MX

macromedia®
FLASH™MX
2004

Flash JavaScript Dictionary

Acknowledgments

Project Management: Barbara Nelson

Writing: Jay Armstrong, Robert Berry, Barbara Herbert, David Jacowitz, Barbara Nelson

Editing: Mary Kraemer

Media Design and Production: Adam Barnett, John Francis

Special thanks to Sharon Selden, Eric Mueller, Ken Eckey, Kent Carlson, Kit Kwan, Jethro Villegas, Gilles Drieu

First Edition: November 2003

Macromedia, Inc.
600 Townsend St.
San Francisco, CA 94103

CONTENTS

INTRODUCTION

The documents on this site assume you are familiar with JavaScript or ActionScript syntax and with basic programming concepts such as functions, parameters, and data types. It also assumes that you understand the concept of working with objects and properties. See the Netscape JavaScript documentation for a reference on JavaScript.

Netscape DevEdge Online has a JavaScript Developer Central site (http://developer.netscape.com/tech/javascript/index.html) that contains documentation and articles useful for understanding JavaScript. The most valuable resource is the Core JavaScript Guide.

Overview of the Macromedia Flash JavaScript API

The ActionScript language lets you write scripts to perform actions in the Macromedia Flash Player environment (that is, while a SWF is playing). The Flash JavaScript API (JSAPI) lets you write scripts to perform several actions in the Flash authoring environment (that is, while a user has the Flash program open). You can write scripts that act like commands and scripts that add tools to the Tools panel. These scripts can be used to help automate the authoring process.

The Flash JSAPI is designed to resemble the Macromedia Dreamweaver and Macromedia Fireworks JavaScript API (which were designed based on the Netscape JavaScript API). The Flash JSAPI is based on a Document Object Model (DOM), which allows Flash documents to be accessed using JavaScript objects. The Flash JSAPI includes all elements of the Netscape JavaScript API, plus the Flash DOM. These added objects and their methods and properties are described in this document. You can use any of the elements of the native JavaScript language in a Flash script, but only elements that make sense in the context of a Flash document will have an effect.

You can use Macromedia Flash Professional or your preferred text editor to write or edit Flash JavaScript (JSFL) files. If you use Flash Professional, these files have a .jsfl extension by default. To make a script appear in the Commands menu, save its JSFL file in the following folder:

- Windows 2000 or Windows XP:

  C:\Documents and Settings\<user>\Local Settings\ Application Data\Macromedia\ Flash MX2004\<language>\Configuration\Commands
- Windows 98:

  C:\Windows\Application Data\Macromedia\Flash MX 2004\ <language>\ Configuration\Commands

- Macintosh OS X:

  Hard Drive/Users/<userName>/Library/Application Support/Macromedia/Flash MX 2004/<language>/Configuration/Commands

JSFL files that create tools need to be stored in the Tools folder, which can be found in the following location:

- Windows 2000 or Windows XP:

  C:\Documents and Settings\<user>\Local Settings\ Application Data\Macromedia\ Flash MX2004\<language>\Configuration\Tools
- Windows 98:

  C:\Windows\Application Data\Macromedia\Flash MX 2004\ <language>\ Configuration\Tools
- Macintosh OS X:

  Hard Drive/Users/<userName>/Library/Application Support/Macromedia/Flash MX 2004/<language>/Configuration/Tools

If a JSFL file has other files that go with it, such as XML files, they should be stored in the same directory as the JSFL file.

You can also create a JSFL file by selecting one or more commands in the History panel and then clicking the Save As Command button in the History panel or selecting the Save As Command from the Options pop-up menu. The command (JSFL) file is saved in the Commands folder. You can then open the file and edit it the same as any other script file.

To run a script, do one of the following actions:

- Select Commands > *Command Name*.
- Select Commands > Run Command and then select the script to run.

To add a tool implemented in a JSFL file to the Flash Tools panel:

1. Copy the JSFL file for the tool and any other associated files to the Tools folder.
2. Select Edit > Customize Tools Panel (Windows) or Flash > Customize Tools Panel (Macintosh).
3. Add the tool to the list of available tools.
4. Click OK.

You can embed individual JSAPI commands in ActionScript files by using the `MMExecute()` command, which is documented in the *Flash MX 2004 ActionScript Dictionary*. However, the `MMExecute()` command has an effect only when it is used in the context of a custom user-interface element, such as a component Property inspector, or a SWF panel within the authoring environment. Even if called from ActionScript, JSAPI commands have no effect in Flash Player or outside the authoring environment.

Flash JavaScript objects contain properties and methods. Properties, each defined as a primitive type such as Boolean, int, Array, Float, or reference data types such as color, object, point, rect, and String, are used to describe the object. Methods are used to perform a function on the object. To access the properties or methods of an object, dot notation is used. Also, most objects have `getProperty()` and `setProperty()` methods, which get the value for a specified property or set the value for a specified property. Most methods take parameters that are used to specify different options for the method.

The Flash Document Object Model

The DOM for the Flash JavaScript API consists of a set of top-level functions (see "Functions" on page 12) and the top-level `flash` object. The `flash` object is guaranteed to be available to a script because it always exists when the Flash authoring environment is open. When referring to this object, you can use `flash` or `fl`. For example, to close all open files, you can use either of the following statements:

```
flash.closeAll();
fl.closeAll();
```

The `flash` object contains the following *child* objects:

Object	How to access
componentsPanel	Use `fl.componentsPanel` to access the `componentsPanel` object. This object corresponds to the Components panel in the Flash authoring environment.
Document	Use `fl.documents` to retrieve an array of all the open documents; use `fl.documents[index]` to access a particular document; use `fl.getDocumentDOM()` to access the current document (the one with focus).
drawingLayer	Use `fl.drawingLayer` to access the `drawingLayer` object.
Effect	Use `fl.effects` to retrieve an array of effect descriptors that corresponds to the effects registered when Flash starts; use `fl.effects[index]` to access a particular effect; use `fl.activeEffect` to access the effect descriptor for the current effect being applied.
Math	Use `fl.Math` to access the `Math` object.
outputPanel	Use `fl.outputPanel` to access the `outputPanel` object. This object corresponds to the Output panel in the Flash authoring environment.
Tools	`fl.tools` is an object that has a `toolObjs` property.þ The `toolObjs` property is an array of `toolObj` objects.þ Each `toolObj` object represents a tool in the Flash Tools panel.
ToolObj	Use `fl.tools.toolObjs` to retrieve an array of all tool objects; use `fl.tools.activeTool` to access the currently active tool object.
XMLUI	Use `fl.xmlui` to access the XMLUI object. This is the same as the ActionScript XMLUI object.

The Document object

An important property of the top-level `flash` object is the `documents` property. The `documents` property is contains an array of `Document` objects that each represent one of the FLA files currently open in the authoring environment. The properties of each `Document` object represent most of the elements that a FLA file can contain. Therefore, a large portion of the DOM is composed of child objects and properties of the `Document` object.

To refer to the first open document, for example, use the statement `flash.documents[0]`. or `fl.documents[0]`. The first document is the first Flash document that was opened during the current session in the authoring environment. When the first opened document is closed, the indexes of the other open documents are decremented.

To find a particular document's index use `fl.findDocumentIndex(nameOfDocument)`.

To access the current document (with focus), use the statement `flash.getDocumentDOM()` or `fl.getDocumentDOM()`. This is the syntax used in most of the examples in this document.

To find a particular document in the `documents` array, iterate through the array and test each document for its `name` property.

All the objects in the DOM that aren't listed in the previous table (see "The Flash Document Object Model" on page 8) are accessed from the `Document` object. For example, to access the library of a document, you use the `library` property of the `Document` object, which retrieves a library object:

```
fl.getDocumentDOM().library
```

To access the array of items in the Library, you use the `items` property of the `Library` object; each element in the array is an Item object:

```
fl.getDocumentDOM().library.items
```

To access a particular item in the Library, you specify a member of the `items` array:

```
fl.getDocumentDOM().library.items[0]
```

In other words, the `Library` object is a child of the `Document` object, and the `Item` object is a child of the `Library` object.

Specifying the target of an action

Unless otherwise specified, methods affect the current focus or selection. For example, the following script doubles the size of the current selection because no particular object is specified:

```
fl.getDocumentDOM().scaleSelection(2, 2);
```

In some cases, you might want an action to specifically target the currently selected item in the Flash document. To do this, use the array that the `Document.selection` property contains. The first element in the array represents the currently selected item, as shown in the following example:

```
var accDescription = fl.getDocumentDOM().selection[0].description;
```

The following script doubles the size of the first element on the Stage that is stored in the element array, instead of the current selection:

```
var element =
  fl.getDocumentDOM().getTimeline().layers[0].frames[0].elements[0];
if (element) {
  element.width = element.width*2;
  element.height = element.height*2;
}
```

You can also do something such as loop through all the elements on the Stage and increase the width and height by a specified amount, as shown in the following example:

```
var elementArray =
  fl.getDocumentDOM().getTimeline().layers[0].frames[0].elements;
  for (var i=0; i < elementArray.length; i++) {
    var offset = 10;
    elementArray[i].width += offset;
    elementArray[i].height += offset;
  }
```

Summary of the DOM structure

The following list displays the DOM structure in outline format. Numbers at the beginning of each line represent the level of an object. For example, an object preceded by "03" is a child of next highest "02" object, which, in turn, is a child of the next highest "01" object.

In some cases, an object is available by specifying a property of its parent object. For example, the `document.timelines` property contains an array of `Timeline` objects. These properties are noted in the following outline.

Finally, some objects are subclasses of other objects, rather than being children of other objects. An object that is a subclass of another object has methods and/or properties of its own in addition to the methods and properties of the other object (the superclass). Subclasses share the same level in the hierarchy as their superclass. For example, Item is a superclass of BitmapItem. These relationships are illustrated in the following outline:

```
01 Functions
01 flash
  02 componentsPanel
  02 Document (fl.documents array)
    03 Matrix
    03 Fill
    03 Stroke
    03 library
      04 Item (library.items array)
      04 BitmapItem (subclass of Item)
      04 folderItem (subclass of Item)
      04 fontItem (subclass of Item)
      04 SoundItem (subclass of Item)
      04 SymbolItem (subclass of Item)
      04 VideoItem (subclass of Item)
    03 Timeline (document.timelines array)
      04 Layer (timeline.layers array)
        05 Frame (layer.frames array)
          06 Element (frame.elements array)
            07 Matrix (Element.matrix)
          06 Instance (abstract class, subclass of Element)
          06 BitmapInstance (subclass of Instance)
          06 CompiledClipInstance (subclass of Instance)
          06 ComponentInstance (subclass of Instance)
            07 Parameter (ComponentInstance.parameters)
          06 EmbeddedVideoInstance (subclass of Instance)
          06 LinkedVideoInstance (subclass of Instance)
          06 SymbolInstance (subclass of Instance)
          06 Text (subclass of Element)
          06 TextRun (subclass of Text)
          06 TextAttrs (subclass of Text)
          06 Shape (subclass of Element)
            07 Contour (Shape.contours)
              08 HalfEdge
                09 Vertex
                09 Edge
            07 Edge (Shape.edges array)
              08 HalfEdge
                09 Vertex
                09 Edge
            07 Vertex (Shape.vertices array)
              08 HalfEdge
                09 Vertex
                09 Edge
```

```
    03 screenOutline
      04 Screen (screenOutline.screens array)
        05 Parameter (screen.parameters array)
  02 drawingLayer
    03 Path
      04 Contour
  02 Effect (fl.effects array)
  02 Math
  02 outputPanel
  02 Tools (fl.tools array)
    03 ToolObj (Tools.toolObjs array)
  02 XMLUI
```

The PolyStar example

Included with this documentation is an example Flash JSAPI script named PolyStar.jsfl. This script replicates the PolyStar tool that can be found in the Flash Tools panel. The PolyStar.jsfl file demonstrates how to build the PolyStar tool using the JSAPI. It includes detailed comments that help describe what the lines of code are doing. Read this file to gain a better understanding of how the JSAPI can be used.

Flash MX 2004 includes an earlier version of the PolyStar.jsfl script that must be removed in order to use the updated PolyStar.jsfl file.

To remove the earlier version of the PolyStar.jsfl that was installed with Flash MX 2004:

1. Select Edit > Customize Tools Panel (Windows) or Flash > Customize Tools Panel (Macintosh).
2. In the dialog box, click the Rectangle tool on the left side of the dialog box. The Rectangle tool and the PolyStar tool should now be listed in the Current Selection list on the right side of the dialog box.
3. Select the PolyStar tool in the Current Selection list.
4. Click Remove.
5. Click OK.
6. Quit Flash.
7. Remove only the PolyStar.jsfl file from the appropriate Tools folder listed in the "Overview of the Macromedia Flash JavaScript API" section. The PolyStar.xml and PolyStar.png files are needed by the new PolyStar.jsfl file that you will install later. When you restart Flash, the PolyStar tool no longer appears in the Customize Tools Panel dialog box.

To install the updated PolyStar example files:

1. Copy the new PolyStar.jsfl file to the Tools folder. The PolyStar.xml and PolyStar.png files that you see in this folder are needed by the new PolyStar.jsfl file.
2. Restart Flash.
3. Select Edit > Customize Tools Panel (Windows) or Flash > Customize Tools Panel (Macintosh). You should see PolyStar tool in the available tools list.
4. Click the Rectangle tool at the left side of the dialog box. The Rectangle Tool should appear in the Current Selection list at the right side of the dialog box.
5. Select the PolyStar tool from the Available Tools list.
6. Click Add.
7. Click OK. The PolyStar tool now appears in the Rectangle tool pop-up menu.

Functions

The following functions are available when creating extensible tools.

```
activate()
configureTool()
deactivate()
keyDown()
keyUp()
mouseDoubleClick()
mouseDown()
mouseMove()
mouseUp()
notifySettingsChanged()
setCursor()
```

activate()

Description

This function is called when the extensible tool becomes active; that is, when the tool is selected in the toolbar. Any setup the tool needs to do should be performed here.

Usage

```
function activate() {
  // statements
}
```

Arguments

None.

Returns

Nothing.

Example

```
function activate() {
  fl.trace( "Tool is active" );
}
```

configureTool()

Description

This function is called when Flash opens and the tool is loaded. Use this function to set any information Flash needs to know about the tool.

Usage

```
function configureTool() {
  // statements
}
```

Arguments

None.

Returns

Nothing.

Example

```
function configureTool() {
  theTool = fl.tools.activeTool;
  theTool.setToolName("myTool");
  theTool.setIcon("myTool.png");
  theTool.setMenuString("My Tool's menu string");
  theTool.setToolTip("my tool's tool tip");
  theTool.setOptionsFile( "mtTool.xml" );
}
```

deactivate()

Description

This function is called when the tool becomes inactive; that is, when the active tool changes from this tool to another one. Use this function to perform any cleanup the tool needs.

Usage

```
function deactivate() {
  // statements
}
```

Arguments

None.

Returns

Nothing.

Example

```
function deactivate() {
  fl.trace( "Tool is no longer active" );
}
```

keyDown()

Description

This function is called when the tool is active and the user presses a key. The script should call `tools.getKeyDown()` to determine which key was pressed.

Usage

```
function keyDown() {
  // statements
}
```

Arguments

None.

Returns

Nothing.

Example

```
function keyDown() {
  fl.trace("key " + fl.tools.getKeyDown() + " was pressed");
}
```

keyUp()

Description

This function is called when the tool is active and a key is released.

Usage

```
function keyUp() {
  // statements
}
```

Arguments

None.

Returns

Nothing.

Example

```
function keyUp() {
  fl.trace("Key is released");
}
```

mouseDoubleClick()

Description

This function is called when the mouse button is double-clicked on the Stage.

Usage

```
function mouseDoubleClick() {
  // statements
}
```

Arguments

None.

Returns

Nothing.

Example

```
function mouseDblClk() {
  fl.trace("Mouse was double-clicked");
}
```

mouseDown()

Description

This function is called whenever the tool is active and the mouse button is pressed while the pointer is over the Stage.

Usage

```
function mouseDown( [ pt ] ) {
  // statements
}
```

Arguments

The optional *pt* argument is a point that specifies the location of the mouse when the button is pressed. It is passed to the function when the mouse button is pressed.

Returns

Nothing.

Example

```
function mouseDown(pt) {
  fl.trace("Mouse button has been pressed");
}
```

mouseMove()

Description

This function is called whenever the mouse moves over a specified point on the Stage. The mouse button can be down or up.

Usage

```
function mouseMove( [ pt ] ) {
  // statements
}
```

Arguments

The optional *pt* argument is a point that specifies the current location of the mouse. It is passed to the function whenever the mouse moves, thus tracking the mouse location. If the Stage is in Edit or Edit-in-place mode, the point coordinates are relative to the object being edited. Otherwise, the point coordinates are relative to the Stage.

Returns

Nothing.

Example

```
function mouseMove(pt) {
  fl.trace("moving");
}
```

mouseUp()

Description

This function is called whenever the mouse button is released after being pressed on the Stage.

Usage

```
function mouseUp() {
  // statements
}
```

Arguments

None.

Returns

Nothing.

Example

```
function mouseUp() {
  fl.trace("mouse is up");
}
```

notifySettingsChanged()

Description

This function is called whenever the tool is active and the user changes options in the Property Inspector. The script should query the `tool` object for the current values of the options.

Usage

```
function notifySettingsChanged() {
  // statements
}
```

Arguments

None.

Returns

Nothing.

Example

```
function notifySettingsChanged() {
  var theTool = fl.tools.activeTool;
  var newValue = theTool.myProp;
}
```

setCursor()

Description

This function is called whenever the mouse moves, to allow the script to set custom pointers. The script should call `tools.setCursor()` to specify the pointer to use.

Usage

```
function setCursor() {
  // statements
}
```

Arguments

None.

Returns

Nothing.

Example

```
function setCursor() {
  fl.tools.setCursor( 1 );
}
```

BitmapInstance

Description

The BitmapInstance object is a subclass of the Instance object and represents a bitmap in a frame.

Methods

In addition to the Instance object methods, you can use the following methods with the BitmapInstance object:

`bitmapInstance.getBits()`
`bitmapInstance.setBits()`

bitmapInstance.getBits()

Description

This method lets you create bitmap effects by getting the bits out of the bitmap, manipulating them, and then returning them to Flash; see `bitmapInstance.setBits()`.

Usage

```
bitmapInstance.getBits()
```

Arguments

None.

Returns

An object that contains `width`, `height`, `depth`, `bits` and, if the bitmap has a color table, `cTab`. The `bits` element is an array of bytes. The `cTab` element is an array of color values of the form `"#rrggbb"`. The length of the array is the length of the color table.

Note: The byte array is meaningful only when referenced by an External Library. It is typically used only when creating an extensible tool or effect.

Example

The following code creates a reference to the currently selected object; tests whether the object is a bitmap; and traces the height, width, and bit depth of the bitmap:

```
var isBitmap = fl.getDocumentDOM().selection[0].instanceType;
if( isBitmap == "bitmap"){
  var bits = fl.getDocumentDOM().selection[0].getBits();
  fl.trace( "height = " + bits.height );
  fl.trace( "width = " + bits.width );
  fl.trace( "depth = " + bits.depth );
}
```

bitmapInstance.setBits()

Description

Sets the bits of an existing bitmap element. This lets you create bitmap effects by getting the bits out of the bitmap, manipulating them, and then returning the bitmap to Flash.

Usage

```
bitmapInstance.setBits(bitmap)
```

Arguments

The *bitmap* argument is an object that contains `height`, `width`, `depth`, `bits`, and `cTab` properties. The `height`, `width` and `depth` properties are integers. The `bits` property is a byte array. The `cTab` property is required only for bitmaps with a bit depth of 8 or less and is a string that represents a color value in the form `"#rrggbb"`.

Note: The byte array is meaningful only when referenced by an External Library. It is typically used only when creating an extensible tool or effect.

Returns

Nothing.

Example

The following code tests whether the current selection is a bitmap, and then reduces the height of the bitmap by 150 pixels:

```
var isBitmap = fl.getDocumentDOM().selection[0].instanceType;
if( isBitmap == "bitmap"){
  var bits = fl.getDocumentDOM().selection[0].getBits();
  bits.height = -150;
  fl.getDocumentDOM().selection[0].setBits(bits);
}
```

Properties

In addition to the Instance object properties, you can use the following properties with the BitmapInstance object:

Property	Data type	Value

`hPixels` **read only**	int	The width of the bitmap in pixels. The following code gets the width of the bitmap in pixels.
		`// get the number of pixels in the horizontal dimension.`
		`var bmObj = fl.getDocumentDOM().selection[0];`
		`var isBitmap = bmObj.instanceType;`
		`if( isBitmap == "bitmap"){`
		`  var numHorizontalPixels = bmObj.hPixels;`
		`}`

`vPixels` **read only**	int	The height of the bitmap in pixels. The following code gets the height of the bitmap in pixels.
		`// get the number of pixels in the vertical dimension`
		`var bmObj = fl.getDocumentDOM().selection[0];`
		`var isBitmap = bmObj.instanceType;`
		`if( isBitmap == "bitmap"){`
		`  var numVerticalPixels = bmObj.vPixels;`
		`}`

BitmapItem

Description

A BitmapItem object refers to a bitmap in the Library of a document. The BitmapItem object is a subclass of the Item object.

Methods

The BitmapItem object has no unique methods; see the Item object.

Properties

In addition to the Item object properties, the BitmapItem object has following properties:

Property	Data type	Value

`allowSmoothing`	Boolean	Set to `true` to allow smoothing of a bitmap; `false` otherwise. The following code sets the `allowSmoothing` property of the first item in the library of the current document to `true`.
		`fl.getDocumentDOM().library.items[0].allowSmoothing = true;`
		`alert(fl.getDocumentDOM().library.items[0].allowSmoothing);`

`compressionType`	string	Determines the type of image compression applied to the bitmap. Accepted values are `"photo"` or `"lossless"`. The value `"photo"` corresponds to JPEG using a quality from 0 to 100 if 'useImportedJPEGQuality' is `false`, or the default document quality value if 'useImportedJPEGQuality' is `true`. The value `"lossless"` corresponds to GIF or PNG formats.
		`fl.getDocumentDOM().library.items[0].compressionType = "photo";`
		`alert(fl.getDocumentDOM().library.items[0].compressionType);`

`quality`	int	To use the default document quality, specify -1; otherwise specify a value from 0 to 100. Available only for JPEG compression.
		`fl.getDocumentDOM().library.items[0].quality = 65;`
		`alert(fl.getDocumentDOM().library.items[0].quality);`

`useImportedJPEGQuality`	Boolean	To use the default imported JPEG quality, specify `true`; otherwise, specify `false`. Available only for JPEG compression.
		`fl.getDocumentDOM().library.items[0].useImportedJPEGQuality = true;`
		`alert(fl.getDocumentDOM().library.items[0].useImportedJPEGQuality);`

CompiledClipInstance

Description

The CompiledClipInstance object is a subclass of the Element object. It is essentially an instance of a movie clip that has been converted to a compiled clip library item.

Methods

The CompiledClipInstance object has no unique methods; see the Element object.

Properties

In addition to the properties of the Element object, the CompiledClipInstance object has the following properties:

Property	Data type	Value

`accName`	string	Equivalent to the Name field in the Accessibility panel. Screen readers identify objects by reading the name aloud.
		`// get the name of the object.`
		`var theName = fl.getDocumentDOM().selection[0].accName;`
		`// set the name of the object.`
		`fl.getDocumentDOM().selection[0].accName = 'Home Button';`

`actionScript`	string	String representing the ActionScript for this instance. Behaves the same as symbolInstance.actionScript
		`//assign some ActionScript to a Button compiled clip instance.`
		`fl.getDocumentDOM().getTimeline().layers[0].frames[0].elements[0].actionScript = "on(click) {trace('button is clicked');}";`
		`//assign some ActionScript to the current selected Button compiled clip instance.`
		`fl.getDocumentDOM().selection[0].actionScript = "on(click) {trace('button is clicked');}";`

`description`	string	Equivalent to the Description field on the Accessibility panel. The description is read by the screen reader.
		`// get the description of the object.`
		`var theDescription = fl.getDocumentDOM().selection[0].description;`
		`// set the description of the object.`
		`fl.getDocumentDOM().selection[0].description= "This is compiled clip number 1";`

`forceSimple`	Boolean	Enables and disables the children of the object to be accessible. This is equivalent to the inverse logic of the Make Child Objects Accessible setting in the Accessibility panel.
		If `forceSimple` is `true`, it is the same as the Make Child Objects Accessible option being unchecked.
		If `forceSimple` is `false`, it is the same as the Make Child Object Accessible option being checked.
		`// query if the children of the object are accessible`
		`var areChildrenAccessible = fl.getDocumentDOM().selection[0].forceSimple;`
		`// allow the children of the object to be accessible`
		`fl.getDocumentDOM().selection[0].forceSimple = false;`

Property	Data type	Value

shortcut	string	This is equivalent to the Shortcut field on the Accessibility panel. The shortcut is read by the screen readers. This property is not available to dynamic text type.
		`// get the shortcut key of the object`
		`var theShortcut = fl.getDocumentDOM().selection[0].shortcut;`
		`// set the shortcut key of the object`
		`fl.getDocumentDOM().selection[0].shortcut = "Ctrl+I";`

silent	Boolean	Enable or disable the accessibility of the object. This is equivalent to the inverse logic of Make Object Accessible setting in the Accessibility panel. That is, if `silent` is `true`, then Make Object Accessible is unchecked. If `silent` is `false`, then Make Object Accessible is checked.
		`// query if the object is accessible`
		`var isSilent =fl.getDocumentDOM().selection[0].silent;`
		`// set the object to be accessible`
		`fl.getDocumentDOM().selection[0].silent = false;`

tabIndex	int	This is equivalent to the Tab Index field on the Accessibility panel (Flash MX 2004 Professional only). Creates a tab order in which objects are accessed when the user presses the Tab key.
		`// get the tabIndex of the object.`
		`var theTabIndex = fl.getDocumentDOM().selection[0].tabIndex;`
		`// set the tabIndex of the object.`
		`fl.getDocumentDOM().selection[0].tabIndex = 1;`

ComponentInstance

Description

The ComponentInstance object is a subclass of the SymbolInstance object and represents a component in a frame.

Methods

The ComponentInstance object has no unique methods; see the SymbolInstance object.

Properties

In addition to all the properties of the SymbolInstance object, the ComponentInstance object has the following property:

Property	Data type	Value

`parameters`
read only
	array	The ActionScript 2.0 properties that are accessible from the Components Property Inspector. See "Parameter" on page 146.
		`var parms = fl.getDocumentDOM().selection[0].parameters;`
		`parms[0].value = "some value";`

componentsPanel

Description

The componentsPanel object, which represents the Components panel, is a property of the flash object and can be accessed by `fl.componentsPanel`.

Methods

You can use the following method with the componentsPanel object:

`componentsPanel.addItemToDocument()`

componentsPanel.addItemToDocument()

Description

Adds the specified component to the document at the specified position.

Usage

```
componentsPanel.addItemToDocument( position, categoryName, componentName )
```

Arguments

The *position* argument is a point (for example, {x:0, y:100}) that specifies the location at which to add the component. Specify *position* relative to the center point of the component, not the component's registration point.

The *categoryName* argument is a String that specifies the name of the component category (for example, "Data Components"). The valid category names are the ones listed in the Components panel.

The *componentName* argument is a String that specifies the name of the component in the specified category (for example, "WebServiceConnector"). The valid component names are the ones listed in the Components panel.

Returns

Nothing.

Examples

```
fl.componentsPanel.addItemToDocument({x:0, y:0}, "UI Components", "CheckBox");
fl.componentsPanel.addItemToDocument({x:0, y:100}, "Data Components",
  "WebServiceConnector");
fl.componentsPanel.addItemToDocument({x:0, y:200}, "UI Components", "Button");
```

Contour

Description

A Contour object represents a closed path of half edges on the boundary of a shape.

Methods

You can use the following method with the Contour object:

`contour.getHalfEdge()`

contour.getHalfEdge()

Description

Returns a HalfEdge object on the contour of the selection.

Usage

```
contour.getHalfEdge()
```

Arguments

None.

Returns

A HalfEdge object.

Example

```
// with a shape selected

var elt = fl.getDocumentDOM().selection[0];
elt.beginEdit();

var contourArray = elt.contours;
var contourCount = 0;
for (i=0;  i<contourArray.length;  i++)
{
  var contour = contourArray[i];
  contourCount++;
  var he = contour.getHalfEdge();

  var iStart = he.id;
  var id = 0;
  while (id != iStart)
  {
    // see if the edge is linear
    var edge = he.getEdge();
    var vrt = he.getVertex();

    var x = vrt.x;
    var y = vrt.y;
    fl.trace("vrt: " + x + ", " + y);

    he = he.getNext();
    id = he.id;
  }
}
elt.endEdit();
```

Properties

Property	Data type	Value

`interior`
read only
	Boolean	The value is `true` if the contour encloses an area; `false` otherwise.
		`var elt = fl.getDocumentDOM().selection[0];`
		`elt.beginEdit();`
		
		`var contourArray = elt.contours;`
		
		`var contourCount = 0;`
		`for (i=0;  i<contourArray.length;  i++) {`
		`  var contour = contourArray[i];`
		`  fl.trace("Next Contour, interior:" + contour.interior );`
		`  contourCount++;`
		`}`
		`elt.endEdit();`

`orientation`
read only
	int	An integer indicating the orientation of the contour. The value of the integer is -1 if the orientation is counterclockwise, 1 if it is clockwise, and 0 if it is a contour with no area.
		`var elt = fl.getDocumentDOM().selection[0];`
		`elt.beginEdit();`
		
		`var contourArray = elt.contours;`
		
		`var contourCount = 0;`
		`for (i=0;  i<contourArray.length;  i++) {`
		`  var contour = contourArray[i];`
		`  fl.trace("Next Contour, orientation:" +`
		`  contour.orientation);`
		`  contourCount++;`
		`}`
		`elt.endEdit();`

Document

Description

The `Document` object represents the Stage.

Methods

You can use the following methods with the `Document` object:

```
document.addDataToDocument()
document.addDataToSelection()
document.addItem()
document.addNewLine()
document.addNewOval()
document.addNewPublishProfile()
document.addNewRectangle()
document.addNewScene()
document.addNewText()
document.align()
document.allowScreens()
document.arrange()
document.breakApart()
document.canRevert()
document.canTestMovie()
document.canTestScene()
document.clipCopy()
document.clipCut()
document.clipPaste()
document.close()
document.convertLinesToFills()
document.convertToSymbol()
document.deletePublishProfile()
document.deleteScene()
document.deleteSelection()
document.distribute()
document.distributeToLayers()
document.documentHasData()
document.duplicatePublishProfile()
document.duplicateScene()
document.duplicateSelection()
document.editScene()
document.enterEditMode()
document.exitEditMode()
document.exportPublishProfile()
document.exportSWF()
document.getAlignToDocument()
document.getCustomFill()
document.getCustomStroke()
document.getDataFromDocument()
document.getElementProperty()
document.getElementTextAttr()
document.getSelectionRect()
document.getTextString()
document.getTimeline()
document.getTransformationPoint()
document.group()
document.importPublishProfile()
document.importSWF()
document.match()
```

```
document.mouseClick()
document.mouseDblClk()
document.moveSelectedBezierPointsBy()
document.moveSelectionBy()
document.optimizeCurves()
document.publish()
document.removeDataFromDocument()
document.removeDataFromSelection()
document.renamePublishProfile()
document.renameScene()
document.reorderScene()
document.resetTransformation()
document.revert()
document.rotateSelection()
document.save()
document.saveAndCompact()
document.scaleSelection()
document.selectAll()
document.selectNone()
document.setAlignToDocument()
document.setCustomFill()
document.setCustomStroke()
document.setElementProperty()
document.setElementTextAttr()
document.setFillColor()
document.setInstanceAlpha()
document.setInstanceBrightness()
document.setInstanceTint()
document.setSelectionBounds()
document.setSelectionRect()
document.setStroke()
document.setStrokeColor()
document.setStrokeSize()
document.setStrokeStyle()
document.setTextRectangle()
document.setTextSelection()
document.setTextString()
document.setTransformationPoint()
document.skewSelection()
document.smoothSelection()
document.space()
document.straightenSelection()
document.swapElement()
document.testMovie()
document.testScene()
document.traceBitmap())
document.transformSelection()
document.unGroup()
document.unlockAllElements()
document.xmlPanel()
```

document.addDataToDocument()

Description

Stores specified data with a document. Data is written to the FLA file and is available to JavaScript when the file is reopened.

Usage

```
document.addDataToDocument( name, type, data )
```

Arguments

The *name* argument is a String that specifies the name of the data to add.

The *type* argument is a String that defines the type of data to add. The valid values for *type* are `"integer"`, `"integerArray"`, `"double"`, `"doubleArray"`, `"string"`, and `"byteArray"`.

The *data* argument is the value to add. Valid types depend on the *type* argument.

Returns

Nothing.

Example

The following example adds an integer value of `12` to the current document:

```
fl.getDocumentDOM().addDataToDocument("myData", "integer", 12);
```

document.addDataToSelection()

Description

Stores specified data with the selected object(s). Data is written to FLA file and is available to JavaScript when the file is reopened. Only symbols and bitmaps support persistent data.

Usage

```
document.addDataToSelection( name, type, data )
```

Arguments

The *name* argument is a String that specifies the name of the persistent data.

The *type* argument defines the type of data. The valid values for *type* are: `"integer"`, `"integerArray"`, `"double"`, `"doubleArray"`, `"string"`, and `"byteArray"`.

The *data* argument is the value to add. Valid types depend on the *type* argument.

Returns

Nothing.

Example

The following example adds an integer value of `12` to the selected object:

```
fl.getDocumentDOM().addDataToSelection("myData", "integer", 12);
```

document.addItem()

Description

Adds an item from the specified Library to the specified `Document` object.

Usage

```
document.addItem( position, item )
```

Arguments

The *`position`* argument specifies the *x* and *y* coordinates of the location at which to add the item. Uses the center of a symbol or the top left corner of a bitmap or video.

The *`item`* argument specifies the item to add and the Library from which to add it.

Returns

A Boolean value: `true` if successful; `false` otherwise.

Example

The following example adds the first item from the Library to the first document at the specified location for the selected symbol, bitmap, or video:

```
var item = fl.documents[0].library.items[0];
fl.documents[0].addItem({x:0,y:0}, item);
```

document.addNewLine()

Description

Adds a new path between two points. The method uses the document's current stroke attributes and adds the path on the current frame and current layer. This method works the same as if you click on the line tool and draw a line.

Usage

```
document.addNewLine( startPoint, endpoint )
```

Arguments

The *`startpoint`* argument is a pair of floating point numbers that specify the *x* and *y* coordinates where the line starts.

The *`endpoint`* argument is a pair of floating point numbers that specify the *x* and *y* coordinates where the line ends.

Returns

Nothing.

Example

The following example adds a line between the specified starting point and ending point:

```
fl.getDocumentDOM().addNewLine({x:216.7, y:122.3}, {x:366.8, y:165.8});
```

document.addNewOval()

Description

Adds a new oval into the specified bounding rectangle. This method performs the same operation as the oval tool. The method uses the document's current default stroke and fill attributes and adds the oval on the current frame and layer. If *bSurpressFill* is set to `true`, the oval is drawn without a fill. If *bSurpressStroke* is set to `true`, the oval is drawn without a stroke. If both *bSuppressFill* or *bSuppressStroke* are set to `true`, the method will do nothing.

Usage

```
document.addNewOval( boundingRectangle[, bSuppressFill[, bSuppressStroke ]] )
```

Arguments

The *boundingRectangle* argument is a rectangle that specifies the bounds of the oval to be added.

The optional *bSuppressFill* argument is a Boolean value that, if it is set to `true`, causes the method to create the shape without a fill. The default value is `false`.

The optional *bSuppressStroke* argument is a Boolean value that, if it is set to `true`, causes the method to create the shape without a stroke. The default value is `false`.

Returns

Nothing.

Example

The following example adds a new oval within the specified coordinates:

```
flash.getDocumentDOM().addNewOval({left:72,top:50,right:236,bottom:228});
```

The following example draws an oval without fill:

```
flash.getDocumentDOM().addNewOval({left:72,top:50,right:236,bottom:228},
  true);
```

The following example draws an oval without stroke:

```
flash.getDocumentDOM().addNewOval({left:72,top:50,right:236,bottom:228},
  false, true);
```

document.addNewPublishProfile()

Description

Adds a new publish profile and makes it the current one.

Usage

```
document.addNewPublishProfile( [profileName ] )
```

Arguments

The optional *profileName* argument is the unique name of the new profile. If you do not specify a name, a default name is provided.

Returns

An integer that is the index of the new profile in the profiles list. Returns -1 if new profile cannot be created.

Example

The following example adds a new publish profile with the default name:

```
alert(fl.getDocumentDOM().addNewPublishProfile());
```

The following example adds a new publish profile with the name `"my profile"`:

```
alert(fl.getDocumentDOM().addNewPublishProfile("my profile"));
```

document.addNewRectangle()

Description

Adds a new rectangle or rounded rectangle, fitting it into the specified bounds. This method performs the same operation as the rectangle tool. The method uses the document's current default stroke and fill attributes and adds the rectangle on the current frame and layer. If the *bSuppressFill* argument is set to `true`, the rectangle is drawn without a fill. If the *bSuppressStroke* argument is set to `true`, the rectangle is drawn without a stroke. Either *bSuppressFill* or *bSuppressStroke* must be set to `false` or the method does nothing.

Usage

```
document.addNewRectangle( boundingRectangle, roundness
  [, bSuppressFill [, bSuppressStroke ] ] )
```

Arguments

The *boundingRectangle* argument is a rectangle that specifies the bounds within which the new rectangle is added. This argument specifies a pixel location for `left`, `top`, `right`, and `bottom`.

The *roundness* argument is an integer value between 0 and 999 that specifies the roundness to use for the corners. The value is specified as number of points. The greater the value, the greater the roundness.

The optional *bSuppressFill* argument is a Boolean value that, if it is set to `true`, causes the method to create the shape without a fill. The default value is `false`.

The optional *bSuppressStroke* argument is a Boolean value that, if it is set to `true`, causes the method to create the rectangle without a stroke. The default value is `false`.

Returns

Nothing.

Example

The following example adds a new rectangle with no round corners within the specified coordinates.

```
flash.getDocumentDOM().addNewRectangle({left:0,top:0,right:100,bottom:100},0);
```

The following example adds a new rectangle with no round corners and without a fill.

```
flash.getDocumentDOM().addNewRectangle({left:0,top:0,right:100,bottom:100},0,
  true);
```

The following example add a new rectangle no round corners and without a stroke.

```
flash.getDocumentDOM().addNewRectangle({left:0,top:0,right:100,bottom:100},0,
  false, true);
```

document.addNewScene()

Description

Adds a new scene (`Timeline` object) as the next scene after the currently selected scene and makes the new scene the currently selected scene. If the specified scene name already exists, the scene is not added and the method returns an error.

Usage

```
document.addNewScene( [name] )
```

Arguments

`[name]`

The *`name`* argument specifies the name of the scene. If you do not specify a name, a new scene name is generated.

Returns

A Boolean value: `true` if the scene is added successfully; `false` otherwise.

Example

The following example adds a new scene named `myScene` after the current scene in the current document. The variable *`success`* will be `true` when the new scene is created, `false` otherwise

```
var success = flash.getDocumentDOM().addNewScene("myScene");
```

The following example adds a new scene using the default naming convention. If only one scene exists newly created scene is named `"Scene 2"`.

```
fl.getDocumentDOM().addNewScene();
```

document.addNewText()

Description

Inserts a new empty text field.

Usage

```
document.addNewText( boundingRectangle )
```

Arguments

The *`boundingRectangle`* argument specifies the size and location of the text field by providing locations in pixels for `left`, `top`, `right`, and `bottom`. The method applies the current text attributes. It should be followed by calling `setTextString()` to populate the new text box.

Returns

Nothing.

Example

The following example creates a new text field in the top, left corner of the stage:

```
fl.getDocumentDOM().addNewText({left:0, top:0, right:100, bottom:100});
```

document.align()

Description

Aligns the selection.

Usage

```
document.align( alignmode [, bUseDocumentBounds ] )
```

Arguments

The *alignmode* argument is a String that specifies how to align the selection. Valid values for *alignmode* are `"left"`, `"right"`, `"top"`, `"bottom"`, `"vertical center"`, and `"horizontal center"`.

The optional *bUseDocumentBounds* argument is a Boolean value that, if it is set to `true`, causes the method to align to the bounds of the document. Otherwise, the method uses the bounds of the selected objects. The default is `false`.

Returns

Nothing.

Example

The following example aligns objects to left and to Stage. This is equivalent to turning on the to-stage setting in the align panel and clicking the Align to left button.

```
fl.getDocumentDOM().align("left", true);
```

document.allowScreens()

Description

Use this method before using the `screenOutline` property. If this method returns the value `true`, you can safely access the `screenOutline` property. Flash displays an error if you access the `screenOutline` property on a document without screens.

Usage

```
document.allowScreens()
```

Arguments

None.

Returns

Returns a Boolean value: `true` if `dom.screenOutline` can be used safely; `false` otherwise.

Example

The following example determines whether `screens` methods can be used in the current document:

```
if(fl.getDocumentDOM().allowScreens()) {
  fl.trace("screen outline is available.");
}
else {
  fl.trace("whoops, no screens.");
}
```

document.arrange()

Description

Arranges the selection on the Stage. This method applies to only to non-shape objects.

Usage

```
document.arrange( arrangeMode )
```

Arguments

The *`arrangeMode`* argument specifies the direction in which to move the selection. The valid values for *`arrangemode`* are `"back"`, `"backward"`, `"forward"`, and `"front"`. Provides the same capabilities as these options provide on the Modify >Arrange menu.

Returns

Nothing.

Example

The following example moves the current selection to the front:

```
fl.getDocumentDOM().arrange("front");
```

document.breakApart()

Description

Performs a break-apart operation on the current selection.

Usage

```
document.breakApart()
```

Arguments

None.

Returns

Nothing.

Example

The following example breaks apart the current selection:

```
fl.getDocumentDOM().breakApart();
```

document.canEditSymbol()

Description

Indicates whether Edit Symbols menu and functionality is enabled. This is independent of the selection status of any objects stage, but rather if one or more symbols are defined. This should not be used to test whether `fl.getDocumentDOM().enterEditMode()` is allowed.

Usage

```
document.canEditSymbol()
```

Arguments

None.

Returns

A Boolean value: `true` if the Edit Symbols menu and functionality is enabled; `false` otherwise.

Example

The following example displays in the Output panel the state of the Edit Symbols menu and functionality:

```
fl.trace("fl.getDocumentDOM().canEditSymbol() returns: " +
  fl.getDocumentDOM().canEditSymbol());
```

document.canRevert()

Description

Determines whether the `document.revert()` or `fl.revertDocument()` methods can be used successfully.

Usage

```
document.canRevert()
```

Arguments

None.

Returns

A Boolean value: `true` if you can use the `document.revert()` or `fl.revertDocument()` methods successfully; `false` otherwise.

Example

The following example checks whether the current document can revert to the previously saved version. If so, `fl.getDocumentDOM().revert()` restores the previously saved version.

```
if(fl.getDocumentDOM().canRevert()){
  fl.getDocumentDOM().revert();
}
```

document.canTestMovie()

Description

`fl.getDocumentDOM().canTestMovie()`

Determines whether you can use the `document.testMovie()` method successfully.

Usage

```
document.canTestMovie()
```

Arguments

None.

Returns

A Boolean value: `true` if you can use the `document.testMovie()` method successfully: `false` otherwise.

Example

The following example tests whether `fl.getDocumentDOM().testMovie()` can be used. If so, it calls the method.

```
if(fl.getDocumentDOM().canTestMovie()){
  fl.getDocumentDOM().testMovie();
}
```

document.canTestScene()

Description

Determines whether you can use the `document.testScene()` method successfully.

Usage

```
document.canTestScene()
```

Arguments

None.

Returns

A Boolean value: `true` if you can use the `document.testScene()` method successfully; `false` otherwise.

Example

The following example first tests whether `fl.getDocumentDOM().testScene()` can be used successfully. If so, it calls the method.

```
if(fl.getDocumentDOM().canTestScene()){
  fl.getDocumentDOM().testScene();
}
```

document.clipCopy()

Description

Copies the current selection from the document to the Clipboard.

Usage

```
document.clipCopy()
```

Arguments

None.

Returns

Nothing.

Example

The following example copies the current selection from the document to the Clipboard:

```
fl.getDocumentDOM().clipCopy();
```

document.clipCut()

Description

Cuts the current selection from the document and writes it to the Clipboard.

Usage

```
document.clipCut()
```

Arguments

None.

Returns

Nothing.

Example

The following example cuts the current selection from the document and writes it to the Clipboard:

```
fl.getDocumentDOM().clipCut();
```

document.clipPaste()

Description

Pastes the contents of the Clipboard into the document.

Usage

```
document.clipPaste( [bInPlace] )
```

Arguments

The optional *bInPlace* argument is a Boolean value that, when set to `true`, causes the method to perform a paste-in-place operation. The default value is `false`, which causes the method to perform a paste operation to the center of the document.

Returns

Nothing.

Example

The following examples pastes the Clipboard contents to the center of the document:

```
fl.getDocumentDOM().clipPaste();
```

The following example pastes the Clipboard contents in place in the current document:

```
fl.getDocumentDOM().clipPaste(true);
```

document.close()

Description

Closes the current document.

Usage

```
document.close( [bPromptToSaveChanges] )
```

Arguments

The optional *bPromptToSaveChanges* argument is a Boolean value that, when set to `true`, causes the method to prompt the user with a dialog box if there are unsaved changes in the document. If *bPromptToSaveChanges* is set to `false`, the user is not prompted to save any changed documents. The default value is `true`.

Returns

Nothing.

Example

The following example closes the current document and prompts the user with a dialog box to save changes:

```
fl.getDocumentDOM().close();
```

The following example closes the current document without saving changes:

```
fl.getDocumentDOM().close(false);
```

document.convertLinesToFills()

Description

Converts lines to fills on the selected objects.

Usage

```
document.convertLinesToFills()
```

Arguments

None.

Returns

Nothing.

Example

The following example converts the current selected lines to fills:

```
fl.getDocumentDOM().convertLinesToFills();
```

document.convertToSymbol()

Description

Converts the selected Stage item(s) to a new symbol.

Usage

```
document.convertToSymbol( type, name, registrationPoint )
```

Arguments

The *type* argument is a String that specifies the type of symbol to create. Valid values for *type* are `"movie clip"`, `"button"`, and `"graphic"`.

The *name* argument is a String that specifies the name for the new symbol, which must be unique. You can submit an empty string to have this method create a unique symbol name for you.

The *registration point* argument specifies the point that represents the (0,0) location for the symbol. Acceptable values are: `"top left"`, `"top center"`, `"top right"`, `"center left"`, `"center"`, `"center right"`, `"bottom left"`, `"bottom center"`, and `"bottom right"`.

Returns

An object for the newly created symbol, or `null` if it cannot create the symbol.

Example

The following examples create a movie clip symbol with a specified name, a button symbol with a specified name, and a movie clip symbol with a default name:

```
newMc = fl.getDocumentDOM().convertToSymbol("movie clip", "mcSymbolName", "top
  left");
newButton = fl.getDocumentDOM().convertToSymbol("button", "btnSymbolName",
  "bottom right");
newClipWithDefaultName = fl.getDocumentDOM().convertToSymbol("movie clip", "",
  "top left");
```

document.deletePublishProfile()

Description

Delete the currently active profile, if there is more than one. There must be at least one profile left.

Usage

```
document.deletePublishProfile()
```

Arguments

None.

Returns

An integer that is the index of the new current profile. If a new profile is not available, the method leaves the current profile unchanged and returns its index.

Example

The following example deletes the currently active profile, if there is more than one, and displays the index of the new currently active profile:

```
alert(fl.getDocumentDOM().deletePublishProfile());
```

document.deleteScene()

Description

Deletes the current scene (Timeline object) and, if the deleted scene was not the last one, sets the next scene as the current Timeline object. If the deleted scene was the last one, sets the first object as the current Timeline object. If only one `Timeline` object (scene) exists, it returns the value `false`.

Usage

```
document.deleteScene()
```

Arguments

None.

Returns

A Boolean value: `true` if the scene is successfully deleted; `false` otherwise.

Example

Assuming there are 3 scenes (`Scene0`, `Scene1`, and `Scene2`) in the current document, the following example makes `Scene2` the current scene and then deletes it:

```
fl.getDocumentDOM().editScene(2);
var success = fl.getDocumentDOM().deleteScene();
```

document.deleteSelection()

Description

Deletes the current selection on the Stage. Displays an error message if there is no selection.

Usage

```
document.deleteSelection()
```

Arguments

None.

Returns

Nothing.

Example

The following example deletes the current selection in the document:

```
fl.getDocumentDOM().deleteSelection();
```

document.distribute()

Description

Distributes the selection.

Usage

```
document.distribute( distributemode [, bUseDocumentBounds ] )
```

Arguments

The *distributemode* argument is a String that specifies where to distribute the selected object. Valid values for *distributeMode* are `"left edge"`, `"horizontal center"`, `"right edge"`, `"top edge"`, `"vertical center"`, and `"bottom edge"`.

The *bUseDocumentBounds* argument is a Boolean value that, when set to `true`, distributes the selected objects using the bounds of the document. Otherwise, the method uses the bounds of the selected object. The default is `false`.

Returns

Nothing.

Example

The following example distributes the selected objects by the top edge:

```
fl.getDocumentDOM().distribute("top edge");
```

The following example distributes the selected objects by top edge and expressly sets the *bUseDcoumentBounds* argument:

```
fl.getDocumentDOM().distribute("top edge", false);
```

The following example distributes the selected objects by their top edges, using the bounds of the document:

```
fl.getDocumentDOM().distribute("top edge", true);
```

document.distributeToLayers()

Description

Performs a distribute-to-layers operation on the current selection, which is the same as selecting the menu item Distribute to Layers. Displays an error if there is no selection.

Usage

```
document.distributeToLayers()
```

Arguments

None.

Returns

Nothing.

Example

The following example distributes the current selection to layers:

```
fl.getDocumentDOM().distributeToLayers();
```

document.documentHasData()

Description

Checks the document for persistent data with the specified name.

Usage

```
document.documentHasData( name )
```

Arguments

The *name* argument is a String that specifies the name of the data to check.

Returns

A Boolean value: `true` if the document has persistent data; `false` otherwise.

Example

The following example checks the document for persistent data with the name `"myData"`:

```
var hasData = fl.getDocumentDOM().documentHasData("myData");
```

document.duplicatePublishProfile()

Description

Duplicates the currently active profile and gives the duplicate version focus.

Usage

```
document.duplicatePublishProfile( [profileName ] )
```

Arguments

The optional *profileName* argument is a String that specifies the unique name of the duplicated profile. If you do not specify a name, the method uses the default name.

Returns

An integer that is the index of the new profile in the profile list. Returns -1 if the profile cannot be duplicated.

Example

The following example duplicates the currently active profile and displays the index of the new profile in the profiles list:

```
alert(fl.getDocumentDOM().duplicatePublishProfile("dup profile"));
```

document.duplicateScene()

Description

Makes a copy of the currently selected scene, giving the new scene a unique name and making it the current scene.

Usage

```
document.duplicateScene()
```

Arguments

None.

Returns

A Boolean value: `true` if the scene is duplicated successfully; `false` otherwise.

Example

The following example duplicates the second scene in the current document:

```
fl.getDocumentDOM().editScene(1); //set the middle scene to current scene
var success = fl.getDocumentDOM().duplicateScene();
```

document.duplicateSelection()

Description

Duplicates the selection on the stage.

Usage

```
document.duplicateSelection()
```

Arguments

None.

Returns

Nothing.

Example

The following example duplicates the current selection, which is similar to Alt-clicking and then dragging an item:

```
fl.getDocumentDOM().duplicateSelection();
```

document.editScene()

Description

Makes the specified scene the currently selected scene for editing.

Usage

```
document.editScene( index )
```

Arguments

The *index* argument is a zero-based integer that specifies which scene to edit.

Returns

Nothing.

Example

Assuming that there are 3 scenes (`Scene0`, `Scene1`, and `Scene2`) in the current document, the following example makes `Scene2` the current scene and then deletes it:

```
fl.getDocumentDOM().editScene(2);
fl.getDocumentDOM().deleteScene();
```

document.enterEditMode()

Description

This method switches the authoring tool into the editing mode specified by the argument. If no argument is specified, the method defaults to Edit Symbol mode, which is the same effect as right-clicking on the symbol to invoke the context menu and then selecting Edit.

Usage

```
document.enterEditMode( [editMode] )
```

Arguments

The optional *`editMode`* argument is a String that specifies the editing mode. Valid values are `"inPlace"`, or `"newWindow"`. No argument means Edit Symbol mode.

Returns

Nothing.

Example

The following example puts Flash in edit-in-place mode for the currently selected symbol:

```
fl.getDocumentDOM().enterEditMode('inPlace');
```

The following example puts Flash in edit-in-new-window mode for the currently selected symbol:

```
fl.getDocumentDOM().enterEditMode('newWindow');
```

document.exitEditMode()

Description

Exits from Edit Symbols mode and returns focus to the document that is one level up.

Usage

```
document.exitEditMode()
```

Arguments

None.

Returns

Nothing.

Example

The following example exits Edit Symbol mode:

```
fl.getDocumentDOM().exitEditMode();
```

document.exportPublishProfile()

Description

Exports the currently active profile to a file.

Usage

```
document.exportPublishProfile( fileURI )
```

Arguments

The *fileURI* argument is a String, expressed as a file:///URL, that specifies the path of the XML file to which the profile is exported.

Returns

Nothing.

Example

The following example exports the currently active profile to the file named profile.xml in the folder /Documents and Settings/username/Desktop on the C drive:

```
fl.getDocumentDOM().exportPublishProfile('file:///C|/Documents and Settings/
  username/Desktop/profile.xml');
```

document.exportSWF()

Description

Export the document to the specified file in the Flash SWF format.

Usage

```
document.exportSWF( [fileURI]{, bCurrentSettings] )
```

Arguments

The optional *fileURI* argument is a String, expressed as a file:///URL, that specifies the name of the exported file. If *fileURI* is empty or not specified, Flash displays the Export Movie dialog box.

The optional *bCurrentSettings* argument is a Boolean value that, when set to `true`, causes Flash to use current SWF publish settings. Otherwise, Flash displays the Export Flash Player dialog box. The default is `false`.

Returns

Nothing.

Example

The following example exports the document to the specified file location with the current publish settings:

```
fl.getDocumentDOM().exportSWF("file:///C|/Documents and Settings/gdrieu/
  Desktop/qwerty.swf");
```

The following example displays the Export Movie dialog box and the Export Flash Player dialog box and then exports the document based on the specified settings:

```
fl.getDocumentDOM().exportSWF("", true);
```

The following example displays the Export Movie dialog box and then exports the document with specified settings:

```
fl.getDocumentDOM().exportSWF();
```

document.getAlignToDocument()

Description

Identical to retrieving the value of the To Stage button in the Align panel. Gets the preference that can be used for `document.align()`, `document.distribute()`, `document.match()`, and `document.space()` methods on the document.

Usage

```
document.getAlignToDocument()
```

Arguments

None.

Returns

A Boolean value: `true` if the preference is set to align the objects to the Stage; `false` otherwise.

Example

The following example retrieves the value of the To Stage button in the Align panel. If the return value is `true`, the To Stage button is active. Otherwise, it is not.

```
var isAlignToDoc = fl.getDocumentDOM().getAlignToDocument();
fl.getDocumentDOM().align("left", isAlignToDoc);
```

document.getCustomFill()

Description

Retrieves the fill object of the selected shape or, if specified, the toolbar.

Usage

```
document.getCustomFill( [locationOfFill] )
```

Arguments

The optional *`locationOfFill`* argument is a String that specifies the location of the fill object. The following values are valid:

- `"toolbar"`, which returns the fill color chip of the toolbar
- `"selection"`, which returns the fill object for the selection

If the argument is not set, it defaults to `"selection"`. If there is no selection, it returns `"undefined"`.

Returns

The fill object specified by the *`locationOfFill`* argument, if successful. Otherwise, it returns `"undefined"`.

Example

In the following examples, the first two lines each retrieve the fill object of the selection, and the third line retrieves the fill object of the toolbar:

```
var fill = fl.getDocumentDOM().getCustomFill();
var fill = fl.getDocumentDOM().getCustomFill("selection");
var fill = fl.getDocumentDOM().getCustomFill("toolbar");
```

document.getCustomStroke()

Description

Returns the stroke object selection; if the `locationOfStroke` argument is set to `true`, returns the fill object of the toolbar.

Usage

```
document.getCustomStroke( [locationOfStroke] )
```

Arguments

The optional *`locationOfStroke`* argument is a String that specifies the location of the stroke object. The following values are valid:

- `"toolbar"`, which, if set, returns the stroke color chip of the toolbar.
- `"selection"`, which, if set, returns the stroke object for the selection.

If the argument is not set, it defaults to `"selection"`. In the case of no selection, it returns `"undefined"`.

Returns

The stroke object specified by the *`locationOfStroke`* argument, if successful. Otherwise, it returns `"undefined"`.

Example

The following example returns the stroke object of the selection:

```
var currentStroke = fl.getDocumentDOM().getCustomStroke()
```

document.getDataFromDocument()

Description

Retrieves the value of the specified data. The type returned depends on the type of data that was stored.

Usage

```
document.getDataFromDocument( name )
```

Arguments

The *name* argument is a String that specifies the name of the data to return.

Returns

The specified data.

Example

The following example returns the value of the data named `"MyData"`, which is stored with the document:

```
var data = fl.getDocumentDOM().getDataFromDocument("MyData");
```

document.getElementProperty()

Description

Gets the specified Element property for the current selection. See "Properties" on page 95 for a list of valid values.

Usage

```
document.getElementProperty( propertyName )
```

Arguments

The name of the Element property for which to retrieve the value.

Returns

The value of the specified property. Returns `null` if the property is an indeterminate state, as when multiple elements are selected with different property values. Returns `"undefined"` if the property is not a valid property of the selected element.

Example

The following example gets the `name` of the Element property for the current selection:

```
//elementName = the instance name of the selected object
var elementName = fl.getDocumentDOM().getElementProperty("name");
```

document.getElementTextAttr()

Description

Gets a specific `TextAttrs` property of the selected text objects. Selected objects that are not text fields are ignored.

Usage

```
document.getElementTextAttr( attrName [, startIndex [, endIndex]] )
```

Arguments

The *`attrName`* argument is a String that specifies the name of the `TextAttrs` property to be returned.

The optional *`startIndex`* argument is an integer that specifies the index of first character, with 0 (zero) specifying the first position.

The optional *`endIndex`* argument is an integer that specifies the index of last character.

Returns

See "TextAttrs" on page 193 for property names and the return values associated with them. If one text field is selected, the property is returned if there is only one value used within the text. Returns `"undefined"` if there are several values used inside the text field. If several text fields are selected, and all the text alignment values are equal, the method returns this value. If several text fields are selected, but all the text alignment values are not equal, the method returns `"undefined"`. If the optional arguments are not passed, the rules above apply to the range of text currently selected, or the whole text field if the text is not currently being edited. If only *`startIndex`* is passed, the property of the character to the right of the index is returned, if all the selected text objects match values. If *`startIndex`* and *`endIndex`* are passed, the value returned will reflect the entire range of characters from *`startIndex`* up to, but not including, *`endIndex`*.

Example

The following example gets the size of the selected text fields:

```
fl.getDocumentDOM().getElementTextAttr("size");
```

The following example gets the color of the character at index 3 in the selected text fields:

```
fl.getDocumentDOM().getElementTextAttr("fillColor", 3);
```

The following example gets the font name of the text from index 2 up to, but not including, index 10 of the selected text fields:

```
fl.getDocumentDOM().getElementTextAttr("face", 2, 10);
```

document.getSelectionRect()

Description

Gets the bounding rectangle of the current selection. If a selection is non-rectangular, the smallest rectangle encompassing the entire selection is returned. The rectangle is based on the document space or when in an edit mode, the registration point of the symbol being edited.

Usage

```
document.getSelectionRect()
```

Arguments

None.

Returns

The bounding rectangle of the current selection, or 0 if nothing is selected.

Example

The following example gets the bounding rectangle for the current selection and then displays its properties:

```
var newRect = fl.getDocumentDOM().getSelectionRect();
var outputStr = "left: " + newRect.left + " top: " + newRect.top + " right: " +
  newRect.right + " bottom: " + newRect.bottom;
alert(outputStr);
```

document.getTextString()

Description

Gets the currently selected text. If the optional parameters are not passed, the current text selection is used. If text is not currently opened for editing, the whole text string is returned. If only *startIndex* is passed, the string starting at that index and ending at the end of the field are returned. If *startIndex* and *endIndex* are passed, the string starting from *startIndex* up to, but not including, *endIndex* is returned.

If there are several text fields selected, the concatenation of all the strings is returned.

Usage

```
document.getTextString( [startIndex [, endIndex]] )
```

Arguments

The optional *startIndex* argument is an integer that is an index of first character to get.

The optional *endIndex* argument is an integer that is an index of last character to get.

Returns

A String that contains the selected text.

Example

The following example gets the string in the selected text fields:

```
fl.getDocumentDOM().getTextString();
```

The following example gets the string at character index 5 in the selected text fields:

```
fl.getDocumentDOM().getTextString(5);
```

The following example gets the string from character index 2 up to, but not including, character index 10:

```
fl.getDocumentDOM().getTextString(2, 10);
```

document.getTimeline()

Description

Retrieves the current Timeline in the document. The current Timeline can be the current scene, the current symbol being edited, or the current screen.

Usage

```
document.getTimeline()
```

Arguments

None.

Returns

The current Timeline object.

Example

The following example gets the Timeline object and displays its name:

```
var timeline = fl.getDocumentDOM().getTimeline();
alert(timeline.name);
```

document.getTransformationPoint()

Description

Gets the location of the transformation point of the current selection. You can use the transformation point for commutations such as rotate and skew.

Usage

```
document.getTransformationPoint()
```

Arguments

None.

Returns

The location of the transformation point.

Example

The following example gets the transformation point for the current selection. The `transPoint.x` property gives the *x* coordinate of the transformation point. The `transPoint.y` property gives the *y* coordinate of the transformation point:

```
var transPoint = fl.getDocumentDOM().getTransformationPoint();
```

document.group()

Description

Converts the current selection to a group.

Usage

```
document.group()
```

Arguments

None.

Returns

Nothing.

Example

The following example converts the objects in the current selection to a group:

```
fl.getDocumentDOM().group();
```

document.importPublishProfile()

Description

Import a profile from a file.

Usage

```
document.importPublishProfile( fileURI )
```

Arguments

The *fileURI* argument is a String that specifies the path, expressed as a file://URL, of the XML file defining the profile to import.

Returns

An integer that is the index of the imported profile in the profiles list. Returns -1 if the profile cannot be imported

Example

The following example imports the profile contained in the profile.xml file and displays its index in the profiles list:

```
alert(fl.getDocumentDOM().importPublishProfile('file:///C|/Documents and
  Settings/username/Desktop/profile.xml'));
```

document.importSWF()

Description

Imports a SWF into the document. Performs the same operation as using the Import menu option to specify a SWF file.

Usage

```
document.importSWF( fileURI )
```

Arguments

The *fileURI* argument is a String that specifies the URI, expressed as a file://URI, for the SWF file to import.

Returns

Nothing.

Example

The following example imports the "mySwf.swf" file from the Flash Configuration folder:

```
fl.getDocumentDOM().importSWF(fl.configURI+"mySwf.swf");
```

document.match()

Description

Makes the size of the selected objects the same.

Usage

```
document.match( bWidth, bHeight [, bUseDocumentBounds] )
```

Arguments

The *bWidth* argument is a Boolean value that, when set to true, causes the method to make the widths of the selected items the same.

The *bHeight* argument is a Boolean value that, when set to true, causes the method to make the heights of the selected items the same.

The optional *bUseDocumentBounds* argument is a Boolean value that, when set to true, causes the method to match the size of the objects to the bounds of the document. Otherwise, the method uses the bounds of the largest object. The default is false.

Returns

Nothing.

Example

The following example matches the width of the selected objects only:

```
fl.getDocumentDOM().match(true,false);
```

The following example Matches the height only:

```
fl.getDocumentDOM().match(false,true);
```

The following example matches the width only to the bounds of the document:

```
fl.getDocumentDOM().match(true,false,true);
```

document.mouseClick()

Description

Effectively performs a mouse click from the arrow tool.

Usage

```
document.mouseClick( position, bToggleSel, bShiftSel )
```

Arguments

The *position* argument is a pair of floating point values that specify the *x* and *y* coordinates of the click in pixels.

The *bToggleSel* argument is a Boolean value that specifies the state of the Shift key: `true` for pressed; `false` for not pressed.

The *bShiftSel* argument is a Boolean value that specifies the state of the application preference Shift select.: `true` for on; `false` for off.

Returns

Nothing.

Example

The following example performs a mouse click at the specified location:

```
fl.getDocumentDOM().mouseClick({x:300, y:200}, false, false);
```

document.mouseDblClk()

Description

Performs a double mouse click from the arrow tool.

Usage

```
document.mouseDblClk( position, bAltDown, bShiftDown, bShiftSelect )
```

Arguments

The *position* argument is a pair of floating point values that specify the *x* and *y* coordinates of the click in pixels.

The *bAltdown* argument is a Boolean value that records whether the Alt key is down at the time of the event: `true` for pressed; `false` for not pressed.

The *bShiftDown* key is a Boolean value that records whether the Shift key was down when the event occurred: `true` for pressed; `false` for not pressed.

The *bShiftSelect* argument is a Boolean value that indicates the state of the application preference Shift select: `true` for on; `false` for off.

Returns

Nothing.

Example

The following example performs a double mouse click at the specified location:

```
fl.getDocumentDOM().mouseDblClk({x:392.9, y:73}, false, false, true);
```

document.moveSelectedBezierPointsBy()

Description

If the selection contains at least one path with at least one Bézier point selected, this method moves all selected Bézier points on all selected paths by the specified amount.

Usage

```
document.moveSelectedBezierPointsBy( delta )
```

Arguments

The *delta* argument is a pair of floating point values that specify the *x* and *y* coordinates in pixels by which the selected Bézier points are moved. For example, passing ({x:1,y:2}) specifies a location that is to the right by one pixel and down by two pixels from the current location.

Returns

Nothing.

Example

The following example moves the selected Bézier points 10 pixels to the right and 5 pixels down:

```
fl.getDocumentDOM().moveSelectedBezierPointsBy({x:10, y:5});
```

document.moveSelectionBy()

Description

Moves selected objects by a specified amount of distance.

Note: When using arrow keys to move the item, the History panel combines all presses of the arrow key as one move step. When the user presses the arrow keys repeatedly, rather than taking multiple steps in the History panel, the method performs one step, and the arguments are updated to reflect the repeated arrow keys.

Usage

```
document.moveSelectionBy( distanceToMove )
```

Arguments

The *distanceToMove* argument is a pair of floating point values that specify the *x* and *y* coordinate values by which the method moves the selection. For example, passing ({x:1,y:2}) specifies a location one pixel to the right and two pixels down from the current location.

Returns

Nothing.

Example

The following example moves the selected item 62 pixels to the right and 84 pixels down:

```
flash.getDocumentDOM().moveSelectionBy({x:62, y:84});
```

document.optimizeCurves()

Description

Optimizes smoothing for the current selection, allowing multiple passes, if specified, for optimal smoothing. Performs the same operation as the Modify > Shape > Optimize menu item.

Usage

```
document.optimizeCurves( smoothing, bUseMultiplePasses )
```

Arguments

The *smoothing* argument is an integer in the range from 0 to 100, with 0 specifying no smoothing, and 100 specifying maximum smoothing.

The *bUseMultiplePasses* argument is a Boolean value that, when set to `true`, indicates that the method should use multiple passes, which is slower but produces a better result. This argument has the same effect as clicking the Use multiple passes button in the Optimize Curves dialog box.

Returns

Nothing.

Example

The following example optimizes the curve of the current selection to 50 degrees of smoothing with multiple passes:

```
fl.getDocumentDOM().optimizeCurves(50, true);
```

document.publish()

Description

Publishes the document according to the active Publish Settings (see File > Publish Settings menu item). This method performs the same action as the File > Publish menu item.

Usage

```
document.publish()
```

Arguments

None.

Returns

Nothing.

Example

The following example publishes the current document:

```
fl.getDocumentDOM().publish();
```

document.removeDataFromDocument()

Description

Removes persistent data with the specified name that has been attached to the document.

Usage

```
document.removeDataFromDocument( name )
```

Arguments

The *name* argument is a String that specifies the name of the data to remove.

Returns

Nothing.

Example

The following example removes from the document the persistent data named `"myData"`:

```
fl.getDocumentDOM().removeDataFromDocument("myData");
```

document.removeDataFromSelection()

Description

Removes persistent data with the specified name that has been attached to the selection.

Usage

```
document.removeDataFromSelection( name )
```

Arguments

name

The *name* argument is a String that specifies the name of the persistent data to remove.

Returns

Nothing.

Example

The following example removes from the selection the persistent data named `"myData"`:

```
fl.getDocumentDOM().removeDataFromSelection("myData");
```

document.renamePublishProfile()

Description

Renames the current profile.

Usage

```
document.renamePublishProfile( [profileNewName ] )
```

Arguments

The optional *profileNewName* argument specifies the new name for the profile. The new name must be unique. If the name is not specified, a default name is provided.

Returns

A Boolean value: `true` if the name is changed successfully; `false` otherwise.

Example

The following example renames the current profile to a default name and displays it:

```
alert(fl.getDocumentDOM().renamePublishProfile());
```

document.renameScene()

Description

Renames the currently selected scene in the Scenes panel. The new name for the selected scene must be unique.

Usage

```
document.renameScene( name )
```

Arguments

name

The *name* argument is a String that specifies the new name of the scene.

Returns

A Boolean value: `true` if the name is changed successfully; `false` otherwise. If the new name is not unique, for example, the method returns `false`.

Example

The following example renames the current scene to `"new name"`:

```
var success = fl.getDocumentDOM().renameScene("new name");
```

document.reorderScene()

Description

Moves the specified scene before another specified scene.

Usage

```
document.reorderScene( sceneToMove, sceneToPutItBefore )
```

Arguments

The *sceneToMove* argument is an integer that specifies which scene to move, with 0 (zero) being the first scene.

The *sceneToPutItBefore* argument is an integer that specifies the scene before which you want to move the scene specified by *sceneToMove*. Specify 0 (zero) for the first scene. For example, if you specify 1 for *sceneToMove* and 0 for *sceneToPutItBefore*, the second scene is placed before the first scene. Specify -1 to move the scene to the end.

Returns

Nothing.

Example

The following example moves the second scene to before the first scene:

```
fl.getDocumentDOM().reorderScene(1, 0);
```

document.resetTransformation()

Description

Resets the matrix for the current selection. Performs the same operation as using the Modify > Transform > Remove transform menu item.

Usage

```
document.resetTransformation()
```

Arguments

None.

Returns

Nothing.

Example

The following example resets the transformation matrix for the current selection:

```
fl.getDocumentDOM().resetTransformation();
```

document.revert()

Description

Reverts the specified document to its previously saved version. This method performs the same action as the File > Revert menu item.

Usage

```
document.revert()
```

Arguments

None.

Returns

Nothing.

Example

The following example reverts the current document to the previously saved version:

```
fl.getDocumentDOM().revert();
```

document.rotateSelection()

Description

Rotates the selection by a specified amount. The effect is the same as using the Free Transform tool to rotate the object.

Usage

```
document.rotateSelection( angle [, rotationPoint] )
```

Arguments

The *angle* argument is a floating point value that specifies the angle of the rotation.

The optional *rotationPoint* argument is a String that specifies which side of the bounding box to rotate. Valid values are: `"top right"`, `"top left"`, `"bottom right"`, `"bottom left"`, `"top center"`, `"right center"`, `"bottom center"`, and `"left center"`. If unspecified, the method uses the transformation point.

Returns

Nothing.

Example

The following example rotates the selection by 45 degree around the transformation point:

```
flash.getDocumentDOM().rotateSelection(45);
```

The following example rotates the selection by 45 degree around the bottom left corner:

```
fl.getDocumentDOM().rotateSelection(45, "bottom left");
```

document.save()

Description

Saves the document in its default location. This method performs the same action as the File > Save menu item.

Usage

```
document.save( [bOkToSaveAs] )
```

Arguments

If the optional *bOkToSaveAs* argument is `true` or omitted, and the file was never saved, then the Save As dialog box appears. If `bOkToSaveAs` is `false` and the file was never saved, the file is not saved.

Returns

A Boolean value: `true` if the save operation completes successfully; `false` otherwise.

Example

The following example saves the current document in its default location:

```
fl.getDocumentDOM().save();
```

document.saveAndCompact()

Description

Saves and compacts the file. This method performs the same action as the File > Save and Compact menu item.

Usage

```
document.saveAndCompact( [bOkToSaveAs] )
```

Arguments

If the optional *bOkToSaveAs* argument is `true` or omitted, and the file was never saved, then the Save As dialog box appears. If *bOkToSaveAs* is `false` and the file was never saved, the file is not saved.

Returns

A Boolean value: `true` if the save-and-compact operation completes successfully; `false` otherwise.

Example

The following example saves and compacts the current document:

```
fl.getDocumentDOM().saveAndCompact();
```

document.scaleSelection()

Description

Scales the selection by a specified amount. The effect is the same as using the Free Transform tool to scale the object.

Usage

```
document.scaleSelection( xScale, yScale [, whichCorner] )
```

Arguments

The *xScale* argument is a floating point value that specifies the amount of *x* by which to scale.

The *yScale* argument is a floating point value that specifies the amount of *y* by which to scale.

The optional *whichCorner* argument is a String value that specifies the edge about which the transformation occurs. If omitted, scaling occurs about the transformation point. Acceptable values are: `"bottom left"`, `"bottom right"`, `"top right"`, `"top left"`, `"top center"`, `"right center"`, `"bottom center"`, and `"left center"`.

Returns

Nothing.

Example

The following example expands the width of the current selection to double the original width and shrinks the height to half:

```
flash.getDocumentDOM().scaleSelection(2.0, 0.5);
```

The following example flips the selection vertically:

```
fl.getDocumentDOM().scaleSelection(1, -1);
```

The following example flips the selection horizontally:

```
fl.getDocumentDOM().scaleSelection(-1, 1);
```

The following example scales the selection vertically by 1.9 from the top center:

```
fl.getDocumentDOM().scaleSelection(1, 1.90, 'top center');
```

document.selectAll()

Description

Selects all items on the stage. Equivalent to pressing Command-A or selecting Select All from the Edit menu.

Usage

```
document.selectAll()
```

Arguments

None.

Returns

Nothing.

Example

The following example selects everything that is currently visible to the user:

```
fl.getDocumentDOM().selectAll();
```

document.selectNone()

Description

Deselects any selected items.

Usage

```
document.selectNone()
```

Arguments

None.

Returns

Nothing.

Example

The following example deselects any items that are selected:

```
fl.getDocumentDOM().selectNone();
```

document.setAlignToDocument()

Description

Sets the preferences for `align()`, `distribute()`, `match()`, and `space()` to act on the document. This method performs the same action as enabling the To Stage button in the Align panel.

Usage

```
document.setAlignToDocument( bToStage )
```

Arguments

The *`bToStage`* argument is a Boolean value that, if set to `true`, aligns objects to the Stage. If set to `false`, it does not.

Returns

Nothing.

Example

The following example enables the To Stage button in the Align panel to align objects with the Stage:

```
fl.getDocumentDOM().setAlignToDocument(true);
```

document.setCustomFill()

Description

Sets the fill settings in the toolbar. This allows for a script to set the fill settings before drawing the object. Rather than having to draw the object, then select it and change the fill settings. It also allows a script to change the toolbar fill settings.

Usage

```
document.setCustomFill( fill )
```

Arguments

The *`fill`* argument sets the Fill color object.

Returns

Nothing.

Example

The following example changes the color of the fill color chip in the toolbar, and any selected shapes, to white. The selected object must be a shape.

```
var fill = fl.getDocumentDOM().getCustomFill();
fill.color = '#FFFFFF';
fill.style = "solid";
fl.getDocumentDOM().setCustomFill(fill);
```

document.setCustomStroke()

Description

Sets the stroke settings in the toolbar, which lets a script set the stroke settings before drawing the object. It eliminates having to draw the object, then select it and change the stroke settings. It also lets a script change the toolbar stroke settings.

Usage

```
document.setCustomStroke( stroke )
```

Arguments

The *stroke* argument is a Stroke object.

Returns

Nothing.

Example

The following argument changes the stroke thickness setting in the Property inspector, and of any selected shapes:

```
var stroke = fl.getDocumentDOM().getCustomStroke();
stroke.thickness += 2;
fl.getDocumentDOM().setCustomStroke(stroke);
```

document.setElementProperty()

Description

Sets the specified Element property on selected object(s) in the document. Does nothing if there is no selection.

Usage

```
document.setElementProperty( property, value )
```

Arguments

The *property* argument is a String that specifies the name of the Element property to set.

The *value* argument specifies the value to set in the specified Element property.

For a complete list of properties and values, see the Element object. This setting does not apply to read-only properties such as `elementType`, `top`, and `left`.

Returns

Nothing.

Example

The following example sets the width of all selected objects to 100 and the height to 50:

```
fl.getDocumentDOM().setElementProperty("width", 100);
fl.getDocumentDOM().setElementProperty("height", 50);
```

document.setElementTextAttr()

Description

Sets the specified property of the selected text items to the specified value. If the optional parameters are not passed, the method sets the style of the currently selected text range, or the whole text field if no text is selected. If only *startIndex* is passed, the method sets that character's attributes. If *startIndex* and *endIndex* are passed, the method sets the attributes on the characters starting from *startIndex* up to, but not including, *endIndex*. If paragraph styles are specified, all the paragraphs that fall within the range are affected.

Usage

```
document.setElementTextAttr( attrName, attrValue [, startIndex [, endIndex]] )
```

Arguments

The *attrName* argument is a String that specifies the name of the `TextAttrs` property to change.

The *attrValue* argument is the value to which to set the `TextAttrs` property. For a list of property names and expected values, see "TextAttrs" on page 193.

The optional *startIndex* argument is an integer value that specifies the index of the first character that is affected.

The optional *endIndex* argument is an integer value that specifies the index of the last character that is affected.

Returns

A Boolean value: `true` if at least one text attribute property is changed; `false` otherwise.

Example

The following examples set the `fillColor, italic, and bold` text attributes for the selected text items:

```
var success = fl.getDocumentDOM().setElementTextAttr("fillColor", "#00ff00");
var pass = fl.getDocumentDOM().setElementTextAttr("italic", true, 10);
var ok = fl.getDocumentDOM().setElementTextAttr("bold", true, 5, 15);
```

document.setFillColor()

Description

Changes the fill color of the selection to the specified color.

Usage

```
document.setFillColor( color )
```

Arguments

The *color* argument is a color string in hexadecimal #rrggbb format (where r is red, g is green, and b is blue), a hexidecimal color value (such as, 0xff0000), or an integer color value. If set to `null`, no fill color is set, which is the same as setting the Fill color swatch in the user interface to no fill.

Returns

Nothing.

Example

The first three statements in the following example set the fill color using each of the different formats for specifying color. The fourth statement sets the fill to no fill.

```
flash.getDocumentDOM().setFillColor("#cc00cc");
flash.getDocumentDOM().setFillColor(0xcc00cc);
flash.getDocumentDOM().setFillColor(120000);
flash.getDocumentDOM().setFillColor(null);
```

document.setInstanceAlpha()

Description

Sets the opacity of the instance.

Usage

```
document.setInstanceAlpha( opacity )
```

Arguments

The *opacity* argument is an integer between 0 (transparent) and 100 (completely saturated) that adjusts the transparency of the instance.

Returns

Nothing.

Example

The following example sets the opacity of the tint to a value of 50:

```
fl.getDocumentDOM().setInstanceAlpha(50);
```

document.setInstanceBrightness()

Description

Sets the brightness for the instance.

Usage

```
document.setInstanceBrightness( brightness )
```

Arguments

The *brightness* argument is an integer that specifies brightness as a value from -100 (black) to 100 (white).

Returns

Nothing.

Example

The following example sets the brightness for the instance to a value of 50:

```
fl.getDocumentDOM().setInstanceBrightness(50);
```

document.setInstanceTint()

Description

Sets the tint for the selected instance.

Usage

```
document.setInstanceTint( color, strength )
```

Arguments

The *color* argument is a color string in hexadecimal #rrggbb format (where r is red, g is green, and b is blue), a hexidecimal color value (such as, 0xff0000), or an integer color value that specifies the color of the tint. This argument is equivalent to picking the `Color: Tint` value for a symbol in the Property Inspector.

The *strength* argument is an integer between 0 and 100 that specifies the opacity of the tint.

Returns

Nothing.

Example

The following example sets the tint for the selected instance to red with an opacity value of 50:

```
fl.getDocumentDOM().setInstanceTint(0xff0000, 50);
```

document.setSelectionBounds()

Description

Moves and resizes the selection in a single operation.

Usage

```
document.setSelectionBounds( boundingRectangle )
```

Arguments

The *boundingRectangle* argument is a rectangle that specifies the new location and size of the selection. The argument specifies location as left and top pixel locations and size as width and height.

Returns

Nothing.

Example

The following example moves the current selection to 10, 20 and resizes it to 100, 200:

```
var l = 10;
var t = 20;
fl.getDocumentDOM().setSelectionBounds({left:l, top:t, right:(100+l),
  bottom:(200+t)});
```

document.setSelectionRect()

Description

Sets the selection to the rectangle, which is always relative to the Stage. This is unlike `getSelectionRect()` in which the rectangle is relative the object being edited.

Equivalent to dragging out a rectangle with the arrow tool. An instance must be fully enclosed by the rectangle to become selected.

Note: Repeating setSelectionRect() via the History panel or menu item repeats the step previous to the setSelectionRect() operation.

Usage

```
document.setSelectionRect( rect [, bReplaceCurrentSelection] )
```

Arguments

The *`rect`* argument is a rectangle object to set as selected.

The *`bReplaceCurrentSelection`* argument is a Boolean value, that if set to `true`, replaces the current selection. If it is `false`, the method adds to the current selection. The default value, if not set, is `true`.

Returns

Nothing.

Example

In the following example, the second selection replaces the first one:

```
fl.getDocumentDOM().setSelectionRect({left:1, top:1, right:200, bottom:200});
fl.getDocumentDOM().setSelectionRect({left:364.0, top:203.0, right:508.0,
  bottom:434.0}, true);
```

In the following example the second selection is added to the first selection. This is the same as the manual operation of holding the Shift key down and selecting a second object.

```
fl.getDocumentDOM().setSelectionRect({left:1, top:1, right:200, bottom:200});
fl.getDocumentDOM().setSelectionRect({left:364.0, top:203.0, right:508.0,
  bottom:434.0}, false);
```

document.setStroke()

Description

Sets the color, width, and style of the selected strokes.

Usage

```
document.setStroke( color, size, strokeType )
```

Arguments

The *`color`* argument is a color string in hexadecimal #rrggbb format (where r is red, g is green, and b is blue), a hexidecimal color value (such as, 0xff0000), or an integer color value.

The *`size`* argument is a floating point value that specifies the new stroke size for the selection.

The *`strokeType`* argument is a string that specifies the new type of stroke for the selection. Valid values are `"hairline"`, `"solid"`, `"dashed"`, `"dotted"`, `"ragged"`, `"stipple"`, and `"hatched"`.

Returns

Nothing.

Example

The following example sets the color of the stroke to red, the size to 3.25, and the type to dashed:

```
fl.getDocumentDOM().setStroke("#ff0000", 3.25, "dashed");
```

document.setStrokeColor()

Description

Changes the stroke color of the selection to the specified color.

Usage

```
document.setStrokeColor( color )
```

Arguments

The *color* argument is a color string in hexadecimal #rrggbb format (where r is red, g is green, and b is blue), a hexidecimal color value (such as, 0xff0000), or an integer color value.

Returns

Nothing.

Example

The three statements in the following example set the stroke color using each of the different formats for specifying color:

```
flash.getDocumentDOM().setFillColor("#cc00cc");
flash.getDocumentDOM().setFillColor(0xcc00cc);
flash.getDocumentDOM().setFillColor(120000);
```

document.setStrokeSize()

Description

Changes the stroke size of the selection to the specified size.

Usage

```
document.setStrokeSize( size )
```

Arguments

The *size* argument is a floating point value from 0.25 to 10 that specifies the stroke size. The method ignores precision greater than two decimal places.

Returns

Nothing.

Example

The following example changes the stroke size for the selection to 5:

```
fl.getDocumentDOM().setStrokeSize(5);
```

document.setStrokeStyle()

Description

Sets the stroke for the current selection.

Usage

```
document.setStrokeStyle( strokeType )
```

Arguments

The *strokeType* argument is a String that specifies the stroke style for the current selection. Valid values are `"hairline"`, `"solid"`, `"dashed"`, `"dotted"`, `"ragged"`, `"stipple"`, and `"hatched"`.

Returns

Nothing.

Example

The following example changes the stroke style for the selection to `"dashed"`:

```
fl.getDocumentDOM().setStrokeStyle("dashed");
```

document.setTextRectangle()

Description

Changes the bounding rectangle for the selected text item to the specified size. This method causes the text to reflow inside the new rectangle; the text item is not scaled or transformed. If the text is horizontal and static, the method only takes into account the width of the rectangle (the height is automatically computed to fit all the text). If the text is vertical, the method only takes into account the height of the rectangle (the width is automatically computed to fit all the text). If the text is dynamic or input, the method is limited by the width and height of the rectangle, but the text field is constrained to fit all the text.

Usage

```
document.setTextRectangle( boundingRectangle )
```

Arguments

The *boundingRectangle* argument is a text rectangle object that specifies the new size within which the text item should flow.

Returns

A Boolean value: `true` if the size of at least one text field is changed; `false` otherwise.

Example

The following example changes the size of the bounding text rectangle to the specified dimensions:

```
var success = fl.getDocumentDOM().setTextRectangle({left:0, top:0, right:50,
  bottom:200});
```

document.setTextSelection()

Description

Sets the text selection of the currently selected text field to the values specified by the *startIndex* and *endIndex* values. Text editing is activated, if it isn't already.

Usage

```
document.setTextSelection( startIndex, endIndex )
```

Arguments

The *startIndex* argument is an integer that specifies the position of the first character to select. The first character position is 0 (zero).

The *endIndex* argument is an integer that specifies the end position of the selection up to, but not including, *endIndex*. The first character position is 0 (zero).

Returns

A Boolean value: `true` if the method can successfully set the text selection; `false` otherwise.

Example

The following example selects the text from the sixth character through the twenty-fifth character:

```
var success = fl.document.setTextSelection(5, 25);
```

document.setTextString()

Description

Inserts a string of text. If the optional parameters are not passed, the existing text selection is replaced; if the text object isn't currently being edited, the whole text string is replaced. If only *startIndex* is passed, the string passed is inserted at this position. If *startIndex* and *endIndex* are passed, the string passed replaces the segment of text starting from *startIndex* up to, but not including, *endIndex*.

Usage

```
document.setTextString( text [, startIndex [, endIndex]] )
```

Arguments

The *text* argument is a string of the characters to insert in the text field.

The optional *startIndex* argument is an integer that specifies first character to replace. The first character position is 0 (zero).

The optional *endIndex* argument is an integer that specifies the last character to replace. The first character position is 0 (zero).

Returns

A Boolean value: `true` if the text of at least one text string is set; `false` otherwise.

Example

The following example replaces the current text selection with “Hello World”:

```
var success = fl.getDocumentDOM().setTextString("Hello World!");
```

The following example inserts “hello” at position 6 of current text selection:

```
var pass = fl.getDocumentDOM().setTextString("hello", 6);
```

The following example inserts “Howdy” starting at position 2 and up to, but not including, position 7 of the current text selection:

```
var ok = fl.getDocumentDOM().setTextString("Howdy", 2, 7);
```

document.setTransformationPoint()

Description

Moves the transformation point of the current selection.

Usage

```
document.setTransformationPoint( transformationPoint )
```

Arguments

The *transformationPoint* argument is a pair of floating point numbers that specifies values for each of the following elements:

- Shapes: *transformationPoint* is set relative to document. 0,0 is the same as the Stage (upper left corner).
- Symbols: *transformationPoint* is set relative to the symbol’s registration point. 0,0 is located at the registration point.
- Text: *transformationPoint* is set relative to the text field. 0,0 is the upper left corner of text field.
- Bitmaps/videos: *transformationPoint* is set relative to bitmap/video. 0,0 is the upper left corner of the bitmap or video.
- Groups: *transformationPoint* is set relative to document. 0,0 is the same as the Stage (upper left corner)

Returns

Nothing.

Example

The following example sets the transformation point of the current selection to 100, 200:

```
fl.getDocumentDOM().setTransformationPoint({x:100, y:200});
```

document.skewSelection()

Description

Skews the selection by a specified amount. The effect is the same as using the Free Transform tool to skew the object.

Usage

```
document.skewSelection( xSkew, ySkew [, whichEdge] )
```

Arguments

The *xSkew* argument is a floating point number that specifies the amount of *x* by which to skew, measured in degrees.

The *ySkew* argument is a floating point number that specifies the amount of *y* by which to skew, measured in degrees.

The optional *whichEdge* argument is a String that specifies the edge where the transformation occurs; if omitted, skew occurs at the transformation point. Valid values are: `"top center"`, `"right center"`, `"bottom center"`, and `"left center"`.

Returns

Nothing.

Example

The following examples skew the selected object by 2.0 vertically and 1.5 horizontally. The second example transforms the object at the top center edge:

```
flash.getDocumentDOM().skewSelection(2.0, 1.5);
flash.getDocumentDOM().skewSelection(2.0, 1.5, "top center");
```

document.smoothSelection()

Description

Smooths the curve of each selected fill outline or curved line. This method performs the same action as the Smooth button in the Tools panel.

Usage

```
document.smoothSelection()
```

Arguments

None.

Returns

Nothing.

Example

The following example smooths the curve of the current selection:

```
fl.getDocumentDOM().smoothSelection();
```

document.space()

Description

Spaces the objects in the selection evenly.

Usage

```
document.space( direction [, bUseDocumentBounds] )
```

Arguments

The direction argument is a String that specifies the direction in which to space the objects in the selection. Valid values for direction are `"horizontal"` or `"vertical"`.

The optional *`bUseDocumentBounds`* argument is a Boolean value that, when set to `true`, spaces the objects to the document bounds. Otherwise, the method uses the bounds of the selected objects. The default is `false`.

Returns

Nothing.

Example

The following example spaces the objects horizontally, relative to the Stage:

```
fl.getDocumentDOM().space("horizontal",true);
```

The following example spaces the objects horizontally, relative to each other:

```
fl.getDocumentDOM().space("horizontal");
```

The following example spaces the objects horizontally, relative to each other, with *`bUseDcoumentBounds`* expressly set to `false`:

```
fl.getDocumentDOM().space("horizontal",false);
```

document.straightenSelection()

Description

Straightens the current selection. This method performs the same action as the Straighten button in the Tools panel.

Usage

```
document.straightenSelection()
```

Arguments

None.

Returns

Nothing.

Example

The following example straightens the curve of the current selection:

```
fl.getDocumentDOM().straightenSelection();
```

document.swapElement()

Description

Swaps the current selection with the specified one. The selection must contain a graphic, button, movie clip, video, or bitmap. Displays an error message if no object is selected or the given object could not be found.

Usage

```
document.swapElement( name )
```

Arguments

The *name* argument is a String that specifies the name of the library item to use.

Returns

Nothing.

Example

The following example swaps the current selection with `Symbol 1` from the library:

```
fl.getDocumentDOM().swapElement('Symbol 1');
```

document.testMovie()

Description

Executes a Test Movie operation on the document.

Usage

```
document.testMovie()
```

Arguments

None.

Returns

Nothing.

Example

The following example tests the movie for the current document:

```
fl.getDocumentDOM().testMovie();
```

document.testScene()

Description

Executes a Test Scene operation on the current scene of the document.

Usage

```
document.testScene()
```

Arguments

None.

Returns

Nothing.

Example

The following example tests the current scene in the document:

```
fl.getDocumentDOM().testScene();
```

document.traceBitmap()

Description

Performs a trace bitmap on the current selection. This method performs the same action as the Modify > Bitmap > Trace Bitmap menu item.

Usage

```
document.traceBitmap( threshold, minimumArea, curveFit, cornerThreshold )
```

Arguments

threshold, *minimumArea*, *curveFit*, *cornerThreshold*

The *threshold* argument is an integer that controls the number of colors in your traced bitmap. Valid values are integers between 0 and 500.

The *minimumArea* argument is an integer that specifies the radius measured in pixels. Valid values are integers between 1 and 1000.

The *curveFit* argument is a String that specifies how smoothly outlines are drawn. Valid values are: `"pixels"`, `"very tight"`, `"tight"`, `"normal"`, `"smooth"`, and `"very smooth"`.

The *cornerThreshold* argument is a String that is similar to *curveFit*, but it pertains to the corners of the bitmap image. Valid values are: `"many corners"`, `"normal"`, and `"few corners"`.

Returns

Nothing.

Example

The following example traces the selected bitmap, using the specified parameters:

```
fl.getDocumentDOM().traceBitmap(0, 500, 'normal', 'normal');
```

document.transformSelection()

Description

Performs a general transformation on the current selection by applying the matrix specified in the arguments. For more information, see the `element.matrix` property.

Usage

```
document.transformSelection( a, b, c, d)
```

Arguments

The *a* argument is a floating point number that specifies the (0,0) element of the transformation matrix.

The *b* argument is a floating point number that specifies the (0,1) element of the transformation matrix.

The *c* argument is a floating point number that specifies the (1,0) element of the transformation matrix.

The *d* argument is a floating point number that specifies the (1,1) element of the transformation matrix.

Returns

Nothing.

Example

The following example stretches the selection by a factor of 2 in the x direction:

```
fl.getDocumentDOM().transformSelection(2.0, 0.0, 0.0, 1.0);
```

document.unGroup()

Description

Ungroups the current selection.

Usage

```
document.unGroup()
```

Arguments

None.

Returns

Nothing.

Example

The following example ungroups the elements in the current selection:

```
fl.getDocumentDOM().unGroup();
```

document.unlockAllElements()

Description

Unlocks all locked objects on the frame that currently appears.

Usage

```
document.unlockAllElements()
```

Arguments

None.

Returns

Nothing.

Example

The following example unlocks all locked objects in the current frame:

```
fl.getDocumentDOM().unlockAllElements();
```

document.xmlPanel()

Description

Posts a XMLUI dialog box.

Usage

```
document.xmlPanel( fileURI )
```

Arguments

The *fileURI* argument is a String that specifies the path, expressed as a file://URL, to the XML file defining the controls in the panel. Need to use the full path.

Returns

An object that has properties defined for all controls defined in the XML file. All properties are returned as strings. The returned object will have one predefined property named `"dismiss"` that will have the string value `"accept"` or `"cancel"`.

Example

The following example loads the Test.xml file and displays each property contained within it:

```
var obj = fl.getDocumentDOM().xmlPanel(fl.configURI + "Commands/Test.xml");
for (var prop in obj) {
  fl.trace("property " + prop + " = " + obj[prop]);
}
```

Properties

Property	Data type	Value

`accName`	string	Equivalent to the Name field in the Accessibility panel. Screen readers identify objects by reading the name aloud.
		The following example gets the accessibility name of the movie:
		`var theName = fl.getDocumentDOM().accName;`
		The following example sets the accessibility name of the movie to `"Main Movie"`:
		`fl.getDocumentDOM().accName = “Main Movie”;`

`autoLabel`	Boolean	This is equivalent to the Auto Label check box on the Accessibility panel.
		Instructs Flash to automatically label objects on the Stage with the text associated with them.
		The following example gets the value of the `autoLabel` property:
		`var isAutoLabel = fl.getDocumentDOM().autoLabel;`
		The following example sets the autoLabel property to `true`, instructing Flash to automatically label objects on the Stage:
		`fl.getDocumentDOM().autoLabel = true;`

Property	Data type	Value
backgroundColor	color	Color representing the background. The following example sets the background color to black. `fl.getDocumentDOM().backgroundColor = '#000000';`
currentPublishProfile	string	Sets and gets the name of the active publish profile for document. The following example displays the name of the current publish profile. `alert(fl.getDocumentDOM().currentPublishProfile);`

currentTimeline	int	Index of the active Timeline. Use get`Timeline()` to get the active Timeline in the document. Setting the active Timeline by changing the value of this property is possible. The effect is equivalent to calling `editScene()`. The only difference is that you don't get an error message if the index of the Timeline is not valid (the property is simply not set, which causes silent failure).
		The following example displays the index of the current Timeline.
		`var myCurrentTL = fl.getDocumentDOM().currentTimeline;`
		`fl.trace("The index of the current timeline is: "+ myCurrentTL);`

description	string	Equivalent to the Description field on the Accessibility panel. The description is read by the screen reader.
		The following example gets the description of the movie:
		`var theDescription = fl.getDocumentDOM().description;`
		The following example sets the description of the movie:
		`fl.getDocumentDOM().description= "This is the main movie";`

forceSimple	Boolean	Enables and disables the children of the object to be accessible. This is equivalent to the inverse logic of the Make Child Objects Accessible setting in the Accessibility panel.
		If `forceSimple` is `true`, it is the same as the Make Child Object Accessible option being unchecked.
		If `forceSimple` is `false`, it is the same as the Make Child Object Accessible option being checked.
		The following example sets the *`areChildrenAccessible`* variable to the NOT value of the `forceSimple` property (to determine if the children of the movie are accessible):
		`var areChildrenAccessible = !fl.getDocumentDOM().forceSimple;`
		The following example sets the `forceSimple` property to allow the children of the movie to be accessible:
		`fl.getDocumentDOM().forceSimple = false;`

frameRate	float	The number of frames per second; the default is 12. The following example sets the frame rate to 25.5 frames per second. `fl.getDocumentDOM().frameRate = 25.5;`

Property	Data type	Value
`height`	int	Height of the document and Stage in pixels. The following example sets the height of the stage to 400 pixels. `fl.getDocumentDOM().height = 400;`

`library`
read only
	object	The Library object for the document.
		The following example gets the library for currently focused document:
		`var myCurrentLib = fl.getDocumentDOM().library;`
		The following example gets the library for an open external library:
		`var externalLib = fl.documents[1].library;`

`livePreview`	Boolean	If `true`, the document should display components as they will appear in the Flash Player. The default is `true`.
`name` **read only**	string	Name of the document. The following example sets the variable *`fileName`* to the file name of the first document: `var fileName = flash.documents[0].name;`
`path` **read only**	string	Path of the document (the value is `"undefined"` if the document was never saved). The following example sets the *`filePath`* variable to the file path of the first document: `var filePath = flash.documents[0].path;`

`publishProfiles`
read only
	array	An array of the publish profile names for the document.
		The following example displays the names of the publish profiles for the document:
		`var myPubProfiles = fl.getDocumentDOM().publishProfiles;`
		`for (var i=0; i < myPubProfiles.length; i++){`
		`  fl.trace(myPubProfiles[i]);`
		`}`

`screenOutline`
read only
	object	The current `screenOutline` object for the document. Before accessing the object for the first time, be sure to use `document.allowScreens()` to determine whether the property exists.
		The following example displays the array of values in the `screenOutline` property:
		`var myArray = new Array();`
		`for(var i in fl.getDocumentDOM().screenOutline) {`
		`myArray.push(" "+i+" :`
		`  "+fl.getDocumentDOM().screenOutline[i]) ;`
		`}`
		`fl.trace("Here is the property dump for screenOutline::`
		`  "+myArray);`

Property	Data type	Value

`selection`	array	Array of the selected objects in the document. If nothing is selected, returns an array of length zero. If no document is open, returns `null`.
		The following example sets the *`selectObj`* variable to the value of the `selection` property, which contains an array of the currently selected objects in the document:
		`var selectObj = fl.getDocumentDOM().selection;`
		If you select a text field and a shape on the stage, *`selectObj`* would contain [object Text], [object Shape]:
		The following example assigns all elements on frame 10 to the current selection:
		`fl.getDocumentDOM().getTimeline().currentFrame = 10;`
		`fl.getDocumentDOM().selection =`
		`fl.getDocumentDOM().getTimeline().layers[0].frames[10].elements;`

`silent`	Boolean	Enables or disables the object to be accessible. This is equivalent to the inverse logic of the Make Movie Accessible setting in the Accessibility panel.
		If `silent` is `true`, it is the same as the Make Movie Accessible option being unchecked
		If `silent` is `false`, it is the same as the Make Movie Accessible option being checked.
		The following example sets the *`isSilent`* variable to the value of the `silent` property:
		`var isSilent = fl.getDocumentDOM().silent;`
		The following example sets the `silent` property to `false`, indicating that the movie is accessible:
		`fl.getDocumentDOM().silent = false;`

`timelines`
read only
	array	An array of Timeline objects. For more information, see the `Timeline` object.
		The following example gets the array of current Timelines in the *`curTimelines`* variable and then displays them.
		`var i = 0;`
		`var curTimelines = fl.getDocumentDOM().timelines;`
		`while(i < fl.getDocumentDOM().timelines.length){`
		`alert(curTimelines[i].name);`
		`++i;`
		`}`

Property	Data type	Value
`viewMatrix` **read only**	object	A Matrix object. The `viewMatrix` is used to transform from object space to document space when the document is in edit mode. The mouse location, as a tool receives it, is relative to the object that is currently being edited. For example, if you create a symbol and double click to edit it, then draw with the polyStar tool, the point (0,0) will be at the registration point of the symbol. However, the drawingLayer object expects values in document space, so if you draw a line from (0,0) using the drawingLayer, it will start at the upper left hand corner of the stage. The view matrix provides a way to transform from the space of the object being edited to document space. The following example gets the value of the `viewMatrix` property. `var mat = fl.getDocumentDOM().viewMatrix;`
`width`	int	Width of the document and Stage in pixels. The following example sets the width of the stage to 400 pixels. `fl.getDocumentDOM().width= 400;`

drawingLayer

Description

The drawingLayer object is accessible from JavaScript as a child of the `Flash` object. The drawingLayer is used for extensible tools when they want to do some temporary drawing while dragging; for example, a selection style marquee.

Methods

The following methods are available for the `drawingLayer` object:

```
drawingLayer.beginDraw()
drawingLayer.beginFrame()
drawingLayer.cubicCurveTo()
drawingLayer.curveTo()
drawingLayer.drawPath()
drawingLayer.endDraw()
drawingLayer.endFrame()
drawingLayer.lineTo()
drawingLayer.moveTo()
drawingLayer.newPath()
drawingLayer.setColor()
```

drawingLayer.beginDraw()

Description

Puts Flash into drawing mode. The drawing mode is used when you want to do some temporary drawing while the mouse is down. This method is typically used only when creating extensible tools.

Usage

```
drawingLayer.beginDraw( [persistentDraw] )
```

Arguments

The optional *`persistentDraw`* argument is a Boolean value that if set to `true` indicates that the drawing appearing in the last frame will remain on the Stage until a new *`beginDraw()`* or *`beginFrame()`* call is made. Note that here, the term "frame" is used in the context of defining where you start and where you end drawing. It does not reference Timeline frames. For example, when the user draws a rectangle, the outline of the shape is previewed while the mouse is being dragged. If you want that preview shape to remain after the user releases the mouse button, set *`persistentDraw`* to `true`.

Returns

Nothing.

Example

```
fl.drawingLayer.beginDraw();
```

drawingLayer.beginFrame()

Description

This call should precede any other drawingLayer calls. Everything drawn between `drawingLayer.beginFrame()` and an `drawingLayer.endFrame()` remains on the Stage until you call the next *`beginFrame`* and *`endFrame`*. A frame is how you define where you start and where you end drawing. Note that here, the term "frame" is used in the context of defining where you start and where you end drawing. It does not reference Timeline frames. This method is typically used only when creating extensible tools.

Usage

```
drawingLayer.beginFrame()
```

Arguments

None.

Returns

Nothing.

Example

```
fl.drawingLayer.beginFrame();
```

drawingLayer.cubicCurveTo()

Description

Draws a cubic curve from the current pen location using the arguments as the coordinates of the cubic segment. Typically used only when creating extensible tools.

Usage

```
drawingLayer.cubicCurveTo( x1Ctrl, y1Ctrl, x2Ctl, y2Ctl, xEnd, yEnd )
```

Arguments

The *`x1Ctl`* argument is a floating point value that is the *x* location of the first control point.

The *`y1Ctl`* argument is a floating point value that is the *y* location of the first control point.

The *`x2Ctl`* argument is a floating point value that is the *x* position of the middle control point.

The *`y2Ctl`* argument is a floating point value that is the *y* position of the middle control point.

The *`xEnd`* argument is a floating point value that is the *x* position of the end control point.

The *`yEnd`* argument is a floating point value that is the *y* position of the end control point.

Returns

Nothing.

Example

```
fl.drawingLayer.cubicCurveTo(0, 0, 1, 1, 2, 0);
```

drawingLayer.curveTo()

Description

Draws a quadratic curve segment starting at the current drawing position and ending at a specified point. This method is typically used only when creating extensible tools.

Usage

```
drawingLayer.curveTo( xCtl, yCtl, xEnd, yEnd )
```

Arguments

The *xCtl* argument is a floating point value that is the *x* position of the control point.

The *yCtl* argument is a floating point value that is the *y* position of the control point.

The *xEnd* argument is a floating point value that is the *x* position of the end control point.

The *yEnd* argument is a floating point value that is the *y* position of the end control point.

Returns

Nothing.

Example

```
fl.drawingLayer.curveTo(0, 0, 2, 0);
```

drawingLayer.drawPath()

Description

Draws the path specified by the *path* argument. Typically used only when creating extensible tools.

Usage

```
drawingLayer.drawPath( path )
```

Arguments

The *path* argument is a Path object to draw.

Returns

Nothing.

Example

```
fl.drawingLayer.drawPath(pathObject);
```

drawingLayer.endDraw()

Description

Exits the drawing mode. The drawing mode is used when you want to do some temporary drawing while the mouse is down. This method is typically used only when creating extensible tools.

Usage

```
drawingLayer.endDraw()
```

Arguments

None.

Returns

Nothing.

Example

```
fl.drawingLayer.endDraw();
```

drawingLayer.endFrame()

Description

Everything drawn between `drawingLayer.beginFrame()` and `drawingLayer.endFrame()` remains on the Stage until you call the `drawingLayer.beginFrame()` and `drawingLayer.endFrame()`. This method is typically used only when creating extensible tools.

Usage

```
drawingLayer.endFrame()
```

Arguments

None.

Returns

Nothing.

Example

```
fl.drawingLayer.endFrame();
```

drawingLayer.lineTo()

Description

Draws a line from the current drawing position to the point (*x, y*). Typically used only when creating extensible tools.

Usage

```
drawingLayer.lineTo( x, y )
```

Arguments

x, *y*

The *x* argument is a floating point value that is the *x* coordinate of the end point of the line to draw.

The *y* argument is a floating point value that is the *y* coordinate of the end point of the line to draw.

Returns

Nothing.

Example

```
fl.drawingLayer.lineTo(20, 30);
```

drawingLayer.moveTo()

Description

Sets the current drawing position. Typically used only when creating extensible tools.

Usage

```
drawingLayer.moveTo( x, y )
```

Arguments

The *x* argument is a floating point value that specifies the *x* coordinate of the position at which to start drawing.

The *y* argument is a floating point value that specifies the *y* coordinate of the position at which to start drawing.

Returns

Nothing.

Example

```
fl.drawingLayer.moveTo(10, 15);
```

drawingLayer.newPath()

Description

Returns a new Path object. This method is typically used only when creating extensible tools.

Usage

```
drawingLayer.newPath()
```

Arguments

None.

Returns

A Path object.

Example

```
fl.drawingLayer.newPath();
```

drawingLayer.setColor()

Description

Sets the color of subsequently drawn data. Applies only to persistent data. In order to use this method, the argument passed to `drawingLayer.beginDraw()` must be set to `true`. This method is typically used only when creating extensible tools.

Usage

```
drawingLayer.setColor( color )
```

Arguments

The *color* argument is a color that is specified by a string, integer, or hexadecimal value.

Returns

Nothing.

Properties

This object has no unique properties.

Edge

Description

The Edge object represents an edge of an object on the Stage.

Methods

The following methods are available for the Edge object:

```
edge.getControl()
edge.getHalfEdge()
edge.setControl()
edge.splitEdge()
```

edge.getControl()

Description

Get a point object set to the location of the specified control point of the edge.

Usage

```
edge.getControl(i)
```

Arguments

The argument `i` is an integer that specifies which control point of the edge to return. Use the value zero for the first control point, 1 for the middle control point, and 2 for the end control point. If the Edge property `isLine` is `true`, the middle control point will be set to the midpoint of the segment joining the beginning and ending control points.

Returns

The specified control point.

Example

```
var shape = fl.getDocumentDOM().selection[0];
var pt = shape.edges[0].getControl(0);
```

edge.getHalfEdge()

Description

Return a HalfEdge object.

Usage

```
edge.getHalfEdge( index )
```

Arguments

The *index* argument is an integer that specifies which half edge to return. The value of index must be either 0 (zero) for the first half edge or 1 (one) for the second half edge.

Returns

A HalfEdge object.

Example

```
var shape = fl.getDocumentDOM().selection[0];
var edge = shape.edges[0];
var hEdge0 = edge.getHalfEdge(0);
var hEdge1 = edge.getHalfEdge(1);
```

edge.setControl()

Description

Sets the position of the control point of the edge. You must call `shape.beginEdit()` before using this method.

Usage

```
edge.setControl( index, x, y )
```

Arguments

The *index* argument specifies which control point to set. Use values 0, 1, or 2 to specify the beginning, middle, and end control points, respectively.

The *x* argument is a floating point value that specifies the horizontal location of the control point. If the Stage is in Edit or Edit-in-place mode, the point coordinate is relative to the object being edited. Otherwise, the point coordinate is relative to the Stage.

The *y* argument is a floating point value that specifies the vertical location of the control point. If the Stage is in Edit or Edit-in-place mode, the point coordinate is relative to the object being edited. Otherwise, the point coordinate is relative to the Stage.

Returns

Nothing.

Example

```
x = 0; y = 1;
var shape = fl.getDocumentDOM().selection[0];
shape.beginEdit();
shape.edges[0].setControl(0, x, y);
shape.endEdit();
```

edge.splitEdge()

Description

Splits the edge into two pieces. You must call `shape.beginEdit()` before using this method.

Usage

```
edge.splitEdge( t )
```

Arguments

The *t* argument is a floating point value between 0 and 1 that specifies where to split the edge.

Returns

Nothing.

Example

```
var shape = fl.getDocumentDOM().selection[0];
shape.edges[0].splitEdge( 0.5 ); // split the edge in half
```

Properties

Property	Data type	Value

`id`
read only
	int	A unique identifier for the edge.
		`var shape = fl.getDocumentDOM().selection[0];`
		`my_shape_id = shape.edges[0].id;`

`isLine`
read only
	int	Has a value of 0 or 1. A value of 1 indicates that the edge is a straight line. In that case, the middle control point bisects the line joining the two end points.
		`var shape = fl.getDocumentDOM().selection[0];`
		`fl.trace(shape.edges[0].isLine);`

Effect

Description

This is a single effect descriptor object. The `activeEffect` and the `effects` properties of the flash object contain this type of object. The Effect object represents an instance of a timeline effect.

Methods

This object has no unique methods.

Properties

Property	Data type	Value
`effectName` **read only**	string	Each effect must be uniquely named, and this name will appear in the Context menu for effects. For example, to interrogate the current effect name, use `fl.activeEffect.effectName`.
`groupName` **read only**	string	Name of the effect group used for the hierarchical context menu for effects. It can also be blank, which causes it to appear ungrouped at the top level of the context menu. For example, to get the group name for the current effect, use `fl.activeEffect.groupName`.
`sourceFile` **read only**	string	The name of JSFL effect source file. This string is used to bind an XML parameter file to its JSFL effect implementation. This XML parameter is required. For example, to get the name of the JSFL effect source file, use `fl.activeEffect.sourceFile`.
`symbolType` **read only**	string	The type of symbol to create during the initial application of effect. If not specified, it will default to graphic. The supported types are: "graphic", "movie clip", and "button". For example, to get the default symbol type for the current effect, use `fl.activeEffect.symbolType`.

User defined Effect parameters–defined in XML	string	These are the user-defined properties specified in the same XML file, which holds the `effectName` and `sourceFile` properties. These specify which UI elements should be created (such as edit fields, check boxes, and list boxes), which is controlled by the `type` attribute. You can specify labels that will appear with the control in addition to default values.
		`var ef = fl.activeEffect;`
		`duration = ef.dur;`

`useXMLToUI`	Boolean	This property allows you to override the default behavior of using XMLUI to construct a dialog box that consists of one or more controls. The default value is `true`. If set to `false`, the standard XMLUI dialog box will not be posted and you are responsible for posting a UI.
		`function configureEffect() {`
		`  // this effect does its own UI.`
		`  fl.activeEffect.useXMLToUI = false;`
		`}`

Element

Description

Everything that appears on the Stage is of the type Element.

Methods

The following methods are available for the Element object:

```
element.getPersistentData()
element.hasPersistentData()
element.removePersistentData()
element.setPersistentData()
```

element.getPersistentData()

Description

Retrieves the value of the data specified by the *name* argument. The type of data depends on the type of the data that was stored (see `element.setPersistentData()`). Only symbols and bitmaps support persistent data.

Usage

```
element.getPersistentData( name )
```

Arguments

The *name* argument is a String that identifies the data to be returned.

Returns

The data specified by the *name* argument. If no name has been stored with the element, the method returns `"undefined"`.

Example

```
var elt = fl.getDocumentDOM().getTimeline().layers[0].frames[0].elements[0];
elt.setPersistentData("myData","integer", 12);
if (elt.hasPersistentData("myData")){
     var data = elt.getPersistentData("myData");
     fl.trace("myData = "+ data);
}
```

element.hasPersistentData()

Description

Determines whether the specified data has been attached to the specified element. Only symbols and bitmaps support persistent data.

Usage

```
element.hasPersistentData( name )
```

Arguments

The *name* argument is a String that specifies the name of the data item to test.

Returns

A Boolean value: `true` if the specified data is attached to the object; `false` otherwise.

Example

```
var elt = fl.getDocumentDOM().getTimeline().layers[0].frames[0].elements[0];
elt.setPersistentData("myData","integer", 12);
if (elt.hasPersistentData("myData")){
     var data = elt.getPersistentData("myData");
     fl.trace("myData = "+ data);
}
```

element.removePersistentData()

Description

Removes any persistent data with the specified name that has been attached to the object. Only symbols and bitmaps support persistent data.

Usage

```
element.removePersistentData( name )
```

Arguments

The *name* argument is a String that specifies the name of the data to remove.

Returns

Nothing.

Example

```
var elt = fl.getDocumentDOM().getTimeline().layers[0].frames[0].elements[0];
if (elt.hasPersistentData("myData")){
  elt.removePersistentData( "myData" );
}
```

element.setPersistentData()

Description

Stores data with an element. The data is available when the FLA file containing the element is reopened. Only symbols and bitmaps support persistent data.

Usage

```
element.setPersistentData( name, type, value )
```

Arguments

The *name* argument is a String that specifies the name to associate with the data. This name is used to retrieve the data.

The *type* argument is a String that defines the type of the data. The allowable values are `"integer"`, `"integerArray"`, `"double"`, `"doubleArray"`, `"string"`, and `"byteArray"`.

The *value* argument specifies the value to associate with the object. The data type of *value* depends on the value of the *type* argument. The specified value should be appropriate to the data type specified by the *type* argument.

Returns

Nothing.

Example

```
fl.getDocumentDOM().getTimeline().layers[0].frames[0].elements[0]
  .setPersistentData( "myData", "integer", 12 );
```

Properties

Property	Data type	Value

`depth`
read only
	int	Has a value greater than 0 for the depth of the object in the view. The drawing order of objects on the Stage specifies which one is on top of the others. Object order can also be managed with the Modify › Arrange menu item.
		`var elt = fl.getDocumentDOM().getTimeline().layers[0].frames[0].elements[0];`
		`var depth = elt.depth;`

`elementType`
read only
	string	Type of element. Valid values are: "shape", "text", "instance" , "shapeObj". A "shapeObj" is created with an extensible tool.
		`// place a movie clip on first frame top layer,`
		`// then run this line of script`
		`var eType = fl.getDocumentDOM().getTimeline().layers[0].frames[0].elements[0].elementType; //eType = instance`

`height`	float	Height of the element. `fl.getDocumentDOM().getTimeline().layers[0].frames[0].elements[0].height = 100;`

`left`
read only
	float	Left side of the element. Use `document.setSelectionBounds()` or `document.moveSelectionBy()` to set this property.
		`//Select an element on the stage and then run this script`
		`var sel = fl.getDocumentDOM().selection[0];`
		`fl.getDocumentDOM().moveSelectionBy({x:100, y:0});`
		`fl.trace("Left = "+sel.left);`

`locked`	Boolean	`true` if the element is locked; `false` otherwise. If the value of `elementType` is "shape", this property is ignored.
		`// lock the first element in first frame, top layer`
		`// similar to Modify > Arrange > Lock:`
		`fl.getDocumentDOM().getTimeline().layers[0].frames[0].elements[0].locked = true;`

`matrix`	object	The `matrix` has properties `a, b, c, d, tx`, and `ty`. `a, b, c, d` are floating point; `tx` and `ty` are coordinates.
		`// move an object by 10 pixels in x and 10 pixels in y:`
		`var mat = fl.getDocumentDOM().getTimeline().layers[0].frames[0].elements[0].matrix;`
		`mat.tx += 10;`
		`mat.ty += 10;`
		`fl.getDocumentDOM().getTimeline().layers[0].frames[0].elements[0].matrix = mat;`

Property	Data type	Value

`name`	string	Name of the element, normally referred to as the Instance name. If type is "shape", the name is ignored.
		`// set the Instance name of the first element in frame 1,`
		`// top layer to 'clip_mc'`
		`fl.getDocumentDOM().getTimeline().layers[0].frames[0].elements[0].name = "clip_mc";`

`top`
read only
	float	Top side of the element. Use `document.setSelectionBounds()` or `document.moveSelectionBy()` to set this property.
		`//Select an element on the stage and then run this script`
		`var sel = fl.getDocumentDOM().selection[0];`
		`fl.getDocumentDOM().moveSelectionBy({x:0, y:100});`
		`fl.trace("Top = "+sel.top);`

`width`	float	Width of the element. `fl.getDocumentDOM().getTimeline().layers[0].frames[0].elements[0].width= 100;`

EmbeddedVideoInstance

Description

The EmbeddedVideoInstance object is a subclass of Instance. There are no unique methods or properties of EmbeddedVideoInstance.

Fill

Description

This object contains all the properties of the Fill color setting of the Toolbar or of a selected shape. To retrieve a Fill object, use `document.getCustomFill()`.

Methods

This object has no unique methods.

Properties

Property	Data type	Value

`color`	string	A color string in hexadecimal format, such as #rrggbb, or an integer containing the value.
		`var fill = fl.getDocumentDOM().getCustomFill();`
		`fill.color = '#FFFFFF';`
		`fl.getDocumentDOM().setCustomFill( fill );`

`colorArray`	array	An array of colors in gradient. Available only if the value of the `fill.style` property is either `"radialGradient"` or `"linearGradient"`.
		`var fill = fl.getDocumentDOM().getCustomFill();`
		`if(fill.style == "linearGradient" || fill.style == "radialGradient")`
		`  alert(fill.colorArray);`

`posArray`	array	An array of integers, all in the range 0 ... 255, indicating the position of the corresponding color. Available only if the value of the `fill.style` property is either `"radialGradient"` or `"linearGradient"`.
		`var fill = fl.getDocumentDOM().getCustomFill();`
		`fill.style = "linearGradient";`
		`fill.colorArray = [ 0x00ff00, 0xff0000, 0x0000ff ];`
		`fill.posArray  = [0,  100, 200];`
		`fl.getDocumentDOM().setCustomFill( fill );`

`style`	string	Represents the fill style. Valid values are: `"solid"`, `"linearGradient"`, or `"radialGradient"`. This property will return the value of `"noFill"` if object has no fill. For example, if the selected object has no fill, then `fl.getDocumentDOM().getCustomFill().style` returns the value `"noFill"`.
		If set to `"linearGradient"` or `"radialGradient"`, the properties `posArray` and `colorArray` are also available.
		`var fill = fl.getDocumentDOM().getCustomFill();`
		`fill.style= "linearGradient";`
		`fill.colorArray = [ 0x00ff00, 0xff0000, 0x0000ff ];`
		`fill.posArray  = [0,  100, 200];`
		`fl.getDocumentDOM().setCustomFill( fill );`

flash

Description

The flash object represents the Flash application.

Methods

The following methods can be used with the flash object:

```
fl.browseForFileURL()
fl.closeAll()
fl.closeDocument()
fl.createDocument()
fl.enableImmediateUpdates()
fl.fileExists()
fl.findDocumentIndex()
fl.getDocumentDOM()
fl.mapPlayerURL()
fl.openDocument()
fl.openProject()
fl.openScript()
fl.quit()
fl.reloadEffects()
fl.reloadTools()
fl.revertDocument()
fl.runScript()
fl.saveAll()
fl.saveDocument()
fl.saveDocumentAs()
fl.setActiveWindow()
fl.trace()
```

fl.browseForFileURL()

Description

Opens a File Open or File Save system dialog box and lets the user specify a file to be opened or saved.

Usage

```
fl.browseForFileURL( browseType [, title [, previewArea ] ])
```

Arguments

The *browseType* argument is a String that specifies the type of file browse operation. Valid values are `"open"`, `"select"` or `"save"`. The values `"open"` and `"select"` both bring up the system File Open dialog box. Each value is provided for compatibility with Dreamweaver. The value `"save"` brings up a system File Save dialog box.

The optional *title* argument is a String that specifies the title for the File Open or File Save dialog box. If this argument is omitted, a default value is used.

The optional *previewArea* argument is ignored by Flash and Fireworks and is present only for compatibility with Dreamweaver.

Returns

A String containing the URL of the file.

Example

```
var fileURL = fl.browseForFileURL("open", "Select file");
var doc = fl.openDocument(fileURL);
```

fl.closeAll()

Description

Closes all open documents, displaying the Save As dialog box for any documents that were not previously saved. The method prompts the user, if necessary, but does not terminate the application.

Usage

```
fl.closeAll()
```

Arguments

None.

Returns

Nothing.

Example

```
fl.closeAll();
```

fl.closeDocument()

Description

Closes the specified document.

Usage

```
fl.closeDocument( documentObject [, bPromptToSaveChanges] )
```

Arguments

documentObject, [*bPromptToSaveChanges*]

The *documentObject* argument is a Document object. If *documentObject* refers to the active document, the Document window might not close until the script that calls this method finishes executing.

The optional *bPromptToSaveChanges* argument is a Boolean value. If it is `false`, the user is not prompted if the document contains unsaved changes. If the value is `true`, and if the document contains unsaved changes, the user is prompted with the standard yes-or-no dialog box. The default value is `true`.

Returns

A Boolean value: `true` if successful; `false` otherwise.

Example

```
//closes the specified document and prompts to save changes
fl.closeDocument(fl.documents[0]);
//closes the specified document without prompting to save changes
fl.closeDocument(fl.documents[0], false);
```

fl.createDocument()

Description

Opens a new document and selects it. Values for size, resolution, and color are the same as the current defaults.

Usage

```
fl.createDocument( [docType] )
```

Arguments

The optional *docType* argument is a String that specifies the type of document to create. Valid values are `"timeline"`, `"presentation"`, and `"application"`. The default value is `"timeline"`.

Returns

If successful, returns the Document object for the newly created document. If an error occurs, the value is undefined.

Example

```
//create a Timeline-based Flash Document
fl.createDocument();
fl.createDocument("timeline");
//create a Slide Presentation document
fl.createDocument("presentation");
//create a Form Application document
fl.createDocument("application");
```

fl.enableImmediateUpdates()

Description

This method lets the script developer enable immediate visual updates of the Timeline when executing effects. Immediate updates are normally suppressed so the user does not see intermediate steps that can be visually distracting and can make the effect appear to take longer than necessary. This method is purely for debugging purposes and should not be used in effects that are deployed in the field. After the effect completes, the internal state is reset to suppress immediate updates.

Usage

```
fl.enableImmediateUpdates(bEnableUpdates)
```

Arguments

The *bEnableUpdates* argument is a Boolean value that specifies whether to enable (`true`) or disable (`false`) immediate visual updates of the Timeline when executing effects.

Returns

Nothing.

Example

```
fl.enableImmediateUpdates(true) ;
fl.trace("Immediate updates are enabled");
```

fl.fileExists()

Description

Checks whether a file already exists on disk.

Usage

```
fl.fileExists( fileURI )
```

Arguments

The *fileURI* argument is a String that contains the path to the file.

Returns

A Boolean value: `true` if the file exists on disk; `false` otherwise.

Example

```
alert(fl.fileExists("file:///C|/example.fla"));
alert(fl.fileExists("file:///C|/example.jsfl"));
alert(fl.fileExists(""));
```

fl.findDocumentIndex()

Description

Finds the index of an open document with the specified name.

Usage

```
fl.findDocumentIndex( name )
```

Arguments

The *name* argument is the document name for which you want to find the index. The document must be open.

Returns

An integer that is the index of the document.

Example

```
var theIndex = fl.findDocumentIndex("test.fla");
```

fl.getDocumentDOM()

Description

Retrieves the DOM (Document object) of the currently active document.

Usage

```
fl.getDocumentDOM()
```

Arguments

None.

Returns

The DOM object of the current document.

Example

```
var myDom = fl.getDocumentDOM();
myDom.clipCopy();
```

fl.mapPlayerURL()

Description

Maps an escaped Unicode URL to a UTF-8 or MBCS URL. Use this method when the string will be used in ActionScript to access an external resource. You must use this method if you need to handle multibyte characters.

Usage

```
fl.mapPlayerURL( URI [, returnMBCS] )
```

Arguments

The *URI* argument is a String that contains the escaped Unicode URL to map.

The optional *returnMBCS* argument is a Boolean value that you must set to `true` if you want an escaped MBCS path returned. Otherwise, the method returns UTF-8. The default value is `false`.

Returns

A String that is the converted URL.

Example

```
// Convert the URL to UTF-8 so the player can load it.
var url = MMExecute( "fl.mapPlayerURL(" + myURL + ", false);" );
mc.loadMovie( url);
```

fl.openDocument()

Description

Opens a Flash (FLA) document for editing in a new Flash Document window and gives it the focus. For a user, the effect is the same as selecting File > Open and then selecting a file. If the specified file is already open, the window that contains the document comes to the front. The window that contains the specified file becomes the currently selected document.

Usage

```
fl.openDocument( fileURI )
```

Arguments

The *fileURI* argument is a String that specifies the name of the file to be opened, expressed as a URI (file:///URI).

Returns

If successful, returns the Document object for the newly opened document. If the file is not found, or is not a valid FLA file, an error is reported and the script is cancelled.

Example

```
doc = fl.openDocument("file:///c|/Document.fla");
```

fl.openProject()

Description

Opens a Flash Project (FLP) file in the authoring tool for editing.

Usage

```
fl.openProject( fileURI )
```

Arguments

The *fileURI* argument is a String that specifies the path of the Flash project file to open, expressed as a URI (file:///URI).

Returns

Nothing.

Example

```
fl.openProject("file:///c|/myProjectFile.flp");
```

fl.openScript()

Description

Opens a script (JSFL, AS, ASC) or other file (XML, TXT) into the Flash text editor.

Usage

```
fl.openScript( fileURI )
```

Arguments

The *fileURI* argument is a String that specifies the path of the JSFL, AS, ASC, XML, TXT or other file that should be loaded into the Flash text editor, expressed as a URI (file:///URI).

Returns

Nothing.

Example

```
fl.openScript("file:///c|/temp/my_test.jsfl");
```

fl.quit()

Description

Quits Flash, prompting the user to save any changed documents.

Usage

```
fl.quit( [bPromptIfNeeded] )
```

Arguments

The optional *bPromptIfNeeded* argument is a Boolean value that is `true` (default) if you want the user to be prompted to save any modified documents. Set this argument to `false` if you do not want the user to be prompted to save modified documents. In the latter case, any modifications in open documents will be discarded and the application will exit immediately. Although it is useful for batch processing, use this method with caution.

Returns

Nothing.

Example

```
//quit with prompt to save any modified documents
fl.quit();
//quit without saving any files
fl.quit(false);
```

fl.reloadEffects()

Description

Reloads all effects descriptors defined in the user's Configuration Effects folder. This permits you to rapidly change the scripts during development, and it provides a mechanism to improve the effects without relaunching the application. Best if used in a command placed in the Commands folder.

Usage

```
fl.reloadEffects()
```

Arguments

None.

Returns

Nothing.

Example

The following example is a one-line script that you can place in the Commands folder. When you need to reload effects, go to the Commands menu and execute the script.

```
fl.reloadEffects();
```

fl.reloadTools()

Description

Rebuilds the toolbar from the toolconfig.xml file. Used only when creating extensible tools.

Usage

```
fl.reloadTools()
```

Arguments

None.

Returns

Nothing.

Example

```
fl.reloadTools();
```

fl.revertDocument()

Description

Reverts the specified document to its last saved version. Unlike the File > Revert menu option, this method does not display a warning window that asks the user to confirm the operation.

Usage

```
fl.revertDocument( documentObject )
```

Arguments

The *documentObject* argument is a Document object. If *documentObject* refers to the active document, the Document window might not revert until the script that calls this method finishes executing.

Returns

A Boolean value: returns `true` if the Revert operation completes successfully; `false` otherwise.

Example

```
fl.revertDocument(fl.getDocumentDOM());
```

fl.runScript()

Description

Executes a JavaScript file. If a function is specified as one of the arguments, it will run the function and also any code in the script that is not within the function. The rest of the code in the script runs before the function is run.

Usage

```
fl.runScript( fileURI [, funcName [, arg1, arg2, ...] ])
```

Arguments

The *fileURI* argument is a String that specifies the name of the script file to execute, expressed as a URI (file:///URI).

The optional *funcName* argument is a String that identifies a function to execute in the JSFL file that is specified in *fileURI*.

The optional *arg* argument specifies one or more arguments to be passed to *funcname*.

Returns

Nothing.

Example

Suppose there is a script file named testScript.jsfl in drive C: and its contents are as follows:

```
function testFunct(num, minNum) {
  fl.trace("in testFunct: 1st arg: " + num + " 2nd arg: " + minNum);
}
for (i=0; i<2; i++) {
  fl.trace("in for loop i=" + i);
}
fl.trace("end of for loop");
//end of testScript.jsfl
```

If you issue the following command:

```
fl.runScript("file:///C|/testScript.jsfl", "testFunct", 10, 1);
```

The following information appears in the Output panel:

```
in for loop i=0
in for loop i=1
end of for loop
in testFunct: 1st arg: 10 2nd arg: 1
```

You can also just call testScript.jsfl without executing a function:

```
fl.runScript("file:///C|/testScript.jsfl");
```

which produces the following in the Output panel:

```
in for loop i=0
in for loop i=1
end of for loop
```

fl.saveAll()

Description

Saves all open documents, displaying the Save As dialog box for any documents that were not previously saved.

Usage

```
fl.saveAll()
```

Arguments

None.

Returns

Nothing.

Example

```
fl.saveAll();
```

fl.saveDocument()

Description

Saves the specified document as a FLA document.

Usage

```
fl.saveDocument( document [, fileURI] )
```

Arguments

The *document* argument is a Document object that specifies the document to be saved. If *document* is `null`, the active document is saved.

The optional *fileURI* argument is a String that specifies the name of the saved document, expressed as a file:///URI. If the *fileURI* argument is `null` or omitted, the document is saved with its current name. If the document is not yet saved, Flash displays the Save As dialog box.

Returns

A Boolean value: `true` if the save operation completes successfully; `false` otherwise.

Example

```
//save the current document
alert(fl.saveDocument(fl.getDocumentDOM()));
//save document 0
alert(fl.saveDocument(fl.documents[0], "file:///C|/example1.fla"));
//save the specified document
alert(fl.saveDocument(fl.documents[1],"file:///C|/example2.fla"));
```

fl.saveDocumentAs()

Description

Displays the Save As dialog box for the specified document.

Usage

```
fl.saveDocumentAs( document )
```

Arguments

The *document* argument is a Document object that specifies the document to save. If *document* is `null`, the active document is saved.

Returns

A Boolean value: `true` if the Save As operation completes successfully; `false` otherwise.

Example

```
alert(fl.saveDocumentAs(fl.documents[0]));
```

fl.setActiveWindow()

Description

Sets the active window to be the specified document. This method is also supported by Dreamweaver and Fireworks. If the document has multiple views (created by Edit In New Window), the first view is selected.

Usage

```
fl.setActiveWindow( document [, bActivateFrame] )
```

Arguments

The *document* argument is a Document object that specifies the document to select as the active window.

The optional *bActivateFrame* argument is present for consistency with the Dreamweaver API. As in Fireworks, it is optional and it is ignored.

Returns

Nothing.

Example

```
fl.setActiveWindow(fl.documents[0]);
```

or

```
var theIndex = fl.findDocumentIndex("myFile.fla");
fl.setActiveWindow(fl.documents[theIndex]);
```

fl.trace()

Description

Sends a text string to the Output panel. Works the same as the trace statement in ActionScript.

Usage

```
fl.trace( message )
```

Arguments

The *message* argument is a String that appears in the Output panel.

Returns

Nothing.

Example

```
fl.trace("hello World!!!");
var myPet = "cat";
fl.trace("I have a " + myPet);
```

Properties

Properties	Data type	Value
activeEffect **read only**	object	The Effect object for the current effect being applied. `var ef = fl.activeEffect;`
componentsPanel **read only**	object	A componentsPanel object, which represents the Components panel. `var comPanel = fl.componentsPanel;`
configDirectory **read only**	string	Returns the full path for the local user's Configuration folder as a platform-specific path. `fl.trace( "My local configuration directory is " + fl.configDirectory );`
configURI **read only**	string	Gives a URI to the user's Configuration folder. `fl.runScript( fl.configURI + "folder/File.jsfl" );`
createNewDocList **read only**	array	An array of strings that represent the various types of documents that can be created. `fl.trace("Number of choices " + fl.createNewDocList.length); for (i = 0; i < fl.createNewDocList.length; i++) fl.trace("choice: " + fl.createNewDocList[i]);`

Properties	Data type	Value
createNewDocListType **read only**	array	An array of strings that represent the various types of documents that can be created. The entries in the array correspond directly (by index) to the entries in the fl.createNewDocList array. `fl.trace("Number of types " + fl.createNewDocListType.length); for (i = 0; i < fl.createNewDocListType.length; i++) fl.trace("type: " + fl.createNewDocListType[i]);`
createNewTemplateList **read only**	array	An array of strings that represent the various types of templates that can be created. `fl.trace("Number of template types: " + fl.createNewTemplateList.length); for (i = 0; i < fl.createNewTemplateList.length; i++) fl.trace("type: " + fl.createNewTemplateList[i]);`
documents **read only**	array	An array of Document objects that represent the documents (FLA files) that are currently open for editing. `//docs is an array of open FLA documents var docs = fl.documents;`
effects **read only**	array	An array of Effect objects, based on XML parameter file. These are not effects, but a description of effects. The array length corresponds to the number of effects (based on the XML parameter definition files, not the number of JSFL implementation files) registered when the program opens. To return the first registered effect, use: `ef = fl.effects[0]`
Math **read only**	see description	The Math object provides methods for matrix and point operations. `//Select an element on the stage, then run this script var mat =fl.getDocumentDOM().selection[0].matrix; var invMat = fl.Math.invertMatrix( mat ); for(var prop in mat){ fl.trace("mat."+prop+" = " + mat[prop]); }`
mruRecentFileList **read only**	array	An array of the complete filenames in the Most Recently Used (MRU) list that the authoring tool manages. `fl.trace("Number of recently opened files: " + fl.mruRecentFileList.length); for (i = 0; i < fl.mruRecentFileList.length; i++) fl.trace("file: " + fl.mruRecentFileList[i]);`
mruRecentFileListType **read only**	array	An array of the file types in the MRU list that the authoring tool manages. This array corresponds to the array in the fl.mruRecentFileList property. `fl.trace("Number of recently opened files: " + fl.mruRecentFileListType.length); for (i = 0; i < fl.mruRecentFileListType.length; i++) fl.trace("type: " + fl.mruRecentFileListType[i]);`
outputPanel **read only**	object	Reference to outputPanel object `fl.outputPanel.trace("Hello!");`
tools **read only**	array	An array of Tool objects. Used only when creating extensible tools. See "Tools" on page 223.

Properties	Data type	Value
`version` **read only**	string	Returns the long string version of the Flash authoring tool, including platform. `alert( fl.version ); // WIN 7,0,0,380`
`xmlui` **read only**	object	XMLUI object. Lets you get and set XMLUI properties in a XMLUI dialog box and lets you accept or cancel the dialog box programmatically. `fl.xmlui.cancel();`

folderItem

Description

The folderItem object is a subclass of the Item object. It does not have any unique methods or properties.

Methods

This object has no unique methods. See "Item" on page 121.

Properties

This object has no unique properties. See "Item" on page 121.

fontItem

Description

The fontItem object is a subclass of the Item Object.

Methods

This object has no unique methods. See “Item” on page 121.

Properties

This object has no unique properties. See “Item” on page 121.

Frame

Description

The Frame object represents frames in the layer.

Methods

This object has no unique methods.

Properties

Property	Data type	Value

`actionScript`	string	String representing ActionScript code.
		`//assign stop() to first frame top layer action`
		`fl.getDocumentDOM().getTimeline().layers[0].frames[0].actionScript = 'stop();';`

`duration`
read only
	int	Number of frames in a frame sequence .
		`// frameSpan is the number of frames in a frame`
		`// sequence that starts at first frame`
		`// in the top layer`
		`var frameSpan =`
		`fl.getDocumentDOM().getTimeline().layers[0].frames[0].duration;`

`elements`
read only
	array	An array of Element objects. The order of elements is the order in which they are stored in the FLA file. Note that if there are multiple shapes on the stage, and each is ungrouped, then Flash treats them as one element. If each shape is grouped, so there are multiple groups on the stage, Flash sees them as separate elements. In other words, Flash treats raw, ungrouped shapes as a single element regardless of how many separate shapes there are on stage. If a frame contains three raw, ungrouped shapes, for example, then `elements.length` in that frame returns a value of 1. Select each shape individually and group it to work around this issue.
		`// myElements is an array of current elements on`
		`// the top layer, first frame:`
		`var myElements =`
		`fl.getDocumentDOM().getTimeline().layers[0].frames[0].elements;`

`labelType`	string	Specifies the type of Frame name. Valid values are `"none"`, `"name"`, `"comment"`, and `"anchor"`. Setting a label to "`none`" clears the `frame.name` property.
		`// set first frame in top layer name to 'First Frame'`
		`// then set its label to comment`
		`fl.getDocumentDOM().getTimeline().layers[0].frames[0].name = 'First Frame';`
		`fl.getDocumentDOM().getTimeline().layers[0].frames[0].labelType = 'comment';`

`motionTweenOrientToPath`	Boolean	If set to `true`, the object will remain oriented to path, if set to `false`, it will not. `fl.getDocumentDOM().getTimeline().layers[0].frames[0].motionTweenOrientToPath = true;`

Property	Data type	Value
motionTweenRotate	string	Specifies how the object will rotate during the tween. Acceptable values are: "none", "auto", "clockwise", and "counter-clockwise". `fl.getDocumentDOM().getTimeline().layers[0].frames[0].motionTweenRotate = 'clockwise';`
motionTweenRotateTimes	int	Number of times to rotate the element in the starting keyframe by the time it gets to the next keyframe. `fl.getDocumentDOM().getTimeline().layers[0].frames[0].motionTweenRotateTimes = 3;`
motionTweenScale	boolean	If set to true, the tween should scale. `fl.getDocumentDOM().getTimeline().layers[0].frames[0].motionTweenScale = true;`
motionTweenSnap	boolean	If set to true, object will automatically snap to motion guide. `fl.getDocumentDOM().getTimeline().layers[0].frames[0].motionTweenSnap = true;`
motionTweenSync	boolean	If set to true, synchronizes symbols. `fl.getDocumentDOM().getTimeline().layers[0].frames[0].motionTweenSync = true;`

name	string	The name of the frame.
		`//set the name of the first frame, top layer`
		`// to 'First Frame':`
		`fl.getDocumentDOM().getTimeline().layers[0].frames[0].name = 'First Frame';`
		`var frameLabel = fl.getDocumentDOM().getTimeline().layers[0].frames[0].name;`

shapeTweenBlend	string	Specifies the blend for a shape tween. Valid values are "distributive" and "angular". `fl.getDocumentDOM().getTimeline().layers[0].frames[0].shapeTweenBlend = 'distributive';`
soundEffect	string	Specifies effects on a sound. Acceptable values are "none", "left channel", "right channel", "fade left to right", "fade right to left", "fade in", "fade out", and "custom". `fl.getDocumentDOM().getTimeline().layers[0].frames[0].soundEffect = 'fade in';`

soundLibraryItem	object	Library item used to create a sound.
		`// assign the first item in the library`
		`// (must be a sound object)`
		`// to soundLibraryItem of the first frame:`
		`fl.getDocumentDOM().getTimeline().layers[0].frames[0].soundLibraryItem =fl.getDocumentDOM().library.items[0];`

soundLoop	int	An integer value that specifies the number of times the sound should loop or repeat. `fl.getDocumentDOM().getTimeline().layers[0].frames[0].soundLoop = 2;`

Property	Data type	Value
soundLoopMode	string	Specifies whether sound should repeat a specific number of times, or loop indefinitely. Valid values are: "repeat" and "loop". `fl.getDocumentDOM().getTimeline().layers[0].frames[0].soundLoopMode = 'repeat';`

soundName	string	Name of sound.
		`// change the soundName in first frame to 'song1.mp3'`
		`// (song1.mp3 must exist in the Library):`
		`fl.getDocumentDOM().getTimeline().layers[0].frames[0].soundName = 'song1.mp3';`

soundSync	string	Specifies the sync behavior of the sound. Acceptable values are: "event", "stop", "start", and "stream". `fl.getDocumentDOM().getTimeline().layers[0].frames[0].soundSync = 'stream';`

startFrame
read only
	int	The frame number of the first frame in a sequence.
		`// stFrame is the number of the first frame`
		`// in the frame sequence. At frame #9,`
		`// if the frame sequence is expanding from`
		`// frame #5 to frame #10, then stFrame = 5`
		`var stFrame =`
		`fl.getDocumentDOM().getTimeline().layers[0].frames[8].startFrame`

tweenEasing	int	The amount of easing. Valid values are -100 to 100 `fl.getDocumentDOM().getTimeline().layers[0].frames[0].tweenEasing = 50;`
tweenType	string	Type of tween; valid values are "motion", "shape", or "none". The value "none" removes the motion tween. Use the timeline.createMotionTween() method to create a tween. `fl.getDocumentDOM().getTimeline().layers[0].frames[0].tweenType = 'motion';`

HalfEdge

Description

Directed side of an edge. An edge has two half edges. You can transverse the contours by walking around these half edges. They are ordered. One half edge represents one side of the edge, the other half edge represents the other side.

Methods

The following methods are available for the halfEdge object:

```
halfEdge.getEdge()
halfEdge.getNext()
halfEdge.getOppositeHalfEdge()
halfEdge.getPrev()
halfEdge.getVertex()
```

halfEdge.getEdge()

Description

Gets the Edge object for the halfEdge object.

Arguments

None.

Returns

An Edge object.

Example

```
var shape = fl.getDocumentDOM().selection[0];
var halfEdge = shape.edges[0].getHalfEdge(0);
var edge = halfEdge.getEdge();
```

halfEdge.getNext()

Description

Gets the next halfEdge on the current contour.

Note: Although halfEdges have a direction and a sequence order, edges do not.

Arguments

None.

Returns

A halfEdge object.

Example

```
var shape = fl.getDocumentDOM().selection[0];
var hEdge = shape.edges[0].getHalfEdge( 0 );
var nextHalfEdge = hEdge.getNext();
```

halfEdge.getOppositeHalfEdge()

Description

Gets the halfEdge object on the other side of the edge.

Arguments

None.

Returns

A halfEdge object.

Example

```
var shape = fl.getDocumentDOM().selection[0];
var hEdge = shape.edges[0].getHalfEdge(0);
var otherHalfEdge = hEdge.getOppositeHalfEdge();
```

halfEdge.getPrev()

Description

Gets the preceding halfEdge object on the current contour.

Note: Although halfEdges have a direction and a sequence order, edges do not.

Arguments

None.

Returns

A halfEdge object.

Example

```
var shape = fl.getDocumentDOM().selection[0];
var hEdge = shape.edges[0].getHalfEdge( 0 );
var prevHalfEdge = hEdge.getPrev();
```

halfEdge.getVertex()

Description

Gets the Vertex at the head of the halfEdge object.

Arguments

None.

Returns

A Vertex object.

Example

```
var shape = fl.getDocumentDOM().selection[0];
var edge = shape.edges[0];
var hEdge = edge.getHalfEdge(0);
var vertex = hEdge.getVertex();
```

Properties

Property	Data type	Value

`id`
read only
	int	A unique identifier for the halfEdge object.
		`var shape = fl.getDocumentDOM().selection[0];`
		`alert(shape.contours[0].getHalfEdge(0).id);`

Instance

Description

Instance is a subclass of the Element object.

Methods

There are no unique methods for this object; see the Element object.

Properties

In addition to all of the Element object properties, Instance has the following properties:

Property	Data type	Value

`instanceType`
read only
	string	Type of Instance. Valid values are `"symbol"`, `"bitmap"`, `"embedded video"`, `"linked video"`, and `"compiled clip"`.
		`//select a movie clip`
		`var type = fl.getDocumentDOM().selection[0].instanceType;`
		`//type = 'symbol'`

`libraryItem`	object	Library item used to instantiate this instance. You can change this property only to another `libraryItem` of the same type (that is, you cannot set a `symbol` instance to refer to a bitmap).
		`// select a symbol then change it to refer to the first item`
		`// in the Library.`
		`fl.getDocumentDOM().selection[0].libraryItem =`
		`fl.getDocumentDOM().library.items[0];`

Item

Description

The Item object is an abstract base class. Anything in the Library derives from Item.

Methods

The `Item` object uses the following methods:

```
item.addData()
item.getData()
item.hasData()
item.removeData()
```

item.addData()

Description

Adds specified data to Library item.

Usage

```
item.addData( name, type, data )
```

Arguments

name, *type*, *data*

The *name* argument is a String that specifies the name of the data.

The *type* argument is a String that specifies the type of data. Valid types are `"integer"`, `"integerArray"`, `"double"`, `"doubleArray"`, `"string"`, and `"byteArray"`.

The *data* argument is the data to add to the specified Library item. The type of data depends on the value of the *type* argument. The logical rules apply, meaning that if *type* is `"integer"`, the value of *data* must be an integer, and so on.

Returns

Nothing.

Example

```
fl.getDocumentDOM().library.items[0].addData("myData", "integer", 12);
```

item.getData()

Description

Retrieves the value of the specified data.

Usage

```
item.getData( name )
```

Arguments

name

The *name* argument is a String that specifies the name of the data to retrieve.

Returns

The data specified by the *name* argument. The type of data returned depends on the type of stored data.

Example

```
var data = fl.getDocumentDOM().library.items[0].getData( "myData" );
```

item.hasData()

Description

Determines whether the Library item has the named data.

Usage

```
item.hasData( name )
```

Arguments

name

The *name* argument is a String that specifies the name of the data to check for in the Library item.

Returns

A Boolean value: `true` if the specified data exists; `false` otherwise.

Example

```
if ( fl.getDocumentDOM().library.items[0].hasData( "myData" ) ){
  fl.trace("Yep, it's there!");
}
```

item.removeData()

Description

Removes persistent data from the Library item.

Usage

```
item.removeData( name )
```

Arguments

name

The *name* argument specifies the name of the data to remove from the Library item.

Returns

Nothing.

Example

```
fl.getDocumentDOM().library.items[0].removeData( "myData" );
```

Properties

Property	Data type	Value
`itemType` **read only**	string	Type of element. Potential values are `"undefined"`, `"component"`, `"movie clip"`, `"graphic"`, `"button"`, `"video"`, `"folder"`, `"font"`, `"sound"`, `"bitmap"`, `"compiled clip"`, and `"video"`. `fl.trace(fl.getDocumentDOM().library.items[0].itemType);`
`linkageClassName`	string	Allows the user to specify an ActionScript 2.0 class that will be associated with the symbol.The `linkageExportForRS` and/or `linkageExportForAS` properties must be set to `true`, The `linkageImportForRS` property must be set to `false`. `fl.getDocumentDOM().library.items[0].linkageClassName = "myClass";`
`linkageExportForAS`	Boolean	If `true`, the item will be exported for ActionScript. User can also set the `linkageExportForRS` and `linkageExportInFirstFrame` properties to `true`. The `linkageImportForRS` property must be set to `false` if this is set to `true`. `fl.getDocumentDOM().library.items[0].linkageExportForAS = true;`
`linkageExportForRS`	Boolean	If `true`, the item will be exported for runtime sharing. Can be set to `true` only if `linkageImportForRS` is set to `false`. The properties `linkageIdentifier` and `linkageURL` must be defined. `fl.getDocumentDOM().library.items[0].linkageExportForRS = true;`
`linkageExportInFirstFrame`	Boolean	If `true`, the item will be exported in the first frame; `false` otherwise. Can be set to `true` only when either `linkageExportForRS` or `linkageExportForAS` are set to `true`. `fl.getDocumentDOM().library.items[0].linkageExportInFirstFrame = true;`
`linkageIdentifier`	string	This is the name for the symbol when referencing it using ActionScript or for runtime sharing. This is the name Flash will use to identify the asset when linking to the destination SWF file. It must be specified if `linkageExportForAS` or `linkageExportForRS` are set to `true`. `fl.getDocumentDOM().library.items[0].linkageIdentifier = "my_mc";`
`linkageImportForRS`	Boolean	If `true`, the item will be imported for runtime sharing; `false` otherwise. `linkageExportForRS` and `linkageExportForAS` must be set to `false`. User must specify an identifier and a URL. `fl.getDocumentDOM().library.items[0].linkageImportForRS = true;`

Property	Data type	Value
`linkageURL`	string	URL where the SWF file containing the shared asset is located. Must be set when `linkageExportForRS` or `linkageImportForRS` are set to `true`. `fl.getDocumentDOM().library.items[0].linkageURL = "theShareSWF.swf";`
`name`	string	Name of the Library item, which includes the folder structure. For example, if Symbol1 is inside a folder called Folder1, the name property would be "Folder1/Symbol1". `fl.trace(fl.getDocumentDOM().library.items[0].name);`

Layer

Description

The Layer object represents a layer in the Timeline. The `layers` property of the Timeline object contains an array of Layer objects, which can be accessed by `fl.getDocumentDOM().getTimeline().layers`

Methods

This object has no unique methods.

Properties

Property	Data type	Value

`color`	string	The color assigned to outline the layer (the equivalent of the Outline color setting in the Layer Properties dialog box). Specified in hexadecimal #rrggbb format (where r is red, g is green, and b is blue), a hexidecimal color value (such as, 0xff0000), or an integer color value.
		The following example stores the value of the first layer in the *`colorValue`* variable:
		`var colorValue =`
		`  fl.getDocumentDOM().getTimeline().layers[0].color;`
		The following example sets the color of the first layer to red:
		`fl.getDocumentDOM().getTimeline().layers[0].color = "#FF0000"`

`frameCount`
read only
	int	Total number of frames in the layer.
		The following example sets the value of variable *`fcNum`* to equal the number of frames in the first layer:
		`var fcNum =`
		`  fl.getDocumentDOM().getTimeline().layers[0].frameCount;`

`frames`
read only
	array	An array of frame objects. The following example sets the variable `frameArray` to the array of frame objects for the frames in the current document:
		`var frameArray =`
		`  fl.getDocumentDOM().getTimeline().layers[0].frames;`
		`To determine if a frame is a keyframe, check whether the`
		`  frame.startFrame property matches the array index.`
		`For example:`
		`var frameArray =`
		`  fl.getDocumentDOM().getTimeline().layers[0].frames;`
		`var n = frameArray.length;`
		`for (i=0; i<n; i++)`
		`{`
		`  if (i==frameArray[i].startFrame)`
		`  {`
		`  alert("Keyframe at: " + i);`
		`  }`
		`}`

Property	Data type	Value

`height`	int	Percentage layer height (the equivalent of the Layer height value in the Layer Properties dialog box). Acceptable values represent percentages of the default height: 100, 200, or 300.
		The following example stores the percentage value of the first layer's height setting:
		`var layerHeight = fl.getDocumentDOM().getTimeline().layers[0].height;`
		The following example sets the height of the first layer to 300 percent:
		`fl.getDocumentDOM().getTimeline().layers[0].height = 300;`

`layerType`	string	The current use of the layer (the equivalent of the Type setting in the Layer Properties dialog box). Acceptable values are `"normal"`, `"guide"`, `"guided"`, `"mask"`, `"masked"`, `"folder"`. The following example makes the first layer in the Timeline a folder: `fl.getDocumentDOM().getTimeline().layers[0].layerType = "folder";`

`locked`	Boolean	Locked status of the layer. If set to `true`, the layer is locked. The default value is `false`.
		The following example stores the Boolean value for the status of the first layer in the *`lockStatus`* variable:
		`var lockStatus = fl.getDocumentDOM().getTimeline().layers[0].locked;`
		The following example sets the status of the first layer to unlocked:
		`fl.getDocumentDOM().getTimeline().layers[0].locked = false;`

`name`	string	Name of the layer. The following example sets the name of the first layer (as it appears in the Timeline, regardless of the Load order) in the current document to "foreground": `fl.getDocumentDOM().getTimeline().layers[0].name = "foreground";`
`outline`	Boolean	Status of outlines for all objects on the layer. If set to `true`, all objects on the layer appear only with outlines. If `false`, objects appear as they were created. The following example makes all objects on the first layer appear only with outlines: `fl.getDocumentDOM().getTimeline().layers[0].outline = true;`

Property	Data type	Value

parentLayer	object	A layer's containing folder, guiding, or masking layer. Acceptable values for the parent layer are a folder, guide, or mask layer that precedes the layer, or the parentLayer of the preceding or following layer. Setting the layer's parentLayer will not move the layer's position in the list; trying to set a layer's parentLayer to a layer that would require moving it has no effect. Uses null for a top level layer.
		The following example uses two layers at the same level on the same Timeline. The first layer (layers[0]) is converted into a folder, and then set as the parent folder of the second layer (layers[1]), accomplishing the same task as moving the second layer inside the first layer:
		`var parLayer = fl.getDocumentDOM().getTimeline().layers[0];`
		`parLayer.layerType = "folder";`
		`fl.getDocumentDOM().getTimeline().layers[1].parentLayer = parLayer;`

visible	Boolean	Show/hide property of the layer's objects on the Stage. If set to true, all objects in the layer are visible; false otherwise. The default value is true. The following example makes all objects in the first layer invisible: `fl.getDocumentDOM().getTimeline().layers[0].visible = false;`

library

Description

The library object represents the Library panel. It is a property of the Document object and can be accessed by `fl.getDocumentDOM().library`.

The library object contains an array of items of different types, including symbols, bitmaps, sounds, and video.

Methods

The following methods are used with the library object:

```
library.addItemToDocument()
library.addNewItem()
library.deleteItem()
library.duplicateItem()
library.editItem()
library.expandFolder()
library.findItemIndex()
library.getItemProperty()
library.getItemType()
library.getSelectedItems()
library.importEmbeddedSWF()
library.itemExists()
library.moveToFolder()
library.newFolder()
library.renameItem()
library.selectAll()
library.selectItem()
library.selectNone()
library.setItemProperty()
library.updateItem()
```

library.addItemToDocument()

Description

Adds the current or specified item to the document at the specified position.

Usage

```
library.addItemToDocument( position [, namePath] )
```

Arguments

The *position* parameter is a point that specifies the *x,y* position of the center of the item on the Stage.

The optional *namePath* argument is a String that specifies the name of the item. If the item is in a folder, specify its name and path using slash notation. If *namePath* is not specified, the current Library selection is used.

Returns

A Boolean value: `true` if the item was successfully added to the document; `false` otherwise.

Example

The following example adds the currently selected item to the current document at the (3, 60) position:

```
fl.getDocumentDOM.library.addItemToDocument({x:3, y:60});
```

The following example adds the item `Symbol1` located in `folder1` of the Library to the current document at the (550, 485) position:

```
fl.getDocumentDOM().library.addItemToDocument({x:550.0, y:485.0}, "folder1/
  Symbol1");
```

library.addNewItem()

Description

Creates a new item of the specified type in the Library panel and sets the new item to the currently selected item.

Usage

```
library.addNewItem( type [, namePath] )
```

Arguments

The *type* parameter is a String that specifies the type of item to create. The only acceptable values for *type* are `"video"`, `"movie clip"`, `"button"`, `"graphic"`, `"bitmap"`, and `"folder"` (so, for example, you cannot add a sound to the Library with this method). Specifying a folder path is the same as using `library.newFolder`.

The optional *namePath* argument is a String that specifies the name of the item to be added. If the item is in a folder, specify its name and path using slash notation.

Returns

A Boolean value: `true` if the item was successfully created; `false` otherwise.

Example

The following example creates a new button item named `start` in a new folder named `folderTwo`:

```
fl.getDocumentDOM().library.addNewItem("button" , "folderTwo/start");
```

library.deleteItem()

Description

Deletes the current or specified item from the Library panel. This method can affect multiple items if several are selected.

Usage

```
library.deleteItem( [namePath] )
```

Arguments

The optional *namePath* argument is a String that specifies the name of the item to be deleted. If the item is in a folder, specify its name and path using slash notation. If no name is specified, Flash deletes the currently selected item or items. To delete all the items in the Library panel, select all items before using this method.

Returns

A Boolean value: `true` if the item(s) is successfully deleted; `false` otherwise.

Example

The following example deletes the currently selected item:

```
fl.getDocumentDOM().library.deleteItem();
```

The following example deletes the item `Symbol1` from the Library folder `Folder1`:

```
fl.getDocumentDOM().library.deleteItem("Folder1/Symbol1"));
```

library.duplicateItem()

Description

Makes a copy of the currently selected item or specified item. The new item has a default name (such as `item copy`) and is set as the currently selected item. If more than one item is selected, the command fails.

Usage

```
library.duplicateItem( [namePath] )
```

Arguments

The optional *namePath* argument is a String that specifies the name of the item to duplicate. If the item is in a folder, specify its name and path using slash notation.

Returns

A Boolean value: `true` if the item is duplicated successfully; `false` otherwise. If more than one item is selected, Flash returns `false`.

Example

The following example creates a copy of the item `square` located in the Library folder `test`:

```
fl.getDocumentDOM().library.duplicateItem("test/square");
```

library.editItem()

Description

Opens the currently selected or specified item in Edit mode.

Usage

```
library.editItem( [namePath] )
```

Arguments

The optional *namePath* argument is a String that specifies the name of the item. If the item is in a folder, specify its name and path using slash notation. If *namePath* is not specified, the single selected Library item is opened in Edit mode. If none or more than one item in the Library is currently selected, the first scene in the file appears for editing.

Returns

A Boolean value: `true` if the item exists and can be edited; `false` if the specified item doesn't exist or cannot be edited.

Example

The following example opens the item `circle` in the `test` folder of the Library for editing:

```
fl.getDocumentDOM().library.editItem("test/circle");
```

library.expandFolder()

Description

Expands or collapses the currently selected or specified folder in the Library.

Usage

```
library.expandFolder( bExpand [, bRecurseNestedParents [, namePath]] )
```

Arguments

The *bExpand* argument is a Boolean value. If it is `true`, the folder is expanded. If `false`, the folder is collapsed.

The optional *bRecurseNestedParents* argument is a Boolean value. If it is `true`, all the folders within the specified folder are expanded or collapsed. The default value is `false`.

The optional *namePath* argument is a String that specifies the name and, optionally, the path of the folder to expand or collapse. If this argument is not specified, the method applies to the currently selected folder.

Returns

A Boolean value: `true` if the item is successfully expanded or collapsed; `false` if unsuccessful or the specified item is not a folder.

Example

The following example closes the `test` folder of the Library:

```
fl.getDocumentDOM().library.expandFolder(false, true, "test");
```

library.findItemIndex()

Description

Returns the Library item's index value (zero-based). The Library index is flat, so folders are considered part of the main index. Folder paths can be used to specify a nested item.

Usage

```
library.findItemIndex( namePath )
```

Arguments

The *namePath* argument is a String that specifies the name of the item. If the item is in a folder, specify its name and path using slash notation.

Returns

An integer value representing the item's zero-based index value.

Example

The following example stores the zero-based index value of the Library item `"square"`, which is in the `"test"` folder, in the variable `sqIndex`, and then displays the index value in a dialog box:

```
var sqIndex = fl.getDocumentDOM().library.findItemIndex("test/square");
alert(sqIndex);
```

library.getItemProperty()

Description

Gets the property for the selected item.

Usage

```
library.getItemProperty( property )
```

Arguments

The *property* argument is a String. For a list of properties, see the Item object and its subclasses.

Returns

A string value for the property.

Example

The following example shows a dialog box that contains the Linkage Identifier value for the symbol when referencing it using ActionScript, or for runtime sharing:

```
alert(fl.getDocumentDOM().library.getItemProperty("linkageIdentifier"));
```

library.getItemType()

Description

Gets the type of object currently selected or specified by a Library path.

Usage

```
library.getItemType( [namePath] )
```

Arguments

The optional *namePath* argument is a String that specifies the name of the item. If the item is in a folder, specify its name and path using slash notation. If *namePath* is not specified, Flash provides the type of the current selection. If more than one item is currently selected and no *namePath* is provided, Flash ignores the command.

Returns

A string value specifying the type of object. Possible values include: `"undefined"`, `"component"`, `"movie clip"`, `"graphic"`, `"button"`, `"video"`, `"folder"`, `"font"`, `"sound"`, `"bitmap"`, and `"compiled clip"`.

Example

The following example shows a dialog box that contains the item type of `symbol 1` located in the subfolder `folder 2`:

```
alert( fl.getDocumentDOM.library.getItemType("folder 1/folder 2/symbol 1");
```

library.getSelectedItems()

Description

Gets the array of all currently selected items in the Library.

Arguments

None.

Returns

An array of values for all items currently selected in the Library.

Example

The following example stores the array of currently selected Library items (in this case, several audio files) in the *`selItems`* variable and then changes the `sampleRate` property of the first audio file in the array to `"11 kHz"`:

```
var selItems = fl.getDocumentDOM().library.getSelectedItems();
selItems[0].sampleRate = "11 kHz";
```

library.importEmbeddedSWF()

Description

This method imports a Shockwave file (SWF) into the Library as a compiled clip. This method is different than using File > Import > SWF. This method allows you to embed a compiled SWF inside the Library. There is no corresponding user interface functionality, and it must be used with an external library or DLL.

Usage

```
library.importEmbeddedSWF( linkageName, swfData [, libName] )
```

Arguments

The *`linkageName`* argument is a String that provides the name of the SWF linkage of the root movie clip.

The *`swfData`* argument is an array of binary SWF data.

The optional *`libName`* argument is a String that specifies the Library name for the created item. If the name is already used, the method creates an alternate name.

Returns

Nothing.

Example

The following example adds the SWF with the *linkageName* value of MyMovie to the Library as a compiled clip named Intro:

```
fl.getDocumentDOM().library.importEmbeddedSWF("MyMovie", swfData, "Intro");
```

library.itemExists()

Description

Checks to see if a specified item exists in the Library.

Usage

```
library.itemExists( namePath )
```

Arguments

The *namePath* argument is a String that specifies the name of the item. If the item is in a folder, specify its name and path using slash notation.

Returns

A Boolean value: true if the specified item exists in the Library; false otherwise.

Example

The following example displays "true" or "false" in a dialog box, depending on whether the item symbol 1 exists in the Library folder folder 1:

```
alert(fl.getDocumentDOM().library.itemExists('folder 1/symbol 1'));
```

library.moveToFolder()

Description

Moves the currently selected or specified Library item to a specified folder. If the *folderPath* argument is empty, the items are moved to the top level.

Usage

```
library.moveToFolder( folderPath [, itemToMove [, bReplace] ] )
```

Arguments

The *folderPath* argument is a String that specifies the path to the folder in the form "FolderName" or "FolderName/FolderName". To move an item to the top level, specify "" (an empty string) for *folderPath*.

The optional *itemToMove* argument is a String that specifies the name of the item to move. If *itemToMove* is not specified, the currently selected items are moved.

The optional *bReplace* argument is a Boolean value. If an item with the same name already exists, specifying true for the *bReplace* argument replaces the existing item with the item being moved. If false, the name of the dropped item is changed to a unique name. The default value is false.

Returns

A Boolean value: `true` if the item is successfully moved; `false` otherwise.

Example

The following example moves the item `Symbol 1` to the Library folder `new` and replaces the item in that folder with the same name:

```
fl.getDocumentDOM().library.moveToFolder("new", "Symbol 1",true);
```

library.newFolder()

Description

The `library.newFolder()` argument creates a new folder with the specified name, or a default name (`"untitled folder #"`) if no *`folderName`* argument is provided, in the currently selected folder.

Usage

```
library.newFolder( [folderPath] )
```

Arguments

The optional *`folderPath`* argument is a String that specifies the name of the folder to be created. If it is specified as a path, and the path doesn't exist, the path is created.

Returns

A Boolean value: `true` if folder is created successfully; `false` otherwise.

Example

The following example creates two new Library folders; the `second` folder is a subfolder of the `first` folder:

```
fl.getDocumentDOM().library.newFolder("first/second");
```

library.renameItem()

Description

Renames the currently selected Library item (works only when one item is selected) in the Library panel.

Usage

```
library.renameItem(name)
```

Arguments

The *`name`* argument is a String that specifies a new name for the Library item.

Returns

A Boolean value: `true` if the name of the item changes successfully. If multiple items are selected, the return value is `false` to match UI behavior.

Example

The following example renames the currently selected Library item to `new name`:

```
fl.getDocumentDOM().library.renameItem("new name");
```

library.selectAll()

Description

Selects or deselects all items in the Library.

Usage

```
library.selectAll( [bSelectAll] )
```

Arguments

The optional *bSelectAll* argument is a Boolean value that specifies whether to select or deselect all items in the Library. Omit this argument or use the default value of `true` to select all the items in the Library; pass `false` to deselect all Library items.

Returns

Nothing.

Example

The following example selects all the items in the Library:

```
fl.getDocumentDOM().library.selectAll();
```

The following example deselects all the items in the Library:

```
fl.getDocumentDOM().library.selectAll(false);
```

library.selectItem()

Description

The `library.selectItem()` method selects a specified Library item.

Usage

```
library.selectItem( namePath [, bReplaceCurrentSelection [, bSelect] ] )
```

Arguments

The *namePath* argument is a String that specifies the name of the item. If the item is in a folder, specify its name and path using slash notation.

The optional *bReplaceCurrentSelection* argument is a Boolean value that specifies whether to replace the current selection or add the item to the current selection. The default value is `true` (replace current selection).

The optional *bSelect* argument is a Boolean value that specifies whether to select or deselect an item. The default value is `true` (select).

Returns

A Boolean value: `true` if the specified item exists, `false` otherwise.

Example

The following example changes the current selection in the Library to `symbol 1` inside `untitled folder 1`:

```
fl.getDocumentDOM().library.selectItem("untitled folder 1/symbol 1");
```

The following example extends what is currently selected in the Library to include `symbol 1` inside `untitled folder 1`:

```
fl.getDocumentDOM().library.selectItem("untitled folder 1/symbol 1", false);
```

The following example deselects `symbol 1` inside `untitled folder 1` and does not change other selected items:

```
fl.getDocumentDOM().library.selectItem("untitled folder 1/symbol 1", true,
  false);
```

library.selectNone()

Description

Deselects all the Library items.

Arguments

None.

Returns

Nothing.

Example

The following example deselects all Library items:

```
fl.getDocumentDOM().library.selectNone();
```

library.setItemProperty()

Description

Sets the property for all selected Library items (ignoring folders).

Usage

```
library.setItemProperty( property, value )
```

Arguments

The *property* argument is a String that is the name of the property to set. For a list of properties, see the Item object and its subclasses.

The *value* argument is the value to assign to the specified property.

Returns

Nothing.

Example

The following example assigns the value `movieclip` to the `itemType` property for the selected Library items.

```
fl.getDocumentDOM().library.setItemProperty("itemType", "movieclip");
```

library.updateItem()

Description

The `library.updateItem()` method updates the specified item.

Usage

```
library.updateItem( [namePath] )
```

Arguments

The optional *`namePath`* argument is a String that specifies the name of the item. If the item is in a folder, specify its name and path using slash notation. This is the same as right-clicking on an item and selecting Update from the menu in the UI. If no name is provided, the current selection is updated.

Returns

A Boolean value: `true` if Flash updated the item successfully; `false` otherwise.

Example

The following example displays a dialog box that shows whether the selected item was updated (`true`) or not (`false`):

```
alert(fl.getDocumentDOM().library.updateItem());
```

Properties

Property	Data type	Value
`items`	array	An array of item objects in the Library The following example stores the array of all library items in the *`itemArray`* variable: `var itemArray = fl.getDocumentDOM().library.items;`

LinkedVideoInstance

Description

The LinkedVideoInstance object is a subclass of Instance. There are no unique methods or properties of LinkedVideoInstance.

Math

Description

The Math object is available as a read-only property of the flash object.

The Math object has methods to perform common mathematical operations.

Methods

The Math object uses the following methods:

```
Math.concatMatrix()
Math.invertMatrix()
Math.pointDistance()
```

Math.concatMatrix()

Description

Performs a matrix concatenation and returns the result.

Usage

```
Math.concatMatrix( mat1, mat2 )
```

Arguments

The *mat1* argument is the first matrix in the application. It must be an object with fields `a`, `b`, `c`, `d`, `tx`, and `ty`.

The *mat2* argument is the second matrix in the application. It must be an object with fields `a`, `b`, `c`, `d`, `tx`, and `ty`.

Returns

A concatenated object matrix.

Example

The following example stores the currently selected object in the *elt* variable , multiplies the object matrix by the view matrix, and stores that value in the *mat* variable:

```
var elt = fl.getDocumentDOM().selection[0];
var mat = fl.Math.concatMatrix( elt.matrix , fl.getDocumentDOM().viewMatrix );
```

Math.invertMatrix()

Description

Returns the inverse of the specified matrix.

Usage

```
Math.invertMatrix( mat )
```

Arguments

The *mat* argument indicates the matrix object. It must have the following fields: `a`, `b`, `c`, `d`, `tx`, and `ty`.

Returns

An inverse object matrix.

Example

The following example stores the currently selected object in the *elt* variable, assigns that matrix to the *mat* variable, and stores the inverse of the matrix in the *inv* variable:

```
var elt = fl.getDocumentDOM().selection[0];
var mat = elt.matrix;
var inv = fl.Math.invertMatrix( mat );
```

Math.pointDistance()

Description

Computes the distance between two points.

Usage

```
Math.pointDistance( pt1, pt2 )
```

Arguments

The *pt1* and *pt2* arguments specify the points between which distance is measured.

Returns

A float value for the distance between the points.

Example

The following example stores the value for the distance between *pt1* and *pt2* in the *dist* variable:

```
var pt1 = {x:10, y:20}
var pt2 = {x:100, y:200}
var dist = fl.Math.pointDistance(pt1, pt2);
```

Properties

This object has no unique properties.

Matrix

Description

The `Matrix` object represents a transformation matrix.

Methods

This object has no unique methods.

Properties

Property	Data type	Value

`a`	float	Indicates the (0,0) element of the transformation matrix. Represents the scale factor of the *x*-axis of the object.
		The properties `a` and `d` in a matrix represent scaling. In the following example the values are set to 2 and 3, respectively, to scale the selected object's width to 2*width, and height to 3*height:
		`var mat = fl.getDocumentDOM().selection[0].matrix;`
		`mat.a = 2;`
		`mat.d = 3;`
		`fl.getDocumentDOM().selection[0].matrix = mat;`
		You can rotate an object by setting the `a`, `b`, `c`, and `d` matrix properties relative to one another, where `a = d` and `b = -c`. For example, values of 0.5, 0.8, -0.8, and 0.5 rotate the object 60 degrees:
		`var mat = fl.getDocumentDOM().selection[0].matrix;`
		`mat.a = 0.5;`
		`mat.b = 0.8;`
		`mat.c = 0.8*(-1);`
		`mat.d = 0.5;`
		`fl.getDocumentDOM().selection[0].matrix = mat;`
		You can set `a = d = 1` and `c = b = 0` to reset the transform, scale, skew, or rotate the object back to its original shape.

`b`	float	Indicates the (0,1) element in the matrix. Represents the veritcal skew of a shape; moves the right edge along the vertical axis.
		The `b` and `c` properties in a matrix represent skewing.
		In the following example, you can set `b` and `c` to -1 and 0, respectively, which skews the object at a 45 degree vertical angle:
		`var mat = fl.getDocumentDOM().selection[0].matrix;`
		`mat.b = -1;`
		`mat.c = 0;`
		`fl.getDocumentDOM().selection[0].matrix = mat;`
		You can set `b = 0` and `c = 0` to skew the object back to its original shape.

`c`	float	Indicates the (1,0) element in the matrix. Skews the object by moving the bottom edge along a horizontal axis.
		The `b` and `c` properties in a matrix represent skewing.
		In the following example, you can set `b` and `c` to -1 and 0, respectively, which skews the object at a 45 degree vertical angle:
		`var mat = fl.getDocumentDOM().selection[0].matrix;`
		`mat.b = -1;`
		`mat.c = 0;`
		`fl.getDocumentDOM().selection[0].matrix = mat;`
		You can set `b = 0` and `c = 0` to skew the object back to its original shape.

Property	Data type	Value

d	float	Indicates the (1,1) element in the matrix. Represents the scale factor of the *y*-axis of the object.
		The properties a and d in a matrix represent scaling. In the following example, the values are set to 2 and 3, respectively, to scale the selected object's width to 2*width, and height to 3*height:
		`var mat = fl.getDocumentDOM().selection[0].matrix;`
		`mat.a = 2;`
		`mat.d = 3;`
		`fl.getDocumentDOM().selection[0].matrix = mat;`
		You can rotate an object by setting the a, b, c, and d matrix properties relative to one another, where a = d and b = -c. For example, values of 0.5, 0.8, -0.8, and 0.5 rotate the object 60 degrees:
		`var mat = fl.getDocumentDOM().selection[0].matrix;`
		`mat.a = 0.5;`
		`mat.b = 0.8;`
		`mat.c = 0.8*(-1);`
		`mat.d = 0.5;`
		`fl.getDocumentDOM().selection[0].matrix = mat;`
		You can set a = d = 1 and c = b = 0 to reset the transform, scale, skew, or rotate the object back to the original shape.

tx	float	References the *x*-axis location of a symbol's registration point or the center of a shape. It defines the *x* translation of the transformation.
		You can move an object by setting the tx and ty matrix properties.
		In the following example, setting tx and ty to 0 will move the registration point of the object to point 0,0 in the document:
		`var mat = fl.getDocumentDOM().selection[0].matrix;`
		`mat.tx = 0;`
		`mat.ty = 0;`
		`fl.getDocumentDOM().selection[0].matrix = mat;`

ty	float	References the *y*-axis location of a symbol's registration point or the center of a shape. It defines the *y* translation of the transformation.
		You can move an object by setting the tx and ty matrix properties.
		In the following example, setting tx and ty to 0 will move the registration point of the object to point 0,0 in the document:
		`var mat = fl.getDocumentDOM().selection[0].matrix;`
		`mat.tx = 0;`
		`mat.ty = 0;`
		`fl.getDocumentDOM().selection[0].matrix = mat;`

outputPanel

Description

This object represents the Output panel, which is used to display troubleshooting information such as syntax errors.

Methods

The outputPanel object uses the following methods:

```
outputPanel.clear()
outputPanel.save()
outputPanel.trace()
```

outputPanel.clear()

Description

Clears the contents of the Output panel. You can use this method in a batch processing application to clear out a listing of errors or save them incrementally by using this method with `outputPanel.save()`.

Usage

```
outputPanel.clear()
```

Arguments

None.

Returns

Nothing.

Example

The following example clears the current contents of the Output panel:

```
fl.outputPanel.clear();
```

outputPanel.save()

Description

Saves the contents of the Output panel to a local text file, in UTF-8 encoding. The local filename is specified as a URI. Optionally, the contents can be appended to the contents of a local file, rather than being overwritten. If the URI is invalid or unspecified, an error is reported.

This method is useful for batch processing. You can create a JSFL file, which compiles several components. Any compile errors appear in the Output panel, and you can use this method to save the resulting errors to a text file, which can be auto-parsed by the build system in use.

Usage

```
outputPanel.save( fileURI [, bAppendToFile] )
```

Arguments

The *fileURI* argument is a String that specifies the local file to contain the Output panel's contents.

The optional *bAppendToFile* argument, if it has a value of `true`, appends the Output panel's contents to the output file. If *bAppendToFile* is `false`, the method overwrites the output file if it already exists. The default value is `false`.

Returns

Nothing.

Example

The following example saves the Output panel's contents to the batch.log file:

```
fl.outputPanel.save("file:///c|/tests/batch.log");
```

outputPanel.trace()

Description

Adds a line to the contents of the Output panel, terminated by a new line. This method shows the Output panel if it is not already visible. This method duplicates the functionality in `fl.trace`.

Usage

```
outputPanel.trace( message )
```

Arguments

The *message* argument is a String that contains the text to add to the Output panel.

Returns

Nothing.

Example

The following example writes "hello world" to the Output panel:

```
fl.outputPanel.trace("hello world");
```

Properties

This object has no unique properties.

Parameter

Description

The Parameter object type is accessed from the `Screen.parameters` array (which corresponds to the Screen Property inspector in the Flash UI) or by the `ComponentInstance.parameters` array (which corresponds to the Component Property inspector in the Flash UI).

Methods

You can use the following methods with the parameter object:

```
parameter.insertItem()
parameter.removeItem()
```

parameter.insertItem()

Description

If a parameter is a list, object, or array, the `value` property is an array. Use this method to insert a value into the array.

Usage

```
parameter.insertItem( index, name, value, type )
```

Arguments

The *`index`* argument is a zero-based integer index that indicates where the item will be inserted into the list, object, or array. If the index is 0, the item goes at the beginning of the list. If index value is greater than the list size, the new item is inserted at the end of the array.

The *`name`* argument is a String that specifies the name of the item to insert. This is a required argument for object parameters.

The *`value`* argument is a String that specifies the value of the item to insert.

The *`type`* argument is a String that specifies the type of item to insert.

Returns

Nothing.

Example

The following example inserts the value of "New Value" into the labelPlacement parameter.

```
//Select an instance of a button component on stage
var parms = fl.getDocumentDOM().selection[0].parameters;
parms[2].insertItem(0, "name", "New Value", "String");
var values = parms[2].value;
for(var prop in values){
     fl.trace("labelPlacement parameter value = " + values[prop].value);
}
```

The following example inserts the value of "New Value" into the autoKeyNav parameter.

```
//Open a Presentation document
var parms = fl.getDocumentDOM().screenOutline.screens[1].parameters;
parms[0].insertItem(0, "name", "New Value", "String");
var values = parms[0].value;
for(var prop in values){
 fl.trace("autoKeyNav value = " + values[prop].value);
}
```

parameter.removeItem()

Description

Use this method to remove an element of the list, object or array type of a Screen or Component parameter.

Usage

```
parameter.removeItem( index )
```

Arguments

The *index* argument is the zero-based integer index of the item to remove from the Screen or Component property.

Returns

Nothing.

Example

The following example removes the element at index 1 from the labelPlacement parameter of a component.

```
//Select an instance of a button component on the stage
var parms = fl.getDocumentDOM().selection[0].parameters;
var values = parms[2].value;
fl.trace("--Original--");
for(var prop in values){
  fl.trace("labelPlacement value = " + values[prop].value);
}
parms[2].removeItem(1);

var newValues = parms[2].value;
fl.trace("--After Remove Item--");
for(var prop in newValues){
  fl.trace("labelPlacement value = " + newValues[prop].value);
}
```

The following example removes the element at index 1 from the autoKeyNav parameter of a screen.

```
//Open a Presentation document
var parms = fl.getDocumentDOM().screenOutline.screens[1].parameters;
var values = parms[0].value;
fl.trace("--Original--");
for(var prop in values){
  fl.trace("autoKeyNav value = " + values[prop].value);
}
parms[0].removeItem(1);
```

```
var newValues = parms[0].value;
fl.trace("--After Remove Item--");
for(var prop in newValues){
  fl.trace("autoKeyNav value = " + newValues[prop].value);
}
```

Properties

Property	Data type	Value

`category`
read only
	string	The `category` property for the screen parameter or componentInstance parameter. Can be used for alternative ways to present a list of parameters. This is not available via the authoring tool UI.
		Example:
		`//for a component`
		`var cat =`
		`  fl.getDocumentDOM().selection[0].parameters[0].category;`
		`//for a screen`
		`var cat =`
		`  fl.getDocumentDOM().screenOutline.screens[1].parameters[0].`
		`  category;`

`listIndex`	int	The value of the selected list item. Valid only if the valueType parameter is `"list"`.
		The following example sets the `listIndex` property to a value of 2 and displays it in the Output panel:
		`var parms =`
		`  fl.getDocumentDOM().screenOutline.screens[1].parameters;`
		`parms[4].listIndex = 2;`
		`fl.trace(" listIndex = " + parms[4].listIndex);`

`name`
read only
	string	The name of the parameter.
		The following example displays the name of the fifth parameter for the selected component:
		`var parms = fl.getDocumentDOM().selection[0].parameters;`
		`fl.trace("name: " + parms[4].name);`
		`The following example displays the name of the fifth parameter`
		`  for the specified screen:`
		`var parms =`
		`  fl.getDocumentDOM().screenOutline.screens[1].parameters;`
		`  fl.trace("name: " + parms[4].name);`

Property	Data type	Value

`value`	Depends on the type of parameter	This property corresponds to the Value field in the Component parameters PI, the Component inspector, or the Screens PI. The type of the value property is determined by the valueType property for the parameter (see the valueType parameter, below).
		Example:
		`var parms =`
		`  fl.getDocumentDOM().screenOutline.screens[1].parameters;`
		`fl.trace("value: " + parms[4].value);`
		`var parms =`
		`  fl.getDocumentDOM().screenOutline.screens[1].parameters;`
		`parms[4].listIndex = 2;`
		
		`for (var i = 0; i < parms.length; i++) {`
		`  fl.trace(i + " name = " + parms[i].name);`
		`  fl.trace(i + " valueType = " + parms[i].valueType);`
		`  fl.trace(i + " value = " + parms[i].value);`
		
		`  if (parms[i].valueType == "List") {`
		`  fl.trace(i + " listIndex = " + parms[i].listIndex);`
		`  fl.trace(i + " List value = " +`
		`parms[i].value[parms[i].listIndex].value);`
		`  }`
		`  fl.trace("");`
		`}`

`valueType` **read only**	string	The type of the screen or component parameter. The type can be one of the following values: "Default" "Array" "Object" "List" "String" "Number" "Boolean" "Font Name" "Color" "Collection" "Web Serivce URL" "Web Service Operation"
`verbose` **read only**	int	Specifies if the parameter is displayed in the Components Parameters PI, the Screens Parameters PI, or the Components Inspector. Contains a value of 0 (non-verbose) or 1 (verbose).

Path

Description

The `Path` object is used to define a sequence of straight line and/or curved line segments, which are typically used when creating extensible tools. The following example shows an instance of a `Path` object being returned from the `Flash` object:

```
path = fl.drawingLayer.newPath().
```

Methods

You can use the following methods with the `Path` object:

```
path.addCubicCurve()
path.addCurve()
path.addPoint()
path.clear()
path.close()
path.makeShape()
path.newContour()
```

path.addCubicCurve()

Description

Appends a cubic Bézier curve segment to the path.

Usage

```
path.addCubicCurve( xAnchor, yAnchor, x2, y2, x3, y3, x4, y4 )
```

Arguments

The *xAnchor* argument is a floating point number that specifies the *x* position of the first control point.

The *yAnchor* argument is a floating point number that specifies the *y* position of the first control point.

The *x2* argument is a floating point number that specifies the *x* position of the second control point.

The *y2* argument is a floating point number that specifies the *y* position of the second control point.

The *x3* argument is a floating point number that specifies the *x* position of the third control point.

The *y3* argument is a floating point number that specifies the *y* position of the third control point.

The *x4* argument is a floating point number that specifies the *x* position of the fourth control point.

The *y4* argument is a floating point number that specifies the *y* position of the fourth control point.

Returns

Nothing.

Example

The following example creates a new path, stores it in the *myPath* variable, and assigns the curve to the path:

```
var myPath = fl.drawingLayer.newPath();
myPath.addCubicCurve(0, 0, 10, 20, 20, 20, 30, 0);
```

path.addCurve()

Description

Appends a quadratic Bézier segment to the path.

Usage

```
path.addCurve( xAnchor, yAnchor, x2, y2, x3, y3 )
```

Arguments

The *xAnchor* argument is a floating point value that specifies the *x* position of the first control point.

The *yAnchor* argument is a floating point value that specifies the *y* position of the first control point.

The *x2* argument is a floating point value that specifies the *x* position of the second control point.

The *y2* argument is a floating point value that specifies the *y* position of the second control point.

The *x3* argument is a floating point value that specifies the *x* position of the third control point.

The *y3* argument is a floating point value that specifies the *y* position of the third control point.

Returns

Nothing.

Example

The following example creates a new path, stores it in the *myPath* variable, and assigns the curve to the path:

```
var myPath = fl.drawingLayer.newPath();
myPath.addCurve(0, 0, 10, 20, 20, 0);
```

path.addPoint()

Description

Adds a point to the path object.

Usage

```
path.addPoint( x, y )
```

Arguments

The *x* argument is a floating point value that specifies the *x* position of the point.

The *y2* argument is a floating point value that specifies the *y* position of the point.

Returns

Nothing.

Example

The following example creates a new path, stores it in the *myPath* variable, and assigns the new point to the path:

```
var myPath = fl.drawingLayer.newPath();
myPath.addPoint(10, 100);
```

path.clear()

Description

Removes all points from the path.

Usage

```
path.clear()
```

Arguments

None.

Returns

Nothing.

Example

The following example shows how to remove all points from a path stored in the *myPath* variable:

```
var myPath = fl.drawingLayer.newPath();
myPath.clear();
```

path.close()

Description

Appends a point at the location of the first point of the path. If the path has no points, no points are added.

Usage

```
path.close()
```

Arguments

None.

Returns

Nothing.

Example

```
var myPath = fl.drawingLayer.newPath();
myPath.close();
```

path.makeShape()

Description

Creates a shape on the Stage using the current stroke and fill settings. The path is cleared after creating the shape. This method has two optional arguments to suppress the fill and/or stroke of the resulting shape object. If the arguments are omitted, or set to `false`, the current values for fill and stroke are used.

Usage

```
path.makeShape( [bSupressFill [, bSupressStroke ] ] )
```

Arguments

The optional *`bSuppressFill`* argument is a Boolean value that, if set to `true`, suppresses the fill that would be applied to the shape; the default value is `false`.

The optional *`bSupressStroke`* argument is a Boolean value that, if set to `true`, suppresses the stroke that would be applied to the shape; the default value is `false`.

Returns

Nothing.

Example

```
var myPath = fl.drawingLayer.newPath();
myPath.makeShape(true, false);
```

path.newContour()

Description

Starts a new contour in the path.

Usage

```
path.newContour()
```

Arguments

None.

Returns

Nothing.

Example

The following example adds a contour before defining a new point on the path referenced by the *`myPath`* variable:

```
var myPath = fl.drawingLayer.newPath();
myPath.addPoint(...);
//...
myPath.close();
myPath.newContour();
myPath.addPoint(x,y);
//....
```

Properties

Property	Data type	Value

`nPts`
read only
	int	An integer representing the number of points in the path.
		The following example displays the number of points in the path referenced by the *`myPath`* variable in the Output panel:
		`var myPath = fl.drawingLayer.newPath();`
		`var numOfPoints = myPath.nPts; fl.trace("The number of points in path: " + numOfPoints);`

Screen

Description

The Screen object represents a single screen in a slide or form document. The Screen object contains properties related to the slide or form. The array of all the Screen objects in the document can be accessed by:

```
fl.getDocumentDOM().screenOutline.screens
```

Methods

This object has no unique methods.

Properties

The Screen object has the following properties:

Properties	Data type	Value

`accName`	string	Equivalent to the Name field in the Accessibility panel. Screen readers identify objects by reading the name aloud.
		The following example stores the value of the name of the object in the *`theName`* variable:
		`var theName = fl.getDocumentDOM().selection[0].accName;`
		
		The following example sets the name of the object to "`Home Button`":
		`fl.getDocumentDOM().selection[0].accName = 'Home Button';`

`childScreens`
read only
	array	The array of child screens for this screen; it is empty if there are no child screens.
		The following example checks to see if the current document is a slide or form document, and if it is, stores the array of childScreens in the *`myChildren`* variable and displays their names in the Output panel:
		`var myChildren = new Array();`
		`if(fl.getDocumentDOM().allowScreens) {`
		`var myParent =`
		`  fl.getDocumentDOM().screenOutline.rootScreen.name`
		`for (i in`
		`  fl.getDocumentDOM().screenOutline.rootScreen.childScreen`
		`  s) {`
		`myChildren.push("`
		`  "+fl.getDocumentDOM().screenOutline.rootScreen.childScre`
		`  ens[i].name);`
		`}`
		`fl.trace(" The child screens of "+myParent+" are`
		`  "+myChildren+". ");`
		`}`

Properties	Data type	Value

description	string	Equivalent to the Description field on the Accessibility panel. The description is read by the screen reader.
		The following example gets the description of the object and stores it in the *theDescription* variable:
		`var theDescription =`
		`  fl.getDocumentDOM().selection[0].description;`
		
		The following example sets the description of the object to "This is Screen 1":
		`fl.getDocumentDOM().selection[0].description= "This is`
		`  Screen 1";`

forceSimple	Boolean	Enables and disables the children of the object to be accessible. This is equivalent to the inverse logic of the Make Child Objects Accessible setting in the Accessibility panel.
		If `forceSimple` is `true`, it is the same as the Make Child Object Accessible option being unchecked.
		If `forceSimple` is `false`, it is the same as the Make Child Object Accessible option being checked.
		The following example stores the value of `forceSimple` in the *areChildrenAccessible* variable (to see if the children of the object are accessible):
		`var areChildrenAccessible =`
		`  !fl.getDocumentDOM().selection[0].forceSimple;`
		
		The following example sets the children of the object to be accessible:
		`fl.getDocumentDOM().selection[0].forceSimple = false;`

hidden	Boolean	The hidden property of the screen specifies visibility of a screen. A screen with the hidden property set to `true` is not visible in any other screen.
		The following example checks to see if the first screen in the outline is hidden and change the visibility of the screen accordingly. Then, a message in the Output panel shows what the visibility of the screen was before the change:
		`if (fl.getDocumentDOM().screenOutline.screens[0].hidden) {`
		`fl.getDocumentDOM().screenOutline.setScreenProperty("hidden`
		`  ", false);`
		`fl.trace(fl.getDocumentDOM().screenOutline.screens[0].name+`
		`  " had it's 'hidden' property set to 'false'");`
		`} else {`
		`fl.getDocumentDOM().screenOutline.setScreenProperty("hidden`
		`  ", true);`
		`fl.trace(fl.getDocumentDOM().screenOutline.screens[0].name+`
		`  " had it's 'hidden' property set to 'true'");`
		`}`

Properties	Data type	Value

`instanceName`
read only
	string	The name used to access the object from ActionScript.
		The following example checks to see if the current document allows screens (because it is a slide or form document). Then it assigns the `instanceName` value of the first child screen in the array to the *`myInstanceName`* variable and opens the Output panel to display the `instanceName` of the screen:
		`var myChildren = new Array();`
		`if(fl.getDocumentDOM().allowScreens) {`
		`var myInstanceName =`
		`fl.getDocumentDOM().screenOutline.rootScreen.childScreen`
		`s[0].instanceName;`
		`fl.trace(" The instanceName is "+myInstanceName+". ");`
		`}`

`name`
read only
	string	The name of the screen.
		The following example checks to see if the current document allows screens (because it is a slide or form document). Then it assigns the `name` value of the first child screen in the array to the *`myName`* variable and opens the Output panel to display the `name` of the screen:
		`var myChildren = new Array();`
		`if(fl.getDocumentDOM().allowScreens) {`
		`var myName =`
		`fl.getDocumentDOM().screenOutline.rootScreen.childScreen`
		`s[0].name;`
		`fl.trace("The name of the screen is "+myName+". ");`
		`}`

`nextScreen`
read only
	object	The next peer screen in the parent's childScreen array. If there's not a peer screen, the value should be `null`.
		The following example first checks to see if the current document is a slide or form document, and if it is, retrieves and displays the sequence of screens in the Output panel:
		`if(fl.getDocumentDOM().allowScreens) {`
		`var myCurrent =`
		`fl.getDocumentDOM().screenOutline.rootScreen.childScreen`
		`s[0].name;`
		`var myNext =`
		`fl.getDocumentDOM().screenOutline.rootScreen.childScreen`
		`s[0].nextScreen.name;`
		`fl.trace(" The next screen to "+myCurrent+" is "+myNext+".`
		`");`
		`}`

`parameters`
read only
	object	The ActionScript properties that are accessible from the Screens Property inspector.
		The following example stores the parameters for the second screen in the outline to the *`parms`* variable, and then assigns the "`some value`" value to the first property:
		`var parms =`
		`fl.getDocumentDOM().screenOutline.screens[1].parameters;`
		`parms[0].value = "some value";`

Properties	Data type	Value

`parentScreen`
read only
	object	The parent screen for nested screens. If `parentScreen` is `null`, the screen is a top level screen.
		The following example stores the values for the `childScreen` and `parentScreen` properties in variables, and then displays those values and their parent/child relationship in the Output panel:
		`if(fl.getDocumentDOM().allowScreens) {`
		`var myCurrent =`
		`fl.getDocumentDOM().screenOutline.rootScreen.childScreen`
		`s[1].name;`
		`var myParent =`
		`fl.getDocumentDOM().screenOutline.rootScreen.childScreen`
		`s[1].parentScreen.name;`
		`fl.trace(" The parent screen to "+myCurrent+" is`
		`"+myParent+". ");`
		`}`

`prevScreen`
read only
	object	The previous peer screen in the parent's childScreen array. If there are no peer screens, the value should be `null`.
		The following example checks to see if the current document is a slide or form document, and if it is, retrieves and displays the sequence of screens in the Output panel:
		`if(fl.getDocumentDOM().allowScreens) {`
		`var myCurrent =`
		`fl.getDocumentDOM().screenOutline.rootScreen.childScreen`
		`s[1].name;`
		`var myNext =`
		`fl.getDocumentDOM().screenOutline.rootScreen.childScreen`
		`s[1].prevScreen.name;`
		`fl.trace(" The previous screen to "+myCurrent+" is`
		`"+myNext+". ");`
		`}`

`silent`	Boolean	Enable/disable the accessibility of the object. This is equivalent to the inverse logic of the Make Object Accessible setting in the Accessibility panel. Setting the `silent` property to `true` is the same as having the Make Object Accessible option unchecked in the Accessibility panel. Setting the `silent` property to `false` is the same as having the Make Object Accessible option checked in the Accessibility panel.
		The following example gets the `silent` value of the object (to see if the object is accessible):
		`var isSilent = fl.getDocumentDOM().selection[0].silent;`
		
		The following example sets the object to be accessible:
		`fl.getDocumentDOM().selection[0].silent = false;`

Properties	Data type	Value

`tabIndex`	int	This is equivalent to the Tab Index field on the Accessibility panel. This value lets you create a tab order in which objects are accessed when the user presses the Tab key.
		The following example gets the `tabIndex` of the object:
		`var theTabIndex = fl.getDocumentDOM().selection[0].tabIndex;`
		
		The following example sets the `tabIndex` of the object to 1:
		`fl.getDocumentDOM().selection[0].tabIndex = 1;`

`timeline`
read only
	object	The `timeline` object for the screen.
		The following example gets the screenOutline propertyof the current slide document, assigns the array of `timeline` properties for the first screen to *`myArray`* and displays those properties in the Output panel:
		`myArray = new Array();`
		`if(fl.getDocumentDOM().screenOutline) {`
		`for(i in fl.getDocumentDOM().screenOutline.screens[0].timeline) {`
		`myArray.push(" "+i+" : "+fl.getDocumentDOM().screenOutline.screens[0].timeline[i]+" ") ;`
		`}`
		`fl.trace("Here are the properties of the screen named "+fl.getDocumentDOM().screenOutline.screens[0].name+": "+myArray);`
		`}`

screenOutline

Description

The `screenOutline` object represents the group of screens in a slide or form document. The object is accessed by using:

```
fl.getDocumentDOM().screenOutline
```

The `screenOutline` object exists only if the document is a slide or form document, so before accessing the property, use `dom.allowScreens()` to verify that a Screens document exists. For example:

```
if(fl.getDocumentDOM().allowScreens) {
var myName =
  fl.getDocumentDOM().screenOutline.rootScreen.childScreens[0].name;
fl.trace("The name of the screen is "+myName+". ");
}
```

Methods

You can use the following methods with the `screenOutline` object:

```
screenOutline.copyScreenFromFile()
screenOutline.deleteScreen()
screenOutline.duplicateScreen()
screenOutline.getSelectedScreens()
screenOutline.insertNestedScreen()
screenOutline.insertScreen()
screenOutline.moveScreen()
screenOutline.renameScreen()
screenOutline.setCurrentScreen()
screenOutline.setScreenProperty()
screenOutline.setSelectedScreens ()
```

screenOutline.copyScreenFromFile()

Description

Inserts all the screens or a named screen and its children from the URI document.

Usage

```
screenOutline.copyScreenFromFile( fileURI [, screenName] )
```

Arguments

The *`fileURI`* argument is a String that specifies a filename for the authoring file that contains the screens to copy into the document, in URI format (for example, `"file:///C:/assets.fla"`).

The optional *`screenName`* argument is the name of the screen to copy. If the *`screenName`* argument is present, Flash will copy that screen and its children. If the *`screenName`* is not specified, Flash will copy the whole document.

Returns

Nothing. If the file is not found or is not a valid FLA file, or if the specified screen is not found, an error is reported and the script is cancelled.

Example

The following example copies the screen titled Screen 5 into the current document:

```
fl.getDocumentDOM().screenOutline.copyScreenFromFile("file:///C:/assets.fla",
  "Screen 5");
```

screenOutline.deleteScreen()

Description

Deletes the specified screen and its children. If no screen is specified, the method deletes the currently selected screen and its children.

Usage

```
screenOutline.deleteScreen( [screenName] )
```

Arguments

The optional *screenName* argument is a String that specifies the name of the screen to be deleted. If no screen is specified, the currently selected screen, and its children, are deleted.

Returns

Nothing.

Example

The following example removes the screen named apple and all its children:

```
fl.getDocumentDOM().screenOutline.deleteScreen("apple");
```

screenOutline.duplicateScreen()

Description

Duplicates the specified screen, giving the duplicate a default name (by appending _copy to the original name). If the *screenName* argument is not specified, the currently selected screens are duplicated.

Usage

```
screenOutline.duplicateScreen( [screenName] )
```

Arguments

The optional *screenName* argument is a String value that specifies the screen name to duplicate. After the screen is duplicated, the duplicated screen name will be the default name such as Screen_copy, Screen_copy2, and so on.

Returns

A Boolean value: `true` if the screen is successfully duplicated; `false` otherwise.

Example

The following example duplicates a screen named apple:

```
fl.getDocumentDOM().screenOutline.duplicateScreen("apple");
```

screenOutline.getSelectedScreens()

Description

Returns an array of screen objects that are currently selected in the screenOutline.

Usage

```
screenOutline.getSelectedScreens()
```

Arguments

None.

Returns

An array of selected screen objects.

Example

The following example stores the selected screen objects in the *myArray* variable and then displays the screen names in the Output panel:

```
var myArray = fl.getDocumentDOM().screenOutline.getSelectedScreens();
for (var i in myArray) {
fl.trace(myArray[i].name)
}
```

screenOutline.insertNestedScreen()

Description

Inserts a nested screen of a specific type into a particular location in the `screenOutline`.

Usage

```
screenOutline.insertNestedScreen( [ name [, referenceScreen
  [, screenTypeName ] ] ])
```

Arguments

The optional *name* argument is a String indicating the name of the new screen to be inserted. An empty name will insert a screen with a default screen name, such as Slide *n* or Form *n* (where *n* is the first available unique number).

The optional *referenceScreen* argument is a String indicating the name of the screen into which the new screen will be inserted as a child. If this argument is not specified, the new screen is inserted as a child of the currently selected screen.

The optional *screenTypeName* argument is a String that specifies the name of the screen type to attach to the new nested screen. The screen type and classname will be set for this screen. If this argument is not specified, the type is inherited from the parent screen. Acceptable values are `"Form"` and `"Slide"`.

Returns

A Screen object.

Example

The following example inserts Slide_2 as a child of Slide_1:

```
fl.getDocumentDOM().screenOutline.insertNestedScreen("Slide_2", "Slide_1",
  "Slide");
```

screenOutline.insertScreen()

Description

Inserts a new blank screen of a specified type into the document at a specified location.

Usage

```
screenOutline.insertScreen( [name [, referenceScreen [, screenTypeName ] ] ])
```

Arguments

The optional *name* argument is a String indicating the name of the new screen to be inserted. If omitted empty name the method will insert a screen with a default screen name, such as Slide *n* or Form *n* (where *n* is the first available unique number).

The optional *referenceScreen* argument is a String indicating the name of the screen before the new screen. If omitted, the new screen is inserted after the currently selected screen. If the *referenceScreen* argument identifies a child screen, the new screen will be a peer of the child screen, and a child screen of the same parent.

The optional *screenTypeName* argument is a String that specifies the screen type to attach to the new screen. The screen type and classname will be set for this screen. Acceptable values are `"Form"` and `"Slide"`.

Returns

A Screen object.

Example

The following example inserts a form with name Shire after the screen Oak Tree:

```
fl.getDocumentDOM().screenOutline.insertScreen("Shire","Oak Tree","Form");
```

The following example inserts a slide with name Frodo after the screen Bilbo:

```
fl.getDocumentDOM().screenOutline.insertScreen("Frodo","Bilbo","Slide");
```

screenOutline.moveScreen()

Description

Moves the given screen in relation to the value of the `referenceScreen` argument; either before, after, as the first child, or as the last child.

Usage

```
screenOutline.moveScreen( screenToMove, referenceScreen, position )
```

Arguments

The *screenToMove* argument is a String that is the screen name to move.

The *referenceScreen* argument is a String that is the reference screen for the move.

The *position* argument is a String that specifies where to move the screen. Acceptable values are "before","after","firstChild", or "lastChild".

Returns

A Boolean value: true if the move was successful; false otherwise.

Example

The following example moves screen Slide_1 to the position firstChild of Slide_2:

```
fl.getDocumentDOM().screenOutline.moveScreen("Slide_1", "Slide_2",
  "firstChild");
```

screenOutline.renameScreen()

Description

Changes the screen with a specified name to a new name.

Usage

```
screenOutline.renameScreen( newScreenName [, oldScreenName
  [, bDisplayError] ] )
```

Arguments

The *newScreenName* argument is a String that specifies the new name of the screen

The optional *oldScreenName* argument is a String that specifies the name of the existing screen to change. If not specified, the name of the currently selected screen will change.

The optional *bDisplayError* argument is a Boolean value that, if set to true, displays an error message if an error occurs. This argument is false by default.

Returns

A Boolean value: true if the renaming is successful; false otherwise.

Example

The following example changes the name of Slide_1 to Intro:

```
fl.getDocumentDOM().screenOutline.renameScreen("Intro", "Slide_1");
```

screenOutline.setCurrentScreen()

Description

Sets the current selection in screenOutline to the specified screen.

Usage

```
screenOutline.setCurrentScreen( name )
```

Arguments

The *name* argument is a String that specifies the name of the screen. If the screen is a child of another screen, you do not need to indicate a path or hierarchy.

Returns

Nothing.

Example

```
fl.getDocumentDOM().screenOutline.setCurrentScreen("Slide_1");
```

screenOutline.setScreenProperty()

Description

Sets the specified property with the specified value for the selected screens.

Usage

```
screenOutline.setScreenProperty( property, value )
```

Arguments

The *property* argument is a String that specifies the property to set.

The *value* argument is the new value for the property. The type of value depends on the property being set.

Returns

Nothing.

Example

The following example changes the visibility of the currently selected screen from hidden to visible:

```
fl.getDocumentDOM().screenOutline.setScreenProperty("hidden", false);
```

screenOutline.setSelectedScreens ()

Description

Selects the specified screens in the Screen Outline pane. If multiple screens are specified, the screen with the last index value of the selection array will be focused on the Stage.

Usage

```
screenOutline.setSelectedScreens ( selection [, bReplaceCurrentSelection ] )
```

Arguments

The *selection* argument is an array of screen names to be selected in Screen Outline pane.

The optional *bReplaceCurrentSelection* argument is a Boolean value that, if `true`, lets you deselect the current selection. The default value is `true`. If `false`, Flash extends the current selection to the specified screens.

Returns

Nothing.

Example

The following example deselects any currently selected screens, and then selects screens `Slide_1`, `Slide_2`, `Slide_3`, and `Slide_4` in the Screen Outline pane:

```
myArray = new Array("Slide 1", "Slide 2", "Slide 3", "Slide 4");
fl.getDocumentDOM().screenOutline.setSelectedScreens(myArray, true);
```

Properties

Property	Data type	Value

`currentScreen`	object	The currently selected screen.
		The following example stores the `currentScreen` object in the *`myScreen`* variable "" and then displays the name of that screen in the Output panel:
		`var myScreen =`
		`  fl.getDocumentDOM().screenOutline.currentScreen;`
		`fl.trace(myScreen.name);`

rootScreen	object	The first screen in screenOutline. You can use screenOutline.rootScreen as a "shortcut" for screenOutline.screens[0].
		The following example displays the name of the first screen.
		`fl.trace(fl.getDocumentDOM().screenOutline.rootScreen.`
		`  childScreens[0].name);`

`screens`
read only
	array	The array of top level screens contained in the document.
		The following example stores the array of screen objects in the *`myArray`* variable and then displays their names in the Output panel:
		`var myArray = new Array();`
		`if(fl.getDocumentDOM().allowScreens) {`
		`for(var i in fl.getDocumentDOM().screenOutline.screens) {`
		`myArray.push("`
		`  "+fl.getDocumentDOM().screenOutline.screens[i].name);`
		`}`
		`fl.trace("The screens array contains objects whose names are:`
		`  "+myArray+". ");`
		`}`

Shape

Description

The Shape object is a subclass of the Element object.

The Shape object provides more precise control when manipulating or creating geometry on the Stage than the Drawing APIs. This finer control is necessary so that scripts can create useful effects and other drawing commands.

All Shape methods and properties that change a shape or any of its subordinate parts must be placed between `shape.beginEdit()` and `shape.endEdit()` calls to function correctly.

Methods

In addition to the Element object methods, you can use the following methods with the Shape object:

```
shape.beginEdit()
shape.deleteEdge()
shape.endEdit()
```

shape.beginEdit()

Description

Defines the start of an edit session. You must use this method before issuing any commands that change the Shape object or any of its subordinate parts.

Usage

```
shape.beginEdit()
```

Arguments

None.

Returns

Nothing.

Example

The following example takes the currently selected shape and removes the first edge in the edge array from it:

```
var shape = fl.getDocumentDOM().selection[0];
shape.beginEdit();
shape.deleteEdge(0);
shape.endEdit();
```

shape.deleteEdge()

Description

Deletes the specified edge. You must call `shape.beginEdit()` before using this method.

Usage

```
shape.deleteEdge( index )
```

Arguments

The *index* argument is a zero-based index that specifies the edge to delete from the `shape.edges` array. This method changes the length of the `shape.edges` array. See "Properties" on page 169.

Returns

Nothing.

Example

The following example takes the currently selected shape and removes the first edge in the edge array:

```
var shape = fl.getDocumentDOM().selection[0];
shape.beginEdit();
shape.deleteEdge(0);
shape.endEdit();
```

shape.endEdit()

Description

Defines the end of an edit session for the shape. All changes made to the Shape object or any of its subordinate parts will be applied to the shape. You must use this method after issuing any commands that change the Shape object or any of its subordinate parts.

Usage

```
shape.endEdit()
```

Arguments

None.

Returns

Nothing.

Example

The following example takes the currently selected shape and removes the first edge in the edge array from it:

```
var shape = fl.getDocumentDOM().selection[0];
shape.beginEdit();
shape.deleteEdge(0);
shape.endEdit();
```

Properties

In addition to the Element object properties, the following properties are available for the Shape object:

Property	Data type	Value

`contours`
read only
	array	An array of Contour objects for the shape.
		The following example stores the first contour in the contours array in the *c* variable and then stores the `halfEdge` object (see "HalfEdge" on page 117) of that contour in the *he* variable:
		`var c = fl.getDocumentDOM().selection[0].contours[0];`
		`var he = c.getHalfEdge();`

`edges`
read only
	array	An array of Edge objects
		The following example stores the array of edges of the first selected item object in the *edges* variable and then assigns the `halfEdge` object to the *hedge0* variable:
		`var edges = fl.getDocumentDOM().selection[0].edges;`
		`var hEdge0 = edges[0].getHalfEdge(0);`

`isGroup`
read only
	Boolean	If `true`, the shape is a group.
		The following example stores the first selected `item` object in the *sel* variable and then uses the `elementType` and `isGroup` properties to test if the selected item is a group:
		`var sel = fl.getDocumentDOM().selection[0];`
		`var theShapeIsReallyAGroup = (sel.elementType == "shape") &&`
		`sel.isGroup;`

`vertices`
read only
	array	An array of Vertex objects.
		The following example stores the first selected `item` object in the *someShape* variable and then displays the number of vertices for that object in the Output panel:
		`var shape = fl.getDocumentDOM().selection[0];`
		`fl.trace(someShape.vertices.length);`

SoundItem

Description

The SoundItem object is a subclass of the Item object.

Methods

There are no unique methods for the SoundItem object; see the Item object.

Properties

In addition to the Item object properties, the following properties are available for the SoundItem object:

Property	Data type	Value
`bitRate`	string	Only available for the MP3 compression type. Acceptable values are `"8kbps"`, `"16kbps"`, `"20kbps"`, `"24kbps"`, `"32kbps"`, `"48kbps"`, `"56kbps"`, `"64kbps"`, `"80kbps"`, `"112kbps"`, `"128kbps"`, `"160kbps"`. Undefined for other compression types. The following example displays the `bitRate` value in the Output panel if the specified item in the Library has MP3 compression type: `alert(fl.getDocumentDOM().library.items[0].bitRate);` Note that when an MP3 is imported to the Library, the Use imported MP3 quality Export setting is checked by default. The `bitRate` property cannot be set with this setting checked.
`bits`	string	Acceptable values are `"2 bit"`, `"3 bit"`, `"4 bit"`, `"5 bit"` when the type of compression is ADPCM. The following example displays the `bits` value in the Output panel if the currently selected item in the Library has ADPCM compression type: `alert(fl.getDocumentDOM().library.items[0].bits);`

`compressionType`	string	Acceptable values are `"Default"`, `"ADPCM"`, `"MP3"`, `"Raw"`, and `"Speech"`.
		The following example changes an item in the Library to compression type Raw:
		`fl.getDocumentDOM().library.items[0].compressionType = "Raw";`
		The following example changes a selected item's compression type to Speech:
		`fl.getDocumentDOM().library.getSelectedItems()[0].compressionType = "Speech";`

`convertStereoToMono`	Boolean	Only available for MP3 and Raw compression types. Setting this to `true` converts a stereo sound to mono; `false` leaves it as stereo. For MP3 compression type, if the bitRate is less than 20 Kbps, this property is ignored and forced to `true`. The following example converts an item in the Library to mono, only if the item has MP3 or RAW compression type: `fl.getDocumentDOM().library.items[0].convertStereoToMono = true;`

Property	Data type	Value
`quality`	string	Only available for MP3 compression type. This value sets the playback quality. Acceptable values are `"Fast"`, `"Medium"`, `"Best"`. The following example sets the playback quality of an item in the Library to Best, if the item has MP3 compression type: `fl.getDocumentDOM().library.items[0].quality = "Best";` Note that when an MP3 is imported to the Library, the Use imported MP3 quality Export setting is checked by default. The `bitRate` property cannot be set with this setting checked.
`sampleRate`	string	Only available for ADPCM, Raw and Speech compression types. This value sets the sample rate for the audio clip. Acceptable values are `"5 kHz"`, `"11 kHz"`, `"22 kHz"`, `"44 kHz"`. The following example sets the sample rate of an item in the Library to 5 kHz, if the item has ADPCM, Raw or Speech compression type: `fl.getDocumentDOM().library.items[0].sampleRate = "5 kHz";`
`useImportedMP3Quality`	Boolean	If `true`, all other properties are ignored and the imported MP3 quality is used. The following example sets an item in the Library to use the imported MP3 quality: `fl.getDocumentDOM().library.items[0].useImportedMP3Quality = true;`

Stroke

Description

The Stroke object contains all the settings for a stroke, including the custom settings. This object represents the information contained in the Stroke Properties inspector. Using the Stroke object together with the `document.setCustomStroke()` method, you can change the stroke settings for the Toolbar, the Properties Inspector, and the current selection. You can also get the stroke settings of the Toolbar and Properties Inspector, or of the current selection, by using the `document.getCustomStroke()` method.

This object always has the following four properties: `style`, `thickness`, `color`, and `breakAtCorners`. Other properties can be set, depending on the value of the `style` property.

Methods

This object has no unique methods.

Properties

Property	Data type	Value

`breakAtCorners`	Boolean	Same as the Sharp Corners setting in the custom Stroke Style dialog box. Valid values are `true` or `false`. The following example sets the `breakAtCorners` property to `true`:
		`var myStroke = fl.getDocumentDOM().getCustomStroke();`
		`myStroke.breakAtCorners = true;`
		`fl.getDocumentDOM().setCustomStroke( myStroke );`

`color`	string	A color string in hexadecimal (#rrggbb) format or an integer containing the value. Represents the stroke color.
		The following example sets the stroke color:
		`var myStroke = fl.getDocumentDOM().getCustomStroke();`
		`myStroke.color = "#000000";`
		`fl.getDocumentDOM().setCustomStroke( myStroke );`

`curve`	string	Can be set only if `style` property is `hatched`. Acceptable values are `"straight"`, `"slight curve"`, `"medium curve"`, and `"very curved"`.
		The following example sets the `curve` property, as well as others, for a stroke having the `hatched` style:
		`var myStroke = fl.getDocumentDOM().getCustomStroke();`
		`myStroke.style = "hatched";`
		`myStroke.curve = "straight";`
		`myStroke.space = "close";`
		`myStroke.jiggle = "wild";`
		`myStroke.rotate = "free";`
		`myStroke.length = "slight";`
		`myStroke.hatchThickness = "thin";`
		`fl.getDocumentDOM().setCustomStroke( myStroke );`

Property	Data type	Value

dash1	int	Only available if the style property is set to dashed. This represents the lengths of the solid part of the line.
		The following example sets the dash1 and dash2 properties for a stroke style of dashed:
		`var myStroke = fl.getDocumentDOM().getCustomStroke();`
		`myStroke.style = "dashed";`
		`myStroke.dash1 = 1;`
		`myStroke.dash2 = 2;`
		`fl.getDocumentDOM().setCustomStroke( myStroke );`

dash2	int	Can be set only if style property is set to dashed. This represents the lengths of the blank part of the line.
		The following example sets the dash1 and dash2 properties for a stroke style of dashed:
		`var myStroke = fl.getDocumentDOM().getCustomStroke();`
		`myStroke.style = "dashed";`
		`myStroke.dash1 = 1;`
		`myStroke.dash2 = 2;`
		`fl.getDocumentDOM().setCustomStroke( myStroke );`

density	string	Can be set only if style property is stipple. Acceptable values are "very dense", "dense", "sparse", and "very sparse".
		The following example sets the density property to "sparse" for the stroke style of stipple:
		`var myStroke = fl.getDocumentDOM().getCustomStroke();`
		`myStroke.style = "stipple";`
		`myStroke.dotspace= 3;`
		`myStroke.variation = "random sizes";`
		`myStroke.density = "sparse";`
		`fl.getDocumentDOM().setCustomStroke( myStroke );`

dotSize	string	Can be set only when style property is set to stipple. Acceptable values are "tiny", "small", "medium", and "large".
		The following example sets the dotsize property to "tiny" for the stroke style of stipple:
		`var myStroke = fl.getDocumentDOM().getCustomStroke();`
		`myStroke.style = "stipple";`
		`myStroke.dotspace= 3;`
		`myStroke.dotsize = "tiny";`
		`myStroke.variation = "random sizes";`
		`myStroke.density = "sparse";`
		`fl.getDocumentDOM().setCustomStroke( myStroke );`

dotspace	int	Can be set only if style property is set to dotted. This value sets the spacing between the dots.
		The following example sets the dotspace property to 3 for a stroke style of dotted:
		`var myStroke = fl.getDocumentDOM().getCustomStroke();`
		`myStroke.style = "dotted";`
		`myStroke.dotspace= 3;`
		`fl.getDocumentDOM().setCustomStroke( myStroke );`

Property	Data type	Value

hatchThickness	string	Can be set only if style property is hatched. Acceptable values are "hairline", "thin", "medium", and "thick".
		The following example sets the hatchThickness property to "thin" for a stroke style of hatched:
		`var myStroke = fl.getDocumentDOM().getCustomStroke();`
		`myStroke.style = "hatched";`
		`myStroke.curve = "straight";`
		`myStroke.space = "close";`
		`myStroke.jiggle = "wild";`
		`myStroke.rotate = "free";`
		`myStroke.length = "slight";`
		`myStroke.hatchThickness = "thin";`
		`fl.getDocumentDOM().setCustomStroke( myStroke );`

jiggle	string	Can be set only if style property is hatched. Acceptable values are "none", "bounce", "loose", and "wild".
		The following example sets the jiggle property to "wild" for a stroke style of hatched:
		`var myStroke = fl.getDocumentDOM().getCustomStroke();`
		`myStroke.style = "hatched";`
		`myStroke.curve = "straight";`
		`myStroke.space = "close";`
		`myStroke.jiggle = "wild";`
		`myStroke.rotate = "free";`
		`myStroke.length = "slight";`
		`myStroke.hatchThickness = "thin";`
		`fl.getDocumentDOM().setCustomStroke( myStroke );`

length	string	Can be set only if style property is hatched. Acceptable values are "equal", "slight", "variation", "medium variation", and "random".
		The following example sets the length property to "slight" for a stroke style of hatched:
		`var myStroke = fl.getDocumentDOM().getCustomStroke();`
		`myStroke.style = "hatched";`
		`myStroke.curve = "straight";`
		`myStroke.space = "close";`
		`myStroke.jiggle = "wild";`
		`myStroke.rotate = "free";`
		`myStroke.length = "slight";`
		`myStroke.hatchThickness = "thin";`
		`fl.getDocumentDOM().setCustomStroke( myStroke );`

pattern	string	Can be set only if the style property is "ragged". Acceptable values are "solid", "simple", "random", "dotted", "random dotted", "triple dotted", and "random triple dotted".
		The following example sets the pattern property to "random" for a stroke style of ragged:
		`var myStroke = fl.getDocumentDOM().getCustomStroke();`
		`myStroke.style = "ragged";`
		`myStroke.pattern = "random";`
		`fl.getDocumentDOM().setCustomStroke( myStroke );`

Property	Data type	Value

rotate	string	Can be set only if style property is hatched. Acceptable values are "none", "slight", "medium", and "free".
		The following example sets the rotate property to "free" for a style stroke of hatched:
		`var myStroke = fl.getDocumentDOM().getCustomStroke();`
		`myStroke.style = "hatched";`
		`myStroke.curve = "straight";`
		`myStroke.space = "close";`
		`myStroke.jiggle = "wild";`
		`myStroke.rotate = "free";`
		`myStroke.length = "slight";`
		`myStroke.hatchThickness = "thin";`
		`fl.getDocumentDOM().setCustomStroke( myStroke );`

space	string	Can be set only if style property is hatched. Acceptable values are "very close", "close", "distant", and "very distant".
		The following example sets the space property to "close" for a stroke style of hatched:
		`var myStroke = fl.getDocumentDOM().getCustomStroke();`
		`myStroke.style = "hatched";`
		`myStroke.curve = "straight";`
		`myStroke.space = "close";`
		`myStroke.jiggle = "wild";`
		`myStroke.rotate = "free";`
		`myStroke.length = "slight";`
		`myStroke.hatchThickness = "thin";`
		`fl.getDocumentDOM().setCustomStroke( myStroke );`

style	string	Describes the stroke style. Acceptable values are "noStroke", "solid", "dashed", "dotted", "ragged", "stipple", and "hatched".
		Some of these values require additional properties of the stroke object to be set, as follows:
		• If value is "solid" or "noStroke" there are no other properties.
		• If value is "dashed" there are two additional properties: "dash1" and "dash2".
		• If value is "dotted" there is one additional property: "dotSpace".
		• If value is "ragged" there are three additional properties: "pattern", "waveHeight", and "waveLength" .
		• If value is "stipple" there are three additional properties: "dotSize", "variation", and "density".
		• If value is "hatched" there are six additional properties: "hatchThickness", "space", "jiggle", "rotate", "curve", and "length".
		The following example sets the stroke style to "ragged":
		`var myStroke = fl.getDocumentDOM().getCustomStroke();`
		`myStroke.style = "ragged";`
		`fl.getDocumentDOM().setCustomStroke( myStroke );`

thickness	float	Represents the stroke size.
		The following example sets the thickness property of the stroke to a value of 2:
		`var myStroke = fl.getDocumentDOM().getCustomStroke();`
		`myStroke.thickness = 2;`
		`fl.getDocumentDOM().setCustomStroke( myStroke );`

Property	Data type	Value

variation	string	Can be set only if style property is stipple. Acceptable values are: "one size", "small variation", "varied sizes", and "random sizes".
		The following example sets the variation property to "random sizes" for a stroke style of stipple.
		`var myStroke = fl.getDocumentDOM().getCustomStroke();`
		`myStroke.style = "stipple";`
		`myStroke.dotspace= 3;`
		`myStroke.variation = "random sizes";`
		`myStroke.density = "sparse";`
		`fl.getDocumentDOM().setCustomStroke( myStroke );`

waveHeight	string	Can be set only if style property is ragged. Acceptable values are "flat", "wavy", "very wavy", and "wild".
		The following example sets the waveHeight property to "flat" for a stroke style of ragged:
		`var myStroke = fl.getDocumentDOM().getCustomStroke();`
		`myStroke.style = "ragged";`
		`myStroke.pattern = "random";`
		`myStroke.waveHeight = "flat";`
		`myStroke.waveLength = "short";`
		`fl.getDocumentDOM().setCustomStroke( myStroke );`

waveLength	string	Can be set only if the style property value is ragged. Acceptable values are "very short", "short", "medium", and "long".
		The following example sets the waveLength property to "short" for a stroke style of ragged.
		`var myStroke = fl.getDocumentDOM().getCustomStroke();`
		`myStroke.style = "ragged";`
		`myStroke.pattern = "random";`
		`myStroke.waveHeight = 'flat";`
		`myStroke.waveLength = "short";`
		`fl.getDocumentDOM().setCustomStroke( myStroke );`

SymbolInstance

Description

SymbolInstance is a subclass of the Instance object and represents a symbol in a frame.

Methods

The SymbolInstance object has no unique methods; see the Instance object.

Properties

In addition to the Instance object properties, the SymbolInstance object has the following properties.

Property	Data type	Value

`accName`	accName	Equivalent to the Name field in the Accessibility panel. Screen readers identify objects by reading the name aloud. This property is not available for graphic symbols.
		The following example stores the value for the Accessibility panel name of the object in the *theName* variable:
		`var theName = fl.getDocumentDOM().selection[0].accName;`
		The following example sets the value for the Accessibility panel name of the object to "`Home Button`":
		`fl.getDocumentDOM().selection[0].accName = "Home Button";`

`actionScript`	string	This applies only to movie clip and button instances. For a graphic symbol instance, the value returns undefined. The ActionScript property value is a string representing the actions assigned to the symbol. The following example assigns an `onClipEvent` action to the first item in the first frame of the first layer of the Timeline: `fl.getDocumentDOM().getTimeline().layers[0].frames[0].elements[0].actionScript = "onClipEvent(enterFrame) {trace('movie clip enterFrame');}";`
`buttonTracking`	string	This applies only to button symbols and sets the same property as the pop-up menu for Track as Button or Track as Menu Item in the Property inspector. For other types of symbols, this property will be ignored. Acceptable values are `"button"` or `"menu"`. The following example sets the first symbol in the first frame of the first layer in the Timeline to Track as Menu Item, as long as that symbol is a button: `fl.getDocumentDOM().getTimeline().layers[0].frames[0].elements[0].buttonTracking = "menu";`

Property	Data type	Value
`colorAlphaAmount`	int	Part of the color transformation for the instance, specifying the Advanced Effect Alpha settings; equivalent to using the Color › Advanced setting in the Property inspector and adjusting the controls on the right of the dialog box. This value either reduces or increases the tint and alpha values by a constant amount. This value is added to the current value. This property is most useful if used with `colorAlphaPercent()`. Allowable values are between -255 to 255. The following example changes the `colorAlphaAmount` of the selected symbol instance: `fl.getDocumentDOM().selection[0].colorAlphaAmount = -100;`
`colorAlphaPercent`	int	Part of the color transformation for the instance, equivalent to using the Color › Advanced setting in the Instance Property inspector (the percentage controls on the left of the dialog box). This value reduces the tint and alpha values by a specified percentage. The current values are multiplied by this percentage. Allowable values are from -100 to 100. The following example sets the `colorAlphaPercent` of the selected symbol instance to 90: `fl.getDocumentDOM().selection[0].colorAlphaPercent = 90;`
`colorBlueAmount`	int	Part of the color transformation for the instance, equivalent to using the Color › Advanced setting in the Instance Property inspector. Allowable values are from -255 to 255. The following example sets the `colorBlueAmount` of the selected symbol instance to 255: `fl.getDocumentDOM().selection[0].colorBlueAmount = 255;`
`colorBluePercent`	int	Part of the color transformation for the instance, equivalent to using the Color › Advanced setting in the Instance Property inspector (the percentage controls on the left of the dialog box). This value reduces the blue values by a specified percentage. The current values are multiplied by this percentage. Allowable values are from -100 to 100. The following example sets the `colorBluePercent` of the selected symbol instance to 80: `fl.getDocumentDOM().selection[0].colorBluePercent = 80;`
`colorGreenAmount`	int	Part of the color transformation for the instance, equivalent to using the Color › Advanced setting in the Instance Property inspector. Allowable values are from -255 to 255. The following example sets the `colorGreenAmount` of the selected symbol instance to 255: `fl.getDocumentDOM().selection[0].colorGreenAmount = 255;`

Property	Data type	Value
colorGreenPercent	int	Part of the color transformation for the instance, equivalent to using the Color › Advanced setting in the Instance Property inspector (the percentage controls on the left of the dialog box). This value reduces the green values by a specified percentage. The current values are multiplied by this percentage. Allowable values are from -100 to 100. The following example sets the colorGreenPercent of the selected symbol instance to 70: `fl.getDocumentDOM().selection[0].colorGreenPercent = 70;`
colorMode	string	The color mode as identified in the symbol Property inspector Color pop-up menu. Acceptable values are "none", "brightness", "tint", "alpha", and "advanced". The following example changes the colorMode property of the first element in the first frame of the first layer of the Timeline to "alpha": `fl.getDocumentDOM().getTimeline().layers[0].frames[0].elements[0].colorMode = "alpha";`
colorRedAmount	int	Part of the color transformation for the instance, equivalent to using the Color › Advanced setting in the Instance Property inspector. Allowable values are from -255 to 255. The following example sets the colorRedAmount of the selected symbol instance to 255: `fl.getDocumentDOM().selection[0].colorRedAmount = 255;`
colorRedPercent	int	Part of the color transformation for the instance, equivalent to using the Color › Advanced setting in the Instance Property inspector (the percentage controls on the left of the dialog box). This value reduces the red values by a specified percentage. The current values are multiplied by this percentage. Allowable values are from -100 to 100. The following example sets the colorRedPercent of the selected symbol instance to 10: `fl.getDocumentDOM().selection[0].colorRedPercent = 10;`

description	string	Equivalent to the Description field on the Accessibility panel. The description is read by the screen reader. This property is not available for graphic symbols.
		The following example stores the value for the Accessibility panel description of the object in the *theDescription* variable:
		`var theDescription = fl.getDocumentDOM().selection[0].description;`
		The following example sets the value for the Accessibility panel description to "Click the home button to go to home":
		`fl.getDocumentDOM().selection[0].description= "Click the home button to go to home";`

firstFrame	int	This applies only to graphic symbols and sets the same property as the First field in the Property inspector. For other types of symbols, this property will be undefined. This property specifies the first frame to appear in the Timeline of the graphic. `fl.getDocumentDOM().getTimeline().layers[0].frames[0].elements[0].firstFrame = 10;`

Property	Data type	Value

forceSimple	Boolean	Enables and disables the children of the object to be accessible. This is equivalent to the inverse logic of the Make Child Objects Accessible setting in the Accessibility panel.
		If forceSimple is true, it is the same as the Make Child Object Accessible option being unchecked.
		If forceSimple is false, it is the same as the Make Child Object Accessible option being checked.
		This property is not available for either graphic or button objects.
		The following example checks to see if the children of the object are accessible:
		`var areChildrenAccessible = !fl.getDocumentDOM().selection[0].forceSimple;`
		The following example allows the children of the object to be accessible:
		`fl.getDocumentDOM().selection[0].forceSimple = false;`

loop	string	This applies only to graphic symbols and sets the same property as the Loop pop-up menu in the Property inspector. For other types of symbols, this property will be undefined. Acceptable values are "loop", "play once", and "single frame" to set the graphic's animation accordingly. The following example sets the first symbol in the first frame of the first layer in the Timeline to Single Frame (display one specified frame of the graphic Timeline), as long as that symbol is a graphic: `fl.getDocumentDOM().getTimeline().layers[0].frames[0].elements[0].loop = 'single frame';`

shortcut	string	Equivalent to the Shortcut field on the Accessibility panel. The shortcut is read by the screen readers. This property is not available for graphic symbols.
		The following example stores the value for the shortcut key of the object in the *theShortcut* variable:
		`var theShortcut = fl.getDocumentDOM().selection[0].shortcut; :`
		The following example sets the shortcut key of the object to "Ctrl+i":
		`fl.getDocumentDOM().selection[0].shortcut = "Ctrl+i";`

silent	Boolean	Enables/disables the object to be accessible, equivalent to the inverse logic of the Make Object Accessible setting in the Accessibility panel.
		silent == true is the equivalent of "Make Object Accessible" being unchecked.
		The expression silent == false is the equivalent of "Make Object Accessible" being checked.
		This property is not available for graphic objects.
		The following example checks to see if the object is accessible:
		`var isSilent = fl.getDocumentDOM().selection[0].silent;`
		The following example sets the object to be accessible:
		`fl.getDocumentDOM().selection[0].silent = false;`

Property	Data type	Value
symbolType	string	The type of symbol, equivalent to the value for Behavior in the Create New Symbol and Convert To Symbol dialog boxes. Acceptable values are "button", "movie clip", and "graphic". The following example sets the first symbol in the first frame of the first layer in the Timeline of the current document to behave as a "graphic": `fl.getDocumentDOM().getTimeline().layers[0].frames[0].elements[0].symbolType = "graphic";`

tabIndex	int	This property is equivalent to the Tab index field on the Accessibility panel (available only in Flash MX Professional 2004). Creates a tab order in which objects are accessed when the user presses the Tab key. This property is not available for graphic symbols.
		The following example sets the tabIndex property of the mySymbol object to 3 and displays that value in the Output panel:
		`mySymbol = fl.getDocumentDOM().selection[0];`
		`mySymbol.tabIndex = 3;`
		`fl.trace(mySymbol.tabIndex);`

SymbolItem

Description

The SymbolItem object is a subclass of the Item object.

Methods

In addition to the Item object methods, you can use the following methods with the SymbolItem object:

```
symbolItem.convertToCompiledClip()
symbolItem.exportSWC()
symbolItem.exportSWF()
```

symbolItem.convertToCompiledClip()

Description

Converts a symbol item in the Library to a compiled movie clip (SWF symbol).

Usage

```
symbolItem.convertToCompiledClip()
```

Arguments

None.

Returns

Nothing.

Example

The following example converts an item in the Library to a compiled movie clip.

```
fl.getDocumentDOM().library.items[0].convertToCompiledClip();
```

symbolItem.exportSWC()

Description

Exports the symbol to a SWC file. The output SWC file is specified using a URI.

Usage

```
symbolItem.exportSWC( outputURI )
```

Arguments

The *outputURI* argument is a String that specifies the URI for the SWC file to which the method will export the symbol. The URI should reference a local file. Flash will not create a folder if the named folder path does not exist.

Returns

Nothing.

Example

The following example exports an item in the Library to the SWC file `my.swc` in the `tests` folder.

```
fl.getDocumentDOM().library.items[0].exportSWC("file:///c|/tests/my.swc");
```

symbolItem.exportSWF()

Description

Exports the symbol item to a SWF file specified by a URI

Usage

```
symbolItem.exportSWF( outputURI )
```

Arguments

The *`outputURI`* argument is a String that specifies the URI for the SWF file to which the method will export the symbol. This URI must reference a local file. Flash will not create a folder if the named folder path doesn't exist.

Returns

Nothing.

Example

The following example exports an item in the Library to the SWF file `my.swf` in the `tests` folder.

```
fl.getDocumentDOM().library.items[0].exportSWF("file:///c|/tests/my.swf");
```

Properties

In addition to the Item object properties, the following properties are available for the SymbolItem object:

Property	Data type	Value
`sourceAutoUpdate`	Boolean	If `true`, the item will be updated when published. The default value is `false`. Used for Shared Library symbols. The following example sets the `sourceAutoUpdate` property: `fl.getDocumentDOM().library.items[0].sourceAutoUpdate = true;`
`sourceFilePath`	string	Path for source FLA file. Must be an absolute path, not a relative path. Used for Shared Library symbols. The following example displays the value of the `sourceFilePath` property in the Output panel: `fl.trace(fl.getDocumentDOM().library.items[0].sourceFilePath);`
`sourceLibraryName`	string	Name of the item in the source file to use. Used for Shared Library symbols. The following example displays the value of the `sourceLibraryName` property in the Output panel: `fl.trace(fl.getDocumentDOM().library.items[0].sourceLibraryName);`

Property	Data type	Value

`symbolType`	string	Acceptable values are `"movie clip"`, `"button"`, or `"graphic"`.
		The following example displays the current value of the `symbolType` property, changes it to `"button"`, and displays it again:
		`alert(fl.getDocumentDOM().library.items[0].symbolType);`
		`fl.getDocumentDOM().library.items[0].symbolType = "button";`
		`alert(fl.getDocumentDOM().library.items[0].symbolType);`

`timeline`
read only
	object	The `timeline` object. See "Timeline" on page 197.
		The following example obtains and displays the number of layers that the selected movie clip in the library contains:
		`var tl = fl.getDocumentDOM().library.getSelectedItems()[0].timeline;`
		`alert(tl.layerCount);`

Text

Description

The Text object is a subclass of the Element object. The Text object represents a single text item in the document. All properties of the text pertain to the entire text block.

To set properties of a run within the text field, see "TextRun" on page 196. To change properties on a selection within a text field, you can use `document.setElementTextAttr()` and specify a range of text, or use the current selection.

To set text properties of the selected text field, use `document.setElementProperty()`. The following example assigns the currently selected text field to the variable *`textVar`*:

```
fl.getDocumentDOM().setElementProperty("variableName", "textVar");
```

Methods

In addition to the Element object methods, you can use the following methods with the Text object:

```
text.getTextAttr()
text.getTextString()
text.setTextAttr()
text.setTextString()
```

text.getTextAttr()

Description

Gets the attribute specified by the *`attrName`* argument for the text identified by the optional arguments *`startIndex`* and *`endIndex`*. If the attribute is not consistent for the specified range, Flash will return `"undefined"`. If the optional arguments for *`startIndex`* and *`endIndex`* are not specified, the method uses the entire text range. If only *`startIndex`* is specified, the range used is a single character at that position. If *`startIndex`* and *`endIndex`* are specified, the range starts from *`startIndex`* and goes up to, but not including, *`endIndex`*.

See "TextAttrs" on page 193 for a list of names to use for *`attrName`*, and for the type of values that will be returned.

Usage

```
text.getTextAttr( attrName [, startIndex [, endIndex] ] )
```

Arguments

The *`attrName`* argument is a String that specifies the name of the `TextAttrs` property to be returned.

The optional *`startIndex`* argument is an integer that is the index of first character.

The optional *`endIndex`* argument is an integer that specifies the end of the range of text, which starts with *`startIndex`* and goes up to, but not including, *`endIndex`*.

Returns

The value of the attribute specified in the argument *`attrName`*.

Example

The following example gets the font size of the currently selected text field and displays it:

```
var TheTextSize = fl.getDocumentDOM().selection[0].getTextAttr("size");
fl.trace(TheTextSize);
```

The following example gets the text fill color of the selected text field:

```
var TheFill = fl.getDocumentDOM().selection[0].getTextAttr("fillColor");
fl.trace(TheFill);
```

The following example gets the size of the character at index 2:

```
var Char2 = fl.getDocumentDOM().selection[0].getTextAttr("size", 2);
fl.trace(Char2);
```

The following example gets the color of the selected text field from character at index 2 up to but not including character at index 8:

```
fl.getDocumentDOM().selection[0].getTextAttr("fillColor", 2, 8);
```

text.getTextString()

Description

Gets the specified range of text. If the optional arguments *startIndex* and *endIndex* are not specified, the whole text string will be returned. If only *startIndex* is specified, the string starting at the index location and ending at the end of the field will be returned. If *startIndex* and *endIndex* are specified, the string starting from startIndex up to, but not including, *endIndex* will be returned.

Usage

```
text.getTextString( [, startIndex [, endIndex] ])
```

Arguments

The optional *startIndex* argument is an integer that specifies the index of the first character (zero-based).

The optional *endIndex* argument is an integer that specifies the range of text, starting from *startIndex* up to, but not including, *endIndex*.

Returns

A string of the text in the specified range.

Example

The following example gets the character(s) starting at index location 4 and through the end of the selected text field:

```
var myText = fl.getDocumentDOM().selection[0].getTextString(4);
fl.trace(myText);
```

The following example gets the characters starting at index location 3 and up to, but not including, the character at index location 9 in the selected text field:

```
var myText = fl.getDocumentDOM().selection[0].getTextString(3, 9);
fl.trace(myText);
```

text.setTextAttr()

Description

The `text.setTextAttr()` method sets the attribute specified by the *`attrName`* argument associated with the text identified by *`startIndex`* and *`endIndex`*. This method can be used to change attributes of text that might span `textRun` elements, or are portions of existing `textRun` elements. Using it may change the position and number of `textRun` elements within this object's `textRun` array. If the optional arguments are not specified, the method uses the entire text object's character range. If only *`startIndex`* is specified, the range is a single character at that position. If *`startIndex`* and *`endIndex`* are specified, the range starts from *`startIndex`* and goes up to, but not including, the character located at *`endIndex`*.

See "TextAttrs" on page 193 for a list of possible entries for *`attrName`*, and for the type of values that will be returned.

Usage

```
text.setTextAttr( attrName, attrValue [, startIndex [, endIndex] ] )
```

Arguments

The *`attrName`* argument is a String that specifies the name of the `TextAttrs` property to change.

The *`attrValue`* argument is the value for the `TextAttrs` property. See "TextAttrs" on page 193 for the list of property name and values expected.

The optional *`startIndex`* argument is an integer that is the index (zero-based) of the first character in the array.

The optional *`endIndex`* argument is an integer that is a range of text, starting from *`startIndex`* up to, but not including, *`endIndex`*.

Returns

Nothing.

Example

The following example sets the selected text field to italic:

```
fl.getDocumentDOM().selection[0].setTextAttr("italic", true);
```

The following example sets the size of the character at index 2 to 10:

```
fl.getDocumentDOM().selection[0].setTextAttr("size", 10, 2);
```

The following example sets the color to red for the third character up to, but not including, the ninth character of the selected text:

```
fl.getDocumentDOM().selection[0].setTextAttr("fillColor", 0xff0000, 2, 8);
```

text.setTextString()

Description

Changes the text string within this text object. If the optional arguments are not specified, the whole text object will be replaced. If only *`startIndex`* is specified, the string specified will be inserted at the *`startIndex`* position. If *`startIndex`* and *`endIndex`* are specified, the specified string will replace the segment of text starting from *`startIndex`* up to, but not including, *`endIndex`*.

Usage

```
text.setTextString( text [, startIndex [, endIndex] ] )
```

Arguments

The *text* argument is a String consisting of the characters to be inserted into this text object.

The optional *startIndex* argument is an integer that specifies the index (zero-based) of the character in the string where the text will be inserted.

The optional *endIndex* argument is an integer that specifies the index of the end point in the selected text string. The new text will overwrite the text from *startIndex* up to, but not including, *endIndex*.

Returns

Nothing.

Example

The following example assigns the string "this is a string" to the selected text field:

```
fl.getDocumentDOM().selection[0].setTextString("this is a string");
```

The following example inserts the string "abc" at index 2 of the selected text field:

```
fl.getDocumentDOM().selection[0].setTextString("abc", 2);
```

The following example replaces the text at index 2 up to, but not including, the character at index 8 of the selected text string with the string "abcdefghij". Characters between *startIndex* and *endIndex* are overwritten. Characters beginning with *endIndex* are pushed down.

```
fl.getDocumentDOM().selection[0].setTextString("abcdefghij", 2, 8);
```

Properties

In addition to the Element object properties, the following properties are available for the Text object:

Property	Data type	Value

accName	string	Equivalent to the Name field in the Accessibility panel. Screen readers identify objects by reading the name aloud. This property is not available for the Dynamic Text type.
		The following example gets the name of the object:
		`var theName = fl.getDocumentDOM().getTimeline().layers[0].frames[0].elements[0].accName;`
		The following example sets the name of the currently selected object:
		`fl.getDocumentDOM().selection[0].accName = "Home Button";`

autoExpand	Boolean	If true, for static text fields, the bounding width will expand to display all text. If true, for Dynamic or Input Text fields, the bounding width and height will expand to display all text. The following example sets the autoExpand property to a value of true: `fl.getDocumentDOM().selection[0].autoExpand = true;`

Property	Data type	Value
`border`	Boolean	If `true`, Flash shows a border around Dynamic or Input Text; generates a warning dialog box if used with Static Text. The following example sets the `border` property to a value of `true`: `fl.getDocumentDOM().getTimeline().layers[0].frames[0].elements[0].border = true;`

`description`	string	Equivalent to the Description field in the Accessibility panel. The description is read by the screen reader.
		The following example gets the description of the object:
		`var theDescription = fl.getDocumentDOM().getTimeline().layers[0].frames[0].elements[0].description;`
		The following example sets the description of the object:
		`fl.getDocumentDOM().getTimeline().layers[0].frames[0].elements[0].description= "Enter your name here";`

`embeddedCharacters`	string	Allows the user to specify characters to embed; the equivalent to entering text into the Character Options dialog box. This works only with Dynamic or Input Text; generates a warning dialog box if used with other text types. The following example sets the embeddedCharacters property to a value of `"abc"`: `fl.getDocumentDOM().selection[0].embeddedCharacters = "abc";`

`embedRanges`	string	This property can be modified or read. It consists of a series of delimited integers that correspond to the items that can be selected in the Character Options dialog box. This works only with Dynamic or Input Text; ignored if used with Static Text.
		Note that these values also correspond to the XML file in the Configuration/FontEmbedding folder.
		The following example sets the embedRanges property to `"1|3|7"`:
		`fl.getDocumentDOM().getTimeline().layers[0].frames[0].elements[0].embedRanges = "1|3|7";`
		The following example resets the property:
		`fl.getDocumentDOM().getTimeline().layers[0].frames[0].elements[0].embedRanges = "";`

`length` **read only**	int	The number of characters in the text object. The following example returns the number of characters in the selected text: `var textLength = fl.getDocumentDOM().selection[0].length;`
`lineType`	string	Sets the line type value to `"single line"`, `"multiline"`, `"multiline no wrap"`, or `"password"`. This works only with Dynamic or Input Text and generates a warning dialog box if used with Static Text. The `"password"` value only works for input text. The following example sets the `lineType` property to the value `"multiline no wrap"`: `fl.getDocumentDOM().selection[0].lineType = "multiline no wrap";`

Property	Data type	Value
maxCharacters	int	Specifies the maximum characters the user can enter into this text object. This works only with Input Text; generates a warning dialog box if used with other text types. The following example sets the value of the maxCharacters property to 30: `fl.getDocumentDOM().selection[0].maxCharacters = 30;`
orientation	string	Acceptable values are "horizontal", "vertical left to right", and "vertical right to left". This works only with Static Text; generates a warning dialog box if used with other text types. The following example sets the orientation property to "vertical right to left": `fl.getDocumentDOM().getTimeline().layers[0].frames[0].elements[0].orientation = "vertical right to left";`
renderAsHTML	Boolean	If true, Flash draws the text as HTML and interprets embedded HTML tags. This works only with Dynamic or Input Text; generates a warning dialog box if used with other text types. The following example sets the renderAsHTML property to true: `fl.getDocumentDOM().selection[0].renderAsHTML = true;`
scrollable	Boolean	If true, the text can be scrolled. This works only with Dynamic or Input Text; generates a warning dialog box if used with Static Text. The following example sets the scrollable property to false: `fl.getDocumentDOM().selection[0].scrollable = false;`
selectable	Boolean	If true, the text can be selected. Input Text is always selectable. Generates a warning dialog if set to false when using input text. The following example sets the selectable property to true: `fl.getDocumentDOM().getTimeline().layers[0].frames[0].elements[0].selectable = true;`
selectionEnd	int	Offset of the end of a text subselection. If there is an insertion point or no selection, selectionEnd is equal to selectionStart. If selectionEnd is set to a value less than selectionStart, selectionStart will be set to selectionEnd. The following example sets the selectionEnd property to 30: `fl.getDocumentDOM().getTimeline().layers[0].frames[0].elements[0].selectionEnd = 30;`
selectionStart	int	Offset of the beginning of a text subselection. 0 is the beginning of the text block. If selectionStart equals selectionEnd, there is no subselection. If selectionStart is set to a value greater than selectionEnd, selectionEnd will be set to selectionStart. The following example sets the start of the text subselection to the sixth character: `fl.getDocumentDOM().selection[0].selectionStart = 5;`

Property	Data type	Value

`shortcut`	string	This is equivalent to the Shortcut field on the Accessibility panel. The shortcut is read by the screen readers. This property is not available to the Dynamic Text type.
		The following example gets the `shortcut` key of the selected object and displays the value:
		`var theShortcut = fl.getDocumentDOM().selection[0].shortcut;`
		`fl.trace(theShortcut);`
		The following example sets the `shortcut` key of the selected object:
		`fl.getDocumentDOM().selection[0].shortcut = "Ctrl+i";`

`silent`	Boolean	Enables /disables the object to be accessible. This is equivalent to the inverse logic of the Make Object Accessible setting in the Accessibility panel.
		If the `silent property is true`, then Make Object Accessible is unchecked.
		If the `silent property is false`, then Make Object Accessible is checked.
		The following example queries if the object is accessible:
		`var isSilent = fl.getDocumentDOM().getTimeline().layers[0].frames[0].elements[0].silent;`
		The following example sets the object to be accessible:
		`fl.getDocumentDOM().getTimeline().layers[0].frames[0].elements[0].silent = false;`

`tabIndex`	int	Equivalent to the Tab Index field on the Accessibility panel.
		Creates a tab order in which objects are accessed when the user presses the Tab key. The UI for this is available only in Flash MX Professional 2004.
		The following example gets the `tabIndex` of the currently selected object:
		`var theTabIndex = fl.getDocumentDOM().selection[0].tabIndex;`
		The following example sets the `tabIndex` of the currently selected object:
		`fl.getDocumentDOM().selection[0].tabIndex = 1;`

`textRuns` **read only**	array	The array of `TextRun` objects. The following example retrieves the value of the `textRuns` property into the `textRuns` variable. `var myTextRuns = fl.getDocumentDOM().selection[0].textRuns;`
`textType`	string	Acceptable values are `"static"`, `"dynamic"`, or `"input"`. The following example sets the `textType` property to `"input"`. `fl.getDocumentDOM().selection[0].textType = "input";`

Property	Data type	Value
`useDeviceFonts`	Boolean	If `true`, Flash will draw text using device fonts. This works only with static text; generates a warning dialog box if used with other text types. The following example causes Flash to use device fonts with static text. `fl.getDocumentDOM().selection[0].useDeviceFonts = true;`
`variableName`	string	Contents of the text object will be stored in `variableName`. This works only with Dynamic or Input Text; generates a warning dialog box if used with other text types. The following example stores the value `"varName"` in the `variableName` property: `fl.getDocumentDOM().getTimeline().layers[0].frames[0].elements[0].variableName = "varName";`

TextAttrs

Description

The TextAttrs object contains all the properties of text that can be applied to a subselection (see "Text" on page 185 for relevant methods).

Methods

The TextAttrs object has no unique methods.

Properties

Property	Data type	Value
`aliasText`	Boolean	Specifies that the text should be drawn using a method optimized for increasing the legibility of small text. The following example sets the aliasText property to true for all the text in the currently selected text field. `fl.getDocumentDOM().setElementTextAttr('aliasText', true);`

`alignment`	string	Specifies paragraph justification: Acceptable values are `"left"`, `"center"`, `"right"`, and `"justify"`.
		The following example selects the characters from index 0 up to but not including index 3 and sets the alignment property to `"justify"`;
		`fl.getDocumentDOM().setTextSelection(0, 3);`
		`fl.getDocumentDOM().setElementTextAttr('alignment', 'justify');`

`autoKern`	Boolean	If `true`, use pair kerning information in the font(s) to kern the text. If `false`, pair kerning information in the font(s) is ignored.
		This applies only to Static Text; a warning dialog appears if used with other text types.
		The following example selects the characters from index 2 up to but not including index 6 and sets the bold property to `true` .
		`fl.getDocumentDOM().setTextSelection(3, 6);`
		`fl.getDocumentDOM().setElementTextAttr('bold', true);`

`bold`	Boolean	If `true`, text appears with bold version of font.
		The following example selects the only the first characteand sets the bold property to `true`.
		`fl.getDocumentDOM().setTextSelection(0, 1);`
		`fl.getDocumentDOM().setElementTextAttr('bold', true);`

`characterPosition`	string	Determines the baseline for the text. Acceptable values are `"normal"`, `"subscript"`, or `"superscript"`. This applies only to Static Text.
		The following example selects tthe characters from index 2 up to but not inlcuding index 6 and sets the `characterPosition` property to `true`.
		`fl.getDocumentDOM().setTextSelection(0, 1);`
		`fl.getDocumentDOM().setElementTextAttr('characterPosition ", "subScript");`

Property	Data type	Value
characterSpacing	int	Integer representing the space between characters. Acceptable values are -60 to 60. This applies only to Static Text; a warning dialog box appears if used with other text types.
face	string	The name of the font, such as "Arial". The following example sets the color of the selected text field from character at index 2 up to but not including character at index 8 to "Arial". `fl.getDocumentDOM().selection[0].setTextAttr("face", "Arial", 2, 8);`
fillColor	string	Specifies the fill color. The argument is a color string in hexadecimal (#rrggbb) format or an integer containing the value. The following example sets the color of the selected text field from the character at index 2 up to but not including character at index 8 to red. `fl.getDocumentDOM().selection[0].setTextAttr("fillColor", 0xff0000, 2, 8);`
indent	int	Paragraph indentation. Acceptable values are from -720 to 720. The following example sets the indent of the selected text field from character at index 2 up to but not including character at index 8 to 100. `fl.getDocumentDOM().selection[0].setTextAttr("indent", 100, 2, 8);`
italic	Boolean	If true, text appears with italic version of font. The following example sets the selected text field to italic. `fl.getDocumentDOM().selection[0].setTextAttr("italic", true);`
leftMargin	int	Paragraph's left margin. Acceptable values are from 0 to 720 The following example sets the leftMargin property of the selected text field from the character at index 2 up to but not including the character at index 8 to 100. `fl.getDocumentDOM().selection[0].setTextAttr("leftMargin", 100, 2, 8);`
lineSpacing	int	Line spacing of paragraph, also known as *leading*. Acceptable values are -360 to 720 The following example sets the selected text field lineSpacing property to 100. `fl.getDocumentDOM().selection[0].setTextAttr("lineSpacing", 100);`
rightMargin	int	Paragraph's right margin. Acceptable values are from 0 to 720 The following example sets the rightMargin property of the selected text field from the character at index 2 up to but not including the character at index 8 to 100. `fl.getDocumentDOM().selection[0].setTextAttr("rightMargin", 100, 2, 8);`

Property	Data type	Value
rotation	Boolean	If true, the characters of the text are rotated 90 degrees; default is false. This applies only to Static Text with a vertical orientation; a warning dialog box appears if used with other text types. The following example sets the rotation of the selected text field to true. `fl.getDocumentDOM().setElementTextAttr("rotation", true);`
size	int	The size of the font. The following example gets the size of the character at index 2 and displays the result in the output panel. `fl.outputPanel.trace(fl.getDocumentDOM().selection[0].getTextAttr("size", 2));`
target	string	String for the target. This can be used only with Static Text. The following example gets the target property of the text field in the first frame, top layer of the current scene and displays it in the outputPanel. `fl.outputPanel.trace(fl.getDocumentDOM().getTimeline().layers[0].frames[0].elements[0].getTextAttr("target"));`
url	string	String of the URL. This can be used only with Static Text. The following example sets the URL of the selected text field to http://www.macromedia.com. `fl.getDocumentDOM().setElementTextAttr("url", "http://www.macromedia.com");`

TextRun

Description

The TextRun object represents a string of characters that have attributes that match all of the properties in the `TextAttrs` object (see "TextAttrs" on page 193).

Methods

The TextRun object has no unique methods.

Properties

Property	Data type	Value
`characters`	string	The text contained in the TextRun object.
`textAttrs`	object	TextAttrs object containing the attributes of the run of text.

Timeline

Description

The `Timeline` object represents the Timeline, which can be accessed for the current document by `fl.getDocumentDOM().getTimeline()`. This method returns the timeline of the current scene or symbol that is being edited.

When working with scenes, each scene's Timeline has an index value, and can be accessed for the current document by `fl.getDocumentDOM().timelines[i]`.

In this example, `i` is the index of the value of the Timeline.

When working with frames in the `Timeline` object methods and properties, remember that the frame value is a zero-based index value (not the actual frame number in the sequence of frames in the Timeline). So, the first frame has a frame index value of 0.

Methods

You can use the following methods with the `Timeline` object:

```
timeline.addMotionGuide()
timeline.addNewLayer()
timeline.clearFrames()
timeline.clearKeyframes()
timeline.convertToBlankKeyframes()
timeline.convertToKeyframes()
timeline.copyFrames()
timeline.createMotionTween()
timeline.cutFrames()
timeline.deleteLayer()
timeline.expandFolder()
timeline.findLayerIndex()
timeline.getFrameProperty()
timeline.getLayerProperty()
timeline.getSelectedFrames()
timeline.getSelectedLayers()
timeline.insertBlankKeyframe()
timeline.insertFrames()
timeline.insertKeyframe()
timeline.pasteFrames()
timeline.removeFrames()
timeline.reorderLayer()
timeline.reverseFrames()
timeline.selectAllFrames()
timeline.setFrameProperty()
timeline.setLayerProperty()
timeline.setSelectedFrames()
timeline.setSelectedLayers()
timeline.showLayerMasking()
```

timeline.addMotionGuide()

Description

Adds a motion guide layer above the current layer, and it attaches the current layer to the newly added guide layer.

Note: The `addMotionGuide` method only functions on a "`Normal`" type layer. It will not do anything to a "`Folder`", "`Mask`", "`Masked`", "`Guide`", or "`Guided`" layer.

Usage

```
timeline.addMotionGuide()
```

Arguments

None.

Returns

An integer value of the zero-based index of the newly added guide layer. If the current layer type is not Normal, then Flash returns -1.

Example

The following example adds a motion guide layer on top of the current layer, and converts the current layer to `"Guided"`:

```
fl.getDocumentDOM().getTimeline().addMotionGuide();
```

timeline.addNewLayer()

Description

Adds a new layer to the document and makes it the current layer.

Usage

```
timeline.addNewLayer( [name] [, layerType [, bAddAbove] ] )
```

Arguments

The optional *`name`* argument is a String that specifies the name for the new layer. If a name is not specified, a new default layer name is assigned to the new layer ("Layer n" where n is the total number of layers).

The optional `layerType` argument is a String that specifies the type of layer to add. If not specified, a `"Normal"` type layer is created.

The optional *`bAddAbove`* argument is a Boolean value that, if set to `true`, causes Flash to add the new layer above the current layer; if `false`, Flash adds the layer below the current layer. The default is `true`.

Returns

An integer value of the zero-based index of the newly added layer.

Example

The following example adds a new layer to the Timeline with a default name generated by Flash:

```
fl.getDocumentDOM().getTimeline().addNewLayer();
```

The following example adds a new folder layer on top of the current layer and names it "Folder1":

```
fl.getDocumentDOM().getTimeline().addNewLayer('Folder1', 'folder', true);
```

timeline.clearFrames()

Description

Deletes all the contents from within a frame or range of frames on the current layer.

Usage

```
timeline.clearFrames( [startFrameIndex [, endFrameIndex] ] )
```

Arguments

The optional *startFrameIndex* argument is an integer that is a zero-based index value that defines the beginning of the range of frames to clear. The range extends from *startFrameIndex* up to, but not including, *endFrameIndex*.

The optional *endFrameIndex* argument is an integer that is a zero-based index value that defines the end of the range of frames to clear. The range extends from *startFrameIndex* up to, but not including, *endFrameIndex*.

If *startFrameIndex* is not specified, the method uses the current selection. If *startFrameIndex* is specified, but *endFrameIndex* is not specified, *endFrameIndex* defaults to the same value as *startFrameIndex*.

Returns

Nothing.

Example

The following example clears the frames from frame 6 up to, but not including, frame 11:

```
fl.getDocumentDOM().getTimeline().clearFrames(5, 10);
```

The following example clears frame 15.

```
fl.getDocumentDOM().getTimeline().clearFrames(14);
```

timeline.clearKeyframes()

Description

Converts a keyframe to a regular frame and deletes its contents on the current layer.

Usage

```
timeline.clearKeyframes( [startFrameIndex [, endFrameIndex] ] )
```

Arguments

The optional *startFrameIndex* argument is an integer that is a zero-based index value that defines the beginning of the range of frames to clear. The range extends from *startFrameIndex* up to, but not including, *endFrameIndex*.

The optional *endFrameIndex* argument is an integer that is a zero-based index value that defines the end of the range of frames to clear. The range extends from *startFrameIndex* up to, but not including, *endFrameIndex*.

If *startFrameIndex* is not specified, Flash uses the current selection. If *startFrameIndex* is specified, but *endFrameIndex* is not specified, *endFrameIndex* defaults to the same value as *startFrameIndex*.

Returns

Nothing.

Example

The following example clears the keyframes from frame 5 up to, but not including, frame 10:

```
fl.getDocumentDOM().getTimeline().clearKeyframes(4, 9);
```

The following example clears the keyframe at frame 15 and converts it to a regular frame:

```
fl.getDocumentDOM().getTimeline().clearKeyframes(14);
```

timeline.convertToBlankKeyframes()

Description

Converts frames to blank keyframes from the *startFrameIndex* up to, but not including, the frame at *endFrameIndex* on the current layer.

Usage

```
timeline.convertToBlankKeyframes( [startFrameIndex [, endFrameIndex] ])
```

Arguments

The optional *startFrameIndex* argument is an integer that is a zero-based index value that specifies the starting frame to convert to keyframes.

The optional *endFrameIndex* argument is an integer that is a zero-based index value that specifies the frame at which the conversion to keyframes will stop. If *endFrameIndex* is not specified and *startFrameIndex* is specified, *endFrameIndex* defaults to the value of *startFrameIndex*.

Returns

Nothing.

Example

The following example converts frame 2 up to, but not including, frame 10 to blank keyframes:

```
fl.getDocumentDOM().getTimeline().convertToBlankKeyframes(1, 9);
```

The following converts frame 5 to a blank keyframe:

```
fl.getDocumentDOM().getTimeline().convertToBlankKeyframes(4);
```

timeline.convertToKeyframes()

Description

Converts to keyframes a range of frames specified from *startFrameIndex* up to, but not including, *endFrameIndex* (or the selection if no frames are specified) on the current layer.

Usage

```
timeline.convertToKeyframes([startFrameIndex [, endFrameIndex] ])
```

Arguments

The optional *startFrameIndex* argument is an integer that is a zero-based index value that specifies the first frame to convert to keyframes. If *startFrameIndex* is not specified, the currently selected frame is converted to a keyframe.

The optional *endFrameIndex* argument is an integer that is a zero-based index value that specifies the frame at which conversion to keyframes will stop. If *endFrameIndex* is not specified and *startFrameIndex* is specified, *endFrameIndex* defaults to the value of *startFrameIndex*.

Returns

Nothing.

Example

The following example converts the selected frames to keyframes:

```
fl.getDocumentDOM().getTimeline().convertToKeyframes();
```

The following example converts the frames from frame 2 up to, but not including, frame 10 to keyframes:

```
fl.getDocumentDOM().getTimeline().convertToKeyframes(1, 9);
```

The following example converts frame 5 to a keyframe:

```
fl.getDocumentDOM().getTimeline().convertToKeyframes(4);
```

timeline.copyFrames()

Description

Copies the frames in the range from *startFrameIndex* up to, but not including, *endFrameIndex* on the current layer to the Clipboard.

Usage

```
timeline.copyFrames( [startFrameIndex [, endFrameIndex] ] )
```

Arguments

The optional *startFrameIndex* argument is an integer that is a zero-based index that specifies the beginning of the range of frames to copy.

The optional *endFrameIndex* argument is an integer that is a zero-based index that specifies the frame at which to stop copying;

If *startFrameIndex* is not specified, Flash uses the current selection. If *startFrameIndex* is specified, but *endFrameIndex* is not specified, *endFrameIndex* defaults to the same value as *startFrameIndex*.

Returns

Nothing.

Example

The following example copies the selected frames to the Clipboard:

```
fl.getDocumentDOM().getTimeline().copyFrames();
```

The following example copies frame 2 up to, but not including frame 10, to the Clipboard:

```
fl.getDocumentDOM().getTimeline().copyFrames(1, 9);
```

The following example copies frame 5 to the Clipboard:

```
fl.getDocumentDOM().getTimeline().copyFrames(4);
```

timeline.createMotionTween()

Description

Sets the `tweenType` property to motion for each selected keyframe on the current layer, and converts the frame's contents to a single symbol instance if they are not already symbol instances. This is the equivalent to the Create Motion Tween menu item in the Flash UI.

Usage

```
timeline.createMotionTween( [startFrameIndex [, endFrameIndex] ] )
```

Arguments

The optional *`startFrameIndex`* argument is an integer that is a zero-based index that specifies the beginning frame at which to create a motion tween; *`startFrameIndex`* up to, but not including, *`endFrameIndex`* is the range of frames. If *`startFrameIndex`* is not specified, Flash uses the current selection.

The optional *`endFrameIndex`* argument is an integer that is a zero-based index that specifies the frame at which to stop the motion tween; *`startFrameIndex`* up to, but not including, *`endFrameIndex`* is the range of frames. If *`startFrameIndex`* is specified, but *`endFrameIndex`* is not specified, *`endFrameIndex`* defaults to the *`startFrameIndex`* value.

Returns

Nothing.

Example

The following example converts the shape in the first frame up to, but not including, frame 10 to a graphic symbol instance and sets the frame `tweenType` to motion:

```
fl.getDocumentDOM().getTimeline().createMotionTween(0, 9);
```

timeline.cutFrames()

Description

Cuts a range of frames on the current layer from the Timeline and saves them to the Clipboard.

Usage

```
timeline.cutFrames( [startFrameIndex [, endFrameIndex] ] )
```

Arguments

The optional *startFrameIndex* argument is an integer that is a zero-based index that specifies the beginning of a range of frames to cut; *startFrameIndex* up to, but not including, *endFrameIndex* is the range of frames. If *startFrameIndex* is not specified, the method uses the current selection.

The optional *endFrameIndex* argument is a zero-based index that specifies the frame at which to stop cutting; *startFrameIndex* up to, but not including, *endFrameIndex* is the range of frames. If *startFrameIndex* is specified, but *endFrameIndex* is not specified, *endFrameIndex* defaults to the *startFrameIndex* value.

Returns

Nothing.

Example

The following example cuts the selected frames from the Timeline and saves them to the Clipboard:

```
fl.getDocumentDOM().getTimeline().cutFrames();
```

The following example cuts frame 2 up to, but not including, frame 10 from the Timeline and saves them to the Clipboard:

```
fl.getDocumentDOM().getTimeline().cutFrames(1, 9);
```

The following example cuts frame 5 from the Timeline and saves it to the Clipboard:

```
fl.getDocumentDOM().getTimeline().cutFrames(4);
```

timeline.deleteLayer()

Description

Deletes a layer. If the layer is a folder, all layers within the folder will be deleted. If the layer index is not specified, Flash deletes the currently selected layers.

Usage

```
timeline.deleteLayer( [index] )
```

Arguments

The optional *index* argument is an integer that is a zero-based index that specifies the layer to be deleted. If there is only one layer in the Timeline, this method has no effect.

Returns

Nothing.

Example

The following example deletes the second layer from the top:

```
fl.getDocumentDOM().getTimeline().deleteLayer(1);
```

The following example deletes the currently selected layers:

```
fl.getDocumentDOM().getTimeline().deleteLayer();
```

timeline.expandFolder()

Description

Expands or collapses the specified folder or folders. Operates on the current layer if a layer is not specified.

Usage

```
timeline.expandFolder( bExpand [, bRecurseNestedParents [, index] ] )
```

Arguments

The *bExpand* argument is a Boolean value that, if set to `true`, causes the method to expand the folder; if `false`, the method collapses the folder.

The optional *bRecurseNestedParents* argument is a Boolean value that, if set to `true`, causes all the layers within the specified folder to be opened or closed, based on the *bExpand* argument.

The optional *index* argument is an integer that specifies the zero-based index value of the folder to expand or collapse. Use `-1` to apply to all layers (you also need to set `recurseNestedFolders` to `true`). This is equivalent to the Expand All/Collapse All menu items in the Flash UI.

Returns

Nothing.

Example

The following examples use this folder structure.

```
Folder 1 ***
--layer 7
--Folder 2 ****
----Layer 5
```

The following example expands Folder 1 only:

```
fl.getDocumentDOM().getTimeline().currentLayer = 1;
fl.getDocumentDOM().getTimeline().expandFolder(true);
```

The following example expands Folder 1 only (assuming that Folder 2 collapsed when Folder 1 last collapsed; otherwise, Folder 2 appears expanded):

```
fl.getDocumentDOM().getTimeline().expandFolder(true, false, 0);
```

The following example collapses all folders in the current Timeline:

```
fl.getDocumentDOM().getTimeline().expandFolder(false, true, -1);
```

timeline.findLayerIndex()

Description

Finds an array of indexes for the layers with the given name. The layers index is flat, so folders are considered part of the main index.

Usage

```
timeline.findLayerIndex( name )
```

Arguments

The *name* argument is a String that specifies the name of the layer to find.

Returns

An array of index values for the specified layer. If the specified layer is not found, Flash returns `undefined`.

Example

The following example shows the index values of all layers named Layer 7 in the Output panel:

```
var layerIndex = fl.getDocumentDOM().getTimeline().findLayerIndex("Layer 7");
fl.trace(layerIndex);
```

timeline.getFrameProperty()

Description

Gets the specified property's value of the selected frames.

Usage

```
timeline.getFrameProperty( property [, startframeIndex [, endFrameIndex] ] )
```

Arguments

The *property* argument is a String that specifies the name of the property for which to get the value. See "Frame" on page 114 for a complete list of properties.

The optional *startFrameIndex* argument is an integer that is a zero-based index that specifies starting frame number for which to get the value. If *startFrameIndex* is not specified, the existing selection is used.

The optional *endFrameIndex* argument is an integer that is a zero-based index that specifies the ending frame to select. The range includes the frames up to, but not including, the frame specified by *endFrameIndex*. If *endFrameIndex* is not specified, Flash will default to the *startFrameIndex* value to stop.

Returns

A value for the specified property, or `undefined` if all the selected frames do not have the same property value.

Example

The following example gets the name of the first frame in the top layer of the current document and displays the name in the Output panel:

```
fl.getDocumentDOM().getTimeline().currentLayer = 0;
fl.getDocumentDOM().getTimeline().setSelectedFrames(0, 0, true);
var frameName = fl.getDocumentDOM().getTimeline().getFrameProperty("name");
fl.trace(frameName);
```

timeline.getLayerProperty()

Description

Gets the specified property's value for the selected layers.

Usage

```
timeline.getLayerProperty( property )
```

Arguments

The *property* argument is String that specifies the name of the property for which to get the value. See "Layer" on page 125 for a list of properties.

Returns

The value of the specified property. Flash looks at the layer's properties to determine the type. If all the specified layers don't have the same property value, Flash returns `undefined`.

Example

The following example gets the name of the top layer in the current document and displays it in the Output panel:

```
fl.getDocumentDOM().getTimeline().currentLayer = 0;
var layerName = fl.getDocumentDOM().getTimeline().getLayerProperty("name");
fl.trace(layerName);
```

timeline.getSelectedFrames()

Gets the currently selected frames in an array.

Arguments

None.

Returns

An array containing 3`n` integers, where `n` is the number of selected regions. The first integer in each pair is the layer index, the second is the start frame of the beginning of the selection, and the third integer specifies the ending frame of that selection range. The ending frame is not included in the selection.

Example

With the top layer being the current layer, the following example shows `0,5,10,0,20,25` in the Output panel:

```
var timeline = fl.getDocumentDOM().getTimeline();
timeline.setSelectedFrames(5,10);
timeline.setSelectedFrames(20,25,false);
var theSelectedFrames = timeline.getSelectedFrames();
fl.trace(theSelectedFrames);
```

timeline.getSelectedLayers()

Description

Gets the zero-based index values of the currently selected layers.

Arguments

None.

Returns

An array of the zero-based index values of the selected layers.

Example

The following example shows `1,0` in the Output panel:

```
fl.getDocumentDOM().getTimeline().setSelectedLayers(0);
fl.getDocumentDOM().getTimeline().setSelectedLayers(1, false);
var layerArray = fl.getDocumentDOM().getTimeline().getSelectedLayers();
fl.trace(layerArray);
```

timeline.insertBlankKeyframe()

Description

Inserts a blank keyframe at the specified frame index; if the frame index is not specified, inserts the blank keyframe using the playhead/selection.

Usage

```
timeline.insertBlankKeyframe( [frameNumIndex] )
```

Arguments

The optional *`frameNumIndex`* argument is an integer that is a zero-based index that specifies the frame at which to insert the keyframe. If *`frameNumIndex`* is not specified, the method uses the current playhead frame number.

If the specified or selected frame is a regular frame, the keyframe is inserted at the frame. For example, if you have a span of 10 frames from numbers 1-10 and you select frame 5 executing `timeline.insertBlankKeyframe()` makes frame 5 a blank keyframe, and the length of the frame span is still 10 frames. If frame 5 is selected and it is a keyframe with a regular frame next to it, `timeline.insertBlankKeyframe()` will insert a blank keyframe at frame 6. If frame 5 is a keyframe and the frame next to it is already a keyframe then no keyframe is inserted but the playhead just moves to frame 6.

Returns

Nothing.

Example

The following example inserts a blank keyframe at frame 20:

```
fl.getDocumentDOM().getTimeline().insertBlankKeyframe(19);
```

The following example insert a blank keyframe at the currently selected frame (or playhead location if none is selected):

```
fl.getDocumentDOM().getTimeline().insertBlankKeyframe();
```

timeline.insertFrames()

Description

Inserts the specified number of frames at the given frame number. If no arguments are specified, the method inserts frames using the current selection (using the selection start frame for location and length to determine how many frames to insert at each group of selected frames). If no selection is present, the method will insert one frame at the current frame into all layers. Otherwise, the method will insert the number of frames specified by the `numFrames` argument into either all the layers (based on the `bAllLayers` argument) or into the current layer at the specified `frameNum` argument (or the current frame, if unspecified).

If the specified or selected frame is a regular frame, the frame is inserted at that frame. For example, if you have a span of 10 frames from numbers 1-10 and you select frame 5, `timeline.insertFrames()` adds a frame at frame 5 and the length of the frame span becomes 11 frames. If frame 5 is selected and it is a keyframe, `timeline.insertFrames()` inserts a frame at frame 6 regardless of whether or not the frame next to it is also a keyframe.

Usage

```
timeline.insertFrames( [numFrames [, bAllLayers [, frameNumIndex] ] ] )
```

Arguments

The optional *`numFrames`* argument is an integer that specifies the number of frames to insert. If not specified, the method inserts frames at the current selection in the current layer

The optional *`bAllLayers`* argument is a Boolean value that, if set to `true`, causes the method to insert the specified number of frames in the *`numFrames`* argument into all layers; if set to `false`, the method inserts frames into the current layer. The default value is `true`.

The optional *`frameNumIndex`* argument is an integer that is a zero-based index that specifies the frame index at which to insert a new frame.

Returns

Nothing.

Example

The following example inserts a frame (or frames, depending on the selection) at the current selection in the current layer:

```
fl.getDocumentDOM().getTimeline().insertFrames();
```

The following example inserts five frames at the current frame in all layers:

Note: If you have multiple layers with frames in them, and then select a frame in one layer when using the following command, Flash inserts the frames in the selected layer only. If you have multiple layers with no frames selected in them, Flash inserts the frames in all layers.

```
fl.getDocumentDOM().getTimeline().insertFrames(5);
```

The following example inserts three frames in the current layer only:

```
fl.getDocumentDOM().getTimeline().insertFrames(3, false);
```

The following example inserts four frames in all layers, starting from the first frame:

```
fl.getDocumentDOM().getTimeline().insertFrames(4, true, 0);
```

timeline.insertKeyframe()

Description

Inserts a keyframe at the specified frame. If the `frameNumIndex` argument is not specified, the method inserts a keyframe using the playhead or selection location.

Works the same as `timeline.insertBlankKeyframe()` except that the inserted keyframe contains the contents of the frame it converted (that is, it's not blank).

Usage

```
timeline.insertKeyframe( [frameNumIndex] )
```

Arguments

The optional *`frameNumIndex`* argument is an integer that is a zero-based index that specifies the frame index at which to insert the keyframe in the current layer. If *`frameNumIndex`* is not specified, the method uses the frame number of the current playhead or selected frame.

Returns

Nothing.

Example

The following example inserts a keyframe at the playhead or selected location:

```
fl.getDocumentDOM().getTimeline().insertKeyframe();
```

The following example inserts a keyframe at frame 10 of the second layer:

```
fl.getDocumentDOM().getTimeline().currentLayer = 1;
fl.getDocumentDOM().getTimeline().insertKeyframe(9);
```

timeline.pasteFrames()

Description

Pastes the range of frames from the Clipboard into the specified frames.

Usage

```
timeline.pasteFrames( [startFrameIndex [, endFrameIndex] ] )
```

Arguments

The optional *`startFrameIndex`* is a zero-based index that specifies the beginning of a range of frames to paste; pastes from *`startFrameIndex`* up to, but not including, the frame specified by *`endFrameIndex`*. If *`startFrameIndex`* is not specified, the method uses the current selection. If *`startFrameIndex`* is specified, but *`endFrameIndex`* is not specified, *`endFrameIndex`* defaults to the *`startFrameIndex`* value.

The optional *`endFrameIndex`* is an integer that is a zero-based index that specifies the frame at which to stop pasting frames; pastes up to, but not including, *`endFrameIndex`*. If *`startFrameIndex`* is specified, but *`endFrameIndex`* is not specified, *`endFrameIndex`* defaults to the *`startFrameIndex`* value.

Returns

Nothing.

Example

The following example pastes the frames on the Clipboard to the currently selected frame or playhead location:

```
fl.getDocumentDOM().getTimeline().pasteFrames();
```

The following example pastes the frames on the Clipboard at frame 2 up to, but not including, frame 10:

```
fl.getDocumentDOM().getTimeline().pasteFrames(1, 9);
```

The following example pastes the frames on the Clipboard starting at frame 5:

```
fl.getDocumentDOM().getTimeline().pasteFrames(4);
```

timeline.removeFrames()

Description

Deletes the frame.

Usage

```
timeline.removeFrames( [startFrameIndex [, endFrameIndex] ] )
```

Arguments

The optional *startFrameIndex* is an integer that is a zero-based index that specifies the first frame at which to start removing frames; *startFrameIndex* up to, but not including, *endFrameIndex* is the range of frames to remove. If *startFrameIndex* is not specified, the method uses the current selection; if there is no selection, all frames at the current playhead on all layers will be removed. If *startFrameIndex* is specified, but *endFrameIndex* is not specified, *endFrameIndex* defaults to the *startFrameIndex* value.

The optional *endFrameIndex* argument is an integer that is a zero-based index that specifies the frame at which to stop removing frames; *startFrameIndex* up to, but not including, *endFrameIndex* is the range of frames to remove. If *startFrameIndex* is not specified, the method uses the current selection; if there is no selection, all frames at the current playhead on all layers will be removed. If *startFrameIndex* is specified, but *endFrameIndex* is not specified, *endFrameIndex* defaults to the *startFrameIndex* value.

Returns

Nothing.

Example

The following example deletes frame 5, up to, but not including, frame 10 of the top layer in the current scene:

```
fl.getDocumentDOM().getTimeline().currentLayer = 0;
fl.getDocumentDOM().getTimeline().removeFrames(4, 9);
```

The following example deletes frame 8 on the top layer in the current scene:

```
fl.getDocumentDOM().getTimeline().currentLayer = 0;
fl.getDocumentDOM().getTimeline().removeFrames(7);
```

timeline.reorderLayer()

Description

Moves the specified layer before or after the specified layer.

Usage

```
timeline.reorderLayer( layerToMove, layerToPutItBy [, bAddBefore] )
```

Arguments

The *layerToMove* argument is an integer that is a zero-based index value that specifies which layer to move.

The *layerToPutItBy* argument is an integer that is a zero-based index value that specifies which layer you want to move the layer next to. For example, if you specify 1 for *layerToMove* and 0 for *layerToPutItBy*, the second item is placed before the first layer.

The optional *bAddBefore* argument, if `true`, moves the layer before *layerToPutItBy*. If `false`, moves the layer after *layerToPutItBy*. The default value is `true`.

Returns

Nothing.

Example

The following example moves the layer at index 2 to the top (on top of layer at index 0):

```
fl.getDocumentDOM().getTimeline().reorderLayer(2, 0);
```

The following example moves the layer at index 3 to after the layer at index 5:

```
fl.getDocumentDOM().getTimeline().reorderLayer(3, 5, false);
```

timeline.reverseFrames()

Description

Reverses a range of frames.

Usage

```
timeline.reverseFrames( [startFrameIndex [, endFrameIndex] ] )
```

Arguments

The optional *startFrameIndex* argument is an integer that is a zero-based index that specifies the first frame at which to start reversing frames; *startFrameIndex* up to, but not including, *endFrameIndex* is the range of frames. If *startFrameIndex* is not specified, Flash uses the current selection.

The optional *endFrameIndex* argument is an integer that is a zero-based index that specifies the first frame at which to stop reversing frames; *startFrameIndex* up to, but not including, *endFrameIndex* is the range of frames. If *startFrameIndex* is not specified, Flash uses the current selection.

Returns

Nothing.

Example

The following example reverses frames of the currently selected frames:

```
fl.getDocumentDOM().getTimeline().reverseFrames();
```

The following example reverses frames from frame 10 up to, but not including, frame 15:

```
fl.getDocumentDOM().getTimeline().reverseFrames(9, 14);
```

timeline.selectAllFrames()

Description

Selects all the frames in the current Timeline.

Arguments

None.

Returns

Nothing.

Example

```
fl.getDocumentDOM().getTimeline().selectAllFrames();
```

timeline.setFrameProperty()

Description

Sets the property of the Frame object for the selected frames.

Usage

```
timeline.setFrameProperty( property, value [, startFrameIndex [,
  endFrameIndex] ] )
```

Arguments

The *property* argument is a String that specifies the name of the property to be modified. See "Frame" on page 114 for complete list of properties.

Note: Duration, elements, and startFrameIndex are read-only properties. Cannot `setFrameProperty` to read-only items.

The *value* argument specifies new value to which you want to set the property. See Frame properties to determine the appropriate values and type.

The optional *startFrameIndex* argument is an integer that is a zero-based index that specifies starting frame number to modify. If *startFrameIndex* is not specified, the existing selection is used.

The optional *endFrameIndex* argument is an integer that is a zero-based index that specifies the first frame at which to stop, up to, but not including, the *endFrameIndex* number to be modified. If *endFrameIndex* is not specified, Flash will default to the *startFrameIndex* value to stop.

Returns

Nothing.

Example

The following example assigns the ActionScript `stop()` command to the first frame of the top layer in the current document:

```
fl.getDocumentDOM().getTimeline().currentLayer = 0;
fl.getDocumentDOM().getTimeline().setSelectedFrames(0,0,true);
fl.getDocumentDOM().getTimeline().setFrameProperty("actionScript", "stop();");
```

The following example sets a motion tween to frame 2 up to, but not including, frame 5, of the current layer:

```
fl.getDocumentDOM().getTimeline().setFrameProperty("tweenType","motion",1,4);
```

timeline.setLayerProperty()

Description

Sets the specified property on all the selected layers to value.

Usage

```
timeline.setLayerProperty( property, value [, layersToChange] )
```

Arguments

The *property* argument is a String that specifies the property to set. See "Layer" on page 125 for a list of properties.

The *value* argument is the value to which you want to set the property. Use the same type of value you would use when setting the property on the Layer object.

The optional *layertsToChange* argument is a String that identifies which layers should be modified. Choices are `"selected"`, `"all"`, and `"others"`. The default value is `"selected"` if a value is not specified.

Returns

Nothing.

Example

The following example sets the selected layer(s) to invisible:

```
fl.getDocumentDOM().getTimeline().setLayerProperty("visible", false);
```

The following example sets the name of the selected layer(s) to `"selLayer"`:

```
fl.getDocumentDOM().getTimeline().setLayerProperty("name", "selLayer");
```

timeline.setSelectedFrames()

Description

Selects a range of frames in the current layer or sets the selected frames to the selection array passed into this method.

Usage

```
timeline.setSelectedFrames( startFrameIndex, endFrameIndex
  [, bReplaceCurrentSelection] )
timeline.setSelectedFrames( selectionList [, bReplaceCurrentSelection] )
```

Arguments

The *startFrameIndex* argument is an integer that is a zero-based index that specifies the beginning frame to set; *startFrameIndex* up to, but not including, *endFrameIndex* specifies the range of frames to select.

The *endFrameIndex* argument is an integer that is a zero-based index that specifies the end of the selection; *endFrameIndex* is the frame after the last frame in the range to select.

The *bReplaceCurrentSelection* argument is a Boolean value that, if it is set to `true`, causes the currently selected frames to be deselected before selecting the specified frames. The default value is `true`.

The *selectionList* argument is an array of three integers, as returned by timeline.getSelectedFrames() on page 206.

Returns

Nothing.

Example

The following example selects the top layer, frame 1, up to, but not including, frame 10; then adds frame 12 up to, but not including, frame 15 on the same layer to the current selection:

```
fl.getDocumentDOM().getTimeline().setSelectedFrames(0, 9);
fl.getDocumentDOM().getTimeline().setSelectedFrames(11, 14, false);
```

The following commands first stores the array of selected frames in the *savedSelectionList* variable, and then show using the array later in the code to reselect those frames after some command or user interaction has changed the selection:

```
var savedSelectionList =
  fl.getDocumentDOM().getTimeline().getSelectedFrames();
// do something that changes the selection
fl.getDocumentDOM().getTimeline().setSelectedFrames(savedSelectionList);
```

The following example selects the top layer, frame 1, up to, but not including, frame 10, then adds frame 12, up to, but not including, frame 15, on the same layer to the current selection:

```
fl.getDocumentDOM().getTimeline().setSelectedFrames([0, 0, 9]);
fl.getDocumentDOM().getTimeline().setSelectedFrames([0, 11, 14], false);
```

timeline.setSelectedLayers()

Description

Sets the layer to be selected, also makes the specified layer the current layer. Selecting a layer also means that all the frames in the layer are selected.

Usage

```
timeline.setSelectedLayers( index [, bReplaceCurrentSelection] )
```

Arguments

The *index* argument is an integer that is a zero-based index for the layer to select.

The optional *bReplaceCurrentSelection* argument is a Boolean value that, if it is set to `true`, causes the method to replace the current selection; if `false`, the method extends the current selection.The default value is `true`.

Returns

Nothing.

Example

The following example selects the top layer:

```
fl.getDocumentDOM().getTimeline().setSelectedLayers(0);
```

The following example adds the next layer to the selection:

```
fl.getDocumentDOM().getTimeline().setSelectedLayers(1, false);
```

timeline.showLayerMasking()

Description

Shows the layer masking during authoring by locking the mask and masked layers. Uses the current layer if no layer is specified. If used on a layer that is not of type Mask or Masked, Flash will display an error in the Output panel.

Usage

```
timeline.showLayerMasking( [layer] )
```

Arguments

The optional *layer* argument is an integer that specifies the zero-based index value of a mask or masked layer to show masking during authoring.

Returns

Nothing.

Example

```
fl.getDocumentDOM().getTimeline().showLayerMasking(0);
```

Properties

Property	Data type	Value

currentFrame	int	The zero-based index value for the frame at the current playhead location.
		The following example sets the playhead of scene 1 to frame 10:
		`fl.getDocumentDOM().timelines[0].currentFrame = 9;`
		The following example stores the value for the current playhead location in the *curFrame* variable:
		`var curFrame = fl.getDocumentDOM().getTimeline().currentFrame;`

currentLayer	int	The currently active layer.
		The following example sets the top layer to active:
		`fl.getDocumentDOM().getTimeline().currentLayer = 0;`
		The following example stores the index value of the currently active layer in the *curLayer* variable:
		`var curLayer = fl.getDocumentDOM().getTimeline().currentLayer;`

Property	Data type	Value
`frameCount` **read only**	int	The length of the longest layer in this timeline. The following example stores the number of frames in the longest layer in the current document in the *`countNum`* variable: `var countNum = fl.getDocumentDOM().getTimeline().frameCount;`
`layerCount` **read only**	int	The number of layers in the specified Timeline. The following example stores the number of layers in the current scene in the *`NumLayer`* variable: `var NumLayer = fl.getDocumentDOM().getTimeline().layerCount;`
`layers` **read only**	array	An array of layer objects. The following example stores the array of layer objects in the current document in the *`currentLayers`* variable: `var currentLayers = fl.getDocumentDOM().getTimeline().layers;`

`name`	string	Name of the current Timeline. This will be the name of the current scene, screen (slide or form), or symbol that is being edited.
		The following example gets the first scene name:
		`var sceneName = fl.getDocumentDOM().timelines[0].name;`
		The following example sets the first scene name to `FirstScene`:
		`fl.getDocumentDOM().timelines[0].name = "FirstScene";`

ToolObj

Description

A ToolObj object represents an individual tool in the Toolbar. To access a ToolObj object, use properties of the Tools object: either the `tools.toolObjs` array or `tools.activeTool`.

Note: Used only when creating extensible tools.

Methods

You can use the following methods with the `ToolObj` object:

```
toolObj.enablePIControl()
toolObj.setIcon()
toolObj.setMenuString()
toolObj.setOptionsFile()
toolObj.setPI()
toolObj.setToolName()
toolObj.setToolTip()
toolObj.showPIControl()
toolObj.showTransformHandles()
```

toolObj.enablePIControl()

Description

Enables or disables the specified control in a PI. Used only when creating extensible tools.

Usage

```
toolObj.enablePIControl( control, bEnable )
```

Arguments

The *control* argument is a String that specifies the name of the control to enable or disable. Legal values depend on the Property inspector invoked by this tool (see `toolObj.setPI()`).

A shape Property inspector has the following controls:

stroke	fill

A text Property inspector has the following controls:

type	font	pointsize
color	bold	italic
direction	alignLeft	alignCenter
alignRight	alignJustify	spacing
position	autoKern	small
rotation	format	lineType
selectable	html	border
deviceFonts	varEdit	options
link	maxChars	target

A movie Property inspector has the following controls:

size	publish	background
framerate	player	profile

The *bEnable* argument is a Boolean value that determines whether to enable (`true`) or disable (`false`) the control.

Returns

Nothing.

Example

The following command in an extensible tool's JavaScript file will set Flash to not show the stroke options in the Property inspector for that tool:

```
theTool.enablePIControl( "stroke",  false);
```

toolObj.setIcon()

Description

Identifies a PNG file to use as a tool icon in the Flash toolbar.

Usage

```
toolObj.setIcon( file )
```

Arguments

The *file* argument is a String that specifies the name of the PNG file to use as the icon. The PNG file must be placed in the Configuration/Tools folder.

Returns

Nothing.

Example

```
theTool.setIcon("arrow1.png");
```

toolObj.setMenuString()

Description

Sets the string that appears in the pop-up menu as the name for the tool.

Usage

```
toolObj.setMenuString( menuStr )
```

Arguments

The *menuStr* argument is a String that specifies the name that appears in the pop-up menu as the name for the tool.

Returns

Nothing.

Example

```
theTool.setMenuString("Arrow Style 1");
```

toolObj.setOptionsFile()

Description

Associates an XML file (located in the Configuration/Tools folder) with the tool to appear in a modal panel that is invoked by an Options... button in the Property inspector.

Usage

```
toolObj.setOptionsFile( xmlFile )
```

Arguments

The *xmlFile* argument is a String that specifies the name of the XML file that has the description of the tool's options.

Returns

Nothing.

Example

```
fl.tools.activeTool.setOptionsFile( "myTool.xml" );
```

toolObj.setPI()

Description

Sets a particular Property inspector to be used when the tool is activated. Valid values are `"shape"`, `"text"`, and `"movie"`. The shape Property inspector is used as a default if no Property inspector is specified.

Usage

```
toolObj.setPI( pi )
```

Arguments

The *pi* argument is a String that specifies the Property inspector to invoke for this tool.

Returns

Nothing.

Example

```
fl.tools.activeTool.setPI( "shape" );
```

toolObj.setToolName()

Description

Assigns a name to the tool for the configuration of the toolbar. The name is used only by the XML layout file that Flash reads to construct the toolbar. The name does not show up in the Flash UI.

Usage

```
toolObj.setToolName( name )
```

Arguments

The *name* argument is a String that specifies the name of the tool.

Returns

Nothing.

Example

```
theTool.setToolName("arrow1");
```

toolObj.setToolTip()

Description

Sets the tooltip that appears when the mouse is held over the tool icon.

Usage

```
toolObj.setToolTip( toolTip )
```

Arguments

The *toolTip* argument is a String that specifies the tooltip to use for the tool.

Returns

Nothing.

Example

```
fl.tools.activeTool.setToolTip("Arrow Style 1 Tool");
```

toolObj.showPIControl()

Description

Shows or hides a control in the Property inspector.

Usage

```
toolObj.showPIControl( control, bShow )
```

Arguments

The *control* argument is a String that specifies the name of the control to show or hide. Valid values depend on the Property Inspector invoked by this tool (see `toolObj.setPI()`).

A shape Property inspector has the following controls:

stroke	fill

A text Property inspector has the following controls:

type	font	pointsize
color	bold	italic
direction	alignLeft	alignCenter
alignRight	alignJustify	spacing

position	autoKern	small
rotation	format	lineType
selectable	html	border
deviceFonts	varEdit	options
link	maxChars	target

The movie Property inspector has the following controls:

size	publish	background
framerate	player	profile

The *bShow* argument is a Boolean value that determines whether to show or hide the specified control; `true` shows the control; `false` hides the control.

Returns

Nothing.

Example

The following command in an extensible tool's JavaScript file will set Flash to not show the fill options in the Property inspector for that tool:

```
fl.tools.activeTool.showPIControl( "fill", false );
```

toolObj.showTransformHandles()

Description

Call this method in the `configureTool()` method of an extensible tool's JavaScript file to indicate that the free transform handles should appear when the tool is active.

Usage

```
toolObj.showTransformHandles( bShow )
```

Arguments

The *bShow* argument is a Boolean value that determines whether to show or hide the free transform handles for the current tool (`true` shows the handles; `false` hides them).

Returns

Nothing.

Example

```
function configureTool() {
  theTool = fl.tools.activeTool;
  theTool.setToolName("ellipse");
  theTool.setIcon("Ellipse.png");
  theTool.setMenuString("Ellipse");
  theTool.setToolTip("Ellipse");
  theTool.showTransformHandles( true );
}
```

Properties

The ToolObj object has the following properties.

Property	Data type	Value

`position`
read only
	int	An integer specifying the position of the tool in the toolbar. Used only when creating extensible tools.
		The following commands in the `mouseDown()` method of an extensible tool's JavaScript file will show that tool's position in the toolbar as an integer in the Output panel:
		`myToolPos = fl.tools.activeTool.position;`
		`fl.trace(myToolPos);`

Tools

Description

The Tools object is accessible from the Flash object (`fl.tools`). The `Tools.toolObjs` property contains an array of ToolObj objects, and the `Tools.activeTool` property returns the ToolObj object for the currently active tool.

Note: The following methods and properties are used only when creating extensible tools.

Methods

You can use the following methods with the Tools object:

```
tools.getKeyDown()
tools.setCursor()
```

tools.getKeyDown()

Description

Returns the most recently pressed key.

Usage

```
tools.getKeyDown()
```

Arguments

None.

Returns

The integer value of the key.

Example

```
var theKey = fl.tools.getKeyDown();
fl.trace(theKey);
```

tools.setCursor()

Description

Sets the pointer to a specified appearance.

Usage

```
tools.setCursor( cursor )
```

Arguments

The *cursor* argument is an integer that specifies the number that defines the pointer appearance, as described in the following list:

- 0 Plus cursor (+)
- 1 black arrow
- 2 white arrow
- 3 four-way arrow
- 4 two-way horizontal arrow

- 5 two-way vertical arrow
- 6 X
- 7 hand cursor

Returns

Nothing.

Example

The following example sets the pointer to a black arrow.

```
fl.tools.setCursor(1);
```

Properties

Property	Data type	Value
`activeTool` **read only**	object	Returns the ToolObj object for the currently active tool. See "ToolObj" on page 217 for details. The following example saves the `activeTool` property in the *`theTool`* variable. `theTool = fl.tools.activeTool;`
`altIsDown` **read only**	Boolean	Identifies if the Alt key is being pressed. The value is `true` if the Alt key is pressed; `false` otherwise. **Example:** `var isAltDown = fl.tools.altIsDown;`
`constrainPoint` **read only**	point	This property takes two points and returns a new adjusted or *constrained* point. The two points are the starting-click point and the drag-to point. If the Shift key is pressed, then the returned point is constrained to follow either a 45 degree constrain (useful for something such as a line with an arrowhead) or to constrain an object to maintain its aspect ratio (such as pulling out a perfect square with the rectangle tool). Example: `pt2 = fl.tools.constrainPoint(pt1, tempPt);`
`ctlIsDown` **read only**	Boolean	Identifies if the Control key is being pressed. The value is `true` if the Control key is pressed; `false` otherwise. Example: `var isCtrldown = fl.tools.ctrlIsDown;`
`mouseIsDown` **read only**	Boolean	Identifies if a mouse button is currently pressed. The value is `true` if the left mouse button is currently down, `false` if the mouse button is up. Example: `var isMouseDown = fl.tools.mouseIsDown;`
`penDownLoc` **read only**	point	The position of the last mouse-down event on the Stage. `penDownLoc` has two properties, *x* and *y*, corresponding to the *x*,*y* location of the mouse on the current document. Example: `var pt1 = fl.tools.penDownLoc;`

Property	Data type	Value
`penLoc` **read only**	point	The current location of the mouse. `penLoc` has two properties: *x* and *y*, corresponding to the *x,y* location of the mouse on the current document. Example: `var tempPt = fl.tools.penLoc;`
`shiftIsDown` **read only**	Boolean	Identifies if the Shift key is being pressed. The value is `true` if the Shift key is pressed; `false` otherwise. Example: `var isShiftDown = fl.tools.shiftIsDown;`
`snapPoint` **read only**	point	This takes a point as input and returns a new point that may be adjusted or *snapped* to the nearest geometric object. If snapping is turned off in the View menu in the Flash UI, the point returned is the original point. Example: `var theSnapPoint = fl.tools.snapPoint(pt1);`
`toolObjs` **read only**	array	An array of ToolObj objects. See "ToolObj" on page 217 for details. Example: `var myTool = fl.tools.toolObjs[i];`

Vertex

Description

The Vertex object is the part of the shape data structure that holds the coordinate data.

Methods

You can use the following methods with the `Vertex` object:

`vertex.getHalfEdge()`
`vertex.setLocation()`

vertex.getHalfEdge()

Description

Gets a HalfEdge object that shares this vertex.

Usage

```
vertex.getHalfEdge()
```

Arguments

None.

Returns

A HalfEdge object.

Example

```
var shape = fl.getDocumentDOM().selection[0];
var hEdge = shape.edges[0].getHalfEdge(0);
var theVertex = hEdge.getVertex();
var someHEdge = theVertex.getHalfEdge(); // not necessarily the same half edge
var theSameVertex = someHEdge.getVertex();
fl.trace('the same vertex: ' + theSameVertex);
```

vertex.setLocation()

Description

Sets the location of the vertex. You must call `shape.beginEdit()` before using this method.

Usage

```
vertex.setLocation( x, y )
```

Arguments

The *x* argument is a floating point value that specifies the *x* coordinate of where the vertex should be positioned, in pixels.

The *y* argument is a floating point value that specifies the *y* coordinate of where the vertex should be positioned, in pixels.

Returns

Nothing.

Example

```
var shape = fl.getDocumentDOM().selection[0];
var hEdge = shape.edges[0].getHalfEdge(0);
var theVertex = hEdge.getVertex();
var someHEdge = theVertex.getHalfEdge();
var vertex = someHEdge.getVertex();
// move the vertex to the origin
vertex.setLocation(0.0, 0.0);
```

Properties

The Vertex object has the following properties:

Property	Data type	Value

x
read only
	float	The *x* location of the vertex in pixels.
		Example:
		`var shape = fl.getDocumentDOM().selection[0];`
		`var hEdge = shape.edges[0].getHalfEdge(0);`
		`var theVertex = hEdge.getVertex();`
		`var someHEdge = theVertex.getHalfEdge(); // not necessarily the same half edge`
		`var theSameVertex = someHEdge.getVertex();`
		`fl.trace('x location of vertex: ' + theSameVertex.x);`

y
read only
	float	The *y* location of the vertex in pixels.
		Example:
		`var shape = fl.getDocumentDOM().selection[0];`
		`var hEdge = shape.edges[0].getHalfEdge(0);`
		`var theVertex = hEdge.getVertex();`
		`var someHEdge = theVertex.getHalfEdge(); // not necessarily the same half edge`
		`var theSameVertex = someHEdge.getVertex();`
		`fl.trace('y location of vertex: ' + theSameVertex.y);`

VideoItem

Description

The VideoItem object is a subclass of the Item object. It does not have any unique methods or properties.

XMLUI

Description

Flash MX 2004 supports custom dialog boxes written in a subset of the XML User Interface Language (XUL). You can write a dialog.xml file and then invoke it from the JavaScript API using the `document.xmlPanel()` method.

An XML User Interface (XMLUI) dialog box can be used by several Flash MX 2004 features, such as Commands and Behaviors, to provide a user interface for features that you build using extensibility.

The XMLUI object provides the ability to get and set properties of an XMLUI dialog box, and accept or cancel out of one. The XMLUI methods can be used in callbacks, such as `oncommand` handlers in buttons.

Methods

You can use the following methods with the XMLUI object:

```
xmlui.accept()
xmlui.cancel()
xmlui.get()
xmlui.set()
```

xmlui.accept()

Description

Makes the current XMLUI dialog box exit with an accept state, which is equivalent to the user clicking the OK button.

Usage

```
xmlui.accept()
```

Arguments

None.

Returns

Nothing.

Example

```
fl.xmlui.accept()
```

xmlui.cancel()

Description

Makes the current XMLUI dialog box exit with a cancel state, which is equivalent to the user clicking the Cancel button.

Usage

```
xmlui.cancel()
```

Arguments

None.

Returns

Nothing.

Example

```
fl.xmlui.cancel();
```

xmlui.get()

Description

Retrieves the value of the specified property of the current XMLUI dialog box.

Usage

```
xmlui.get(name)
```

Arguments

The *name* argument is a String that specifies the name of the XMLUI property to retrieve.

Returns

A string value for the specified property. In cases where you might expect a Boolean value of `true` or `false`, returns the string `"true"` or `"false"`.

Example

```
fl.xmlui.get("propertyDefinedInXML");
```

xmlui.set()

Description

Modifies the value of the specified property of the current XMLUI dialog box.

Usage

```
xmlui.set()
```

Arguments

The *name* argument is a String that specifies the name of XMLUI property to modify.

The *value* argument is a String that specifies the value to which you want to set the XMLUI property.

Returns

Nothing.

Example

```
fl.xmlui.set("propertyDefinedInXML","value");
```

Properties

There are no unique properties for the XMLUI object.