

[Writing Scripts 7](#)

[JavaScript Debugging 11](#)

[Reference 18](#)

[Examples 88](#)

[The Socket Object 106](#)

[Encoding Names 113](#)



# Overview

---

## About this Guide

The After Effects 6.0 Render Automation & Scripting Guide demonstrates how to take procedural control of your After Effects projects via scripting. This feature set is available only in After Effects 6.0 Professional edition.

With the use of system level scripting, you can streamline your render pipeline and avoid a lot of repetitive pointing and clicking. If you have used expressions or other JavaScript-like techniques for animating, or worked with system scripting in AppleScript or Visual Basic, you will recognize the power of application scripting in After Effects. With some practice, and with sufficient experience using the JavaScript language, you can take control of your graphics pipeline.

This overview and the “Writing Scripts” chapter familiarize you with the basic concepts for creating and executing After Effects scripts. Following these is a chapter on using the JavaScript debugger, and a chapter containing some specific scripting examples to help you better understand how scripting works. The final section of the guide is an A to Z reference guide of all of the keywords, objects and methods available to you in After Effects. There is also an appendix which addresses use of the socket object for interfacing between After Effects and TCP/IP functionality.

## After Effects objects

You may not think of After Effects as a collection of hierarchical objects, but when you make use of render queue items, compositions and projects, that is how they appear in scripting. Just as the expressions features in After Effects give you access to virtually any property within any effect of any layer inside any composition of your project (each of which we refer to as an object), scripting gives you to access the hierarchy of objects within After Effects and make changes to these objects.

The object model in After Effects scripting is based on ECMAScript (or more specifically, the 3rd Edition of the ECMA-262 Standard). Further documentation on this standard can be found at

<http://www.ecma-international.org>

## What about expressions and motion math?

Scripting does not access individual properties of a layer, and thus it does not overlap with the functionality of expressions in After Effects, although it is based on a similar programmatical model. Scripting takes control of more global areas of After Effects, such as the application itself, the Project window, and the Render Queue, from which expressions can in some cases receive data but to which expressions cannot make changes.



The similarity between expressions and scripting is, however, apparent in that they are both drawn from the same language, ECMA standard JavaScript. Thus, knowing how to utilize one is helpful in understanding the other.

Motion math is no longer installed with After Effects 6.0; its functionality has been superseded by expressions. Although scripting is not used to create animation data like expressions and motion math, all mathematical and logical operators common to ECMAScript are available in scripting.

## What you should know

This guide is primarily for users who manage a graphics pipeline (which may include other scriptable applications as well). There are a few techniques available beyond automating and modifying rendering (for more on these, see “Examples” on page 88), but the predominant use for scripting in After Effects 6.0 is expected to be this type of render automation.

This functionality is also offered via third-party network rendering management solutions. These products feature software designed to help manage this process, so it is possible to take advantage of this functionality without having to perform manual editing of scripts.

Although this guide is intended to provide an understanding of the extensions which have been added to the ECMAScript/JavaScript language for scripting of After Effects projects, to take full advantage of what is possible with scripting you will also need an understanding of writing scripts at the system level (for integration with AppleScript on the Mac and DOS shell scripts on Windows systems) and a background in how to work with JavaScript.

Much of what scripting can accomplish replicates what can be done via the After Effects user interface, so a thorough knowledge of the application itself is also essential to understanding how to use this functionality.

Note that JavaScript objects normally referred to as “properties” are consistently called “attributes” in this guide, so as to avoid confusion with After Effects’ own definition of a Property (which refers to an animatable value of an effect or transform within an individual layer).

## Activating full scripting features

For security reasons, the scripting features which operate outside of the After Effects application (such as adding and deleting files and folders on volumes, or accessing the network) are disabled by default.

To enable these features, under Preferences > General check “Allow Scripts to Write Files and Access Network” (unselected by default).

By selecting this box, you enable the following:

- Writing files
- Creating directories
- Setting the current directory
- Creating a socket
- Opening a socket
- Listening to a socket

## Activating the JavaScript Debugger

The JavaScript Debugger is disabled by default so that casual users do not encounter it. When editing or writing scripts, the JavaScript Debugger can help you diagnose script problems more quickly.

To activate the JavaScript Debugger on the local machine when a script error is encountered:

- Select Preferences > General > “Enable JavaScript Debugger.”

Note that the JavaScript Debugger operates only when executing a script, not with expressions, even though expressions also make use of JavaScript.

## Accessing and writing scripts

There is no script editing functionality within After Effects itself. To create and edit scripts, use an external text editing application which creates files with Unicode UTF-8 text encoding. Beware of applications such as Microsoft Word which by default add header information to files (these will create line 0 errors in scripts, causing them to fail). It is best to use an application designed for scripting such as BBEdit on the Mac or EmEditor on Windows.

A script can reside anywhere, although to appear in the Scripts menu it must be saved in the Scripts folder within the After Effects application directory. For more information, see “Writing Scripts” on page 7.

## Recordable actions and scripts

There is no built-in method for recording a series of actions in After Effects into a script, as you can with Photoshop actions. Scripts are created outside of After Effects and then executed within it, or externally via a command-line or third-party render management software.

## Uses of After Effects scripting

The primary use for scripting in After Effects 6.0 is render automation. Anyone charged with managing a complex rendering pipeline will be interested in this. Render automation can be accomplished either by hand-coding scripts or via a third-party network rendering solution which supports automated management of network rendering pipelines.

There are other uses for scripting; it can be a shortcut around tedious tasks that would otherwise involve repetitious pointing and clicking.

See “Examples” on page 88 for examples of what scripts can do.

# Writing Scripts

---

## Projects and objects

When you use After Effects, you create projects, compositions, and Render Queue items along with all of the elements that they contain: footage, images, solids, layers, masks, effects, and keyframes. Each of these items is an object.

The heart of a scriptable application is the object model. In After Effects, the object model is comprised of projects, items, compositions, layers, and Render Queue items. Each object has its own special attributes, and every object in an After Effects project has its own identity (although not all are accessible to scripting).

You should be familiar with the After Effects object model in order to create scripts. More resources for learning about how to script are included in the final section of this chapter.

## The Scripts menu, folder & editing scripts

### The Scripts folder & menu

After Effects scripts reside in a Scripts directory, within the same directory as your After Effects 6.0 application file. Only scripts contained in this Scripts folder will be automatically listed in the Script menu, although a script file can reside anywhere.

To run a script that does not appear in the Script menu, use the File > Run Script > Choose File... command and choose the script in the Open dialog. Alternatively, you can send After Effects a script from a command line (on Windows) or from AppleScript (on Mac).

To be seen in the Open dialog, your script must include the proper ".jsx" file extension. If your script names do not end with this extension, the application won't see them.

### Shutdown & Startup

Within the Scripts folder are two subfolders called "Startup" and "Shutdown." After Effects runs scripts in these folders automatically on starting and quitting the application, respectively.

In the Startup folder you can place scripts that you wish to execute at startup of the application. They are executed after the application is initialized and all plug-ins are loaded.

Scripting shares a global environment, so any script executed at startup can define functions and properties that are available to all scripts. Defining a function in a startup script will make it available to all other scripts. In other words, variables and functions, once defined by running a script which contains them, persist in succeeding scripts.

Please note that this persistence of global settings also means that if you are not careful about giving variables in scripts unique names, you can inadvertently reassign global variables intended to persist throughout a session.

Properties can also be embedded in existing objects such as the Application object (see "Application Object" on page 23) to extend the application for other scripts.



The Shutdown folder scripts are executed as the application quits. This occurs after the project is closed, but before any other application shutdown occurs.

### Editing scripts (Windows)

You can use any text editor to create, edit and save scripts, but it is recommended that you choose an application that saves with UTF-8 encoding and does not add its own headers.

Windows applications that are useful for editing scripts include EM Editor or the built-in Windows Notepad (be sure to set Encoding within save options to UTF-8).

### Editing scripts (Mac)

You can use any text editor to create, edit and save scripts, but it is recommended that you choose an application that does not automatically add header information when saving files, as Microsoft Word does, and which saves with Unicode (UTF-8) encoding.

Mac applications that are useful for editing scripts include BBEdit or the built-in OS X Textedit (be sure to set the Save type in Preferences to Unicode [UTF-8]).

### Sending a script to After Effects from the system

If you are familiar with how to run a script from the command line in Windows or via AppleScript, you can send a script directly to the open After Effects application which will then run automatically.

### To include After Effects scripting in an AppleScript (Mac only):

Following are three examples of AppleScripts that will send an existing .jsx file containing an After Effects script to the application without using the After Effects user interface to execute the script.

In the first example, you copy your After Effects script directly into the AppleScript and then run it, as follows (your script text would appear in quotes following the DoScript command):

```
tell application " Adobe After Effects 6.0"
    DoScript "alert (\"You just sent an alert to After Effects\")"
end tell
```

Alternatively, you could pop up a dialog asking for the location of the .jsx file to be executed, as follows:

```
set thefile to choose file
tell application "Adobe After Effects 6.0"
    DoScript thefile
end tell
```

Finally, this script is perhaps most useful when you are working directly on editing a .jsx script and want to send it to After Effects for testing or to run. To use it effectively you must enter the application that contains the open .jsx file (in this example it is TextEdit); if you do not know the proper name of the application, type in your best guess to replace "TextEdit" and AppleScript will prompt you to locate it.



Simply highlight the script text that you want to run, and then activate this AppleScript:

```
( *  
This script sends the current selection to After Effects as a script.  
* )
```

```
tell application "TextEdit"  
  
    set the_script to selection as text  
  
end tell  
  
tell application "Adobe After Effects 6.0"  
    activate  
    DoScript the_script  
end tell
```

For more information on using AppleScript, check out David Pogue's Mac OS X, the Missing Manual (Pogue Press/O'Reilly) or Ethan Wilde's AppleScript for Applications (Peachpit Press).

### To include After Effects scripting in a command line (Windows only):

Following are examples of DOS shell scripts that will send an existing .jsx file containing an After Effects script to the application without using the After Effects user interface to execute the script.

In the first example, you would copy and paste your After Effects script directly into the command line script and then run it, as follows (your script text would appear in quotes following the afterfx.exe -s command):

```
afterfx.exe -s "alert ("You just sent an alert to After Effects")"
```

Alternatively, you could specify the location of the .jsx file to be executed, as follows:

```
afterfx.exe -r c:\myDocuments\Scripts\yourAEScriptHere.jsx
```

Furthermore, two other options, "-sq" and "-rq" perform the same functions as "-s" and "-r" respectively, but additionally prevent any dialogs or other user interaction from being displayed while the script is running.

```
afterfx.exe -sq "alert ("You just sent an alert to After Effects")"
```

```
afterfx.exe -rq c:\myDocuments\Scripts\yourAEScriptHere.jsx
```

## Testing and troubleshooting

Any After Effects script that contains an error preventing it from being completed will generate an error dialog from the application. This error includes information about the nature of the error and the line of the script on which it occurred.

Additionally, After Effects includes a JavaScript debugger. By default, this is de-activated in the Preferences to prevent confusion for users who might make use of scripts that they themselves are not responsible for creating and editing.

More information on activating and using the debugger can be found in “JavaScript Debugging” on page 11.

## More resources to learn scripting

Many resources exist for learning more about scripting that uses the ECMA standard.

The After Effects Scripting engine supports the 3rd Edition of the ECMA-262 Standard, including its notational and lexical conventions, types, objects, expressions and statements.

For a complete listing of the keywords and operators included with ECMAScript, please refer to [Ecma-262.pdf](#), available at:

<http://www.ecma-international.org/publications/standards/ECMA-262.HTM>

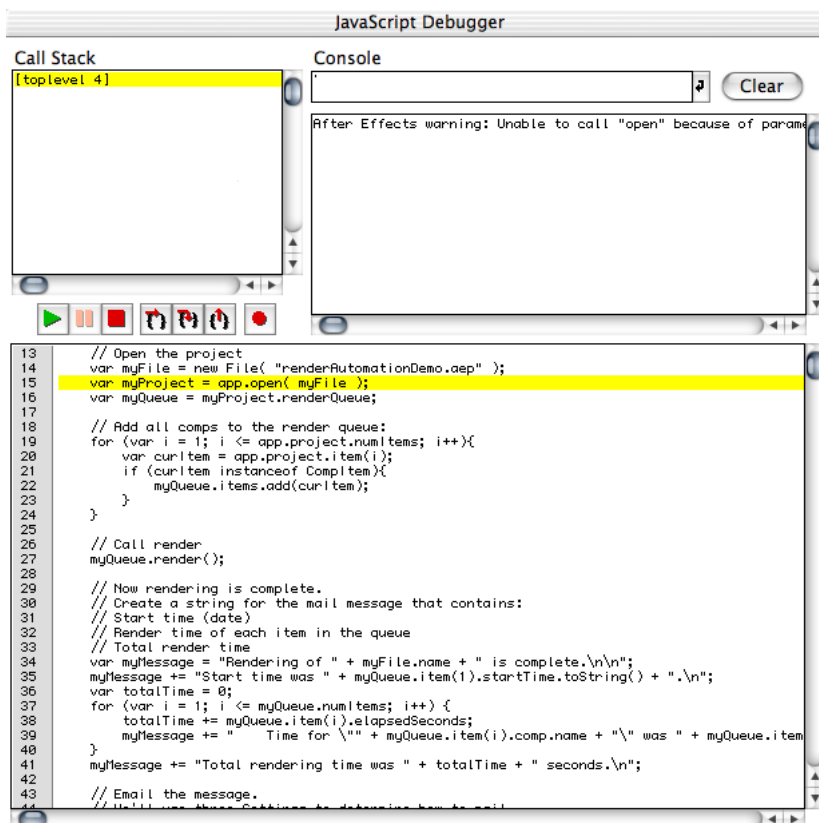
Books that deal with JavaScript 1.2 are also useful for understanding how scripting works in After Effects. One book that is something of a standard for JavaScript users is *JavaScript, The Definitive Guide* by David Flanagan (O’Reilly). Another very readable source is *JavaScript: A Beginner’s Guide* by John Pollock (Osborne). Both of these texts contain information that pertains only to extensions of JavaScript for internet browsers, however they also contain thorough descriptions of scripting fundamentals.

There are also books for using AppleScript and creating Windows command line scripts, each of which can be used to send scripts to After Effects.

# JavaScript Debugging

## The JavaScript Debugger window

This section describes the JavaScript Debugger window which appears when the Enable JavaScript Debugger preference is selected in General Preferences (it is deselected by default) and there is an error when executing a script.



### JavaScript Debugger Window

**A.** Command line **B.** Debug output view **C.** Stack trace view **D.** JavaScript source view **E.** Resume **F.** Pause **G.** Stop **H.** Step over **I.** Step into **J.** Step out **K.** Breakpoints display

The current stack trace appears in the upper-left pane of the script debugger window. This **stack trace view** displays the calling hierarchy at the time of the breakpoint. Double-clicking a line in this view changes the current scope, enabling you to inspect and modify scope specific data.

All debugging output appears in the upper-right pane of the **script debugger window**. Specifically, output from the *print* method of the \$ object appears in this debug output view.



The currently executing JavaScript source appears in the lower pane of the **script debugger window**. Double-clicking a line in this JavaScript source view sets or clears an unconditional breakpoint on that line. That is, if a breakpoint is in effect for that line, double-clicking it clears the breakpoint, and vice-versa. The line number display on the left part displays a red dot for all lines with a breakpoint.

If debugging is disabled in After Effects preferences, the user will see an error message but not the Debugger itself. This is the typical setup that will be used in situations where professional roles are divided between those writing and administering scripts (technical directors, system administrators, etc.) and those using them (the artist/animators). If you are writing and debugging your own scripts, you will want to enable the debugger.

## Controlling Code Execution in the Script Debugger Window

This section describes the buttons that control the execution of code when the Script Debugger window is active. Most of these buttons also provide a keyboard shortcut available as a Ctrl-key combination on Windows platforms or a Cmd-key combination on Mac OS platforms.



Resume  
Cmd-R (Mac OS)  
Ctrl-R (Windows)

Resume execution of the script with the script debugger window open. When the script terminates, the application closes the script debugger window automatically. Closing the debugger window manually also causes script execution to resume. This button is enabled when script execution is paused or stopped.



Pause  
Cmd-P (Mac OS)  
Ctrl-P (Windows)

Halt the currently executing script temporarily and reactivate the script debugger window. This button is enabled when a script is running.



Stop  
Cmd-K (Mac OS)  
Ctrl-K (Windows)

Stop execution of the script and generate a runtime error. This button is enabled when a script is running.



Step Over  
Cmd-S (Mac OS)  
Ctrl-S (Windows)

Halt after executing a single JavaScript statement in the script; if the statement calls a JavaScript function, execute the function in its entirety before stopping.



Step Into  
Cmd-T (Mac OS)  
Ctrl-T (Windows)

Halt after executing a single JavaScript statement in the script or after executing a single statement in any JavaScript function that the script calls.



Step Out  
Cmd-U (Mac OS)  
Ctrl-U (Windows)

When the debugger is paused within the body of a JavaScript function, clicking this button resumes script execution until the function returns. When paused outside the body of a function, clicking this button resumes script execution until the script terminates.



Script Breakpoints Display  
(no keyboard shortcut)

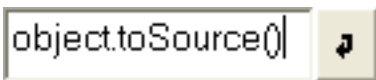
Clicking this button displays the Script Breakpoints Window shown below.

## Using the JavaScript command line entry field

You can use the Script Debugger's command line entry field to enter and execute JavaScript code interactively within a specified stack scope. Commands entered in this field execute with a timeout of one second. If a command takes longer than one second to execute, the script terminates and generates a timeout error.

### Command line entry field

Enter in this field a JavaScript statement to execute within the stack scope of the line highlighted in the Stack Trace view. When you've finished entering the JavaScript expression, you can execute it by clicking the command line entry button or pressing the Enter key. Click the button next to the field or press Enter to execute the JavaScript code in the command line entry field. The application executes the contents of the command line entry field within the stack scope of the line highlighted in the Stack Trace view.



Command line entry field

The command line entry field accepts any JavaScript code, making it very convenient to use for inspecting or changing the contents of variables.

NOTE: To list the contents of an object as if it were JavaScript source code, enter the `object.toSource()` command.

## Setting Breakpoints

You can set breakpoints in the debugger itself, by calling methods of the `$` object, or by defining them in your JavaScript code.

## Setting Breakpoints in the Script Debugger Window

When the Script Debugger window is active, you can double-click a line in the source view to set or clear a breakpoint at that line. Alternatively, you can click the Script Breakpoints Display button to display the Script Breakpoints window and set or clear breakpoints in this window as described in “Setting Breakpoints in the Script Breakpoints Window” and in “Clearing Breakpoints in the Script Breakpoints Window”.

## Setting Breakpoints in JavaScript Code

Adding the debugger statement to a script sets an unconditional breakpoint. For example, the following code causes the script to halt and display the script debug window as soon as it enters the `setupBox` function.

```
function setupBox(box) {  
    // break unconditionally at the next line  
    debugger;  
    box.width  = 48;  
    box.height = 48;  
    box.url    = "none";  
}
```

To execute a breakpoint in runtime code, call the `$.bp()` method, as shown in the following example:

```
function setupBox(box) {  
    box.width  = (box.width == undefined) ? $.bp() : 48;  
    box.height = (box.height == undefined) ? $.bp() : 48;  
    box.url    = (box.url == undefined) ? $.bp() : "none";  
}
```

This example breaks into the debugger if any of the width, height, or url attributes of the custom element are undefined. Of course, you wouldn't put `bp` method calls into production code — it's more appropriate for shipping code to set default values for undefined properties, as the previous example does.

## Script Breakpoints Window

This section describes the information and controls that the Script Breakpoints window provides. Display of the Script Breakpoints window is controlled by the Script Breakpoints button in the main Script Debugger Window.



Script Breakpoints button

This dialog displays all defined breakpoints. This dialog does not display:

- Breakpoints defined by the debugger statement in JavaScript code.
- Temporary breakpoints.

The Script Breakpoints window provides the following controls:

- The Line field contains the line number of the breakpoint.
- The Condition field may contain a Javascript expression to evaluate when the breakpoint is reached. If the expression evaluates to false, the breakpoint is not executed.
- Breakpoints set in this window persist across multiple executions of a script. When the application quits, or a script is reloaded, it removes all breakpoints.

### Setting Breakpoints in the Script Breakpoints Window

Take the following steps to set a breakpoint in the Script Breakpoints Window:

Click New to create a new breakpoint, or click the breakpoint that you wish to edit.

Enter a line number in the Line Number field, or change the existing line number.

Optionally, enter a condition such as (i>5) in the Condition field. This can be any valid JavaScript expression. If the result of evaluating the expression is true, the breakpoint activates.

## The Debugger Object (\$)

The \$ Object (Debugger Object) provides properties and methods you can use to debug your JavaScript code. For example, you can call its methods to set or clear breakpoints programmatically, or to change the language flavor of the script currently executing. It also provides properties that hold information about the version of the host platform's operating system.

NOTE: The \$ object is not a standard JavaScript object.

### Properties

<code>error</code>	Error	Retrieve the last runtime error. Reading this property returns an Error object containing information about the last runtime error.
<code>version</code>	String	Returns the version number of the JavaScript engine as a three-part number like e.g. "3.1.11". Read only.
<code>os</code>	String	Outputs the current operating system version. Read only.

### Debug output

```
write (text, ...);  
writeln (text, ...);
```

Write the given string to the Debug Output window. The `writeln` method appends a New Line character to its arguments.

### Parameters

<code>text</code>	String	All parameters are concatenated to a single string.
-------------------	--------	---



## Returns

None.

```
clearbp (scriptletName, line);
```

Clear a breakpoint. The breakpoint is defined by the name of the scriptlet or function and the line number. If the scriptlet name is the empty string or missing, the name of the currently executing scriptlet is used. If the line number is zero or not supplied, the current line number is used. Thus, the call \$.clearbp() without parameters clears a breakpoint at the current position.

The special string "NEXTCALL" as the scriptlet name causes the engine to clear a breakpoint at the next function call.

## Parameters

scriptletName	String	The name of the scriptlet where the breakpoint is to be cleared.
line	Number	The line number where the breakpoint is to be cleared.

## Returns

None.

```
bp([condition]);
```

Execute a breakpoint at the current position. Optionally, a condition may be supplied. The condition is a JavaScript expression string that is evaluated before the breakpoint is executed. The breakpoint is only executed if the expression returns true. If no condition is given, the use of the debugger statement is recommended instead as it is a more widely supported JavaScript standard statement.

## Parameters

condition	String	An optional Javascript expression string that is evaluated before the breakpoint is executed. The expression needs to evaluate to the equivalent of true in order to activate the breakpoint.
-----------	--------	---

## Returns

None.

## Other Methods

```
gc ()
```

Initiate a garbage collection. Garbage collection is the process by which the JavaScript interpreter cleans up memory it is no longer using. This is done automatically. Occasionally when debugging a script, it may be useful to call this process.

## Returns

None.

# Reference

## Introduction

This chapter lists and describes all syntax (keywords, statements, operators, classes, objects, methods, attributes, and global functions) recognized by the After Effects scripting engine.

Note that in ECMAScript and JavaScript, a named piece of data of a certain type is commonly referred to as a property. However, in After Effects we already have a separate definition of a “property”: it is a specific editable value within a layer. Therefore in this section we use the synonymous term “attribute” to refer to these same pieces of data.

## ECMAScript Language Specification

The After Effects Scripting engine supports the 3rd Edition of the ECMA-262 Standard, including its notational and lexical conventions, types, objects, expressions and statements.

For a complete listing of the keywords and operators included with ECMAScript, please refer to Ecma-262.pdf, available at:

<http://www.ecma-international.org/publications/standards/ECMA-262.HTM>

## Keywords and statement syntax

Table 5.1 lists and describes all keywords and statements recognized by the After Effects scripting engine.

Table 1      Keywords and Statement Syntax

Keyword/State- ment	Description
break	Standard JavaScript construct. Exit the currently executing loop.
continue	Standard JavaScript construct. Cease execution of the current loop iteration.
case	label used in a switch statement
default	label used in a switch statement when a case label is not found
do - while	Standard JavaScript construct. Similar to the while loop, except loop condition evaluation occurs at the end of the loop.
false	Literal representing boolean false.
for	Standard JavaScript loop construct.



Keyword/State- ment	Description
<code>for - in</code>	Standard JavaScript construct. Provides a way to easily loop through the properties of an object.
<code>function</code>	Used to define a function.
<code>if/if - else</code>	Standard JavaScript conditional constructs.
<code>new</code>	Standard JavaScript constructor statement.
<code>null</code>	Assigned to a variable, array element, or object property to indicate that it does not contain a legal value.
<code>return</code>	Standard JavaScript way of returning a value from a function or exiting a function.
<code>switch</code>	Standard JavaScript way of evaluating an expression and attempting to match the expression's value to a <code>case</code> label.
<code>this</code>	Standard JavaScript method of indicating the current object.
<code>true</code>	Literal representing boolean true.
<code>undefined</code>	Indicates that the variable, array element, or object property has not yet been assigned a value.
<code>var</code>	Standard JavaScript syntax used to declare a local variable.
<code>while</code>	Standard JavaScript construct. Similar to the <code>do - while</code> loop, except loop condition evaluation occurs at the beginning of the loop.
<code>with</code>	Standard JavaScript construct used to specify an object to use in ensuing statements.

## Operators

Table 5.2 lists and describes all operators recognized by the After Effects scripting engine. Table 5.3 shows the precedence and associativity for all operators.

Table 2 Description of Operators

Operators	Description
<code>new</code>	Allocate object.
<code>delete</code>	Deallocate object.
<code>typeof</code>	Returns data type.
<code>void</code>	Returns undefined value.
<code>.</code>	Structure member.
<code>[ ]</code>	Array element.
<code>( )</code>	Function call.

Operators	Description
++	Pre- or post-increment.
--	Pre- or post-decrement.
-	Unary negation or subtraction.
~	Bitwise NOT.
!	Logical NOT.
*	Multiply.
/	Divide.
%	Modulo division.
+	Add.
<<	Bitwise left shift.
>>	Bitwise right shift.
>>>	Unsigned bitwise right shift.
<	Less than.
<=	Less than or equal.
>	Greater than.
>=	Greater than or equal.
==	Equal.
!=	Not equal.
&	Bitwise AND.
^	Bitwise XOR.
	Bitwise OR.
&&	Logical AND.
	Logical OR.
? :	Conditional (ternary).
=	Assignment.
+=	Assignment with add operation.
-=	Assignment with subtract operation.
*=	Assignment with multiply operation.
/=	Assignment with divide operation.
%=	Assignment with modulo operation.

Operators	Description
<<=	Assignment with bitwise left shift operation.
>>=	Assignment with bitwise right shift operation.
>>>=	Assignment with bitwise right shift unsigned operation.
&=	Assignment with bitwise AND operation.
^=	Assignment with bitwise XOR operation.
=	Assignment with bitwise OR operation.
,	Multiple evaluation.

Table 3 Operator Precedence

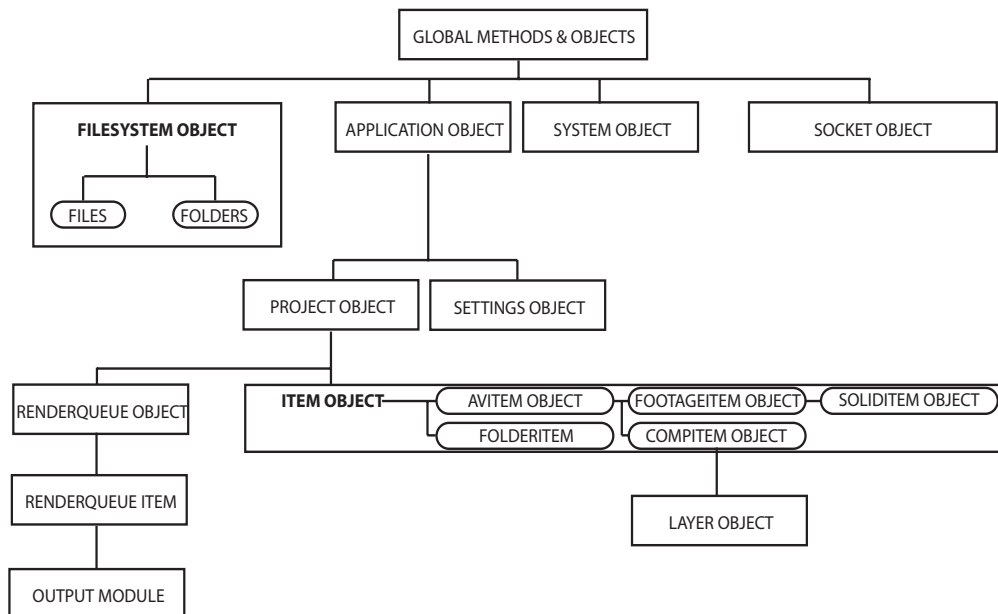
Operators (Listed from highest precedence —top row—to lowest)	Associativity
[ ], ( ), .	left to right
new, delete, -(unary negation), ~, !, typeof, void, ++, --	right to left
*, /, %	left to right
+, -(subtraction)	left to right
<<, >>, >>>	left to right
<, <=, >, >=	left to right
==, !=	left to right
&	left to right
^	left to right
	left to right
&&	left to right
	left to right
?:	right to left
=, /=, %=, <<=, >>=, >>>=, &=, ^=,  =, +=, -=, *=	right to left
,	left to right

## Reference for Objects, Methods, Attributes, and Globals

The remainder of this chapter lists and describes all predefined identifiers recognized by After Effects.

These extensions to ECMAScript are organized within a hierarchy of objects which corresponds closely to how they are organized in the application itself. As you look through this A-Z reference section, you can refer to the following diagram for an overview of where the various objects fall within the hierarchy.

Please make particular note of the fact that Item is an object with subclasses that inherit their attributes and methods and add more of their own.



This chart depicts the hierarchy of objects in After Effects scripting. Each object is described in detail in this chapter.

## Global Functions

This section describes globally available functions which are specific to After Effects. Any JavaScript object or function can call the functions in this section.

### Functions

<code>alert()</code>	see “ <code>alert()</code> Global Function” on page 23	pops up an alert dialog displaying a specified text string
<code>prompt()</code>	see “ <code>prompt()</code> Global Function” on page 74	opens a dialog box with a text field into which the user can enter a text string
<code>write()</code>	see “ <code>write()</code> Global Function” on page 86	writes output to the Info palette, with no line break added
<code>writeln()</code>	see “ <code>writeln()</code> Global Function” on page 87	writes output to the info palette, adding a line break at the end
<code>clearOutput()</code>	see “ <code>clearOutput()</code> Global Function” on page 32	clears the Info palette
<code>confirm()</code>	see “ <code>confirm()</code> Global Function” on page 34	prompts the user with a modal dialog and yes/no buttons which clear the dialog and return a boolean
<code>fileGetDialog()</code>	see “ <code>fileGetDialog()</code> Global Function” on page 46	presents the platform’s standard file > open dialog

<code>filePutDialog()</code>	see “filePutDialog() Global Function” on page 46	presents the platform’s standard file > save dialog
<code>folderGetDialog()</code>	see “folderGetDialog() Global Function” on page 59	displays a dialog in which the user can select a folder

## alert() Global Function

`alert(text)`

### Description

The Alert global function opens an alert dialog which can contain a text alert. The user then has the option of clicking “okay” to close the window.

### Parameters

<code>text</code>	text string that is displayed in the dialog, which can display up to 240 characters
-------------------	---

### Example

```
alert ( "Your pants are on fire.");
```

## Application Object

`app`

### Description

The application (app) global object enables access to data and functionality within the After Effects application. Attributes of the Application object provide access to specific objects within After Effects. Methods of the Application object can create documents, open existing documents, control Watch Folder mode, purge memory and quit the After Effects application. When the After Effects application quits, it closes the open project, prompting the user to save or discard changes as necessary, and creates a project file as necessary.

### Methods

<code>newProject()</code>	see “Application new-Project() Method” on page 28	opens a new project in After Effects
<code>open()</code>	see “Application open() Method” on page 29	opens a project or an open project dialog
<code>quit()</code>	see “Application quit() Method” on page 31	quits the application
<code>watchFolder()</code>	see “Application watch-Folder() Method” on page 31	starts watch folder mode. Does not return until watch folder mode is turned off

<code>pauseWatchFolder()</code>	see "Application pause-WatchFolder() Method" on page 29	pauses a current watch folder process
<code>endWatchFolder()</code>	see "Application end-WatchFolder() Method" on page 26	ends a current watch folder process
<code>purge()</code>	see "Application purge() Method" on page 30	purges a targeted type of cached information (replicates Purge options in the Edit menu)
<code>beginUndoGroup()</code>	see "Application beginUndoGroup() Method" on page 25	groups the actions that follow it into a single undoable step
<code>endUndoGroup()</code>	see "Application endUndoGroup() Method" on page 25	ends an undo group; needed only when one script contains more than one undo group

**Attributes**

<code>project</code>	see "Project Object" on page 66	an instance of the After Effect Project and all of its associated methods & attributes
<code>isRenderEngine</code>	see "Application isRenderEngine Attribute" on page 27	Identifies whether the local After Effects application is installed as a render engine
<code>language</code>	see "Application language Attribute" on page 27	identifies the language in which the application is running
<code>version</code>	see "Application version Attribute" on page 32	identifies the version number of the After Effects application
<code>isProfessionalVersion</code>	see "Application isProfessionalVersion Attribute" on page 26	identifies if the After Effects version is the Professional Version
<code>serialNumber</code>	see "Application serialNumber Attribute" on page 31	identifies the serial number of the After Effects installation
<code>registeredName</code>	see "Application registeredName Attribute" on page 30	identifies the name to which the After Effects installation is registered
<code>registeredCompany</code>	see "Application registeredCompany Attribute" on page 30	identifies the company to which the After Effects installation is registered
<code>isWatchFolder</code>	see "Application isWatchFolder Attribute" on page 27	boolean which returns true when the local application is running in Watch Folder mode



<code>onError</code>	see “Application <code>onError</code> Attribute” on page 28	a callback which is called when an error occurs in the application
<code>settings</code>	see “Settings Object” on page 83	calls settings within After Effects that can be set via scripting
<code>exitCode</code>	see “Application <code>exitCode</code> Attribute” on page 26	used only when executing script externally (i.e. from a command line or AppleScript). Set to zero, indicates no error occurred; set to a positive number, indicates an error occurred while running the script.

## Application `beginUndoGroup()` Method

`app.beginUndoGroup(undoString)`

### Description

An undo group allows a script to logically group all of its actions as a single undoable action (for use with the Edit Undo/Redo menu items). Should be used in conjunction with the `application.endUndoGroup()` method.

Please note that `beginUndoGroup()` and `endUndoGroup()` pairs can be nested. Groups within groups become part of the larger group, and will undo correctly. In such cases, the names of inner groups are ignored.

### Parameters

<code>undoString</code>	(mandatory) the text that will appear for the Undo command in the Edit menu (i.e. “Undo <code>undoString</code> ”)
-------------------------	--

### See also

“Application `endUndoGroup()` Method” on page 25

## Application `endUndoGroup()` Method

`app.endUndoGroup()`

### Description

This ends the undo group begun with the `app.beginUndoGroup()` method. You can use this method to place an end to an undo group in the middle of a script, should you wish to use more than one undo group for a single script.

If you are using only a single undo group for a given script, you do not need to use this method; in its absence at the end of a script, the system will close the undo group automatically.

Calling this method without having set a `beginUndoGroup()` method yields an error.

### Parameters

None

**Returns**

None.

**See also**

“Application beginUndoGroup() Method” on page 25

## Application endWatchFolder() Method

```
app.endWatchFolder()
```

**Description**

The `endWatchFolder()` method ends watch folder mode.

**Parameters**

None

**See also**

“Application watchFolder() Method” on page 31

“Application pauseWatchFolder() Method” on page 29

## Application exitCode Attribute

```
app.exitCode
```

**Description**

The `exitCode` attribute is used only when executing a script from outside After Effects (i.e. from a command line or AppleScript).

On Mac & Windows, the `exitCode` is set to 0 (EXIT\_SUCCESS) at the beginning of each script evaluation. In the event of an error while the script is running, it will be set to a positive integer.

**Type**

Integer; read/write.

**Example**

```
app.exitCode = 2; //on quit, if value is 2, no error has occurred
```

## Application isProfessionalVersion Attribute

```
app.isProfessionalVersion
```

**Description**

The `isProfessionalVersion` attribute is a boolean used to determine if the locally installed After Effects application is the Standard or Professional version.

**Type**

Boolean; read-only.

### Example

```
var PB = app.isProductionBundle;  
alert("It is " + PB + " that you are running the Production  
Bundle.");
```

## Application isRenderEngine Attribute

*app.isRenderEngine*

### Description

The isRenderEngine attribute is a boolean used to determine if an installation of After Effects is a Render Engine only installation.

### Type

Boolean; read-only.

## Application isWatchFolder Attribute

*app.isWatchFolder*

### Description

The isWatchFolder attribute is a boolean used to determine if the Watch Folder dialog is currently displayed (and the application is currently watching a folder for rendering). This returns true when the Watch Folder dialog is open.

### Type

Boolean; read-only.

## Application language Attribute

*app.language*

### Description

The language attribute indicates in which language After Effects is running. The codes for the language attribute are:

- Language.ENGLISH
- Language.FRENCH
- Language.GERMAN
- Language.JAPANESE

### Type

Language enumerated type (listed above).

### Example

```
var lang = app.language;  
if (lang == Language.JAPANESE){  
  alert("After Effects is running in Japanese.");  
}
```

```
else if (lang == Language.ENGLISH){
  alert("After Effects is running in English.");
}
else if (lang == Language.FRENCH){
  alert("After Effects is running in French.");
}
else{
  alert("After Effects is running in German.");
}
```

## Application `newProject()` Method

`app.newProject()`

### Description

The `newProject` method opens a new project in After Effects, replicating the File > New > New Project menu command. If a project is already open and has been edited the user will be prompted to save.

Use `app.project.close(CloseOptions.DO_NOT_SAVE_CHANGES)` to close an open project before opening a new one.

### Parameters

None.

### Returns

Project object; null if the user cancels a Save dialog in response to having an open project that has been edited since the last save.

### Example

```
app.project.close(CloseOptions.DO_NOT_SAVE_CHANGES);
app.newProject();
```

### See also

“Project `close()` Method” on page 68

## Application `onError` Attribute

`app.onError`

### Description

The `onError` attribute takes a function to perform an action when an error occurs. By creating a function and assigning it to `onError`, you can respond to the error systematically, e.g. close and restart the application, noting the error in a log file if it occurred during rendering.

### Type

Function that takes a string, or null if no function is assigned.

### Example

```
function err(errString) {
```

```
        alert(errString);  
    )
```

```
app.onError = err
```

## Application open() Method

```
app.open()
```

```
app.open(file)
```

### Description

The `open()` method opens a project. If the file parameter is null (i.e. if no argument is used) the user will be presented with a dialog to select and open a file.

### Parameters

file	(Optional) File object being opened
------	-------------------------------------

### Returns

Project object (the file specified as a parameter), or null if the user cancels the Open dialog.

### Example

```
var my_file = new File("../my_folder/my_test.aep");  
if (my_file.exists){  
    new_project = app.open(my_file);  
    if (new_project){  
        alert(new_project.file.name);  
    }  
}
```

## Application pauseWatchFolder() Method

```
app.pauseWatchFolder(pause)
```

### Description

The `pauseWatchFolder()` method pauses searching the target directory for render items.

### Parameters

pause	boolean (paused - true or false)
-------	----------------------------------

### See also

“Application watchFolder() Method” on page 31

“Application endWatchFolder() Method” on page 26

## Application purge() Method

`app.purge(target)`

### Description

The purge method replicates the functionality and target options of the Purge options within the Edit menu. The target parameter contains the area of memory to be purged; the options for target are listed as enumerated variables below.

### Parameters

target	the type of elements to purge from memory; use one of Enumerated Types below
--------	--

### Enumerated Types

PurgeTarget.ALL_CACHES	purges all data that After Effects has cached to physical memory
PurgeTarget.UNDO_CACHES	purges all data saved in the undo cache
PurgeTarget.SNAPSHOT_CACHES	purges all data cached as comp/layer snapshots
PurgeTarget.IMAGE_CACHES	purges all saved image data

## Application registeredCompany Attribute

`app.registeredCompany`

### Description

Text string; name(if any) that the user of the application entered as the registered company at the time of installation.

### Type

Text string; read-only.

### Example

```
var company = app.registeredCompany;  
alert("Your company name is " + company + ".");
```

## Application registeredName Attribute

`app.registeredName`

### Description

The registeredName attribute contains the text string that the user of the application entered for the registered name at the time of installation.

### Type

Text string; read-only.

### Example

```
var userName = app.registeredName;  
confirm("Are you " + userName + "?");
```

## Application serialNumber Attribute

*app.serialNumber*

### Description

The serialNumber attribute contains an alphanumeric string which is the serial number of the installed version of After Effects.

### Type

String; read-only.

### Example

```
var serial = app.serialNumber;  
alert("This copy is serial number " + serial);
```

## Application quit() Method

*app.quit()*

### Description

The quit method quits the application.

### Parameters

None.

## Application watchFolder() Method

*app.watchFolder(folder\_object\_to\_watch)*

### Description

The *watchFolder()* method starts a watch folder (network rendering) process pointed at a specified folder.

### Parameters

<i>folder_object_to_watch</i>	The Folder object to be watched
-------------------------------	---------------------------------

### Example

```
var theFolder = new Folder("c:\\\\tool");  
app.watchFolder(theFolder);
```

### See also

"Application endWatchFolder() Method" on page 26

"Application pauseWatchFolder() Method" on page 29

## Application version Attribute

*app.version*

### Description

The version attribute returns an alphanumerical string indicating which version of After Effects is running.

### Type

String; read-only.

### Example

```
var ver = app.version;  
alert("This machine is running version " + ver + " of After  
Effects.");
```

## AVItem Object

*app.project.item(index)*

### Description

The AVItem object provides access to attributes and methods of audio/visual files imported into After Effects.

### (Attribute from “Item Object” on page 63)

name	see “Item name Attribute” on page 63	the name of the object as shown in the Project window
------	---	--

## clearOutput() Global Function

*clearOutput()*

### Description

The clearOutput global function clears the output in the info palette.

### Parameters

None.

### Example

```
clearOutput();
```



# Collection Object

## Description

A Collection object acts like an array that provides access to its elements by index. Like an array, a collection associates a set of objects or values as a logical group and provides random access to them. However, most collection objects are read-only. You do not assign objects to them yourself—their contents update automatically as objects are created or deleted.

The index numbering of a collection starts with 1, not 0.

## Objects

Item Collection	see “Item Collection” on page 62	a collection of all of the items (imported files, folders, solids, etc.) found in the Project window
OutputModule Collection	see “OutputModule Collection” on page 64	contains all of the OutputModule items in the project
RenderQueueItem Collection	see “RenderQueueItem Collection” on page 77	contains all of the RenderQueue items in the project

## Attributes

length	the number of objects in the collection (applies to all collections)
--------	--

## Methods

[ ]	retrieves an object or objects in the collection via its index number
-----	---

# CompItem Object

*app.project.item(index)*

## Description

The CompItem object provides access to attributes and methods of Compositions. These are accessed via their index number.

## Attributes

frameDuration	see “CompItem frameDuration Attribute” on page 34	the duration of a single frame in seconds. This is the inverse of the framerate.
workAreaStart	see “CompItem workAreaStart Attribute” on page 34	the work area start time (in seconds)
workAreaDuration	see “CompItem workAreaDuration Attribute” on page 34	the work area duration (in seconds)

(Attributes inherited from “Item Object” on page 63 & “AVItem Object” on page 32)

name	see “Item name Attribute” on page 63	the name of the object as shown in the Project window
------	--------------------------------------	---

## CompItem frameDuration Attribute

```
app.project.item(index).frameDuration
```

### Description

The frameDuration attribute returns the duration of a frame, in seconds. This is the inverse of the framerate (or frames per second). This attribute is read-only.

### Type

Floating-point value; read-only.

### See also

“CompItem workAreaDuration Attribute” on page 34

## CompItem workAreaDuration Attribute

```
app.project.item(index).workAreaDuration
```

### Description

The workAreaDuration attribute determines the duration, in seconds, of the work area. This value is the difference of the start point time of the Composition work area and the end point.

### Type

Floating-point value; time, in seconds. Read/write.

## CompItem workAreaStart Attribute

```
app.project.item(index).workAreaStart
```

### Description

The workAreaStart attribute determines the time, in seconds, where the Composition work area begins.

### Type

Floating-point value; time, in seconds. Read/write.

## confirm() Global Function

```
confirm(text)
```

### Description

The Confirm global function prompts the user with a modal dialog and yes/no buttons which clear the dialog. These return a boolean; true if yes, false if no.

## Parameters

text	text string; Mac UI can display 256 characters, Windows, 30 characters
------	--

## Returns

Boolean.

## Example

```
var shouldAdd = confirm("Add to Render Queue?");
if (shouldAdd == "true"){
    proj.renderQueue.items.add(myCompItem);
}
```

# File Class

## Description

The File class contains methods and attributes common to File objects. A File object corresponds to a disk file.

Also included in this class are all attributes and methods within the FileSystem class, as those apply to Files as well as Folders.

Note that the difference between the File Class and File Object is that the class attributes and methods require no specific instance of a File, whereas class methods and attributes do.

### (Class Attributes inherited from “FileSystem Class” on page 47)

fs	see “FileSystem fs Class Attribute” on page 51	The name of the file system; read-only.
----	--	---

## Methods

File() new File()	see “File() Class Method” on page 36	constructs a new File object.
openDialog()	see “File openDialog() Class Method” on page 41	Opens the built-in OS dialog to select an existing file to open.
saveDialog()	see “File saveDialog() Class Method” on page 43	Opens the built-in OS dialog to select a file name to save a file into.

### (Class Methods inherited from “FileSystem Class” on page 47)

decode()	see “FileSystem decode() Class Method” on page 50	decodes the input string from UTF-8
encode()	see “FileSystem encode() Class Method” on page 50	encodes the input string in UTF-8

## File() Class Method

```
File(path)
```

```
new File(path)
```

### Description

This function constructs a new File object. If the given path name refers to an already existing folder, a Folder object is returned instead.

The CRLF sequence is preset to the system default, and the encoding is preset to the default system encoding.

### Parameters

Path, expressed as a string. If missing, a temporary name is generated.

### Returns

File (or Folder if path refers to an existing folder).

## File Object

```
File("path")
```

### Description

The File object contains methods and attributes common to File objects. A Folder object corresponds to a directory or folder.

Also included in this object are all attributes and methods within the FileSystem object, as those apply to Files as well as Folders.

### Attributes

creator	see "File creator Attribute" on page 39	The Macintosh file creator as a four-character string.
encoding	see "File encoding Attribute" on page 39	Gets or sets the encoding for subsequent read/write operations.
eof	see "File eof Attribute" on page 39	Has the value true if a read attempt caused the current position to be behind the end of the file.
hidden	see "File hidden Attribute" on page 40	Set to true if the file is invisible.
length	see "File length Attribute" on page 40	The size of the file in bytes.
lineFeed	see "File lineFeed Attribute" on page 40	The way line feed characters are written.
readonly	see "File readonly Attribute" on page 43	When set, prevents the file from being altered or deleted.
type	see "File type Attribute" on page 45	The Macintosh file type as a four-character string.

**(Attributes inherited from “FileSystem Object” on page 47)**

absoluteURI	see “FileSystem absoluteURI Attribute” on page 49	The full path name for the object in URI notation.
alias	see “FileSystem alias Attribute” on page 49	Returns true if the object refers to a file system alias.
created	see “FileSystem created Attribute” on page 49	The creation date of the object.
error	see “FileSystem error Attribute” on page 50	Contains a message describing the last file system error.
exists	see “FileSystem exists Attribute” on page 51	Returns true if the path name of this object refers to an actually existing file or folder.
fsName	see “FileSystem fsName Attribute” on page 51	The file-system specific name of the object as a full path name.
modified	see “FileSystem modified Attribute” on page 52	The date of the object's last modification.
name	see “FileSystem name Attribute” on page 52	The name of the object without the path specification.
parent	see “FileSystem parent Attribute” on page 52	The folder object containing this object.
path	see “FileSystem path Attribute” on page 53	The path portion of the absolute URI.
relativeURI	see “FileSystem relativeURI Attribute” on page 53	The path name for the object in URI notation, relative to the current folder.

**Methods**

close( )	see “File close() Method” on page 38	Closes the open file.
copy( )	see “File copy() Method” on page 38	Copies the file to the given location.
open( )	see “File open() Method” on page 40	Open the file for subsequent read/write operations.
read( )	see “File read() Method” on page 42	Read the contents of the file from the current position on.
readch( )	see “File readch() Method” on page 42	Read one single text character.
readln( )	see “File readln() Method” on page 43	Read one line of text.

<code>seek()</code>	see "File seek() Method" on page 44	Seek to a certain position in the file.
<code>tell()</code>	see "File tell() Method" on page 44	Returns the current position in the file as a an offset in bytes.
<code>write()</code>	see "File write() Method" on page 45	Write the given string to the file.
<code>writeln()</code>	see "File writeln() Method" on page 45	Write the given string to the file and append a line feed sequence.

**(Methods inherited from "FileSystem Object" on page 47)**

<code>getRelativeURI()</code>	see "FileSystem getRelativeURI() Method" on page 51	Calculate and return the relative URI, given a base path, in URI notation.
<code>remove()</code>	see "FileSystem remove() Method" on page 53	Delete the file or folder that this object represents.
<code>rename()</code>	see "FileSystem rename() Method" on page 53	Rename the object to the new name.
<code>resolve()</code>	see "FileSystem resolve() Method" on page 54	Attempt to resolve the file system alias that this object points to.

## File close() Method

*File(path).close()*

### Description

The `close()` method closes the open file. The return value is true if the file was closed, false on I/O errors.

### Parameters

None.

### Returns

Boolean.

## File copy() Method

*File(path).copy(target)*

### Description

The `copy()` method copies the file to the given location.

You can supply an URI path name as well as another File object. If there is a file at the target location, it is overwritten.

The method returns true if the copy was successful, false otherwise. The method resolves any aliases to find the source file.

## Parameters

target	File object or String specifying the target location
--------	--

## Returns

Boolean.

## File creator Attribute

*File(path).creator*

### Description

The creator attribute is the Macintosh file creator as a four-character string. On Windows, the return value is always "????".

### Type

String; read-only.

## File encoding Attribute

*File(path).encoding*

### Description

The encoding attribute gets or sets the encoding for subsequent read/write operations.

The encoding is one of several predefined constants that follow the common Internet encoding names. Valid names are UCS-2, X-SJIS, ISO-8851-9, ASCII or the like.

A special encoder, BINARY, is used to read binary files. This encoder 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. See "Encoding Names" on page 113 for a list of encodings. If an unrecognized encoding is used, the encoding reverts to the system default encoding.

### Type

String; read/write.

## File eof Attribute

*File(path).eof*

### Description

The File eof attribute has the value true if a read attempt caused the current position to be past the end of the file.

If the file is not open, the value is true.

### Type

Boolean; read-only.

## File hidden Attribute

*File(path).hidden*

### Description

The File hidden attribute has the value true if the file is invisible. Assigning a Boolean value sets or clears this attribute.

### Type

Boolean; read/write.

## File length Attribute

*File(path).length*

### Description

The File length attribute is size of the file in bytes. When setting the file size, the file must not be open.

### Type

Number; read-only.

## File lineFeed Attribute

*File(path).lineFeed*

### Description

The File lineFeed attribute determines the way line feed characters are written. This can be one of the three values macintosh, unix or windows (actually, only the first character is interpreted).

### Type

String (one of: macintosh, unix, windows); read/write.

## File open() Method

*File(path).open(mode, type, creator)*

### Description

The File open() method opens the file for subsequent read/write operations. The type and creator arguments are optional and Macintosh specific; they specify the file type and creator as two four-character strings. They are used if the file is newly created. On other platforms, they are ignored.



When `open()` is used to open a file for read access, 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`, `UCS4-LE` or `UTF-8`. If the marker character cannot be found, it checks for zero bytes at the current location and makes an assumption about one of the above formats (except for `UTF-8`). If everything fails, the encoding property is set to the system encoding. The method resolves any aliases to find the file.

You should be careful if you try to open 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 may destroy your data!

The return value is true if the file has been opened successfully, false otherwise.

### Parameters

<code>mode</code>	one of <code>r</code> , <code>w</code> or <code>e</code> :  <code>r</code> (read) Opens for reading. If the file does not exist or cannot be found the call fails.  <code>w</code> (write) Opens an empty file for writing. If the file exists, its contents are destroyed.  <code>e</code> (edit) Opens an existing file for reading and writing.
<code>type</code>	The Macintosh file type; a four-byte character string; ignored on non-Macintosh operating systems.
<code>creator</code>	The Macintosh file creator; a four-byte character string; ignored on non-Macintosh operating systems.

### Returns

Boolean.

## File.openDialog() Class Method

*File.openDialog(prompt, select)*

### Description

The `File.openDialog` class method presents the File > Open dialog that is standard for the platform on which After Effects is running. This method overlaps somewhat with the easier to use `fileGetDialog()` global function.

### Parameters

<code>prompt</code>	An optional prompt (expressed as a string) that is displayed as part of the dialog if the dialog permits the display of an additional message.
<code>select</code>	This argument allows the pre-selection of the files that the dialog displays. Unfortunately, this argument is different on the Macintosh and on Windows.

<i>select</i> (Win)	Windows selection string is actually a list of file types with explanative text. This list is displayed in the bottom of the dialog as a drop-down list box so the user can select which types of files to display. The elements of this list are separated by commas. Each element starts with the descriptive text, followed by a colon and the file search masks for this text. Again, each search mask is separated by a semicolon. A Selection list that allowed the selection of all text files (*.TXT and *.DOC) or all files would look like this:  Text Files:*.TXT;*.DOC,All files:*.  A single asterisk character is a placeholder for all files.
<i>select</i> (Mac)	On the Macintosh, the optional second argument is a callback function. This function takes one argument, which is a File object. When the dialog is set up, it calls this callback function for each file that is about to be displayed. If the function returns anything else than true, the file is not displayed. This is only true for the openFileDialog() method, the saveDialog() method ignores this callback method.

**Returns**

File object, or null if user cancels the dialog.

**See also**

"fileGetDialog() Global Function" on page 46

## File read() Method

*File(path).read(chars)*

**Description**

The File read() method reads the contents of the file from the current position on. Returns a string that contains up to the number of characters that were supposed to be read.

**Parameters**

chars	The number of characters to read, expressed as an integer. If the number of characters to read is not supplied, the entire file is read in one big chunk, starting at the current position. If the file is encoded, multiple bytes may be read to create single Unicode characters.
-------	---

**Returns**

String.

## File readch() Method

*File(path).readch()*

**Description**

The File readch() method reads one single text character. Line feeds are recognized as CR, LF, CRLF or LFCR pairs. If the file is encoded, multiple bytes may be read to create single Unicode characters.

**Parameters**

None.

**Returns**

String.

## File readln() Method

*File(path).readln()*

**Description**

The File readch() method reads one line of text. Line feeds are recognized as CR, LF, CRLF or LFCR pairs. If the file is encoded, multiple bytes may be read to create single Unicode characters.

**Parameters**

None.

**Returns**

String.

## File readonly Attribute

*File(path).readonly*

**Description**

The File readonly attribute, when set, prevents the file from being altered or deleted.

**Type**

Boolean; read/write.

## File saveDialog() Class Method

*File.saveDialog(prompt, select)*

**Description**

The File.saveDialog class method presents the File > Save dialog that is standard for the platform on which After Effects is running. This method overlaps somewhat with the easier to use filePutDialog() global function.

**Parameters**

prompt	An optional prompt (expressed as a string) that is displayed as part of the dialog if the dialog permits the display of an additional message.
select	This argument allows the pre-selection of the files that the dialog displays. Unfortunately, this argument is different on the Macintosh and on Windows.

<code>select (Win)</code>	<p>Windows selection string is actually a list of file types with explanative text. This list is displayed in the bottom of the dialog as a drop-down list box so the user can select which types of files to display. The elements of this list are separated by commas. Each element starts with the descriptive text, followed by a colon and the file search masks for this text. Again, each search mask is separated by a semicolon. A Selection list that allowed the selection of all text files (*.TXT and *.DOC) or all files would look like this:</p> <p>Text Files:*.TXT;*.DOC, All files:*</p> <p>A single asterisk character is a placeholder for all files.</p>
<code>select (Mac)</code>	<p>On the Macintosh, the optional second argument is a callback function. This function takes one argument, which is a File object. When the dialog is set up, it calls this callback function for each file that is about to be displayed. If the function returns anything else than true, the file is not displayed. This is only true for the <code>openDialog()</code> method, the <code>saveDialog()</code> method ignores this callback method.</p>

**Returns**

File object, or null if user cancels the dialog.

**See also**

“filePutDialog() Global Function” on page 46

## File seek() Method

*File(path).seek(pos, mode)*

**Description**

The File seek() method seeks to a certain position in the file. This method does not permit seeking to positions less than 0 or greater than the current file size.

**Parameters**

<code>pos</code>	The new current position inside the file as an offset in bytes (an integer), dependent on the seek mode.
<code>mode</code>	The seek mode (0 = seek to absolute position, 1 = seek relative to the current position, 2 = seek backwards from the end of the file).

**Returns**

Boolean; true if the position was changed.

## File tell() Method

*File(path).tell()*

**Description**

The File tell() method returns the current position in the file as an offset in bytes.

**Parameters**

None.

**Returns**

Integer.

## File type Attribute

*File(path).type*

**Description**

The File type attribute holds the Macintosh file type as a four-character string.

On the Macintosh, the file type is returned. On Windows, "appl" is returned for .EXE files, "shlb" for .DLLs and "TEXT" for any other file. If the file does not exist, the file type is "????".

**Type**

Boolean; read-only.

## File write() Method

*File(path).write(text, ...)*

**Description**

The File write() method writes a given string to the file. The parameters of this function are concatenated to a single string. Returns true on success.

For encoded files, writing a single Unicode character may result in multiple bytes being written. Take care not to write to a file that is open in another application or object. This may cause loss of data, since a second write issued from another File object may overwrite existing data.

**Parameters**

text	A string or set of strings. All arguments are concatenated to form the string to be written.
------	--

**Returns**

Boolean.

## File writeln() Method

*File(path).writeln(text, ...)*

**Description**

The File writeln() method writes a given string to the file. The parameters of this function are concatenated to a single string, and a Line Feed sequence is appended. Returns true on success.

If the file is encoded, multiple bytes may be read to create single Unicode characters.

### Parameters

<code>text</code>	A string or set of strings. All arguments are concatenated to form the string to be written.
-------------------	--

### Returns

Boolean.

## fileGetDialog() Global Function

`fileGetDialog(prompt, typeList)`

### Description

The fileGetDialog global function presents the File > Open dialog that is standard for the platform on which After Effects is running.

The typeList is a semicolon-separated list of four-character Mac OS file types followed by Windows file extensions. For example, a value of "EggP aep" for this argument specifies that the file open dialog is to display After Effects project items only; other file types will be grayed out.

### Parameters

<code>prompt</code>	message which displays on the title bar of the dialog; truncated if too long
<code>typeList</code>	a platform-specific value indicating a list of file types to display.

### Returns

File object, or null if the user cancels the dialog.

## filePutDialog() Global Function

`filePutDialog(prompt, default, type)`

### Description

The filePutDialog global function presents the File > Save dialog that is standard for the platform on which After Effects is running.

### Parameters

<code>prompt</code>	message which displays on the title bar of the dialog; truncated if too long
<code>default</code>	default file name to display in the file-saving dialog; this value must observe the file-naming conventions of the platform on which After Effects is running
<code>type</code>	specified file type

### Returns

File object, or null if the user cancels the dialog.

## FileSystem Class

File.

Folder.

### Description

The FileSystem class contains methods and attributes common to both File and Folder objects. A File object corresponds to a disk file, while a Folder object matches a directory or folder.

This attribute and methods differ from those found under the FileSystemObject in that they can be applied without referring to a particular instance of a file or folder.

### Class Attributes

fs	see "FileSystem fs Class Attribute" on page 51	The name of the files system; read-only.
----	--	--

### Class Methods

decode ( )	see "FileSystem alias Attribute" on page 49	decodes the input string from UTF-8
encode ( )	see "FileSystem encode() Class Method" on page 50	encodes the input string in UTF-8

## FileSystem Object

File("path").

Folder("path").

### Description

The FileSystem object contains methods and attributes common to both File and Folder objects. A File object corresponds to a disk file, while a Folder object matches a directory or folder. "FileSystem" is a name used to refer to both Folders and Files.

These attributes and methods differ from those found under the FileSystem Class in that they cannot be applied without referring to a particular instance of a file or folder, expressed as a path to that file or folder.

You can use absolute path names and relative path names. Absolute path names start with one or two slash characters. These path names describe the full path from a root directory down to a file or folder. Relative path names start from a known location, the current directory. A relative path name starts either with a directory name or with one of the special names "." and "..". The name "." refers to the current directory, and the name ".." refers to the parent directory. The slash character is used to separate path elements. Special characters are encoded in UTF-8 notation.

The FileSystem objects support a common convention. A volume name may be the first part of an absolute path. The objects know where to look for the volume names on the Macintosh and Windows and they translate the volume names accordingly.

A path name can also start with the tilde “~” character. This character stands for the user’s home directory (on Mac). On Windows, a directory with the environment variable HOME or, failing that, the desktop is used as a home directory.

The following table illustrates how the root element of a full path name is used on different file systems. In these examples, the current drive is C: on Windows and “Macintosh HD” on the Macintosh.

URI	Windows Name	Macintosh Name
/d/dir/name.ext	D:\dir\name.ext	Macintosh HD:d:dir:name.ext
/Macintosh HD/dir/name.ext	C:\Macintosh HD\dir\name.ext	Macintosh HD:dir:name.ext

Thus if you have to use a script with URI notation on both Mac and Windows, try to use relative path names, or try to originate your path names from the home directory. If that is not possible, it is recommended that you work with Mac OS X aliases andn UNC names on Windows, and store files on a machine that is remote to the Windows machine on which the script is running.

### Attributes

absoluteURI	see “FileSystem absoluteURI Attribute” on page 49	The full path name for the object in URI notation.
alias	see “FileSystem alias Attribute” on page 49	Returns true if the object refers to a file system alias.
created	see “FileSystem created Attribute” on page 49	The creation date of the object.
error	see “FileSystem error Attribute” on page 50	Contains a message describing the last file system error.
exists	see “FileSystem exists Attribute” on page 51	Returns true if the path name of this object refers to an actually existing file or folder.
fsName	see “FileSystem fsName Attribute” on page 51	The file-system specific name of the object as a full path name.
modified	see “FileSystem modified Attribute” on page 52	The date of the object's last modification.
name	see “FileSystem name Attribute” on page 52	The name of the object without the path specification.
parent	see “FileSystem parent Attribute” on page 52	The folder object containing this object.
path	see “FileSystem path Attribute” on page 53	The path portion of the absolute URI.
relativeURI	see “FileSystem relativeURI Attribute” on page 53	The path name for the object in URI notation, relative to the current folder.



## Methods

<code>getRelativeURI()</code>	see "FileSystem getRelativeURI() Method" on page 51	Calculate and return the relative URI, given a base path, in URI notation.
<code>remove()</code>	see "FileSystem remove() Method" on page 53	Delete the file or folder that this object represents.
<code>rename()</code>	see "FileSystem rename() Method" on page 53	Rename the object to the new name.
<code>resolve()</code>	see "FileSystem resolve() Method" on page 54	Attempt to resolve the file system alias that this object points to.

## FileSystem absoluteURI Attribute

*File(path).absoluteURI*

*Folder(path).absoluteURI*

### Description

The absoluteURI attribute of File or Folder is the full path name for the object in URI notation.

### Type

String; read-only.

## FileSystem alias Attribute

*File(path).alias*

*Folder(path).alias*

### Description

The alias attribute of File or Folder returns true if the object refers to a file system alias.

### Type

Boolean; read-only.

## FileSystem created Attribute

*File(path).created*

*Folder(path).created*

### Description

The created attribute of File or Folder is the creation date of the object. If the object does not refer to a folder or file on the disk, the value is null.

### Type

Date, or null if the object does not refer to a file or folder on disk; read-only.

## FileSystem decode() Class Method

*File.decode(string)*

*Folder.decode(string)*

### Description

The decode() class method of File or Folder decodes escaped characters and then interprets them as UTF-8. More information on the encoding standard can be found in the RFC 2396 document (search for this online).

### Parameters

<code>string</code>	the string to be decoded.
---------------------	---------------------------

### Returns

String.

### See also

"FileSystem encode() Class Method" on page 50

## FileSystem encode() Class Method

*File.encode(string)*

*Folder.encode(string)*

### Description

The encode() class method of File or Folder converts the input string to UTF-8 and then encodes it such that all characters are usable in a URI (or URL). More information on the encoding standard can be found in the RFC 2396 document (search for this online).

### Parameters

<code>string</code>	the string to be encoded.
---------------------	---------------------------

### Returns

String.

### See also

"FileSystem alias Attribute" on page 49

## FileSystem error Attribute

*File(path).error*

*Folder(path).error*

### Description

The error attribute of File or Folder contains a message describing the last file system error. Setting this value clears any error message and resets the error bit for opened files.

**Type**

Boolean; read/write (writing resets the error bit).

## FileSystem exists Attribute

*File(path).exists*

*Folder(path).exists*

**Description**

The exists attribute of File or Folder returns true if the path name of this object refers to an already existing file or folder.

**Type**

Boolean; read-only.

## FileSystem fs Class Attribute

*File.fs*

*Folder.fs*

**Description**

The fs class attribute of File or Folder holds the name of the file system (operating system). Possible values are "Windows" or "Macintosh".

**Type**

String; read-only.

**Example**

```
write("The local file system is " + File.fs);
```

## FileSystem fsName Attribute

*File(path).fsName*

*Folder(path).fsName*

**Description**

The fsName attribute of File or Folder is the file-system specific name of that object as a full path name.

**Type**

String; read-only.

## FileSystem getRelativeURI() Method

*File(path).getRelativeURI(string)*

*Folder(path).getRelativeURI(string)*

### Description

The `getRelativeURI()` method of `File` or `Folder` calculates and returns the relative URI, given a base path, in URI notation. If the base path is omitted, the path of the current folder is assumed.

### Parameters

<code>string</code>	the base path, in URI notation
---------------------	--------------------------------

### Returns

String.

## FileSystem modified Attribute

*`File(path).modified`*

*`Folder(path).modified`*

### Description

The modified attribute of `File` or `Folder` is the date of the object's last modification. If the object does not refer to a folder or file on disk, the value is null.

### Type

Date, or null if no valid `FileSystem` object is referenced; read-only.

## FileSystem name Attribute

*`File(path).name`*

*`Folder(path).name`*

### Description

The name attribute of `File` or `Folder` is the name of the object without the path specification.

### Type

String; read-only.

## FileSystem parent Attribute

*`File(path).parent`*

*`Folder(path).parent`*

### Description

The parent attribute of `File` or `Folder` is the folder object containing this object. If this object already is the root folder of a volume, the property value is null.

### Type

Folder, or null if the object has no parent; read-only.

## FileSystem path Attribute

*File(path).path*

*Folder(path).path*

### Description

The path attribute of File or Folder is the path portion of the absolute URI. If the name does not have a path, this property contains the empty string.

### Type

String, empty if name has no path; read-only.

## FileSystem relativeURI Attribute

*File(path).relativeURI*

*Folder(path).relativeURI*

### Description

The relativeURI attribute of File or Folder is the path name for the object in URI notation, relative to the current folder.

### Type

String; read-only.

## FileSystem remove() Method

*File(path).remove()*

*Folder(path).remove()*

### Description

The remove() method of File or Folder deletes the file or folder that this object represents. Folders must be empty before they can be deleted. The return value is true if the file or folder has been deleted.

IMPORTANT: The remove() method deletes the referenced file or folder immediately. It does not move the referenced file or folder to the system trash. The effects of the remove method cannot be undone. It is recommended that you prompt the user for permission to delete a file or folder before deleting it. The method does not resolve aliases; it rather deletes the file alias itself.

### Parameters

None.

### Returns

Boolean.

## FileSystem rename() Method

*File(path).rename(string)*

*Folder(path).rename(string)*

**Description**

The rename() method of File or Folder renames the object to a new name. The new name must not have a path. This method returns true if the object was renamed. The method does not resolve aliases, but rather renames the alias file.

**Parameters**

string	the new name for the object.
--------	------------------------------

**Returns**

Boolean.

## FileSystem resolve() Method

*File(path).resolve()*

*Folder(path).resolve()*

**Description**

The resolve() method of File or Folder attempts to resolve the file system alias that this object points to. If successful, a new File or Folder object is returned that points to the resolved file system element. If the object is not an alias, or if the alias could not be resolved, the return value is null.

**Parameters**

None.

**Returns**

FileSystem object (File or Folder) or null if no alias.

## Folder Class

*Folder.*

**Description**

The Folder class contains methods and attributes common to Folder objects. A Folder object corresponds to a directory or folder.

Also included in this class are all attributes and methods within the FileSystem class, as those apply to Folders as well as Files.

Note that the difference between the Folder Class and Folder Object is that the class attributes and methods require no specific instance of a Folder, whereas class methods and attributes do.

**Attributes**

current	see "Folder current Class Attribute" on page 57	The current folder is returned as a Folder object.
---------	---	--

startup	see "Folder startup Class Attribute" on page 58	The folder containing the executable image of the running application.
system	see "Folder system Class Attribute" on page 58	The folder containing the operating system files.
temp	see "Folder temp Class Attribute" on page 59	The default folder for temporary files.
trash	see "Folder trash Class Attribute" on page 59	The folder containing deleted items.

**(Class Attributes from "fileGetDialog() Global Function" on page 46)**

fs	see "FileSystem fs Class Attribute" on page 51	The name of the files system; read-only.
----	--	--

**Methods**

Folder() new Folder()	see "Folder() Class Method" on page 56	Construct a new Folder object.
selectDialog()	see "Folder selectDialog() Class Method" on page 58	Open a dialog box that permits you to select a folder using the OS specific folder select dialog.

**(Class Methods from "fileGetDialog() Global Function" on page 46)**

decode()	see "FileSystem decode() Class Method" on page 50	decodes the input string from UTF-8
encode()	see "FileSystem encode() Class Method" on page 50	encodes the input string in UTF-8

## Folder Object

```
Folder("path").
```

**Description**

The Folder object contains methods and attributes common to Folder objects. A Folder object corresponds to a directory or folder.

Also included in this object are all attributes and methods within the FileSystem object, as those apply to Folders as well as Files.

**(Attributes inherited from "FileSystem Object" on page 47)**

absoluteURI	see "FileSystem absoluteURI Attribute" on page 49	The full path name for the object in URI notation.
alias	see "FileSystem alias Attribute" on page 49	Returns true if the object refers to a file system alias.

<code>created</code>	see "FileSystem created Attribute" on page 49	The creation date of the object.
<code>error</code>	see "FileSystem error Attribute" on page 50	Contains a message describing the last file system error.
<code>exists</code>	see "FileSystem exists Attribute" on page 51	Returns true if the path name of this object refers to an actually existing file or folder.
<code>fsName</code>	see "FileSystem fsName Attribute" on page 51	The file-system specific name of the object as a full path name.
<code>modified</code>	see "FileSystem modified Attribute" on page 52	The date of the object's last modification.
<code>name</code>	see "FileSystem name Attribute" on page 52	The name of the object without the path specification.
<code>parent</code>	see "FileSystem parent Attribute" on page 52	The folder object containing this object.
<code>path</code>	see "FileSystem path Attribute" on page 53	The path portion of the absolute URI.
<code>relativeURI</code>	see "FileSystem relativeURI Attribute" on page 53	The path name for the object in URI notation, relative to the current folder.

## Methods

<code>create()</code>	see "Folder create() Method" on page 57	Attempt to create a folder at the location the path name points to.
<code>getFiles()</code>	see "Folder getFiles() Method" on page 57	Get a list of File and Folder objects contained in the folder object.

## (Methods inherited from "FileSystem Object" on page 47)

<code>getRelativeURI()</code>	see "FileSystem getRelativeURI() Method" on page 51	Calculate and return the relative URI, given a base path, in URI notation.
<code>remove()</code>	see "FileSystem remove() Method" on page 53	Delete the file or folder that this object represents.
<code>rename()</code>	see "FileSystem rename() Method" on page 53	Rename the object to the new name.
<code>resolve()</code>	see "FileSystem resolve() Method" on page 54	Attempt to resolve the file system alias that this object points to.

## Folder() Class Method

`Folder(path)`

`new Folder(path)`



### Description

This function constructs a new Folder object. If the given path name refers to an already existing disk file, a File object is returned instead.

The folder that the path name refers to does not need to exist. If the argument is omitted, a temporary name is generated.

### Parameters

path	path for the folder created, expressed as a string
------	--

### Returns

Folder (or File if path refers to an existing file).

## Folder create() Method

*Folder(path).create()*

### Description

The create() method attempts to create a folder at the location the path name points to.

### Parameters

None.

### Returns

Boolean; true if the folder was created.

## Folder current Class Attribute

*Folder.current*

### Description

The current attribute of Folder is the current folder is returned as a Folder object. Assigning either a Folder object or a string containing the new path name sets the current folder.

### Type

Folder; read/write.

## Folder getFiles() Method

*Folder.getFiles(mask)*

### Description

The Folder getFiles() method returns a list of File and Folder objects contained in the folder object. The mask paramter is the search mask for the file names, expressed as a string. It may contain question marks and asterisks and is preset to \* to find all files.

Alternatively, a function may be supplied. This function is called with a File or Folder object for every file or folder in the directory search. If the function returns true, the object is added to the array.

On Windows, all aliases end with the extension ".lnk". This extension is stripped from the file name when found to preserve compatibility with other operating systems. You can, however, search for all aliases by supplying the search mask "\*.lnk". This is NOT recommended, however, because it is not portable.

### Parameters

mask	String	search mask for the files names (see above)
------	--------	---

### Returns

Array of File & Folder objects or null if the folder does not exist.

## Folder selectDialog() Class Method

*Folder.selectDialog(prompt, preset)*

### Description

The Folder SelectDialog() method opens a dialog box that permits you to select a folder using the OS specific folder select dialog. Both arguments are optional.

### Parameters

prompt	String	displays a prompt text if the dialog allows the display of such a message. Optional
preset	Folder	a folder that is pre-selected when the dialog opens.

### Returns

Folder object pointing to the selected folder, or null if the user cancels the dialog.

## Folder startup Class Attribute

*Folder.startup*

### Description

The startup attribute of Folder is the folder containing the executable image of the running application.

### Type

Folder; read-only.

## Folder system Class Attribute

*Folder.system*

### Description

The system attribute of Folder is the folder containing the operating system files.

**Type**

Folder; read-only.

## Folder temp Class Attribute

*Folder.temp*

**Description**

The temp attribute of Folder is the default folder for temporary files.

**Type**

Folder; read-only.

## Folder trash Class Attribute

*Folder.trash*

**Description**

The trash attribute of Folder is the folder containing deleted items.

**Type**

Folder; read-only.

## folderGetDialog() Global Function

*folderGetDialog(prompt)*

**Description**

The folderGetDialog global function displays a dialog in which the user can select a folder.

**Parameters**

<i>prompt</i>	message which displays on the title bar of the dialog; truncated if too long
---------------	--

**Returns**

Folder object, or null if the user cancels the dialog.

## FolderItem Object

*app.project.*

**Description**

The FolderItem object corresponds to any folder in your Project Window. It can contain various types of items (footage, compositions, solids) as well as other folders.

## FootageItem Object

*app.project.item(index)*

```
app.project.items[index]
```

### Description

The FootageItem object represents a footage item imported into the project, which appears in the Project window. Used with an index number it will return corresponding footage item.

## FootageItem file Attribute

```
app.project.item(index).file
```

### Description

The file attribute is the File object of the footage's source file.

### Type

File object; read only

## FootageItem replace() Method

```
app.project.item(index).replace(file)
```

### Description

The FootageItem replace() method replaces the FootageItem with the file given as a parameter.

### Parameters

file	File object
------	-------------

## FootageItem replaceWithSequence() Method

```
app.project.item(index).replaceWithSequence(file, forceAlphabetical)
```

### Description

The FootageItem replaceWithSequence() method replaces the FootageItem with the image sequence given as a parameter.

### Parameters

file	File object	
forceAlphabetical	boolean	a value of true is equivalent to activating the Force alphabetical order checkbox option in the File > Replace > Footage > File dialog box.

## ImportOptions Object

```
new ImportOptions();
```

```
new ImportOptions(File);
```

## Description

The ImportOptions object provides the ability to create, change and access options for the importFile() method. You can create ImportOptions using one of two constructors, one of which takes arguments, the other which does not.

## Constructors

If importFile() is set without arguments it has a "file" which does not exist unless it is set in another statement:

```
new ImportOptions().file = new File("myfile.psd");
```

Otherwise importFile can be set with a single argument, which is a File object:

```
var my_io = new ImportOptions( new File( "myfile.psd" ) );
```

## Methods

canImportAs ( )	see "ImportOptions can-ImportAs() Method" on page 61	sets the ImportAsType, allowing the input to be restricted to a particular type
-----------------	--	---

## Attributes

importAs	see "ImportOptions importAs Attribute" on page 62	contains the ImportAsType
sequence	see "ImportOptions sequence Attribute" on page 62	boolean to determine whether a sequence or an individual file is imported
forceAlphabetical	see "ImportOptions forceAlphabetical Attribute" on page 62	boolean to determine whether the "Force Alphabetical Order" option is set

## ImportOptions canImportAs() Method

```
ImportOptions.canImportAs (ImportAsType)
```

## Description

The canImportAs() method is used to determine whether a given file can be imported as a given ImportAsType, passed in as an argument.

## Parameters

ImportAsType; one of:

ImportAsType.COMP

ImportAsType.FOOTAGE

ImportAsType.COMP\_CROPPED\_LAYERS

ImportAsType.PROJECT

**Returns**

Boolean.

**Example**

```
var io = new ImportOptions( File("c:\\foo.psd"));  
io.canImportAs( ImportAsType.COMP )
```

## ImportOptions forceAlphabetical Attribute

`ImportOptions.forceAlphabetical`

**Description**

The `forceAlphabetical` attribute is a boolean. A value of true is equivalent to activating the “Force alphabetical order” checkbox option in the File > Import > File... dialog.

**Type**

Boolean; read/write.

## ImportOptions importAs Attribute

`ImportOptions.importAs`

**Description**

The `importAs` attribute holds the `importAsType` for the file to be imported. You can set it by setting a file of the type you want to import as an argument.

**Type**

`ImportAsType`; read/write. One of:

`ImportAsType.COMP_CROPPED_LAYERS`

`ImportAsType.FOOTAGE`

`ImportAsType.COMP`

`ImportAsType.PROJECT`

## ImportOptions sequence Attribute

`ImportOptions.sequence`

**Description**

The `sequence` attribute is a boolean; it determines whether a sequence or an individual file is imported.

**Type**

Boolean; read/write.

## Item Collection

*app.project.items*

### Description

The Item Collection contains all of the Item objects in the project. This is the equivalent of all of the items (files, folders, solids, etc.) found in the Project window of a given project.

### Attributes

<code>length</code>	the number of objects in the collection (applies to all collections)
---------------------	--

### Methods

<code>[ ]</code>	retrieves an object or objects in the collection via its index number
------------------	---

## Item Object

```
app.project.item(index)
```

```
app.project.items[index]
```

### Description

The Item object represents an item that can appear in the Project window. FootageItem, CompItem, and FolderItem are all types of Item.

Note that numbering of the index for item starts at 1, not 0.

### Attribute

<code>name</code>	see "Item name Attribute" on page 63	the name of the object as shown in the Project window
-------------------	--------------------------------------	---

## Item name Attribute

```
app.project.item(index).name
```

### Description

The item name attribute is the name of the item as displayed in the Project window.

### Type

String; read/write.

## OutputModule Object

```
app.project.renderQueue.item(index).outputModule(index)
```

### Description

The outputModule object of renderQueueItem generates a single file or sequence via a render, and contains attributes and methods relating to that file to be rendered. It returns an Output Module with the given index number. The indexed items are numbered beginning with 1.

## Methods

<code>remove()</code>	see “OutputModule remove() Method” on page 65	removes the Output Module
<code>saveAsTemplate()</code>	see “OutputModule saveAsTemplate() Method” on page 66	saves a new Output Module Template with the given name
<code>applyTemplate()</code>	see “OutputModule applyTemplate() Method” on page 64	applies a pre-set Output Module Template

## Attributes

<code>file</code>	see “OutputModule file Attribute” on page 65	the path and name of the file to be rendered
<code>postRenderAction</code>	see “OutputModule postRenderAction Attribute” on page 65	one of the postRenderAction types
<code>name</code>	see “OutputModule name Attribute” on page 65	the name of the Output Module as presented to the user
<code>templates</code>	see “OutputModule templates Attribute” on page 66	array of all Output Module templates

# OutputModule Collection

`app.project.renderQueue.items.outputModule`

## Description

The Output Module Collection contains all of the Output Modules in the project.

## Attributes

<code>length</code>	the number of objects in the collection (applies to all collections)
---------------------	--

## Methods

<code>[]</code>	retrieves an object or objects in the collection via its index number
<code>add()</code>	adds an Output Module with a specified template

## See also

“Collection Object” on page 33

# OutputModule applyTemplate() Method

`app.project.renderQueue.item(index).outputModules[i].applyTemplate(templateName)`

## Description

Applies an existing Output Module template, identified by name.



### Parameters

templateName	name of the template to be applied
--------------	------------------------------------

### Returns

None.

## OutputModule file Attribute

```
app.project.renderQueue.item(index).outputModules[i].file
```

### Description

The file attribute is the File object to which the output module is set to render.

### Type

File object; read-write.

## OutputModule name Attribute

```
app.project.renderQueue.item(index).outputModules[i].name
```

### Description

The name attribute is the output module name as it is presented to the user, expressed as a string.

### Type

String; read-only.

## OutputModule postRenderAction Attribute

```
app.project.renderQueue.item(index).outputModules[i].postRenderAction
```

### Description

The postRenderAction attribute returns the Post Render Action (listed below).

### Type

PostRenderAction (read/write); one of the following:

```
postRenderAction.NONE
```

```
postRenderAction.IMPORT
```

```
postRenderAction.IMPORT_AND_REPLACE_USAGE
```

```
postRenderAction.SET_PROXY
```

## OutputModule remove() Method

```
app.project.renderQueue.item(index).outputModules[i].remove()
```

**Description**

Deletes an Output Module.

**Parameters**

None.

**Returns**

None.

## OutputModule saveAsTemplate() Method

```
app.project.renderQueue.item(index).outputModules[i].saveAsTemplate(name)
```

**Description**

Saves an Output Module with the name given as a parameter.

**Parameters**

name	name of the new template
------	--------------------------

**Returns**

None.

## OutputModule templates Attribute

```
app.project.renderQueue.item(index).outputModules[i].templates
```

**Description**

The templates attribute is an array of strings; these are the names of the templates in the local installation of After Effects.

**Type**

Array; read-only.

## Project Object

```
app.project
```

**Description**

The project object enables access to data and functionality within a particular After Effects project.

Attributes of the Project object provide access to specific objects within an After Effects project, such as imported files and footage, comps, as well as project settings such as the timecode base.

Methods of the Project object can import footage, create solids, compositions and folders, and save changes.

## Methods

<code>consolidateFootage()</code>	see “Project consolidateFootage() Method” on page 69	replicates the functionality of File > Consolidate All Footage
<code>removeUnusedFootage()</code>	see “Project removeUnusedFootage() Method” on page 71	replicates the functionality of File > Remove Unused Footage
<code>reduceProject()</code>	see “Project reduceProject() Method” on page 71	replicates the functionality of File > Reduce Project
<code>close()</code>	see “Project close() Method” on page 68	closes the project with normal save options
<code>save()</code>	see “Project save() Method” on page 72	saves the project (or pops up a Save dialog if project has never been saved)
<code>importPlaceholder()</code>	see “Project importPlaceholder() Method” on page 70	replicates the functionality of File > Import > Placeholder...
<code>importFile()</code>	see “Project importFile() Method” on page 69	replicates the functionality of File > Import > File...

## Attributes

<code>file</code>	see “Project file Attribute” on page 69	file object of the currently open project
<code>rootFolder</code>	see “Project rootFolder Attribute” on page 71	folderItem containing all the contents of the project; the equivalent of the Project window
<code>activeItem</code>	see “Project activeItem Attribute” on page 68	the currently active item, or null if none is active or multiple items are active
<code>bitsPerChannel</code>	see “Project bitsPerChannel Attribute” on page 68	color depth of the current project
<code>transparencyGridThumbnails</code>	see “Project transparencyGridThumbnails Attribute” on page 73	determines if thumbnail views should use the transparency checkerboard pattern
<code>timecodeDisplayType</code>	see “Project timecodeDisplayType Attribute” on page 72	the method with which timecode is set to display
<code>timecodeBaseType</code>	see “Project timecodeBaseType Attribute” on page 72	the Timecode Base as set in the File > Project Settings dialog

<code>timecodeNTSCDropFrame</code>	see “Project timecode-NTSCDropFrame Attribute” on page 73	equivalent to Drop Frame or Non-Drop Frame in the File > Project Settings dialog
<code>timecodeFilmType</code>	see “Project timecode-FilmType Attribute” on page 73	the method with which timecode is set to display
<code>numItems</code>	see “Project numItems Attribute” on page 70	total number of items contained in the project
<code>selection</code>	see “Project selection Attribute” on page 72	an array of the items selected in the Project window

## Project activeItem Attribute

`app.project.activeItem`

### Description

The project attribute `activeItem` returns the item which is currently active and is to be acted upon, or a null if no item is currently selected or if multiple items are selected.

### Type

The item which is currently active; read-only.

## Project bitsPerChannel Attribute

`app.project.bitsPerChannel`

### Description

The `bitsPerChannel` attribute is an integer describing the color depth of the current project (either 8 or 16 bits).

### Type

Integer (8 or 16 only); read/write.

## Project close() Method

`app.project.close(CloseOptions)`

### Description

Closes the project with the option of saving changes automatically, prompting the user to save changes or closing without saving changes.

### Parameters

<code>CloseOptions</code>	the action to be performed on close (see Enumerated Types, below)
---------------------------	---

### Enumerated Types

<code>CloseOptions.DO_NOT_SAVE_CHANGES</code>	close without saving
---	----------------------

<code>CloseOptions.PROMPT_TO_SAVE_CHANGES</code>	send a prompt asking whether to save changes before close
<code>CloseOptions.SAVE_CHANGES</code>	save automatically on close option

**Returns**

Boolean. False only in one case: the file has not been previously saved; the user is presented with a save dialog, and cancels the save.

## Project consolidateFootage() Method

*app.project.consolidateFootage()*

**Description**

Replicates the functionality of File > Consolidate All Footage.

**Parameters**

None.

**Returns**

Integer; the total number of footage items removed.

## Project file Attribute

*app.project.file*

**Description**

The file attribute is a File object representing the project that is currently open.

**Type**

File Object or null if project has not been saved; read-only.

## Project importFile() Method

*app.project.importFile(ImportOptions)*

**Description**

Replicates the functionality of the Import File dialog.

**Parameters**

<code>ImportOptions</code>	the options as set in the ImportOptions object
----------------------------	--

**Returns**

FootageItem

**Example**

```
app.project.importFile( ImportOptions( File( "sample.psd" ) ) )
```

**See also**

“ImportOptions Object” on page 60

## Project importPlaceholder() Method

```
app.project.importPlaceholder(name, width, height, framerate, duration)
```

**Description**

Replicates the functionality of File > Import > Placeholder...; adds a placeholder footage item of a specified name, width, height, framerate, and duration to the project.

**Parameters**

name	The name of the placeholder
width	The width in pixels of the placeholder footage
height	The height in pixels of the placeholder footage
framerate	The frame rate of the placeholder footage
duration	The duration of the placeholder footage, in seconds

**Returns**

FootageItem.

## Project importFileWithDialog() Method

```
app.project.importFileWithDialog()
```

**Description**

Replicates the functionality of File > Import > File... and produces an Import dialog for the user. Unlike importFile(), importWithDialog() does not take arguments.

**Returns**

FootageItem; or undefined if user cancels a dialog.

## Project numItems Attribute

```
app.project.numItems
```

**Description**

The numItems attribute represents the total number of items contained in the project, including folders and all types of footage.

**Type**

Integer; read-only.

**Example**

```
n = app.project.numItems
```

```
alert("There are " + n + " items in this project.")
```

## Project reduceProject() Method

```
app.project.reduceProject(array_of_items)
```

### Description

Replicates the functionality of File > Reduce Project.

### Parameters

<code>array_of_items</code>	the items to which the project is to be reduced
-----------------------------	---

### Returns

Integer; the total number of items removed.

### Example

```
var theItems = new Array();
theItems[theItems.length] = app.project.item(1);
theItems[theItems.length] = app.project.item(3);

app.project.reduceProject(theItems);
```

## Project removeUnusedFootage() Method

```
app.project.removeUnusedFootage()
```

### Description

Replicates the functionality of File > Remove Unused Footage.

### Parameters

None.

### Returns

Integer; the total number of footage items removed.

## Project rootFolder Attribute

```
app.project.rootFolder
```

### Description

The rootFolder attribute is the root folder containing the root contents of the project; this is a conceptual folder that contains all items items the Project window, but not items contained inside other folders in the Project window.

### Type

FolderItem; read-only.

## Project save() Method

```
app.project.save()
```

```
app.project.save(File)
```

### Description

Saves the project (or prompts the user if the file has never previously been saved). Passing in a File object is equivalent to the save as command and allows you to save a project to a new file.

### Parameters

File	the File object to save
------	-------------------------

### Returns

None.

## Project selection Attribute

```
app.project.selection
```

### Description

The selection attribute contains an array of the items selected in the Project window.

### Type

Array; read-only.

## Project timecodeBaseType Attribute

```
app.project.timecodeBaseType
```

### Description

The timecodeBaseType attribute reveals the Timecode Base as set in the File > Project Settings dialog.

### Type

Enumerated type (read/write); one of the following:

```
TimecodeBaseType.24FPS
```

```
TimecodeBaseType.25FPS
```

```
TimecodeBaseType.30FPS
```

```
TimecodeBaseType.48FPS
```

```
TimecodeBaseType.50FPS
```

```
TimecodeBaseType.60FPS
```

```
TimecodeBaseType.100FPS
```

## Project timecodeDisplayType Attribute

```
app.project.timecodeDisplayType
```



### Description

The `timecodeDisplayType` attribute describes the method with which timecode is set to display. The enumerated values are found in the File > Project Settings dialog in a pull-down menu.

### Type

Enumerated type (read/write); one of the following:

`TimecodeDisplayType.TIMECODE`

`TimecodeDisplayType.FRAMES`

`TimecodeDisplayType.FEET_AND_FRAMES`

## Project `timecodeFilmType` Attribute

*app.project.timecodeFilmType*

### Description

The `timecodeFilmType` attribute describes the film type that has been selected for the Feet + Frames option in the File > Project Settings dialog.

### Type

Enumerated type (read/write); one of the following:

`TimecodeFilmType.MM16`

`TimecodeFilmType.MM35`

## Project `timecodeNTSCDropFrame` Attribute

*app.project.timecodeNTSCDropFrame*

### Description

The `timecodeNTSCDropFrame` attribute describes how timecode for 29.97 fps footage is displayed. This corresponds to the Drop Frame or Non-Drop Frame pulldown options under “NTSC” in the File > Project Settings dialog.

### Type

Boolean (read/write); true if NTSC Drop Frame is set as the current project display style.

## Project `transparencyGridThumbnails` Attribute

*app.project.transparencyGrid*

### Description

The `transparencyGridThumbnails` attribute determines if thumbnail views should use the transparency checkerboard pattern (yes or no).

### Type

Boolean (read/write).

## prompt() Global Function

`prompt(prompt, default)`

### Description

The prompt global function opens a dialog box with a text field into which the user can enter a text string. The text string is returned as a value, or is null if the dialog is cancelled.

### Parameters

<code>prompt</code>	text string which appears in the prompt dialog
<code>default</code>	text string which appears by default in the text field

### Returns

String, or null if dialog is cancelled. Read-only.

### Example

```
// presuming a project loaded with at least one comp is open:
var myCompItem = app.project.item(1);
var newName = prompt( "What would you like to name the comp?"); //
rename it

if (newName) { //if the user cancels, newName is null
    myCompItem.name = newName; // newName now holds a string
}
```

## RenderQueue Object

`app.project.renderQueue`

### Description

The RenderQueue object enables access to data and functionality within the Render Queue area of a particular After Effects project. This object is pivotal to render automation.

Attributes of the RenderQueue object provide access to items in the Render Queue and their render status.

Methods of the RenderQueue object can start, pause and stop the render process.

The RenderQueueItem object provides access to the specific settings for an item to be rendered.

### Methods

<code>showWindow()</code>	see "RenderQueue showWindow() Method" on page 77	boolean to show/hide the Render Queue window
<code>render()</code>	see "RenderQueue render() Method" on page 76	starts the render; does not return until render is complete

<code>pauseRendering()</code>	see “RenderQueue pauseRendering() Method” on page 76	pauses the render
<code>stopRendering()</code>	see “RenderQueue stopRendering() Method” on page 77	stops the render
<code>item</code>	see “RenderQueueItem Object” on page 78	object containing the individual settings for a Render Queue item

### Attributes

<code>rendering</code>	see “RenderQueue rendering Attribute” on page 76	determines whether a render is in progress
<code>numItems</code>	see “RenderQueue numItems Attribute” on page 75	total number of items in the Render Queue
<code>items</code>	see “RenderQueue items Attribute” on page 75	the collected items in the Render Queue

## RenderQueue items Attribute

*app.project.renderQueue.items*

### Description

The items attribute of renderQueue provides a collection of all items in the Render Queue as a collection.

### Type

RQItemCollection; read-only.

### See also

“RenderQueueItem Collection” on page 77

## RenderQueue numItems Attribute

*app.project.renderQueue.numItems*

### Description

The numItems attribute indicates the total number of render queue items in the Render Queue.

### Type

Integer; read-only.

## RenderQueue pauseRendering() Method

*app.project.renderQueue.pauseRendering(pause)*

### Description

Pauses the Render Queue; equivalent to use of the Pause button in the Render Queue window during a render.

### Parameters

pause	a boolean; set to true, it pauses the render, set to false, it continues a paused render
-------	--

### Returns

None.

## RenderQueue render() Method

*app.project.renderQueue.render()*

### Description

Starts the Render Queue; equivalent to use of the Render button in the Render Queue window. Does not return until render is complete.

Set the `app.onError` if you wish to be notified of errors during the rendering process.

Set the `RenderQueueItem.onStatusChanged` attribute of a particular `RenderQueueItem` to get updates while the render is progressing.

### Parameters

None.

### Returns

None.

### See also

"Application `onError` Attribute" on page 28

"RenderQueueItem `onStatusChanged` Attribute" on page 80

## RenderQueue rendering Attribute

*app.project.renderQueue.rendering*

### Description

The rendering attribute indicates whether rendering is in progress. This is a read only attribute; use the `render()` and `stopRendering()` methods to control it. If the render is paused, this is set to true.

### Type

Boolean; read-only.

## RenderQueue showWindow() Method

*app.project.renderQueue.showWindow(doShow)*

### Description

The showWindow method of RenderQueue is a boolean; if true, it makes the Render Queue window visible, if false, it hides the window.

### Parameters

doShow	boolean; if true, shows the Render Queue window, if false, conceals it
--------	--

### Returns

None.

## RenderQueue stopRendering() Method

*app.project.renderQueue.stopRendering()*

### Description

Stops the Render Queue; equivalent to use of the Stop button in the Render Queue window during a render. Useful to call in the event of an onStatusChanged callback.

### Parameters

None.

### Returns

None.

### See also

“RenderQueueItem onStatusChanged Attribute” on page 80.

## RenderQueueItem Collection

*app.project.renderQueue.items*

### Description

The RenderQueueCollection contains all of the Render Queue items. This is the equivalent of all of the items found in the Render Queue window of a given project.

### Attributes

length	the number of objects in the collection (applies to all collections)
--------	--

### Methods

[ ]	retrieves an object or objects in the collection via its index number
add()	adds an RenderQueueItem for a specified composition

**See also**

“Collection Object” on page 33

## RenderQueueItem Object

**Description**

The RenderQueueItem object is an individual item in the Render Queue.

**Methods**

<code>outputModule()</code>	see “OutputModule Object” on page 63	returns an Output Module for the item
<code>remove()</code>	see “RenderQueueItem remove() Method” on page 81	deletes the item from the Render Queue
<code>saveAsTemplate()</code>	see “RenderQueueItem saveAsTemplate() Method” on page 81	saves a new Render Settings Template with the given name
<code>applyTemplate()</code>	see “RenderQueueItem applyTemplate() Method” on page 79	applies a pre-set Render Settings Template

**Attributes**

<code>numOutputModules</code>	see “RenderQueueItem numOutputModules Attribute” on page 80	the total number of Output Modules assigned to a given Render Queue item
<code>render</code>	see “RenderQueueItem render Attribute” on page 81	boolean which shows true if this item will render when the queue is started
<code>startTime</code>	see “RenderQueueItem startTime Attribute” on page 82	Date object representing time program began rendering the item
<code>elapsedSeconds</code>	see “RenderQueueItem elapsedSeconds Attribute” on page 79	the time elapsed in the current render, in seconds
<code>timeSpanStart</code>	see “RenderQueueItem timeSpanStart Attribute” on page 83	the start time, in seconds, in the comp to be rendered
<code>timeSpanDuration</code>	see “RenderQueueItem timeSpanDuration Attribute” on page 82	the duration of the comp to be rendered, in seconds
<code>comp</code>	see “RenderQueueItem comp Attribute” on page 79	the composition being rendered by this RQ item
<code>outputModules</code>	see “RenderQueueItem outputModules Attribute” on page 81	a collection of the Output Modules

templates	see “RenderQueueItem templates Attribute” on page 82	an array of the Render Settings templates
status	see “RenderQueueItem status Attribute” on page 82	the current status of a Render Queue item
onStatusChanged	see “RenderQueueItem onStatusChanged Attribute” on page 80	condition in which the status of an item changes (e.g. from RENDERING to DONE status)
logType	see “RenderQueueItem logType Attribute” on page 80	returns one of the log types

## RenderQueueItem applyTemplate() Method

```
app.project.renderQueue.item.applyTemplate(templateName)
```

### Description

The applyTemplate method of renderQueueItem applies a Render Settings template to the item.

### Parameters

templateName	the name of the template to apply
--------------	-----------------------------------

### Returns

None.

## RenderQueueItem comp Attribute

```
app.project.renderQueue.item(index).comp
```

### Description

The comp attribute returns the CompItem object that will be rendered by this Render Queue item. This is a read-only attribute; to change the Composition, the Render Queue item must be deleted and re-created.

### Type

CompItem; read-only.

## RenderQueueItem elapsedSeconds Attribute

```
app.project.renderQueue.item(index).elapsedSeconds
```

### Description

The elapsedSeconds attribute shows the number of seconds spent rendering the item.

### Type

Integer, or null if item has not been rendered; read-only.

## RenderQueueItem logType Attribute

*app.project.renderQueue.item(index).outputModule.LogType*

### Description

The logType attribute returns one of the log types (listed below).

### Type

LogType (read/write); one of the following:

`LogType.ERRORS_ONLY`

`LogType.ERRORS_AND_SETTINGS`

`LogType.ERRORS_AND_PER_FRAME_INFO`

## RenderQueueItem numOutputModules Attribute

*app.project.renderQueue.item(index).numOutputModules*

### Description

The numOutputModules attribute represents the total number of Output Modules assigned to a given Render Queue item.

### Type

Integer; read-only.

## RenderQueueItem onStatusChanged Attribute

*app.project.renderQueue.item(index).onStatusChanged*

### Description

The onStatusChanged attribute is invoked whenever the value of the RenderQueueItem.status attribute is changed.

Note that changes can not be made to render queue items (or to the application at large) while a render is in progress (including when paused). This mirrors the regular application functionality.

### Type

Function.

### Example

```
function myStatusChanged() {  
    alert(app.project.renderQueue.item(1).status)  
}
```

```
app.project.renderQueue.item(1).onStatusChanged = myStatusChanged();  
app.project.renderQueue.item(1).render = false; //shows dialog
```



## RenderQueueItem outputModules Attribute

*app.project.renderQueue.item(index).outputModules*

### Description

The outputModules attribute returns the collection of Output Modules for the item.

### Type

OMCollection; read-only.

## RenderQueueItem remove() Method

*app.project.renderQueue.item(index).remove()*

### Description

The remove method of renderQueueItem deletes the referenced item from the Render Queue.

### Parameters

None.

### Returns

None.

## RenderQueueItem render Attribute

*app.project.renderQueue.item(index).render*

### Description

The render attribute determines whether an item will render when the Render Queue is started.

### Type

Boolean; read/write.

## RenderQueueItem saveAsTemplate() Method

*app.project.renderQueue.item(index).saveAsTemplate(name)*

### Description

The saveAsTemplate method of RenderQueueItem saves the item's current render settings as a new template with the name passed as a parameter.

### Parameters

name	the name of the new template
------	------------------------------

### Returns

None.

## RenderQueueItem startTime Attribute

*app.project.renderQueue.item(index).startTime*

### Description

The startTime attribute returns a Date object showing the day and time that the item started rendering.

### Type

Date; null if the item has not started rendering. Read-only.

## RenderQueueItem status Attribute

*app.project.renderQueue.item(index).RQItemStatus*

### Description

The status attribute represents the current render status of the item.

### Type

RQItemStatus - one of the following attributes:

RQItem-Status.WILL_CONTINUE	render has been paused
RQItemStatus.NEEDS_OUTPUT	item lacks a valid output path
RQItemStatus.UNQUEUED	The render item is listed in the Render Queue window but is not ready to render.
RQItemStatus.QUEUED	The composition is ready to render.
RQItemStatus.RENDERING	The composition is rendering.
RQItemStatus.USER_STOPPED	The rendering process was stopped by the user.
RQItemStatus.ERR_STOPPED	The rendering process was stopped due to an error.
RQItemStatus.DONE	The rendering process for the item is complete.

## RenderQueueItem templates Attribute

*app.project.renderQueue.item(index).templates*

### Description

The templates attribute returns an array of the names of Render Settings templates available for the item. It is a read-only attribute.

### Type

Array; read-only.

## RenderQueueItem timeSpanDuration Attribute

*app.project.renderQueue.item(index).timeSpanDuration*

### Description

The `timeSpanDuration` attribute determines the duration, in seconds, of the comp to be rendered. This achieves the same effect as setting a custom end time in the Render Settings dialog, although the duration is determined by subtracting the start time from the end time.

### Type

Floating-point value; read/write.

## RenderQueueItem `timeSpanStart` Attribute

`app.project.renderQueue.item(index).timeSpanStart`

### Description

The `timeSpanStart` attribute determines the time in the comp, in seconds, at which rendering will begin. This is the equivalent of setting a custom start time in the Render Settings dialog.

### Type

Floating-point value; read/write.

## Settings Object

### Description

The Settings object provides an easy way to manage settings for scripts. The settings are persistent between application launches, saved in the After Effects Preferences file.

### Methods

<code>saveSetting()</code>	see "Settings <code>saveSetting()</code> Method" on page 84	can save a default value for a preferences item
<code>getSetting()</code>	see "Settings <code>getSetting()</code> Method" on page 83	retrieves a setting found in the Prefs file
<code>haveSetting()</code>	see "Settings <code>haveSetting()</code> Method" on page 84	used to determine whether a given section name and key name have a setting assigned

## Settings `getSetting()` Method

`app.settings.getSetting(sectionName, keyName)`

### Description

The `getSetting` method retrieves a setting found in the Prefs file.

### Parameters

<code>sectionName</code>	text string which holds the name of a section of settings; in the prefs file these are the names enclosed in brackets and quotations
--------------------------	--

keyName	text string which describes an individual setting name; these are listed in quotations below the sectionName
---------	--

**Returns**

String representing the value of the setting.

**Example**

```
var n = app.settings.getSetting("Eraser - Paint Settings", "Aligned Clone");  
alert("The setting is " + n);
```

**See also**

"Settings haveSetting() Method" on page 84

"Settings saveSetting() Method" on page 84

## Settings haveSetting() Method

*app.settings.haveSetting(sectionName, keyName)*

**Description**

The haveSetting method is used to determine whether a given section name and key name have a setting assigned.

**Returns**

Boolean.

**See also**

"Settings getSetting() Method" on page 83

"Settings saveSetting() Method" on page 84

## Settings saveSetting() Method

*app.settings.saveSetting(sectionName, keyName, value)*

**Description**

The saveSetting method can save a default value for a scripting preferences item.

**Parameters**

sectionName	text string which holds the name of a section of settings; in the prefs file these are the names enclosed in brackets and quotations
keyName	text string which describes an individual setting name; these are listed in quotations below the sectionName
value	value assigned to the setting

**See also**

"Settings getSetting() Method" on page 83

"Settings haveSetting() Method" on page 84

## System Object

*system*

### Description

The System object provides access to attributes found on the user's system, such as the user name or the name and version of the operating system.

### Attributes

<code>userName</code>	see "System user-Name Attribute" on page 86	The user name logged in to the current session of the OS.
<code>machineName</code>	see "System machineName Attribute" on page 85	The name of the host machine.
<code>osName</code>	see "System osName Attribute" on page 85	The name of the operating system (OS) currently running.
<code>osVersion</code>	see "System osVersion Attribute" on page 86	The version of the operating system (OS) currently running.

## System machineName Attribute

*system.machineName*

### Description

The machineName attribute specifies the name of the machine on which the program is running, and is expressed as a text string.

Type

String; read-only.

### Example

```
alert ( "Your machine is called " + system.machineName + ".");
```

## System osName Attribute

*system.osName*

### Description

The osName attribute specifies the name of the os on which the program is running, and is expressed as a text string.

**Type**

String; read-only.

**Example**

```
alert ( "Your OS is " + system.osname + ".");
```

## System osVersion Attribute

*system.osVersion*

**Description**

The osVersion attribute specifies the version of the current local operating system, and is expressed as a text string.

**Type**

String; read-only.

**Example**

```
alert ( "Your OS is " + system.osname + " running version " + system.osversion);
```

## System userName Attribute

*system.userName*

**Description**

The userName attribute specifies the name of the user logged on to the system, and is expressed as a text string.

**Type**

String; read-only.

**Example**

```
confirm( "You are: " + system.userName + " running on " +  
system.machineName + ".");
```

## write() Global Function

*write(text)*

**Description**

The write global function writes output to the Info palette, with no line break added.

**Parameters**

text	text string; truncated if too long for the info palette
------	---

**Example**

```
write("This text appears in Info palette.");
```

**See also**

“writeln() Global Function” on page 87

## writeln() Global Function

```
writeln(text)
```

**Description**

The write global function writes output to the info palette and adds a line break at the end.

**Parameters**

text	text string
------	-------------

**Example**

```
writeln("This line of text appears in the console window with a line  
break at the end.");
```

**See also**

“write() Global Function” on page 86

# Examples

---

## About the scripts included with After Effects

Following are sample scripts included on your CD with an overview of what they do and a step-by-step breakdown of how they work.

### Save and increment

This script automatically saves a new copy of the open After Effects project and increments a three-digit number in its name to distinguish it from preceding versions of the project. This script is saved as `save_and_increment.jsx` on your install CD.

The first step is to determine whether the currently open project has ever been saved. This is accomplished with an opening if/else statement. The first condition, `!app.project.file` is saying that if the project has not been saved, an alert telling the user to save the project is popped up and the script ends.

```
if (!app.project.file) {  
    alert ("This project must be saved before running this script.");
```

Next, if the project has been saved at least once before, we set some variables to point to the name of the file and to the numbering and file extension that we plan to add to it. The `lastIndexOf()` JavaScript searches a string backwards (from end to start) and in this case looks for the dot that separates the name from the extension.

```
} else {  
    var currFile = app.project.file;  
    var currFileName = currFile.name;  
    var extPos = currFileName.lastIndexOf(".");  
    var ext = "";
```

Now we set the `currFileName` variable to the current name, before the dot.

```
    if (extPos != -1) {  
        ext = currFileName.substring(extPos, currFileName.length);  
        currFileName = currFileName.substring(0, extPos);  
    }
```

Next we set a variable that will increment versions starting with 0, and we check to see if there is an underscore character four characters from the end of `currFileName`. If there is, we assume that the incrementer has run before, as its job is to assign a 3 digit suffix after an underscore incremented one higher than the last suffix. In that case we set `incrementer` to the current numerical string and extract the name without this numerical extension.





```

var incrementer = 0;
if (currFileName.charAt(currFileName.length -4) == "_") {
    incrementer = currFileName.substring(currFileName.length - 3,
currFileName.length);
    currFileName = currFileName.substring(0, currFileName.length -
4);
}

```

Now we add an incrementer loop and test for whether numbering has extended to two or three digits (e.g. if the numbering has reached “\_010” or above, or “\_100” or above), assigning a zero for each if not.

```

    incrementer++;
var istring = incrementer + "";
if (incrementer < 10) {
    istring = "0" + istring;
}
if (incrementer < 100) {
    istring = "0" + istring;
}

```

Finally we create a new file using our updated name and extension, pop up an alert letting the user know the new file name being saved, and save the project with the new file name.

```

var newFile = File(currFile.path + "/" + currFileName + "_" +
istring + ext);
alert(newFile.fsName);
app.project.save(newFile);
}

```

## Render named items

This script allows you to find compositions in the open project with a particular text string in their names and send all such compositions to the Render Queue.

To start, we check to see if a default string for rendering has already been set in the user preferences. If so, we set this as a user prompt, handy if you’re always looking for the same string (for example, “FINAL” or “CURRENT”). If not, we set a new sectionName and keyName for the preferences file along with a placeholder value for the string that will be entered by the user.

```

var sectionName = "AE Example Scripts";
var keyName = "Render comps with this string";
var searchString = "";

if (app.settings.haveSetting(sectionName, keyName)) {
    searchString = app.settings.getSetting(sectionName, keyName);
}

```

```
}
```

Now we pop up a prompt to the user asking for what text string we should use.

```
searchString = prompt("What string to render?", searchString);
```

We next go through the project looking for the text entered by the user, and seeing if the item that contains that text is a composition, sending all compositions with that text string in their names to the Render Queue. If the user cancels, the text is undefined. Otherwise, we save the new setting in preferences, convert it to all lower case letters for consistency's sake (keeping in mind that the search will not be case sensitive).

```
if (searchString) {

    app.settings.saveSetting(sectionName, keyName, searchString);
    searchString = searchString.toLowerCase();
    for (i = 1; i <= app.project.numItems; ++i) {
        var curItem = app.project.item(i);
        if (curItem instanceof CompItem) {
            if (curItem.name.toLowerCase().indexOf(searchString) != -1)
            {
                app.project.renderQueue.items.add(curItem);
            }
        }
    }
}
```

Finally, we make the Render Queue window visible and bring it to the front, ready for the user to assign save locations for the new render queue items.

```
app.project.renderQueue.showWindow(true);
}
```

## New render locations

This script allows the user to select queued items in the Render Queue and assign a new render destination for them.

First, we prompt the user for a new folder to use as a render destination.

```
var newLocation = folderGetDialog("Select a render destination...");
```

Next, we make certain that the user entered a new location (and didn't cancel the dialog). Then we create a loop for each selected render queue item. If this item is queued, we take the current render location, give it a new name and location, and then pop up an alert stating the new file path.

```
if (newLocation) { //boolean to see if the user cancelled
    for (i = 1; i <= app.project.renderQueue.numItems; ++i) {
```

```

var curItem = app.project.renderQueue.item(i);
if (curItem.status == RQItemStatus.QUEUED) {
    for (j = 1; j <= curItem.numOutputModules; ++j) {
        var curOM = curItem.outputModule(j);
        var oldLocation = curOM.file;
        curOM.file = new File(newLocation.toString() + "/" +
oldLocation.name);
        alert(curOM.file.fsName);
    }
}
}
}

```

## Smart Import

This script allows the user to import the full, nested contents of a folder just by selecting it. It attempts to detect whether each item is a still, moving footage, or an image sequence. The user still has to make other choices via dialogs, such as which layer of a multi-layer image (e.g. a .psd) to import.

First, we prompt the user for a folder whose contents are to be imported, and ascertain that the user chooses a folder rather than cancelling the dialog. We then call a function which appears below to import all of the files, one by one.

```

var targetFolder = folderGetDialog("Import Items from Folder...");
//returns a folder or null
if (targetFolder) {
    function processFile (theFile) {
        var importOptions = new ImportOptions (theFile);
        //create a variable containing ImportOptions
        importSafeWithError (importOptions);
    }
}

```

Now we add a function to test whether a given file is part of a sequence. This uses Regular Expressions, which are a special type of JavaScript designed to reduce the number of steps required to evaluate a string. The firstone tests for the presence of sequential numbers anywhere in the file name, followed by another making certain that the sequential files aren't of a type that can't be imported as a sequence (moving image files).

We then check adjacent files to see if a sequence exists, stopping after we've evaluated ten files to save processing time.

```

function testForSequence (files){
    var searcher = new RegExp ("[0-9]+");
    var movieFileSearcher = new RegExp ("(mov|avi|mpg)$", "i");
    var parseResults = new Array;

```

```

    for (x = 0; (x < files.length) & x < 10; x++) {
        //test that we have a sequence, stop parsing after 10 files
        var movieFileResult = movieFileSearcher.exec(files[x].name);
        if (! movieFileResult) {
            var currentResult = searcher.exec(files[x].name);

```

If no match is found using the Regular Expression looking for a number string, we get null and assume there is no image sequence. Otherwise, we want an array consisting of the matched string and its location within the file name.

```

            if (currentResult) {
                //we have a match - the string contains numbers
                //the match of those numbers is stored in the array[1]
                //take that number and save it into parseResults
                parseResults[parseResults.length] = currentResult[0];
            }
            else {
                parseResults[parseResults.length] = null;
            }
        }
        else {
            parseResults[parseResults.length] = null;
        }
    }
}

```

Now if all of the files just evaluated indicated that they are part of a numbered sequence, we assume that we have a sequence and return the first file of that sequence. Otherwise, we end this function.

```

var result = null;
for (i = 0; i < parseResults.length; ++i) {
    if (parseResults[i]) {
        if (! result) {
            result = files[i];
        }
    } else {
        //case in which a file name did not contain a number
        result = null;
        break;
    }
}

```

```

    return result;
}

```

Next we add a function to pop up error dialogs if there is a problem with any file we are attempting to import.

```

function importSafeWithError (importOptions) {
    try {
        app.project.importFile (importOptions);
    } catch (error) {
        alert(error.toString() + importOptions.file.fsName);
    }
}

```

Next comes a function to actually import any image sequence that we discover using `testForSequence()`, above. Note that there is an option for forcing alphabetical order in sequences, which is commented out in the script as written. If you want to force alphabetical order, un-comment the line which reads, “`importOptions.forceAlphabetical = true`” by removing the two slashes at the beginning of that line.

```

function processFolder(theFolder) {
    var files = theFolder.GetFiles();
    //Get an array of files in the target folder
    //test whether theFolder contains a sequence
    var sequenceStartFile = testForSequence(files);
    //if it does contain a sequence, import the sequence
    if (sequenceStartFile) {
        var importOptions = new ImportOptions (sequenceStartFile);
        //create a variable containing ImportOptions
        importOptions.sequence = true;
        //importOptions.forceAlphabetical = true;
        //un-comment this if you want to force alpha order by
default
        importSafeWithError (importOptions);
    }

    //otherwise, import the files and recurse

    for (index in files) {
        //Go through the array, set each element to singleFile, run
this:
        if (files[index] instanceof File) {
            if (! sequenceStartFile) {

```

```

        //if file is already part of a sequence, don't import it
        individually
            processFile (files[index]);
            //calls the processFile function above
        }
    }
    if (files[index] instanceof Folder) {
        processFolder (files[index]); // recursion
    }
}

processFolder(targetFolder);
}

```

## Render and Mail

This script renders all queued items in an open project, and sends an email report to indicate when the render has completed. It makes use of two other scripts which follow, `email_methods.jsx` (to send the email properly) and `email_setup.jsx` (which establishes the sender, recipient, and email server).

We start by establishing conditions under which the script will run. An open project with at least one item queued is required.

```

{
    var safeToRunScript = true;

    safeToRunScript = app.project != null;
    if (! app.project) {
        alert ("A project must be open to run this script.");
    }
    if (safeToRunScript) {
        debugger;
        //check the render queue and make certain at least one item is
        queued
        safeToRunScript = false;
        for (i = 1; i <= app.project.renderQueue.numItems; ++i) {
            if (app.project.renderQueue.item(i).status ==
                RQItemStatus.QUEUED) {
                safeToRunScript = true;
                break;
            }
        }
    }
}

```

```

    }
    if (! safeToRunScript) {
        alert ("You do not have any items set to render.");
    }
}

```

Now we check whether we have email settings already saved in the Preferences. If so, we don't need to prompt the user. If not, we run the email\_setup.jsx script which prompts the user as to the mail gateway, sender and recipient addresses. If there are saved settings that you need to change, you can always run email\_setup.jsx to make new settings which overwrite the existing ones.

```

    if (safeToRunScript) {

        var settings = app.settings;
        if ( !settings.haveSetting("Email Settings", "Mail Server") ||
            !settings.haveSetting("Email Settings", "Reply-to
Address") ||
            !settings.haveSetting("Email Settings", "Render Report
Recipient")){

            // We don't have the settings yet, so run email_setup.jsx
            // to prompt for them
            var email_setupfile = new File("email_setup.jsx");
            email_setupfile.open("r");
            eval( email_setupfile.read() );
            email_setupfile.close();
        }

        var myQueue = app.project.renderQueue //creates a shortcut for
RQ

```

Now we're ready to render. Once rendering is complete, the script creates a text string for the email message which contains the start time of the render, the render time of each item in the queue, and the total render time.

```

myQueue.render();

var projectName = "Unsaved Project";
if (app.project.file) {
    projectName = app.project.file.name;
}

```

```
var myMessage = "Rendering of " + projectName + " is
complete.\n\n";
```

Now email the message, using the three settings from the email\_methods.jsx script which has been automatically run to prompt the user for the server, above.

```
if ( !settings.haveSetting("Email Settings", "Mail Server") ||
    !settings.haveSetting("Email Settings", "Reply-to
Address") ||
    !settings.haveSetting("Email Settings", "Render Report
Recipient")){
    alert("Can't send an email, I don't have all the settings I
need.
    Aborting.");
} else {
    // Load code from a file with handy emailing methods:
    var emailCodeFile = new File("email_methods.jsx");
    emailCodeFile.open("r");
    eval( emailCodeFile.read() );
    emailCodeFile.close();
```

Finally, we send an error if for any reason we are unable to send the mail.

```
var serverSetting = settings.getSetting("Email Settings",
"Mail
Server");
var fromSetting = settings.getSetting("Email Settings",
"Reply-
to Address");
var toSetting    = settings.getSetting("Email Settings",
"Render
Report Recipient");
var myMail = new EmailSocket(serverSetting);
if (! myMail.send (fromSetting, toSetting, "AE Render
Completed", myMessage) ) {
    alert("Sending mail failed");
}
}
}
}
```



## Email Methods

This script creates an email object for use with the Render and Email script, described above. It uses code which is specific to the socket object and therefore requires advanced understanding of networking to edit; the comments below describe its operation.

```
// Create an email object. The function may be called both
// as a global function and as a constructor. It takes the
// name of the email server, and an optional Boolean that,
// if true, prints debugging messages.
// This object is not guaranteed to work for all SMTP servers,
// some of them may require a different set of commands.

// functions:
// send (fromAddress, toAddress, subject, text) - send an email
// auth (name, pass) - do an authorization via POP3
// both functions return false on errors

// sample:
// e = new EmailSocket ("mail.host.com");
// authorize via POP3 (not all servers require authorization)
// e.auth ("myname", "mypass");
// send the email
// e.send ("me@my.com", "you@you.com", "My Subject", "Hi there!")

// This script makes use of the Socket object, and creates a new
class
// called EmailSocket that is derived from Socket. For more infor-
mation on
// creating new classes in this way, consult chapter 7 of JavaScript,
The
// Definitive Guide, by David Flanagan (O'Reilly).

//This is the constructor for the email socket. It takes as
arguments:
//server - the address of the email server (is not checked for
validity here)
//dbg - a boolean, if true, prints additional error information

function EmailSocket (server, dbg) {
    var obj = new Socket;
    obj._host = server;
    obj._debug = (dbg == true);
```

```
    obj.__proto__ = EmailSocket.prototype;
    return obj;
}

// correct the prototype chain to point to the Socket prototype chain
// - this is what actually causes the derivation from Socket.

EmailSocket.prototype.__proto__ = Socket.prototype;

// This sets up the send() member function. send() takes as
// arguments:
// from - the email address of the sender. This is not validated.
// to - the email address of the recipient. If there is an error,
// and the from address is incorrect, you will not be notified.
// subject - the contents of the subject field.
// text - the body of the message.
//
// Returns:
// true if sending succeeded
// false otherwise (if there was an error)
//
// Note that this code uses a local function object to create
// the function that is assigned to send.

EmailSocket.prototype.send = function (from, to, subject, text) {
    // open the socket on port 25 (SMTP)
    if (!this.open (this._host + ":25"))
        return false;
    try {
        // discard the greeting
        var greeting = this.read();
        if (this._debug)
            write ("RECV: " + greeting);
        // issue EHLO and other commands
        this._SMTP ("EHLO " + from);
        this._SMTP ("MAIL FROM: " + from);
        this._SMTP ("RCPT TO: " + to);
        this._SMTP ("DATA");
        // send subject and time stamp
        this.writeln ("From: " + from);
```

```
this.writeln ("To: " + to);
this.writeln ("Date: " + new Date().toString());
if (typeof subject != undefined)
    this.writeln ("Subject: " + subject);
this.writeln();
// send the text
if (typeof text != undefined)
    this.writeln (text);
// terminate with a single dot and wait for response
this._SMTP (".");
// terminate the session
this._SMTP ("QUIT");
this.close();
return true;
}
catch (e) {
    this.close();
    return false;
}
}

// Authorize via POP3. Supply name and password.
//
// Returns:
// true if sending succeeded
// false otherwise (if there was an error)
//
// Arguments:
// name - the userName of the account
// pass - the password

EmailSocket.prototype.auth = function (name, pass) {
    // open the connection on port 110 (POP3)
    if (!this.open (this._host + ":110"))
        return false;
    try {
        // discard the greeting
        var greeting = this.read();
        if (this._debug)
            write ("RCV: " + greeting);
```

```
// issue POP3 commands
this._POP3 ("USER " + name);
this._POP3 ("PASS " + pass);
this._POP3 ("QUIT");
this.close();
return true;
}
catch (e) {
    this.close();
    return false;
}
}

// Users of the EmailSocket do not need to be concerned with
// the following method. It is an implementation helper.

// local function to send a command & check a POP3 reply
// throws in case of error
EmailSocket.prototype._POP3 = function (cmd) {
    if (this._debug)
        writeln ("SEND: " + cmd);
    if (!this.writeln (cmd))
        throw "Error";
    var reply = this.read();
    if (this._debug)
        write ("RECV: " + reply);
    // the reply starts by either + or -
    if (reply [0] == "+")
        return;
    throw "Error";
}

// Users of the EmailSocket do not need to be concerned with
// the following method. It is an implementation helper.

// local function to send a command & check a SMTP reply
// throws in case of error
EmailSocket.prototype._SMTP = function (cmd) {
    if (this._debug)
        writeln ("SEND: " + cmd);
```

```

    if (!this.writeln (cmd))
        throw "Error";
    var reply = this.read();
    if (this._debug)
        write ("RECV: " + reply);
    // the reply is a three-digit code followed by a space
    var match = reply.match (/^(\d{3})\s/m);
    if (match.length == 2) {
        var n = Number (match [1]);
        if (n >= 200 && n <= 399)
            return;
    }
    throw "Error";
}

// nice to have: a toString()
// This function allows the email object to be printed.

EmailSocket.prototype.toString = function() {
    return "[object Email]";
}

```

## Email Setup

A simple script that prompts the user for the server name, email sender, and email recipient which are saved as Settings for the Render and Email script (above). You can run this script as standalone any time you want to change the settings. The script will run email\_setup.jsx whenever the settings don't exist; under normal circumstances this would only happen the first time, or if the settings/preferences file is deleted.

This script is a good example of how a script can create settings which are saved in Preferences for the sole use of scripting (as opposed to altering existing After Effects Preferences settings).

```

{
    // This script sets up 3 email settings.
    // It can be run all by itself, but it is also called
    // within "3-Render and Mail.jsx" if the settings aren't yet set.

    var serverValue = prompt("Enter name of mail server:");
    var fromValue = prompt("Enter reply-to email address:");
    var toValue = prompt("Enter recipient's email address");
    if (serverValue != null && serverValue != "") {

```

```
        app.settings.saveSetting("Email Settings", "Mail Server",
        serverValue);
    }
    if (fromValue != null && fromValue != "") {
        app.settings.saveSetting("Email Settings", "Reply-to Address",
        fromValue);
    }
    if (toValue != null && toValue != "") {
        app.settings.saveSetting("Email Settings", "Render Report
        Recipient", toValue);
    }
}
```

Other scripts included on your After Effects CD

## Dialogs and Console

This script shows how to use the various dialogs (`alert()`, `prompt()`, `confirm()`) and how to write to the info palette (`write()`, `writeln()` and `clearOutput()`). Although this script serves no practical use, these dialogs and info palette prompts are highly useful and should be familiar to all script creators.

```
// Use confirm() to let the user tell us whether he can see the
"info" window.
// Depending how the user clicks, true or false is returned.
if (confirm("Can you see the \"info\" palette?")){

    // Start by clearing the information area.
    clearOutput();

    // write and writeln will write to the info tab with or without a
    //'newline'
    // at the end.
    write("Roses are red,");
    writeln("violets are blue");
    write("Sugar is sweet,");
    writeln("and so is Equal.");

    var reply = prompt( "Did you like my poem?");
    if (reply == "yes" || reply == "YES"){
        alert("See the info window for a special secret fortune.");
        // This gets rid of the old writing on the info tab.
        clearOutput();
    }
}
```

```
        writeln("You have a future as a literary critic.");
    }
    else {
        alert("Hmm, I'll try once more...");
        writeln(".....");
        writeln("Roses are red, violets are blue,");
        writeln("I've got some gum, on the sole of my shoe.");
    }

    alert("Okay, all done with this test.");
}
else {
    // alert() just displays a message in a dialog box.
    alert("Please make it so you can see the info palette and run this
script
again");
}
```

## File Fun

This script shows how to open files, open projects, collect names of the Comps in the scene, prompt a user for where to write a file, write to a text file, and save the text file. It is useful only as an example of how the individual methods and attributes operate; it doesn't serve any useful production purpose.

```
// First, close any project that might be open.
if (app.project != null){
    // 3 choices here, CloseOptions.DO_NOT_SAVE_CHANGES,
    // CloseOptions.PROMPT_TO_SAVE_CHANGES, and CloseOptions.SAVE_CHANGES
    app.project.close(CloseOptions.DO_NOT_SAVE_CHANGES);
}

// Prompt the user to pick a project file:
// First argument is a prompt, second is the file type.
var pfile = fileGetDialog("Select a project file to open", "EggP
aep");

if (pfile == null){
    alert("No project file selected. Aborting.");
} else {
    // Open that file. It becomes the current project.
    var my_project = app.open( pfile );
}
```

```
// Build a default text file name from the project's filename.
// Remove the ".aep" file extension (if present), then add
//_compnames.txt.
var default_text_filename;
var suffix_index = pfile.name.lastIndexOf(".aep");
if (suffix_index != -1){
    default_text_filename = pfile.name.substring(0,suffix_index);
}else {
    default_text_filename = pfile.name;
}
default_text_filename += "_compnames.txt";

// Create another file object for the file we'll write out.
// First argument is the prompt, second is a default file name,
third is
//the file type.
var text_file = filePutDialog("Select a file to output your
results",
    default_text_filename, "TEXT txt");

if (text_file == null){
    alert("No output file selected. Aborting.");
} else {
    // opens file for writing. First argument is mode ("w" for
writing),
    // second argument is file type (for mac only),
    // third argument is creator (mac only, "?????" is no specific
app).
    text_file.open("w","TEXT","?????");

    // Write the heading of the file:
    text_file.writeln("Here is a list of all the comps in " +
pfile.name);
    text_file.writeln();
    for (var i = 1; i <= app.project.numItems; i++){
        if (app.project.item(i) instanceof CompItem){
            text_file.writeln(app.project.item(i).name);
        }
    }
}
```



```
text_file.close();  
  
alert("All done!");  
}  
}
```

# The Socket Object

## Introduction

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, or `read()` or `write()` to transfer data. The object also contains a `listen()` method to establish 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 (who listens on port 80); it then send 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 above code, the variable `homepage` contains the contents of the Adobe home page together with a 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
    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.

## JavaScript Reference

### Properties

<code>connected</code>	Boolean	Contains <code>true</code> if the connection is still active. Read only.
<code>eof</code>	Boolean	This property has the value <code>true</code> if the receive buffer is empty. Read only.

error	String	Contains a message describing the last error. Setting this value clears any error message.
host	String	Contains 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. Defaults to 10 (ten seconds).

## Methods

```
[new] Socket ( );
```

Creates a new Socket object.

## Returns

Object.

```
close();
```

Terminates the open connection. The return value is true if the connection was closed, false on I/O errors. Deleting the connection has the same effect. Remember, however, that JavaScript garbage collects the object at some null time, so the connection may stay open longer than you want to if you do not close it explicitly.

## Returns

Boolean

```
listen (Number port [, String encoding]);
```

Instructs the object to start listening for an incoming connection. The port argument is the TCP/IP port number where the object should listen on; typical values are 80 for a Web server, 23 for a Telnet server and so on. The encoding parameter is optional. The call to listen() is mutually exclusive to a call to open(). The result is true if the connection object successfully started listening, false otherwise.

## Parameters

port	Number	The port number to listen on. Valid port numbers are 1 to 65535.
encoding	String	The encoding to be used for the connection. Typical values are "ASCII", "binary", or "UTF-8". This parameter defaults to ASCII.

## Returns

Boolean

```
open (String computer [, String encoding]);
```

Open the connection for subsequent read/write operations. The computer name is the name or IP address, followed by a colon and the port number to connect to. The port number is mandatory. Valid computer names are e.g. "www.adobe.com:80" or "192.150.14.12:80". The encoding parameter is optional; currently, it can be one of "ASCII", "binary" or "UTF-8". The call to open() is mutually exclusive to a call to listen().

### Parameters

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 mandatory. Valid computer names are e.g. "www.adobe.com:80" or "192.150.14.12:80".
encoding	String	The encoding to be used for the connection. Typical values are "ASCII", "binary", or "UTF-8". This parameter defaults to ASCII.

### Returns

Boolean

```
poll();
```

Check 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 new Socket object or null.

```
read ([Number count]);
```

Read up to the given number of characters from the connection. Returns a string that contains up to the number of characters that were supposed to be read. If no count is supplied, the connection attempts to read as many characters it can get until the remote server closes the connection or a timeout occurs.

### Parameters

count	Number	The number of characters to read. If no count is supplied, the connection attempts to read as many characters it can get until the remote server closes the connection or a timeout occurs.
-------	--------	---

### Returns

String

```
readln();
```

Read one line of text up to the next line feed. Line feeds are recognized as CR, LF, CRLF or LFCR pairs.

### Returns

String

```
write (String text, ...);
```

Write the given string to the connection. The parameters of this function are concatenated to a single string. Returns true on success.

### Parameters

text	String	All arguments are concatenated to form the string to be written.
------	--------	--

### Returns

Boolean

```
writeln (String text, ...);
```

Write the given string to the connection and append a Line Feed character. The parameters of this function are concatenated to a single string. Returns true on success.

### Parameters

text	String	All arguments are concatenated to form the string to be written.
------	--------	--

### Returns

Boolean

## 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".

```
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);
    }
}
```

```
}  
}
```



# Encoding Names

## Supported encoding names

The following list of names is a basic set of encoding names supported by the FileSystem 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.

Note, however, that the FileSystem object cannot process extended Unicode character with values greater than 65535. These characters are left encoded as specified in the UTF-16 standard in as two characters in the range from 0xD700-0xDFFF.

Built-in encodings are:

US-ASCII, ASCII, ISO646-US, ISO-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. This encoder is convenient for reading and writing binary data.



## Additional encodings

In Windows, all encodings use so-called code pages. These code pages are assigned numeric values. The usual Western character set that Windows uses is e.g. the code page 1252. Windows code pages may be selected by prepending the number of the code page with "CP" or "WINDOWS-" like "CP1252" for the code page 1252. The File object has a lot of other encoding names built-in that match predefined code page numbers. If a code page is not present, the encoding cannot be selected.

On the Macintosh, encoders may be selected by name rather than by code page number. The File object queries the Macintosh OS directly for an encoder. As far as Macintosh character sets are identical with Windows code pages, the Macintosh also knows the Windows code page numbers.

## Common encoding names

The following encoding names are implemented both on Windows and Macintosh systems:

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 Macintosh encoding names

These names are alias names for encodings that the Macintosh operating system 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

# Index

---

## A

- accessing and writing scripts [6](#)
- activating
  - full scripting features [5](#)
  - JavaScript Debugger [6](#)
- After Effects objects [4](#)
- Application Object [23](#)
- Attributes, reference for [21](#)

## B

- breakpoints
  - Script Breakpoints Window [15](#)
  - setting [14](#)
  - setting in JavaScript code [15](#)
  - setting in the Script Breakpoints Window [16](#)
  - setting in the Script Debugger Window [15](#)

## C

- Chat server sample [110](#)
- code execution
  - controlling in the Script Debugger Window [13](#)
- Command line entry field [14](#)
- Common encoding names [114](#)
- Console [102](#)

## D

- Debugger Object (\$) [16](#)
- Dialogs [102](#)

## E

- ECMAScript Language Specification [18](#)
- email Methods [97](#)
- email Setup [101](#)

- encoding names, supported [113](#)
- encodings
  - additional [114](#)
  - additional Macintosh encoding names [115](#)
  - additional Windows encoding names [115](#)
- expressions and motion math [4](#)

## F

- File Fun script [103](#)

## G

- Global Functions [22](#)
- Globals, reference for [21](#)

## J

- JavaScript command line entry field, using [14](#)
- JavaScript Debugger window
  - about [11](#)
  - activating [6](#)
- JavaScript Reference [107](#)

## K

- Keywords syntax [18](#)

## L

- learning resources
  - additional [10](#)
  - what you should know [5](#)

## M

- Methods,reference for [21](#)

## N

- New render locations script [90](#)



**O**

## Objects

- After Effects [4](#)
- Application [23](#)
- reference for [21](#)

Operators [19](#)**R**

- recordable actions and scripts [6](#)
- reference for Objects, Methods, Attributes, and Globals [21](#)
- Render and Mail [94](#)
- Render named items script [89](#)
- resources to learn scripting [10](#)

**S**

- sample, Chat Server [110](#)
- Save and increment [88](#)
- Script Breakpoints Window [15](#)
- scripting
  - what you should know [5](#)

scripting, uses of [6](#)

Scripts folder and menu [7](#)

scripts included with After Effects [88](#)

setting

- breakpoints [14](#)

- breakpoints in JavaScript code [15](#)

- breakpoints in the Script Breakpoints Window [16](#)

- breakpoints in the Script Debugger Window [15](#)

shutdown [7](#)

Smart Import [91](#)

startup [7](#)

statement syntax [18](#)

supported encoding names [113](#)

**T**

testing and troubleshooting [10](#)

**U**

Uses of After Effects scripting [6](#)