macromedia®
DREAMWEAVER™ 3

Extending Dreamweaver

CONTENTS

CHAPTER 15

Data Translators 273

INDEX 295

CHAPTER 1
Extending Dreamweaver Overview

You can extend Dreamweaver several ways using HTML, JavaScript and C. You can write your own objects, behavior actions, commands, property inspectors, floating palettes, data translators, and C libraries that can be called from within JavaScript code. To accomplish these tasks, you must be proficient in JavaScript and thoroughly understand HTML elements. You don't need to be proficient in C unless you need some functionality that is not available through the Dreamweaver JavaScript API.

This manual describes the Document Object Model (DOM) that Dreamweaver supports and the Dreamweaver JavaScript API—the custom JavaScript functions that are built into Dreamweaver and that can be used in any object, behavior action, command, property inspector, floating palette, or data translation file. The manual also describes the APIs that are specific to each type of file, and provides several commented code examples.

Throughout this book, object, behavior action, command, property inspector, floating palette, and data translation files are referred to collectively as *extensions*.

Prerequisites

Because Dreamweaver extensions must be written in JavaScript, this manual assumes that readers are familiar with JavaScript syntax and basic programming concepts such as functions, variables, statements, operators, conditionals, and loops. This manual does not attempt to teach programming in general or JavaScript in particular.

If you are not yet an advanced JavaScript developer, but are familiar with the syntax of the language, start with objects to learn about extensibility. First read "The Document Object Model" on page 9, and then see "C-Level Extensibility" on page 209 for more information.

Anyone who wants to extend Dreamweaver should have a good JavaScript reference to help with syntax questions (for example, is it substring() or subString()?). Useful JavaScript references include *Javascript Bible* by Danny Goodman (IDG), *JavaScript: The Definitive Guide* by David Flanagan (O'Reilly), and *Pure JavaScript* by R. Allen Wyke, Jason D. Gilliam, and Charlton Ting (Sams). For a free JavaScript reference, see Netscape's DevEdge Online website at http://developer.netscape.com:80/docs/manuals/javascript.html.

Errata

This documentation was written before the code in Dreamweaver 3 was complete. Thus, there may be some discrepancies between the final implementation of the JavaScript API in Dreamweaver 3 and how it is documented in this book. A list of known issues can be found in the Extensibility section of the Dreamweaver Support Center at http://www.macromedia.com/support/dreamweaver/extend.html.

CHAPTER 2
The Document Object Model

HTML documents consist of a tree of tags that reveals the document's structure. The root of the tree is the HTML tag. The two largest branches of the tree are HEAD and BODY. Offshoots of HEAD include TITLE, STYLE, SCRIPT, ISINDEX, BASE, META, and LINK. Offshoots of BODY include headings (H1, H2, and so on), block-level elements (P, DIV, FORM, and so on), text-level elements (FONT, BR, IMG, and so on), and the ADDRESS element. Leaves on the offshoots include attributes such as WIDTH, HEIGHT, ALT, and HREF.

A document object model, or DOM, is also a tree that discloses the document's structure. The DOM, however, reports this structure in terms of objects and properties, rather than in terms of tags and attributes.

The root of the DOM tree is the document itself, the HTML object is the trunk, and the rest of the objects in the document branch from the HTML object as the HTML tags and attributes do.

The document object model in Dreamweaver

A browser's DOM determines how the JavaScript in an HTML document works in that browser. Similarly, Dreamweaver's DOM determines how the JavaScript in extensions works in Dreamweaver.

Dreamweaver's DOM combines a subset of the Netscape Navigator 4.0 DOM with a subset of the World Wide Web Consortium (W3C)'s DOM Level 1. With the incorporation of DOM Level 1, every part of an HTML page has become an object—including tags (which the W3C calls elements), comments, and text.

Objects can be referred to by index (document.forms[3].elements[1]) or by name (document.myForm.myButton). Objects with the same name are collapsed into an array. You can access a particular object in the group by index (for example, the first radio button with the name myRadioGroup in myForm would be referenced as document.myForm.myRadioGroup[0]).

The following table gives an overview of the properties, methods, and events supported by each object; these are described in more detail in books such as *JavaScript: The Definitive Guide* (O'Reilly). Additional details about the W3C properties and methods, which are less thoroughly documented by third parties, follow the table. A bullet (•) marks read-only properties.

Object	Properties	Methods	Events
window	document • navigator • innerWidth • innerHeight • screenX • screenY •	alert() confirm() escape() unescape() close() setTimeout() clearTimeout() setInterval() clearInterval() resizeTo()	onResize
navigator	platform •	None	None

Object	Properties	Methods	Events
document	forms • (an array of form objects) images • (an array of image objects) layers • (an array of LAYER, ILAYER, and absolutely positioned DIV and SPAN objects) *child objects by name* • nodeType • parentNode • childNodes • documentElement • body • URL • parentWindow •	getElementsByTagName() hasChildNodes()	onLoad
all tags/elements	nodeType • parentNode • childNodes • tagName • *attributes by name* innerHTML outerHTML	getAttribute() setAttribute() removeAttribute() getElementsByTagName() hasChildNodes()	
form	**In addition to the properties available for all tags:** **elements • (an array of button, checkbox, password, radio, reset, select, submit, text, file, hidden, image, and textarea objects)** *child objects by name* •	**Only those methods available to all tags.**	**None**
layer	**In addition to the properties available for all tags:** visibility left top width height zIndex	**Only those methods available to all tags.**	**None**

Object	Properties	Methods	Events
image	In addition to the properties available for all tags: src	Only those methods available to all tags.	onMouseOver onMouseOut onMouseDown onMouseUp
button reset submit	In addition to the properties available for all tags: form •	In addition to the methods available for all tags: blur() focus()	onClick
checkbox radio	In addition to the properties available for all tags: checked form •	In addition to the methods available for all tags: blur() focus()	onClick
password text file hidden image (field) textarea	In addition to the properties available for all tags: form • value	In addition to the methods available for all tags: blur() focus() select()	onBlur onFocus
select	In addition to the properties available for all tags: form • options • (an array of option objects) selectedIndex	In addition to the methods available for all tags: blur() (Windows only) focus() (Windows only)	onBlur (Windows only) onChange onFocus (Windows only)
option	In addition to the properties available for all tags: text	Only those methods available to all tags.	None
array boolean date function math number object string regexp	Matches Netscape 4	Matches Netscape 4	None

Object	Properties	Methods	Events
text	nodeType • parentNode • childNodes • data	hasChildNodes()	None
comment	nodeType • parentNode • childNodes • data	hasChildNodes()	None
nodelist	length •	item()	None

While Dreamweaver has a DOM that resembles that of a browser, the property inspectors, floating palettes, and parameters and options dialog boxes associated with extensions are not browsers. For this reason, links (A tags) are not supported. In addition, while live plugins (set to play at all times) are supported in the BODY of extensions, Java applets and ActiveX controls are not.

The dreamweaver object and its properties

Dreamweaver implements the standard objects as defined by the browsers and the W3C, as well as two custom objects: dreamweaver and site. The dreamweaver object has two read-only properties associated with it: appName and appVersion.

appName always has the value "Dreamweaver". appVersion has a value of the form "*versionNumber* [*languageCode*] (*platform*)". For example, the value of the appVersion property for the Swedish Windows version of Dreamweaver 3 would be "3.0 [se] (Win32)"; the value for the English Macintosh version would be "3.0 [en] (MacPPC)".

The appName and appVersion properties were implemented in Dreamweaver 3 and are not available in earlier versions of Dreamweaver. To determine whether the version of Dreamweaver is 3 or later, you can simply check for the existence of the appVersion or appName property. To check for a specific version of Dreamweaver, check first for the existence of appVersion and then for the version number. For example:

```
if (dreamweaver.appVersion && dreamweaver.appVersion.indexOf('3.01') != -1){
 // execute code
}
```

The site object has no properties. For more information about the methods of the dreamweaver and site objects, see "The Dreamweaver JavaScript API" on page 19.

DOM details

Unlike the Netscape DOM, DOM Level 1 has not been documented in hundreds of third-party books and web sites. Thus, this document will describe the DOM Level 1 properties and methods, and the values they return, in some detail.

DOM Level 1 introduces four constants that describe the types of objects that make up the document tree (called *nodes*). These constants, which usually show up as return values for the nodeType property, are:

```
Node.DOCUMENT_NODE
Node.ELEMENT_NODE
Node.COMMENT_NODE
Node.TEXT_NODE
```

Properties and methods of the document object

The following table lists the new properties and methods of the document object in Dreamweaver, along with their return values (with explanations, where appropriate). A bullet (•) marks read-only properties.

Property or method	Return value and explanation
nodeType •	Node.DOCUMENT_NODE
parentNode •	null
parentWindow •	The JavaScript object corresponding to the document's parent window. (This property is not included in DOM Level 1; however, it is supported by IE 4.0.)
childNodes •	A nodelist containing all the immediate children of the document object. Typically the document will have a single child: the HTML object.
documentElement •	The JavaScript object corresponding to the HTML tag. This property is shorthand for getting the value of document.childNodes and extracting the HTML tag from the nodelist.
body •	The JavaScript object corresponding to the BODY tag. This property is shorthand for calling document.documentElement.childNodes and extracting the BODY tag from the nodelist. For frameset documents, this property returns the node for the outermost frameset instead.
URL •	The file:// URL for the document or, if the file has not been saved, an empty string.

Property or method	Return value and explanation
getElementsByTagName(*tagName*)	A nodelist that can be used to step through tags of type *tagName* (for example, IMG, DIV, and so on). Provides similar functionality to the dreamweaver.getObjectTags() function.
	If the *tag* argument is LAYER, the function returns all LAYER and ILAYER tags and all absolutely positioned DIV and SPAN tags.
	If the *tag* argument is INPUT, the function returns all form elements. (For this shortcut to work properly, all form field names must begin with a letter.)
hasChildNodes()	TRUE

Properties and methods of HTML tag objects

Every HTML tag is represented by a JavaScript object. Tags are organized in a tree hierarchy, where tag x is a parent of tag y if y falls completely within x's opening and closing tags (<x>x content surrounding <y>y content</y></x>). The following table lists the properties and methods of tag objects in Dreamweaver, along with their return values (with explanations, where appropriate). A bullet (•) marks read-only properties.

Property or method	Return value and explanation
nodeType •	Node.ELEMENT_NODE
parentNode •	The parent tag. If this is the HTML tag, then the document object is returned instead.
childNodes •	A nodelist containing all the immediate children of the tag.
tagName •	The HTML name for the tag, such as IMG, A, or BLINK. This value is always returned in uppercase letters.
attrName	A string containing the value of the specified tag attribute. tag.*attrName* cannot be used if *attrName* is a reserved word in the JavaScript language (for example, class). In this case, use getAttribute() and setAttribute() instead.
innerHTML	The HTML source contained between the begin tag and the end tag. For example, in the code Hello, World! , p.innerHTML would return Hello, World!. If you write to this property, the DOM tree is immediately updated to reflect the new structure of the document. (This property is not included in DOM Level 1; however, it is supported by IE 4.0.)

Property or method	Return value and explanation
outerHTML	The HTML source for this tag, including the tag itself. For the example code above, p.outerHTML would return `<p><b>Hello</b>, World!</p>`. If you write to this property, the DOM tree is immediately updated to reflect the new structure of the document. (This property is not included in DOM Level 1; however, it is supported by IE 4.0.)
getAttribute(*attrName*)	The value of the specified attribute if it is explicitly specified; otherwise, null.
getTranslatedAttribute(*attrName*)	The translated value of the specified attribute, or the same value that would be returned by getAttribute() if the attribute's value is not translated. (This property is not included in DOM Level 1; it was added to Dreamweaver 3 to support attribute translation.)
setAttribute(*attrName, attrValue*)	No return value. Sets the specified attribute to the specified value: for example, img.setAttribute("src", "image/roses.gif").
removeAttribute(*attrName*)	No return value. Removes the specified attribute and its value from the HTML for this tag.
getElementsByTagName(*tagName*)	A nodelist that can be used to step through child tags of type *tagName* (for example, IMG, DIV, and so on).
	If the *tag* argument is LAYER, the function returns all LAYER and ILAYER tags and all absolutely positioned DIV and SPAN tags.
	If the *tag* argument is INPUT, the function returns all form elements. (For this shortcut to work properly, all form field names must begin with a letter.)
hasChildNodes()	A Boolean value indicating whether the tag has any children.
hasTranslatedAttributes()	A Boolean value indicating whether the tag has any translated attributes. (This property is not included in DOM Level 1; it was added to Dreamweaver 3 to support attribute translation.)

Properties and methods of text objects

Each contiguous block of text in an HTML document (for example, the text within a P tag) is represented by a JavaScript object. Text objects never have children. The following table lists the properties and methods of text objects in Dreamweaver, along with explanations where appropriate. A bullet (•) marks read-only properties.

Property or method	Return value and explanation
nodeType •	Node.TEXT_NODE
parentNode •	The parent tag.
childNodes •	An empty nodelist.
data	The actual text string. Entities in the text are represented as a single character (for example, the text Joseph & I is returned as Joseph & I).
hasChildNodes()	FALSE

Properties and methods of comment objects

Each HTML comment is represented by a JavaScript object. The following table lists the properties and methods of comment objects in Dreamweaver, along with explanations where appropriate. A bullet (•) marks read-only properties.

Property or method	Return value and explanation
nodeType •	Node.COMMENT_NODE
parentNode •	The parent tag.
childNodes •	An empty nodelist.
data	The text string between the comment markers (<!-- and -->).
hasChildNodes()	FALSE

How JavaScript works in extensions

When Dreamweaver processes extensions, it compiles everything between SCRIPT tags and executes any code within SCRIPT tags that is not part of a function declaration (for example, initialization of global variables). It also reads in, compiles, and executes scripts in external JavaScript files specified in the SRC attributes of SCRIPT tags.

Note: If any JavaScript code in your extension files contains the string '</SCRIPT>', the JavaScript interpreter reads this as an actual closing SCRIPT tag and reports an unterminated string literal error. To avoid this problem, break the string into pieces and concatenate them, like this: '<' + '/SCRIPT>'.

In command and behavior action files, Dreamweaver executes code in the onLoad event handler (if one appears in the BODY tag) when the user chooses the command or action from a menu. In object files, Dreamweaver executes code in the onLoad event handler on the BODY tag if the body of the document contains a form. Dreamweaver ignores the onLoad handler on the BODY tag in data translator, property inspector, and floating palette files. In all extensions, Dreamweaver executes code in other event handlers (for example, onBlur="alert('This is a required field.')") when the user interacts with the form fields to which they are attached.

JavaScript URLs (for example, <a href="javascript:alert('hi')">) are not supported, nor is the document.write() statement.

Running scripts at startup or shutdown

In Dreamweaver 3, if you place a command file in the Configuration/Startup folder, the command runs as Dreamweaver is starting up. Startup commands load before the menus.xml file, before the files in the ThirdPartyTags folder, and before any other commands, objects, behaviors, inspectors, floating palettes, or translators. Thus, you can use startup commands to modify the menus.xml file or other extension files. You can also show warnings or prompt the user for information, but you cannot call dreamweaver.runCommand().

Similarly, if you place a command file in the Configuration/Shutdown folder, the command will run as Dreamweaver is shutting down. You can call "dreamweaver.runCommand()" on page 37 from shutdown commands, show warnings, or prompt the user for information, but you cannot stop the shutdown process.

For more information about commands, see "Commands" on page 227.
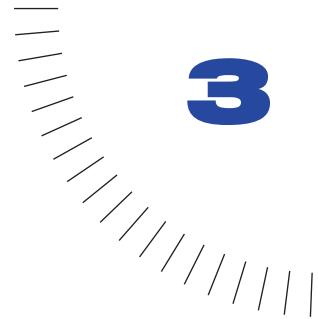
CHAPTER 3
The Dreamweaver JavaScript API

The objects, properties, and especially the methods described in "The Document Object Model" on page 9, were a good foundation on which to build extensibility into Dreamweaver. Several tasks unique to authoring environments cannot be accomplished with the methods available in the Netscape, Microsoft, or W3C DOMs, however. The most obvious example of this is selection, which is integral to the user experience in an authoring environment: DOM Level 1 and the proprietary browser DOMs do not address selection because users cannot select and modify content (except in form fields) in a browser window.

To make creating useful Dreamweaver extensions and customizing Dreamweaver menus possible, Dreamweaver exposes more than 400 JavaScript functions to developers beyond the standards-based DOM methods. This represents a huge expansion over what was available in Dreamweaver 2. Almost any task that the user can accomplish in Dreamweaver with the menus, floating palettes, inspectors, Site window, or Document window can now be done with JavaScript.

Understanding the objects in the API

All of the custom functions in the following sections are methods of the dreamweaver object, the site object, or the object that represents a document's DOM. Methods that fit into the latter category are listed here as dom.*functionName()*. To produce the desired results, you must first get the DOM of a document and call the functions as methods of that DOM. You cannot simply type dom.*functionName()*. For example:

```
var currentDOM = dreamweaver.getDocumentDOM('document');
currentDOM.setSelection(100,200);
currentDOM.clipCopy();
var otherDOM = dreamweaver.openDocument(dreamweaver.getSiteRoot() + "html/
foo.htm");
otherDOM.endOfDocument();
otherDOM.clipPaste();
```

Unless otherwise noted, methods of the dom object can only operate on open documents, and running a function on a document that isn't open will cause Dreamweaver to report an error. Methods of the dom object that can operate only on the active document or on closed documents indicate this fact in their descriptions.

In Dreamweaver 3, dw is synonymous with dreamweaver. Thus all of the methods of the dreamweaver object can be referred to as dw.*functionName()*. This notation appears in code examples throughout this and following chapters.

About enablers

Because every menu item in Dreamweaver 3 is implemented as a JavaScript function, a JavaScript mechanism is required to determine when menu items should be enabled. This mechanism is a series of functions called *enablers*.

An enabler determines whether its associated main function can operate in the current context. For example, site.canGet() determines whether Dreamweaver can perform a Get operation (site.get()).

Each function specification in this API lists the enabler for the function, if applicable. (Some functions do not have enablers, either because the menu item associated with the function is always enabled, or because the function is unrelated to menus.) Function specifications for the enablers appear at the end of this chapter.

How this chapter is organized

The methods in the Dreamweaver JavaScript API are grouped functionally and then alphabetically by object and then by method name. For example, methods that deal with creating, applying, and deleting CSS styles are grouped under CSS Style functions; methods of the dom object are listed first, followed by methods of the dreamweaver object, followed by methods of the dreamweaver.cssStylePalette object. Optional arguments are enclosed in curly braces ({ }).

The most important function

Virtually all of the functions involved in DOM manipulations require that you first determine which DOM to change. This is the task of dreamweaver.getDocumentDOM(), and thus it is the most important function.

dreamweaver.getDocumentDOM()

Availability 2.0

Description Provides access to the tree of objects for the specified document. After the tree of objects is returned to the caller, the caller can edit the tree to change the contents of the document.

Arguments *sourceDoc*

The argument must be "document", "parent", "parent.frames[*number*]", "parent.frames['*frameName*']", or a URL. document specifies the document that has the focus and contains the current selection. parent specifies the parent frameset (if the currently selected document is in a frame), and parent.frames[*number*] and parent.frames['*frameName*'] specify a document that is in a particular frame within the frameset containing the current document. If the argument is a relative URL, it is relative to the extension file. In Dreamweaver 3, *sourceDoc* defaults to document if omitted.

Note: If the argument is "document", the caller must be applyBehavior(), deleteBehavior(), objectTag(), or any function in a command or property inspector file in order to perform edits to the document.

Returns The JavaScript document object at the root of the tree.

Enabler None.

Example The following code snippet uses dreamweaver.getDocumentDOM() to access the background color of the current document:

```
var theDOM = dreamweaver.getDocumentDOM("document");
theDOM.body.bgcolor = "#000000";
```

Behavior functions

Behavior functions let you add behaviors to and remove behaviors from an object, find out which behaviors are attached to an object, get information about the object to which a behavior is attached, and so on. Methods of the dreamweaver.behaviorInspector object either control or act on the selection in the Behavior inspector, not in the current document.

dom.addBehavior()

Availability	3.0
Description	Adds a new event/action pair to the selected element. This function is valid only for the active document.
Arguments	*event, action, {eventBasedIndex}*

- The first argument is the JavaScript event handler that should be used to attach the behavior to the element; for example, onClick, onMouseOver, or onLoad.

- The second argument is the function call that would be returned by applyBehavior() if the action were added using the Behavior inspector; for example, "MM_popupMsg('Hello World')".

- The third argument is the position at which this action should be added. *eventBasedIndex* is a zero-based index; if two actions already are associated with the specified event, and you specify *eventBasedIndex* as 1, this action will be executed between the other two. If this argument is omitted, the action is added after all existing actions for the specified event.

Returns	Nothing.
Enabler	None.

dom.getBehavior()

Availability 3.0

Description Gets the action at the specified position within the specified event. This function acts on the current selection and is valid only for the active document.

Arguments *event, {eventBasedIndex}*

- The first argument is the JavaScript event handler through which the action is attached to the element; for example, onClick, onMouseOver, or onLoad.

- The second argument is the position of the action to get. For example, if two actions are associated with the specified event, 0 is the first one and 1 is the second. If this argument is omitted, all the actions for the specified event are returned.

Returns A string representing the function call (for example, "MM_swapImage('document.Image1','document.Image1','foo.gif','#933292969950')"), or an array of strings if *eventBasedIndex* was omitted.

Enabler None.

dom.reapplyBehaviors()

Availability 3.0

Description Checks to make sure that the functions associated with any behavior calls on the specified node are in the HEAD of the document, and inserts them if they are missing.

Arguments *{elementNode}*

The argument is a an element node within the current document. If the argument is omitted, Dreamweaver checks all element nodes in the document for orphaned behavior calls.

Returns Nothing.

Enabler None.

dom.removeBehavior()

Availability   3.0

Description   Removes the action at the specified position within the specified event. This function acts on the current selection and is valid only for the active document.

Arguments   *event, {eventBasedIndex}*

- The first argument is the event handler through which the action is attached to the element; for example, onClick, onMouseOver, or onLoad. If this argument is omitted, all actions are removed from the element.

- The second argument is the position of the action to be removed. For example, if two actions are associated with the specified event, 0 is the first one and 1 is the second. If this argument is omitted, all the actions for the specified event are removed.

Returns   Nothing.

Enabler   None.

dreamweaver.getBehaviorElement()

Availability	2.0
Description	Gets the DOM object corresponding to the tag to which the behavior is being applied. This function is applicable only in behavior action files.
Arguments	None.
Returns	A DOM object or null. This function returns null under the following circumstances:

- When the current script is not executing within the context of the Behavior inspector.

- When the Behavior inspector is being used to edit a behavior in a timeline.

- When the currently executing script was invoked by dreamweaver.popupAction().

- When the Behavior inspector is attaching an event to a link wrapper and the link wrapper does not yet exist.

- When this function appears outside of an action file.

Enabler	None.
Example	dreamweaver.getBehaviorElement() can be used in the same way as dreamweaver.getBehaviorTag() to determine whether the selected action is appropriate for the selected HTML tag, except that it gives you access to more information about the tag and its attributes. For example, if you write an action that can be applied only to a hyperlink (A HREF) that does not target another frame or window, you can use getBehaviorElement() as part of the function that initializes the user interface for the parameters dialog box.

```
function initializeUI(){
  var theTag = dreamweaver.getBehaviorElement();
  var CANBEAPPLIED = (theTag.tagName == "A"
             && theTag.getAttribute("HREF") != null
             && theTag.getAttribute("TARGET") == null);
  if (CANBEAPPLIED) {
   // display the action UI
  } else{
   // display a helpful message that tells the user
   // that this action can only be applied to a
   // hyperlink without an explicit target]
  }
}
```

dreamweaver.getBehaviorTag()

|	
Availability	1.2
Description	Gets the source of the tag to which the behavior is being applied. This function is applicable only in action files.
Arguments	None.
Returns	A string representing the source of the tag. This is the same string that is passed as an argument (*HTMLelement*) to the canAcceptBehavior() function. If this function appears outside of an action file, the return value is an empty string.
Enabler	None.
Example	If you write an action that can be applied only to a hyperlink (A HREF), you can use getBehaviorTag() as part of the function that initializes the user interface for the parameters dialog box.

```
function initializeUI(){
 var theTag = dreamweaver.getBehaviorTag().toUpperCase();
 var CANBEAPPLIED = (theTag.indexOf('HREF') != -1));
 if (CANBEAPPLIED) {
  // display the action UI
 } else{
  // display a helpful message that tells the user
  // that this action can only be applied to a
  // hyperlink
 }
}
```

dreamweaver.popupAction()

Availability 2.0

Description Presents the user with a parameters dialog box for the specified behavior action. To the user, the effect is the same as selecting the action from the Actions pop-up menu in the Behavior inspector. This function lets extension files other than actions attach behaviors to objects in the user's document. It blocks other edits until the user dismisses the dialog box.

Note: This function can be called only within objectTag() or in any script in a command or property inspector file.

Arguments *actionName, {funcCall}*

- The first argument is the name of a file in the Configuration/Behaviors/Actions folder that contains a JavaScript behavior action (for example, "Timeline/Play Timeline.htm").

- The second argument is a string containing a function call for the action specified in actionName (for example, "MM_playTimeline(...)"). This argument, if specified, is supplied by the applyBehavior() function in the action file.

Returns The function call for the behavior action. When the user clicks OK in the parameters dialog box, the behavior is added to the current document (the appropriate functions are added to the HEAD of the document, HTML may be added to the top of the BODY, and other edits may be made to the document). The function call (for example, "MM_playTimeline(...)") is not added to document, but becomes the return value of this function.

Enabler None.

dreamweaver.behaviorInspector.getBehaviorAt()

Availability   3.0

Description   Gets the event/action pair at the specified position in the Behavior inspector.

Arguments   *positionIndex*

Returns   An array of two items:

- An event handler

- A function call or JavaScript statement

Enabler   None.

Example   Because positionIndex is a zero-based index, if the Behavior inspector displays the list shown, a call to dw.behaviorInspector.getBehaviorAt(2) returns an array containing two strings: "onMouseOver" and "MM_changeProp('document.moon','document.moon','src','sun.gif','IMG')".

dreamweaver.behaviorInspector.getBehaviorCount()

Availability   3.0

Description   Counts the number of actions attached to the currently selected element through event handlers.

Arguments   None.

Returns   An integer representing the number of actions attached to the element. This number is equivalent to the number of actions visible in the Behavior inspector and includes both Dreamweaver behavior actions and custom JavaScript.

Enabler   None.

**Example**   A call to dw.behaviorInspector.getBehaviorCount() for the selected link <A HREF="javascript:setCookie()" onClick="MM_popupMsg('A cookie has been set.');parent.rightframe.location.href='aftercookie.html'"> would return 2.

dreamweaver.behaviorInspector.getSelectedBehavior()

Availability	3.0
Description	Gets the position in the Behavior inspector of the selected action.
Arguments	None.
Returns	An integer representing the position in the Behavior inspector of the selected action, or –1 if no action is selected.
Enabler	None.
Example	If the first action in the Behavior inspector is selected, as shown, a call to dw.behaviorInspector.getSelectedBehavior() returns 0.
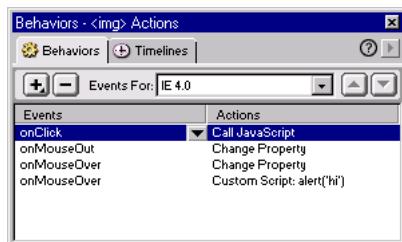
dreamweaver.behaviorInspector.moveBehaviorDown()

Availability   3.0

Description   Moves a behavior action lower in sequence by changing its execution order within the scope of an event.

Arguments   *positionIndex*

The argument is the position of the action in the Behavior inspector. The first action in the list is at position 0.

Returns   Nothing.

Enabler   None.

Example   Assuming the Behavior inspector setup shown in this example, calling dw.behaviorInspector.moveBehaviorDown(2) would swap the positions of the Custom Script and the Change Property actions on the onMouseOver event. Calling dw.behaviorInspector.moveBehaviorDown() for any other position would have no effect because the onClick and onMouseOut events each have only one behavior associated with them, and the behavior at position 3 is already at the bottom of the onMouseOver group.

dreamweaver.behaviorInspector.moveBehaviorUp()

Availability   3.0

Description   Moves a behavior higher in sequence by changing its execution order within the scope of an event.

**Arguments**   *positionIndex*

The argument is the position of the action in the Behavior inspector. The first action in the list is at position 0.

Returns   Nothing.

Enabler   None.

Example   Assuming the Behavior inspector setup shown in this example, calling dw.behaviorInspector.moveBehaviorUp(3) would swap the positions of the Custom Script and the Change Property actions on the onMouseOver event. Calling dw.behaviorInspector.moveBehaviorUp() for any other position would have no effect because the onClick and onMouseOut events each have only one behavior associated with them, and the behavior at position 2 is already at the top of the onMouseOver group.

dreamweaver.behaviorInspector.setSelectedBehavior()

Availability	3.0
Description	Selects the action at the specified position in the Behavior inspector.
Arguments	*positionIndex*
	The argument is the position of the action in the Behavior inspector. The first action in the list is at position 0. To deselect all actions, specify a *positionIndex* of –1. Specifying a position for which no action exists is equivalent to specifying –1.
Returns	Nothing.
Enabler	None.
Example	Assuming the Behavior inspector setup shown in this example, calling dw.behaviorInspector.setSelection(3) would select the Change Property action associated with the onMouseOut event.

Clipboard functions

Clipboard functions are all related to cutting, copying, and pasting. On the Macintosh, some clipboard functions can also apply to edit fields in dialog boxes and floating palettes. Functions that can operate in edit fields are implemented both as methods of the dreamweaver object and as methods of the dom object. The dreamweaver version of the function operates on the selection in the window that has focus: the current Document window, the HTML inspector, or the Site window. On the Macintosh, the function can also operate on the selection in an edit field if the field has focus. The dom version of the function always operates on the selection in the specified document.

dom.clipCopy()

Availability	3.0
Description	Copies the selection, including any HTML markup that defines the selection, to the clipboard.
Arguments	None.
Returns	Nothing.
Enabler	None.

dom.clipCopyText()

Availability	3.0
Description	Copies the selected text to the clipboard, ignoring any HTML markup.
Arguments	None.
Returns	Nothing.
Enabler	dom.canClipCopyText()

dom.clipCut()

|	
Availability	3.0
Description	Removes the selection, including any HTML markup that defines the selection, to the clipboard.
Arguments	None.
Returns	Nothing.
Enabler	None.

dom.clipPaste()

Availability   3.0

Description   Pastes the contents of the clipboard into the current document at the current insertion point or in place of the current selection. If the clipboard contains HTML, it is interpreted as such.

Arguments   None.

Returns   Nothing.

Enabler   dom.canClipPaste()

Example   If the clipboard contains <code>return true;</code>, a call to dw.getDocumentDOM().clipPaste() would result in the following:

dom.clipPasteText()

Availability	3.0
Description	Pastes the contents of the clipboard into the current document at the current insertion point or in place of the current selection, replacing any linefeeds in the clipboard content with BR tags. If the clipboard contains HTML, it is not interpreted; angle brackets are pasted as < and >.
Arguments	None.
Returns	Nothing.
Enabler	dom.canClipPasteText()
Example	If the clipboard contains <code>return true;</code>, a call to dw.getDocumentDOM().clipPasteText() would result in the following:
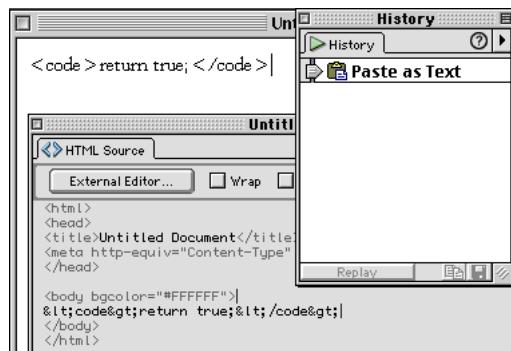
dreamweaver.clipCopy()

Availability	3.0
Description	Copies the current selection (from whichever Document window, dialog box, floating palette, or Site window pane has focus) to the clipboard.
Arguments	None.
Returns	Nothing.
Enabler	dreamweaver.canClipCopy()

dreamweaver.clipCut()

Availability	3.0
Description	Removes the selection (from whichever Document window, dialog box, floating palette, or Site window pane has focus) to the clipboard.
Arguments	None.
Returns	Nothing.
Enabler	dreamweaver.canClipCut()

dreamweaver.clipPaste()

Availability	3.0
Description	Pastes the contents of the clipboard into the current document, dialog box, floating palette, or Site window pane.
Arguments	None.
Returns	Nothing.
Enabler	dreamweaver.canClipPaste()

dreamweaver.getClipboardText()

Availability	3.0
Description	Gets all the text that is stored on the clipboard.
Arguments	None.
Returns	A string containing the contents of the clipboard, if the clipboard contains text (which may be HTML); otherwise, nothing.
Enabler	None.

Command functions

Command functions help you make the most of the files in the Configuration/Commands folder. They manage the Command menu and call commands from other types of extension files.

dreamweaver.editCommandList()

Availability	3.0
Description	Opens the Edit Command List dialog box.
Arguments	None.
Returns	Nothing.
Enabler	None.

dreamweaver.runCommand()

Availability 3.0

Description Executes the specified command. To the user, the effect is the same as choosing the command from a menu; if a dialog box is associated with the command, it appears (and the command script blocks other edits until the user dismisses the dialog box). This function provides the ability to call a command from another extension file.

Note: This function can be called only within the objectTag() function or in any script in a command, property inspector file, or menu.

Arguments *commandFile, {commandArg1}, {commandArg2},...{commandArgN}*

- The argument is the name of a file in the Configuration/Commands folder.

- The second and remaining arguments are passed to *commandFile* as arguments.

Returns Nothing.

Enabler None.

Example You can write a custom property inspector for tables that lets users get to the Format Table command from a button on the inspector by calling the following function from the button's onClick event handler:

```
function callFormatTable(){
  dw.runCommand('Format Table.htm');
}
```

Conversion functions

Conversion functions deal with converting tables to layers, layers to tables, and Cascading Style Sheets (CSS) styles to HTML markup. Each function exactly duplicates the behavior of one of the conversion commands in the File or Modify menu.

dom.convertLayersToTable()

Availability	3.0
Description	Opens the Convert Layers to Table dialog box.
Arguments	None.
Returns	Nothing.
Enabler	dom.canConvertLayersToTable()

dom.convertTablesToLayers()

Availability	3.0
Description	Opens the Convert Tables to Layers dialog box.
Arguments	None.
Returns	Nothing.
Enabler	dom.canConvertTablesToLayers()

dom.convertTo30()

Availability	3.0
Description	Opens the Convert to 3.0 Compatible dialog box.
Arguments	None.
Returns	Nothing.
Enabler	None.

CSS style functions

CSS style functions handle the application, removal, creation, and deletion of CSS styles. Methods of the dreamweaver.cssStylePalette object either control or act on the selection in the Style palette, not in the current document.

dom.applyCSSStyle()

Availability 3.0

Description Applies the specified style to the specified element. This function is valid only for the active document.

Arguments *elementNode, styleName,* {*classOrID*}

- The first argument is an element node in the DOM. If *elementNode* is specified as NULL or empty string (''), the function acts on the current selection.

- The second argument is the name of a CSS style.

- The third argument is the attribute with which the style should be applied (either "class" or "id"). If *elementNode* is specified as NULL or empty string and no tag exactly surrounds the selection, the style is applied using SPAN tags. If the selection is an insertion point, Dreamweaver uses heuristics to determine to which tag the style should be applied.

Returns Nothing.

Enabler None.

Example The following code applies the red style to the selection, either by surrounding the selection with SPAN tags or by applying a CLASS attribute to the tag that surrounds the selection:

```
var theDOM = dreamweaver.getDocumentDOM('document');
theDOM.applyCSSStyle('','red');
```

dom.removeCSSStyle()

Availability   3.0

Description   Removes the CLASS or ID attribute from the specified element, or removes the SPAN tag that completely surrounds the specified element. This function is valid only for the active document.

Arguments   *elementNode, {classOrID}*

- The first argument is an element node in the DOM. If *elementNode* is specified as an empty string (''), the function acts on the current selection.

- The second argument is the attribute that should be removed (either "class" or "id"). If *classOrID* is not specified, it defaults to "class". If no CLASS attribute is defined for *elementNode*, then the SPAN tag surrounding *elementNode* is removed.

Returns   Nothing.

Enabler   None.

dreamweaver.cssStylePalette.deleteSelectedStyle()

Availability   3.0

Description   Deletes from the document the style that is currently selected in the Style palette.

Arguments   None.

Returns   Nothing.

Enabler   None.

dreamweaver.cssStylePalette.duplicateSelectedStyle()

Availability   3.0

Description   Duplicates the style that is currently selected in the Style palette and displays the Duplicate Style dialog box to let the user assign a name or selector to the new style.

Arguments   None.

Returns   Nothing.

Enabler   None.

dreamweaver.cssStylePalette.editSelectedStyle()

Availability   3.0

Description   Opens the Style Definition dialog box for the style that is currently selected in the Style palette.

Arguments   None.

Returns   Nothing.

Enabler   None.

dreamweaver.cssStylePalette.editStyleSheet()

Availability   3.0

Description   Opens the Edit Style Sheet dialog box.

Arguments   None.

Returns   Nothing.

Enabler   None.

dreamweaver.cssStylePalette.getSelectedStyle()

Availability   3.0

Description   Gets the name of the style that is currently selected in the Style palette.

Arguments   None.

Returns   A string representing the name of the style, or an empty string if no style is selected.

Enabler   None.

Example   If the style red is selected, as shown, a call to dw.cssStylePalette.getSelectedStyle() returns "red".

dreamweaver.cssStylePalette.getSelectedTarget()

Availability   3.0

Description   Gets the selected element in the Apply To pop-up menu at the top of the Style palette.

Arguments   None.

Returns   The object to which the style should be applied, or NULL if the target is the current selection.

Enabler   None.

Example   Before applying a style, use dw.cssStylePalette.getSelectedTarget() in case the user has changed the target, as shown.

For example:

```
var currDOM = dw.getDocumentDOM();
currDOM.applyCSSStyle(dw.cssStylePalette.getSelectedTarget(), "codeRed");
```

dreamweaver.cssStylePalette.getStyles()

Availability	3.0
Description	Gets a list of all the class styles in the active document.
Arguments	None.
Returns	An array of strings representing the names of all class styles in the document.
Enabler	None.
Example	Assuming the CSS Style palette setup shown in this example, a call to dw.cssStylePalette.getStyles() would return an array containing the following strings:

- "bigCode"
- "red"
- "sectionHead"
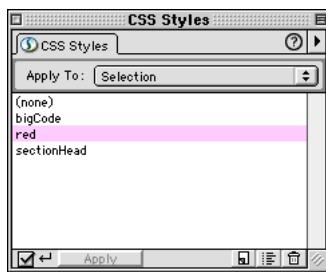
dreamweaver.cssStylePalette.newStyle()

Availability	3.0
Description	Opens the New Style dialog box.
Arguments	None.
Returns	Nothing.
Enabler	None.

External application functions

External application functions handle operations related to the browsers and external editors defined in the Preview in Browser and External Editors preferences. These functions let you get information about these external applications and open files with them.

dreamweaver.browseDocument()

Availability 2.0, enhanced in 3.0

Description Opens the specified URL in the specified browser.

Arguments *fileName, {browserName}*

- The first argument is the name of the file to be opened, expressed as an absolute URL.

- The second argument, added in Dreamweaver 3, is the name of a browser as defined in the Preview in Browser preferences. If omitted, this argument defaults to the user's primary browser.

Returns Nothing.

Enabler None.

Example The following function uses dreamweaver.browseDocument() to open the Hotwired home page in a browser:

```
function goToHotwired(){
  dreamweaver.browseDocument('http://www.hotwired.com/');
}
```

In Dreamweaver 3, you can expand this operation to open the document in Internet Explorer using the following code:

```
function goToHotwired(){
  var prevBrowsers = dw.getBrowserList();
  var theBrowser = "";
  for (var i=1; i < prevBrowsers.length; i+2){
    if (prevBrowsers[i].indexOf('Iexplore.exe') != -1){
      theBrowser = prevBrowsers[i];
      break;
    }
  }
  dw.browseDocument('http://www.hotwired.com/',theBrowser);
}
```

For more information on dw.getBrowserList(), see "dreamweaver.getBrowserList()" on page 45.

dreamweaver.getBrowserList()

Availability	3.0
Description	Gets a list of all the browsers in the Preview in Browser submenu.
Arguments	None.
Returns	An array containing a pair of strings for each browser in the list. The first string in each pair is the name of the browser, and the second string is its location on the user's machine, expressed as a file:// URL. If no browsers appear in the submenu, the function returns nothing.
Enabler	None.

dreamweaver.getExtensionEditorList()

Availability	3.0
Description	Gets a list of editors for the specified file from the External Editors preferences.
Arguments	*fileURL*
	The argument can be a complete file:// URL, a file name, or a file extension (including the period).
Returns	An array containing a pair of strings for each editor in the list. The first string in each pair is the name of the editor, and the second string is its location on the user's machine, expressed as a file:// URL. If no editors appear in the preferences, the function returns an array of one empty string.
Enabler	None.
Example	A call to dw.getExtensionEditorList(".gif") might return an array containing the following strings:

- "Fireworks 3"
- "file:///C|/Program Files/Macromedia/Fireworks 3/Fireworks 3.exe"

dreamweaver.getPrimaryBrowser()

Availability	3.0
Description	Gets the path to the primary browser.
Arguments	None.
Returns	A string containing the path on the user's hard drive to the primary browser, expressed as a file:// URL, or nothing if no primary browser is defined.
Enabler	None.

dreamweaver.getPrimaryExtensionEditor()

Availability	3.0
Description	Gets the primary editor for the specified file.
Arguments	*fileURL*
Returns	An array containing a pair of strings. The first string in the pair is the name of the editor, and the second string is its location on the user's machine, expressed as a file:// URL. If no primary editor is defined, the function returns an array of one empty string.
Enabler	None.

dreamweaver.getSecondaryBrowser()

Availability	3.0
Description	Gets the path to the secondary browser.
Arguments	None.
Returns	A string containing the path on the user's hard drive to the secondary browser, expressed as a file:// URL, or nothing if no secondary browser is defined.
Enabler	None.

dreamweaver.openWithApp()

Availability	3.0
Description	Opens the specified file with the specified application.
Arguments	*fileURL, appURL*

- The first argument is the path to the file to be opened, expressed as a file:// URL.

- The second argument is the path to the application that is to open the file, expressed as a file:// URL.

Returns	Nothing.
Enabler	None.

dreamweaver.openWithBrowseDialog()

Availability	3.0
Description	Opens the Select External Editor dialog box to let the user choose the application with which to open the specified file.
Arguments	*fileURL*
Returns	Nothing.
Enabler	None.

dreamweaver.openWithExternalTextEditor()

Availability	3.0
Description	Opens the current document in the external text editor specified in the External Editors preferences.
Arguments	None.
Returns	Nothing.
Enabler	None.

dreamweaver.openWithImageEditor()

Availability	3.0
Description	Opens the specified file with the specified image editor.

Note: This function invokes a special Fireworks integration mechanism that returns information to the active document if Fireworks is specified as the image editor. To prevent errors if no document is active, this function should never be called from the Site window.

Arguments   *fileURL, appURL*

- The first argument is the path to the file to be opened, expressed as a file:// URL.

- The second argument is the path to the application with which to open the file, expressed as a file:// URL.

Returns   Nothing.

Enabler   None.

File manipulation functions

File manipulation functions deal with creating, opening, and saving documents, and with exporting cascading style sheets to external files. They accomplish such tasks as browsing for files or folders, creating files based on templates, closing documents, and getting information about recently opened files.

dreamweaver.browseForFileURL()

Availability	1.0, enhanced in 2.0 and 3.0
Description	Opens the specified type of dialog box with the specified label in the title bar.
Arguments	*openSelectOrSave*, {*titleBarLabel*}, {*bShowPreviewPane*}, {*bSupressSiteRootWarnings*}

- The first argument indicates the type of dialog box: open, select, or save.

- The second argument, added in Dreamweaver 2, is the label that should appear in the title bar of the dialog box. If this argument is omitted, Dreamweaver uses the default label supplied by the operating system.

- The third argument, added in Dreamweaver 2, is a Boolean value indicating whether to display the image preview pane in the dialog box. If this argument is TRUE, the dialog box filters for image files; if omitted, it defaults to FALSE.

- The fourth argument, added in Dreamweaver 3, is a Boolean value indicating whether to suppress warnings about the selected file being outside the site root. If this argument is omitted, it defaults to FALSE.

Returns	A string containing the name of the file, expressed as a file:// URL.
Enabler	None.

dreamweaver.browseForFolderURL()

Availability	3.0
Description	Opens the Choose Folder dialog box with the specified label in the title bar.
Arguments	{*titleBarLabel*}, {*directoryToStartIn*}

- The first argument is the label that should appear in the title bar of the dialog box. If omitted, *titleBarLabel* defaults to "Choose Folder."

- The second argument is the path to the directory to start in, expressed as a file:// URL.

Returns	A string containing the name of the folder, expressed as a file:// URL.
Enabler	None.
Example	The following code returns the URL of a folder:

```
return dw.browseForFolderURL('Select a Folder', dw.getSiteRoot());
```

dreamweaver.closeDocument()

Availability	3.0
Description	Closes the specified document.
Arguments	*documentObject*
	The argument is the object at the root of a document's DOM tree (the value returned by dreamweaver.getDocumentDOM()). If *documentObject* refers to the active document, the Document window might not close until the script that calls this function finishes executing.
Returns	Nothing.
Enabler	None.

dreamweaver.createDocument()

Availability	2.0
Description	Opens a new document either in the same window or in a new window. The new document becomes the active document.
	Note : This function can be called only from menus.xml or a command or property inspector file. If a behavior action or object tries to call this function, Dreamweaver displays an error message.
Arguments	{*bOpenInSameWindow*}
	The argument is a Boolean value indicating whether to open the new document in the current window. If *bOpenInSameWindow* is FALSE or omitted, or the function is called on the Macintosh, the new document opens in a separate window.
Returns	The document object for the newly created document. This is the same value returned by dreamweaver.getDocumentDOM().
Enabler	None.

dreamweaver.exportCSS()

Availability	3.0
Description	Opens the Export Styles as CSS File dialog box.
Arguments	None.
Returns	Nothing.
Enabler	dreamweaver.canExportCSS()

dreamweaver.exportEditableRegionsAsXML()

Availability	3.0
Description	Opens the Export Editable Regions as XML dialog box.
Arguments	None.
Returns	Nothing.
Enabler	None.

dreamweaver.getRecentFileList()

Availability	3.0
Description	Gets a list of all the files in the recent files list at the bottom of the File menu.
Arguments	None.
Returns	An array of strings representing the paths of the most recently accessed files, expressed as file:// URLs. If there are no recent files, the function returns nothing.
Enabler	None.

dreamweaver.importXMLIntoTemplate()

Availability	3.0
Description	Opens the Import XML dialog box.
Arguments	None.
Returns	Nothing.
Enabler	None.

dreamweaver.newFromTemplate()

Availability	3.0
Description	Creates a new document from the specified template. If no argument is supplied, the Select Template dialog box appears.
Arguments	{*templateURL*} The argument is the path to a template in the current site, expressed as a file:// URL.
Returns	Nothing.
Enabler	None.

dreamweaver.openDocument()

Availability 2.0

Description Opens a document for editing in a new Dreamweaver window and gives it the focus. To the user, the effect is the same as choosing File > Open and selecting a file. If the specified file is already open, the window containing the document comes to the front. The window containing the specified file becomes the currently selected document. In Dreamweaver 2, if check in/check out is enabled, the file is checked out before it is opened. In Dreamweaver 3, you must use dreamweaver.openDocumentFromSite() to get this behavior.

Note: This function cannot be called from behavior action or object files. An error will result.

Arguments *fileName*

The argument is the name of the file to be opened, expressed as a URL. If the URL is relative, it is relative to the file containing the script that called this function.

Returns The document object for the specified file. This is the same value that is returned by dreamweaver.getDocumentDOM().

Enabler None.

dreamweaver.openDocumentFromSite()

Availability 3.0

Description Opens a document for editing in a new Dreamweaver window and gives it the focus. To the user, the effect is the same as double-clicking a file in the Site window. If the specified file is already open, the window containing the document comes to the front. The window containing the specified file becomes the currently selected document.

Note: This function cannot be called from behavior action or object files. An error will result.

Arguments *fileName*

The argument is the name of the file to be opened, expressed as a URL. If the URL is relative, it is relative to the file containing the script that called this function.

Returns The document object for the specified file. This is the same value that is returned by dreamweaver.getDocumentDOM().

Enabler None.

dreamweaver.openInFrame()

Availability	3.0
Description	Opens the Open In Frame dialog box. When the user selects a document, it is opened into the active frame.
Arguments	None.
Returns	Nothing.
Enabler	dreamweaver.canOpenInFrame()

dreamweaver.releaseDocument()

Availability	2.0
Description	Explicitly releases a previously referenced document from memory.
	Documents referenced by dreamweaver.getObjectTags(), dreamweaver.getObjectRefs(), dreamweaver.getDocumentPath(), or dreamweaver.getDocumentDOM() are automatically released when the script that contains the call finishes executing. If the script opens many documents, you must use this function to explicitly release documents before finishing the script to avoid running out of memory.
	Note: This function is relevant only for documents that were referenced by a URL, that are not currently open in a frame or Document window, and that are not extension files. (Extension files are loaded into memory at startup and are not released until you quit Dreamweaver.)
Arguments	*documentObject*
	The argument is the object at the root of a document's DOM tree (the value returned by dreamweaver.getDocumentDOM()).
Returns	Nothing.
Enabler	None.

dreamweaver.revertDocument()

Availability	3.0
Description	Reverts the specified document to the previously saved version.
Arguments	*documentObject*
	The argument is the object at the root of a document's DOM tree (the value returned by dreamweaver.getDocumentDOM()).
Returns	Nothing.
Enabler	dreamweaver.canRevertDocument()

dreamweaver.saveAll()

Availability 3.0

Description Saves all open documents, opening the Save As dialog box for any documents that have not previously been saved.

Arguments None.

Returns Nothing.

Enabler dreamweaver.canSaveAll()

dreamweaver.saveDocument()

Availability 2.0

Description Saves the specified file on a local drive.

Note: In Dreamweaver 2, if the file is read-only, Dreamweaver tries to check it out. If the document is still read-only after this attempt—or if it cannot be created—an error message appears.

Arguments *documentObject, {fileURL}*

- The argument is the object at the root of a document's DOM tree (the value returned by dreamweaver.getDocumentDOM()).

- The second argument is a URL representing a location on a local drive. If the URL is relative, it is relative to the extension file. In Dreamweaver 2, this argument is required. If *fileURL* is omitted in Dreamweaver 3, the file is saved to its current location if it has been previously saved; otherwise, a Save dialog box appears.

Returns A Boolean value indicating success (TRUE) or failure (FALSE).

Enabler dreamweaver.canSaveDocument()

dreamweaver.saveDocumentAs()

Availability 3.0

Description Opens the Save As dialog box.

Arguments *documentObject*

The argument is the object at the root of a document's DOM tree (the value returned by dreamweaver.getDocumentDOM()).

Returns Nothing.

Enabler None.

dreamweaver.saveDocumentAsTemplate()

Availability	3.0
Description	Opens the Save As Template dialog box.
Arguments	*documentObject*
	The argument is the object at the root of a document's DOM tree (the value returned by dreamweaver.getDocumentDOM()).
Returns	Nothing.
Enabler	dreamweaver.canSaveDocumentAsTemplate()

dreamweaver.saveFrameset()

Availability	3.0
Description	Saves the specified frameset, or opens the Save As dialog box if the frameset has not previously been saved.
Arguments	*documentObject*
	The argument is the object at the root of a document's DOM tree (the value returned by dreamweaver.getDocumentDOM()).
Returns	Nothing.
Enabler	dreamweaver.canSaveFrameset()

dreamweaver.saveFramesetAs()

Availability	3.0
Description	Opens the Save As dialog box for the frameset file that includes the specified DOM.
Arguments	*documentObject*
	The argument is the object at the root of a document's DOM tree (the value returned by dreamweaver.getDocumentDOM()).
Returns	Nothing.
Enabler	dreamweaver.canSaveFramesetAs()

Find/replace functions

Find/replace functions handle find and replace operations. They cover both basic functionality, such as finding the next instance of a search pattern, and complex replacement operations that require no user interaction.

dreamweaver.findNext()

Availability	3.0
Description	Finds the next instance of the search string that was specified previously by dreamweaver.setUpFind(), by dreamweaver.setUpComplexFind(), or by the user in the Find dialog box, and selects the instance in the document.
Arguments	None.
Returns	Nothing.
Enabler	dreamweaver.canFindNext()

dreamweaver.replace()

Availability	3.0
Description	Verifies that the current selection matches the search criteria that was specified previously by dreamweaver.setUpFindReplace(), by dreamweaver.setUpComplexFindReplace(), or by the user in the Replace dialog box; then replaces it with the content specified in that query.
Arguments	None.
Returns	Nothing.
Enabler	None.

dreamweaver.replaceAll()

Availability	3.0
Description	Replaces each section of the current document that matches the search criteria that was specified previously by dreamweaver.setUpFindReplace(), by dreamweaver.setUpComplexFindReplace(), or by the user in the Replace dialog box, with the content specified in that query.
Arguments	None.
Returns	Nothing.
Enabler	None.

dreamweaver.setUpComplexFind()

Availability	3.0
Description	Prepares for an advanced text or tag search by loading the specified XML query.
Arguments	*xmlQueryString*

The argument is a string of XML code beginning with <dwquery> and ending with </dwquery>. (To get a string of the proper format, set up the query in the Find dialog box, click the Save Query button, open the query file in a text editor, and copy everything from the beginning of the <dwquery> tag to the end of the </dwquery> tag.)

Returns	Nothing.
Enabler	None.
Example	The first line of the following code sets up a tag search and specifies that the scope of the search should be the current document; the second line performs the search operation:

```
dw.setUpComplexFind('<dwquery><queryparams matchcase="false"
ignorewhitespace="true"
useregexp="false"/><find><qtag qname="a"><qattribute qname="href" qcompare="="
qvalue="#"></qattribute><qattribute qname="onMouseOut" qcompare="=" qvalue=""
qnegate="true"></qattribute></qtag></find></dwquery>');
dw.findNext();
```

dreamweaver.setUpComplexFindReplace()

Availability   3.0

Description   Prepares for an advanced text or tag search by loading the specified XML query.

Arguments   *xmlQueryString*

The argument is a string of XML code beginning with <dwquery> and ending with </dwquery>. (To get a string of the proper format, set up the query in the Find dialog box, click the Save Query button, open the query file in a text editor, and copy everything from the beginning of the <dwquery> tag to the end of the </dwquery> tag.)

Returns   Nothing.

Enabler   None.

Example   The first statement in the following code sets up a tag search and specifies that the scope of the search should be four files; the second statement performs the search and replace operation:

```
dw.setUpComplexFindReplace('<dwquery><queryparams matchcase="false"
ignorewhitespace="true" useregexp="false"/><find><qtag qname="a"><qattribute
qname="href"
qcompare="=" qvalue="#"></qattribute><qattribute qname="onMouseOut"
qcompare="="
qvalue="" qnegate="true"></qattribute></qtag></find><replace action="setAttribute"
param1="onMouseOut"
param2="this.style.color='#000000';this.style.fontWeight='normal'"/>
</dwquery>');
dw.replaceAll();
```

dreamweaver.setUpFind()

Availability 3.0

Description Prepares for a text or HTML source search by defining the search parameters for a subsequent dw.findNext() operation.

Arguments *searchObject*

The argument is an object for which the following properties can be defined:

- *searchString* is the text to search for.

- *searchSource* is a Boolean value indicating whether to search the HTML source.

- {*matchCase*} is a Boolean value indicating whether the search is case-sensitive. If this property is not explicitly set, it defaults to FALSE.

- {*ignoreWhitespace*} is a Boolean value indicating whether white space differences should be ignored. *ignoreWhitespace* defaults to FALSE if *useRegularExpressions* is TRUE and TRUE if *useRegularExpressions* is FALSE.

- {*useRegularExpressions*} is a Boolean value indicating whether the searchString uses regular expressions. If this property is not explicitly set, it defaults to FALSE.

Returns Nothing.

Enabler None.

Example The following code demonstrates three ways to create a *searchObject* object:

```
var searchParams;
searchParams.searchString = 'bgcolor="#FFCCFF"';
searchParams.searchSource = true;
dw.setUpFind(searchParams);

var searchParams = {searchString: 'bgcolor="#FFCCFF"', searchSource: true};
dw.setUpFind(searchParams);

dw.setUpFind({searchString: 'bgcolor="#FFCCFF"', searchSource: true});
```

dreamweaver.setUpFindReplace()

Availability	3.0
Description	Prepares for a text or HTML source search by defining the search parameters and the scope for a subsequent dw.replace() or dw.replaceAll() operation.
Arguments	*searchObject*

The argument is an object for which the following properties can be defined:

- *searchString* is the text to search for.

- *replaceString* is the text with which to replace the selection.

- *searchSource* is a Boolean value indicating whether to search the HTML source.

- {*matchCase*} is a Boolean value indicating whether the search is case-sensitive. If this property is not explicitly set, it defaults to FALSE.

- {*ignoreWhitespace*} is a Boolean value indicating whether white space differences should be ignored. *ignoreWhitespace* defaults to FALSE if *useRegularExpressions* is TRUE and TRUE if *useRegularExpressions* is FALSE.

- {*useRegularExpressions*} is a Boolean value indicating whether the searchString uses regular expressions. If this property is not explicitly set, it defaults to FALSE.

Returns	Nothing.
Enabler	None.
Example	The following code demonstrates three ways to create a *searchObject* object:

```
var searchParams;
searchParams.searchString = 'bgcolor="#FFCCFF"';
searchParams.replaceString = 'bgcolor="#CCFFCC"';
searchParams.searchSource = true;
dw.setUpFindReplace(searchParams);

var searchParams = {searchString: 'bgcolor="#FFCCFF"', replaceString: 'bgcolor="#CCFFCC"',
searchSource: true};
dw.setUpFindReplace(searchParams);

dw.setUpFindReplace({searchString: 'bgcolor="#FFCCFF"', replaceString:
'bgcolor="#CCFFCC"', searchSource: true});
```

dreamweaver.showFindDialog()

Availability	3.0
Description	Opens the Find dialog box.
Arguments	None.
Returns	Nothing.
Enabler	dreamweaver.canShowFindDialog()

dreamweaver.showFindReplaceDialog()

Availability	3.0
Description	Opens the Replace dialog box.
Arguments	None.
Returns	Nothing.
Enabler	dreamweaver.canShowFindDialog()

Frame and frameset functions

Frame and frameset functions cover only two tasks: getting the names of the frames in a frameset, and splitting a frame in two.

dom.getFrameNames()

Availability	3.0
Description	Gets a list of all the named frames in the frameset.
Arguments	None.
Returns	An array of strings, each a name of a frame in the current frameset. Any unnamed frames are skipped. If none of the frames in the frameset is named, an empty array is returned.
Enabler	None.
Example	For a document containing four frames, two of which are named, a call to dw.getDocumentDOM().getFrameNames() might return an array containing the following strings:

- "navframe"
- "main_content"

dom.splitFrame()

Availability	3.0
Description	Splits the selected frame vertically or horizontally.
Arguments	*splitDirection*
	The argument must be one of the following: "up", "down", "left", or "right".
Returns	Nothing.
Enabler	dom.canSplitFrame()

General editing functions

General editing functions handle common editing tasks that happen in the Document window. These functions insert text, HTML, and objects; apply, change, and remove font and character markup; modify tags and attributes; and more.

dom.applyCharacterMarkup()

Availability	3.0
Description	Applies the specified type of character markup to the selection. If the selection is an insertion point, applies the specified character markup to any subsequently-typed text.
Arguments	*tagName*
	The argument is the tag name associated with the character markup. It must be one of the following strings: "b", "cite", "code", "dfn", "em", "i", "kbd", "samp", "s", "strong", "tt", "u", or "var".
Returns	Nothing.
Enabler	None.

dom.applyFontMarkup()

Availability	3.0
Description	Applies the FONT tag and the specified attribute and value to the current selection.
Arguments	*attribute, value*
	• The first argument must be "face", "size", or "color".
	• The second argument is the value that should be assigned to the attribute; for example, "Arial, Helvetica, sans-serif", "5", or "#FF0000".
Returns	Nothing.
Enabler	None.

dom.deleteSelection()

Availability	3.0
Description	Deletes the selection in the document.
Arguments	None.
Returns	Nothing.
Enabler	None.

dom.editAttribute()

Availability	3.0
Description	Displays the appropriate interface for editing the specified attribute. In most cases, this is a dialog box. This function is valid only for the active document.
Arguments	*attribute*
Returns	Nothing.
Enabler	None.

dom.exitBlock()

Availability	3.0
Description	Exits the current paragraph or heading block, leaving the cursor outside of all block elements.
Arguments	None.
Returns	Nothing.
Enabler	None.

dom.getFontMarkup()

Availability	3.0
Description	Gets the value of the specified attribute of the FONT tag for the current selection.
Arguments	*attribute*
	The argument must be "face", "size", or "color".
Returns	A string containing the value of the specified attribute, or an empty string if the attribute is not set.
Enabler	None.

dom.getLinkHref()

Availability	3.0
Description	Gets the link that surrounds the current selection. This function is equivalent to looping through the parents and grandparents of the current node until a link is found, and then calling getAttribute('HREF') on the link.
Arguments	None.
Returns	A string containing the name of the linked file, expressed as a file:// URL.
Enabler	None.

dom.getLinkTarget()

Availability	3.0
Description	Gets the target of the link that surrounds the current selection. This function is equivalent to looping through the parents and grandparents of the current node until a link is found, and then calling getAttribute('TARGET') on the link.
Arguments	None.
Returns	A string containing the value of the TARGET attribute for the link, or an empty string if no target is specified.
Enabler	None.

dom.getListTag()

Availability	3.0
Description	Gets the style of the selected list.
Arguments	None.
Returns	A string containing the tag associated with the list ("ul", "ol", or "dl"), or an empty string if no tag is associated with the list. This value is always returned in lowercase letters.
Enabler	None.

dom.getTextAlignment()

Availability	3.0
Description	Gets the alignment of the block that contains the selection.
Arguments	None.
Returns	A string containing the value of the ALIGN attribute for the tag associated with the block, or an empty string if the ALIGN attribute is not set for the tag. This value is always returned in lowercase letters.
Enabler	None.

dom.getTextFormat()

Availability	3.0
Description	Gets the block format of the selected text.
Arguments	None.
Returns	A string containing the block tag associated with the text (for example, "p", "h1", "pre", and so on, or an empty string if no block tag is associated with the selection). This value is always returned in lowercase letters.
Enabler	None.

dom.hasCharacterMarkup()

Availability	3.0
Description	Checks whether the selection already has the specified character markup.
Arguments	*markupTagName*
	The argument is the name of the tag you're checking for. It must be one of the following strings: "b", "cite", "code", "dfn", "em", "i", "kbd", "samp", "s", "strong", "tt", "u", or "var".
Returns	A Boolean value indicating whether the entire selection has the specified character markup. The function returns FALSE if only part of the selection has the specified markup.
Enabler	None.

dom.indent()

Availability	3.0
Description	Indents the selection using BLOCKQUOTE tags. If the selection is inside a list, indents the selection by nesting another list of the same type inside the current list.
Arguments	None.
Returns	Nothing.
Enabler	None.

dom.insertHTML()

Availability 3.0

Description Inserts HTML content into the document at the current insertion point.

Arguments *contentToInsert, {bReplaceCurrentSelection}*

- The first argument is the content you want to insert.

- The second argument is a Boolean value indicating whether the content should replace the current selection. If *bReplaceCurrentSelection* is False, the content is inserted after the current selection.

Returns Nothing.

Enabler None.

Example The following code inserts 130 into the current document:

```
var theDOM = dw.getDocumentDOM();
theDOM.insertHTML('<b>130</b>');
```

This appears in the Document window as:

dom.insertObject()

Availability 3.0

Description Inserts the specified object, prompting the user for parameters if necessary.

Arguments *objectName*

The argument is the name of an object in the Configuration/Objects folder.

Returns Nothing.

Enabler None.

Example A call to dreamweaver.getDocumentDOM().insertObject('Button'); inserts a form button into the current document at the current insertion point or after the current selection.

Note: Although object files can be stored in separate folders, it's important that their file names be unique. If a file called Button.htm exists in the Forms folder and also in the MyObjects folder, Dreamweaver cannot distinguish between them.

dom.insertText()

Availability	3.0
Description	Inserts text content into the document at the current insertion point.
Arguments	*contentToInsert, {bReplaceCurrentSelection}*

- The first argument is the content you want to insert.

- The second argument is a Boolean value indicating whether the content should replace the current selection. If *bReplaceCurrentSelection* is False, the content is inserted after the current selection.

Returns	Nothing.
Enabler	None.
Example	The following code inserts 130 into the current document:

```
var theDOM = dw.getDocumentDOM();
theDOM.insertText('<b>130</b>');
```

This appears in the Document window as:

dom.newBlock()

Availability	3.0
Description	Creates a new block with the same tag and attributes as the block that contains the current selection, or creates a new paragraph if the cursor is outside of all blocks.
Arguments	None.
Returns	Nothing.
Enabler	None.
Example	If the current selection is inside a center-aligned paragraph, a call to dreamweaver.getDocumentDOM().newBlock() inserts <p align="center"> after the current paragraph.

dom.outdent()

Availability	3.0
Description	Outdents the selection.
Arguments	None.
Returns	Nothing.
Enabler	None.

dom.removeCharacterMarkup()

Availability	3.0
Description	Removes the specified type of character markup from the selection.
Arguments	*tagName*
	The argument is the tag name associated with the character markup. It must be one of the following strings: "b", "cite", "code", "dfn", "em", "i", "kbd", "samp", "s", "strong", "tt", "u", or "var".
Returns	Nothing.
Enabler	None.

dom.removeFontMarkup()

Availability	3.0
Description	Removes the specified attribute and its value from a FONT tag. If removing the attribute would leave only <FONT>, then the FONT tag is also removed.
Arguments	*attribute*
	The argument must be "face", "size", or "color".
Returns	Nothing.
Enabler	None.

dom.removeLink()

Availability	3.0
Description	Removes the hyperlink from the selection.
Arguments	None.
Returns	Nothing.
Enabler	None.

dom.resizeSelection()

Availability	3.0
Description	Resizes the selected object to the specified dimensions. To resize a layer or hotspot, use dom.resizeSelectionBy().
Arguments	*newWidth, newHeight*
Returns	Nothing.
Enabler	None.

dom.setAttributeWithErrorChecking()

Availability	3.0
Description	Sets the specified attribute to the specified value for the current selection, prompting the user if the value is of the wrong type, or if it is out of range. This function is valid only for the active document.
Arguments	*attribute, value*
Returns	Nothing.
Enabler	None.

dom.setLinkHref()

Availability	3.0
Description	Makes the selection a hyperlink, or changes the value of the link that surrounds the current selection.
Arguments	*linkHREF*
	The argument is the URL (document-relative path, root-relative path, or absolute URL) to link to. If the argument is omitted, the Select HTML File dialog box appears.
Returns	Nothing.
Enabler	dom.canSetLinkHref()

dom.setLinkTarget()

Availability	3.0
Description	Sets the target of the link that surrounds the current selection. This function is equivalent to looping through the parents and grandparents of the current node until a link is found, and then calling setAttribute('TARGET') on the link.
Arguments	{*linkTarget*}
	The argument is a string representing a frame or window name, or one of the reserved targets ("_self", "_parent", "_top", or "_blank"). If the argument is omitted, the Set Target dialog box appears.
Returns	Nothing.
Enabler	None.

dom.setListBoxKind()

Availability	3.0
Description	Changes the kind of the selected SELECT menu.
Arguments	*kind*
	The argument must be either "menu" or "list box".
Returns	Nothing.
Enabler	None.

dom.showListPropertiesDialog()

Availability	3.0
Description	Opens the List Properties dialog box.
Arguments	None.
Returns	Nothing.
Enabler	dom.canShowListPropertiesDialog()

dom.setListTag()

Availability	3.0
Description	Sets the style of the selected list.
Arguments	*listTag*
	The argument is the tag associated with the list. It must be "ol", "ul", "dl", or an empty string.
Returns	Nothing.
Enabler	None.

dom.setTextAlignment()

Availability	3.0
Description	Sets the ALIGN attribute of the block that contains the selection to the specified value.
Arguments	*alignValue*
	The argument must be "left", "center", or "right".
Returns	Nothing.
Enabler	None.

dom.setTextFieldKind()

Availability	3.0
Description	Sets the format of the selected text field.
Arguments	*fieldType*
	The argument must be "input", "textarea", or "password".
Returns	Nothing.
Enabler	None.

dom.showFontColorDialog()

Availability	3.0
Description	Opens the Color Picker dialog box.
Arguments	None.
Returns	Nothing.
Enabler	None.

dreamweaver.deleteSelection()

Availability	3.0
Description	Deletes the selection in the active document or the Site window; or, on the Macintosh, the edit field that has focus in a dialog box or floating palette.
Arguments	None.
Returns	Nothing.
Enabler	dreamweaver.canDeleteSelection()

dreamweaver.editFontList()

Availability	3.0
Description	Opens the Edit Font List dialog box.
Arguments	None.
Returns	Nothing.
Enabler	None.

dreamweaver.getFontList()

Availability	3.0
Description	Gets a list of all the font groups that appear in the text Property inspector and in the Style Definition dialog box.
Arguments	None.
Returns	An array of strings representing each item in the font list.
Enabler	None.
Example	For the default installation of Dreamweaver, a call to dw.getFontList() would return an array containing the following items:

- "Arial, Helvetica, sans-serif"
- "Times New Roman, Times, serif"
- "Courier New, Courier, mono"
- "Georgia, Times New Roman, Times, serif"
- "Verdana, Arial, Helvetica, sans-serif"

dreamweaver.getKeyState()

Availability   3.0

Description   Determines whether the specified modifier key is down.

Arguments   *key*

The argument must be one of the following: "Cmd", "Ctrl", "Alt", or "Shift". In Windows, "Cmd" and "Ctrl" both refer to the Control key; on the Macintosh, "Alt" refers to the Option key.

Returns   A Boolean value indicating whether the key is down.

Enabler   None.

Example   The following code checks that both the Shift and Control keys (Windows) or Shift and Command keys (Macintosh) are down before performing an operation:

```
if (dw.getKeyState("Shift") && dw.getKeyState("Cmd")){
  // execute code
}
```

Global application functions

Global application functions act on the entire application. They provide such functionality as quitting and accessing preferences.

dreamweaver.getShowDialogsOnInsert()

Availability	3.0
Description	Checks whether the Show Dialog When Inserting Objects option is turned on in the General preferences.
Arguments	None.
Returns	A Boolean value indicating whether the option is on.
Enabler	None.

dreamweaver.quitApplication()

Availability	3.0
Description	Quits Dreamweaver after the script that calls this function finishes executing.
Arguments	None.
Returns	Nothing.
Enabler	None.

dreamweaver.showAboutBox()

Availability	3.0
Description	Opens the About box.
Arguments	None.
Returns	Nothing.
Enabler	None.

dreamweaver.showPreferencesDialog()

Availability   3.0

Description   Opens the Preferences dialog box.

Arguments   {*whichTab*}

The agument must be one of the following strings: "general", "external editors", "floaters", "fonts", "highlighting", "html colors", "html format", "html rewriting", "invisible elements", "layers", "browsers", "quick tag editor", "site ftp", "status bar", "css styles", and "translation". If Dreamweaver does not recognize the argument as a valid pane name, or if the argument is omitted, the dialog box opens to the last-active pane.

Returns   Nothing.

Enabler   None.

Global document functions

Global document functions act on the entire document. They check spelling, check target browsers, set page properties, and determine correct object references for elements in the document.

dom.checkSpelling()

Availability   3.0

Description   Checks the spelling in the document, opening the Check Spelling dialog box if necessary, and notifies the user when the check is complete.

Arguments   None.

Returns   Nothing.

Enabler   None.

dom.checkTargetBrowsers()

Availability   3.0

Description   Runs a target browser check on the document. To run a target browser check on a folder or group of files, use site.checkTargetBrowsers().

Arguments   None.

Returns   Nothing.

Enabler   None.

dom.showPagePropertiesDialog()

Availability	3.0
Description	Opens the Page Properties dialog box.
Arguments	None.
Returns	Nothing.
Enabler	None.

dreamweaver.getElementRef()

Availability	2.0
Description	Gets the Netscape or IE object reference for a specific tag object in the DOM tree.
Arguments	*NSorIE, tagObject*

- The first argument must be either "NS 4.0" or "IE 4.0". The DOM and rules for nested references differ in Navigator 4.0 and Internet Explorer 4.0. This argument specifies for which browser to return a valid reference.

- The second argument is a tag object in the DOM tree.

Returns	A string representing a valid JavaScript reference to the object, such as document.layers['myLayer'].

- Dreamweaver returns correct references for Internet Explorer for A, AREA, APPLET, EMBED, DIV, SPAN, INPUT, SELECT, OPTION, TEXTAREA, OBJECT, and IMG tags.

- Dreamweaver returns correct references for Navigator for A, AREA, APPLET, EMBED, LAYER, ILAYER, SELECT, OPTION, TEXTAREA, OBJECT, and IMG tags, and for absolutely positioned DIV and SPAN tags. For DIV and SPAN tags that are not absolutely positioned, Dreamweaver returns "cannot reference *<tag>*".

- Dreamweaver does not return references for unnamed objects. If an object does not contain either a NAME or an ID attribute, then Dreamweaver returns "unnamed *<tag>*". If the browser does not support a reference by name, Dreamweaver references the object by index (for example, document.myform.applets[3]).

- Dreamweaver returns references for named objects contained in unnamed forms and layers (for example, document.forms[2].myCheckbox).

Enabler	None.

History functions

History functions deal with undoing, redoing, recording, and playing steps that appear in the History palette. A step is any repeatable change to the document or to a selection in the document. Methods of the dreamweaver.historyPalette object either control or act on the selection in the History palette, not in the current document.

dom.redo()

Availability	3.0
Description	Redoes the step that was just undone in the document.
Arguments	None.
Returns	Nothing.
Enabler	dom.canRedo()

dom.undo()

Availability	3.0
Description	Undoes the previous step in the document.
Arguments	None.
Returns	Nothing.
Enabler	dom.canUndo()

dreamweaver.getRedoText()

Availability	3.0
Description	Gets the text associated with the editing operation that will be redone if the user chooses Edit > Redo or presses Ctrl+Y (Windows) or Command+Y (Macintosh).
Arguments	None.
Returns	A string continaing the text associated with the editing operation that will be redone.
Enabler	None.
Example	If the user's last action was make the selection bold, a call to dw.getRedoText() would return "Repeat Apply Bold".

dreamweaver.getUndoText()

Availability   3.0

Description   Gets the text associated with the editing operation that will be undone if the user chooses Edit > Undo or presses Ctrl+Z (Windows) or Command+Z (Macintosh).

Arguments   None.

Returns   A string containing the text associated with the editing operation that will be undone.

Enabler   None.

Example   If the user's last action was to apply a CSS style to a selected range of text, a call to dw.getUndoText() would return "Undo Apply <span>".

dreamweaver.playRecordedCommand()

Availability   3.0

Description   Plays the recorded command in the active document.

Arguments   None.

Returns   Nothing.

Enabler   dreamweaver.canPlayRecordedCommand()

dreamweaver.redo()

Availability   3.0

Description   Redoes the step that was just undone in the Document window, dialog box, floating palette, or Site window pane that has focus.

Arguments   None.

Returns   Nothing.

Enabler   dreamweaver.canRedo()

dreamweaver.startRecording()

Availability   3.0

Description   Starts recording steps in the active document. The previously-recorded command is immediately discarded.

Arguments   None.

Returns   Nothing.

Enabler   dreamweaver.isRecording() (must return FALSE)

dreamweaver.stopRecording()

Availability	3.0
Description	Stops recording without prompting the user.
Arguments	None.
Returns	Nothing.
Enabler	dreamweaver.isRecording() (must return TRUE)

dreamweaver.undo()

Availability	3.0
Description	Undoes the previous step in the Document window, dialog box, floating palette, or Site window pane that has focus.
Arguments	None.
Returns	Nothing.
Enabler	dreamweaver.canUndo()

dreamweaver.historyPalette.clearSteps()

Availability	3.0
Description	Clears all steps from the History palette, and disables the Undo and Redo menu items.
Arguments	None.
Returns	Nothing.
Enabler	None.

dreamweaver.historyPalette.copySteps()

Availability	3.0
Description	Copies the specified history steps to the clipboard. Dreamweaver warns the user of possible unintended consequences if the specified steps include an unrepeatable action.
Arguments	*arrayOfIndices*
	The argument is an array of position indices in the History palette.
Returns	Nothing.
Enabler	None.
Example	The following code copies the first four steps in the History palette:
	dw.historyPalette.copySteps([0,1,2,3]);

dreamweaver.historyPalette.getSelectedSteps()

Availability	3.0
Description	Determines which portion of the History palette is selected.
Arguments	None.
Returns	An array containing the position indices of all of the selected steps.
Enabler	None.
Example	If the the second, third, and fourth steps are selected in the History palette, as shown, a call to dw.historyPalette.getSelectedSteps() would return [1,2,3].

dreamweaver.historyPalette.getStepCount()

Availability	3.0
Description	Gets the number of steps in the History palette.
Arguments	None.
Returns	An integer representing the number of steps that are currently listed in the History palette.
Enabler	None.

dreamweaver.historyPalette.getStepsAsJavaScript()

Availability	3.0
Description	Gets the JavaScript equivalent of the selected steps.
Arguments	None.
Returns	A string containing the JavaScript that corresponds to the selected steps.
Enabler	None.
Example	If the three steps shown in this example are selected in the History palette, a call to dw.historyPalette.getStepsAsJavaScript(dw.historyPalette.getSelectedSteps()) returns "dw.getDocumentDOM().insertText('Hey diddle diddle, a cat and a fiddle, the cow jumped over the moon.');\ndw.getDocumentDOM().newBlock();\n dw.getDocumentDOM().insertHTML('<img src=\"../wdw99/50browsers/images/sun.gif\">', true);\n":
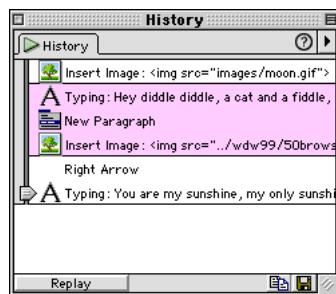
dreamweaver.historyPalette.getUndoState()

Availability	3.0
Description	Gets the current undo state.
Arguments	None.
Returns	The position of the Undo marker in the History palette.
Enabler	None.

dreamweaver.historyPalette.replaySteps()

Availability   3.0

Description   Replays the specified history steps in the active document. Dreamweaver warns the user of possible unintended consequences if the specified steps include an unrepeatable action.

Arguments   *arrayOfIndices*

The argument is an array of position indices in the History palette.

Returns   Nothing.

Enabler   None.

Example   A call to dw.historyPalette.replaySteps([0,2,3]) plays the first, third, and fourth steps in the History palette.

dreamweaver.historyPalette.saveAsCommand()

Availability   3.0

Description   Opens the Save As Command dialog box, which lets the user save the specified steps as a command. Dreamweaver warns the user of possible unintended consequences if the steps include an unrepeatable action.

Arguments   *arrayOfIndices*

The argument is an array of position indexes in the History palette.

Returns   Nothing.

Enabler   None.

Example   The following code saves the fourth, sixth, and eighth steps in the History palette as a command:

dw.historyPalette.saveAsCommand([3,5,7]);

dreamweaver.historyPalette.setSelectedSteps()

Availability   3.0

Description   Selects the specified steps in the History palette.

Arguments   *arrayOfIndices*

The argument is an array of position indices in the History palette. If no argument is supplied, all steps are deselected.

Returns   None.

Enabler   None.

Example   The following code selects the first, second, and third steps in the History palette:

dw.historyPalette.setSelection([0,1,2]);

dreamweaver.historyPalette.setUndoState()

Availability	3.0
Description	Performs the correct number of undo or redo operations to arrive at the specified undo state.
Arguments	*undoState*
	The argument is the object returned by dw.historyPalette.getUndoState().
Returns	Nothing.
Enabler	None.

HTML style functions

HTML style functions handle applying, creating, and deleting HTML styles. Methods of the dreamweaver.htmlStylePalette object either control or act on the selection in the HTML Style palette, not in the current document.

dom.applyHTMLStyle()

Availability	3.0
Description	Applies the specified HTML style to the current selection. This function is valid only for the active document.
Arguments	*htmlStyleName*
Returns	Nothing.
Enabler	None.

dreamweaver.htmlStylePalette.deleteSelectedStyle()

Availability	3.0
Description	Removes the selected style from the HTML Style palette.
Arguments	None.
Returns	Nothing.
Enabler	dreamweaver.htmlStylePalette.canEditSelection()

dreamweaver.htmlStylePalette.duplicateSelectedStyle()

Availability	3.0
Description	Duplicates the selected style, and opens the Define HTML Style dialog box.
Arguments	None.
Returns	Nothing.
Enabler	dreamweaver.htmlStylePalette.canEditSelection()

dreamweaver.htmlStylePalette.editSelectedStyle()

Availability	3.0
Description	Opens the Define HTML Style dialog box for the selected style.
Arguments	None.
Returns	Nothing.
Enabler	dreamweaver.htmlStylePalette.canEditSelection()

dreamweaver.htmlStylePalette.getSelectedStyle()

Availability	3.0
Description	Gets the name of the selected style in the HTML Style palette.
Arguments	None.
Returns	A string containing the name of the selected style.
Enabler	None.

dreamweaver.htmlStylePalette.getStyles()

Availability	3.0
Description	Gets a list of all of the names of the defined HTML styles.
Arguments	None.
Returns	An array of strings, each representing the name of an HTML style. If no HTML styles are defined, an empty array is returned.
Enabler	None.

dreamweaver.htmlStylePalette.newStyle()

Availability	3.0
Description	Opens the Define HTML Style dialog box for a new, untitled style.
Arguments	None.
Returns	Nothing.
Enabler	None.

dreamweaver.htmlStylePalette.setSelectedStyle()

Availability	3.0
Description	Selects the specified style in the HTML Style palette.
Arguments	*htmlStyleName*
Returns	Nothing.
Enabler	None.

Keyboard functions

Keyboard functions mimic document navigation tasks that are accomplished by pressing the arrow, Backspace, Delete, Page Up, and Page Down keys. In addition to such general arrow and key methods as arrowLeft() and backspaceKey(), Dreamweaver also provides methods for moving to the next or previous word or paragraph, and moving to the end of line or document or start of the line or document.

dom.arrowDown()

Availability	3.0
Description	Moves the cursor down the specified number of times.
Arguments	{*nTimes*}, {*bShiftIsDown*}

- The first argument is the number of times the cursor should be moved down. If this argument is omitted, the default is 1.

- The second argument is a Boolean value indicating whether to extend the selection. If this argument is omitted, the default is FALSE.

Returns	Nothing.
Enabler	None.

dom.arrowLeft()

Availability	3.0
Description	Moves the cursor left the specified number of times.
Arguments	{*nTimes*}, {*bShiftIsDown*}

- The first argument is the number of times the cursor should be moved left. If this argument is omitted, the default is 1.

- The second argument is a Boolean value indicating whether to extend the selection. If this argument is omitted, the default is FALSE.

Returns	Nothing.
Enabler	None.

dom.arrowRight()

Availability	3.0
Description	Moves the cursor right the specified number of times.
Arguments	*{nTimes}*, *{bShiftIsDown}*

- The first argument is the number of times the cursor should be moved right. If this argument is omitted, the default is 1.

- The second argument is a Boolean value indicating whether to extend the selection. If this argument is omitted, the default is FALSE.

Returns	Nothing.
Enabler	None.

dom.arrowUp()

Availability	3.0
Description	Moves the cursor up the specified number of times.
Arguments	*{nTimes}*, *{bShiftIsDown}*

- The first argument is the number of times the cursor should be moved up. If this argument is omitted, the default is 1.

- The second argument is a Boolean value indicating whether to extend the selection. If this argument is omitted, the default is FALSE.

Returns	Nothing.
Enabler	None.

dom.backspaceKey()

Availability	3.0
Description	Equivalent to pressing the Backspace key the specified number of times. The exact behavior differs depending on whether there is a current selection or just an insertion point.
Arguments	*{nTimes}*

The argument is the number of times a Backspace operation should occur. If omitted, the default is 1.

Returns	Nothing.
Enabler	None.

dom.deleteKey()

Availability	3.0
Description	Equivalent to pressing the Delete key the specified number of times. The exact behavior differs depending on whether there is a current selection or just an insertion point.
Arguments	*{nTimes}*
	The argument is the number of times a Delete operation should occur. If omitted, the default is 1.
Returns	Nothing.
Enabler	None.

dom.endOfDocument()

Availability	3.0
Description	Moves the insertion point to the end of the document (that is, after the last visible content in the Document window, or after the closing HTML tag in the HTML inspector, depending on which window has focus).
Arguments	*{bShiftIsDown}*
	The argument is a Boolean value indicating whether to extend the selection. If omitted, the default is FALSE.
Returns	Nothing.
Enabler	None.

dom.endOfLine()

Availability	3.0
Description	Moves the insertion point to the end of the line.
Arguments	*{bShiftIsDown}*
	The argument is a Boolean value indicating whether to extend the selection. If omitted, the default is FALSE.
Returns	Nothing.
Enabler	None.

dom.nextParagraph()

Availability	3.0
Description	Moves the insertion point to the beginning of the next paragraph (or skips multiple paragraphs if *nTimes* is greater than 1).
Arguments	{*nTimes*}, {*bShiftIsDown*}

- The first argument is the number of paragraphs ahead the cursor should jump. If this argument is omitted, the default is 1.

- The second argument is a Boolean value indicating whether to extend the selection. If this argument is omitted, the default is FALSE.

Returns	Nothing.
Enabler	None.

dom.nextWord()

Availability	3.0
Description	Moves the insertion point to the beginning of the next word (or skips multiple words if *nTimes* is greater than 1).
Arguments	{*nTimes*}, {*bShiftIsDown*}

- The first argument is the number of words ahead the cursor should jump. If this argument is omitted, the default is 1.

- The second argument is a Boolean value indicating whether to extend the selection. If this argument is omitted, the default is FALSE.

Returns	Nothing.
Enabler	None.

dom.pageDown()

Availability	3.0
Description	Moves the insertion point down one page (equivalent to pressing Page Down).
Arguments	{*nTimes*}, {*bShiftIsDown*}

- The first argument is the number of pages down the cursor should move. If this argument is omitted, the default is 1.

- The second argument is a Boolean value indicating whether to extend the selection. If this argument is omitted, the default is FALSE.

Returns	Nothing.
Enabler	None.

dom.pageUp()

Availability   3.0

Description   Moves the insertion point up one page (equivalent to pressing Page Up).

Arguments   {*nTimes*}, {*bShiftIsDown*}

- The first argument is the number of pages up the cursor should move. If this argument is omitted, the default is 1.

- The second argument is a Boolean value indicating whether to extend the selection. If this argument is omitted, the default is FALSE.

Returns   Nothing.

Enabler   None.

dom.previousParagraph()

Availability   3.0

Description   Moves the insertion point to the beginning of the previous paragraph (or skips multiple paragraphs if *nTimes* is greater than 1).

Arguments   {*nTimes*}, {*bShiftIsDown*}

- The first argument is the number of paragraphs back the cursor should jump. If this argument is omitted, the default is 1.

- The second argument is a Boolean value indicating whether to extend the selection. If this argument is omitted, the default is FALSE.

Returns   Nothing.

Enabler   None.

dom.previousWord()

Availability   3.0

Description   Moves the insertion point to the beginning of the previous word (or skips multiple words if *nTimes* is greater than 1).

Arguments   {*nTimes*}, {*bShiftIsDown*}

- The first argument is the number of words back the cursor should jump. If this argument is omitted, the default is 1.

- The second argument is a Boolean value indicating whether to extend the selection. If this argument is omitted, the default is FALSE.

Returns   Nothing.

Enabler   None.

dom.startOfDocument()

Availability	3.0
Description	Moves the insertion point to the beginning of the document (that is, before the first visible content in the Document window, or before the opening HTML tag in the HTML inspector, depending on which window has focus).
Arguments	{*bShiftIsDown*}
	The argument is a Boolean value indicating whether to extend the selection. If omitted, the default is FALSE.
Returns	Nothing.
Enabler	None.

dom.startOfLine()

Availability	3.0
Description	Moves the insertion point to the beginning of the line.
Arguments	{*bShiftIsDown*}
	The argument is a Boolean value indicating whether to extend the selection. If omitted, the default is FALSE.
Returns	Nothing.
Enabler	None.

Layer and image map functions

Layer and image map functions handle aligning, resizing, and moving layers and image map hotspots. The function description indicates if it applies to layers or to hotspots.

dom.align()

Availability	3.0
Description	Aligns the selected layers or hotspots left, right, top or bottom.
Arguments	*alignDirection*
	The argument is the edge on which the layers or hotspots should be aligned— "left", "right", "top", or "bottom".
Returns	Nothing.
Enabler	dom.canAlign()

dom.arrange()

Availability	3.0
Description	Moves the selected hotspots in the specified direction.
Arguments	*toBackOrFront*
	The argument is the direction in which the hotspots should be moved—"front" or "back".
Returns	Nothing.
Enabler	dom.canArrange()

dom.makeSizesEqual()

Availability	3.0
Description	Makes the selected layers or hotspots equal in height, width, or both. The last layer or hotspot selected is the guide.
Arguments	*bHoriz, bVert*
	• The first argument is a Boolean value indicating whether to resize the layers or hotspots horizontally.
	• The second argument is a Boolean value indicating whether to resize the layers or hotspots vertically.
Returns	Nothing.
Enabler	None.

dom.moveSelectionBy()

Availability	3.0
Description	Moves the selected layers or hotspots by the specified number of pixels horizontally and vertically.
Arguments	*x, y*

- The first argument is the number of pixels the selection should be moved horizontally.
- The second argument is the number of pixels the selection should be moved vertically.

Returns	Nothing.
Enabler	None.

dom.resizeSelectionBy()

Availability	3.0
Description	Resizes the currently selected layer or hotspot.
Arguments	*left, top, bottom, right*

- The first argument is the new position of the left boundary of the layer or hotspot.
- The second argument is the new position of the top boundary of the layer or hotspot.
- The third argument is the new position of the bottom boundary of the layer or hotspot.
- The fourth argument is the new position of the right boundary of the layer or hotspot.

Returns	Nothing.
Enabler	None.
Example	If the selected layer has the Left, Top, Width, and Height properties shown, calling dw.getDocumentDOM().resizeSelectionBy(–10,–30,30,10) would be equivalent to resetting Left to 40, Top to 20, Width to 240, and Height to 240.
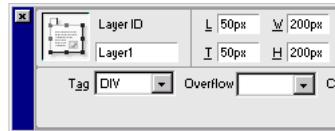
dom.setLayerTag()

Availability	3.0
Description	Specifies the HTML tag that defines the selected layer or layers.
Arguments	*tagName*
	The argument must be "layer", "ilayer", "div", or "span".
Returns	Nothing.
Enabler	None.

Library and template functions

Library and template functions manage operations related to library items and templates, such as creating, updating, and breaking links between a document and a template or library item. Methods of the dreamweaver.libraryPalette object either control or act on the selection in the Library palette, not in the current document. Likewise, methods of the dreamweaver.templatePalette object control or act on the selection in the Template palette.

dom.applyTemplate()

Availability	3.0
Description	Applies a template to the current document. If no argument is supplied, the Select Template dialog box appears. This function is valid only for the active document.
Arguments	{*templateURL*}
	The argument is the path to a template in the current site, expressed as a file:// URL.
Returns	Nothing.
Enabler	dom.canApplyTemplate()

dom.detachFromLibrary()

Availability	3.0
Description	Detaches the selected instance of a library item from its associated LBI file by removing the locking tags from around the selection. This function is equivalent to clicking Detach from Original in the Property inspector.
Arguments	None.
Returns	Nothing.
Enabler	None.

dom.detachFromTemplate()

Availability	3.0
Description	Detaches the current document from its associated template.
Arguments	None.
Returns	Nothing.
Enabler	None.

dom.getAttachedTemplate()

Availability	3.0
Description	Gets the path of the template that is associated with the document.
Arguments	None.
Returns	A string containing the path of the template, expressed as a file:// URL.
Enabler	None.

dom.getEditableRegionList()

Availability	3.0
Description	Gets a list of all the editable regions in the body of the document.
Arguments	None.
Returns	An array of element nodes.
Enabler	None.
Example	See "dom.getSelectedEditableRegion()" on page 95.

dom.getIsLibraryDocument()

Availability	3.0
Description	Determines whether the document is a library item.
Arguments	None.
Returns	A Boolean value indicating whether the document is an LBI file.
Enabler	None.

dom.getIsTemplateDocument()

Availability	3.0
Description	Determines whether the document is a template.
Arguments	None.
Returns	A Boolean value indicating whether the document is a DWT file.
Enabler	None.

dom.getSelectedEditableRegion()

Availability	3.0
Description	If the selection or insertion point is inside an editable region, gets the position of the editable region among all others in the body of the document.
Arguments	None.
Returns	An index into the array returned by dom.getEditableRegionList().
Enabler	None.
Example	The following code shows a dialog box containing the contents of the selected editable region:

```
var theDOM = dw.getDocumentDOM();
var edRegs = theDOM.getEditableRegionList();
var selReg = theDOM.getSelectedEditableRegion();
alert(edRegs[selReg].innerHTML);
```

dom.insertLibraryItem()

Availability	3.0
Description	Inserts an instance of a library item into the document.
Arguments	*libraryItemURL*
	The argument the path to an LBI file, expressed as a file:// URL.
Returns	Nothing.
Enabler	None.

dom.markSelectionAsEditable()

Availability   3.0

Description   Displays the New Editable Region dialog box. When the user clicks New Region, Dreamweaver marks the selection as editable and leaves any text as is.

Arguments   None.

Returns   Nothing.

Enabler   dom.canMarkSelectionAsEditable()

dom.newEditableRegion()

Availability   3.0

Description   Displays the New Editable Region dialog box. When the user clicks New Region, Dreamweaver inserts the name of the region, surrounded by curly braces, into the document at the cursor location.

Arguments   None.

Returns   Nothing.

Enabler   dom.canMakeNewEditableRegion()

dom.removeEditableRegion()

Availability   3.0

Description   Removes an editable region from the document. If the editable region contains any content, the content is preserved—only the editable region markers are removed.

Arguments   None.

Returns   Nothing.

Enabler   dom.canRemoveEditableRegion()

dom.updateCurrentPage()

Availability   3.0

Description   Updates the document's library items, templates, or both. This function is valid only for the active document.

Arguments   {*typeOfUpdate*}

The argument, if supplied, must be "library", "template", or "both". If omitted, the default is "both".

Returns   Nothing.

Enabler   None.

dreamweaver.updatePages()

Availability	3.0
Description	Opens the Update Pages dialog box and selects the specified options.
Arguments	{*typeOfUpdate*}
	The argument must be "library", "template", or "both". If the argument is omitted, it defaults to "both".
Returns	Nothing.
Enabler	None.

dreamweaver.libraryPalette.deleteSelectedItem()

Availability	3.0
Description	Removes the selected library item from the Library palette and deletes its associated LBI file from the Library folder at the root of the current site. Instances of the deleted item may still exist in pages throughout the site.
Arguments	None.
Returns	Nothing.
Enabler	None.

dreamweaver.libraryPalette.getSelectedItem()

Availability	3.0
Description	Gets the path of the selected library item.
Arguments	None.
Returns	A string containing the path of the library item, expressed as a file:// URL.
Enabler	None.

dreamweaver.libraryPalette.newFromDocument()

Availability	3.0
Description	Creates a new library item based on the selection in the current document.
Arguments	*bReplaceCurrent*
	The argument is a Boolean value indicating whether to replace the selection with an instance of the newly-created library item.
Returns	Nothing.
Enabler	None.

dreamweaver.libraryPalette.recreateFromDocument()

Availability	3.0
Description	Creates an LBI file for the selected instance of a library item in the current document. This function is equivalent to clicking Recreate in the Property inspector.
Arguments	None.
Returns	Nothing.
Enabler	None.

dreamweaver.libraryPalette.renameSelectedItem()

Availability	3.0
Description	Turns the name of the selected library item into an edit field, allowing the user to rename the selection.
Arguments	None.
Returns	Nothing.
Enabler	None.

dreamweaver.templatePalette.deleteSelectedTemplate()

Availability	3.0
Description	Deletes the selected template from the templates folder.
Arguments	None.
Returns	Nothing.
Enabler	None.

dreamweaver.templatePalette.getSelectedTemplate()

Availability	3.0
Description	Gets the path of the selected template.
Arguments	None.
Returns	A string containing the path of the template, expressed as a file:// URL.
Enabler	None.

dreamweaver.templatePalette.newBlankTemplate()

Availability	3.0
Description	Creates a new template.
Arguments	None.
Returns	Nothing.
Enabler	None.

dreamweaver.templatePalette.renameSelectedTemplate()

Availability	3.0
Description	Turns the name of the selected template into an edit field, allowing the user to rename the selection.
Arguments	None.
Returns	Nothing.
Enabler	None.

Menu functions

Menu functions deal with optimizing and reloading the menus in Dreamweaver. The dreamweaver.getMenuNeedsUpdating() and dreamweaver.notifyMenuUpdated() functions are designed specifically to prevent unnecesssary update routines from running on the dynamic menus that are built into Dreamweaver.

dreamweaver.getMenuNeedsUpdating()

Availability	3.0
Description	Checks whether the specified menu needs to be updated.
Arguments	*menuId*
	The argument is a string containing the value of the id attribute for the menu item (as specified in the menus.xml file).
Returns	A Boolean value indicating whether the menu needs to be updated. This function returns FALSE only if dreamweaver.notifyMenuUpdated() has been called with this *menuId*, and the return value of *menuListFunction* has not changed since that time. See "dreamweaver.notifyMenuUpdated()" on page 100 for more information.
Enabler	None.

dreamweaver.notifyMenuUpdated()

Availability   3.0

Description   Notifies Dreamweaver when the specified menu needs to be updated.

Arguments   *menuId, menuListFunction*

- The first argument is a string containing the value of the id attribute for the menu item (as specified in the menus.xml file).

- The second argument must be one of the following strings:
  "dw.cssStylePalette.getStyles()", "dw.getDocumentDOM().getFrameNames()",
  "dw.getDocumentDOM().getEditableRegionList", "dw.getBrowserList()",
  "dw.getRecentFileList()", "dw.getTranslatorList()", "dw.getFontList()",
  "dw.getDocumentList()", "dw.htmlStylePalette.getStyles()", **or** "site.getSites()".

Returns   Nothing.

Enabler   None.

dreamweaver.reloadMenus()

Availability   3.0

Description   Reloads the entire menu structure from the menus.xml file in the Configuration folder.

Arguments   None.

Returns   Nothing.

Enabler   None.

Path functions

Path functions get and manipulate the paths to various files and folders on the user's hard drive. These functions determine the path to the root of the site in which the current document resides, convert relative paths to absolute URLs, and more.

dreamweaver.getConfigurationPath()

Availability	2.0
Description	Gets the path to the Configuration folder, expressed as a file:// URL.
Arguments	None.
Returns	A string containing the path to the Configuration folder.
Enabler	None.
Example	This function is useful when referencing other extension files, which are all stored in the Configuration folder inside the Dreamweaver application folder. For example:

```
var sortCmd = dreamweaver.getConfigurationPath() + "/Commands/Sort Table.htm"
var sortDOM = dreamweaver.getDocumentDOM(sortCmd);
```

dreamweaver.getDocumentPath()

Availability	1.2
Description	Gets the path of the specified document, expressed as a file:// URL. This function is equivalent to calling dreamweaver.getDocumentDOM() and then reading the URL property of the return value.
Arguments	*sourceDoc*

The argument must be "document", "parent", "parent.frames[*number*]", or "parent.frames['*frameName*']". document specifies the document that has the focus and contains the current selection. "parent" specifies the parent frameset (if the currently selected document is in a frame), and "parent.frames[*number*]" and "parent.frames['*frameName*']" specify a document that is in a particular frame within the frameset containing the current document.

Returns	One of the following values:

- A string containing the URL of the specified document if the file has been saved.

- An empty string if the file has not been saved.

Enabler	None.

dreamweaver.getSiteRoot()

Availability	1.2
Description	Gets the local root folder (as specified in the Site Definition dialog box) for the site associated with the currently selected document, expressed as a file:// URL.
Arguments	None.
Returns	One of the following values:

- A string containing the URL of the local root folder of the site within which the file has been saved.

- An empty string if the file is not associated with a site.

Enabler	None.

dreamweaver.relativeToAbsoluteURL()

Availability	2.0
Description	Given a relative URL and a point of reference (either the path to the current document or the site root), converts the relative URL to an absolute (file://) URL.
Arguments	*relURL, docPath, siteRoot*

- The first argument is the URL to be converted.

- The second argument is the path to a document on the user's disk (for example, the current document), expressed as a file:// URL, or an empty string if *relURL* is a root-relative URL.

- The third argument is the path to the site root, expressed as a file:// URL, or an empty string if *relURL* is a document-relative URL.

Returns	An absolute URL string. The return value is generated as follows:

- If *relURL* is an absolute URL, no conversion takes place, and the return value is the same as *relURL*.

- If *relURL* is a document-relative URL, the return value is the combination of *docPath* + *relURL*.

- If *relURL* is a root-relative URL, the return value is the combination of *siteRoot* + *relURL*.

Enabler	None.

Quick Tag Editor functions

Quick Tag Editor functions navigate through the tags within and surrounding the current selection. They remove any tag in the hierarchy, wrap the selection inside a new tag, and show the Quick Tag Editor to let the user edit specific attributes for the tag.

dom.selectChild()

Availability	3.0
Description	Selects a child of the current selection. Calling this function is equivalent to selecting the next tag to the right in the tag selector at the bottom of the Document window.
Arguments	None.
Returns	Nothing.
Enabler	None.

dom.selectParent()

Availability	3.0
Description	Selects the parent of the current selection. Calling this function is equivalent to selecting the next tag to the left in the tag selector at the bottom of the Document window.
Arguments	None.
Returns	Nothing.
Enabler	None.

dom.stripTag()

Availability	3.0
Description	Removes the tag from around the current selection, leaving the contents, if any. If the selection contains more than one tag or no tags, Dreamweaver reports an error.
Arguments	None.
Returns	Nothing.
Enabler	None.

dom.wrapTag()

Availability	3.0
Description	Wraps the specified tag around the current selection. If the selection is unbalanced, Dreamweaver reports an error.
Arguments	*startTag*
	The argument is the source associated with the opening tag.
Returns	Nothing.
Enabler	None.
Example	The following code wraps a link around the current selection:

```
var theDOM = dw.getDocumentDOM();
var theSel = theDOM.getSelectedNode;
if (theSel.nodeType = Node.TEXT_NODE){
  theDOM.wrapTag('<a href="foo.html">');
}
```

dreamweaver.showQuickTagEditor()

Availability	3.0
Description	Displays the Quick Tag Editor for the current selection.
Arguments	{*nearWhat*}, {*mode*}

- The first argument, if specified, must be either "selection" or "tag selector". The default value if this argument is omitted is "selection".
- The second argument, if specified, must be "default", "wrap", "insert", or "edit". If *mode* is "default" or omitted, Dreamweaver uses heuristics to determine the mode to use for the current selection. *mode* is ignored if *nearWhat* is "tag selector".

Returns	Nothing.
Enabler	None.

Selection functions

Selection functions get and set the selection in open documents. For information on getting or setting the selection in the Site window, see "Site functions" on page 109.

dom.getSelectedNode()

Availability	3.0
Description	Gets the selected node. This function is equivalent to calling dom.getSelection() and then passing the return value to dom.offsetsToNode().
Arguments	None.
Returns	The tag, text, or comment object that completely contains the specified range of characters.
Enabler	None.

dom.getSelection()

Availability	3.0
Description	Gets the selection, expressed as byte offsets into the document's HTML source.
Arguments	*bAllowMultiple*
	The argument is a Boolean value indicating whether the function should return multiple offsets if more than one table cell, image map hotspot, or layer is selected. If this argument is omitted, it defaults to FALSE.
Returns	For simple selections, an array containing two integers. The first integer is the byte offset of the beginning of the selection. The second integer is the byte offset of the end of the selection. If the two numbers are the same, then the current selection is an insertion point.
	For complex selections (multiple table cells, multiple layers, or multiple image map hotspots), an array containing 2*n* integers, where *n* is the number of selected items. The first integer in each pair is the byte offset of the beginning of the selection (including the opening TD, DIV, SPAN, LAYER, ILAYER, or MAP tag); the second integer in each pair is the byte offset of the end of the selection (including the closing TD, DIV, SPAN, LAYER, ILAYER, or MAP tag). If multiple table rows are selected, the offsets of each cell in each row are returned. The selection never includes the TR tags.
Enabler	None.

dom.nodeToOffsets()

Availability	3.0
Description	Gets the position of a specific node in the DOM tree, expressed as byte offsets into the document's HTML source. Valid for any document on a local drive.
Arguments	*node*
	The argument must be a tag, comment, or range of text that is a node in the tree returned by dreamweaver.getDocumentDOM().
Returns	An array containing two integers. The first integer is the byte offset of the beginning of the tag, text, or comment; the second integer is the byte offset of the end of the node.
Enabler	None.
Example	The following code selects the first image object in the current document:

```
var theDOM = dreamweaver.getDocumentDOM("document");
var theImg = theDOM.images[0];
var offsets = dreamweaver.nodeToOffsets(theImg);
dreamweaver.setSelection(offsets[0], offsets[1]);
```

dom.offsetsToNode()

Availability	3.0
Description	Gets the object in the DOM tree that completely contains the range of characters between the specified begin and end points. Valid for any document on a local drive.
Arguments	*offsetBegin, offsetEnd*
	The arguments are the begin and end points, respectively, of a range of characters, expressed as byte offsets into the document's HTML source.
Returns	The tag, text, or comment object that completely contains the specified range of characters.
Enabler	None.
Example	The following code displays an alert if the selection is an image:

```
var offsets = dreamweaver.getSelection();
var theSelection = dreamweaver.offsetsToNode(offsets[0], offsets[1]);
if (theSelection.nodeType == Node.ELEMENT_NODE && theSelection.tagName == 'IMG'){
  alert('The current selection is an image.');
}
```

dom.selectAll()

Availability   3.0

Description   Performs a Select All operation.

Note: In most cases this function selects all content in the active document. In some cases (for example, when the insertion point is inside a table), however, it selects only part of the active document. To set the selection to the entire document, use dom.setSelection().

Arguments   None.

Returns   Nothing.

Enabler   None.

dom.selectTable()

Availability   3.0

Description   Selects an entire table.

Arguments   None.

Returns   Nothing.

Enabler   dom.canSelectTable()

dom.setSelectedNode()

Availability   3.0

Description   Sets the selected node. This function is equivalent to calling dom.nodeToOffsets() and then passing the return value to dom.setSelection().

Arguments   *node, {bSelectInside}, {bJumpToNode}*

- The first argument is a text, comment, or element node in the document.

- The second argument is a Boolean value indicating whether to select the innterHTML of the node. This argument is relevant only if *node* is an element node, and it defaults to FALSE if omitted.

- The third argument is a Boolean value indicating whether to scroll the Document window, if necessary, to make the selection visible. If omitted, this argument defaults to FALSE.

Returns   Nothing

Enabler   None.

dom.setSelection()

Availability   3.0

Description   Sets the selection in the document.

Arguments   *offsetBegin, offsetEnd*

The arguments are the begin and end points, respectively, for the new selection, expressed as byte offsets into the document's HTML source. If the two numbers are the same, the new selection is an insertion point. If the new selection is not a valid HTML selection, it is expanded to include the characters in the first valid HTML selection. For example, if *offsetBegin* and *offsetEnd* define the range SRC="myImage.gif" within <IMG SRC="myImage.gif">, the selection expands to include the entire IMG tag.

Returns   Nothing.

Enabler   None.

dreamweaver.selectAll()

Availability   3.0

Description   Performs a Select All operation in the active document window or the Site window; or, on the Macintosh, the edit field that has focus in a dialog box or floating palette.

Note: If the operation takes place in the active document, in most cases it selects all content in the active document. In some cases (for example, when the insertion point is inside a table), however, it selects only part of the active document. To set the selection to the entire document, use dom.setSelection().

Arguments   None.

Returns   Nothing.

Enabler   dreamweaver.canSelectAll()

Site functions

Site functions handle operations that affect files in the site files or site map. These functions create links between files; get, put, check in, and check out files; select and deselect files; create and remove files; get information about the sites that the user has defined; and more.

site.addLinkToExistingFile()

Availability	3.0
Description	Opens the Select HTML File dialog box to let the user select a file, and then creates a link from the selected document to that file.
Arguments	None.
Returns	Nothing.
Enabler	site.canAddLink()

site.addLinkToNewFile()

Availability	3.0
Description	Opens the Link to New File dialog box to let the user specify details for the new file, and then creates a link from the selected document to that file.
Arguments	None.
Returns	Nothing.
Enabler	site.canAddLink()

site.changeLinkSitewide()

Availability	3.0
Description	Opens the Change Link Sitewide dialog box.
Arguments	None.
Returns	Nothing.
Enabler	None.

site.changeLink()

Availability	3.0
Description	Opens the Select HTML File dialog box to let the user select a new file for the link.
Arguments	None.
Returns	Nothing.
Enabler	site.canChangeLink()

site.checkIn()

Availability	3.0
Description	Checks in the selected files, and handles dependent files in one of the following ways:

- If the user has selected Prompt on Put/Check In in the Site FTP preferences, the Dependent Files dialog box appears.

- If the user has previously selected the Don't Show Me Again option in the Dependent Files dialog box and then clicked Yes, dependent files are uploaded and no dialog box appears.

- If the user has previously selected the Don't Show Me Again option in the Dependent Files dialog box and then clicked No, dependent files are not uploaded and no dialog box appears.

Arguments	*siteOrURL*
	The argument must be the keyword "site", indicating that the function should act on the selection in the Site window, or the URL for a single file.
Returns	Nothing.
Enabler	site.canCheckIn()

site.checkLinks()

Availability	3.0
Description	Opens the Link Checker dialog box and checks links in the specified files.
Arguments	*scopeOfCheck*
	The argument specifies where links will be checked. It must be "document", "selection", or "site".
Returns	Nothing.
Enabler	None.

site.checkOut()

Availability	3.0
Description	Checks out the selected files, and handles dependent files in one of the following ways:

- If the user has selected Prompt on Get/Check Out in the Site FTP preferences, the Dependent Files dialog box appears.

- If the user has previously selected the Don't Show Me Again option in the Dependent Files dialog box and then clicked Yes, dependent files are downloaded and no dialog box appears.

- If the user has previously selected the Don't Show Me Again option in the Dependent Files dialog box and then clicked No, dependent files are not downloaded and no dialog box appears.

Arguments	*siteOrURL*
	The argument must be the keyword "site", indicating that the function should act on the selection in the Site window, or the URL for a single file.
Returns	Nothing.
Enabler	site.canCheckOut()

site.checkTargetBrowsers()

Availability	3.0
Description	Runs a target browser check on the selected files.
Arguments	None.
Returns	Nothing.
Enabler	None.

site.defineSites()

Availability	3.0
Description	Opens the Define Sites dialog box.
Arguments	None.
Returns	Nothing.
Enabler	None.

site.deleteSelection()

Availability	3.0
Description	Deletes the selected files.
Arguments	None.
Returns	Nothing.
Enabler	None.

site.locateInSite()

Availability	3.0
Description	Locates the specified file or files in the specified pane of the Site window, and selects the found files.
Arguments	*localOrRemote, siteOrURL*

- The first argument must be either "local" or "remote".
- The second argument must be the keyword "site", indicating that the function should act on the selection in the Site window, or the URL for a single file.

Returns	Nothing.
Enabler	site.canLocateInSite()

site.findLinkSource()

Availability	3.0
Description	Opens the file that contains the selected link or dependent file, and highlights the text of the link or the reference to the dependent in that file. This function operates only on files in the Site Map view.
Arguments	None.
Returns	Nothing.
Enabler	site.canFindLinkSource()

site.get()

Availability	3.0
Description	Gets the specified files, and handles dependent files in one of the following ways:

- If the user has selected Prompt on Get/Check Out in the Site FTP preferences, the Dependent Files dialog box appears.

- If the user has at some point selected the Don't Show Me Again option in the Dependent Files dialog box and then clicked Yes, dependent files are downloaded and no dialog box appears.

- If the user has at some point selected the Don't Show Me Again option in the Dependent Files dialog box and then clicked No, dependent files are not downloaded and no dialog box appears.

Arguments	*siteOrURL*
	The argument must be the keyword "site", indicating that the function should act on the selection in the Site window, or the URL for a single file.
Returns	Nothing.
Enabler	site.canGet()

site.getCheckOutUser()

Availability	3.0
Description	Gets the login and check out name associated with the current site.
Arguments	None.
Returns	A string containing a login and check out name, if defined, or an empty string if check in/check out is disabled.
Enabler	None.
Example	A call to site.getCheckOutUser() might return "lori (loriLaptop)". If no check out name is specified, only the login is returned (for example, "lori").

site.getCheckOutUserForFile()

Availability	3.0
Description	Gets the login and check out name of the user that has the specified file checked out.
Arguments	*fileName*
	The argument is the path to the file being inquired about, expressed as a file:// URL.
Returns	A string containing the login and check out name of the user that has the file checked out, or an empty string if the file is not checked out.
Enabler	None.
Example	A call to site.getCheckOutUserForFile("file://C:/sites/avocado8/index.html") might return "lori (loriLaptop)". If no check out name is specified, only the login is returned (for example, "lori").

site.getConnectionState()

Availability	3.0
Description	Gets the current connection state.
Arguments	None.
Returns	A Boolean value indicating whether the remote site is connected.
Enabler	site.canConnect()

site.getCurrentSite()

Availability	3.0
Description	Gets the current site.
Arguments	None.
Returns	A string containing the name of the current site.
Enabler	None.
Example	If you have several sites defined, a call to site.getCurrentSite() returns the one that is currently showing in the Current Sites List in the Site window.

site.getFocus()

Availability	3.0
Description	Determines which pane of the Site window has focus.
Arguments	None.
Returns	One of the following strings: "local", "remote", or "site map".
Enabler	None.

site.getLinkVisibility()

Availability	3.0
Description	Checks whether all of the selected links in the site map are visible (that is, not marked hidden).
Arguments	None.
Returns	A Boolean value indicating whether all of the selected links are visible.
Enabler	None.

site.getSelection()

Availability	3.0
Description	Determines which files are currently selected in the Site window.
Arguments	None.
Returns	An array of strings representing the paths of the selected files and folders, expressed as file:// URLs, or an empty array if no files or folders are selected.
Enabler	None.

site.getSites()

Availability	3.0
Description	Gets a list of the defined sites.
Arguments	None.
Returns	An array of strings representing the names of the defined sites, or an empty array if no sites are defined.
Enabler	None.

site.invertSelection()

Availability	3.0
Description	Inverts the selection in the site map.
Arguments	None.
Returns	Nothing.
Enabler	None.

site.makeEditable()

Availability	3.0
Description	Turns off the read-only flag on the selected files.
Arguments	None.
Returns	Nothing.
Enabler	site.canMakeEditable()

site.makeNewDreamweaverFile()

Availability	3.0
Description	Creates a new Dreamweaver file in the Site window (in the same directory as the first selected file or folder).
Arguments	None.
Returns	Nothing.
Enabler	site.canMakeNewFileOrFolder()

site.makeNewFolder()

Availability	3.0
Description	Creates a new folder in the Site window (in the same directory as the first selected file or folder).
Arguments	None.
Returns	Nothing.
Enabler	site.canMakeNewFileOrFolder()

site.newHomePage()

Availability	3.0
Description	Opens the New Home Page dialog box to let the user create a new home page.
Arguments	None.
Returns	Nothing.
Enabler	None.

site.newSite()

Availability	3.0
Description	Opens the Site Definition Dialog box for a new, unnamed site.
Arguments	None.
Returns	Nothing.
Enabler	None.

site.open()

Availability	3.0
Description	Opens the files that are currently selected in the Site window. If any folders are selected, they are expanded in the Site files view.
Arguments	None.
Returns	Nothing.
Enabler	site.canOpen()

site.put()

Availability   3.0

Description   Puts the selected files, and handles dependent files in one of the following ways:

- If the user has selected Prompt on Put/Check In in the Site FTP preferences, the Dependent Files dialog box appears.

- If the user has previously selected the Don't Show Me Again option in the Dependent Files dialog box and then clicked Yes, dependent files are uploaded and no dialog box appears.

- If the user has previously selected the Don't Show Me Again option in the Dependent Files dialog box and then clicked No, dependent files are not uploaded and no dialog box appears.

Arguments   *siteOrURL*

The argument must be the keyword "site", indicating that the function should act on the selection in the Site window, or the URL for a single file.

Returns   Nothing.

Enabler   site.canPut()

site.recreateCache()

Availability   3.0

Description   Recreates the cache for the current site.

Arguments   None.

Returns   Nothing.

Enabler   site.canRecreateCache()

site.refresh()

Availability   3.0

Description   Refreshes the file listing on the specified side of the Site window.

Arguments   *whichSide*

The argument must be "local", or "remote". If the site map has focus and *whichSide* is "local", the site map refreshes.

Returns   Nothing.

Enabler   site.canRefresh()

site.remoteIsValid()

Availability	3.0
Description	Determines whether the remote site is valid.
Arguments	None.
Returns	A Boolean value indicating whether a remote site has been defined, and, if the server type is Local/Network, whether the drive is mounted.
Enabler	None.

site.removeLink()

Availability	3.0
Description	Removes the selected link from the document above it in the site map.
Arguments	None.
Returns	Nothing.
Enabler	site.canRemoveLink()

site.renameSelection()

Availability	3.0
Description	Turns the name of the selected file into an edit field, allowing the user to rename the file. If more than one file is selected, this function acts on the last selected file.
Arguments	None.
Returns	Nothing.
Enabler	None.

site.saveAsImage()

Availability	3.0
Description	Opens the Save As dialog box to let the user save the site map as an image.
Arguments	*fileType*
	The argument is the type of image that should be saved. Valid values for Windows are "bmp" and "png"; valid values for Macintosh are "pict" and "jpeg". If the argument is omitted, or if the value is not valid on the current platform, the default is "bmp" in Windows and "pict" on the Macintosh.
Returns	Nothing.
Enabler	None.

site.selectAll()

Availability   3.0

Description   Selects all files in the active view (either the site map or the site files).

Arguments   None.

Returns   Nothing.

Enabler   None.

site.selectHomePage()

Availability   3.0

Description   Opens the Open File dialog box to let the user choose a new home page.

Arguments   None.

Returns   Nothing.

Enabler   None.

site.selectNewer()

Availability   3.0

Description   Selects all files that are newer on the specified side of the Site window.

Arguments   *whichSide*

   The argument must be either "local" or "remote".

Returns   Nothing.

Enabler   site.canSelectNewer()

site.setAsHomePage()

Availability   3.0

Description   Designates the file that is selected in the Site Files view as the home page for the site.

Arguments   None.

Returns   Nothing.

Enabler   None.

site.setConnectionState()

Availability	3.0
Description	Sets the connection state of the current site.
Arguments	*bConnected*
Returns	Nothing.
Enabler	None.

site.setCurrentSite()

Availability	3.0
Description	Opens the specified site in the local pane of the Site window.
Arguments	*whichSite*
	The argument is the name of a defined site (as it appears in the Current Sites List in the Site window or the Define Sites dialog box).
Returns	Nothing.
Enabler	None.
Example	If three sites are defined (for example, avocado8, dreamcentral, and testsite), a call to site.setCurrentSite("dreamcentral"); makes dreamcentral the current site.

site.setFocus()

Availability	3.0
Description	Gives focus to the specified pane of the Site window. If the specified pane is not showing, displays the pane and gives it focus.
Arguments	*whichPane*
	The argument must be one of the following strings: "local", "remote", or "site map".
Returns	Nothing.
Enabler	None.

site.setLayout()

Availability	3.0
Description	Opens the Site Map Layout pane of the Site Definition dialog box.
Arguments	None.
Returns	Nothing.
Enabler	site.canSetLayout()

site.setLinkVisibility()

Availability	3.0
Description	Shows or hides the current link.
Arguments	*bShow*
	The argument is a Boolean value indicating whether to remove the Hidden designation from the current link.
Returns	Nothing.
Enabler	None.

site.setSelection()

Availability	3.0
Description	Selects files or folders in the active pane in the Site window.
Arguments	*arrayOfURLs*
	The argument is an array of strings, each a path to a file or folder in the current site, expressed as a file:// URL.
	Note: Leave off the trailing slash (/) when specifying folder paths.
Returns	Nothing.
Enabler	None.

site.synchronize()

Availability	3.0
Description	Opens the Synchronize Files dialog box.
Arguments	None.
Returns	Nothing.
Enabler	site.canSynchronize()

site.undoCheckOut()

Availability	3.0
Description	Removes the lock files that are associated with the selected files from the local and remote sites, and replaces the local copy of the selected files with the remote copy.
Arguments	None.
Returns	Nothing.
Enabler	site.canUndoCheckOut()

site.viewAsRoot()

Availability	3.0
Description	Temporarily moves the selected file to the top position in the Site map.
Arguments	None.
Returns	Nothing.
Enabler	site.canViewAsRoot()

String manipulation functions

String manipulation functions help you get information about a string, as well as convert a string from Latin 1 encoding to platform-native encoding and back.

dreamweaver.getTokens()

Availability	1.0
Description	Accepts a string and splits it into tokens.
Arguments	*searchString, separatorCharacters*

- The first argument is the string to be separated into tokens.
- The second argument is the character or characters that signifies the end of a token. Separator characters in quoted strings are ignored. If *separatorCharacters* contains a space, all white-space characters (for example, tabs) are treated as separator characters as if you had explicitly specified them. Two or more consecutive white space characters are treated as a single separator.

Returns	An array of token strings.
Enabler	None.
Example	dreamweaver.getTokens('foo("my arg1", 34)', '(),') returns the tokens:

- foo
- "my arg 1"
- 34

dreamweaver.latin1ToNative()

Availability   2.0

Description   Converts a string in Latin1 encoding to the native encoding on the user's machine. This function is intended for displaying the user interface of an extension file in another language.

Note: This function has no effect in Windows because Windows encodings are already based on Latin1.

Arguments   *stringToConvert*

The argument is the (already translated) string to be converted from Latin1 encoding to native encoding.

Returns   The converted string.

Enabler   None.

dreamweaver.nativeToLatin1()

Availability   2.0

Description   Converts a string in native encoding to Latin1 encoding.

Note: This function has no effect in Windows because Windows encodings are already based on Latin1.

Arguments   *stringToConvert*

The argument is the string to be converted from native encoding to Latin1 encoding.

Returns   The converted string.

Enabler   None.

Table editing functions

Table functions add and remove table rows and columns, change column widths and row heights, convert measurements from pixels to percents and back, and perform other standard table editing tasks.

dom.convertWidthsToPercent()

Availability	3.0
Description	Converts all WIDTH attributes in the current table from pixels to percentages.
Arguments	None.
Returns	Nothing.
Enabler	None.

dom.convertWidthsToPixels()

Description	Converts all WIDTH attributes in the current table from percentages to pixels.
Arguments	None.
Returns	Nothing.
Enabler	None.

dom.decreaseColspan()

Availability	3.0
Description	Decreases the column span by 1.
Arguments	None.
Returns	Nothing.
Enabler	dom.canDecreaseColspan()

dom.decreaseRowspan()

Availability	3.0
Description	Decreases the row span by 1.
Arguments	None.
Returns	Nothing.
Enabler	dom.canDecreaseRowspan()

dom.deleteTableColumn()

Availability	3.0
Description	Removes the selected table column or columns.
Arguments	None.
Returns	Nothing.
Enabler	dom.canDeleteTableColumn()

dom.deleteTableRow()

Availability	3.0
Description	Removes the selected table row or rows.
Arguments	None.
Returns	Nothing.
Enabler	dom.canDeleteTableRow()

dom.doDeferredTableUpdate()

Availability	3.0
Description	If the Faster Table Editing option is selected in the General preferences, forces the table layout to reflect recent changes without moving the selection outside the table. This function has no effect if the Faster Table Editing option is not selected.
Arguments	None.
Returns	Nothing.
Enabler	None.

dom.getTableExtent()

Availability	3.0
Description	Gets the number of columns and rows in the selected table.
Arguments	None.
Returns	An array containing two whole numbers. The first array item is the number of columns, and the second array item is the number of rows. If no table was selected, nothing is returned.
Enabler	None.

dom.increaseColspan()

Availability	3.0
Description	Increases the column span by 1.
Arguments	None.
Returns	Nothing.
Enabler	dom.canIncreaseColspan()

dom.increaseRowspan()

Availability	3.0
Description	Increases the row span by 1.
Arguments	None.
Returns	Nothing.
Enabler	dom.canIncreaseRowspan()

dom.insertTableColumns()

Availability	3.0
Description	Inserts the specified number of table columns into the current table.
Arguments	*numberOfCols, bBeforeSelection*

- The first argument is the number of columns to insert.
- The second argument is a Boolean value indicating whether the columns should be inserted before the column that contains the selection.

Returns	Nothing.
Enabler	dom.canInsertTableColumns()

dom.insertTableRows()

Availability	3.0
Description	Inserts the specified number of table rows into the current table.
Arguments	*numberOfRows, bBeforeSelection*

- The first argument is the number of rows to insert.
- The second argument is a Boolean value indicating whether the rows should be inserted above the row that contains the selection.

Returns	Nothing.
Enabler	dom.canInsertTableRows()

dom.mergeTableCells()

Availability	3.0
Description	Merges the selected table cells.
Arguments	None.
Returns	Nothing.
Enabler	dom.canMergeTableCells()

dom.removeAllTableHeights()

Availability	3.0
Description	Removes all HEIGHT attribites from the selected table.
Arguments	None.
Returns	Nothing.
Enabler	None.

dom.removeAllTableWidths()

Availability	3.0
Description	Removes all WIDTH attributes from the selected table.
Arguments	None.
Returns	Nothing.
Enabler	None.

dom.setTableCellTag()

Availability	3.0
Description	Specifies the tag for the selected cell.
Arguments	*tdOrTh*
	The argument must be either "td" or "th".
Returns	Nothing.
Enabler	None.

dom.setTableColumns()

Availability	3.0
Description	Sets the number of columns in the selected table.
Arguments	*numberOfCols*
Returns	Nothing.
Enabler	None.

dom.setTableRows()

Availability	3.0
Description	Sets the number of rows in the selected table.
Arguments	*numberOfRows*
Returns	Nothing.
Enabler	None.

dom.showInsertTableRowsOrColumnsDialog()

Availability	3.0
Description	Opens the Insert Rows or Columns dialog box.
Arguments	None.
Returns	Nothing.
Enabler	dom.canInsertTableColumns() or dom.canInsertTableRows()

dom.splitTableCell()

Availability	3.0
Description	Splits the current table table cell into the specified number of rows or columns. If one or both of the arguments is omitted, the Split Cells dialog box appears.
Arguments	{*colsOrRows*}, {*numberToSplitInto*}

- The first argument, if supplied, must be either "columns" or "rows".

- The second argument, if supplied, is the number of rows or columns to split the cell into.

Returns	Nothing.
Enabler	dom.canSplitTableCell()

Timeline functions

Timeline functions act on timelines. They add, remove, and change objects in a timeline; add behaviors, frames, and keyframes to a timeline; specify whether the timeline should play and loop automatically; and more. All of the functions in this section are methods of the dreamweaver.timelineInspector object because they affect the contents of the Timeline inspector.

dreamweaver.timelineInspector.addBehavior()

Availability	3.0
Description	Opens the Behavior inspector, and automatically supplies the correct *onFrameN* event (where *N* is the frame marked by the playback head) when the user chooses an action and clicks OK.
Arguments	None.
Returns	Nothing.
Enabler	None.

dreamweaver.timelineInspector.addFrame()

Availability	3.0
Description	Adds a frame to the current timeline at the frame that contains the playback head.
Arguments	None.
Returns	Nothing.
Enabler	dreamweaver.timelineInspector.canAddFrame()

dreamweaver.timelineInspector.addKeyframe()

Availability	3.0
Description	Adds a keyframe to the selected animation bar at the frame that contains the playback head.
Arguments	None.
Returns	Nothing.
Enabler	dreamweaver.timelineInspector.canAddKeyFrame()

dreamweaver.timelineInspector.addObject()

Availability 3.0

Description Adds the currently selected object to the timeline.

Arguments None.

Returns Nothing.

Enabler None.

dreamweaver.timelineInspector.addTimeline()

Availability 3.0

Description Adds a new timeline to the current document.

Arguments None.

Returns Nothing.

Enabler None.

dreamweaver.timelineInspector.changeObject()

Availability 3.0

Description Opens the Change Object dialog box.

Arguments None.

Returns Nothing.

Enabler dreamweaver.timelineInspector.canChangeObject()

dreamweaver.timelineInspector.getAutoplay()

Availability 3.0

Description Gets the state of the Autoplay option for the current timeline.

Arguments None.

Returns A Boolean value indicating whether the Autoplay option is selected.

Enabler None.

dreamweaver.timelineInspector.getCurrentFrame()

Availability	3.0
Description	Gets the current frame of the current timeline.
Arguments	None.
Returns	A frame number.
Enabler	None.

dreamweaver.timelineInspector.getLoop()

Availability	3.0
Description	Gets the state of the Loop option for the current timeline.
Arguments	None.
Returns	A Boolean value indicating whether the Loop option is selected.
Enabler	None.

dreamweaver.timelineInspector.recordPathOfLayer()

Availability	3.0
Description	Records the path of a layer as the user drags it.
Arguments	None.
Returns	Nothing.
Enabler	None.

dreamweaver.timelineInspector.removeBehavior()

Availability	3.0
Description	Removes the selected behavior from the timeline.
Arguments	None.
Returns	Nothing.
Enabler	dreamweaver.timelineInspector.canRemoveBehavior()

dreamweaver.timelineInspector.removeFrame()

Availability	3.0
Description	Removes the selected frame from the timeline.
Arguments	None.
Returns	Nothing.
Enabler	dreamweaver.timelineInspector.canRemoveFrame()

dreamweaver.timelineInspector.removeKeyframe()

Availability	3.0
Description	Removes the selected keyframe from an animation bar.
Arguments	None.
Returns	Nothing.
Enabler	dreamweaver.timelineInspector.canRemoveKeyFrame()

dreamweaver.timelineInspector.removeObject()

Availability	3.0
Description	Removes the currently selected object from the timeline.
Arguments	None.
Returns	Nothing.
Enabler	dreamweaver.timelineInspector.canRemoveObject()

dreamweaver.timelineInspector.removeTimeline()

Availability	3.0
Description	Removes the current timeline from the document.
Arguments	None.
Returns	Nothing.
Enabler	None.

dreamweaver.timelineInspector.renameTimeline()

Availability	3.0
Description	Opens the Rename Timeline dialog box for the current timeline.
Arguments	None.
Returns	Nothing.
Enabler	None.

dreamweaver.timelineInspector.setAutoplay()

Availability	3.0
Description	Sets the Autoplay option for the current timeline.
Arguments	*bAutoplay*
	The argument is a Boolean value indicating whether to turn on the Autoplay option.
Returns	Nothing.
Enabler	None.

dreamweaver.timelineInspector.setCurrentFrame()

Availability	3.0
Description	Moves the playback head to the specified frame.
Arguments	*frameNumber*
Returns	Nothing.
Enabler	None.

dreamweaver.timelineInspector.setLoop()

Availability	3.0
Description	Sets the Loop option for the current timeline.
Arguments	*bLoop*
	The argument is a Boolean value indicating whether to turn on the Loop option.
Returns	Nothing.
Enabler	None.

Toggle functions

Toggle functions get and set various options that are either on or off.

dom.getEditNoFramesContent()

Availability	3.0
Description	Gets the current state of the Modify > Frameset > Edit NoFrames Content option.
Arguments	None.
Returns	A Boolean value indicating whether the NOFRAMES content is the active view (TRUE) or not (FALSE).
Enabler	None.

dom.getPreventLayerOverlaps()

Availability	3.0
Description	Gets the current state of the Prevent Layer Overlaps option.
Arguments	None.
Returns	A Boolean value indicating whether the option is on (TRUE) or off (FALSE).
Enabler	None.

dom.getShowFrameBorders()

Availability	3.0
Description	Gets the current state of the View > Frame Borders option.
Arguments	None.
Returns	A Boolean value indicating whether frame borders are visible (TRUE) or not (FALSE).
Enabler	None.

dom.getShowGrid()

Availability	3.0
Description	Gets the current state of the View > Grid > Show option.
Arguments	None.
Returns	A Boolean value indicating whether the grid is visible (TRUE) or not (FALSE).
Enabler	None.

dom.getShowHeadView()

Availability	3.0
Description	Gets the current state of the View > Head Content option.
Arguments	None.
Returns	A Boolean value indicating whether head content is visible (TRUE) or not (FALSE).
Enabler	None.

dom.getShowImageMaps()

Availability	3.0
Description	Gets the current state of the View > Image Maps option.
Arguments	None.
Returns	A Boolean value indicating whether image maps are visible (TRUE) or not (FALSE).
Enabler	None.

dom.getShowLayerBorders()

Availability	3.0
Description	Gets the current state of the View > Layer Borders option.
Arguments	None.
Returns	A Boolean value indicating whether layer borders are visible (TRUE) or not (FALSE).
Enabler	None.

dom.getShowRulers()

Availability	3.0
Description	Gets the current state of the View > Rulers > Show option.
Arguments	None.
Returns	A Boolean value indicating whether the rulers are visible (TRUE) or not (FALSE).
Enabler	None.

dom.getShowTableBorders()

Availability   3.0

Description   Gets the current state of the View > Table Borders option.

Arguments   None.

Returns   A Boolean value indicating whether table borders are visible (TRUE) or not (FALSE).

Enabler   None.

dom.getShowTracingImage()

Availability   3.0

Description   Gets the current state of the View > Tracing Image > Show option.

Arguments   None.

Returns   A Boolean value indicating whether the option is on (TRUE) or off (FALSE).

Enabler   None.

dom.getSnapToGrid()

Availability   3.0

Description   Gets the current state of the View > Grid > Snap To option.

Arguments   None.

Returns   A Boolean value indicating whether grid snapping is on (TRUE) or off (FALSE).

Enabler   None.

dom.setEditNoFramesContent()

Availability   3.0

Description   Turns the Modify > Frameset > Edit NoFrames Content option on (TRUE) or off (FALSE).

Arguments   *bEditNoFrames*

Returns   Nothing.

Enabler   dom.canEditNoFramesContent()

dom.setPreventLayerOverlaps()

Availability	3.0
Description	Turns the Prevent Layer Overlaps option on (TRUE) or off (FALSE).
Arguments	*bPreventLayerOverlaps*
Returns	Nothing.
Enabler	None.

dom.setShowFrameBorders()

Availability	3.0
Description	Turns the View > Frame Borders option on (TRUE) or off (FALSE).
Arguments	*bShowFrameBorders*
Returns	Nothing.
Enabler	None.

dom.setShowGrid()

Availability	3.0
Description	Turns the View > Grid > Show option on (TRUE) or off (FALSE).
Arguments	*bShowGrid*
Returns	Nothing.
Enabler	None.

dom.setShowHeadView()

Availability	3.0
Description	Turns the View > Head Content option on (TRUE) or off (FALSE).
Arguments	*bShowHead*
Returns	Nothing.
Enabler	None.

dom.setShowImageMaps()

Availability	3.0
Description	Turns the View > Image Maps option on (TRUE) or off (FALSE).
Arguments	*bShowImageMaps*
Returns	Nothing.
Enabler	None.

dom.setShowLayerBorders()

Availability	3.0
Description	Turns the View > Layer Borders option on (TRUE) or off (FALSE).
Arguments	*bShowLayerBorders*
Returns	Nothing.
Enabler	None.

dom.setShowRulers()

Availability	3.0
Description	Turns the View >Rulers > Show option on (TRUE) or off (FALSE).
Arguments	*bShowRulers*
Returns	Nothing.
Enabler	None.

dom.setShowTableBorders()

Availability	3.0
Description	Turns the View > Table Borders option on (TRUE) or off (FALSE).
Arguments	*bShowTableBorders*
Returns	Nothing.
Enabler	None.

dom.setShowTracingImage()

Availability	3.0
Description	Turns the View > Tracing Image > Show option on (TRUE) or off (FALSE).
Arguments	*bShowTracingImage*
Returns	Nothing.
Enabler	None.

dom.setSnapToGrid()

Availability	3.0
Description	Turns the View > Grid > Snap To option on (TRUE) or off (FALSE).
Arguments	*bSnapToGrid*
Returns	Nothing.
Enabler	None.

dreamweaver.getHideAllFloaters()

Availability	3.0
Description	Gets the current state of the Hide Floating Palettes option.
Arguments	None.
Returns	A Boolean value indicating whether the Hide Floating Palettes (TRUE) or the Show Floating Palettes (FALSE) menu item is available.
Enabler	None.

dreamweaver.getShowInvisibleElements()

Availability	3.0
Description	Gets the current state of the View > Invisible Elements option.
Arguments	None.
Returns	A Boolean value indicating whether invisible element markers are visible (TRUE) or not (FALSE).
Enabler	None.

dreamweaver.getShowStatusBar()

Availability	3.0
Description	Gets the current state of the View > Status Bar option.
Arguments	None.
Returns	A Boolean value indicating whether the status bar is visible (TRUE) or not (FALSE).
Enabler	None.

dreamweaver.setHideAllFloaters()

Availability	3.0
Description	Turns on either the Hide Floating Palettes option (TRUE) or the Show Floating Palettes option (FALSE).
Arguments	*bShowFloatingPalettes*
Returns	Nothing.
Enabler	None.

dreamweaver.setShowInvisibleElements()

Availability	3.0
Description	Turns the View > Invisible Elements option on (TRUE) or off (FALSE).
Arguments	*bViewInvisibleElements*
Returns	Nothing.
Enabler	None.

dreamweaver.setShowStatusBar()

Availability	3.0
Description	Turns the View > Status Bar option on (TRUE) or off (FALSE).
Arguments	*bShowStatusBar*
Returns	Nothing.
Enabler	None.

site.getShowDependents()

Availability	3.0
Description	Gets the current state of the Show Dependent Files option.
Arguments	None.
Returns	A Boolean value indicating whether dependent files are visible in the site map (TRUE) or not (FALSE).
Enabler	None.

site.getShowHiddenFiles()

Availability	3.0
Description	Gets the current state of the Show Files Marked as Hidden option.
Arguments	None.
Returns	A Boolean value indicating whether hidden files are visible in the site map (TRUE) or not (FALSE).
Enabler	None.

site.getShowPageTitles()

Availability	3.0
Description	Gets the current state of the Show Page Titles option.
Arguments	None.
Returns	A Boolean value indicating whether page titles are visible in the site map (TRUE) or not (FALSE).
Enabler	None.

site.getShowToolTips()

Availability	3.0
Description	Gets the current state of the Tool Tips option.
Arguments	None.
Returns	A Boolean value indicating whether tool tips are visible in the Site window (TRUE) or not (FALSE).
Enabler	None.

site.setShowDependents()

Availability	3.0
Description	Turns the Show Dependent Files option in the site map on (TRUE) or off (FALSE).
Arguments	*bShowDependentFiles*
Returns	Nothing.
Enabler	None.

site.setShowHiddenFiles()

Availability	3.0
Description	Turns the Show Files Marked as Hidden option in the site map on (TRUE) or off (FALSE).
Arguments	*bShowHiddenFiles*
Returns	Nothing.
Enabler	None.

site.setShowPageTitles()

Availability	3.0
Description	Turns the Show Page Titles option in the site map on (TRUE) or off (FALSE).
Arguments	*bShowPageTitles*
Returns	Nothing.
Enabler	site.canShowPageTitles()

site.setShowToolTips()

Availability	3.0
Description	Turns the Tool Tips option on (TRUE) or off (FALSE).
Arguments	*bShowToolTips*
Returns	Nothing.
Enabler	None.

Translation functions

Translation functions either deal directly with translators or with the results of translation. These functions get information about or run a translator, edit content in a locked region, and specify that the translated source should be used when getting and setting selection offsets.

dom.runTranslator()

Availability	3.0
Description	Runs the specified translator on the document. This function is valid only for the active document.
Arguments	*translatorName*
	The argument is the name of a translator as it appears in the Translation preferences.
Returns	Nothing.
Enabler	None.

dreamweaver.editLockedRegions()

Availability	2.0
Description	Depending on the value of the argument, makes locked regions editable or noneditable. By default, locked regions are noneditable; if you try to edit a locked region before specifically making it editable with this function, Dreamweaver beeps and disallows the change.
	Note: Editing locked regions can have unintended consequences for library items and templates. It is not recommend that you use this function outside the context of data translators.
Arguments	*bAllowEdits*
	The argument is a Boolean value indicating that edits are allowed (TRUE) or not allowed (FALSE). Dreamweaver automatically restores locked regions to their default (noneditable) state when the script that calls this function finishes executing.
Returns	Nothing.
Enabler	None.

dreamweaver.getTranslatorList()

Availability	3.0
Description	Gets a list of the installed translators.
Arguments	None.
Returns	An array of strings, each representing the name of a translator as it appears in the Translation preferences.
Enabler	None.

dreamweaver.useTranslatedSource()

Availability	2.0
Description	Specifies that the values returned by dreamweaver.nodeToOffsets() and dreamweaver.getSelection() and used by dreamweaver.offsetsToNode() and dreamweaver.setSelection() should be offsets into the translated source (the HTML contained in the DOM after a translator is run), not the untranslated source.

Note: This function is only relevant in property inspector files.

Arguments	*bUseTranslatedSource*

The default value of the argument is FALSE. Dreamweaver automatically uses the untranslated source for subsequent calls to dw.getSelection(), dw.setSelection(), dw.nodeToOffsets(), and dw.offsetsToNode() when the script that calls dw.useTranslatedSource() finishes executing, if dw.useTranslatedSource() is not explicitly called with an argument of FALSE before then.

Returns	Nothing.
Enabler	None.

Visual layout functions

Visual layout functions handle operations that affect the layout environment. They affect the source, position, and opacity of the tracing image; get and set the ruler origin and units; turn the grid on and off and change its settings; and play or stop playing plugins.

dom.getRulerOrigin()

Availability	3.0
Description	Gets the origin of the ruler.
Arguments	None.
Returns	An array of two integers. The first array item is the x coordinate of the origin, and the second array item is the y coordinate of the origin. Both values are in pixels.
Enabler	None.

dom.getRulerUnits()

Availability	3.0
Description	Gets the current ruler units.
Arguments	None.
Returns	A string containing one of the following values:

- "in"
- "cm"
- "px"

Enabler	None.

dom.getTracingImageOpacity()

Availability	3.0
Description	Gets the opacity setting for the document's tracing image.
Arguments	None.
Returns	A value between 0 and 100, or nothing if no opacity is set.
Enabler	dom.hasTracingImage()

dom.loadTracingImage()

Availability	3.0
Description	Opens the Select Image Source dialog box. If the user selects an image and clicks OK, the Page Properties dialog box opens with the Tracing Image field filled in.
Arguments	None.
Returns	Nothing.
Enabler	None.

dom.playAllPlugins()

Availability	3.0
Description	Plays all plugin content in the document.
Arguments	None.
Returns	Nothing.
Enabler	None.

dom.playPlugin()

Availability	3.0
Description	Plays the selected plugin item.
Arguments	None.
Returns	Nothing.
Enabler	dom.canPlayPlugin()

dom.setRulerOrigin()

Availability	3.0
Description	Sets the origin of the ruler.
Arguments	*xCoordinate, yCoordinate*
	• The first argument is a value, expressed in pixels, on the horizontal axis.
	• The second argument is a value, expressed in pixels, on the vertical axis.
Returns	Nothing.
Enabler	None.

dom.setRulerUnits()

Availability	3.0
Description	Sets the current ruler units.
Arguments	*units*
	The argument must be "px", "in", or "cm".
Returns	Nothing.
Enabler	None.

dom.setTracingImagePosition()

Availability	3.0
Description	Moves the top left corner of the tracing image to the specified coordinates. If the arguments are omitted, the Page Properties dialog box appears.
Arguments	*x, y*
Returns	Nothing.
Enabler	dom.hasTracingImage()

dom.setTracingImageOpacity()

Availability	3.0
Description	Sets the opacity of the tracing image.
Arguments	*opacityPercentage*
	The argument must be a number between 0 and 100.
Returns	Nothing.
Enabler	dom.hasTracingImage()
Example	The following code sets the opacity of the tracing image to 30%:
	dw.getDocumentDOM().setTracingOpacity('30');

dom.snapTracingImageToSelection()

Availability	3.0
Description	Aligns the top left corner of the tracing image with the top left corner of the current selection.
Arguments	None.
Returns	Nothing.
Enabler	dom.hasTracingImage()

dom.stopAllPlugins()

Availability	3.0
Description	Stops all plugin content that is currently playing in the document.
Arguments	None.
Returns	Nothing.
Enabler	None.

dom.stopPlugin()

Availability	3.0
Description	Stops the selected plugin item.
Arguments	None.
Returns	A Boolean value indicating whether the selection is currently being played with a plugin.
Enabler	dom.canStopPlugin()

dreamweaver.arrangeFloatingPalettes()

Availability	3.0
Description	Moves the visible floating palettes to their default positions.
Arguments	None.
Returns	Nothing.
Enabler	None.

dreamweaver.showGridSettingsDialog()

Availability	3.0
Description	Opens the Grid Settings dialog box.
Arguments	None.
Returns	Nothing.
Enabler	None.

Window functions

Window functions deal with operations related to the Document window and the floating palettes. These functions show and hide floating palettes, determine which part of the Document window has focus, and set the active document. For operations related specifically to the Site window, see "Site functions" on page 109.

dom.getFocus()

Availability	3.0
Description	Determines the part of the document that has focus.
Arguments	None.
Returns	One of the following strings:

- "head" if the HEAD area is active

- "body" if the BODY or NOFRAMES area is active

- "frameset" if a frameset or any of its frames is selected

- "none" if the focus is not in the document (for example, if it's in the Property inspector or another floating palette)

Enabler	None.

dom.getWindowTitle()

Availability	3.0
Description	Gets the title of the window that contains the document.
Arguments	None.
Returns	A string containing the text that appears between the TITLE tags in the document, or nothing if the document is not in an open window.
Enabler	None.

dreamweaver.getActiveWindow()

Availability	3.0
Description	Gets the document in the active window.
Arguments	None.
Returns	The document object corresponding to the document in the active window; or, if the document is in a frame, the document object corresponding to the frameset.
Enabler	None.

dreamweaver.getDocumentList()

Availability	3.0
Description	Gets a list of all the open documents.
Arguments	None.
Returns	An array of document objects, each corresponding to an open Document window. If a Document window contains a frameset, the document object refers to the frameset, not the contents of the frames.
Enabler	None.

dreamweaver.getFocus()

Availability	3.0
Description	Determines the part of the application that has focus.
Arguments	*bAllowFloaters*
Returns	One of the following strings:

- "document" if the Document window is active

- "site" if the Site window is active

- *floaterName* if *bAllowFloaters* is TRUE and a floating palette has focus, where *floaterName* is "objects", "properties", "launcher", "library", "css styles", "html styles", "behaviors", "timelines", "html", "layers", "frames", "templates", or "history"

- (Macintosh) "none" if neither the Site window nor any Document windows are open

Enabler	None.

dreamweaver.getFloaterVisibility()

Availability	3.0
Description	Checks whether the specified palette or inspector is visible.
Arguments	*floaterName*

The argument is the name of a floating palette. The built-in palettes must be referenced using one the following strings: "objects", "properties", "launcher", "site files", "site map", "library", "css styles", "html styles", "behaviors", "timelines", "html", "layers", "frames", "templates", or "history". If *floaterName* does not match one of the built-in palette names, Dreamweaver searches in the Configuration/Floaters folder for a file called *floaterName*.htm.

Returns	TRUE if the floater is visible and frontmost, FALSE otherwise or if Dreamweaver cannot find a floater with the name *floaterName*.
Enabler	None.

dreamweaver.setActiveWindow()

Availability	3.0
Description	Activates the window containing the specified document.
Arguments	*documentObject, {bActivateFrame}*

- The argument is the object at the root of a document's DOM tree (the value returned by dreamweaver.getDocumentDOM()).

- The second argument, applicable only if *documentObject* is inside a frameset, is a Boolean value indicating whether to activate the frame that contains the document as well as the window that contains the frameset.

Returns	Nothing.
Enabler	None.

dreamweaver.setFloaterVisibility()

Availability	3.0
Description	Specifies whether to make a particular floating palette or inspector visible.
Arguments	*floaterName, bIsVisible*

- The argument is the name of a floating palette. The built-in palettes must be referenced using one the following strings: "objects", "properties", "launcher", "site files", "site map", "library", "css styles", "html styles", "behaviors", "timelines", "html", "layers", "frames", "templates", or "history". If *floaterName* does not match one of the built-in palette names, Dreamweaver searches in the Configuration/Floaters folder for a file called *floaterName*.htm. If Dreamweaver cannot find a floater with the name *floaterName*, this function has no effect.

- The second argument is a Boolean value indicating whether to make the floater visible.

Returns	Nothing.
Enabler	None.

dreamweaver.showProperties()

Availability	3.0
Description	Makes the Property inspector visible and gives it focus.
Arguments	None.
Returns	Nothing.
Enabler	None.

dreamweaver.toggleFloater()

Availability   3.0

Description   Shows, hides, or brings to the front the specified palette or inspector.

Note: This function is only meaningful in the menus.xml file. To show, bring forward, or hide a floater, use dw.setFloaterVisibility().

Arguments   *floaterName*

Returns   Nothing.

Enabler   None.

Deprecated functions

Deprecated functions work, but have been superceded by newer features in Dreamweaver. You should use the newer alternatives, because support for the deprecated functions may be withdrawn in future Dreamweaver versions.

dreamweaver.getBehaviorEvent()

Availability   1.2, deprecated in 2.0 because actions are now chosen before events.

Description   In a behavior action file, gets the event that triggers this action.

Arguments   None.

Returns   A string representing the event. This is the same string that is passed as an argument (*event*) to the canAcceptBehavior() function.

Example   The following instance of getBehaviorEvent() is from the initializeUI() function in the Drag Layer action file. The role of the following snippet is similar to that of canAcceptBehavior()—that is, it checks whether the selected event is appropriate for the selected action. Such a construct is more useful than canAcceptBehavior(), because it lets you offer the user some guidance about which events are suitable for calling the selected action. canAcceptBehavior() can only make the action unavailable in the Actions pop-up menu if the user chooses an inappropriate event.

```
theEvent = dreamweaver.getBehaviorEvent().toLowerCase();
CANBEAPPLIED = (theEvent != "onmousedown" && theEvent != "onmousemove");
if (CANBEAPPLIED) {
  [display the Drag Layer UI]
} else{
  [display a helpful message that tells the user which events are appropriate for
   the Drag Layer action.]
}
```

dreamweaver.getObjectRefs()

Availability 1.0

Description Scans the specified documents for instances of the specified tags (or, if no tags are specified, for all tags in the document) and formulates browser-specific references to them. This function is equivalent to calling getElementsByTagName() and then calling dreamweaver.getElementRef() for each tag in the nodelist.

Arguments *NSorIE, sourceDoc, {tag1}, {tag2},... {tagN}*

- The first argument must be either "NS 4.0" or "IE 4.0". The DOM and rules for nested references differ in Navigator 4.0 and Internet Explorer 4.0. This argument specifies for which browser to return a valid reference.

- The second argument must be "document", "parent", "parent.frames[*number*]", "parent.frames['*frameName*']", or a URL. document specifies the document that has the focus and contains the current selection. parent specifies the parent frameset (if the currently selected document is in a frame), and parent.frames[*number*] and parent.frames['*frameName*'] specify a document that is in a particular frame within the frameset containing the current document. If the argument is a relative URL, it is relative to the extension file.

- The third and subsequent arguments, if supplied, are the names of tags (for example, "IMG", "FORM", "HR").

Returns An array of strings, each a valid JavaScript reference to a named instance of the requested tag type in the specified document (for example, "document.myLayer.document.myImage") for the specified browser.

- Dreamweaver returns correct references for Internet Explorer for A, AREA, APPLET, EMBED, DIV, SPAN, INPUT, SELECT, OPTION, TEXTAREA, OBJECT, and IMG tags.

- Dreamweaver returns correct references for Navigator for A, AREA, APPLET, EMBED, LAYER, ILAYER, SELECT, OPTION, TEXTAREA, OBJECT, and IMG tags, and for absolutely positioned DIV and SPAN tags. For DIV and SPAN tags that are not absolutely positioned, Dreamweaver returns "cannot reference <*tag*>".

- Dreamweaver does not return references for unnamed objects. If an object does not contain either a NAME or an ID attribute, then Dreamweaver returns "unnamed <*tag*>". If the browser does not support a reference by name, Dreamweaver references the object by index (for example, document.myform.applets[3]).

- Dreamweaver does return references for named objects contained in unnamed forms and layers (for example, document.forms[2].myCheckbox).

When the same list of arguments is passed to getObjectTags(), the two functions return arrays of the same length and with parallel content.

Example   dreamweaver.getObjectRefs("NS 4.0", "document", "IMG"), depending on the contents of the active document, might return an array with the following items:

- "document.bullet"
- "document.layers['headerLayer'].document.header"
- "document.photoLayer.document.headshot"

dreamweaver.getObjectTags()

Availability   1.0

Description   Scans the specified document for instances of the specified tags or, if no tags are specified, for all tags in the document. This function is equivalent to calling getElementsByTagName() and then getting outerHTML for each element in the nodelist.

Arguments   *sourceDoc, {tag1}, {tag2},... {tagN}*

- The first argument must be "document", "parent", "parent.frames[*number*]", "parent.frames['*frameName*']", or a URL. document specifies the document that has the focus and contains the current selection. parent specifies the parent frameset (if the currently selected document is in a frame), and parent.frames[*number*] and parent.frames['*frameName*'] specify a document that is in a particular frame within the frameset containing the current document. If the argument is a relative URL, it is relative to the extension file.

- The second and subsequent arguments, if supplied, are the names of tags (for example, "IMG", "FORM", "HR").

Returns   An array of strings, each the HTML source for an instance of the requested tag type in the specified document.

- If one of the *tag* arguments is LAYER, the function returns all LAYER and ILAYER tags and all absolutely positioned DIV and SPAN tags.

- If one of the *tag* arguments is INPUT, the function returns all form elements. To get a particular type of form element, specify INPUT/*TYPE*, where *TYPE* is button, text, radio, checkbox, password, textarea, select, hidden, reset, or submit.

When the same list of arguments is passed to getObjectRefs(), the two functions return arrays of the same length.

Example   dreamweaver.getObjectTags("document", "IMG"), depending on the contents of the active document, might return an array with the following items:

- '<IMG SRC="/images/dot.gif" WIDTH="10" HEIGHT="10" NAME="bullet">'

- '<IMG SRC="header.gif" WIDTH="400" HEIGHT="32" NAME="header">'

- '<IMG SRC="971208_nj.jpg" WIDTH="119" HEIGHT="119" NAME="headshot">'

dreamweaver.getSelection()

Availability   2.0, deprecated in 3.0 in favor of dom.getSelection().

Description   Gets the selection in the current document, expressed as byte offsets into the document's HTML source.

Arguments   None.

Returns   An array containing two integers. The first integer is the byte offset of the beginning of the selection; the second integer is the byte offset of the end of the selection. If the two numbers are the same, then the current selection is an insertion point.

dreamweaver.nodeToOffsets()

Availability   2.0, deprecated in 3.0 in favor of dom.nodeToOffsets().

Description   Gets the position of a specific node in the DOM tree, expressed as byte offsets into the document's HTML source.

Arguments   *node*

The argument must be a tag, comment, or range of text that is a node in the tree returned by dreamweaver.getDocumentDOM().

Returns   An array containing two integers. The first integer is the byte offset of the beginning of the tag, text, comment; the second integer is the byte offset of the end of the node.

Example   The following code selects the first image object in the current document:

```
var theDOM = dreamweaver.getDocumentDOM("document");
var theImg = theDOM.images[0];
var offsets = dreamweaver.nodeToOffsets(theImg);
dreamweaver.setSelection(offsets[0], offsets[1]);
```

dreamweaver.offsetsToNode()

Availability 2.0, deprecated in 3.0 in favor of dom.offsetsToNode().

Description Gets the object in the DOM tree that completely contains the range of characters between the specified begin and end points.

Arguments *offsetBegin, offsetEnd*

The arguments are the begin and end points, respectively, of a range of characters, expressed as byte offsets into the document's HTML source.

Returns The tag, text, or comment object that completely contains the specified range of characters.

Example The following code displays an alert if the selection is an image:

```
var offsets = dreamweaver.getSelection();
var theSelection = dreamweaver.offsetsToNode(offsets[0], offsets[1]);
if (theSelection.nodeType == Node.ELEMENT_NODE && theSelection.tagName == 'IMG'){
  alert('The current selection is an image.');
}
```

dreamweaver.popupCommand()

Availability 2.0, deprecated in 3.0 in favor of dreamweaver.runCommand().

Description Executes the specified command. To the user, the effect is the same as choosing the command from a menu; if a dialog box is associated with the command, it appears. This function provides the ability to call a command from another extension file. It blocks other edits until the user dismisses the dialog box.

Note: This function can be called only within objectTag() or in any script in a command or property inspector file.

Arguments *commandFile*

The argument is the name of a command file within the Configuration/Commands folder (for example, "Format Table.htm").

Returns Nothing.

dreamweaver.setSelection()

Availability   2.0, deprecated in 3.0 in favor of dom.setSelection().

Description   Sets the selection in the current document. This function can move the selection only within the current document; it cannot change the focus to a different document.

Arguments   *offsetBegin, offsetEnd*

The arguments are the begin and end points, respectively, for the new selection, expressed as byte offsets into the document's HTML source. If the two numbers are the same, the new selection is an insertion point. If the new selection is not a valid HTML selection, it is expanded to include the characters in the first valid HTML selection. For example, if *offsetBegin* and *offsetEnd* define the range SRC="myImage.gif" within <IMG SRC="myImage.gif">, the selection expands to include the entire IMG tag.

Returns   Nothing.

Enablers

Enabler functions determine whether to enable menu items based on whether Dreamweaver can perform specific operations in the current context. The function specifications describe the general circumstances under which each function returns TRUE. However, the descriptions are not intended to be comprehensive and may exclude some cases in which the function would return FALSE.

dom.canAlign()

Availability   3.0

Description   Checks whether Dreamweaver can perform an Align Left, Align Right, Align Top, or Align Bottom operation.

Arguments   None.

Returns   A Boolean value indicating whether two or more layers or hotspots are selected.

dom.canApplyTemplate()

Availability   3.0

Description   Checks whether Dreamweaver can perform an Apply To Page operation. This function is valid only for the active document.

Arguments   None.

Returns   A Boolean value indicating whether the document is not a library item or a template, and that the selection is not within the NOFRAMES tag.

dom.canArrange()

Availability   3.0

Description   Checks whether Dreamweaver can perform a Bring to Front or Move to Back operation.

Arguments   None.

Returns   A Boolean value indicating whether a hotspot is selected.

dom.canClipCopyText()

Availability   3.0

Description   Checks whether Dreamweaver can perform a Copy as Text operation.

Arguments   None.

Returns   A Boolean value indicating whether the selection is a range (that is, not an insertion point).

dom.canClipPaste()

Availability   3.0

Description   Checks whether Dreamweaver can perform a Paste operation.

Arguments   None.

Returns   A Boolean value indicating whether the clipboard contains any content that can be pasted ino Dreamweaver.

dom.canClipPasteText()

Availability   3.0

Description   Checks whether Dreamweaver can perform a Paste as Text operation.

Arguments   None.

Returns   A Boolean value indicating whether the clipboard contains any content that can be pasted ino Dreamweaver as text.

dom.canConvertLayersToTable()

Availability   3.0

Description   Checks whether Dreamweaver can perform a Convert Layers to Table operation.

Arguments   None.

Returns   A Boolean value indicating whether all content in the BODY of the document is contained within layers.

dom.canConvertTablesToLayers()

Availability   3.0

Description   Checks whether Dreamweaver can perform a Convert Tables to Layers operation.

Arguments   None.

Returns   A Boolean value indicating whether all the content in the BODY of the document is contained within tables, and the document is not based on a template.

dom.canDecreaseColspan()

Availability   3.0

Description   Checks whether Dreamweaver can perform a Decrease Colspan operation.

Arguments   None.

Returns   A Boolean value indicating whether the current cell has a COLSPAN attribute, and whether that attribute's value is greater than or equal to 2.

dom.canDecreaseRowspan()

Availability   3.0

Description   Checks whether Dreamweaver can perform a Decrease Rowspan operation.

Arguments   None.

Returns   A Boolean value indicating whether the current cell has a ROWSPAN attribute, and whether that attribute's value is greater than or equal to 2.

dom.canDeleteTableColumn()

Availability   3.0

Description   Checks whether Dreamweaver can perform a Delete Column operation.

Arguments   None.

Returns   A Boolean value indicating whether a the insertion point is inside a cell, or if a cell or column is selected.

dom.canDeleteTableRow()

Availability   3.0

Description   Checks whether Dreamweaver can perform a Delete Row operation.

Arguments   None.

Returns   A Boolean value indicating whether a the insertion point is inside a cell, or if a cell or row is selected.

dom.canEditNoFramesContent()

Availability   3.0

Description   Checks whether Dreamweaver can perform an Edit NoFrames Content operation.

Arguments   None.

Returns   A Boolean value indicating whether the current document is a frameset or within a frameset.

dom.canIncreaseColspan()

Availability   3.0

Description   Checks whether Dreamweaver can perform an Increase Colspan operation.

Arguments   None.

Returns   A Boolean value indicating whether there are any cells to the right of the current cell.

dom.canIncreaseRowspan()

Availability   3.0

Description   Checks whether Dreamweaver can perform an Increase Rowspan operation.

Arguments   None.

Returns   A Boolean value indicating whether there are any cells below the current cell.

dom.canInsertTableColumns()

Availability   3.0

Description   Checks whether Dreamweaver can perform an Insert Column(s) operation.

Arguments   None.

Returns   A Boolean value indicating whether the selection is inside a table. This function returns FALSE if the selection is an entire table.

dom.canInsertTableRows()

Availability   3.0

Description   Checks whether Dreamweaver can perform an Insert Row(s) operation.

Arguments   None.

Returns   A Boolean value indicating whether the selection is inside a table. This function returns FALSE if the selection is an entire table.

dom.canMakeNewEditableRegion()

Availability   3.0

Description   Checks whether Dreamweaver can perform a New Editable Region operation.

Arguments   None.

Returns   A Boolean value indicating whether the current document is a template (.dwt) file.

dom.canMarkSelectionAsEditable()

Availability   3.0

Description   Checks whether Dreamweaver can perform a Mark Selection as Editable operation.

Arguments   None.

Returns   A Boolean value indicating whether there is a selection, and whether the current document is a template (.dwt) file.

dom.canMergeTableCells()

Availability   3.0

Description   Checks whether Dreamweaver can perform a Merge Cells operation.

Arguments   None.

Returns   A Boolean value indicating whether the selection is a rectangular grouping of table cells.

dom.canPlayPlugin()

Availability   3.0

Description   Checks whether Dreamweaver can perform a Play operation. This function is valid only for the active document.

Arguments   None.

Returns   A Boolean value indicating whether the selection can be played with a plugin.

dom.canRedo()

Availability   3.0

Description   Checks whether Dreamweaver can perform a Redo operation.

Arguments   None.

Returns   A Boolean value indicating whether any steps remain to redo.

dom.canRemoveEditableRegion()

Availability	3.0
Description	Checks whether Dreamweaver can perform an Unmark Editable Region operation.
Arguments	None.
Returns	A Boolean value indicating whether the current document is a template.

dom.canSelectTable()

Availability	3.0
Description	Checks whether Dreamweaver can perform a Select Table operation.
Arguments	None.
Returns	A Boolean value indicating whether the insertion point or selection is within a table.

dom.canSetLinkHref()

Availability	3.0
Description	Checks whether Dreamweaver can change the link around the current selection or create one if necessary.
Arguments	None.
Returns	A Boolean value indicating whether the selection is an image, text, or an insertion point inside a link. A text selection is defined as a selection for which the text Property inspector would appear.

dom.canShowListPropertiesDialog()

Availability	3.0
Description	Determines whether Dreamweaver can show the List Properties dialog box.
Arguments	None.
Returns	A Boolean value indicating whether the selection is within an LI tag.
Enabler	None.

dom.canSplitFrame()

Availability	3.0
Description	Checks whether Dreamweaver can perform a Split Frame [Left	Right	Up	Down] operation.
Arguments	None.
Returns	A Boolean value indicating whether the selection is within a frame.

dom.canSplitTableCell()

Availability	3.0
Description	Checks whether Dreamweaver can perform a Split Cell operation.
Arguments	None.
Returns	A Boolean value indicating whether the insertion point is inside a table cell or the selection is a table cell.

dom.canStopPlugin()

Availability	3.0
Description	Checks whether Dreamweaver can perform a Stop operation.
Arguments	None.
Returns	A Boolean value indicating whether the selection is currently being played with a plugin.

dom.canUndo()

Availability	3.0
Description	Checks whether Dreamweaver can perform an Undo operation.
Arguments	None.
Returns	A Boolean value indicating whether any steps remain to undo.

dom.hasTracingImage()

Availability	3.0
Description	Checks whether the document has a tracing image.
Arguments	None.
Returns	A Boolean value indicating whether the document has a tracing image.

dreamweaver.canClipCopy()

Availability   3.0

Description   Checks whether Dreamweaver can perform a Copy operation.

Arguments   None.

Returns   A Boolean value indicating whether there is anything selected that can be copied to the clipboard.

dreamweaver.canClipCut()

Availability   3.0

Description   Checks whether Dreamweaver can perform a Cut operation.

Arguments   None.

Returns   A Boolean value indicating whether there is anything selected that can be cut to the clipboard.

dreamweaver.canClipPaste()

Availability   3.0

Description   Checks whether Dreamweaver can perform a Paste operation.

Returns   A Boolean value indicating whether the clipboard contains any content that can be pasted into the current document or the active pane in the Site window; or, on the Macintosh, an edit field in a floating palette or dialog box.

dreamweaver.canDeleteSelection()

Availability   3.0

Description   Checks whether Dreamweaver can delete the current selection. Depending on the window that has focus, the deletion may occur in the Document window or the Site window; or, on the Macintosh, in an edit field in a dialog box or floating palette.

Arguments   None.

Returns   A Boolean value indicating whether the selection is a range (that is, not an insertion point).

dreamweaver.canExportCSS()

Availability   3.0

Description   Checks whether Dreamweaver can perform an Export CSS Styles operation.

Arguments   None.

Returns   A Boolean value indicating whether the document contains any class styles defined in the HEAD.

dreamweaver.canFindNext()

Availability   3.0

Description   Checks whether Dreamweaver can perform a Find Next operation.

Arguments   None.

Returns   A Boolean value indicating whether a search pattern has already been established.

dreamweaver.canOpenInFrame()

Availability   3.0

Description   Checks whether Dreamweaver can perform an Open in Frame operation.

Arguments   None.

Returns   A Boolean value indicating whether the selection or insertion point is within a frame.

dreamweaver.canPlayRecordedCommand()

Availability   3.0

Description   Determines whether Dreamweaver can perform a Play Recorded Command operation.

Arguments   None.

Returns   A Boolean value indicating whether there is an active document and a previously-recorded command that can be played.

dreamweaver.canRedo()

Availability   3.0

Description   Checks whether Dreamweaver can perform a Redo operation in the current context.

Arguments   None.

Returns   A Boolean value indicating whether any operations can be undone.

dreamweaver.canRevertDocument()

Availability   3.0

Description   Checks whether Dreamweaver can perform a Revert (to the last-saved version) operation.

Arguments   *documentObject*

The argument is the object at the root of a document's DOM tree (the value returned by dreamweaver.getDocumentDOM()).

Returns   A Boolean value indicating whether the document is in an unsaved state and a saved version of the document exists on a local drive.

dreamweaver.canSaveAll()

Availability   3.0

Description   Checks whether Dreamweaver can perform a Save All operation.

Arguments   None.

Returns   A Boolean value indicating whether one or more unsaved documents are open.

dreamweaver.canSaveDocument()

Availability   3.0

Description   Checks whether Dreamweaver can perform a Save operation on the specified document.

Arguments   *documentObject*

The argument is the object at the root of a document's DOM tree (the value returned by dreamweaver.getDocumentDOM()).

Returns   A Boolean value indicating whether the document has any unsaved changes.

dreamweaver.canSaveDocumentAsTemplate()

Availability   3.0

Description   Checks whether Dreamweaver can perform a Save As Template operation on the specified document.

Arguments   *documentObject*

The argument is the object at the root of a document's DOM tree (the value returned by dreamweaver.getDocumentDOM()).

Returns   A Boolean value indicating whether the document can be saved as a template.

dreamweaver.canSaveFrameset()

Availability   3.0

Description   Checks whether Dreamweaver can perform a Save Frameset operation on the specified document.

Arguments   *documentObject*

The argument is the object at the root of a document's DOM tree (the value returned by dreamweaver.getDocumentDOM()).

Returns   A Boolean value indicating whether the document is a frameset with unsaved changes.

dreamweaver.canSaveFramesetAs()

Availability   3.0

Description   Checks whether Dreamweaver can perform a Save Frameset As operation on the specified document.

Arguments   *documentObject*

The argument is the object at the root of a document's DOM tree (the value returned by dreamweaver.getDocumentDOM()).

Returns   A Boolean value indicating whether the document is a frameset.

dreamweaver.canSelectAll()

Availability   3.0

Description   Checks whether Dreamweaver can perform a Select All operation.

Arguments   None.

Returns   A Boolean value indicating whether a Select All operation can be performed.

dreamweaver.canShowFindDialog()

Availability   3.0

Description   Checks whether Dreamweaver can perform a Find operation.

Arguments   None.

Returns   A Boolean value indicating whether the Site window or a Document window is open. This function returns FALSE when the selection is in the HEAD.

dreamweaver.canUndo()

Availability   3.0

Description   Checks whether Dreamweaver can perform an Undo operation in the current context.

Arguments   None.

Returns   A Boolean value indicating whether any operations can be undone.

dreamweaver.isRecording()

Availability   3.0

Description   Reports whether Dreamweaver is currently recording a command.

Arguments   None.

Returns   A Boolean value indicating whether Dreamweaver is recording a command.

dreamweaver.htmlStylePalette.canEditSelection()

Availability   3.0

Description   Checks whether Dreamweaver can edit, delete, or duplicate the selection in the HTML Style palette.

Arguments   None.

Returns   A Boolean value. This function returns FALSE if no style is selected, or if one of the "clear" styles is selected.

dreamweaver.timelineInspector.canAddFrame()

Availability   3.0

Description   Checks whether Dreamweaver can perform an Add Frame operation.

Arguments   None.

Returns   A Boolean value indicating whether the Timeline inspector has any animation bars or behaviors.

dreamweaver.timelineInspector.canAddKeyFrame()

Availability   3.0

Description   Checks whether Dreamweaver can perform an Add Keyframe operation.

Arguments   None.

Returns   A Boolean value indicating whether the selection in the Timeline inspector is part of an animation bar.

dreamweaver.timelineInspector.canChangeObject()

Availability   3.0

Description   Checks whether Dreamweaver can perform a Change Object operation.

Arguments   None.

Returns   A Boolean value indicating whether the selection in the Timeline inspector is part of an animation bar.

dreamweaver.timelineInspector.canRemoveBehavior()

Availability   3.0

Description   Checks whether Dreamweaver can perform a Remove Behavior operation.

Arguments   None.

Returns   A Boolean value indicating whether the selection in the Timeline inspector is a behavior.

dreamweaver.timelineInspector.canRemoveFrame()

Availability   3.0

Description   Checks whether Dreamweaver can perform a Remove Frame operation.

Arguments   None.

Returns   A Boolean value indicating whether the Timeline inspector has any animation bars or behaviors.

dreamweaver.timelineInspector.canRemoveKeyFrame()

Availability   3.0

Description   Checks whether Dreamweaver can perform a Remove Keyframe operation.

Arguments   None.

Returns   A Boolean value indicating whether the current frame in the Timeline inspector is a keyframe.

dreamweaver.timelineInspector.canRemoveObject()

Availability   3.0

Description   Checks whether Dreamweaver can perform a Remove Object operation.

Arguments   None.

Returns   A Boolean value indicating whether the Timeline inspector has any animation bars.

site.canAddLink()

Availability   3.0

Description   Checks whether Dreamweaver can perform an Add Link to [Existing File | New File] operation.

Arguments   None.

Returns   A Boolean value indicating that the selected document in the site map is an HTML file.

site.canChangeLink()

Availability   3.0

Description   Checks whether Dreamweaver can perform a Change Link operation.

Arguments   None.

Returns   A Boolean value indicating that an HTML or Flash file links to the selected file in the site map.

site.canCheckIn()

Availability   3.0

Description   Determines whether Dreamweaver can perform a Check In operation.

Arguments   *siteOrURL*

The argument must be the keyword "site", indicating that the function should act on the selection in the Site window, or the URL for a single file.

Returns   A Boolean value indicating whether all of the following conditions are true:

- A remote site has been defined.

- If a Document window has focus, the file has been saved in a local site; or, if the Site window has focus, one or more files or folders is selected.

- Check in/check out is turned on.

site.canCheckOut()

Availability	3.0
Description	Determines whether Dreamweaver can perform a Check Out operation on the specified file or files.
Arguments	*siteOrURL*
	The argument must be the keyword "site", indicating that the function should act on the selection in the Site window, or the URL for a single file.
Returns	A Boolean value indicating whether all of the following conditions are true:

- A remote site has been defined.
- If a Document window has focus, the file is part of a local site and is not already checked out; or, if the Site window has focus, one or more files or folders is selected and at least one of the selected files is not already checked out.
- Check in/check out is turned on.

site.canConnect()

Availability	3.0
Description	Checks whether Dreamweaver can connect to the remote site.
Arguments	None.
Returns	A Boolean value indicating whether the current remote site is an FTP site.

site.canFindLinkSource()

Availability	3.0
Description	Checks whether Dreamweaver can perform a Find Link Source operation.
Arguments	None.
Returns	A Boolean value indicating that the selected link in the site map is not the home page.

site.canGet()

Availability	3.0
Description	Determines whether Dreamweaver can perform a Get operation.
Arguments	*siteOrURL*

The argument must be the keyword "site", indicating that the function should act on the selection in the Site window, or the URL for a single file.

Returns If the argument is "site", Boolean value indicating whether one or more files or folders is selected in the Site window and a remote site has been defined. If the argument is a URL, a Boolean value indicating whether the document belongs to a site for which a remote site has been defined.

site.canLocateInSite()

Availability	3.0
Description	Determines whether Dreamweaver can perform a Locate in Local Site or Locate in Remote Site operation (depending on the argument).
Arguments	*localOrRemote, siteOrURL*

- The first argument must be either "local" or "remote".

- The second argument must be the keyword "site", indicating that the function should act on the selection in the Site window, or the URL for a single file.

Returns One of the following values:

- If the second argument is "site", a Boolean value indicating whether both panes contain site files (not the site map) and whether the selection is in the opposite pane from the argument.

- If the first argument is "local" and the second argument is a URL, a Boolean value indicating whether the document belongs to a site.

- If the first argument is "remote" and the second argument is a URL, a Boolean value indicating whether the document belongs to a sitefor which a remote site has been defined, and, if the server type is Local/Network, whether the drive is mounted.

site.canMakeEditable()

Availability	3.0
Description	Checks whether Dreamweaver can perform a Turn Off Read Only operation.
Arguments	None.

Returns A Boolean value indicating whether one or more of the selected files is locked.

site.canMakeNewFileOrFolder()

Availability 3.0

Description Checks whether Dreamweaver can perform a New File or New Folder operation in the Site window.

Arguments None.

Returns A Boolean value indicating whether any files are visible in the selected pane of the Site window.

site.canOpen()

Availability 3.0

Description Checks whether Dreamweaver can open the files or folders that are currently selected in the Site window.

Arguments None.

Returns A Boolean value indicating whether any files or folders are selected in the Site window.

site.canPut()

Availability 3.0

Description Determines whether Dreamweaver can perform a Put operation.

Arguments *siteOrURL*

The argument must be the keyword "site", indicating that the function should act on the selection in the Site window, or the URL for a single file.

Returns If the argument is "site", a Boolean value indicating whether any files or folders are selected in the Site window and a remote site has been defined. If the argument is a URL, a Boolean value indicating whether the document belongs to a site for which a remote site has been defined.

site.canRecreateCache()

Availability 3.0

Description Checks whether Dreamweaver can perform a Recreate Site Cache operation.

Arguments None.

Returns A Boolean value indicating whether the Use Cache To Speed Link Updates option is enabled for the current site.

site.canRefresh()

Availability   3.0

Description   Checks whether Dreamweaver can perform a Refresh [Local | Remote] operation.

Arguments   *localOrRemote*

The argument must be either "local" or "remote".

Returns   TRUE if *localOrRemote* is "local"; otherwise, a Boolean value indicating whether a remote site has been defined.

site.canRemoveLink()

Availability   3.0

Description   Checks whether Dreamweaver can perform a Remove Link operation.

Arguments   None.

Returns   A Boolean value indicating that an HTML or Flash file links to the selected file in the site map.

site.canSetLayout()

Availability   3.0

Description   Determines whether Dreamweaver can perform a Layout operation.

Arguments   None.

Returns   A Boolean value indicating whether the site map is visible.

site.canSelectNewer()

Availability   3.0

Description   Determines whether Dreamweaver can perform a Select Newer [Remote | Local] operation.

Arguments   *localOrRemote*

The argument must be either "local" or "remote".

Returns   A Boolean value whether the document belongs to a site for which a remote site has been defined.

site.canShowPageTitles()

Availability	3.0
Description	Determines whether Dreamweaver can perform a Show Page Titles operation.
Arguments	None.
Returns	A Boolean value indicating whether the site map is visible.

site.canSynchronize()

Availability	3.0
Description	Determines whether Dreamweaver can perform a Synchronize operation.
Arguments	None.
Returns	A Boolean value indicating whether a remote site has been defined.

site.canUndoCheckOut()

Availability	3.0
Description	Determines whether Dreamweaver can perform an Undo Checkout operation.
Arguments	*siteOrURL*
	The argument must be the keyword "site", indicating that the function should act on the selection in the Site window, or the URL for a single file.
Returns	A Boolean value indicating whether the specified file or at least one of the selected files is checked out (by any user).

site.canViewAsRoot()

Availability	3.0
Description	Determines whether Dreamweaver can perform a View as Root operation.
Arguments	None.
Returns	A Boolean value indicating whether the specified file is an HTML or Flash file.
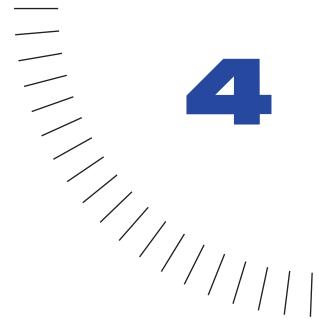
CHAPTER 4
The File I/O API

· ·

Dreamweaver 3 includes a C shared library called DWfile that gives authors of objects, commands, behaviors, data translators, floating palettes, and property inspectors the ability to read and write files on the local file system. This chapter describes the file I/O API and how to use it.

For general information on how C libraries interact with the JavaScript interpreter in Dreamweaver, see "C-Level Extensibility" on page 209.

Verifying that DWfile is installed

To access the functions in the DWfile library, the library must be present in the Configuration/JSExtensions folder and loaded by Dreamweaver. Because DWfile did not ship with Dreamweaver 2 (it was available as a separate download that had to be installed by the user), it is useful to verify that the library is available before calling any of its functions. You can do this by checking the typeof(DWfile). If DWfile does not exist, typeof(DWfile) will return undefined. For example, if you are using DWfile in the context of a command, you might check for the existence of DWfile as part of the canAcceptCommand() function:

```
// Returns TRUE if typeof(DWfile) is not undefined, FALSE otherwise.
function canAcceptCommand(){
  return (typeof(DWfile) != "undefined");
}
```

The file I/O API

All of the functions in the file I/O API are methods of the DWfile object. Optional arguments are enclosed in curly braces ({ }). Functions added for Dreamweaver 3 have an availability of 3.0. Functions with an availability of 2.0 were included in the version of DWfile that was supplied as a download for Dreamweaver 2 from the Macromedia web site. This version of DWfile may have been installed with some third-party objects.

DWfile.copy()

Availability	3.0
Description	Copies the specified file to the specified URL.
Arguments	*originalURL, copyURL*

- The first argument is the file you want to copy, expressed as a file:// URL.

- The second argument is the location to which the copied file should be saved, expressed as a file:// URL.

Returns	TRUE if the copy succeeded, FALSE otherwise.
Example	The following code copies a file called myconfig.cfg to myconfig_backup.cfg.

```
var fileURL = "file:///c|/Config/myconfig.cfg";
var newURL ="file:///c|/Config/myconfig_backup.cfg";
DWfile.copy(fileURL, newURL);
```

DWfile.createFolder()

Availability	2.0
Description	Creates a folder (directory) at the specified location.
Arguments	*folderURL*

The argument is the location of the folder you want to create, expressed as a file:// URL.

Returns	TRUE if the folder was successfully created, FALSE otherwise.
Example	The following code attempts to create a folder called tempFolder at the top level of the C drive and displays an alert box indicating whether the operation was successful.

```
var folderURL = "file:///c|/tempFolder";
if (DWfile.createFolder(folderURL)){
 alert("Created " + folderURL);
}else{
 alert("Unable to create " + folderURL);
}
```

DWfile.exists()

Availability   2.0

Description   Tests for the existence of the specified file.

Arguments   *fileURL*

The argument is the file you are looking for, expressed as a file:// URL.

Returns   TRUE if the file exists, FALSE otherwise.

Example   The following code checks for a file called mydata.txt and displays an alert box that tells the user whether the file exists or not.

```
var fileURL = "file:///c|/temp/mydata.txt";
if (DWfile.exists(fileURL)){
  alert( fileURL + " exists!");
}else{
  alert( fileURL + " does not exist.");
}
```

DWfile.getAttributes()

Availability   2.0

Description   Gets the attributes of the specified file or folder.

Arguments   *fileURL*

The argument is the file or folder for which you want to get attributes, expressed as a file:// URL.

Returns   A string representing the attributes of the specified file or folder, or NULL if the file or folder does not exist. Characters in the string represent the attributes as follows:

- R is read only.

- D is folder (directory).

- H is hidden.

- S is system file or folder.

Example   The following code gets the attributes of the file mydata.txt and displays an alert box if the file is read only.

```
var URL = "file:///c|/temp/mydata.txt";
var str = DWfile.getAttributes(URL);
if (str && (str.indexOf("R") != -1)){
  alert(URL + " is read only!");
}
```

DWfile.getModificationDate()

Availability   2.0

Description   Gets the time when the file was last modified.

Arguments   *fileURL*

The argument is the file for which you are checking the last-modified time, expressed as a file:// URL.

Returns   A string containing a hexidecimal number that represents the number of time units that have elapsed since some base time. The exact meaning of time units and base time is platform dependent; in Windows, for example, a time unit is 100ns, and the base time is January 1st, 1600.

Example   It's most useful to call the function twice and compare the return values because the value returned by this function is platform-dependent and is not a recognizable date and time. For example, the following code gets the modification dates of file1.txt and file2.txt and displays an alert box indicating which file is newer.

```
var file1 = "file:///c|/temp/file1.txt";
var file2 = "file:///c|/temp/file2.txt";
var time1 = DWfile.getModificationDate(file1);
var time2 = DWfile.getModificationDate(file2);
if (time1 == time2){
  alert("file1 and file2 were saved at the same time");
}else if (time1 < time2){
  alert("file1 older that file2");
}else{
  alert("file1 is newer than file2");
}
```

DWfile.listFolder()

Availability   2.0

Description   Gets a list of the contents of the specified folder.

Arguments   *folderURL, {constraint}*

- The first argument is the folder for which you want a contents list, expressed as a file:// URL, plus an optional wildcard file mask. Valid wildcards are * (matches 1 or more characters) and ? (matches a single character).

- The second argument, if supplied, must be either "files" (return only files) or "directories" (return only directories). If omitted, the function returns both files and directories.

Returns   An array of strings representing the contents of the folder.

Example   The following code gets a list of all the text (.txt) files in the temp folder and displays the list in an alert box.

```
var folderURL = "file:///c|/temp";
var fileMask = "*.txt";
var list = listFolder(folderURL + "/" + fileMask, "files");
if (list){
  alert(folderURL + " contains: " + list.join("\n"));
}
```

DWfile.read()

Availability   2.0

Description   Reads the contents of the specified file into a string.

Arguments   *fileURL*

The argument is the file you want to read, expressed as a file:// URL.

Returns   A string containing the contents of the file, or NULL if the read fails.

Example   The following code reads the file mydata.txt and, if successful, displays an alert box with the contents of the file.

```
var fileURL = "file:///c|/temp/mydata.txt";
var str = DWfile.read( fileURL);
if (str){
  alert( fileURL + " contains: " + str);
}
```

DWfile.remove()

Availability   3.0

Description   Moves the specified file to the Recycling Bin or Trash.

Note: Windows displays a confirmation dialog box if the file is read-only.

Arguments   *fileURL*

The argument is the file you want to remove, expressed as a file:// URL.

Returns   TRUE if the operation succeeded, FALSE otherwise.

DWfile.write()

Availability   2.0

Description   Writes the specified string to the specified file. If the specified file does not yet exist, it is created.

Arguments   *fileURL, text, {mode}*

- The first argument is the file you are writing to, expressed as a file:// URL.

- The second argument is the string to be written.

- The third argument, if supplied, must be "append". If this argument is omitted, the contents of the file are overwritten by the string.

Returns   TRUE if the string was successfully written to the file, FALSE otherwise.

Example   The following code attempts to write the string "xxx" to the file mydata.txt and displays an alert if the write succeeded. It then attempts to append the string "aaa" to the file and displays a second alert if the write succeeded. After executing this script, the file mydata.txt will contain the text xxxaaa and nothing else.

```
var fileURL = "file:///c|/temp/mydata.txt";
if (DWfile.write(fileURL, "xxx")){
 alert("Wrote xxx to " + fileURL);
}
if (DWfile.write(fileURL, "aaa", "append")){
 alert("Appended aaa to " + fileURL);
}
```

5

CHAPTER 5
The Design Notes API

Both Dreamweaver 3 and Fireworks 3 offer a new way for web designers and developers to store and retrieve extra information about documents—information such as review comments, change notes, or the source file for a GIF or JPEG—in files called Design Notes.

MMNotes is a C shared library that lets extensions authors read and write Design Notes files. Like the DWfile shared library, MMNotes has a JavaScript API that makes it possible to call the functions contained in the library from objects, comands, behaviors, floating palettes, property inspectors, and data translators.

What's different about MMNotes is that it also has a C API that gives other applications an opportunity to read and write Design Notes files. The MMNotes shared library can be used independently of Dreamweaver, even if Dreamweaver is not installed.

For more information about using the Design Notes feature from within Dreamweaver, see *Using Dreamweaver*.

How Design Notes work

Each Design Notes file stores information for a single document. If one or more documents in a directory has a Design Notes file associated with it, Dreamweaver creates a _notes subfolder in that directory in which to store Design Notes files. The _notes folder and the Design Notes files it contains are not visible in the Site window, but they show up in the Finder or Windows Explorer. A Design Notes file name is made up of the main file name plus the extension .mno. For example, the Design Notes file associated with avocado8.gif is avocado8.gif.mno.

Design Notes files are XML files that store information in a series of key/value pairs. The key describes the type of information that is being stored, and the value represents the information itself. Keys are limited to 64 characters.

The Design Notes file for foghorn.gif might look like this:

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<info>
  <infoitem key="FW_source" value="file:///C|sites/dreamcentral/images/sourceFiles/
foghorn.png" />
  <infoitem key="Author" value="Heidi B." />
  <infoitem key="Status" value="Final draft, approved by Jay L." />
</info>
```

The Design Notes JavaScript API

All of the functions in the Design Notes JavaScript API are methods of the MMNotes object. Optional arguments are enclosed in curly braces ({ }).

MMNotes.open()

Description   Opens the Design Notes file associated with the specified file, or creates one if none yet exists.

Arguments   *filePath, {bForceCreate}*

- The first argument is the path to the main file with which the Design Notes file is associated, expressed as a file:// URL.

- The second argument is a Boolean value indicating whether to create the Note even if Design Notes are turned off for the site or *filePath* is not associated with any site.

Returns   The file handle for the Design Notes file, or zero (0) if the file was not opened or created.

Example   See "MMNotes.set()" on page 185.

MMNotes.close()

Description Closes the specified Design Notes file and saves any changes. If all key/value pairs have been removed, Dreamweaver deletes the Design Notes file.

Arguments *fileHandle*

The argument is the file handle returned by MMNotes.open().

Returns Nothing.

Example See "MMNotes.set()" on page 185.

MMNotes.set()

Description Creates or updates one key/value pair in a Design Notes file.

Arguments *fileHandle, keyName, valueString*

- The first argument is the file handle returned by MMNotes.open().
- The second argument is a string containing the name of the key.
- The third argument is a string containing the value.

Returns A Boolean value indicating whether the operation was successful.

Example The following code opens the Design Notes file associated with a file in the dreamcentral site called peakhike99/index.html, adds a new key/value pair, changes the value of an existing key, and then closes the Note file.

```
var noteHandle = MMNotes.open('file:///c|sites/dreamcentral/
peakhike99/index.html',TRUE);
MMNotes.set(noteHandle,"Author","M. G. Miller");
MMNotes.set(noteHandle,"Last Changed","August 28, 1999");
MMNotes.close(noteHandle);
```

MMNotes.remove()

Description Removes the specified key (and its value) from the specified Design Notes file.

Arguments *fileHandle, keyName*

- The first argument is the file handle returned by MMNotes.open().
- The second argument is a string containing the name of the key to be removed.

Returns A Boolean value indicating whether the operation was successful.

MMNotes.get()

Description   Gets the value of the specified key in the specified Design Notes file.

Arguments   *fileHandle, keyName*

- The first argument is the file handle returned by MMNotes.open().
- The second argument is a string containing the name of the key.

Returns   A string containing the value of the key.

Example   See "MMNotes.getKeys()" on page 186.

MMNotes.getKeyCount()

Description   Gets the number of key/value pairs in the specified Design Notes file.

Arguments   *fileHandle*

The argument is the file handle returned by MMNotes.open().

Returns   An integer representing the number of key/value pairs in the Design Notes file.

MMNotes.getKeys()

Description   Gets a list of all the keys in a Design Notes file.

Arguments   *fileHandle*

The argument is the file handle returned by MMNotes.open().

Returns   Array of strings, each containing the name of a key.

Example   The following code might be used in a custom floating palette to display the Design Notes information for the active document:

```
var noteHandle = MMNotes.open(dw.getDocumentDOM().URL);
var theKeys = MMNotes.getKeys(noteHandle);
var noteString = "";
var theValue = "";
for (var i=0; i < theKeys.length; i++){
  theValue = MMNotes.get(noteHandle,theKeys[i]);
  noteString +=  theKeys[i] + " = " theValue + "\n";
}
document.theForm.bigTextField.value = noteString;
```

MMNotes.getSiteRootForFile()

Description Determines the site root for the specified Design Notes file.

Arguments *fileURL*

The argument is the path to a local file, expressed as a file:// URL.

Returns A string containing the path of the Local Root Folder for the site, expressed as a file:// URL, or an empty string if Dreamweaver is not installed or the Design Notes file is outside of any site.

MMNotes.getVersionNum()

Description Gets the version number of the MMNotes shared library.

Arguments None.

Returns A string containing the version number.

MMNotes.getVersionName()

Description Gets the version name of the MMNotes shared library, which indicates the application that implemented it.

Arguments None.

Returns A string containing the name of the application that implemented MMNotes shared library.

Example Calling MMNotes.getVersionName() from a Dreamweaver command, object, behavior, property inspector, floating palette, or data translator returns "Dreamweaver". Calling MMNotes.getVersionName() from Fireworks also returns "Dreamweaver" because Fireworks uses the same version of the library (the one created by the Dreamweaver engineering team).

MMNotes.filePathToLocalURL()

Description Converts the specified local drive path to a file:// URL.

Arguments *drivePath*

The argument is a string containing the full drive path.

Returns A string containing the file:// URL for the specified file.

Example A call to MMNotes.filePathToLocalURL('C:/sites/webdev/index.htm') returns "file:///c|sites/webdev/index.htm".

MMNotes.localURLToFilePath()

Description	Converts the specified file:// URL to a local drive path.
Arguments	*fileURL*
	The argument is the path to a local file, expressed as a file:// URL.
Returns	A string containing the local drive path for the specified file.
Example	A call to MMNotes.localURLToFilePath('file:///MacintoshHD/images/moon.gif') returns "MacintoshHD:images:moon.gif".

The Design Notes C API

In addition to the JavaScript API, the MMNotes shared library also exposes a C API that lets other applications create Design Notes files. It is not necessary to call these C functions directly if you are using the MMNotes shared library in Dreamweaver; the JavaScript versions of the functions call them for you.

This section contains descriptions of the functions, their arguments, and their return values; definitions for all of the functions and data types are in the MMInfo.h file in the Extending/c_files folder inside the Dreamweaver application folder.

Optional arguments are enclosed in curly braces ({ }).

FileHandle OpenNotesFile()

Description	Opens the Design Notes file associated with the specified file, or creates one if none yet exists.
Arguments	const char* *localFileURL*, {BOOL *bForceCreate*}

- The first argument is a string containing the path to the main file with which the Design Notes file is associated, expressed as a file:// URL.

- The second argument is a Boolean value indicating whether to create the Design Notes file even if Design Notes are turned off for the site or *filePath* is not associated with any site.

void CloseNotesFile()

Description	Closes the specified Design Notes file and saves any changes. If all key/value pairs have been removed from the Note, Dreamweaver deletes it.
Arguments	FileHandle *noteHandle*
	The argument is the file handle returned by OpenNotesFile().
Returns	Nothing.

BOOL SetNote()

Description Creates or updates one key/value pair in a Design Notes file.

Arguments FileHandle *noteHandle*, const char *keyName[64]*, const char* *value*

- The first argument is the file handle returned by OpenNotesFile().
- The second argument is a string containing the name of the key.
- The third argument is a string containing the value.

Returns A Boolean value indicating whether the operation was successful.

BOOL RemoveNote()

Description Removes the specified key (and its value) from the specified Design Notes file.

Arguments FileHandle *noteHandle*, const char *keyName[64]*

- The first argument is the file handle returned by OpenNotesFile().
- The second argument is a string containing the name of the key to be removed.

Returns A Boolean value indicating whether the operation was successful.

int GetNoteLength()

Description Gets the length of the value associated with the specified key.

Arguments FileHandle *noteHandle*, const char *keyName[64]*

- The first argument is the file handle returned by OpenNotesFile().
- The second argument is a string containing the name of the key.

Returns An integer representing the length of the value.

Example See "BOOL GetNote()" on page 190.

BOOL GetNote()

Description Gets the value of the specified key in the specified Design Notes file.

Arguments FileHandle *noteHandle*, const char *keyName[64]*, char* *valueBuf*, int *valueBufLength*

- The first argument is the file handle returned by OpenNotesFile().

- The second argument is a string containing the name of the key.

- The third argument is the buffer where the value should be stored.

- The fourth argument is the integer returned by GetNoteLength(*noteHandle,keyName*), indicating the maximum length of the value buffer.

Returns A Boolean value indicating whether the operation was successful, and stores the value of the key in *valueBuf*.

Example The following code gets the value of the comments key in the Design Notes file associated with welcome.html:

```
FileHandle noteHandle = OpenNotesFile("file:///c|sites/avocado8/iwjs/welcome.html");
int valueLength = GetNotesLength( noteHandle, "comments");
char* valueBuffer = new char[valueLength + 1]
GetNote(noteHandle, "comments", valueBuffer, valueLength + 1);
printf("Comments: %s",valueBuffer);
CloseNotesFile(noteHandle);
```

int GetNotesKeyCount()

Description Gets the number of key/value pairs in the specified Design Notes file.

Arguments FileHandle *noteHandle*

The argument is the file handle returned by OpenNotesFile().

Returns An integer representing the number of key/value pairs in the Design Notes file.

BOOL GetNotesKeys()

Description   Gets a list of all the keys in a Design Notes file.

Arguments   FileHandle *noteHandle*, char* *keyBufArray[64]*, int *keyArrayMaxLen*

- The first argument is the file handle returned by OpenNotesFile().

- The second argument is the buffer array where the keys should be stored.

- The third argument is the integer returned by GetNotesKeyCount(*noteHandle*), indicating the maximum number of items in the key buffer array.

Returns   A Boolean value indicating whether the operation was successful, and stores the key names in *keyBufArray*.

Example   The following code prints the key names and values of all the keys in the Design Notes file associated with welcome.html:

```
typedef char[64] InfoKey;
FileHandle noteHandle = OpenNotesFile("file:///c|sites/avocado8/iwjs/welcome.html");
if (noteHandle > 0){
  int keyCount = GetNotesKeyCount(noteHandle);
  if (keyCount <= 0)
    return;
  InfoKey* keys = new InfoKey[keyCount];
  BOOL succeeded = GetNotesKeys(noteHandle, keys, keyCount);

  if (succeeded){
    for (int i=0; i < keyCount; i++){
      printf("Key is: %s\n", keys[i]);
      printf("Value is: %s\n\n", GetNote(noteHandle, keys[i]));
    }
  delete keys;
  }
CloseNotesFile(noteHandle);
```

BOOL GetSiteRootForFile()

Description   Determines the site root for the specified Design Notes file.

Arguments   char* *filePath*, char* *siteRootBuf*, int *siteRootBufMaxLen,* {InfoPrefs* *infoPrefs*}

- The first argument is the file handle returned by OpenNotesFile().

- The second argument is the buffer where the site root should be stored.

- The third argument is the maximum size of *siteRootBuf*.

- The fourth argument is a reference to a struct in which the preferences for the site should be stored.

Returns   A Boolean value indicating whether the operation was successful, and stores the site root in *siteRootBuf*. If *infoPrefs* is specified, the function also returns the Design Notes preferences for the site. The InfoPrefs struct has two variables: bUseDesignNotes and bUploadDesignNotes, both of type BOOL.

BOOL GetVersionNum()

Description   Gets the version number of the MMNotes shared library.

Arguments   char* *versionNumBuf*, int *versionNumBufMaxLen*

- The first argument is the buffer where the version number should be stored.
- The second argument is the maximum size of *versionNumBuf*.

Returns   A Boolean value indicating whether the operation was successful, and stores the version number in *versionNumBuf*.

BOOL GetVersionName()

Description   Gets the version name of the MMNotes shared library, which indicates the application that implemented it.

Arguments   char* *versionNameBuf*, int *versionNameBufMaxLen*

- The first argument is the buffer where the version name should be stored.
- The second argument is the maximum size of *versionNameBuf*.

Returns   A Boolean value indicating whether the operation was successful, and stores the version name in *versionNameBuf*.

BOOL FilePathToLocalURL()

Description   Converts the specified local drive path to a file:// URL.

Arguments   const char* *drivePath*, char* *localURLBuf*, int *localURLMaxLen*

- The first argument is a string containing the full drive path.
- The second argument is the buffer shere the file:// URL should be stored..
- The third argument is the maximum size of *localURLBuf*.

Returns   A Boolean value indicating whether the operation was successful, and stores the file:// URL in *localURLBuf*.

BOOL LocalURLToFilePath()

Description   Converts the specified file:// URL to a local drive path.

Arguments   const char* *localURL*, char* *drivePathBuf*, int *drivePathMaxLen*

- The first argument is the path to a local file, expressed as a file:// URL.
- The second argument is the buffer where the local drive path should be stored.
- The third argument is the maximum size of *drivePathBuf*.

Returns   A Boolean value indicating whether the operation was successful, and stores the local drive path in *drivePathBuf*.
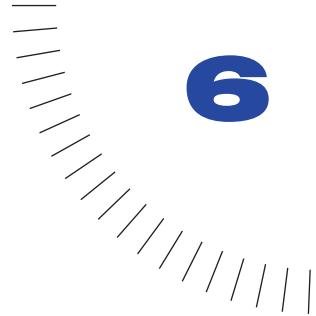
CHAPTER 6
The Fireworks Integration API

FWLaunch is a C shared library that gives authors of objects, commands, behaviors, and property inspectors the ability to communicate with Fireworks. This chapter describes the Fireworks integration API and how to use it; for general information on how C libraries interact with the JavaScript interpreter in Dreamweaver, see "C-Level Extensibility" on page 209

The Fireworks integration API

All functions in the Fireworks integration API are methods of the FWLaunch object. Optional arguments are enclosed in curly braces ({ }).

FWLaunch.bringDWToFront()

Availability	Dreamweaver 3.0, Fireworks 3.0
Description	Brings Dreamweaver to the front.
Arguments	None.
Returns	Nothing.

FWLaunch.bringFWToFront()

Availability	Dreamweaver 3.0, Fireworks 3.0
Description	Brings Fireworks to the front if it is running.
Arguments	None.
Returns	Nothing.

FWLaunch.execJsInFireworks()

Availability   Dreamweaver 3.0, Fireworks 3.0

Description   Passes the specified string of JavaScript to Fireworks for execution.

Arguments   *javascriptOrFileURL*

The argument is either a string of literal JavaScript or the path to a .js or .jsf file, expressed as a file:// URL.

Returns   A cookie object if the JavaScript was passed successfully, or a nonzero error code indicating that one of the following errors has occurred:

- 1: Invalid usage; *javascriptOrFileURL* was specified as NULL or empty string, or the path to the .js or .jsf file was invalid.

- 2: File I/O error; Fireworks is unable to create a response file because the disk is full.

- 3: Error notifying Dreamweaver; the user is not running a valid version of Dreamweaver (3.0 or later).

- 4: Error launching Fireworks process; the function did not launch a valid version of Fireworks (3.0 or later).

- 5: User canceled the operation.

FWLaunch.getJsResponse()

Availability   Dreamweaver 3.0, Fireworks 3.0

Description   Determines whether Fireworks is still executing the JavaScript passed to it by FWLaunch.execJsInFireworks(), whether the script has completed successfully, or whether an error has occurred.

Arguments   *progressTrackerCookie*

The argument is the cookie object returned by FWLaunch.execJsInFireworks().

Returns   A string containing the result of the script passed to FWLaunch.execJsInFireworks() if the operation completed successfully, NULL if Fireworks is still executing the JavaScript, or a nonzero error code indicating that one of the following errors has occurred:

- 1: Invalid usage; a JavaScript error occurred as Fireworks was executing the script.

- 2: File I/O error; Fireworks is unable to create a response file because the disk is full.

- 3: Error notifying Dreamweaver; the user is not running a valid version of Dreamweaver (3.0 or later).

- 4: Error launching Fireworks process; the function did not launch a valid version of Fireworks (3.0 or later).

- 5: User canceled the operation.

Returns   The following code passes the string "prompt('Please enter your name:')" to FWLaunch.execJsInFireworks() and then checks for the result:

```
var progressCookie = FWLaunch.execJsInFireworks("prompt('Please enter your name:')");
var doneFlag = false;
while (!doneFlag){
  // check for completion every 1/2 second
  setTimeout('checkForCompletion()',500);
}

function checkForCompletion(){
  if (progressCookie != null) {
    var response = FWLaunch.getJsResponse(progressCookie);
    if (response != null) {
      if (typeof(response) == "number") {
        // error or user-cancel, time to close the window and let the user know we got an error
        window.close();
        alert("An error occurred.");
      }else{
        // got a valid response!
        alert("Nice to meet you, " + response);
        window.close();
      }
      doneFlag = true;
    }
  }
}
```

FWLaunch.mayLaunchFireworks()

Availability   Dreamweaver 2.0, Fireworks 2.0

Description   Determines whether it is possible to launch a Fireworks optimization session.

Arguments   None.

Returns   A Boolean value indicating whether the platform is Windows, or if Macintosh, whether another Fireworks optimization session is not already running.

FWLaunch.optimizeInFireworks()

Availability   Dreamweaver 2.0, Fireworks 2.0

Description   Launches a Fireworks optimization session for the specified image.

Arguments   *docURL, imageURL, {targetWidth}, {targetHeight}*

- The first argument is the path to the active document, expressed as a file:// URL.

- The second argument is the path to the selected image. If the path is relative, it is relative to *docURL*.

- The third argument, if supplied, is the width to which the image should be resized.

- The fourth argument, if supplied, is the height to which the image should be resized.

Returns   0 if a Fireworks optimization session is successfully launched for the specified image; otherwise, a nonzero error code indicating that one of the following errors has occurred:

- 1: Invalid usage; *docURL*, *imageURL*, or both were specified as NULL or empty string.

- 2: File I/O error; Fireworks is unable to create a response file because the disk is full.

- 3: Error notifying Dreamweaver; the user is not running a valid version of Dreamweaver (2.0 or later).

- 4: Error launching Fireworks process; the function did not launch a valid version of Fireworks (2.0 or later).

- 5: User canceled the operation.

FWLaunch.validateFireworks()

Availability   Dreamweaver 2.0, Fireworks 2.0

Description   Looks for the specified version of Fireworks on the user's hard drive.

Arguments   {*versionNumber*}

The argument is a floating point number greater than or equal to 2.0; it represents the version of Fireworks that should be searched for. If this argument is omitted, the default is 2.0.

Returns   A Boolean value indicating whether the specified version of Fireworks was found.

Example   The following code checks whether Fireworks 3.0 is installed:

```
if (FWLaunch.validateFireworks(3.0)){
  alert( "Fireworks 3.0 is installed.");
}else{
  alert( "Fireworks 3.0 is not installed.");
}
```

A simple Fireworks integration example

The following command asks Fireworks to prompt the user for her name, and then returns the name to Dreamweaver.

```
<html>
<head>
<title>Prompt in Fireworks</title>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1">
<script>

function commandButtons(){
  return new Array("Prompt", "promptInFireworks()", "Cancel", "readyToCancel()",
"Close","window.close()");
}

var gCancelClicked = false;
var gProgressTrackerCookie = null;

function readyToCancel() {
  gCancelClicked = true;
}

function promptInFireworks() {
  var isFireworks3 = FWLaunch.validateFireworks(3.0);
  if (!isFireworks3) {
    alert("You must have Fireworks 3.0 or later to use this command");
    return;
  }

  // Tell Fireworks to execute the prompt() method.
  gProgressTrackerCookie = FWLaunch.execJsInFireworks("prompt('Please enter your
name:')");

  // null means it wasn't launched, a number means an error code
  if (gProgressTrackerCookie == null || typeof(gProgressTrackerCookie) == "number") {
    window.close();
    alert("an error occurred");
    gProgressTrackerCookie = null;
  } else {
    // bring Fireworks to the front
    FWLaunch.bringFWToFront();
    // start the checking to see if Fireworks is done yet
    checkOneMoreTime();
  }
}

function checkOneMoreTime() {
  // Call checkJsResponse() every 1/2 second to see if Fireworks is done yet
  window.setTimeout("checkJsResponse();", 500);
}
```

```
function checkJsResponse() {
 var response = null;

 // The user clicked the cancel button, close the window
 if (gCancelClicked) {
   window.close();
   alert("cancel clicked");
 } else {
   // We're still going, ask Fireworks how it's doing
   if (gProgressTrackerCookie != null)
     response = FWLaunch.getJsResponse(gProgressTrackerCookie);

   if (response == null) {
     // still waiting for a response, call us again in 1/2 a second
     checkOneMoreTime();
   } else if (typeof(response) == "number") {
     // if the response was a number, it means an error occurred
     // the user cancelled in Fireworks
     window.close();
     alert("an error occurred.");

   } else {
     // got a valid response!  This return value might not always be a
     // useful one, since not all functions in Fireworks return a string,
     // but we know this one does, so we can show the user what we got.
     window.close();
     FWLaunch.bringDWToFront();  // bring Dreamweaver to the front
     alert("Nice to meet you, " + response + "!");
   }
 }
}

</script>
</head>
<body>
<form>
<table width="313" nowrap>
<tr>
<td>This command asks Fireworks to execute the prompt() function.
When you click Prompt, Fireworks comes forward and asks you to
enter a value into a dialog box. That value is then returned to
Dreamweaver and displayed in an alert.</td>
</tr>
</table>
</form>
</body>
</html>
```

7
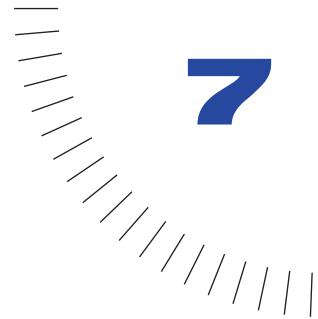
CHAPTER 7
The HTTP API

. . . .

In Dreamweaver 3, extensions are no longer limited to working within the local
file system. Dreamweaver now provides a mechansim for getting information
from and sending information to a web server via hypertext transfer protocol
(HTTP). This chapter describes the HTTP API and how to use it.

The HTTP API

All of the functions in the HTTP API are methods of the MMHttp object. Most take at least a URL as an argument, and most return an object. The default port for URL arguments is 80; to specify a port other than 80, append a colon and the port number to the URL. For example:

MMHttp.getText("http://www.myserver.com:8025");

For functions that return an object, the object has two properties: statusCode and data.

statusCode indicates the status of the operation; possible values include, but are not limited to:

- 200: Status OK

- 400: Unintelligible request

- 404: Requested URL not found

- 405: Server does not support requested method

- 500: Unknown server error

- 503: Server capacity reached

For a comprehensive list of status codes for your server, check with your Internet service provider or system administrator.

The value of the data property varies according to the function; possible values are specified in the individual function listings.

Functions that return an object also have a "callback" version. Callback functions allow other functions to execute while the web server processes an HTTP request. This is useful if you are making multiple HTTP requests from Dreamweaver. The callback version of a function passes its ID and return value directly to the function specified as its first argument.

Optional arguments are enclosed in curly braces ({ }).

MMHttp.clearTemp()

Description Deletes all files in the Configuration/Temp folder inside the Dreamweaver application folder.

Arguments None.

Returns Nothing.

Example The following code, when saved in a file inside the Configuration/Shutdown folder, removes all of the files from the Configuration/Temp folder when the user quits Dreamweaver:

```
<html>
<head>
<title>Clean Up Temp Files on Shutdown</title>
</head>
<body onLoad="MMHttp.clearTemp()">
</body>
</html>
```

MMHttp.getFile()

Description Gets the file at the specified URL and saves it in the Configuration/Temp folder inside the Dreamweaver application folder on the user's hard drive. Dreamweaver automatically creates subfolders that mimic the directory structure of the server; for example, if the specified file is at http://www.dreamcentral.com/people/index.html, Dreamweaver stores the index.html file in the people folder inside the www.dreamcentral.com folder.

Arguments *URL, {prompt}, {saveURL}, {titleBarLabel}*

- The first argument is an absolute URL on a web server; if the "http://" part of the URL is omitted, it is assumed.

- The second argument is a Boolean value that specifies whether to prompt the user to save the file. If *saveURL* is outside the Configuration/Temp folder, a *prompt* value of FALSE is ignored for security reasons.

- The third argument is the location on the user's hard drive where the file should be saved, expressed as a file:// URL. If prompt is TRUE or *saveURL* is outside the Configuration/Temp folder, the user can override *saveURL* in the Save dialog box.

- The fourth argument is the label that should appear in the title bar of the Save dialog box.

Returns   An object that represents the reply from the server. The data property of this object is a string containing the location at which the file was saved, expressed as a file:// URL. Normally the statusCode property of the object contains the status code received from the server. However, if a disk error occurs while saving the file on the local drive, the statusCode property contains an integer representing one of the following error codes if the operation was not successful:

- 1: Unspecified error
- 2: File not found
- 3: Invalid path
- 4: Number of open files limit reached
- 5: Access denied
- 6: Invalid file handle
- 7: Cannot remove current working directory
- 8: No more directory entries
- 9: Error setting file pointer
- 10 Hardware error
- 11: Sharing violation
- 12:Lock violation
- 13: Disk full
- 14: End of file reached

Example   The following code gets an HTML file, saves all the files in the Configuration/ Temp folder, and then opens the local copy of the HTML file in a browser:

```
var httpReply = MMHttp.getFile("http://www.dreamcentral.com/people/profiles/scott.html",
false);
if (httpReply.statusCode == 200){
 var saveLoc = httpReply.data;
 dw.browseDocument(saveLoc);
}
```

MMHttp.getFileCallback()

Description Gets the file at the specified URL, saves it in the Configuration/Temp folder inside the Dreamweaver application folder on the user's hard drive, and then calls the specified function with the request ID and reply result. When saving the file locally, Dreamweaver automatically creates subfolders that mimic the directory structure of the server; for example, if the specified file is at http://www.dreamcentral.com/people/index.html, Dreamweaver stores the index.html file in the people folder inside the www.dreamcentral.com folder.

Arguments *callbackFunction, URL, {prompt}, {saveURL}, {titleBarLabel}*

- The first argument is the name of the JavaScript function to call when the HTTP request is complete.

- The second argument is an absolute URL on a web server; if the "http://" part of the URL is omitted, it is assumed.

- The third argument is a Boolean value that specifies whether to prompt the user to save the file. If *saveURL* is outside the Configuration/Temp folder, a *prompt* value of FALSE is ignored for security reasons.

- The fourth argument is the location on the user's hard drive where the file should be saved, expressed as a file:// URL. If prompt is TRUE or *saveURL* is outside the Configuration/Temp folder, the user can override *saveURL* in the Save dialog box.

- The fifth argument is the label that should appear in the title bar of the Save dialog box.

Returns An object that represents the reply from the server. The data property of this object is a string containing the location at which the file was saved, expressed as a file:// URL. Normally the statusCode property of the object contains the status code received from the server. However, if a disk error occurs while saving the file on the local drive, the statusCode property contains an integer representing an error code. See "MMHttp.getFile()" on page 203 for a list of possible error codes.

MMHttp.getText()

Description   Retrieves the content of the document at the specified URL.

Arguments   *URL*

         The argument is an absolute URL on a web server; if the "http://" part of the URL is omitted, it is assumed.

Returns   An object that represents the reply from the server. The data property of this object is a string containing the contents of the document.

Example   The following code gets the content of a file on a web server and puts it in a new, untitled Dreamweaver document:

```
var httpReply = MMHttp.getText("http://www.dreamcentral.com/people/profiles/lori.html");
if (httpReply.statusCode == 200){
 var newDoc =  dw.createDocument();
 newDoc.documentElement.outerHTML = httpReply.data;
}
```

MMHttp.getTextCallback()

Description   Retrieves the content of the document at the specified URL and passes it to the specified function.

Arguments   *callbackFunc, URL*

- The first argument is the name of the JavaScript function to call when the HTTP request is complete.

- The second argument is an absolute URL on a web server; if the "http://" part of the URL is omitted, it is assumed.

Returns   An object that represents the reply from the server. The data property of this object is a string containing the contents of the document.

Example   The following code populates a form field with the text returned by the MMHttp.GetTextCallback() function or shows an error message if the function returned an error:

```
var requestID = MMHttp.getTextCallback("httpCallback", "www.dreamcentral.com/
index.html")

function httpCallback(requestID,reply) {
 if (reply.statusCode == 200) {
   document.theForm.docContents.value = reply.data;
 }else{
   alert("Request #: " + requestID + "returned the following error: " + reply.statusCode);
 }
}
```

MMHttp.postText()

Description   Performs an HTTP post of the specified data to the specified URL. Typically the data associated with a post operation is form-encoded text, but it could be any type of data that the server expects to receive.

Arguments   *URL, dataToPost, {contentType}*

- The first argument is an absolute URL on a web server; if the "http://" part of the URL is omitted, it is assumed.

- The second argument is the data to be posted. If the third argument is "application/x-www-form-urlencoded" or omitted, *dataToPost* must be form encoded according to section 8.2.1 of the RFC 1866 specification (available at http://www.faqs.org/rfcs/rfc1866.html).

- The third argument is the content type of the data to be posted. If omitted, this argument defaults to "application/x-www-form-urlencoded".

Returns   An object that represents the reply from the server. The data property of this object is a string containing the data resulting from the post operation.

MMHttp.postTextCallback()

Description   Performs an HTTP post of the text to the specified URL and passes the reply from the server to the specified function. Typically the data associated with a post operation is form-encoded text, but it could be any type of data that the server expects to receive.

Arguments   *callbackFunc, URL, dataToPost, {contentType}*

- The first argument is the name of the JavaScript function to call when the HTTP request is complete.

- The second argument is an absolute URL on a web server; if the "http://" part of the URL is omitted, it is assumed.

- The third argument is the data to be posted. If the third argument is "application/x-www-form-urlencoded" or omitted, *data* must be form encoded according to section 8.2.1 of the RFC 1866 specification (available at http://www.faqs.org/rfcs/rfc1866.html)

- The fourth argument is the content type of the data to be posted. If omitted, this argument defaults to "application/x-www-form-urlencoded".

Returns   An object that represents the reply from the server. The data property of this object is a string containing the data resulting from the post operation.

CHAPTER 8
C-Level Extensibility

The C-level extensibility mechanism lets you implement Dreamweaver extensibility files using a combination of JavaScript and your own C code. You define functions using C, bundle them in a DLL or shared library, save the library in the Configuration/JSExtensions folder within the Dreamweaver application folder, and then call the functions from JavaScript using the JavaScript interpreter that is built into Dreamweaver.

For example, you may want to define a Dreamweaver object that inserts the contents of a user-specified file into the current document. Because client-side JavaScript does not provide any support for file I/O, you must write a function in C to provide this functionality.

You could use the following HTML and JavaScript to create a simple Insert Text from File object. Notice that the objectTag() function calls a C function named readContentsOfFile(), which is stored in a library named myLibrary.

```
<HTML>
<HEAD>
<SCRIPT>
function objectTag() {
  fileName = document.forms[0].myFile.value;
  return myLibrary.readContentsOfFile(fileName);
}
</SCRIPT>
</HEAD>

<BODY>
<FORM>
Enter the name of the file to be inserted:
<INPUT TYPE="file" NAME="myFile">
</FORM>
</BODY>
</HTML>
```

The readContentsOfFile() function accepts a list of arguments from the user, unpacks the argument containing the file name, reads the contents of the file, and packages the contents of the file as the return value. For more information about the JavaScript data structures and functions that appear in readContentsOfFile(), see "The C-level extensibility API" on page 211.

```
JSBool
readContentsOfFile(JSContext *cx, JSObject *obj, unsigned int argc, jsval *argv, jsval *rval)
{
  char *fileName, *fileContents;
  JSBool success;
  unsigned int length;

  /* Make sure caller passed in exactly one argument. If not,
   * then tell the interpreter to abort script execution. */
  if (argc != 1){
    JS_ReportError(cx, "Wrong number of arguments", 0);
    return JS_FALSE;
  }

  /* Convert the argument to a string */
  fileName = JS_ValueToString(cx, argv[0], &length);
  if (fileName == NULL){
    JS_ReportError(cx, "The argument must be a string", 0);
    return JS_FALSE;
  }

  /* Use the string (the file name) to open and read a file */
  fileContents = exerciseLeftToTheReader(fileName);

  /* Store file contents in rval, which is the return value passed
   * back to the caller */
  success = JS_StringToValue(cx, fileContents, 0, *rval);
  free(fileContents);

  /* Return true to continue or false to abort the script */
  return success;
}
```

To ensure that the readContentsOfFile() function executes as designed rather than causing a JavaScript error, you must register the function with the JavaScript interpreter by including a function called MM_Init() in your library. When Dreamweaver loads the library at startup, it calls the MM_Init() function to get three pieces of information:

• The JavaScript name of the function

• A pointer to the function

• The number of arguments that the function expects

The MM_Init() function for myLibrary might look like this:

```
void
MM_Init()
{
  JS_DefineFunction("readContentsOfFile", readContentsOfFile, 1);
}
```

Your library must include exactly one instance of the following macro:

```
/* MM_STATE is a macro that expands to some definitions that are
 * needed to interact with Dreamweaver. This macro must
 * be defined exactly once in your library. */
MM_STATE
```

Note: The library can be implemented in either C or C++, but the file containing MM_Init() and MM_STATE must be implemented in C. The C++ compiler garbles function names, making it impossible for Dreamweaver to find the MM_Init() function.

The C-level extensibility API

The C code in your library must interact with Dreamweaver's JavaScript interpreter at three different times:

- At startup, to register the library's functions.

- When the function is called, to unpack the arguments that are being passed from JavaScript to C.

- Before the function returns, to package the return value.

To accomplish these tasks, the interpreter defines several data types and exposes an API. Definitions for the following data types and functions appear in the file mm_jsapi.h. For your library to work properly, you must include mm_jsapi.h at the top of each file in your library with the following line:

```
#include "mm_jsapi.h"
```

typedef struct JSContext JSContext

Description   A pointer to this opaque data type is passed to the C-level function. Some of the functions in the API accept this pointer as one of their arguments.

typedef struct JSObject JSObject

Description   A pointer to this opaque data type is passed to the C-level function. This data type represents an object, which may be an array object or some other object type.

typedef struct jsval jsval

Description   An opaque data structure that can contain an integer, or a pointer to a float, string, or object. Some functions in the API can be used to read the values of function arguments by reading the contents of a jsval, and some can be used to write the function's return value by writing a jsval.

typedef enum { JS_FALSE = 0, JS_TRUE = 1 } JSBool

Description   A simple data type used to store a Boolean value.

typedef JSBool (*JSNative)(JSContext *cx, JSObject *obj, unsigned int argc, jsval *argv, jsval *rval)

Description   The function signature for C-level implementations of JavaScript functions, where:

- cx is a pointer to an opaque JSContext structure, which must be passed to some of the functions in the JavaScript API. This variable holds the interpreter's execution context.

- *obj* is a pointer to the object in whose context the script executes. While the script is running, the this keyword is equal to this object.

- *argc* is the number of arguments being passed to the function.

- *argv* is a pointer to an array of jsvals. The array is argc elements in length.

- *rval* is a pointer to a single jsval. The function's return value should be written to *rval.

The function returns JS_TRUE upon success or JS_FALSE upon failure. If JS_FALSE is returned, the current script stops executing.

JSBool JS_DefineFunction()

Description   Registers a C-level function with the JavaScript interpreter in Dreamweaver. After this function returns, JavaScript scripts that call the function specified in name will execute the code pointed to by call.

Typically, this function is called from MM_Init(), which Dreamweaver calls during startup.

Arguments   char *name, JSNative *call,* unsigned int *nargs*

- *name* is the name of the function as it is exposed to JavaScript.

- *call* is a pointer to a C-level function. The function must accept the same arguments as readContentsOfFile, and it must return a JSBool indicating success or failure.

- *nargs* is the number of arguments that the function expects to receive.

Returns   A Boolean value indicating success (JS_TRUE) or failure (JS_FALSE).

char *JS_ValueToString()

Description Extracts a function argument from a jsval, converts it to a string (if possible), and passes the converted value back to the caller.

Arguments JSContext *cx, jsval v, unsigned int *pLength

- *cx* is the opaque JSContext pointer that was passed to the JavaScript function.

- *v* is the jsval from which the string is to be extracted.

- *pLength* is a pointer to an unsigned integer. This function sets *plength equal to the length of the string in bytes.

Returns A pointer to a string on success or NULL on failure.

JSBool JS_ValueToInteger()

Description Extracts a function argument from a jsval, converts it to an integer (if possible), and passes the converted value back to the caller.

Arguments JSContext *cx, jsval v, long *lp

- *cx* is the opaque JSContext pointer that was passed to the JavaScript function.

- *v* is the jsval from which the string is to be extracted.

- *lp* is a pointer to a 4-byte long integer. This function stores the converted value in *lp.

Returns A Boolean value indicating success (JS_TRUE) or failure (JS_FALSE).

JSBool JS_ValueToDouble()

Description Extracts a function argument from a jsval, converts it to a double (if possible), and passes the converted value back to the caller.

Arguments JSContext *cx, jsval v, double *dp

- *cx* is the opaque JSContext pointer that was passed to the JavaScript function.

- *v* is the jsval from which the string is to be extracted.

- *dp* is a pointer to an 8-byte double. This function stores the converted value in *dp.

Returns A Boolean value indicating success (JS_TRUE) or failure (JS_FALSE).

JSBool JS_ValueToBoolean()

Description   Extracts a function argument from a jsval, converts it to a Boolean value (if possible), and passes the converted value back to the caller.

Arguments   JSContext *cx, jsval v, JSBool *bp

- *cx* is the opaque JSContext pointer that was passed to the JavaScript function.

- *v* is the jsval from which the string is to be extracted.

- *bp* is a pointer to a JSBool. This function stores the converted value in *bp.

Returns   A Boolean value indicating success (JS_TRUE) or failure (JS_FALSE).

JSBool JS_ValueToObject()

Description   Extracts a function argument from a jsval, converts it to an object (if possible), and passes the converted value back to the caller. If the object is an array, use JS_GetArrayLength() and JS_GetElement() to read its contents.

Arguments   JSContext *cx, jsval v, JSObject **op

- *cx* is the opaque JSContext pointer that was passed to the JavaScript function.

- *v* is the jsval from which the string is to be extracted.

- *op* is a pointer to a (JSObject *). This function stores the converted value in *op.

Returns   A Boolean value indicating success (JS_TRUE) or failure (JS_FALSE).

JSBool JS_StringToValue()

Description   Stores a string return value in a jsval.

Arguments   JSContext *cx, char *bytes, size_t sz, jsval *vp

- *cx* is the opaque JSContext pointer that was passed to the JavaScript function.

- *bytes* is the string to be stored in the jsval. The string data is copied, so the caller should free the string when it is no longer needed. If the string size is not specified (see the *sz* argument), then the string must be null terminated.

- *sz* is the size of the string, in bytes. If *sz* is 0, then the length of the null-terminated string is computed automatically.

- *vp* is a pointer to the jsval into which the contents of the string should be copied.

Returns   A Boolean value indicating success (JS_TRUE) or failure (JS_FALSE).

JSBool JS_DoubleToValue()

Description   Stores a floating-point number return value in a jsval.

Arguments   JSContext *cx, double *dv,* jsval *vp

- *cx* is the opaque JSContext pointer that was passed to the JavaScript function.

- *dv* is an 8-byte floating-point number.

- *vp*is a pointer to the jsval into which the contents of the double should be copied.

Returns   A Boolean value indicating success (JS_TRUE) or failure (JS_FALSE).

JSBool JS_IntegerToValue()

Description   Stores an integer return value in a jsval.

Arguments   long *lv*

Returns   A Boolean value indicating success (JS_TRUE) or failure (JS_FALSE).

JSBool JS_BooleanToValue()

Description   Stores a Boolean return value in a jsval.

Arguments   JSBool *bv*

Returns   A Boolean value indicating success (JS_TRUE) or failure (JS_FALSE).

JSBool JS_ObjectToValue()

Description   Stores an object return value in a jsval. Use JS_ NewArrayObject() to create an array object, and use JS_SetElement() to define its contents.

Arguments   JSObject *obj*

Returns   A Boolean value indicating success (JS_TRUE) or failure (JS_FALSE).

char *JS_ObjectType()

Description   Given an object reference, returns a string describing the type of the object. For array objects, the return value is Array.

Arguments   JSObject *obj*

Typically, this argument is passed in and converted using JS_ValueToObject().

Returns   A pointer to a null-terminated string. The caller should *not* free this string when it has finished.

JSObject *JS_NewArrayObject()

Description Creates a new object that contains an array of jsvals.

Arguments JSContext *cx, unsigned int *length,* jsval *v

- *cx* is the opaque JSContext pointer that was passed to the JavaScript function.

- *length* is the number of elements that the array will hold.

- *v* is an optional pointer to the jsvals to be stored in the array. If the return value is not NULL, *v* is an array containing *length* elements. If the return value is NULL, then the initial content of the array object is undefined (and may be set using JS_SetElement()).

Returns A pointer to a new array object, or NULL upon failure.

long JS_GetArrayLength()

Description Given a pointer to an array object, gets the number of elements in the array.

Arguments JSContext *cx, JSObject *obj

- *cx* is the opaque JSContext pointer that was passed to the JavaScript function.

- *obj* is a reference to an array object.

Returns The number of elements in the array, or -1 upon failure.

JSBool JS_GetElement()

Description Reads a single element of an array object.

Arguments JSContext *cx, JSObject *obj, unsigned int *index,* jsval *v

- *cx* is the opaque JSContext pointer that was passed to the JavaScript function.

- *obj* is a pointer to an array object.

- *index* is an integer index into the array. The first element is index 0, and the last element is index (length - 1).

- *v* is a pointer to a jsval where the contents of the jsval in the array should be copied.

Returns A Boolean value indicating success (JS_TRUE) or failure (JS_FALSE).

JSBool JS_SetElement()

Description   Writes a single element of an array object.

Arguments   JSContext *cx, JSObject *obj, unsigned int *index,* jsval *v

- *cx* is the opaque JSContext pointer that was passed to the JavaScript function.

- *obj* is a pointer to an array object.

- *index* is an integer index into the array. The first element is index 0, and the last element is index (length - 1).

- *v* is a pointer to a jsval whose contents should be copied to the jsval in the array.

Returns   A Boolean value indicating success (JS_TRUE) or failure (JS_FALSE).

JSBool JS_ExecuteScript()

Description   Compiles and executes a string of Javascript. If the script generates a return value, it is returned in *rval.

Arguments   JSContext *cx, JSObject *obj, char *script, unsigned in *sz,* jsval *rval

- *cx* is the opaque JSContext pointer that was passed to the JavaScript function.

- *obj* is a pointer to the object in whose context the script executes. While the script is running, the this keyword is equal to this object. Usually this is the JSObject pointer that was passed to the JavaScript function.

- *script* is a string containing JavaScript code. If the string size is not specified (see the *sz* argument), then the string must be null-terminated.

- *sz* is the size of the string, in bytes. If sz is 0, then the length of the null-terminated string is computed automatically.

- *rval* is a pointer to a single jsval. The function's return value is stored in *rval.

Returns   A Boolean value indicating success (JS_TRUE) or failure (JS_FALSE).

JSBool JS_ReportError()

Description Describes the reason for a script error. Call this function before returning JS_FALSE to give the user information about why the script failed (for example, "wrong number of arguments").

Arguments JSContext *cx, char *error, size_t sz

- *cx* is the opaque JSContext pointer that was passed to the JavaScript function.

- *error* is a string containing the error message. The string is copied, so the caller should free the string when it is no longer needed. If the string size is not specified (see sz, below), then the string must be null-terminated.

- *sz* is the size of the string, in bytes. If sz is 0, then the length of the null-terminated string is computed automatically.

Returns A Boolean value indicating success (JS_TRUE) or failure (JS_FALSE).

Calling a C function from JavaScript

Once you know how C-level extensibility works in Dreamweaver and the data types and functions it relies on, it's useful to walk through an example of how to build a library and call a function.

This exercise requires three files, which are included in the Extending/c_files folder inside the Dreamweaver application folder:

- mm_jsapi.h is a header file that includes definitions for all the data types and functions described in "The C-level extensibility API" on page 211

- Sample.c is an example file that defines the computeSum() function.

- Sample.mak is a makefile that you can use to build Sample.c into a DLL with Microsoft Visual C++; Sample.proj is the equivalent file for building a CFM Library with Metrowerks CodeWarrior. If you are using another tool, you can create the makefile yourself.

To build the DLL in Windows:

1 In Microsoft Visual C++, choose File > Open Workspace and select Sample.mak.

2 Choose Build > Rebuild All.

   When the build operation is complete, a file called Sample.dll appears in the folder containing Sample.mak (or one of its subfolders).

To build the shared library on the Macintosh:

1 Open Sample.proj in Metrowerks CodeWarrior.

2 Build the project to generate a CFM Library.

When the build operation has finished, a file called Sample appears in the folder containing Sample.proj (or in one of its subfolders).

To call the computeSum() function from the Insert Horizontal Rule object:

1 In the Configuration folder within the Dreamweaver application folder, create a folder called JSExtensions.

2 Copy Sample.dll (Windows) or Sample (Macintosh) to the JSExtensions folder.

3 In a text editor, open the file called horizontal_rule.htm that resides in the Configuration/Objects/Common folder.

4 Add the line alert(Sample.computeSum(2,2)); to the objectTag() function so that it appears as follows:

```
function objectTag() {
  // Return the html tag that should be inserted
  alert(Sample.computeSum(2,2));
  return "<HR>";
}
```

5 Save the file and restart Dreamweaver.

To execute the computeSum() function:

Choose Insert > Horizontal Rule. A dialog box containing the number 4—the result of computing the sum of 2 plus 2—appears.

9
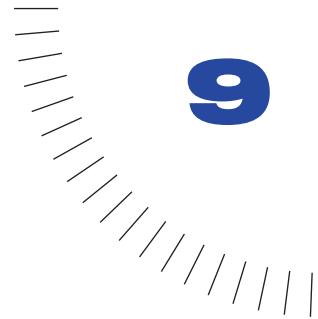
CHAPTER 9
Objects
· ·

Objects are designed to insert a specific string of code into the user's document. An object appears in a panel on the Object palette and in the Insert menu once its object file is stored in a subfolder within the Configuration/Objects folder inside the Dreamweaver application folder.

Objects have two components: the object file that defines what is inserted in your document; and the 18-pixel by 18-pixel GIF image that appears in the Object palette.

Objects are HTML files. The BODY of an object file can contain an HTML form that accepts parameters for the object (for example, the number of rows and columns in the table to be inserted). The HEAD of an object file contains JavaScript functions that process form input from the BODY and control what is added to the user's document.

Note: The simplest objects contain only the HTML to be inserted, with no BODY and HEAD tag. See "Customizing Dreamweaver" in *Using Dreamweaver* for more information.

How object files work

When a user selects an object by clicking an icon in the Object palette or by choosing an item in the Insert menu, the following chain of events occurs:

1 The object file is scanned for a FORM tag. If a form exists—and if the Show Dialog When Inserting Objects option is selected in the General preferences—Dreamweaver calls the windowDimensions() function, if defined, to determine the size of the dialog box in which to display the form. If no form exists in the object file, Dreamweaver does not display a dialog box, and step 2 is skipped.

2 If Dreamweaver displayed a dialog box in step 1, the user enters parameters for the object (such as the number of rows and columns in a table) in the dialog box and clicks OK.

3 The objectTag() function is called, and its return value is inserted into the document after the current selection (it does not replace the current selection).

The object API

There are only three custom functions in the object API, and none is required. The functions in the object API differ from the functions in the main JavaScript API in three ways:

- They are not methods of the dreamweaver, dom, or site object.

- They are significant only in the context of object files. That is, Dreamweaver automatically calls the objectTag() function if it is defined in an object file, whereas in any other extension file a function named objectTag() acts like a user-defined function—you have to call it explicitly.

- You are responsible for writing the body of each function and returning a value, if required. This is the opposite of how the functions in the main API work: those you call and pass arguments to, and Dreamweaver generates return values, if any. Here Dreamweaver calls the functions and passes arguments to them, and you generate return values, if any.

displayHelp()

Description   If this function is defined, displays a Help button below the OK and Cancel buttons in the parameters dialog box. This function is called when the user clicks the Help button.

Arguments   None.

Returns   Nothing.

Example   The following instance of displayHelp() opens in a browser window a file with instructions for using the object:

```
function displayHelp(){
  dreamweaver.browseDocument('http://people.netscape.com/andreww/dreamweaver/
objects.html');
}
```

objectTag()

Description   Inserts a string of code into the user's document.

Arguments   None.

Returns   The string to be inserted.

Example   The following instance of objectTag() inserts an OBJECT/EMBED combination for a specific ActiveX control and plugin:

```
function objectTag() {
return '\n' +
'<OBJECT CLASSID="clsid:166F1OOB-3A9R-11FB-8075444553540000" \n' +
'CODEBASE="http://www.mysite.com/product/cabs/myproduct.cab#version=1,0,0,0" \n' +
'NAME="MyProductName"> \n' +
'<PARAM NAME="SRC" VALUE=""> \n' +
'<EMBED SRC="" HEIGHT="" WIDTH="" NAME="MyProductName"> \n' +
'</OBJECT>'
```

windowDimensions()

Description Sets specific dimensions for the parameters dialog box. If this function is not defined, the window dimensions are computed automatically.

Note: Do not define this function unless you want an options dialog box larger than 640 pixels by 480 pixels.

Arguments *platform*

The value of the argument is either "macintosh" or "windows", depending on the user's platform.

Returns A string of the form "widthInPixels,heightInPixels".

The returned dimensions are smaller than the size of the entire dialog window because they do not include the area for the OK and Cancel buttons. If the returned dimensions do not accommodate all options, scroll bars appear.

Example The following instance of windowDimensions() sets the dimensions of the parameters dialog box to 648 pixels by 520 pixels:

```
function windowDimensions(platform){
  return "648,520";
}
```

Adding objects to the Object palette

Dreamweaver automatically adds any files that are inside one of the subfolders in the Configuration/Objects folder to the panel associated with the subfolder. For example, a file inside the Configuration/Objects/MyObjects folder would appear on the MyObjects panel of the Object palette.

Note: Although object files can be stored in separate folders, it's important that their file names be unique. The dom.insertObject() function, for example, looks for a specified file anywhere within the Objects folder without regard to subfolders. If a file called Button.htm exists in the Forms folder and also in the MyObjects folder, Dreamweaver cannot distinguish between them.

Each object file has an associated 18-pixel by 18-pixel GIF image that appears in the Object palette. The image file must have the same base name as the object file (for example, object.gif is associated with object.htm) to ensure that the files remain connected.

If you create a larger object image, Dreamweaver will scale it to 18 pixels by 18 pixels. If you do not create an image for your object, a broken image icon appears in the Object palette.

Adding objects to the Insert menu

Dreamweaver automatically adds to the bottom of the Insert menu any files that are inside one of the subfolders in the Configuration/Objects folder.

To control the position of an object in the Insert menu or any other menu, or to add an object to multiple menus, you can modify the menus.xml file. In Dreamweaver 3, this file controls the entire menu structure for Dreamweaver. For more information about modifying the menus.xml file, see Chapter 16, "Customizing Dreamweaver" in *Using Dreamweaver*.

Note: In previous versions of Dreamweaver, the items in the Insert menu were controlled by a file called InsertMenu.htm. This file has been superseded by menus.xml.
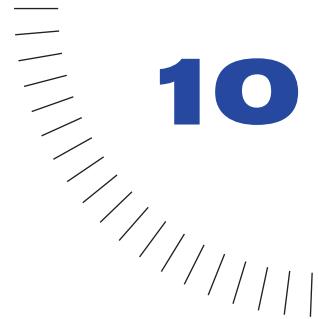
CHAPTER 10
Commands

· ·

Commands can be used to perform almost any kind of edit to the user's current document, other open documents, or to any HTML document on a local drive. Commands can insert, remove, or rearrange HTML tags and attributes, comments, and text.

Commands are HTML files. The BODY of a command file can contain an HTML form that accepts options for the command (for example, how a table should be sorted and by which column). The HEAD of a command file contains JavaScript functions that process form input from the BODY and control what edits are made to the user's document.

How commands work

When a user clicks a menu that contains a command, the following chain of events occurs:

1 Dreamweaver calls the canAcceptCommand() function, if defined, in each command file referenced in the menu to see whether this command is appropriate for the selection. If canAcceptCommand() returns FALSE, the command is dimmed in the menu.

2 The user selects a command from the menu.

3 Dreamweaver calls the receiveArguments() function, if defined, in the selected command file to give the command an opportunity to process any arguments passed from the dreamweaver.runCommand() function.

4 Dreamweaver calls the commandButtons() function, if defined, to determine which buttons appear along the right side of the options dialog box and what code should be executed when the user clicks the buttons.

5 Dreamweaver scans the command file for a FORM tag. If a form exists, Dreamweaver calls the windowDimensions() function to determine the size of the options dialog box containing the BODY elements of the file. If windowDimensions() is not defined, the size of the dialog box is determined automatically.

6 If the command file's BODY tag contains an onLoad handler, Dreamweaver executes it (regardless of whether a dialog box is displayed). If no dialog box appears, the remaining steps do not occur.

7 The user selects options for the command. Dreamweaver executes event handlers associated with the fields as the user encounters them.

8 The user clicks one of the buttons defined by commandButtons().

9 Dreamweaver executes the code associated with the button that was clicked.

10 The dialog box remains visible until one of the scripts in the command calls window.close().

The command API

There are four custom functions in the command API, and none is required. The functions in the command API differ from the functions in the main JavaScript API in three ways:

- They are not methods of the dreamweaver, dom, or site object.

- They are significant only in the context of command files. That is, Dreamweaver automatically calls the commandButtons() function if it is defined in a command or menu command file, whereas in any other extension file a function named commandButtons() acts like a user-defined function—you have to call it explicitly.

- You are responsible for writing the body of each function and returning a value, if required. This is the opposite of how the functions in the main API work: those you call and pass arguments to, and Dreamweaver generates return values, if any. Here Dreamweaver calls the functions and passes arguments to them, and you generate return values, if any.

canAcceptCommand()

Description   Determines whether the command is appropriate for the current selection.

Note: Do not define canAcceptCommand() unless it returns FALSE in at least one case. If the function is not defined, the command is assumed to be appropriate; making this assumption saves time and improves performance.

Arguments   None.

Returns   TRUE if the command is allowed; FALSE if it is not, dimming the command in the menu.

Example   The following instance of canAcceptCommand() makes the command available only when the selection is a table:

```
function canAcceptCommand(){
  var selArr=dreamweaver.getSelection();
  var selObj=dreamweaver.offsetsToNode(selArr[0],selArr[1]);
  return (selObj.nodeType == Node.ELEMENT_NODE && selObj.tagName=="TABLE");
}
```

commandButtons()

Description　Defines the buttons that should appear on the right side of the options dialog box and their behavior when clicked. If this function is not defined, no buttons appear, and the BODY of the command file expands to fill the entire dialog box.

Arguments　None.

Returns　An array containing an even number of elements. The first element is a string containing the label for the topmost button. The second element is a string of JavaScript code that defines the behavior of the topmost button when clicked. Remaining elements define additional buttons in the same manner.

Example　The following instance of commandButtons() defines three buttons: OK, Cancel, and Help.

```
function commandButtons(){
  return new Array("OK","doCommand()","Cancel","window.close()","Help","showHelp()");
}
```

receiveArguments()

Description　Processes any arguments passed from the menu item.

Arguments　*{arg1}*, *{arg2}*,...*{argN}*

If the arguments attribute is defined for a menuitem tag, the value of that attribute is passed to the receiveArguments() function as one or more arguments. The arguments attribute is useful for distinguishing between two menu items that call the same menu command.

Returns　Nothing.

windowDimensions()

Description Sets specific dimensions for the parameters dialog box to speed up display. If this function is not defined, the window dimensions are computed automatically.

Note: Do not define this function unless you want an options dialog box larger than 640 pixels by 480 pixels.

Arguments *platform*

The value of the argument is either "macintosh" or "windows", depending on the user's platform.

Returns A string of the form "widthInPixels,heightInPixels".

The returned dimensions are smaller than the size of the entire dialog window because they do not include the area for the OK and Cancel buttons. If the returned dimensions do not accommodate all options, scroll bars appear.

Example The following instance of windowDimensions() sets the dimensions of the parameters dialog box to 648 pixels by 520 pixels:

```
function windowDimensions(platform){
  return "648,520";
}
```

A simple command example

The following command converts the selected text to all lowercase characters. The command is very simple; it does not display a dialog box, so the commandButtons() function is not defined.

```
<HTML>
<HEAD>
<TITLE>Make Lower Case</TITLE>
<SCRIPT LANGUAGE="javascript">

function canAcceptCommand(){
 // Get the DOM of the current document
 var theDOM = dw.getDocumentDOM();
 // Get the offsets of the selection
 var theSel = theDOM.getSelection();
 // Get the selected node
 var theSelNode = theDOM.getSelectedNode();
 // Get the children of the selected node
 var theChildren = theSelNode.childNodes;

 // If the selection is not an insertion point, and
 // either the selection or its first child is a
 // text node, return TRUE.
 return (theSel[0] != theSel[1] && (theSelNode.nodeType == Node.TEXT_NODE ||
theChildren[0].nodeType == Node.TEXT_NODE));
}

function changeToLowerCase() {
 // Get the DOM again
 var theDOM = dw.getDocumentDOM();
 // Get the offsets of the selection
 var theSel = theDOM.getSelection();

 // Get the outerHTML of the HTML tag (the
 // entire contents of the document)
 var theDocEl = theDOM.documentElement;
 var theWholeDoc = theDocEl.outerHTML;

 // Extract the selection
 var selText = theWholeDoc.substring(theSel[0],theSel[1]);

 // Re-insert the modified selection into the document
 theDocEl.outerHTML = theWholeDoc.substring(0,theSel[0]) + selText.toLowerCase() +
theWholeDoc.substring(theSel[1]);

 // Set the selection back to where it was when you
 // started
 theDOM.setSelection(theSel[0],theSel[1]);
}

</SCRIPT>
```

```
</HEAD>

<BODY onLoad="changeToLowerCase()">

<!-- The function that does all the work in this command is called from the onLoad handler
on the BODY tag. There is no form in the BODY, so no dialog box appears. -->

</BODY>
</HTML>
```

Adding commands to the Commands menu

Dreamweaver automatically adds to the bottom of the Commands menu any files
that are inside one of the subfolders in the Configuration/Objects folder. To
prevent a command from appearing in the Commands menu, put the following
comment on the first line of the file:

```
<!-- MENU-LOCATION=NONE -->
```

In Dreamweaver 1 and 2, if you wanted to control the position or text associated
with a command, you could add it explicitly to the CommandMenu.htm file. In
Dreamweaver 3, this file has been superseded by the menus.xml file, which
controls the entire menu structure for Dreamweaver. For more information about
modifying the menus.xml file, see Chapter 16, "Customizing Dreamweaver" in
Using Dreamweaver.
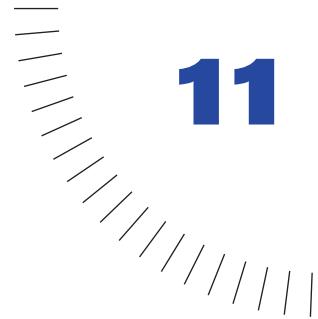
CHAPTER 11
Menu Commands

· ·

Dreamweaver 3 introduces a new type of command that makes menus more flexible and dynamic menus possible. Like regular commands, menu commands can be used to perform almost any kind of edit to the current document, other open documents, or any HTML document on a local drive. The API for menu commands expands the regular command API to accomplish several tasks related to displaying and calling the command from the menu system.

Note : Because menu commands are directly related to the menu system in Dreamweaver, you should read Chapter 16, "Customizing Dreamweaver" in *Using Dreamweaver* before continuing.

Menu commands are HTML files. The BODY of a menu command file can contain an HTML form that accepts options for the command (for example, how a table should be sorted and by which column). The HEAD of a menu command file contains JavaScript functions that process form input from the BODY and control what edits are made to the user's document.

Menu commands are stored in the Configuration/Menus folder inside the Dreamweaver application folder.

How menu commands work

When the user clicks a menu that contains a menu command, the following chain of events occurs:

1 If any menuitem tag in the menu contains the dynamic attribute, Dreamweaver calls the getDynamicContent() function in the associated menu command file to populate the menu.

2 Dreamweaver calls the canAcceptCommand() function in each menu command file referenced in the menu to check whether the command is appropriate for the selection. If canAcceptCommand() returns FALSE, the menu item is dimmed. If canAcceptCommand() returns TRUE or is not defined, Dreamweaver calls the isCommandChecked() function to determine whether to display a check mark next to the menu item. If the isCommandChecked() function is not defined, no check mark is displayed.

3 Dreamweaver calls the setMenuText() function to determine the text that should appear in the menu. If setMenuText() is not defined, Dreamweaver uses the text that is specified in the menuitem tag.

4 The user selects an item from the menu.

5 Dreamweaver calls the receiveArguments() function, if defined, in the selected menu command file to let the command process any arguments passed from the menu item.

6 Dreamweaver calls the commandButtons() function, if defined, to determine which buttons appear along the right side of the options dialog box and what code should be executed when the user clicks the buttons.

7 Dreamweaver scans the menu command file for a FORM tag. If a form exists, Dreamweaver calls the windowDimensions() function to determine the size of the options dialog box containing the BODY elements of the file. If windowDimensions() is not defined, the size of the dialog box is determined automatically.

8 If the menu command file's BODY tag contains an onLoad handler, Dreamweaver executes the code associated with the handler (regardless of whether a dialog box is displayed). If no dialog box appears, the remaining steps do not occur.

9 The user selects options in the dialog box. Dreamweaver executes event handlers associated with the fields as the user encounters them.

10 The user clicks one of the buttons defined by commandButtons().

11 Dreamweaver executes the code associated with the clicked button.

12 The dialog box remains visible until one of the scripts in the menu command calls window.close().

The menu command API

There are seven custom functions in the menu command API, and none is required. The functions in the menu command API differ from the functions in the main JavaScript API in three ways:

- They are not methods of the dreamweaver, dom, or site object.

- They are significant only in the context of menu command files. That is, Dreamweaver automatically calls the getDynamicContent() function if it is defined in a menu command file, whereas in any other extension file a function named getDynamicContent() acts like a user-defined function—you have to call it explicitly.

- You are responsible for writing the body of each function and returning a value, if required. This is the opposite of how the functions in the main API work: those you call and pass arguments to, and Dreamweaver generates return values, if any. Here Dreamweaver calls the functions and passes arguments to them, and you generate return values, if any.

canAcceptCommand()

Description   Determines whether the menu item should be active or dimmed.

Arguments   {arg1}, {arg2},...{argN}}

If the arguments attribute is defined for a menuitem tag, the value of that attribute is passed to the canAcceptCommand() function (and to the isCommandChecked(), receiveArguments(), and setMenuText() functions) as one or more arguments. The arguments attribute is useful for distinguishing between two menu items that call the same menu command.

Returns   A Boolean value indicating whether the item should be enabled.

commandButtons()

Description   Defines the buttons that should appear on the right side of the options dialog box and their behavior when clicked. If this function is not defined, no buttons appear, and the BODY of the command file expands to fill the entire dialog box.

Arguments   None.

Returns   An array containing an even number of elements. The first element is a string containing the label for the topmost button. The second element is a string of JavaScript code that defines the behavior of the topmost button when clicked. Remaining elements define additional buttons in the same manner.

Example   The following instance of commandButtons() defines three buttons: OK, Cancel, and Help.

```
function commandButtons(){
  return new Array("OK","doCommand()","Cancel","window.close()","Help","showHelp()");
}
```

getDynamicContent()

Description Retrieves the content for the dynamic portion of the menu.

Arguments *menuID*

The argument is the value of the id attribute in the menuitem tag associated with the item.

Returns An array of strings. Each string contains the name of a menu item and its unique ID, separated by a semicolon. If the function returns null, then no change is made to the menu.

Example The following instance of getDynamicContent() returns an array of four menu items (My Menu Item 1, My Menu Item 2, and so on):

```
function getDynamicContent(){
  var stringArray= new Array();
  var i=0;
  var numItems = 4;

  for (i=0; i<numItems;i++)
    stringArray[i] = new String("My Menu Item " + i + ";id=" + i);

  return stringArray;
}
```

isCommandChecked()

Description Determines whether to display a check mark next to the menu item.

Arguments *{arg1}, {arg2},...{argN}*

If the arguments attribute is defined for a menuitem tag, the value of that attribute is passed to the isCommandChecked() function (and to the canAcceptCommand(), receiveArguments(), and setMenuText() functions) as one or more arguments. The arguments attribute is useful for distinguishing between two menu items that call the same menu command.

Returns A Boolean value indicating whether a check mark should appear next to the menu item.

receiveArguments()

Description Processes any arguments passed from the menu item.

Arguments *{arg1}, {arg2},...{argN}*

If the arguments attribute is defined for a menuitem tag, the value of that attribute is passed to the receiveArguments() function (and to the canAcceptCommand(), isCommandChecked(), and setMenuText() functions) as one or more arguments. The arguments attribute is useful for distinguishing between two menu items that call the same menu command.

Returns Nothing.

setMenuText()

Description   Specifies the text that should appear in the menu.

Note: Do not use this function if you are using getDynamicContent().

Arguments   {arg1}, {arg2},...{argN}

If the arguments attribute is defined for a menuitem tag, the value of that attribute is passed to the setMenuText() function (and to the canAcceptCommand(), isCommandChecked(), and receiveArguments() functions) as one or more arguments. The arguments attribute is useful for distinguishing between two menu items that call the same menu command.

Returns   The string that should appear in the menu.

windowDimensions()

Description   Sets specific dimensions for the parameters dialog box to speed up display. If this function is not defined, the window dimensions are computed automatically.

Note: Do not define this function unless you want an options dialog box larger than 640 pixels by 480 pixels.

Arguments   *platform*

The value of the argument is either "macintosh" or "windows", depending on the user's platform.

Returns   A string of the form "widthInPixels,heightInPixels".

The returned dimensions are smaller than the size of the entire dialog window because they do not include the area for the OK and Cancel buttons. If the returned dimensions do not accommodate all options, scroll bars appear.

Example   The following instance of windowDimensions() sets the dimensions of the parameters dialog box to 648 pixels by 520 pixels:

```
function windowDimensions(platform){
  return "648,520";
}
```

A simple menu command

The following menu command is associated with two menu items: Undo and Redo. It checks the arguments attribute of the menuitem tag and performs a dw.undo() or a dw.redo() operation depending on the value of the first (and only) argument.

```
<HTML>
<HEAD>
<!-- Copyright 1999 Macromedia, Inc. All rights reserved. -->
<TITLE>Edit Clipboard</TITLE>
<SCRIPT LANGUAGE="javascript">
function receiveArguments()
{
  if (arguments.length != 1) return;

  var whatToDo = arguments[0];
  if (whatToDo == "undo")
  {
    dw.undo();
  }
  else if (whatToDo == "redo")
  {
    dw.redo();
  }
}

function canAcceptCommand()
{
  var selarray;
  if (arguments.length != 1) return false;
  var bResult = false;

  var whatToDo = arguments[0];
  if (whatToDo == "undo")
  {
    bResult = dw.canUndo();
  }
  else if (whatToDo == "redo")
  {
    bResult = dw.canRedo();
  }
  return bResult;
}

function setMenuText()
{
  if (arguments.length != 1) return "";

  var whatToDo = arguments[0];
  if (whatToDo == "undo")
    return dw.getUndoText();
```

```
  else if (whatToDo == "redo")
    return dw.getRedoText();
  else return "";
}

</SCRIPT>
</HEAD>
<BODY>
</BODY>
</HTML>
```

In this command, the receiveArguments() function both processes the arguments and executes the command, but this need not be the case. More complex menu commands might call different functions to execute the command. For example, the following code checks whether the first argument is "foo"; if it is, it calls the doOperationX() function and passes it the second argument. If the first argument is "bar", it calls the doOperationY() function and passes it the second argument. doOperationX() or doOperationY() is responsible for executing the command.

```
function receiveArguments(){
  if (arguments.length != 2) return;

  var whatToDo = arguments[0];

  if (whatToDo == "foo"){
    doOperationX(arguments[1]);
  }else if (whatToDo == "bar"){
    doOperationX(arguments[1]);
  }
}
```

A simple dynamic menu

The following menu command does two things: it generates the Preview in Browser submenu, and it launches the current file (or the selected files in the Site window) in the browser that the user selects from the submenu.

```
<HTML>
<HEAD>
<!-- Copyright 1999 Macromedia, Inc. All rights reserved. -->
<TITLE>Preview Browsers</TITLE>
<SCRIPT LANGUAGE="javascript">
<!--
 // getDynamicContent returns the contents of a dynamically generated menu.
 // returns an array of strings to be placed in the menu, with a unique
 // identifier for each item separated from the menu string by a semicolon.
 //
 // return null from this routine to indicate that you are not adding any
 // items to the menu
 function getDynamicContent(itemID)
 {
   var browsers = null;
   var PIB = null;
   var i;
   var j=0;
   var bUpdate = dw.getMenuNeedsUpdating(itemID);

   if (bUpdate)
   {
     browsers = new Array();
     PIB = dw.getBrowserList();
     // each browser pair has the name of the browser and the path that leads
     // to the application on disk. We only put the names in the menus.
     for (i=0; i<PIB.length; i=i+2)
     {
       browsers[j] = new String(PIB[i]);

       if (dw.getPrimaryBrowser() == PIB[i+1])
         browsers[j] += "\tF12";
       if (navigator.platform == "MacPPC")
       {
         if (dw.getSecondaryBrowser() == PIB[i+1])
          browsers[j] += "\t ?F12";
       }
       else
       {
         if (dw.getSecondaryBrowser() == PIB[i+1])
          browsers[j] += "\t Ctrl+F12";
       }

       browsers[j] += ";id='"+PIB[i]+"'";
       j = j+1;
     }
```

```
      dw.notifyMenuUpdated(itemID, "dw.getBrowserList()");
    }
    return browsers;
  }

function canAcceptCommand()
  {
    var bHaveDocument;

    if (dw.getFocus() == 'site')
      bHaveDocument = site.getSelection().length > 0;
    else
      bHaveDocument = dw.getDocumentDOM('document') != null;

    return bHaveDocument;
  }

function receiveArguments()
  {
    var theBrowser = arguments[0];
    if (dw.getFocus() == 'site')
      dw.browseDocument(site.getSelection(),theBrowser);
    else
      dw.browseDocument(dw.getDocumentPath('document'),theBrowser);
  }

// -->
</SCRIPT>
</HEAD>
<BODY>
</BODY>
</HTML>
```

CHAPTER 12
Property Inspectors

The Property inspector is perhaps the most familiar floating palette in the Dreamweaver interface. It is indispensable for defining, reviewing, and changing the name, size, appearance, and other attributes of the selection, as well as for launching internal and external editors for the selected element.

Dreamweaver has several built-in interfaces for the Property inspector that let you set properties for many standard HTML tags. With custom property inspector files, you can override these built-in interfaces or create new ones to inspect custom tags.

Property inspector files are HTML files that reside in the Configuration/ Inspectors folder inside the Dreamweaver application folder. The first line of a property inspector file (the line above the opening HTML tag) must be a comment in the following format:

<!-- tag:*tagNameOrKeyword*,priority:*1to10*,selection:*exactOrWithin*,hline,vline -->

where:

- *tagNameOrKeyword* is the tag to be inspected or one of the following keywords: *COMMENT* (for comments), *LOCKED* (for locked regions), or *ASP* (for ASP tags).

- *1to10* is the priority of the inspector file: 1 indicates that this inspector should be used only when no others can inspect the selection; 10 indicates that this inspector takes precedence over all others that can inspect the selection.

- *exactOrWithin* indicates whether the selection can be within the tag (within) or must exactly contain the tag (exact).

- hline (optional) indicates that a horizontal gray line should appear between the upper and lower halves of the inspector in expanded mode.

- vline (optional) indicates that a vertical gray line should appear between the tag name field and the rest of the properties in the inspector (see the image property inspector for an example).

The following comment would be appropriate for an inspector that is designed to inspect the HAPPY tag:

```
<!-- tag:HAPPY,priority:8,selection:exact,hline,vline -->
```

The BODY of a property inspector file contains an HTML form. Instead of displaying the form contents in a dialog box, however, Dreamweaver uses the form to define the input areas and layout of the inspector.

How property inspector files work

At startup, Dreamweaver reads the first line of each .htm and .html file in the Configuration/Inspectors folder, looking for the comment string that defines the type, priority, and selection type of a property inspector. Files that do not have this comment as their first line are ignored.

When the user makes a selection in Dreamweaver or moves the insertion point to a different location, the following chain of events occurs:

1 Dreamweaver looks for any inspectors that have a selection type of within.

2 If there are any within inspectors, Dreamweaver searches up the document tree from the currently selected tag to check whether there are inspectors for any of the tags that surround the selection. If—and only if—there are no within inspectors, Dreamweaver looks for any inspectors that have a selection type of exact.

3 For the first tag found that has one or more inspectors, Dreamweaver calls each inspector's canInspectSelection() function. If this function returns FALSE, Dreamweaver no longer considers the inspector a candidate for inspecting the selection.

4 If more than one potential inspector remains after calling canInspectSelection(), Dreamweaver sorts the remaining inspectors by priority.

5 If more than one potential inspector shares the same priority, Dreamweaver chooses an inspector alphabetically by name.

6 The chosen inspector appears in the Property inspector floating palette. If the property inspector file defines the displayHelp() function, a small ? icon is displayed in the upper right corner of the inspector.

7 Dreamweaver calls the inspectSelection() function to gather information about the current selection and populate the inspector's fields.

8 Event handlers attached to the fields in the property inspector interface execute as the user encounters them. (For example, you may have an onBlur event that calls setAttribute() to set an attribute to the value just entered by the user.)

The property inspector API

There are three custom functions in the property inspector API, two of which (canInspectSelection() and inspectSelection()) are required. The functions in the property inspector API differ from the functions in the main JavaScript API in three ways:

- They are not methods of the dreamweaver, dom, or site object.

- They are significant only in the context of property inspector files. That is, Dreamweaver automatically calls the canInspectSelection() function in a property inspector file, whereas in any other extension file a function named canInspectSelection() acts like a user-defined function—you have to call it explicitly.

- You are responsible for writing the body of each function and returning a value, if required. This is the opposite of how the functions in the main API work: those you call and pass arguments to, and Dreamweaver generates return values, if any. Here Dreamweaver calls the functions and passes arguments to them, and you generate return values, if any.

canInspectSelection()

Description   Determines whether the property inspector is appropriate for the current selection.

Arguments   None.

Use dom.getSelectedNode() to get the current selection as a JavaScript object.

Returns   TRUE if the inspector can inspect the current selection; FALSE if it cannot.

Example   The following instance of canInspectSelection() returns TRUE if the selection contains the CLASSID attribute and the value of that attribute is clsid:D27CDB6E-AE6D-11cf-96B8-444553540000 (the class ID for Flash Player):

```
function canInspectSelection(){
 var theDOM = dw.getDocumentDOM();
 var currSel = theDOM.getSelection();
 var theObj = theDOM.offsetsToNode(currSel[0],currSel[1]);
 return (theObj.nodeType == Node.ELEMENT_NODE && theObj.hasAttribute("classid") &&
theObj.getAttribute("classid").toLowerCase()=="clsid:D27CDB6E-AE6D-11cf-96B8-
444553540000");
}
```

displayHelp()

Description   If this function is defined, a ? icon appears in the upper right corner of the property inspector. This function is called when the user clicks the icon.

Arguments   None.

Returns   Nothing.

Example   The following instance of displayHelp() opens in a browser window a file that explains the fields in the property inspector:

```
function displayHelp(){
  dreamweaver.browseDocument('http://www.hooha.com/dw/inspectors/inspHelp.html');
}
```

inspectSelection()

Description   Refreshes the contents of the user-input fields based on the attributes of the current selection.

Arguments   *maxOrMin*

The argument is either max or min, depending on whether the inspector is in its expanded or contracted state.

Returns   Nothing.

Example   The following instance of inspectSelection() gets the value of the CONTENT attribute and uses it to populate a form field called keywords:

```
function inspectSelection(){
  var currSel = dreamweaver.getSelection();
  var theObj = dreamweaver.offsetsToNode(currSel[0],currSel[1]);
  document.forms[0].keywords.value = theObj.getAttribute("content");
}
```

A simple property inspector example

The following property inspector inspects a fictional tag called INTJ. The INTJ tag is empty (that is, it has no closing tag), so its selection type is exact. As long as the selection is exactly an INTJ tag, the inspector should show up—so the canInspectSelection() function returns TRUE every time. To have a different inspector appear depending on the value of the INTJ tag's TYPE attribute, for example, the canInspectSelection() function must check the value of the TYPE attribute to determine which inspector is the right one. (This is how the Keywords and Description inspectors work, because "keywords" and "description" are not tags but values of the META tag's NAME attribute.)

```
<!-- tag:INTJ,priority:5,selection:exact,vline,hline -->
<HTML>
<HEAD>
<TITLE>Interjection Inspector</TITLE>
<SCRIPT LANGUAGE="JavaScript">

function canInspectSelection(){
 return true;
}

function inspectSelection(){
 // Get the DOM of the current document
 var theDOM = dw.getDocumentDOM();
 // Get the selected node
 var theObj = theDOM.getSelectedNode();

 // Get the value of the TYPE attribute on the INTJ tag
 var theType = theObj.getAttribute('type');
 // Initialize a variable called typeIndex to -1. This will be
 // used to store the menu index that corresponds to
 // the value of the TYPE attribute
 var typeIndex = -1;

 // If there was a TYPE attribute
 if (theType){
  // If the value of TYPE is "jeepers", set typeIndex to 0
  if (theType.toLowerCase() == "jeepers"){
   typeIndex = 0;
  // If the value of TYPE is "jinkies", set typeIndex to 1
  }else if (theType.toLowerCase() == "jinkies"){
   typeIndex = 1;
  // If the value of TYPE is "zoinks", set typeIndex to 2
  }else if (theType.toLowerCase() == "zoinks"){
   typeIndex = 2;
  }
 }

 // If the value of the TYPE attribute was "jeepers",
 // "jinkies", or "zoinks", choose the corresponding
 // option from the pop-up menu in the interface
```

```
  if (typeIndex != -1){
    document.topLayer.document.topLayerForm.intType.selectedIndex = typeIndex;
  }
}

function setInterjectionTag(){
 // Get the DOM of the current document
 var theDOM = dw.getDocumentDOM();
 // Get the selected node
 var theObj = theDOM.getSelectedNode();

 // Get the index of the selected option in the pop-up menu
 // in the interface
 var typeIndex = document.topLayer.document.topLayerForm.intType.selectedIndex;
 // Get the value of the selected option in the pop-up menu
 // in the interface
 var theType =
document.topLayer.document.topLayerForm.intType.options[typeIndex].value;

 // Set the value of the TYPE attribute to theType
 theObj.setAttribute('type',theType);
}

</SCRIPT>
</HEAD>

<BODY>
<SPAN ID="image" STYLE="position:absolute; width:23px; height:17px; z-index:16; left:
3px; top: 2px">
<IMG SRC="interjection.gif" WIDTH="36" HEIGHT="36" NAME="interjectionImage"></
SPAN>
<SPAN ID="label" STYLE="position:absolute; width:23px; height:17px; z-index:16; left:
44px; top: 5px">Interjection</SPAN>

<!-- If your form fields are in different layers, you must create a separate form inside each
layer and reference it as shown in the inspectSelection() and setInterjectionTag() functions
above. -->
```

```
<SPAN ID="topLayer" STYLE="position:absolute; z-index:1; left: 125px; top: 3px; width:
431px; height: 32px">
<FORM NAME="topLayerForm">
<TABLE BORDER="0" CELLPADDING="0" CELLSPACING="0">
<TR>
<TD VALIGN="baseline" ALIGN="right">Type:</TD>
<TD VALIGN="baseline" ALIGN="right">
<SELECT NAME="intType" STYLE="width:86" onChange="setInterjectionTag()">
<OPTION VALUE="jeepers">Jeepers</OPTION>
<OPTION VALUE="jinkies">Jinkies</OPTION>
<OPTION VALUE="zoinks">Zoinks</OPTION>
</SELECT>
</TR>
</TABLE>
</FORM>
</SPAN>

</BODY>
</HTML>
```

CHAPTER 13
Floating Palettes

In Dreamweaver 3, you can now create any kind of floating palette or inspector without the size and layout limitations of property inspectors.

A custom property inspector should still be your first choice for setting the properties of the current selection. However, custom floating palettes offer more room and flexibility for displaying information about the entire document or multiple selections.

Dreamweaver already has several built-in floating palettes that are accessible from the Window menu; you can add your own palettes to this menu using the extensible menus feature. For more information on adding items to the menu system, see Chapter 16, "Customizing Dreamweaver," in *Using Dreamweaver*.

Floating palette files are HTML files that reside in the Configuration/Floaters folder inside the Dreamweaver application folder. The BODY of a floating palette file contains an HTML form; event handlers attached to form elements may call JavaScript code that performs arbitrary edits to the current document.

How floating palette files work

Custom floating palettes can be moved, resized, and tabbed together just like the floating palettes that are built into Dreamweaver. Custom floaters differ from built-in floaters in the following ways:

- It is not possible to display an icon in the tab of a custom floating palette; the tab always shows the contents of the floater's TITLE tag.

- Custom floating palettes display in the default gray. Setting the BGCOLOR attribute in the BODY tag has no effect.

- All custom floating palettes either appear always on top of the Document window or float behind it when inactive, depending on the setting for All Other Floaters in the Floating Palettes preferences.

Floating palette files also differ somewhat from other extensions. Unlike other extension files, Dreamweaver does not load floating palette files into memory at startup unless the floaters were visible when Dreamweaver last shut down. If the floaters were not visible when Dreamweaver last shut down, the files that define them are only loaded when referenced from one of the following functions: dreamweaver.getFloaterVisibility(), dreamweaver.setFloaterVisibility(), or dreamweaver.toggleFloater().

When one of the files inside the Configuration folder calls dw.getFloaterVisibility(*floaterName*), dw.setFloaterVisibility(*floaterName*), or dw.toggleFloater(*floaterName*), the following chain of events occurs:

1 If *floaterName* is not one of the reserved floater names, Dreamweaver searches the Configuration/Floaters folder for a file called *floaterName*.htm. (For a complete list of reserved floater names, see "dreamweaver.getFloaterVisibility()" on page 151.) If *floaterName*.htm is not found, Dreamweaver searches for *floaterName*.html. If no file is found, nothing further happens.

2 If the floating palette file is being loaded for the first time, the initialPosition() function is called, if defined, to determine the floater's default position on the screen, and the initialTabs() function is called, if defined, to determine the floater's default tab grouping.

3 The selectionChanged() and documentEdited() functions are called on the assumption that changes probably occurred while the floater was hidden.

4 When the floater is visible, the following things happen:

• When the selection changes, the selectionChanged() function is called, if defined.

• When the user makes changes to the document, the documentEdited() function is called, if defined.

• Event handlers attached to the fields in the floater interface execute as the user encounters them. (For example, a button with an onClick event handler that calls dw.getDocumentDOM().body.innerHTML='' would remove everything between the opening and closing BODY tags in the document when clicked.)

5 When the user quits Dreamweaver, the current visibility, position, and tab grouping of the floater are saved. The next time Dreamweaver starts up, it loads the floating palette files for any floaters that were visible at the last shutdown and displays the floaters in their last position and tab grouping.

The floating palette API

There are four custom functions in the floating palette API, and none is required. The functions in the floating palette API differ from the functions in the main JavaScript API in three ways:

- They are not methods of the dreamweaver, dom, or site object.

- They are significant only in the context of floating palette files. That is, Dreamweaver automatically calls the documentEdited() function if it is defined in a floating palette file, whereas in any other extension file a function named documentEdited() acts like a user-defined function—you have to call it explicitly.

- You are responsible for writing the body of each function and returning a value, if required. This is the opposite of how the functions in the main API work: those you call and pass arguments to, and Dreamweaver generates return values, if any. Here Dreamweaver calls the functions and passes arguments to them, and you generate return values, if any.

documentEdited()

Description   Called when the floater becomes visible and after the current series of edits is complete—that is, multiple edits may occur before this function is called. This function should be defined only if the floater must track edits to the document.

Note: Only define documentEdited() only if you absolutely require it, because its existence impacts performance.

Arguments   None.

Returns   Nothing.

Example   The following instance of documentEdited() scans the document for layers and updates a text field that displays the number of layers in the document:

```
function documentEdited(){
  /* create a list of all the layers in the document */
  var theDOM = dw.getDocumentDOM();
  var layersInDoc = theDOM.getElementsByTagName("layer");
  var layerCount = layersInDoc.length;

  /* update the numOfLayers field with the new layer count */
  document.theForm.numOfLayers.value = layerCount;
}
```

selectionChanged()

Description Called when the floater becomes visible, and whenever the selection changes (when focus switches to a new document, or when the insertion pointer moves to a new location in the current document). This function should be defined only if the floater must track the selection.

Note: Only define selectionChanged() if you absolutely require it, because its existence impacts performance.

Arguments None.

Returns Nothing.

Example The following instance of selectionChanged() shows a different panel (layer) in the floater depending on whether the selection is a script marker or something else:

```
function selectionChanged(){
 /* get the selected node */
 var theDOM = dw.getDocumentDOM();
 var theNode = dw.getSelectedNode();

 /* check to see if the node is a script marker */
 if (theNode.nodeType == Node.ELEMENT_NODE && theNode.tagName == "SCRIPT"){
   document.layers['blanklayer'].visibility = 'hidden';
   document.layers['scriptlayer'].visibility = 'visible';
 }else{
   document.layers['scriptlayer'].visibility = 'hidden';
   document.layers['blanklayer'].visibility = 'visible';
 }
}
```

initialPosition()

Description Determines the initial position of the floater the first time it is called. If this function is not defined, the default position is the center of the screen.

Arguments *platform*

Possible values for *platform* are "Mac" and "Win".

Returns A string of the form "leftPosInPixels,topPosInPixels".

Example The following instance of initialPosition() specifies that the first time the floater appears, it should be 420 pixels from the left and 20 pixels from the top in Windows, and 400 pixels from the left side of the screen and 20 pixels from the top of the screen on the Macintosh:

```
function initialPosition(platform){
 var initPos = "420,20";
 if (platform == "macintosh"){
   initPos = "390,20";
 }
 return initPos;
}
```

initialTabs()

Description Determines which other floaters are tabbed together with this one the first time the floater appears. If any listed floater has appeared previously, it is not included in the tab group. Thus, to assure that two custom floaters are tabbed together, each should reference the other in each initialTabs() function.

Arguments None.

Returns A string of the form "floaterName1,floaterName2,...floaterNameN".

Example The following instance of initialTabs() specifies that the first time the floater appears, it should be tabbed together with the scriptEditor floater:

```
function initialTabs(){
  return "scriptEditor";
}
```

About performance

Declaring the selectionChanged() or documentEdited() function in your custom floaters, risks adversely impacting Dreamweaver's performance. Consider that documentEdited() is called after every keystroke, and selectionChanged() is called after every press of an arrow key. It's important to test your floater against many different scenarios, using large documents (100K or more of HTML) whenever possible.

To help you avoid performance penalties, setTimeout() has been implemented as a global method in Dreamweaver 3. As in the browsers, setTimeout() takes two arguments: the JavaScript to be called, and the amount of time in milliseconds to wait before calling it.

The setTimeout() method lets you build pauses into your processing during which the user can continue interacting with the application. You must build in these pauses explicitly because the screen freezes while scripts are processing, preventing the user from performing further edits (and you from updating the interface or the floater).

The following code is from a floater that displays information about every layer in the document. It uses setTimeout() to pause for half a second after processing each layer:

```
/* create a flag that specifies whether an edit is being processed, and set it to FALSE. */
document.running = false;

/* this function called when document is edited */
function documentEdited(){
  /* create a list of all the layers to be processed */
  var dom = dw.getDocumentDOM();
  document.layers = dom.getElementsByTagName("layer");
  document.numLayers = document.layers.length;
  document.numProcessed = 0;

  /* set a timer to call processLayer(); if we didn't get
   * to finish processing the previous edit, then the timer
   * is already set. */
  if (document.running = false){
    setTimeout("processLayer()", 500);
  }

  /* set the processing flag to TRUE */
  document.running = true;
}

/* process one layer */
function processLayer(){
  /* display information for the next unprocessed layer.
     displayLayer() is a function you would write to
     perform the "magic".  */
  displayLayer(document.layers[document.numProcessed]);

  /* if there's more work to do, set a timeout to process
   * the next layer.  If we're finished, set the document.running
   * flag to FALSE. */
  document.numProcessed = document.numProcessed + 1;
  if (document.numProcessed < document.numLayers){
    setTimeout("processLayer()", 500);
  }else{
    document.running = false;
  }
}
```

A simple floating palette example

The following floating palette contains a text field that shows the contents of the selected Script marker (the yellow icon that appears in the Document window to mark the location of a script). If no Script marker is selected, a layer that contains the text (no script selected) appears.

```
<html>
<head>
<title>Script Editor</title>
<script language="JavaScript">

function selectionChanged(){
 /* get the selected node */
 var theDOM = dw.getDocumentDOM();
 var theNode = theDOM.getSelectedNode();

 /* check to see if the node is a script marker */
 if (theNode.nodeType == Node.ELEMENT_NODE && theNode.tagName == "SCRIPT"){
  document.layers['scriptlayer'].visibility = 'visible';
  document.layers['scriptlayer'].document.theForm.scriptCode.value = theNode.innerHTML;
  document.layers['blanklayer'].visibility = 'hidden';
 }else{
  document.layers['scriptlayer'].visibility = 'hidden';
  document.layers['blanklayer'].visibility = 'visible';
 }
}

/* update the document with any changes made by
   the user in the textarea */
function updateScript(){
 var theDOM = dw.getDocumentDOM();
 var theNode = dw.getSelectedNode();
 theNode.innerHTML = document.layers['scriptlayer'].document.theForm.scriptCode.value;
}

</script>
</head>

<body>
<div id="blanklayer" style="position:absolute; width:422px; height:181px; z-index:1; left:
8px; top: 11px; visibility: hidden">
<center>
<br>
<br>
<br>
<br>
<br>
(no script selected)
</center>
</div>
```

```
<div id="scriptlayer" style="position:absolute; width:422px; height:181px; z-index:1; left:
8px; top: 11px; visibility: visible">
<form name="theForm">
<textarea name="scriptCode" cols="80" rows="20" wrap="VIRTUAL"
onBlur="updateScript()"></textarea>
</form>
</div>

</body>
</html>
```

Remember that it is not sufficient to save this code in a file called scriptEditor.htm in the Configuration/Floaters folder; you must also call dw.setFloaterVisibility('scriptEditor',true), or dw.toggleFloater('scriptEditor') from somewhere in order to load the floater and make it visible. The most obvious place from which to do this is the Window menu in menus.xml file. The menuitem tag might look like this:

```
<menuitem name="Script Editor" enabled="true"
command="dw.toggleFloater('scriptEditor')"
checked="dw.getFloaterVisibility('scriptEditor')" />
```

CHAPTER 14
Behaviors

. . . .

Behaviors let nonprogrammers make their HTML pages interactive. They offer web designers an easy way to assign an actions to page elements by filling in an HTML form.

You should write behavior actions when you want to share functions with nonprogrammers, or when you want to insert the same JavaScript function repeatedly but change the parameters each time.

Note: You cannot use behaviors to insert VBScript functions directly; however, you can add a VBScript function indirectly by editing the DOM in the applyBehavior() function.

The term "behavior" refers to the combination of an event (such as onClick, onLoad, or onSubmit) and an action (for example, Check Plugin, Go to URL, Swap Image). The browser determines which HTML elements accept which events. Files listing events that each browser supports are stored in the Configuration/Behaviors/ Events folder within the Dreamweaver application folder.

Actions are HTML files. The BODY of an action file generally contains an HTML form that accepts parameters for the action (for example, parameters indicating which layers are to be shown or hidden). The HEAD of an action file contains JavaScript functions that process form input from the BODY and control the functions, arguments, and event handlers that are inserted into a user's document.

How behaviors work

When a user selects an HTML element in a Dreamweaver document and opens the Behavior inspector, the following chain of events occurs:

1 The user clicks the + button to display the Actions pop-up menu.

2 Dreamweaver calls the canAcceptBehavior() function in each action file to see whether this action is appropriate for the document or the selected element. If the return value of this function is FALSE, Dreamweaver dims the action in the Actions pop-up menu. (For example, the Control Shockwave action is dimmed when the user's document has no Shockwave movies.) If the return value is a list of events, Dreamweaver compares each event with the valid events for the currently selected HTML element and target browser until it finds a match.

3 Dreamweaver populates the Events pop-up menu with the matched event from canAcceptBehavior() at the top of the list; if no match exists, the default event for the HTML element (marked in the event file with an asterisk) becomes the top item. The remaining events in the menu are culled from the event file.

4 The user selects an action from the Action pop-up menu.

5 Dreamweaver calls the windowDimensions() function, if defined, to determine the size of the parameters dialog box. If windowDimensions() is not defined, the size is determined automatically.

6 Dreamweaver displays a dialog box containing the BODY elements of the action file. If the action file's BODY tag contains an onLoad handler, Dreamweaver executes it.

7 The user fills in the parameters for the action. Dreamweaver executes event handlers associated with the form as the user encounters them.

8 The user clicks OK.

9 Dreamweaver calls the behaviorFunction() and applyBehavior() functions in the selected action file. These functions return strings that are inserted into the user's document.

10 If the user later double-clicks the action in the Actions column, Dreamweaver reopens the parameters dialog box, executing the onLoad handler. Dreamweaver then calls the inspectBehavior() function in the selected action file, which fills in the fields with the data that the user entered previously.

Inserting multiple functions in the user's file

Actions can insert multiple functions—the main behavior function plus any number of helper functions—into the HEAD. Two or more behaviors can even share helper functions, as long as the function definition is exactly the same in each action file. One way of ensuring that shared functions are identical is to store each helper function in an external JavaScript file and insert it into the appropriate action files using <SCRIPT SRC="*externalFile*.js">.

When the user deletes a behavior, Dreamweaver will attempt to remove any unused helper functions associated with the behavior. If other behaviors are using a helper function, it will not be deleted. Because the algorithm for deleting helper functions errs on the side of caution, Dreamweaver may occasionally leave behind an unused function in the user's document.

What to do when an action requires a return value

Sometimes an event handler must have a return value (for example, onMouseOver="window.status='This is a link'; return true"). But if Dreamweaver inserts "return behaviorName(args)" into the event handler, behaviors later in the list are skipped.

To get around this limitation, set a variable called document.MM_returnValue to the desired return value within the string returned by behaviorFunction(). This setting causes Dreamweaver to insert return document.MM_returnValue at the end of the list of actions in the event handler. See the Validate Form.js file in the Configuration/ Behaviors/Actions folder within the Dreamweaver application folder for an example of the use of MM_returnValue.

The behavior API

There are eight custom functions in the behavior API, two of which (applyBehavior() and behaviorFunction()) are required. The functions in the behavior API differ from the functions in the main JavaScript API in three ways:

- They are not methods of the dreamweaver, dom, or site object.

- They are significant only in the context of behavior files. That is, Dreamweaver automatically calls the applyBehavior() function if it is defined in a behavior file, whereas in any other extension file a function named applyBehavior() acts like a user-defined function—you have to call it explicitly.

- You are responsible for writing the body of each function and returning a value, if required. This is the opposite of how the functions in the main API work: those you call and pass arguments to, and Dreamweaver generates return values, if any. Here Dreamweaver calls the functions and passes arguments to them, and you generate return values, if any.

applyBehavior()

Description

Inserts into the user's document an event handler that calls the function inserted by behaviorFunction(). This function can also perform other edits on the user's document, but it must not delete the object to which the behavior is being applied or the object that receives the action.

Arguments

uniqueName

The argument is a unique identifier among all instances of all behaviors in the user's document. Its format is *functionNameInteger*, where *functionName* is the name of the function inserted by behaviorFunction(). This argument is useful if you insert a tag into the user's document and you want to assign a unique value to its NAME attribute.

Returns

A string containing the function call to be inserted in the user's document, usually after accepting parameters from the user. If applyBehavior() determines that the user made an invalid entry, the function can return an error string instead of the function call. If the string is empty (return "";) reports no error; but if the string is non-empty and not a function call, Dreamweaver displays a dialog box with the text: Invalid Input supplied for this behavior: [the string returned from applyBehavior()]. If the return value is NULL (return;), Dreamweaver indicates that an error occurred but offers no specific information.

Note: Quotation marks within the returned string must be preceded by a backslash (\) to avoid errors reported by the JavaScript interpreter.

Example

The following instance of applyBehavior() returns a call to the function MM_openBrWindow() and passes it parameters given by the user (the height and width of the window; whether the window should have scroll bars, a toolbar, a location bar, and other features; and the URL that should open in the window):

```
function applyBehavior() {
 var i,theURL,theName,arrayIndex = 0;
 var argArray = new Array(); //use array to produce correct number of commas w/o spaces
 var checkBoxNames = new
Array("toolbar","location","status","menubar","scrollbars","resizable");

 for (i=0; i<checkBoxNames.length; i++) {
   theCheckBox = eval("document.theForm." + checkBoxNames[i]);
   if (theCheckBox.checked) argArray[arrayIndex++] = (checkBoxNames[i] + "=yes");
 }
 if (document.theForm.width.value)
   argArray[arrayIndex++] = ("width=" + document.theForm.width.value);
 if (document.theForm.height.value)
   argArray[arrayIndex++] = ("height=" + document.theForm.height.value);
 theURL = escape(document.theForm.URL.value);
 theName = document.theForm.winName.value;
 return "MM_openBrWindow('"+theURL+"','"+theName+"','"+argArray.join()+"')";
}
```

behaviorFunction()

Description   Inserts one or more functions—surrounded by <SCRIPT LANGUAGE="JavaScript"> </SCRIPT> tags, if none yet exist—into the HEAD of the user's document.

Arguments   None.

Returns   Either a string containing the JavaScript functions to be inserted in the user's document, or a string containing the names of the functions to be inserted into the user's document. This value must be exactly the same every time (it cannot depend on input from the user). The functions are inserted only once, regardless of how many times the action is applied to elements in the document.

Note: Quotation marks within the returned string must be preceded by a backslash (\) to avoid errors reported by the JavaScript interpreter.

Example   The following instance of behaviorFunction() returns a function called MM_popupMsg():

```
function behaviorFunction(){
 return ""+
 "function MM_popupMsg(theMsg) { //v1.0\n"+
 " alert(theMsg);\n"+
 "}";
}
```

The following code would be equivalent to the preceding behaviorFunction() declaration, and is the method used to declare behaviorFunction() in all of the behaviors that ship with Dreamweaver:

```
function MM_popupMsg(theMsg){ //v1.0
 alert(theMsg);
}

function behaviorFunction(){
 return "MM_popupMsg";
}
```

canAcceptBehavior()

Description   Determines whether the action is allowed for the selected HTML element and specifies the default event that should trigger the action. May also check for the existence of certain objects (such as Shockwave movies) in the user's document and disallow the action if these objects do not appear.

Arguments   *HTMLelement*

The argument is the selected HTML element.

Returns   One of the following values:

- TRUE if the action is allowed but has no preferred events.

- A list of preferred events (in descending order of preference) for this action. Specifying preferred events overrides the default event (as denoted with an asterisk in the event file) for the selected object. See steps 2 and 3 in "How behaviors work" on page 262.

- FALSE if the action is not allowed.

If canAcceptBehavior() returns FALSE, the action is dimmed in the Actions pop-up menu in the Behavior inspector.

Example   The following instance of canAcceptBehavior() returns a list of preferred events for the behavior if the document has any named images:

```
function canAcceptBehavior(){
 var theDOM = dreamweaver.getDocumentDOM();
 // Get an array of all images in the document
 var allImages = theDOM.getElementsByTagName('IMG');
 if (allImages.length > 0){
   return "onMouseOver, onClick, onMouseDown";
 }else{
   return FALSE;
 }
}
```

displayHelp()

Description   If this function is defined, a Help button appears below the OK and Cancel buttons in the parameters dialog box. This function is called when the user clicks the Help button.

Arguments   None.

Returns   Nothing.

Example   The following instance of displayHelp() opens in a browser window a file with instructions for using the behavior:

```
function displayHelp(){
  dreamweaver.browseDocument('http://www.hotwired.com/webmonkey/javascript/
code_library/wm_pos2_elmnt_dw/?tw=javascript');
}
```

deleteBehavior()

Description   Undoes any edits performed by applyBehavior().

Note: Dreamweaver automatically deletes the function declaration and the event handler associated with a behavior when the user deletes the behavior in the Behavior inspector. Thus, it is necessary to define deleteBehavior() only if the applyBehavior() function performed additional edits on the user's document (for example, if it inserted an EMBED tag).

Arguments   *applyBehaviorString*

This argument is the string that was returned earlier by the applyBehavior() function.

Returns   Nothing.

identifyBehaviorArguments()

Description   Identifies arguments from a behavior function call as nav, dep, URL, NS4.0ref, IE4.0ref, objName, or other so that URLs in behaviors can be updated if the user saves the document to another location, and so that the referenced files can be displayed in the site map and considered dependent files for the purposes of uploading to and downloading from a server.

Arguments   *theFunctionCall*

This argument is the string that was returned earlier by the applyBehavior() function.

Returns   A string containing a comma-separated list of the types of arguments in the function call. The length of the list must equal the number of arguments in the function call. Argument types will always be one of the following:

- nav specifies that the argument is a navigational URL, and therefore that it should be displayed in the site map.

- dep specifies that the argument is a dependent file URL, and therefore that it should be included with all other dependent files when a document containing this behavior is downloaded from or uploaded to a server.

- URL specifies that the argument is both a navigational URL and a dependent URL (or that it is a URL of unknown type), and therefore that it should be displayed in the site map and considered a dependent file when uploading to or downloading from a server.

- NS4.0ref specifies that the argument is a Netscape DOM object reference.

- IE4.0ref specifies that the argument is an Internet Explorer DOM object reference.

- objName specifies that the argument is a simple object name, as specified in the NAME attribute for the object. This type was added in Dreamweaver 3.

- other specifies that the argument is none of the above types.

Example   This simple example of identifyBehaviorArguments() would work for the Open Browser Window behavior action, which returns a function that always has three arguments (the URL to open, the name of the new window, and the list of window properties):

```
function identifyBehaviorArguments(fnCallStr) {
  return "URL,other,other";
}
```

A more complex version of identifyBehaviorArguments() is necessary for behavior functions that have a variable number of arguments (such as Show/Hide Layer). This version of identifyBehaviorArguments() relies on the fact that there is a minimum number of arguments, and additional arguments always come in multiples of the minimum number. In other words, a function with a minimum number of arguments of 4 may have 4, 8, or 12 arguments, but it will never have 10 arguments.

```
function identifyBehaviorArguments(fnCallStr) {
  var listOfArgTypes;
  var itemArray = dreamweaver.getTokens(fnCallStr, '(),');

  //The array of items returned by getTokens() includes the function name,
  //so the number of *arguments* in the array is the length of the array
  //minus one. Divide by 4 to get the number of groups of arguments.
  var numArgGroups = ((itemArray.length - 1)/4);
  //For each group of arguments
  for (i=0; i < numArgGroups; i++){

    //Add a comma and "NS4.0ref,IE4.0ref,other,dep" (because this
    //hypothetical behavior function has a minimum of four arguments:
    //the Netscape object reference, the IE object reference, a dependent
    //URL, and perhaps a property value such as "show" or "hide") to the
    //existing list of argument types, or if no list yet exists, add only
    //"NS4.0ref,IE4.0ref,other,dep"
    var listOfArgTypes += ((listOfArgTypes)?",":"") +
    "NS4.0ref,IE4.0ref,other,dep";
  }
}
```

inspectBehavior()

Description   Inspects the function call for a previously applied behavior in the user's document and sets the values of the options in the parameters dialog box accordingly. If inspectBehavior() is not defined, the default option values appear.

Note: inspectBehavior() must rely solely on information passed to it via the *applyBehaviorString* argument. Do not attempt to obtain other information about the user's document (for example, using dreamweaver.getDocumentDOM()) within this function.

Arguments   *applyBehaviorString*

This argument is the string that was returned earlier by the applyBehavior() function.

Returns   Nothing.

Example   The following instance of inspectBehavior(), taken from Display Status Message.htm, fills in the Message field in the parameters form with the message that the user selected when the behavior was originally applied:

```
function inspectBehavior(msgStr){
  var startStr = msgStr.indexOf('"') + 1;
  var endStr = msgStr.lastIndexOf('"');
  if (startStr > 0 && endStr > startStr) {
    document.theForm.message.value = unescQuotes(msgStr.substring(startStr,endStr));
  }
}
```

Note: For more information about the unescQuotes() function, see the string.js file in the Configuration/Shared/macromedia/scripts/cmn folder.

windowDimensions()

Description   Sets specific dimensions for the parameters dialog box to speed up display. If this function is not defined, the window dimensions are computed automatically.

Note: Do not define this function unless you want an options dialog box larger than 640 pixels by 480 pixels.

Arguments   *platform*

The value of the argument is either "macintosh" or "windows", depending on the user's platform.

Returns   A string of the form "widthInPixels,heightInPixels".

The returned dimensions are smaller than the size of the entire dialog window because they do not include the area for the OK and Cancel buttons. If the returned dimensions do not accommodate all options, scroll bars appear.

Example   The following instance of windowDimensions() sets the dimensions of the parameters dialog box to 648 pixels by 520 pixels:

```
function windowDimensions(platform){
  return "648,520";
}
```

A simple behavior example

To understand better how behaviors work and how you can create one, it's helpful to look at an example. The Configuration/Behaviors/Actions folder inside the Dreamweaver application folder is full of examples; however, many of these are likely to be too complex a starting point for all but the most advanced developers. The simplest action file to start with is Call JavaScript.htm (along with its counterpart, Call JavaScript.js, which contains all the JavaScript functions).

The following code also presents a relatively simple example. It checks the brand of the browser and goes to one page if the brand is Netscape, and another if the brand is Microsoft Internet Explorer. This code could easily be expanded to check for other brands—such as Opera and WebTV—and modified to perform other actions besides going to URLs.

```html
<html>
<head>
<title>behavior "Check Browser Brand"</title>
<meta http-equiv="Content-Type" content="text/html">
<script language="JavaScript">

// The function that will be inserted into the
// HEAD of the user's document
function checkBrowserBrand(netscapeURL,explorerURL) {
  if (navigator.appName == "Netscape") {
    if (netscapeURL) location.href = netscapeURL;
  }else if (navigator.appName == "Microsoft Internet Explorer") {
    if (explorerURL) location.href = explorerURL;
  }
}

//****************** API **********************

function canAcceptBehavior(){
  return true;
}

// Return the name of the function to be inserted into
// the HEAD of the user's document
function behaviorFunction(){
    return "checkBrowserBrand";
}

// Create the function call that will be inserted
// with the event handler
function applyBehavior() {
  var nsURL = escape(document.theForm.nsURL.value);
  var ieURL = escape(document.theForm.ieURL.value);
  if (nsURL && ieURL) {
    return "checkBrowserBrand(\"" + nsURL + "\",\"" + ieURL + "\")";
  }else{
    return "Please enter URLs in both fields."
  }
}

// Extract the arguments from the function call
// in the event handler and repopulate the
// parameters form
function inspectBehavior(fnCall){
  var argArray = getTokens(fnCall, "()',");
  var nsURL = unescape(argArray[1]);
```

```
  var ieURL = unescape(argArray[2]);
  document.theForm.nsURL.value = nsURL;
  document.theForm.ieURL.value = ieURL;
}

//***************** LOCAL FUNCTIONS  *******************

// Put the cursor in the first text field
// and select the contents, if any
function initializeUI(){
  document.theForm.nsURL.focus();
  document.theForm.nsURL.select();
}

// Let the user browse to the Navigator and
// IE URLs
function browseForURLs(whichButton){
  var theURL = dreamweaver.browseForFileURL();
  if (whichButton == "nsURL"){
    document.theForm.nsURL.value = theURL;
  }else{
    document.theForm.ieURL.value = theURL;
  }
}

//*************** END OF JAVASCRIPT ****************
</script>
</head>
<body>
<form method="post" action="" name="theForm">
<table border="0" cellpadding="8">
<tr>
<td nowrap="nowrap">  Go to this URL if the browser is Netscape
Navigator:<br>
<input type="text" name="nsURL" size="50" value="">  
<input type="button" name="nsBrowse" value="Browse..."
onClick="browseForURLs('nsURL')"></td>
</tr>
<tr>
<td nowrap="nowrap">  Go to this URL is the browser is Microsoft Internet
Explorer:<br>
<input type="text" name="ieURL" size="50" value="">  
<input type="button" name="ieBrowse" value="Browse..."
onClick="browseForURLs('ieURL')"></td>
</tr>
</table>
</form>
</body>
</html>
```

CHAPTER 15
Data Translators

Data translators translate specialized markup—such as server-side includes, conditional JavaScript statements, or other non-HTML code such as ePerl, PHP3, JSP, CFML, or ASP—into HTML code that can be read and displayed by Dreamweaver. In Dreamweaver 3 you can translate attributes within tags as well as entire tags or blocks of code.

Translated tags or blocks of code must be enclosed in locked regions to preserve the original markup. Translated attributes do not require locks, which makes inspecting the tags that contain them a simpler process.

All data translators—block/tag or attribute—are HTML files. The HEAD of a data translator contains two or more JavaScript functions: one that specifies what the translator does and the files it acts on, one that performs the actual translation, and any number of supporting functions. The BODY of a data translator is empty.

Data translation—especially for entire tags or blocks of code—might involve complex operations that either cannot be done with JavaScript or that would be more effeciently done with C. If you are familiar with C or C++, you should also read "C-Level Extensibility" on page 209.

How data translators work

Dreamweaver handles all translator files the same way, regardless of whether they translate entire tags or only attributes. At startup, Dreamweaver reads all the files in the Configuration/Translators folder and calls the getTranslatorInfo() function to obtain information about the translator. Dreamweaver ignores any file in which getTranslatorInfo() does not exist or contains an error that causes it to be undefined.

Note : To prevent JavaScript errors from interfering with startup, errors in any translator file are reported only after all translators are loaded. For more information on debugging translators, see "Finding bugs in your translator" on page 293.

Dreamweaver also calls the translateMarkup() function in all applicable translator files (as specified in the Translation preferences) whenever the user may have added new—or changed existing—content that needs translation. Dreamweaver therefore calls translateMarkup() when the user:

- Opens a file in Dreamweaver.

- Switches back to the Document window after making changes in the HTML inspector.

- Changes the properties of an object in the current document.

- Inserts an object (using either the Object palette or the Insert menu).

- Refreshes the current document after making changes to it in another application.

- Applies a template to the document.

- Pastes or drags content into or within the Document window.

- Saves changes to a dependent file.

- Invokes a command, behavior, property inspector, or other extension that sets the innerHTML or outerHTML property of any tag object or the data property of any comment object.

- Chooses File > Convert > 3.0 Browser Compatible.

- Chooses Modify > Layout Mode > Convert Tables to Layers.

- Chooses Modify > Layout Mode > Convert Layers to Tables.

- Chooses Modify > Translate > *translatorName*.

- Changes a tag or attribute in the Quick Tag Editor and presses Tab or Enter.

The data translator API

There are only two custom functions in the data translator API, but both are required. The functions in the data translator API differ from the functions in the main JavaScript API in three ways:

- They are not methods of the dreamweaver, dom, or site object.

- They are significant only in the context of data translator files. That is, Dreamweaver automatically calls the translateMarkup() function if it is defined in a data translator file, whereas in any other extension file a function named translateMarkup() acts like a user-defined function—you have to call it explicitly.

- You are responsible for writing the body of each function and returning a value, if required. This is the opposite of how the functions in the main API work: those you call and pass arguments to, and Dreamweaver generates return values, if any. Here Dreamweaver calls the functions and passes arguments to them, and you generate return values, if any.

getTranslatorInfo()

Description Provides information about the translator and the files it can act upon.

Arguments None.

Returns An array of strings. The elements of the array must appear in the following order:

- *translatorClass* uniquely identifies the translator. This string must begin with a letter and can contain only alphanumeric characters, hyphens (-), and underscores (_).

- *title* describes the translator in no more than 40 characters. This string appears in the Modify > Translate menu as well as in the Translation preferences.

- *nExtensions* specifies the number of file extensions to follow. If *nExtensions* is 0, the translator can run on any file.

- *extension* specifies a file extension (for example, "htm" or "SHTML") that this translator can work with. This string is case insensitive and should not contain a leading period. The array should contain the same number of *extension* elements as are specified in *nExtensions*.

- *nRegExps* specifies the number of regular expressions that will follow. If *nRegExps* is 0, *runDefault* is the next element in the array.

- *regExps* specifies a regular expression to check for. The array should contain the same number of *regExps* elements as are specified in *nRegExps*, and at least one of the *regExps* must match a piece of the document's HTML source before the translator can act on a file.

- *runDefault* specifies the default preference for running this translator. Possible values are "allFiles", "noFiles", "byExtension", and "byExpression". If you set *runDefault* to "byExtension" but do not specify any extensions (see *extension*, above), the effect is the same as setting "allFiles". If you set *runDefault* to "byExpression" but do not specify any expressions (see *regExps*, above), the effect is the same as setting "noFiles". Whatever value you set for *runDefault* can be overridden by the user in the Preferences dialog box.

Example   The following instance of getTranslatorInfo() gives information about a translator for server-side includes:

```
function getTranslatorInfo(){
 var transArray = new Array(11);

 transArray[0] = "SSI";
 transArray[1] = "Server-Side Includes";
 transArray[2] = "4";
 transArray[3] = "htm";
 transArray[4] = "stm";
 transArray[5] = "html";
 transArray[6] = "shtml";
 transArray[7] = "2";
 transArray[8] = "<!--#include file";
 transArray[9] = "<!--#include virtual";
 transArray[10] = "byExtension";

 return transArray;
}
```

translateMarkup()

Description   Performs the translation.

Arguments   *docName, siteRoot, docContent*

- The first argument is a string containing the file:// URL for the document to be translated.

- The second argument is a string containing the file:// URL for the root of the site that contains the document to be translated. If the document is outside a site, this string might be empty.

- The third argument is a string containing the contents of the document.

Returns   A string containing the translated document.

Example   The following instance of translateMarkup() calls the C function translateASP(), which is contained in a DLL (Windows) or code library (Macintosh) called ASPTrans:

```
function translateMarkup(docName, siteRoot, docContent){
  var translatedString = "";
  if (docContent.length > 0){
    translatedString = ASPTrans.translateASP(docName, siteRoot, docContent);
  }
  return translatedString;
}
```

For an all-JavaScript example, see "A simple attribute translator example" on page 279 or "A simple block/tag translator example" on page 285.

Determining what kind of translator to use

All translators are the same to a certain extent: they must contain the getTranslatorInfo() and translateMarkup() functions, and they must reside in the Configuration/Translators folder. They differ, however, in the kind of code that they insert into the user's document, and in how that code must be inspected.

- To translate small pieces of server markup that determine attribute values or that conditionally add attributes to a standard HTML tag, write an attribute translator. Standard HTML tags that contain translated attributes can be inspected with the property inspectors that are built into Dreamweaver. It is not necessary to write a custom property inspector. See "Adding a translated attribute to a tag" on page 278.

- To translate an entire tag (for example, a server-side include) or a block of code (for example, JavaScript, Cold Fusion, PHP, or other scripting), write a block/tag translator. The code generated by a block/tag translator cannot be inspected with the property inspectors that are built into Dreamweaver. You must write a custom inspector for the translated content if you want users to be able to change the properties of the original code. See "Locking translated tags or blocks of code" on page 284.

Adding a translated attribute to a tag

Attribute translation relies heavily on the ability of the parser in Dreamweaver 3 to ignore server markup. Dreamweaver 3 already ignores the most common kinds of server markup (including ASP, CFML, and PHP) by default; if you are using server markup that has different start and end markers, you must modify the third-party tag database to ensure that your translator works properly. For more information on modifying the third-party tag database, see Chapter 16, "Customizing Dreamweaver," in *Using Dreamweaver*.

With Dreamweaver handling the preservation of the original server markup, the translator's task is to generate a valid attribute value that can be viewed in the Document window. (Hence, if you are using server markup only for attributes that do not have a user-visible effect, you do not need a translator.)

The translator creates an attribute value that has a visible effect in the Document window by adding a special attribute, mmTranslatedValue, to the tag containing the server markup. The mmTranslatedValue attribute and its value are not visible in the HTML inspector, however, nor are they saved with the document.

The mmTranslatedValue attribute must be unique within the tag. If it is likely that your translator will need to translate more than one attribute in a single tag, you must add a routine in the translator that appends numbers to mmTranslatedValue (for example, mmTranslatedValue1, mmTranslatedValue2, and so on).

The value of the mmTranslatedValue attribute must be a URL-encoded string containing at least one valid attribute/value pair. This means that mmTranslatedValue="src=%22open.jpg%22" is a valid translation for both src="<? if (dayType == weekday) then open.jpg else closed.jpg" ?> and <? if (dayType == weekday) then src="open.jpg" else src="closed.jpg" ?>. mmTranslatedValue="%22open.jpg%22" is not valid for either example because it contains only the value, not the attribute.

Translating more than one attribute at a time

The mmTranslatedValue can contain more than one valid attribute/value pair. Consider the following untranslated code:

```
<img <? if (dayType==weekday) then src="open.jpg" width="320" height="100" else
src="closed.jpg" width="100" height="320" ?> alt="We're open 24 hours a day from
12:01am Monday until 11:59pm Friday">
```

The translated markup might look like this:

```
<img <? if (dayType==weekday) then src="open.jpg" width="320" height="100" else
src="closed.jpg" width="100" height="320" ?>
mmTranslatedValue="src=%22open.jpg%22 width=%22320%22 height=%22100%22"
alt="We're open 24 hours a day from 12:01am Monday until 11:59pm Friday">
```

Notice that the spaces between the attribute/value pairs in the mmTranslatedValue are not encoded. Because Dreamweaver looks for these spaces when it attempts to render the translated value, each attribute/value pair in the mmTranslatedValue must be encoded separately and then pieced back together to form the full mmTranslatedValue. For an example of how to do this, see "A simple attribute translator example" on page 279.

A simple attribute translator example

To understand attribute translation a little better, it's helpful to look at an example. The following translator is handles "Pound Conditional" (Poco) markup, a made-up syntax that's somewhat similar to ASP or PHP. The first step in making this translator work properly is to create a tagspec for Poco markup; this will prevent Dreamweaver from parsing the untranlsated Poco statements.

The tagspec for Poco markup looks like this:

```
<tagspec tag_name="poco" start_string="<#" end_string="#>"
detect_in_attribute="true" icon="poco.gif" icon_width="17"
icon_height="15"></tagspec>
```

The poco.xml file containing this tagspec is stored in the Configuration/
ThirdPartyTags folder along with the icon for Poco tags.

```html
<html>
<head>
<title>Conditional Translator</title>
<meta http-equiv="Content-Type" content="text/html; charset="">
<script language="JavaScript">

/*************************************************************
 * This translator handles the following statement syntaxes:    *
 * <# if (condition) then foo else bar #>                       *
 * <# if (condition) then att="foo" else att="bar" #>          *
 * <# if (condition) then att1="foo" att2="jinkies"            *
 * att3="jeepers" else att1="bar" att2="zoinks" #>            *
 *                                                             *
 * It does not handle statements with no else clause.          *
 *************************************************************/

var count = 1;

function translateMarkup(docNameStr, siteRootStr, inStr){
  var count = 1;        // Counter to ensure unique mmTranslatedValues
  var outStr = inStr;   // String that will be manipulated
  var spacer = "";      // String to manage space between encoded attributes
  var start = inStr.indexOf('<# if'); // First instance of Pound Conditional code

  /* Declared but not initalized. */
  var attAndValue;      // Boolean indicating whether the attribute is part of
                        // the conditional statement
  var trueStart;        // The beginning of the true case
  var falseStart;       // The beginning of the false case
  var trueValue;        // The HTML that would render in the true case
  var attName;          // The name of the attribute that is being'
                        // set conditionally.
  var equalSign;        // The position of the equal sign just to the
                        // left of the <#, if there is one
  var transAtt;         // The entire translated attribute
  var transValue;       // The value that must be URL-encoded
  var back3FromStart;   // Three characters back from the start position
                        // (used to find equal sign to the left of <#
  var tokens;           // An array of all the attributes set in the true case
  var end;              // The end of the current conditional statement.


  // As long as there's still a <# conditional that hasn't been translated
  while (start != -1){
    back3FromStart = start-3;
    end = outStr.indexOf(' #>',start);
    equalSign = outStr.indexOf('="<# if',back3FromStart);
    attAndValue = (equalSign != -1)?false:true;
    trueStart = outStr.indexOf('then', start);
```

```
falseStart = outStr.indexOf(' else', start);
trueValue = outStr.substring(trueStart+5, falseStart);
tokens = dreamweaver.getTokens(trueValue,' ');


// If attAndValue is false, find out what attribute you're
// translating by backing up from the equal sign to the
// first space. The substring between the space and the
// equal sign is the attribute.
if (!attAndValue){
  for (var i=equalSign; i > 0; i--){
    if (outStr.charAt(i) == " "){
      attName = outStr.substring(i+1,equalSign);
      break;
    }
  }
  transValue = attName + '="' + trueValue + '"';
  transAtt = ' mmTranslatedValue' + count + '="' + escape(transValue) + '"';
  outStr = outStr.substring(0,end+4) + transAtt + outStr.substring(end+4);


// If attAndValue is true, and tokens is greater than
// 1, then trueValue is a series of attribute/value
// pairs, not just one. In that case, each attribute/value
// pair must be encoded separately and then added back
// together to make the translated value.
}else if (tokens.length > 1){
  transAtt = ' mmTranslatedValue' + count + '="'
  for (var j=0; j < tokens.length; j++){
    tokens[j] = escape(tokens[j]);
    if (j>0){
      spacer=" ";
    }
    transAtt += spacer + tokens[j];
  }
  transAtt += '"';
  outStr = outStr.substring(0,end+3) + transAtt + outStr.substring(end+3)

// If attAndValue is true and tokens is not greater
// than 1, then trueValue is a single attribute/value pair.
// This is the simplest case, where all that is necessary is
// to encode trueValue.
}else{
  transValue = trueValue;
  transAtt = ' mmTranslatedValue' + count + '="' + escape(transValue) + '"';
  outStr = outStr.substring(0,end+3) + transAtt + outStr.substring(end+3);
}

// Increment the counter so that the next instance
// of mmTranslatedValue will have a unique name, and
// then find the next <# conditional in the code.
count++;
```

```
        start = outStr.indexOf('<# if',end);

  }

  // Return the translated string.
  return outStr
}

function getTranslatorInfo(){
  returnArray = new Array(7);

  returnArray[0] = "Pound_Conditional";//The translatorClass
  returnArray[1] = "Pound Conditional Translator";//The title
  returnArray[2] = "2"; //The number of extensions
  returnArray[3] = "html"; //The first extension
  returnArray[4] = "htm";//The second extension
  returnArray[5] = "1";//  The number of expressions
  returnArray[6] = "<#";//The first expression


  return returnArray
}

</script>
</head>

<body>
</body>
</html>
```

Inspecting translated attributes

When server markup is used to specify a single attribute, and the attribute is represented in a Property inspector, Dreamweaver displays the server markup in the Property inspector.
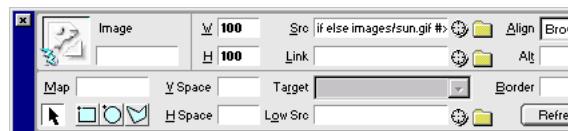
The markup appears whether or not a translator is associated with it. If a translator is associated with the markup, a lightning bolt appears over the icon in the inspector. The translator runs whenever the user edits the server markup shown in the inspector.

Note: The lightning bolt icon does not appear when text or table cells, rows, or columns are selected. Translation still occurs if the user edits server markup in the inspector and a translator exists that handles that type of markup.

When server markup is used to control more than one attribute in a tag, the server markup does not show up in the Property inspector. However, the lightning bolt signals the user that translated markup exists for the selected element.
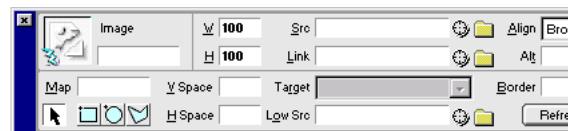
The fields in the inspector are still editable; users can enter values for attributes that may be controlled by server markup, resulting in duplicate attributes. If both a translated value and a regular value are set for a particular attribute, Dreamweaver displays the translated value in the Document window. You must decide whether your translator will look for duplicate attributes and remove them.

Locking translated tags or blocks of code

In most cases, you want a translator to change markup so that Dreamweaver can display it, but you want the original markup—not the changes—to be saved. To address this need, Dreamweaver provides special XML tags in which to wrap translated content and to refer to the original code. The syntax of the XML tags is as follows:

```
<MM:BeginLock translatorClass="translatorClass" type="tagNameOrType"
depFiles="dependentFilesList" orig="encodedOrignalMarkup">
Translated content
<MM:EndLock>
```

where:

translatorClass is the unique identifier for the translator (the first string in the array returned by getTranslatorInfo()).

tagNameOrType is a string identifying the type of markup (or the tag name associated with the markup) contained in the lock. The string can contain only alphanumeric, hyphen (-), or underscore (_) characters. You can check this value in the canInspectSelection() function of a custom property inspector to determine if the inspector is the right one for the content. For more information, see "Creating property inspectors for locked content" on page 289. Locked content cannot be inspected by any of Dreamweaver's built-in property inspectors. For example, specifying type="IMG" does not make the Image inspector appear.

dependentFilesList is a string containing a comma-separated list of files on which the locked markup depends. Files are referenced from the hard drive (for example, C:\sites\avocado8\copyright.html in Windows or MyHD:sites:avocado8:copyright.html on Macintosh). If the user updates one of the files in the *dependentFilesList*, Dreamweaver automatically retranslates the content in the document containing the list.

encodedOriginalMarkup is a string containing the original, untranslated markup, encoded using a small subset of URL encoding (use %22 for ", %3C for <, %3E for >, and %25 for %). The quickest way to URL encode a string is to use the escape() method. For example, if myString equals '<img src="foo.gif">', escape(myString) returns %3Cimg%20src=%22foo.gif%22%3E.

The following example shows the locked portion of code that might be generated from the translation of the server-side include `<!--#include virtual="/footer.html" -->`:

```
<MM:BeginLock translatorClass="MM_SSI" type="ssi"
depFiles="C:\sites\webdev\footer.html" orig="%3C!--#include virtual=%22/
footer.html%22%20--%3E">
<!-- begin footer -->
<CENTER>
<HR SIZE=1 NOSHADE WIDTH=100%>

<BR>

[<A TARGET="_top" HREF="/">home</A>]
[<A TARGET="_top" HREF="/products/">products</A>]
[<A TARGET="_top" HREF="/services/">services</A>]
[<A TARGET="_top" HREF="/support/">support</A>]
[<A TARGET="_top" HREF="/company/">about us</A>]
[<A TARGET="_top" HREF="/help/">help</A>]
</CENTER>
<!-- end footer -->
<MM:EndLock>
```

A simple block/tag translator example

To understand translation a little better, it's helpful to look at a translator that is written entirely in JavaScript (that is, one that does not rely on a C library for any functionality). The following translator would be more efficient if written in C, but the JavaScript version is simpler—which makes it perfect for demonstrating how translators work.

Like most translators, this one is designed to mimic server behavior. Assume that your web server is configured to replace the KENT tag with a different picture of an engineer depending on the day of the week, the time of day, and the user's platform. The translator does the same thing, only locally.

```
<html>
<head>
<title>Kent Tag Translator</title>
<meta http-equiv="Content-Type" content="text/html; charset="">
<script language="JavaScript">
/*********************************************************
 * The getTranslatorInfo() function provides information about the  *
 * translator, including its class and name, the types of documents *
 * that are likely to contain the markup to be translated, the      *
 * regular expressions that a document containing the markup to      *
 * be translated would match, and the default Translation           *
 * preference (whether the translator should run on all files, no   *
 * files, in files with the specified extensions, or in files matching *
 * the specified expressions).                                      *
 *********************************************************/
function getTranslatorInfo(){
  //Create a new array with 6 slots in it
  returnArray = new Array(6);

  returnArray[0] = "DREAMWEAVER_TEAM"    // The translatorClass
  returnArray[1] = "Kent Tags"           // The title
  returnArray[2] = "0"                   // The number of extensions
  returnArray[3] = "1"                   // The number of expressions
  returnArray[4] = "<kent"               // Expression
  returnArray[5] = "byExpression"        // Default Translation preference
  return returnArray;
}


/*********************************************************
 * The translateMarkup() function performs the actual translation.  *
 * In this translator, the translateMarkup() function is written    *
 * entirely in JavaScript (that is, it does not rely on a C library) -- *
 * and it's also extremely inefficient. It's a simple example, however, *
 * which is good for learning.                                      *
 *********************************************************/
function translateMarkup(docNameStr, siteRootStr, inStr){
  var outStr = "";                         // The string to be returned after translation
  var start = inStr.indexOf('<kent>');     // The first position of the KENT tag
                                           // in the document.
  var replCode = replaceKentTag();         // Calls the replaceKentTag() function
                                           // to get the code that will replace KENT.

  //If the document does not contain any content, terminate the translation.
  if ( inStr.length <= 0 ){
    return "";
  }
```

```
      // As long as start, which is equal to the location in inStr of the
      // KENT tag, is not equal to -1 (that is, as long as there is another
      // KENT tag in the document)
      while (start != -1){
        // Copy everything up to the start of the KENT tag.
        // This is very important, as translators should never change
        // anything other than the markup that is to be translated.
        outStr = inStr.substring(0, start);

        // Replace the KENT tag with the translated HTML, wrapped in special
        // locking tags. For more information on the replacement operation, see
        // the comments in the replaceKentTag() function.
        outStr = outStr + replCode;

        // Copy everything after the KENT tag.
        outStr = outStr + inStr.substring(start+6);

        // Use the string you just created for the next trip through
        // the document. This is the most inefficient part of all.
        inStr = outStr;
        start = inStr.indexOf('<kent>');

      }
      // When there are no more KENT tags in the document, return outStr.
      return outStr;
}

/***********************************************************
 * The replaceKentTag() function assembles the HTML that will     *
 * replace the KENT tag and the special locking tags that will    *
 * surround the HTML. It calls the getImage() function to         *
 * determine the SRC of the IMG tag.                              *
 ***********************************************************/
function replaceKentTag(){
  // The image to display.
  var image = getImage();
  // The location of the image on the local disk.
  var depFiles = dreamweaver.getSiteRoot() + image;
  // The IMG tag that will be inserted between the lock tags.
  var imgTag = '<IMG SRC="/' + image + '" WIDTH="320" HEIGHT="240" ALT="Kent">\n';
  // The first part of the opening lock tag. The remainder of the tag is assembled below.
  var start = '<MM:BeginLock translatorClass="DREAMWEAVER_TEAM" type="kent"';
  // The closing lock tag.
  var end = '<MM:EndLock>';

  //Assemble the lock tags and the replacement HTML.
  var replCode = start + ' depFiles="' + depFiles + '"';
  replCode = replCode + ' orig="%3Ckent%3E">\n';
  replCode = replCode + imgTag;
  replCode = replCode + end;

  return replCode;
```

```
}

/**********************************************************
 * The getImage() function determines which image to display    *
 * based on the day of the week, the time of day and the        *
 * user's platform. The day and time are figured based on UTC   *
 * time (Greenwich Mean Time) minus 8 hours, which gives        *
 * Pacific Standard Time (PST). No allowance is made for Daylight *
 * Savings Time in this routine.                                *
 **********************************************************/
function getImage(){
  var today = new Date();              // Today's date & time.
  var day = today.getUTCDay();         // The day of the week in the GMT time zone.
                                       // 0=Sunday, 1=Monday, and so on.
  var hour = today.getUTCHours();      // The current hour in GMT, based on the
                                       // 24-hour clock.
  var SFhour = hour - 8;               // The time in San Francisco, based on the
                                       // 24-hour clock.
  var platform = navigator.platform;   // The user's platform. All Windows machines
                                       // are identified by Dreamweaver as "Win32",
                                       // all Macs as "MacPPC".
  var imageRef;                        // The image reference to be returned.
  // If SFhour is negative, you have two adjustments to make.
  // First, you have to subtract one from the day count because it is already the wee
  // hours of the next day in GMT. Second, you have to add SFhour to 24 to
  // give a valid hour in the 24-hour clock.
  if (SFhour < 0){
    day = day - 1;
      // If setting the day count back one would make it negative, it's Saturday,
      // so set the count to 6.
      if (day < 0){
        day = 6;
    }
    SFhour = SFhour + 24;
  }

  // Now determine which photo to show based on whether it's a workday or a
  // weekend; what time it is; and, if it's a time and day when Kent is
  // working, what platform the user is on.

  //If it's not Sunday
  if (day != 0){
    //And it's between 10am and noon, inclusive
    if (SFhour >= 10 && SFhour <= 12){
      imageRef = "images/kent_tiredAndIrritated.jpg";
     //Or else it's between 1pm and 3pm, inclusive
    }else if (SFhour >= 13 && SFhour <= 15){
      imageRef =  "images/kent_hungry.jpg";
    //Or else it's between 4pm and 5pm, inclusive
    }else if (SFhour >= 16 && SFhour <= 17){
      //If user is on Mac, show Kent working on Mac
      if (platform == "MacPPC"){
```

```
         imageRef = "images/kent_gettingStartedOnMac.jpg";
      //If user is on Win, show Kent working on Win
      }else{
         imageRef = "images/kent_gettingStartedOnWin.jpg";
      }
   //Or else it's after 6pm but before the stroke of midnight
   }else if (SFhour >= 18){
      //If it's Saturday
      if (day == 6){
         imageRef = "images/kent_dancing.jpg";
      //If it's not Saturday, check the user's platform
      }else if (platform == "MacPPC"){
         imageRef = "images/kent_hardAtWorkOnMac.jpg";
      }else{
         imageRef = "images/kent_hardAtWorkOnWin.jpg";
      }
   }else{
      imageRef = "images/kent_sleeping.jpg";
   }
//If it's after midnight and before 10am, or anytime on Sunday
}else{
   imageRef = "images/kent_sleeping.jpg";
}

   return imageRef;
}


</script>
</head>

<body>
</body>
</html>
```

Creating property inspectors for locked content

Once you've created a translator, you'll need to create a property inspector for the content so that the user can change its properties (for example, the file to be included, or one of the conditions in a conditional statement). Inspecting translated content is a unique problem for several reasons:

- The user may want to change the properties of the translated content, and those changes must be reflected in the untranslated content.

- The DOM contains the translated content (that is, the lock tags and the tags they surround are nodes in the DOM), but the outerHTML property of the documentElement and the dreamweaver.getSelection() and dreamweaver.nodeToOffsets() functions act on the untranslated source.

- The tags you're inspecting are different before and after translation.

If the user might disable your translator, and another inspector is not already handling the untranslated tags, you may need to create two property inspectors— one for the untranslated tag or tags, and one for the locked region created by translation. The interfaces of the two inspectors can be identical, but the comment at the top of the file and the canInspectSelection() and inspectSelection() functions—must differ.

For example, an inspector for the HAPPY tag might have a comment that looks like this:

```
<!-- tag:HAPPY,priority:5,selection:exact,hline,vline -->
```

The inspector for the translated HAPPY tag, however, would have a comment that looks like this:

```
<!-- tag:*LOCKED*,priority:5,selection:within,hline,vline -->
```

The canInspectSelection() function for the untranslated HAPPY inspector is simple: since the selection type is exact, it can return TRUE without further analysis. For the translated HAPPY inspector, this function is more complicated; the keyword *LOCKED* indicates that the inspector is appropriate when the selection is within a locked region, but because a document could have several locked regions, further checks must be performed to determine if the inspector matches this particular locked region.

Yet another problem is inherent in inspecting translated content. When you call dom.getSelection(), the values returned by default are offsets into the untranslated source. To expand the selection properly so that the locked region (and only the locked region) is selected, use the following method:

```
var currentDOM = dw.getDocumentDOM();
var offsets = currentDOM.getSelection();
var theSelection = currentDOM.offsetsToNode(offsets[0],offsets[0]+1);
```

Using offsets[0]+1 as the second argument assures that you remain within the opening lock tag when you convert the offsets to a node. If you use offsets[1] as the second argument, you risk selecting the node above the lock.

Once you have the selection (after first making sure that its nodeType node.ELEMENT_NODE), you can inspect the type attribute to see if the locked region matches this inspector. For example:

```
if (theSelection.nodeType == node.ELEMENT_NODE && theSelection.getAttribute('type')
== 'happy'){
  return true;
}else{
  return false
}
```

To populate the fields in the inspector for the translated tag, you must parse the value of the orig attribute. For example, if the untranslated code is <HAPPY TIME="22"> and the property inspector has a field labeled Time, you must extract the value of the TIME attribute from the orig string.

```
function inspectSelection() {
 var currentDOM = dw.getDocumentDOM();
 var currSelection = currentDOM.getSelection();
 var theObj = currentDOM.offsetsToNode(curSelection[0],curSelection[0]+1);

 if (theObj.nodeType != Node.ELEMENT_NODE) {
  return;
 }

 // To convert the encoded characters back to their
 // original values, use the unescape() method.
 var origAtt = unescape(theObj.getAttribute("ORIG"));

 // Convert the string to lower case for processing
 var origAttLC = origAtt.toLowerCase();

 var timeStart = origAttLC.indexOf('time=');
 var timeEnd = origAttLC.indexOf('"',timeStart+6);
 var timeValue = origAtt.substring(timeStart+6,timeEnd);

 document.layers['timelayer'].document.timeForm.timefield.value = timeValue;
}
```

Once you've parsed the orig attribute in order to populate the fields in the property inspector for the translated tag, your first instinct is probably to set the value of the orig attribute if the user changes the value in any of the fields. This is not the best course of action, however, as you are likely to run into the restriction against making changes in a locked region. You can avoid this problem by changing the original markup and then retranslating.

The property inspector for translated server-side includes (Configuration/
Inspectors/ssi_translated.js) demonstrates this technique in its setComment()
function. Rather than rewriting the orig attribute, the inspector assembles a new
SSI comment. It then inserts that comment into the document in place of the old
one by rewriting the entire document contents, which in turn generates a new orig
attribute. The following code summarizes this technique:

```
// Assemble the new include comment. radioStr and URL are
// variables defined earlier in the code.
newInc = "<!--#include " + radioStr + "=" + "" + URL + "" +" -->";

// Get the contents of the document.
var entireDocObj = dreamweaver.getDocumentDOM();
var docSrc = entireDocObj.documentElement.outerHTML;

// Store everything up to the SSI comment and everything after
// the SSI comment in the beforeSelStr and afterSelStr variables.
var beforeSelStr = docSrc.substring(0, curSelection[0] );
var afterSelStr  = docSrc.substring(curSelection[1]);

// Assemble the new contents of the document.
docSrc = beforeSelStr + newInc + afterSelStr;

// Set the outerHTML of the HTML tag (represented by
// the documentElement object) to the new contents,
// and then set the selection back to the locked region
// surrounding the SSI comment.
entireDocObj.documentElement.outerHTML = docSrc;
entireDocObj.setSelection(curSelection[0], curSelection[0]+1);
```

Finding bugs in your translator

If the translateMarkup() function contains certain types of errors, the translator loads properly, but it fails silently when invoked. Although failing silently prevents Dreamweaver from becoming unstable, it can hinder development, especially when trying to find one small syntax error in 10 or 20 or 30 lines of code.

If your translator fails, one debugging method that works well is to turn the translator into a command, as follows:

1 Copy the entire contents of the translator file to a new document, and save it in the Configuration/Commands folder inside the Dreamweaver application folder.

2 At the top of the document, between the SCRIPT tags, add the following function:

```
function commandButtons(){
  return new Array( "OK","translateMarkup(dreamweaver.getDocumentPath('document'),
dreamweaver.getSiteRoot(),
dreamweaver.getDocumentDOM().documentElement.outerHTML);
window.close()", "Cancel", "window.close()");
}
```

3 At the end of the translateMarkup() function, comment out the return *whateverTheReturnValueIs* line and replace it with dreamweaver.getDocumentDOM().documentElement.outerHTML = *whateverTheReturnValueIs*. For example:

```
 // return theCode;
  dreamweaver.getDocumentDOM().documentElement.outerHTML = theCode;
}
/* end of translateMarkup() */
```

4 In the BODY of the document, add a form with no input fields. For example:

```
<body>
<form>
Hello.
</form>
</body>
```

5 Restart Dreamweaver, and then choose your "translator command" from the Commands menu. When you click OK, the translateMarkup() function is called, simulating translation.

   If you see no error message and translation still fails, you probably have a logic error in your code.

6 Add alert() statements in strategic spots throughout the translateMarkup() function so you can make sure you're hitting the proper branches, and so you can check the values of variables and properties at different points. For example:

```
for (var i=0; i< foo.length; i++){
  alert("we're at the top of the foo.length array, and the value of i is " + i);
  /* rest of loop */
}
```

7 After adding in the alert() statements, choose your command from the Commands menu, click Cancel, and then choose it again. This reloads the command file, incorporating your changes.

INDEX

O

object object 10
objects
    adding to Insert menu 225
    adding to Object palette 224
    API 222
    user experience 222
objectTag() 223
offsetsToNode()
    dom.offsetsToNode() 106
    dreamweaver.offsetsToNode() 157
onBlur 10
onChange 10
onClick 10
onFocus 10
onLoad 10
onMouseDown 10
onMouseOut 10
onMouseOver 10
onMouseUp 10
onResize 10
open()
    MMNotes.open() 184
    site.open() 117
openDocument() 51
openDocumentFromSite() 51
openInFrame() 52
OpenNotesFile() 188
openWithApp() 46
openWithBrowseDialog() 47
openWithExternalTextEditor() 47
openWithImageEditor() 47
optimizeInFireworks() 196
option object 10
outdent() 67
outerHTML property 15

P

pageDown() 88
pageUp() 89
parentNode property
    of comment objects 17
    of document objects 14
    of tag objects 15
    of text objects 17
parentWindow property 14
password (field) object 10

playAllPlugins() 147
playPlugin() 147
playRecordedCommand() 77
popupAction() 27
popupCommand() 157
postText() 207
postTextCallback() 207
previousParagraph() 89
previousWord() 89
property inspectors
    LOCKED keyword 289
    API 247
    for locked content 289
    lightning bolt icon 283
    required functions 247
    sample code 249
    translated attributes in 283
    user experience 246
put() 118

Q

quitApplication() 73

R

radio object 10
read() 181
reapplyBehaviors() 23
receiveArguments()
    in menu commands 238
    in regular commands 230
recordPathOfLayer() 132
recreateCache() 118
recreateFromDocument() 98
redo()
    dom.redo() 76
    dreamweaver.redo() 77
refresh() 118
regexp object 10
relativeToAbsoluteURL() 102
releaseDocument() 52
reloadMenus() 100
remoteIsValid() 119
remove() 182, 185
removeAllTableHeights() 128
removeAllTableWidths() 128
removeAttribute() 15