Client-Side JavaScript Reference

Version 1.3

Recycled and Recyclable Paper

Version 1.3

©1999 Netscape Communications Corporation. All Rights Reserved

Printed in the United States of America.   00 99 98   5 4 3 2 1

Netscape Communications Corporation, 501 East Middlefield Road, Mountain View, CA 94043

New Features in this Release

JavaScript version 1.3 provides the following new features and enhancements:

- **ECMA compliance.** JavaScript 1.3 is fully compatible with ECMA-262. See the *Client-Side JavaScript Guide* for details.
- **Unicode support.** The Unicode character set can be used for all known encoding, and you can use the Unicode escape sequence in string literals. See `escape` and `unescape`. See the *Client-Side JavaScript Guide* for details.
- **Changes to the Array object.**
    - When you specify a single numeric parameter with the `Array` constructor, you specify the initial length of the array.
    - The `push` method returns the new length of the array rather than the last element added to the array.
    - The `splice` method always returns an array containing the removed elements, even if only one element is removed.
    - The `toString` method joins an array and returns a string containing each array element separated by commas, rather than returning a string representing the source code of the array.
    - The `length` property contains an unsigned, 32-bit integer with a value less than 2^{32}.

- **Changes to the Date object.**
  - Removed platform dependencies to provide a uniform behavior across platforms.
  - Changed the range for dates to -100,000,000 days to 100,000,000 days relative to 01 January, 1970 UTC.
  - Added a milliseconds parameter to the `Date` constructor.
  - Added the `getFullYear`, `setFullYear`, `getMilliseconds`, and `setMilliseconds` methods.
  - Added the `getUTCDate`, `getUTCDay`, `getUTCFullYear`, `getUTCHours`, `getUTCMilliseconds`, `getUTCMinutes`, `getUTCMonth`, `getUTCSeconds`, `setUTCDate`, `setUTCFullYear`, `setUTCHours`, `setUTCMilliseconds`, `setUTCMinutes`, `setUTCMonth`, `setUTCSeconds`, and `toUTCString` methods.
  - Added a day parameter to the `setMonth` method.
  - Added minutes, seconds, and milliseconds parameters to the `setHours` method.
  - Added seconds and milliseconds parameters to the `setMinutes` method.
  - Added a milliseconds parameter to the `setSeconds` method.
  - Added a milliseconds parameter to the `UTC` method.
  - Deprecated the `getYear`, `setYear`, and `toGMTString` methods.
- **Changes to the Function object.**
  - Added the `apply` method, which allows you to apply a method of another object in the context of a different object (the calling object).
  - Added the `call` method, which allows you to call (execute) a method of another object in the context of a different object (the calling object).
  - Deprecated the `arguments.caller` property.

- **Changes to the String object.**
    - The `charCodeAt` and `fromCharCode` methods use Unicode values rather than ISO-Latin-1 values.
    - The `replace` method supports the nesting of a function in place of the second argument.
- **New method toSource.** The `toSource` method returns a string representing the source code of the object. See `Array.toSource`, `Boolean.toSource`, `Date.toSource`, `Function.toSource`, `Number.toSource`, `Object.toSource`, `RegExp.toSource`, and `String.toSource`.
- **New top-level properties Infinity, NaN, and undefined.** `Infinity` is a numeric value representing infinity. `NaN` is a value representing Not-A-Number. `undefined` is the value undefined.
- **New top-level function isFinite.** `isFinite` evaluates an argument to determine whether it is a finite number.
- **Changes to the top-level eval function.** You should not indirectly use the `eval` function by invoking it via a name other than `eval`.
- **New strict equality operators === and !==.** The `===` (strict equal) operator returns true if the operands are equal and of the same type. The `!==` (strict not equal) operator returns true if the operands are not equal and/or not of the same type. See "Comparison Operators" on page 635 and "Using the Equality Operators" on page 637.
- **Changes to the equality operators == and !=.** The use of the `==` (equal) and `!=` (not equal) operators reverts to the JavaScript 1.1 implementation. If the two operands are not of the same type, JavaScript attempts to convert the operands to an appropriate type for the comparison. See "Using the Equality Operators" on page 637.

- **Changes to the behavior of conditional tests.**
  - You should not use simple assignments in a conditional statement; for example, do not specify the condition `if(x = y)`. Previous JavaScript versions converted `if(x = y)` to `if(x == y)`, but 1.3 generates a runtime error. See "if...else" on page 623.
  - Any object whose value is not `undefined` or `null`, including a Boolean object whose value is false, evaluates to true when passed to a conditional statement. See "Boolean" on page 51.
- **The JavaScript console.** The JavaScript console is a window that can display all JavaScript error messages. Then, when a JavaScript error occurs, the error message is directed to the JavaScript console and no dialog box appears. See the *Client-Side JavaScript Guide* for details.

Contents

About this Book

JavaScript is Netscape's cross-platform, object-based scripting language for client and server applications. This book is a reference manual for the JavaScript language, including both core and client-side JavaScript.

This preface contains the following sections:

- New Features in this Release
- What You Should Already Know
- JavaScript Versions
- Where to Find JavaScript Information
- Document Conventions

New Features in this Release

For a summary of JavaScript 1.3 features, see "New Features in this Release" on page 3. Information on these features has been incorporated in this manual.

What You Should Already Know

This book assumes you have the following basic background:

- A general understanding of the Internet and the World Wide Web (WWW).
- Good working knowledge of HyperText Markup Language (HTML).

Some programming experience with a language such as C or Visual Basic is useful, but not required.

JavaScript Versions

Each version of Navigator supports a different version of JavaScript. To help you write scripts that are compatible with multiple versions of Navigator, this manual lists the JavaScript version in which each feature was implemented.

The following table lists the JavaScript version supported by different Navigator versions. Versions of Navigator prior to 2.0 do not support JavaScript.

Table 1 JavaScript and Navigator versions

JavaScript version	Navigator version
JavaScript 1.0	Navigator 2.0
JavaScript 1.1	Navigator 3.0
JavaScript 1.2	Navigator 4.0–4.05
JavaScript 1.3	Navigator 4.06–4.5

Each version of the Netscape Enterprise Server also supports a different version of JavaScript. To help you write scripts that are compatible with multiple versions of the Enterprise Server, this manual uses an abbreviation to indicate the server version in which each feature was implemented.

Table 2 JavaScript and Netscape Enterprise Server versions

Abbreviation	Enterpriser Server version
NES 2.0	Netscape Enterprise Server 2.0
NES 3.0	Netscape Enterprise Server 3.0

Where to Find JavaScript Information

The client-side JavaScript documentation includes the following books:

- The *Client-Side JavaScript Guide* provides information about the JavaScript language and its objects. This book contains information for both core and client-side JavaScript.
- The *Client-Side JavaScript Reference* (this book) provides reference material for the JavaScript language, including both core and client-side JavaScript.

If you are new to JavaScript, start with the *Client-Side JavaScript Guide*. Once you have a firm grasp of the fundamentals, you can use the *Client-Side JavaScript Reference* to get more details on individual objects and statements.

If you are developing a client-server JavaScript application, use the material in the client-side books to familiarize yourself with core and client-side JavaScript. Then, use the *Server-Side JavaScript Guide* and *Server-Side JavaScript Reference* for help developing a server-side JavaScript application.

DevEdge, Netscape's online developer resource, contains information that can be useful when you're working with JavaScript. The following URLs are of particular interest:

- `http://developer.netscape.com/docs/manuals/javascript.html`

  The JavaScript page of the DevEdge library contains documents of interest about JavaScript. This page changes frequently. You should visit it periodically to get the newest information.

- `http://developer.netscape.com/docs/manuals/`

  The DevEdge library contains documentation on many Netscape products and technologies.

- `http://developer.netscape.com`

  The DevEdge home page gives you access to all DevEdge resources.

Document Conventions

Occasionally this book tells you where to find things in the user interface of Navigator. In these cases, the book describes the user interface in Navigator 4.5. The interface may be different in earlier versions of the browser.

JavaScript applications run on many operating systems; the information in this book applies to all versions. File and directory paths are given in Windows format (with backslashes separating directory names). For Unix versions, the directory paths are the same, except that you use slashes instead of backslashes to separate directories.

This book uses uniform resource locators (URLs) of the following form:

`http://`*`server.domain/path/file`*`.html`

In these URLs, *server* represents the name of the server on which you run your application, such as `research1` or `www`; *domain* represents your Internet domain name, such as `netscape.com` or `uiuc.edu`; *path* represents the directory structure on the server; and *file*`.html` represents an individual file name. In general, items in italics in URLs are placeholders and items in normal monospace font are literals. If your server has Secure Sockets Layer (SSL) enabled, you would use `https` instead of `http` in the URL.

This book uses the following font conventions:

- `The monospace font` is used for sample code and code listings, API and language elements (such as method names and property names), file names, path names, directory names, HTML tags, and any text that must be typed on the screen. (*`Monospace italic font`* is used for placeholders embedded in code.)
- *Italic type* is used for book titles, emphasis, variables and placeholders, and words used in the literal sense.
- **Boldface type** is used for glossary terms.

1 Object Reference

- **Objects, Methods, and Properties**
- **Top-Level Properties and Functions**
- **Event Handlers**

Chapter

1

Objects, Methods, and Properties

This chapter documents all the JavaScript objects, along with their methods and properties. It is an alphabetical reference for the main features of JavaScript.

The reference is organized as follows:

- Full entries for each object appear in alphabetical order; properties and functions not associated with any object appear in Chapter 2, "Top-Level Properties and Functions."

  Each entry provides a complete description for an object. Tables included in the description of each object summarize the object's methods and properties.

- Full entries for an object's methods and properties appear in alphabetical order after the object's entry.

  These entries provide a complete description for each method or property, and include cross-references to related features in the documentation.

Anchor

A place in a document that is the target of a hypertext link.

Client-side object

Implemented in	JavaScript 1.0
	JavaScript 1.2: added `name`, `text`, `x`, and `y` properties

Created by Using the HTML `A` tag or calling the `String.anchor` method. The JavaScript runtime engine creates an `Anchor` object corresponding to each `A` tag in your document that supplies the `NAME` attribute. It puts these objects in an array in the `document.anchors` property. You access an `Anchor` object by indexing this array.

To define an anchor with the `String.anchor` method:

```
theString.anchor(nameAttribute)
```

where:

`theString`	A `String` object.
`nameAttribute`	A string.

To define an anchor with the `A` tag, use standard HTML syntax. If you specify the `NAME` attribute, you can use the value of that attribute to index into the `anchors` array.

Description If an `Anchor` object is also a `Link` object, the object has entries in both the `anchors` and `links` arrays.

Property Summary

Property	Description
`name`	A string specifying the anchor's name.
`text`	A string specifying the text of an anchor.
`x`	The horizontal position of the anchor's left edge, in pixels, relative to the left edge of the document.
`y`	The vertical position of the anchor's top edge, in pixels, relative to the top edge of the document.

Method Summary This object inherits the `watch` and `unwatch` methods from `Object`.

Examples

Example 1: An anchor. The following example defines an anchor for the text "Welcome to JavaScript":

```
<A NAME="javascript_intro"><H2>Welcome to JavaScript</H2></A>
```

If the preceding anchor is in a file called `intro.html`, a link in another file could define a jump to the anchor as follows:

```
<A HREF="intro.html#javascript_intro">Introduction</A>
```

Example 2: anchors array. The following example opens two windows. The first window contains a series of buttons that set `location.hash` in the second window to a specific anchor. The second window defines four anchors named "0," "1," "2," and "3." (The anchor names in the document are therefore 0, 1, 2, ... (document.anchors.length-1).) When a button is pressed in the first window, the `onClick` event handler verifies that the anchor exists before setting `window2.location.hash` to the specified anchor name.

`link1.html`, which defines the first window and its buttons, contains the following code:

```
<HTML>
<HEAD>
<TITLE>Links and Anchors: Window 1</TITLE>
</HEAD>
<BODY>
<SCRIPT>
window2=open("link2.html","secondLinkWindow",
   "scrollbars=yes,width=250, height=400")
function linkToWindow(num) {
   if (window2.document.anchors.length > num)
      window2.location.hash=num
   else
      alert("Anchor does not exist!")
}
</SCRIPT>
```

```
<B>Links and Anchors</B>
<FORM>
<P>Click a button to display that anchor in window #2
<P><INPUT TYPE="button" VALUE="0" NAME="link0_button"
   onClick="linkToWindow(this.value)">
<INPUT TYPE="button" VALUE="1" NAME="link0_button"
   onClick="linkToWindow(this.value)">
<INPUT TYPE="button" VALUE="2" NAME="link0_button"
   onClick="linkToWindow(this.value)">
<INPUT TYPE="button" VALUE="3" NAME="link0_button"
   onClick="linkToWindow(this.value)">
<INPUT TYPE="button" VALUE="4" NAME="link0_button"
   onClick="linkToWindow(this.value)">
</FORM>
</BODY>
</HTML>
```

`link2.html`, which contains the anchors, contains the following code:

```
<HTML>
<HEAD>
<TITLE>Links and Anchors: Window 2</TITLE>
</HEAD>
<BODY>
<A NAME="0"><B>Some numbers</B> (Anchor 0)</A>
<UL><LI>one
<LI>two
<LI>three
<LI>four</UL>
<P><A NAME="1"><B>Some colors</B> (Anchor 1)</A>
<UL><LI>red
<LI>orange
<LI>yellow
<LI>green</UL>
<P><A NAME="2"><B>Some music types</B> (Anchor 2)</A>
<UL><LI>R&B
<LI>Jazz
<LI>Soul
<LI>Reggae
<LI>Rock</UL>
<P><A NAME="3"><B>Some countries</B> (Anchor 3)</A>
<UL><LI>Afghanistan
<LI>Brazil
<LI>Canada
<LI>Finland
<LI>India</UL>
</BODY>
</HTML>
```

See also `Link`

name

A string specifying the anchor's name.

Property of	`Anchor`
Read-only	
Implemented in	JavaScript 1.2

Description The `name` property reflects the value of the `NAME` attribute.

Examples The following example displays the name of the first anchor in a document:

```
alert("The first anchor is " + document.anchors[0].name)
```

text

A string specifying the text of an anchor.

Property of	`Anchor`
Read-only	
Implemented in	JavaScript 1.2

Description The `text` property specifies the string that appears within the `A` tag.

Examples The following example displays the text of the first anchor in a document:

```
alert("The text of the first anchor is " + document.anchors[0].text)
```

x

The horizontal position of the anchor's left edge, in pixels, relative to the left edge of the document.

Property of	`Anchor`
Read-only	
Implemented in	JavaScript 1.2

See also `Anchor.y`

y

The vertical position of the anchor's top edge, in pixels, relative to the top edge of the document.

Property of	`Anchor`
Read-only	
Implemented in	JavaScript 1.2

See also `Anchor.x`

Applet

Includes a Java applet in a web page.
Client-side object

Implemented in	JavaScript 1.1

Created by The HTML `APPLET` tag. The JavaScript runtime engine creates an `Applet` object corresponding to each applet in your document. It puts these objects in an array in the `document.applets` property. You access an `Applet` object by indexing this array.

To define an applet, use standard HTML syntax. If you specify the `NAME` attribute, you can use the value of that attribute to index into the `applets` array. To refer to an applet in JavaScript, you must supply the `MAYSCRIPT` attribute in its definition.

Description The author of an HTML page must permit an applet to access JavaScript by specifying the `MAYSCRIPT` attribute of the `APPLET` tag. This prevents an applet from accessing JavaScript on a page without the knowledge of the page author. For example, to allow the `musicPicker.class` applet access to JavaScript on your page, specify the following:

```
<APPLET CODE="musicPicker.class" WIDTH=200 HEIGHT=35
   NAME="musicApp" MAYSCRIPT>
```

Accessing JavaScript when the `MAYSCRIPT` attribute is not specified results in an exception.

For more information on using applets, see the LiveConnect information in the *Client-Side JavaScript Guide*.

Property Summary The `Applet` object inherits all public properties of the Java applet.

Method Summary The `Applet` object inherits all public methods of the Java applet.

Examples The following code launches an applet called `musicApp`:

```
<APPLET CODE="musicSelect.class" WIDTH=200 HEIGHT=35
   NAME="musicApp" MAYSCRIPT>
</APPLET>
```

For more examples, see the LiveConnect information in the *Client-Side JavaScript Guide*.

See also `MimeType`, `Plugin`

Area

Defines an area of an image as an image map. When the user clicks the area, the area's hypertext reference is loaded into its target window. `Area` objects are a type of `Link` object.

Client-side object

Implemented in	JavaScript 1.1

For information on `Area` objects, see `Link`.

Array

Lets you work with arrays.

Core object

Implemented in	JavaScript 1.1, NES 2.0 JavaScript 1.3: added `toSource` method; changed `length` property; changed `push` and `splice` methods.
ECMA version	ECMA-262

Created by The `Array` object constructor:

```
new Array(arrayLength)
new Array(element0, element1, ..., elementN)
```

An array literal:

```
[element0, element1, ..., elementN]
```

JavaScript 1.2 when you specify `LANGUAGE="JavaScript1.2"` *in the* `<SCRIPT>` *tag:*

```
new Array(element0, element1, ..., elementN)
```

JavaScript 1.2 when you do not specify `LANGUAGE="JavaScript1.2"` *in the* `<SCRIPT>` *tag:*

```
new Array([arrayLength])
new Array([element0[, element1[, ..., elementN]]])
```

JavaScript 1.1:

```
new Array([arrayLength])
new Array([element0[, element1[, ..., elementN]]])
```

Parameters

`arrayLength`	The initial length of the array. You can access this value using the `length` property. If the value specified is not a number, an array of length 1 is created, with the first element having the specified value. The maximum length allowed for an array is 4,294,967,295.
`element`*`N`*	A list of values for the array's elements. When this form is specified, the array is initialized with the specified values as its elements, and the array's `length` property is set to the number of arguments.

Description An array is an ordered set of values associated with a single variable name.

The following example creates an `Array` object with an array literal; the `coffees` array contains three elements and a length of three:

```
coffees = ["French Roast", "Columbian", "Kona"]
```

Indexing an array. You index an array by its ordinal number. For example, assume you define the following array:

```
myArray = new Array("Wind","Rain","Fire")
```

You then refer to the first element of the array as `myArray[0]` and the second element of the array as `myArray[1]`.

Specifying a single parameter. When you specify a single numeric parameter with the `Array` constructor, you specify the initial length of the array. The following code creates an array of five elements:

```
billingMethod = new Array(5)
```

The behavior of the `Array` constructor depends on whether the single parameter is a number.

- If the value specified is a number, the constructor converts the number to an unsigned, 32-bit integer and generates an array with the `length` property (size of the array) set to the integer. The array initially contains no elements, even though it might have a non-zero length.
- If the value specified is not a number, an array of length 1 is created, with the first element having the specified value.

The following code creates an array of length 25, then assigns values to the first three elements:

```
musicTypes = new Array(25)
musicTypes[0] = "R&B"
musicTypes[1] = "Blues"
musicTypes[2] = "Jazz"
```

You can construct a *dense* array of two or more elements starting with index 0 if you define initial values for all elements. A dense array is one in which each element has a value. The following code creates a dense array with three elements:

```
myArray = new Array("Hello", myVar, 3.14159)
```

Increasing the array length indirectly. An array's length increases if you assign a value to an element higher than the current length of the array. The following code creates an array of length 0, then assigns a value to element 99. This changes the length of the array to 100.

```
colors = new Array()
colors[99] = "midnightblue"
```

Creating an array using the result of a match. The result of a match between a regular expression and a string can create an array. This array has properties and elements that provide information about the match. An array is the return value of `RegExp.exec`, `String.match`, and `String.replace`. To help explain these properties and elements, look at the following example and then refer to the table below:

```
<SCRIPT LANGUAGE="JavaScript1.2">
//Match one d followed by one or more b's followed by one d
//Remember matched b's and the following d
//Ignore case

myRe=/d(b+)(d)/i;
myArray = myRe.exec("cdbBdbsbz");

</SCRIPT>
```

The properties and elements returned from this match are as follows:

Property/Element	Description	Example
`input`	A read-only property that reflects the original string against which the regular expression was matched.	cdbBdbsbz
`index`	A read-only property that is the zero-based index of the match in the string.	1
`[0]`	A read-only element that specifies the last matched characters.	dbBd
`[1], ...[n]`	Read-only elements that specify the parenthesized substring matches, if included in the regular expression. The number of possible parenthesized substrings is unlimited.	[1]=bB [2]=d

Backward Compatibility

JavaScript 1.2. When you specify a single parameter with the `Array` constructor, the behavior depends on whether you specify `LANGUAGE="JavaScript1.2"` in the `<SCRIPT>` tag:

- If you specify `LANGUAGE="JavaScript1.2"` in the `<SCRIPT>` tag, a single-element array is returned. For example, `new Array(5)` creates a one-element array with the first element being 5. A constructor with a single parameter acts in the same way as a multiple parameter constructor. You cannot specify the `length` property of an `Array` using a constructor with one parameter.

- If you do not specify `LANGUAGE="JavaScript1.2"` in the `<SCRIPT>` tag, you specify the initial length of the array as with other JavaScript versions.

JavaScript 1.1 and earlier. When you specify a single parameter with the `Array` constructor, you specify the initial length of the array. The following code creates an array of five elements:

```
billingMethod = new Array(5)
```

JavaScript 1.0. You must index an array by its ordinal number; for example `myArray[0]`.

Property Summary

Property	Description
`constructor`	Specifies the function that creates an object's prototype.
`index`	For an array created by a regular expression match, the zero-based index of the match in the string.
`input`	For an array created by a regular expression match, reflects the original string against which the regular expression was matched.
`length`	Reflects the number of elements in an array
`prototype`	Allows the addition of properties to all objects.

Method Summary

Method	Description
`concat`	Joins two arrays and returns a new array.
`join`	Joins all elements of an array into a string.
`pop`	Removes the last element from an array and returns that element.
`push`	Adds one or more elements to the end of an array and returns the new length of the array.
`reverse`	Transposes the elements of an array: the first array element becomes the last and the last becomes the first.
`shift`	Removes the first element from an array and returns that element
`slice`	Extracts a section of an array and returns a new array.
`splice`	Adds and/or removes elements from an array.
`sort`	Sorts the elements of an array.
`toSource`	Returns an array literal representing the specified array; you can use this value to create a new array. Overrides the `Object.toSource` method.
`toString`	Returns a string representing the array and its elements. Overrides the `Object.toString` method.
`unshift`	Adds one or more elements to the front of an array and returns the new length of the array.
`valueOf`	Returns the primitive value of the array. Overrides the `Object.valueOf` method.

In addition, this object inherits the `watch` and `unwatch` methods from `Object`.

Examples **Example 1.** The following example creates an array, `msgArray`, with a length of 0, then assigns values to `msgArray[0]` and `msgArray[99]`, changing the length of the array to 100.

```
msgArray = new Array()
msgArray[0] = "Hello"
msgArray[99] = "world"
// The following statement is true,
// because defined msgArray[99] element.
if (msgArray.length == 100)
   myVar="The length is 100."
```

See also the examples for `onError`.

Example 2: Two-dimensional array. The following code creates a two-dimensional array and assigns the results to `myVar`.

```
myVar="Multidimensional array test; "
a = new Array(4)
for (i=0; i < 4; i++) {
   a[i] = new Array(4)
   for (j=0; j < 4; j++) {
      a[i][j] = "["+i+","+j+"]"
   }
}
for (i=0; i < 4; i++) {
   str = "Row "+i+":"
   for (j=0; j < 4; j++) {
      str += a[i][j]
   }
   myVar += str +"; "
}
```

This example assigns the following string to `myVar` (line breaks are used here for readability):

```
Multidimensional array test;
Row 0:[0,0][0,1][0,2][0,3];
Row 1:[1,0][1,1][1,2][1,3];
Row 2:[2,0][2,1][2,2][2,3];
Row 3:[3,0][3,1][3,2][3,3];
```

See also `Image`

concat

Joins two arrays and returns a new array.

Method of	`Array`
Implemented in	JavaScript 1.2, NES 3.0

Syntax `concat(`*`arrayName2`*`, `*`arrayName3`*`, ..., `*`arrayNameN`*`)`

Parameters

`arrayName2...` `arrayName`*`N`*	Arrays to concatenate to this array.

Description `concat` does not alter the original arrays, but returns a "one level deep" copy that contains copies of the same elements combined from the original arrays. Elements of the original arrays are copied into the new array as follows:

- Object references (and not the actual object): `concat` copies object references into the new array. Both the original and new array refer to the same object. If a referenced object changes, the changes are visible to both the new and original arrays.
- Strings and numbers (not `String` and `Number` objects): `concat` copies strings and numbers into the new array. Changes to the string or number in one array does not affect the other arrays.

If a new element is added to either array, the other array is not affected.

The following code concatenates two arrays:

```
alpha=new Array("a","b","c")
numeric=new Array(1,2,3)
alphaNumeric=alpha.concat(numeric) // creates array ["a","b","c",1,2,3]
```

The following code concatenates three arrays:

```
num1=[1,2,3]
num2=[4,5,6]
num3=[7,8,9]
nums=num1.concat(num2,num3) // creates array [1,2,3,4,5,6,7,8,9]
```

constructor

Specifies the function that creates an object's prototype. Note that the value of this property is a reference to the function itself, not a string containing the function's name.

Property of	`Array`
Implemented in	JavaScript 1.1, NES 2.0
ECMA version	ECMA-262

Description See `Object.constructor`.

index

For an array created by a regular expression match, the zero-based index of the match in the string.

Property of	`Array`
Static	
Implemented in	JavaScript 1.2, NES 3.0

input

For an array created by a regular expression match, reflects the original string against which the regular expression was matched.

Property of	`Array`
Static	
Implemented in	JavaScript 1.2, NES 3.0

join

Joins all elements of an array into a string.

Method of	`Array`
Implemented in	JavaScript 1.1, NES 2.0
ECMA version	ECMA-262

Syntax `join(`*`separator`*`)`

Parameters

`separator`	Specifies a string to separate each element of the array. The separator is converted to a string if necessary. If omitted, the array elements are separated with a comma.

Description The string conversions of all array elements are joined into one string.

Examples The following example creates an array, `a`, with three elements, then joins the array three times: using the default separator, then a comma and a space, and then a plus.

```
a = new Array("Wind","Rain","Fire")
myVar1=a.join()      // assigns "Wind,Rain,Fire" to myVar1
myVar2=a.join(", ")  // assigns "Wind, Rain, Fire" to myVar1
myVar3=a.join(" + ") // assigns "Wind + Rain + Fire" to myVar1
```

See also `Array.reverse`

length

An unsigned, 32-bit integer that specifies the number of elements in an array.

Property of	`Array`
Implemented in	JavaScript 1.1, NES 2.0 JavaScript 1.3: `length` is an unsigned, 32-bit integer with a value less than 2^{32}.
ECMA version	ECMA-262

Description The value of the `length` property is an integer with a positive sign and a value less than 2 to the 32 power (2^{32}).

You can set the `length` property to truncate an array at any time. When you extend an array by changing its `length` property, the number of actual elements does not increase; for example, if you set `length` to 3 when it is currently 2, the array still contains only 2 elements.

Examples In the following example, the `getChoice` function uses the `length` property to iterate over every element in the `musicType` array. `musicType` is a select element on the `musicForm` form.

```
function getChoice() {
   for (var i = 0; i < document.musicForm.musicType.length; i++) {
      if (document.musicForm.musicType.options[i].selected == true) {
         return document.musicForm.musicType.options[i].text
      }
   }
}
```

The following example shortens the array `statesUS` to a length of 50 if the current length is greater than 50.

```
if (statesUS.length > 50) {
   statesUS.length=50
}
```

pop

Removes the last element from an array and returns that element. This method changes the length of the array.

Method of	`Array`
Implemented in	JavaScript 1.2, NES 3.0

Syntax `pop()`

Parameters None.

Example The following code creates the `myFish` array containing four elements, then removes its last element.

```
myFish = ["angel", "clown", "mandarin", "surgeon"];
popped = myFish.pop();
```

See also `push`, `shift`, `unshift`

prototype

Represents the prototype for this class. You can use the prototype to add properties or methods to all instances of a class. For information on prototypes, see `Function.prototype`.

Property of	`Array`
Implemented in	JavaScript 1.1, NES 2.0
ECMA version	ECMA-262

push

Adds one or more elements to the end of an array and returns the new length of the array. This method changes the length of the array.

Method of	`Array`
Implemented in	JavaScript 1.2, NES 3.0 JavaScript 1.3: `push` returns the new length of the array rather than the last element added to the array.

Syntax `push(`*`element1`*`, ..., `*`elementN`*`)`

Parameters

`element1, ..., elementN`	The elements to add to the end of the array.

Description The behavior of the `push` method is analogous to the `push` function in Perl 4. Note that this behavior is different in Perl 5.

Backward Compatibility **JavaScript 1.2.** The `push` method returns the last element added to an array.

Example The following code creates the `myFish` array containing two elements, then adds two elements to it. After the code executes, `pushed` contains 4. (In JavaScript 1.2, `pushed` contains "lion" after the code executes.)

```
myFish = ["angel", "clown"];
pushed = myFish.push("drum", "lion");
```

See also `pop`, `shift`, `unshift`

reverse

Transposes the elements of an array: the first array element becomes the last and the last becomes the first.

Method of	`Array`
Implemented in	JavaScript 1.1, NES 2.0
ECMA version	ECMA-262

Syntax `reverse()`

Parameters None

Description The `reverse` method transposes the elements of the calling array object.

Examples The following example creates an array `myArray`, containing three elements, then reverses the array.

```
myArray = new Array("one", "two", "three")
myArray.reverse()
```

This code changes `myArray` so that:

- `myArray[0]` is `“three”`
- `myArray[1]` is `“two”`
- `myArray[2]` is `“one”`

See also `Array.join`, `Array.sort`

shift

Removes the first element from an array and returns that element. This method changes the length of the array.

Method of	`Array`
Implemented in	JavaScript 1.2, NES 3.0

Syntax `shift()`

Parameters None.

Example The following code displays the `myFish` array before and after removing its first element. It also displays the removed element:

```
myFish = ["angel", "clown", "mandarin", "surgeon"];
document.writeln("myFish before: " + myFish);
shifted = myFish.shift();
document.writeln("myFish after: " + myFish);
document.writeln("Removed this element: " + shifted);
```

This example displays the following:

```
myFish before: ["angel", "clown", "mandarin", "surgeon"]
myFish after: ["clown", "mandarin", "surgeon"]
Removed this element: angel
```

See also `pop`, `push`, `unshift`

slice

Extracts a section of an array and returns a new array.

Method of	`Array`
Implemented in	JavaScript 1.2, NES 3.0

Syntax `slice(`*`begin`*`[,`*`end`*`])`

Parameters

`begin` Zero-based index at which to begin extraction.

`end` Zero-based index at which to end extraction:

- `slice` extracts up to but not including `end`. `slice(1,4)` extracts the second element through the fourth element (elements indexed 1, 2, and 3)
- As a negative index, `end` indicates an offset from the end of the sequence. `slice(2,-1)` extracts the third element through the second to last element in the sequence.
- If `end` is omitted, `slice` extracts to the end of the sequence.

Description `slice` does not alter the original array, but returns a new "one level deep" copy that contains copies of the elements sliced from the original array. Elements of the original array are copied into the new array as follows:

- For object references (and not the actual object), `slice` copies object references into the new array. Both the original and new array refer to the same object. If a referenced object changes, the changes are visible to both the new and original arrays.
- For strings and numbers (not `String` and `Number` objects), `slice` copies strings and numbers into the new array. Changes to the string or number in one array does not affect the other array.

If a new element is added to either array, the other array is not affected.

Example In the following example, `slice` creates a new array, `newCar`, from `myCar`. Both include a reference to the object `myHonda`. When the color of `myHonda` is changed to `purple`, both arrays reflect the change.

```
<SCRIPT LANGUAGE="JavaScript1.2">

//Using slice, create newCar from myCar.
myHonda = {color:"red",wheels:4,engine:{cylinders:4,size:2.2}}
myCar = [myHonda, 2, "cherry condition", "purchased 1997"]
newCar = myCar.slice(0,2)

//Write the values of myCar, newCar, and the color of myHonda
// referenced from both arrays.
document.write("myCar = " + myCar + "<BR>")
document.write("newCar = " + newCar + "<BR>")
document.write("myCar[0].color = " + myCar[0].color + "<BR>")
document.write("newCar[0].color = " + newCar[0].color + "<BR><BR>")

//Change the color of myHonda.
myHonda.color = "purple"
document.write("The new color of my Honda is " + myHonda.color +
"<BR><BR>")

//Write the color of myHonda referenced from both arrays.
document.write("myCar[0].color = " + myCar[0].color + "<BR>")
document.write("newCar[0].color = " + newCar[0].color + "<BR>")

</SCRIPT>
```

This script writes:

```
myCar = [{color:"red", wheels:4, engine:{cylinders:4, size:2.2}}, 2,
   "cherry condition", "purchased 1997"]
newCar = [{color:"red", wheels:4, engine:{cylinders:4, size:2.2}}, 2]
myCar[0].color = red newCar[0].color = red
The new color of my Honda is purple
myCar[0].color = purple
newCar[0].color = purple
```

sort

Sorts the elements of an array.

Method of	`Array`
Implemented in	JavaScript 1.1, NES 2.0 JavaScript 1.2: modified behavior.
ECMA version	ECMA-262

Syntax `sort(`*`compareFunction`*`)`

Parameters

`compareFunction`	Specifies a function that defines the sort order. If omitted, the array is sorted lexicographically (in dictionary order) according to the string conversion of each element.

Description If `compareFunction` is not supplied, elements are sorted by converting them to strings and comparing strings in lexicographic ("dictionary" or "telephone book," *not* numerical) order. For example, "80" comes before "9" in lexicographic order, but in a numeric sort 9 comes before 80.

If `compareFunction` is supplied, the array elements are sorted according to the return value of the compare function. If a and b are two elements being compared, then:

- If `compareFunction(a, b)` is less than 0, sort `b` to a lower index than `a`.
- If `compareFunction(a, b)` returns 0, leave `a` and `b` unchanged with respect to each other, but sorted with respect to all different elements.
- If `compareFunction(a, b)` is greater than 0, sort `b` to a higher index than `a`.

So, the compare function has the following form:

```
function compare(a, b) {
   if (a is less than b by some ordering criterion)
      return -1
   if (a is greater than b by the ordering criterion)
      return 1
   // a must be equal to b
   return 0
}
```

To compare numbers instead of strings, the compare function can simply subtract b from a:

```
function compareNumbers(a, b) {
   return a - b
}
```

JavaScript uses a stable sort: the index partial order of a and b does not change if a and b are equal. If a's index was less than b's before sorting, it will be after sorting, no matter how a and b move due to sorting.

The behavior of the `sort` method changed between JavaScript 1.1 and JavaScript 1.2.

In JavaScript 1.1, on some platforms, the sort method does not work. This method works on all platforms for JavaScript 1.2.

In JavaScript 1.2, this method no longer converts undefined elements to null; instead it sorts them to the high end of the array. For example, assume you have this script:

```
<SCRIPT>
a = new Array();
a[0] = "Ant";
a[5] = "Zebra";

function writeArray(x) {
   for (i = 0; i < x.length; i++) {
      document.write(x[i]);
      if (i < x.length-1) document.write(", ");
   }
}

writeArray(a);
a.sort();
document.write("<BR><BR>");
writeArray(a);
</SCRIPT>
```

In JavaScript 1.1, JavaScript prints:

```
ant, null, null, null, null, zebra
ant, null, null, null, null, zebra
```

In JavaScript 1.2, JavaScript prints:

```
ant, undefined, undefined, undefined, undefined, zebra
ant, zebra, undefined, undefined, undefined, undefined
```

Examples The following example creates four arrays and displays the original array, then the sorted arrays. The numeric arrays are sorted without, then with, a compare function.

```
<SCRIPT>
stringArray = new Array("Blue","Humpback","Beluga")
numericStringArray = new Array("80","9","700")
numberArray = new Array(40,1,5,200)
mixedNumericArray = new Array("80","9","700",40,1,5,200)

function compareNumbers(a, b) {
   return a - b
}

document.write("<B>stringArray:</B> " + stringArray.join() +"<BR>")
document.write("<B>Sorted:</B> " + stringArray.sort() +"<P>")

document.write("<B>numberArray:</B> " + numberArray.join() +"<BR>")
document.write("<B>Sorted without a compare function:</B> " + numberArray.sort() +"<BR>")
document.write("<B>Sorted with compareNumbers:</B> " + numberArray.sort(compareNumbers)
+"<P>")

document.write("<B>numericStringArray:</B> " + numericStringArray.join() +"<BR>")
document.write("<B>Sorted without a compare function:</B> " + numericStringArray.sort()
+"<BR>")
document.write("<B>Sorted with compareNumbers:</B> " +
numericStringArray.sort(compareNumbers) +"<P>")

document.write("<B>mixedNumericArray:</B> " + mixedNumericArray.join() +"<BR>")
document.write("<B>Sorted without a compare function:</B> " + mixedNumericArray.sort()
+"<BR>")
document.write("<B>Sorted with compareNumbers:</B> " +
mixedNumericArray.sort(compareNumbers) +"<BR>")
</SCRIPT>
```

This example produces the following output. As the output shows, when a compare function is used, numbers sort correctly whether they are numbers or numeric strings.

```
stringArray: Blue,Humpback,Beluga
Sorted: Beluga,Blue,Humpback

numberArray: 40,1,5,200
Sorted without a compare function: 1,200,40,5
Sorted with compareNumbers: 1,5,40,200

numericStringArray: 80,9,700
Sorted without a compare function: 700,80,9
Sorted with compareNumbers: 9,80,700

mixedNumericArray: 80,9,700,40,1,5,200
Sorted without a compare function: 1,200,40,5,700,80,9
Sorted with compareNumbers: 1,5,9,40,80,200,700
```

See also `Array.join`, `Array.reverse`

splice

Changes the content of an array, adding new elements while removing old elements.

Method of	`Array`
Implemented in	JavaScript 1.2, NES 3.0 JavaScript 1.3: returns an array containing the removed elements

Syntax `splice(`*`index`*`, `*`howMany`*`, [element1][, ..., `*`elementN`*`])`

Parameters

`index`	Index at which to start changing the array.
`howMany`	An integer indicating the number of old array elements to remove. If `howMany` is 0, no elements are removed. In this case, you should specify at least one new element.
`element1, ..., element`*`N`*	The elements to add to the array. If you don't specify any elements, splice simply removes elements from the array.

Description If you specify a different number of elements to insert than the number you're removing, the array will have a different length at the end of the call.

The `splice` method returns an array containing the removed elements. If only one element is removed, an array of one element is returned

Backward Compatibility **JavaScript 1.2.** The `splice` method returns the element removed, if only one element is removed (`howMany` parameter is 1); otherwise, the method returns an array containing the removed elements.

Examples The following script illustrate the use of `splice`:

```
<SCRIPT LANGUAGE="JavaScript1.2">

myFish = ["angel", "clown", "mandarin", "surgeon"];
document.writeln("myFish: " + myFish + "<BR>");

removed = myFish.splice(2, 0, "drum");
document.writeln("After adding 1: " + myFish);
document.writeln("removed is: " + removed + "<BR>");

removed = myFish.splice(3, 1)
document.writeln("After removing 1: " + myFish);
document.writeln("removed is: " + removed + "<BR>");

removed = myFish.splice(2, 1, "trumpet")
document.writeln("After replacing 1: " + myFish);
document.writeln("removed is: " + removed + "<BR>");

removed = myFish.splice(0, 2, "parrot", "anemone", "blue")
document.writeln("After replacing 2: " + myFish);
document.writeln("removed is: " + removed);

</SCRIPT>
```

This script displays:

```
myFish: ["angel", "clown", "mandarin", "surgeon"]

After adding 1: ["angel", "clown", "drum", "mandarin", "surgeon"]
removed is: undefined

After removing 1: ["angel", "clown", "drum", "surgeon"]
removed is: mandarin

After replacing 1: ["angel", "clown", "trumpet", "surgeon"]
removed is: drum

After replacing 2: ["parrot", "anemone", "blue", "trumpet", "surgeon"]
removed is: ["angel", "clown"]
```

toSource

Returns a string representing the source code of the array.

Method of	`Array`
Implemented in	JavaScript 1.3

Syntax `toSource()`

Parameters None

Description The `toSource` method returns the following values:

- For the built-in `Array` object, `toSource` returns the following string indicating that the source code is not available:

  ```
  function Array() {
     [native code]
  }
  ```
- For instances of `Array`, `toSource` returns a string representing the source code.

This method is usually called internally by JavaScript and not explicitly in code. You can call `toSource` while debugging to examine the contents of an array.

Examples To examine the source code of an array:

```
alpha = new Array("a", "b", "c")
alpha.toSource() //returns ["a", "b", "c"]
```

See also `Array.toString`

toString

Returns a string representing the specified array and its elements.

Method of	`Array`
Implemented in	JavaScript 1.1, NES 2.0
ECMA version	ECMA-262

Syntax `toString()`

Parameters None.

Description The `Array` object overrides the `toString` method of `Object`. For `Array` objects, the `toString` method joins the array and returns one string containing each array element separated by commas. For example, the following code creates an array and uses `toString` to convert the array to a string.

```
var monthNames = new Array("Jan","Feb","Mar","Apr")
myVar=monthNames.toString() // assigns "Jan,Feb,Mar,Apr" to myVar
```

JavaScript calls the `toString` method automatically when an array is to be represented as a text value or when an array is referred to in a string concatenation.

Backward Compatibility **JavaScript 1.2.** In JavaScript 1.2 and earlier versions, `toString` returns a string representing the source code of the array. This value is the same as the value returned by the `toSource` method in JavaScript 1.3 and later versions.

See also `Array.toSource`

unshift

Adds one or more elements to the beginning of an array and returns the new length of the array.

Method of	`Array`
Implemented in	JavaScript 1.2, NES 3.0

Syntax `arrayName.unshift(`*`element1`*`,..., `*`elementN`*`)`

Parameters

`element1,..., elementN`	The elements to add to the front of the array.

Example The following code displays the `myFish` array before and after adding elements to it.

```
myFish = ["angel", "clown"];
document.writeln("myFish before: " + myFish);
unshifted = myFish.unshift("drum", "lion");
document.writeln("myFish after: " + myFish);
document.writeln("New length: " + unshifted);
```

This example displays the following:

```
myFish before: ["angel", "clown"]
myFish after: ["drum", "lion", "angel", "clown"]
New length: 4
```

See also `pop`, `push`, `shift`

valueOf

Returns the primitive value of an array.

Method of	`Array`
Implemented in	JavaScript 1.1
ECMA version	ECMA-262

Syntax `valueOf()`

Parameters None

Description The `Array` object inherits the `valueOf` method of `Object`. The `valueOf` method of `Array` returns the primitive value of an array or the primitive value of its elements as follows:

Object type of element	Data type of returned value
Boolean	Boolean
Number or Date	number
All others	string

This method is usually called internally by JavaScript and not explicitly in code.

See also `Object.valueOf`

Boolean

The `Boolean` object is an object wrapper for a boolean value.

Core object

Implemented in	JavaScript 1.1, NES 2.0 JavaScript 1.3: added `toSource` method
ECMA version	ECMA-262

Created by The `Boolean` constructor:

```
new Boolean(value)
```

Parameters

`value`	The initial value of the `Boolean` object. The value is converted to a `boolean` value, if necessary. If value is omitted or is 0, -0, null, false, `NaN`, undefined, or the empty string (`""`), the object has an initial value of false. All other values, including any object or the string `"false"`, create an object with an initial value of true.

Description Do not confuse the primitive Boolean values true and false with the true and false values of the Boolean object.

Any object whose value is not `undefined` or `null`, including a Boolean object whose value is false, evaluates to true when passed to a conditional statement. For example, the condition in the following `if` statement evaluates to `true`:

```
x = new Boolean(false);
if(x) //the condition is true
```

This behavior does not apply to Boolean primitives. For example, the condition in the following `if` statement evaluates to `false`:

```
x = false;
if(x) //the condition is false
```

Do not use a `Boolean` object to convert a non-boolean value to a boolean value. Instead, use Boolean as a function to perform this task:

```
x = Boolean(expression) //preferred
x = new Boolean(expression) //don't use
```

If you specify any object, including a Boolean object whose value is false, as the initial value of a Boolean object, the new Boolean object has a value of true.

```
myFalse=new Boolean(false)   // initial value of false
g=new Boolean(myFalse)       //initial value of true
myString=new String("Hello") // string object
s=new Boolean(myString)      //initial value of true
```

In JavaScript 1.3 and later versions, do not use a Boolean object in place of a Boolean primitive.

Backward Compatibility

JavaScript 1.2 and earlier versions. When a `Boolean` object is used as the condition in a conditional test, JavaScript returns the value of the `Boolean` object. For example, a `Boolean` object whose value is false is treated as the primitive value false, and a `Boolean` object whose value is true is treated as the primitive value `true` in conditional tests. If the `Boolean` object is a `false` object, the conditional statement evaluates to `false`.

Property Summary

Property	Description
`constructor`	Specifies the function that creates an object's prototype.
`prototype`	Defines a property that is shared by all Boolean objects.

Method Summary

Method	Description
`toSource`	Returns an object literal representing the specified Boolean object; you can use this value to create a new object. Overrides the `Object.toSource` method.
`toString`	Returns a string representing the specified object. Overrides the `Object.toString` method.
`valueOf`	Returns the primitive value of a Boolean object. Overrides the `Object.valueOf` method.

In addition, this object inherits the `watch` and `unwatch` methods from `Object`.

Examples The following examples create `Boolean` objects with an initial value of false:

```
bNoParam = new Boolean()
bZero = new Boolean(0)
bNull = new Boolean(null)
bEmptyString = new Boolean("")
bfalse = new Boolean(false)
```

The following examples create `Boolean` objects with an initial value of true:

```
btrue = new Boolean(true)
btrueString = new Boolean("true")
bfalseString = new Boolean("false")
bSuLin = new Boolean("Su Lin")
```

constructor

Specifies the function that creates an object's prototype. Note that the value of this property is a reference to the function itself, not a string containing the function's name.

Property of	`Boolean`
Implemented in	JavaScript 1.1, NES 2.0
ECMA version	ECMA-262

Description See `Object.constructor`.

prototype

Represents the prototype for this class. You can use the prototype to add properties or methods to all instances of a class. For information on prototypes, see `Function.prototype`.

Property of	`Boolean`
Implemented in	JavaScript 1.1, NES 2.0
ECMA version	ECMA-262

toSource

Returns a string representing the source code of the object.

Method of	`Boolean`
Implemented in	JavaScript 1.3

Syntax `toSource()`

Parameters None

Description The `toSource` method returns the following values:

- For the built-in `Boolean` object, `toSource` returns the following string indicating that the source code is not available:

```
function Boolean() {
   [native code]
}
```

- For instances of `Boolean`, `toSource` returns a string representing the source code.

This method is usually called internally by JavaScript and not explicitly in code.

See also `Object.toSource`

toString

Returns a string representing the specified Boolean object.

Method of	`Boolean`
Implemented in	JavaScript 1.1, NES 2.0
ECMA version	ECMA-262

Syntax `toString()`

Parameters None.

Description The `Boolean` object overrides the `toString` method of the `Object` object; it does not inherit `Object.toString`. For `Boolean` objects, the `toString` method returns a string representation of the object.

JavaScript calls the `toString` method automatically when a Boolean is to be represented as a text value or when a Boolean is referred to in a string concatenation.

For `Boolean` objects and values, the built-in `toString` method returns the string `"true"` or `"false"` depending on the value of the boolean object. In the following code, `flag.toString` returns `"true"`.

```
var flag = new Boolean(true)
var myVar=flag.toString()
```

See also `Object.toString`

valueOf

Returns the primitive value of a Boolean object.

Method of	`Boolean`
Implemented in	JavaScript 1.1
ECMA version	ECMA-262

Syntax `valueOf()`

Parameters None

Description The `valueOf` method of `Boolean` returns the primitive value of a Boolean object or literal Boolean as a Boolean data type.

This method is usually called internally by JavaScript and not explicitly in code.

Examples

```
x = new Boolean();
myVar=x.valueOf()      //assigns false to myVar
```

See also `Object.valueOf`

Button

A push button on an HTML form.

Client-side object

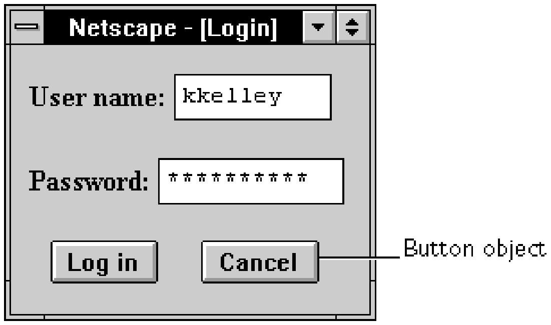
Implemented in	JavaScript 1.0
	JavaScript 1.1: added `type` property; added `onBlur` and `onFocus` event handlers; added `blur` and `focus` methods.
	JavaScript 1.2: added `handleEvent` method.

Created by The HTML `INPUT` tag, with `"button"` as the value of the `TYPE` attribute. For a given form, the JavaScript runtime engine creates appropriate `Button` objects and puts these objects in the `elements` array of the corresponding `Form` object. You access a `Button` object by indexing this array. You can index the array either by number or, if supplied, by using the value of the `NAME` attribute.

Event handlers

- `onBlur`
- `onClick`
- `onFocus`
- `onMouseDown`
- `onMouseUp`

Description A `Button` object on a form looks as follows:



A `Button` object is a form element and must be defined within a `FORM` tag.

The `Button` object is a custom button that you can use to perform an action you define. The button executes the script specified by its `onClick` event handler.

Property Summary

Property	Description
`form`	Specifies the form containing the `Button` object.
`name`	Reflects the `NAME` attribute.
`type`	Reflects the `TYPE` attribute.
`value`	Reflects the `VALUE` attribute.

Method Summary

Method	Description
`blur`	Removes focus from the button.
`click`	Simulates a mouse-click on the button.
`focus`	Gives focus to the button.
`handleEvent`	Invokes the handler for the specified event.

In addition, this object inherits the `watch` and `unwatch` methods from `Object`.

Examples The following example creates a button named `calcButton`. The text "Calculate" is displayed on the face of the button. When the button is clicked, the function `calcFunction` is called.

```
<INPUT TYPE="button" VALUE="Calculate" NAME="calcButton"
   onClick="calcFunction(this.form)">
```

See also `Form`, `Reset`, `Submit`

blur

Removes focus from the button.

Method of	`Button`
Implemented in	JavaScript 1.0

Syntax `blur()`

Parameters None

Examples The following example removes focus from the button element `userButton`:

```
userButton.blur()
```

This example assumes that the button is defined as

```
<INPUT TYPE="button" NAME="userButton">
```

See also `Button.focus`

click

Simulates a mouse-click on the button, but does not trigger the button's `onClick` event handler.

Method of	`Button`
Implemented in	JavaScript 1.0

Syntax `click()`

Parameters None.

Security Submitting a form to a `mailto:` or `news:` URL requires the `UniversalSendMail` privilege. For information on security, see the *Client-Side JavaScript Guide*.

focus

Navigates to the button and gives it focus.

Method of	`Button`
Implemented in	JavaScript 1.0

Syntax `focus()`

Parameters None.

See also `Button.blur`

form

An object reference specifying the form containing the button.

Property of	`Button`
Read-only	
Implemented in	JavaScript 1.0

Description Each form element has a `form` property that is a reference to the element's parent form. This property is especially useful in event handlers, where you might need to refer to another element on the current form.

Examples **Example 1.** In the following example, the form `myForm` contains a `Text` object and a button. When the user clicks the button, the value of the `Text` object is set to the form's name. The button's `onClick` event handler uses `this.form` to refer to the parent form, `myForm`.

```
<FORM NAME="myForm">
Form name:<INPUT TYPE="text" NAME="text1" VALUE="Beluga">
<P>
<INPUT NAME="button1" TYPE="button" VALUE="Show Form Name"
   onClick="this.form.text1.value=this.form.name">
</FORM>
```

Example 2. The following example shows a form with several elements. When the user clicks `button2`, the function `showElements` displays an alert dialog box containing the names of each element on the form `myForm`.

```
function showElements(theForm) {
   str = "Form Elements of form " + theForm.name + ": \n "
   for (i = 0; i < theForm.length; i++)
      str += theForm.elements[i].name + "\n"
   alert(str)
}
</script>
<FORM NAME="myForm">
Form name:<INPUT TYPE="text" NAME="text1" VALUE="Beluga">
<P>
<INPUT NAME="button1" TYPE="button" VALUE="Show Form Name"
   onClick="this.form.text1.value=this.form.name">
<INPUT NAME="button2" TYPE="button" VALUE="Show Form Elements"
   onClick="showElements(this.form)">
</FORM>
```

The alert dialog box displays the following text:

```
JavaScript Alert:
Form Elements of form myForm:
text1
button1
button2
```

Example 3. The following example uses an object reference, rather than the `this` keyword, to refer to a form. The code returns a reference to `myForm`, which is a form containing `myButton`.

```
document.myForm.myButton.form
```

See also `Form`

handleEvent

Invokes the handler for the specified event.

Method of	`Button`
Implemented in	JavaScript 1.2

Syntax `handleEvent(`*`event`*`)`

Parameters

`event`	The name of an event for which the object has an event handler.

Description For information on handling events, see the *Client-Side JavaScript Guide*.

name

A string specifying the button's name.

Property of	`Button`
Implemented in	JavaScript 1.0

Security **JavaScript 1.1.** This property is tainted by default. For information on data tainting, see the *Client-Side JavaScript Guide*.

Description The `name` property initially reflects the value of the `NAME` attribute. Changing the `name` property overrides this setting.

Do not confuse the `name` property with the label displayed on a button. The `value` property specifies the label for the button. The `name` property is not displayed on the screen; it is used to refer programmatically to the object.

If multiple objects on the same form have the same `NAME` attribute, an array of the given name is created automatically. Each element in the array represents an individual `Form` object. Elements are indexed in source order starting at 0. For example, if two `Text` elements and a `Button` element on the same form have their `NAME` attribute set to `"myField"`, an array with the elements `myField[0]`, `myField[1]`, and `myField[2]` is created. You need to be aware of this situation in your code and know whether `myField` refers to a single element or to an array of elements.

Examples In the following example, the `valueGetter` function uses a `for` loop to iterate over the array of elements on the `valueTest` form. The `msgWindow` window displays the names of all the elements on the form:

```
newWindow=window.open("http://home.netscape.com")

function valueGetter() {
   var msgWindow=window.open("")
   for (var i = 0; i < newWindow.document.valueTest.elements.length; i++) {
      msgWindow.document.write(newWindow.document.valueTest.elements[i].name + "<BR>")
   }
}
```

In the following example, the first statement creates a window called `netscapeWin`. The second statement displays the value `"netscapeHomePage"` in the Alert dialog box, because `"netscapeHomePage"` is the value of the `windowName` argument of `netscapeWin`.

```
netscapeWin=window.open("http://home.netscape.com","netscapeHomePage")

alert(netscapeWin.name)
```

See also `Button.value`

type

For all `Button` objects, the value of the `type` property is `"button"`. This property specifies the form element's type.

Property of	`Button`
Read-only	
Implemented in	JavaScript 1.1

Examples The following example writes the value of the `type` property for every element on a form.

```
for (var i = 0; i < document.form1.elements.length; i++) {
   document.writeln("<BR>type is " + document.form1.elements[i].type)
}
```

value

A string that reflects the button's `VALUE` attribute.

Property of	`Button`
Read-only on Mac and UNIX; modifiable on Windows	
Implemented in	JavaScript 1.0

Security **JavaScript 1.1.** This property is tainted by default. For information on data tainting, see the *Client-Side JavaScript Guide*.

Description This string is displayed on the face of the button.

The `value` property is read-only for Macintosh and UNIX systems. On Windows, you can change this property.

When a `VALUE` attribute is not specified in HTML, the `value` property is an empty string.

Do not confuse the `value` property with the `name` property. The `name` property is not displayed on the screen; it is used to refer programmatically to the objects.

Examples The following function evaluates the `value` property of a group of buttons and displays it in the `msgWindow` window:

```
function valueGetter() {
   var msgWindow=window.open("")
   msgWindow.document.write("submitButton.value is " +
      document.valueTest.submitButton.value + "<BR>")
   msgWindow.document.write("resetButton.value is " +
      document.valueTest.resetButton.value + "<BR>")
   msgWindow.document.write("helpButton.value is " +
      document.valueTest.helpButton.value + "<BR>")
   msgWindow.document.close()
}
```

This example displays the following values:

```
Query Submit
Reset
Help
```

The previous example assumes the buttons have been defined as follows:

```
<INPUT TYPE="submit" NAME="submitButton">
<INPUT TYPE="reset" NAME="resetButton">
<INPUT TYPE="button" NAME="helpButton" VALUE="Help">
```

See also `Button.name`

Checkbox

A checkbox on an HTML form. A checkbox is a toggle switch that lets the user set a value on or off.

Client-side object

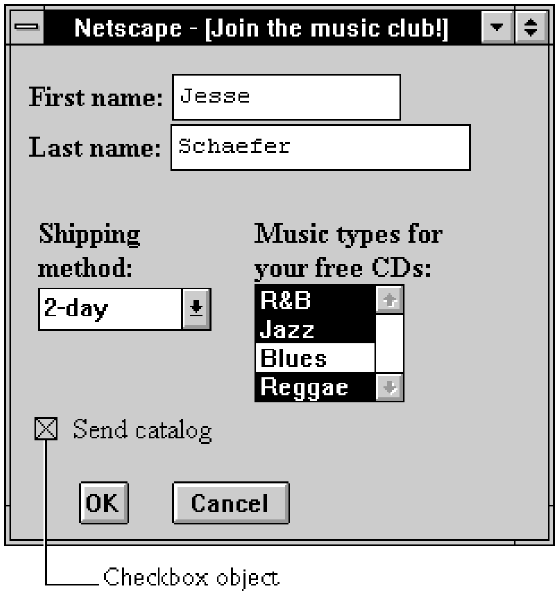
Implemented in	JavaScript 1.0 JavaScript 1.1: added `type` property; added `onBlur` and `onFocus` event handlers; added `blur` and `focus` methods. JavaScript 1.2: added `handleEvent` method.

Created by The HTML `INPUT` tag, with `"checkbox"` as the value of the `TYPE` attribute. For a given form, the JavaScript runtime engine creates appropriate `Checkbox` objects and puts these objects in the `elements` array of the corresponding `Form` object. You access a `Checkbox` object by indexing this array. You can index the array either by number or, if supplied, by using the value of the `NAME` attribute.

Event handlers

- `onBlur`
- `onClick`
- `onFocus`

Description A `Checkbox` object on a form looks as follows:



A `Checkbox` object is a form element and must be defined within a `FORM` tag.

Use the `checked` property to specify whether the checkbox is currently checked. Use the `defaultChecked` property to specify whether the checkbox is checked when the form is loaded or reset.

Property Summary

Property	Description
`checked`	Boolean property that reflects the current state of the checkbox.
`defaultChecked`	Boolean property that reflects the `CHECKED` attribute.
`form`	Specifies the form containing the `Checkbox` object.
`name`	Reflects the `NAME` attribute.
`type`	Reflects the `TYPE` attribute.
`value`	Reflects the `TYPE` attribute.

Method Summary

Method	Description
`blur`	Removes focus from the checkbox.
`click`	Simulates a mouse-click on the checkbox.
`focus`	Gives focus to the checkbox.
`handleEvent`	Invokes the handler for the specified event.

In addition, this object inherits the `watch` and `unwatch` methods from `Object`.

Examples **Example 1.** The following example displays a group of four checkboxes that all appear checked by default:

```
<B>Specify your music preferences (check all that apply):</B>
<BR><INPUT TYPE="checkbox" NAME="musicpref_rnb" CHECKED> R&B
<BR><INPUT TYPE="checkbox" NAME="musicpref_jazz" CHECKED> Jazz
<BR><INPUT TYPE="checkbox" NAME="musicpref_blues" CHECKED> Blues
<BR><INPUT TYPE="checkbox" NAME="musicpref_newage" CHECKED> New Age
```

Example 2. The following example contains a form with three text boxes and one checkbox. The user can use the checkbox to choose whether the text fields are converted to uppercase. Each text field has an `onChange` event handler that converts the field value to uppercase if the checkbox is checked. The checkbox has an `onClick` event handler that converts all fields to uppercase when the user checks the checkbox.

```
<HTML>
<HEAD>
<TITLE>Checkbox object example</TITLE>
</HEAD>
<SCRIPT>
function convertField(field) {
   if (document.form1.convertUpper.checked) {
      field.value = field.value.toUpperCase()}
}
function convertAllFields() {
   document.form1.lastName.value = document.form1.lastName.value.toUpperCase()
   document.form1.firstName.value = document.form1.firstName.value.toUpperCase()
   document.form1.cityName.value = document.form1.cityName.value.toUpperCase()
}
</SCRIPT>
```

```
<BODY>
<FORM NAME="form1">
<B>Last name:</B>
<INPUT TYPE="text" NAME="lastName" SIZE=20 onChange="convertField(this)">
<BR><B>First name:</B>
<INPUT TYPE="text" NAME="firstName" SIZE=20 onChange="convertField(this)">
<BR><B>City:</B>
<INPUT TYPE="text" NAME="cityName" SIZE=20 onChange="convertField(this)">
<P><INPUT TYPE="checkBox" NAME="convertUpper"
   onClick="if (this.checked) {convertAllFields()}"
   > Convert fields to upper case
</FORM>
</BODY>
</HTML>
```

See also `Form`, `Radio`

blur

Removes focus from the checkbox.

Method of	`Checkbox`
Implemented in	JavaScript 1.0

Syntax `blur()`

Parameters None

See also `Checkbox.focus`

checked

A Boolean value specifying the selection state of the checkbox.

Property of	`Checkbox`
Implemented in	JavaScript 1.0

Security **JavaScript 1.1.** This property is tainted by default. For information on data tainting, see the *Client-Side JavaScript Guide*.

Description If a checkbox button is selected, the value of its `checked` property is true; otherwise, it is false.

You can set the `checked` property at any time. The display of the checkbox button updates immediately when you set the `checked` property.

See also `Checkbox.defaultChecked`

click

Simulates a mouse-click on the checkbox, but does not trigger its `onClick` event handler. The method checks the checkbox and sets toggles its value.

Method of	`Checkbox`
Implemented in	JavaScript 1.0

Syntax `click()`

Parameters None.

Examples The following example toggles the selection status of the `newAge` checkbox on the `musicForm` form:

```
document.musicForm.newAge.click()
```

defaultChecked

A Boolean value indicating the default selection state of a checkbox button.

Property of	`Checkbox`
Implemented in	JavaScript 1.0

Security **JavaScript 1.1.** This property is tainted by default. For information on data tainting, see the *Client-Side JavaScript Guide*.

Description If a checkbox is selected by default, the value of the `defaultChecked` property is true; otherwise, it is false. `defaultChecked` initially reflects whether the `CHECKED` attribute is used within an `INPUT` tag; however, setting `defaultChecked` overrides the `CHECKED` attribute.

You can set the `defaultChecked` property at any time. The display of the checkbox does not update when you set the `defaultChecked` property, only when you set the `checked` property.

See also `Checkbox.checked`

focus

Gives focus to the checkbox.

Method of	`Checkbox`
Implemented in	JavaScript 1.0

Syntax `focus()`

Parameters None

Description Use the `focus` method to navigate to a the checkbox and give it focus. The user can then toggle the state of the checkbox.

See also `Checkbox.blur`

form

An object reference specifying the form containing the checkbox.

Property of	`Checkbox`
Read-only	
Implemented in	JavaScript 1.0

Description Each form element has a `form` property that is a reference to the element's parent form. This property is especially useful in event handlers, where you might need to refer to another element on the current form.

See also `Form`

handleEvent

Invokes the handler for the specified event.

Method of	`Checkbox`
Implemented in	JavaScript 1.2

Syntax `handleEvent(`*`event`*`)`

Parameters

`event`	The name of an event for which the specified object has an event handler.

name

A string specifying the checkbox's name.

Property of	`Checkbox`
Implemented in	JavaScript 1.0

Security **JavaScript 1.1.** This property is tainted by default. For information on data tainting, see the *Client-Side JavaScript Guide*.

Description If multiple objects on the same form have the same `NAME` attribute, an array of the given name is created automatically. Each element in the array represents an individual `Form` object. Elements are indexed in source order starting at 0. For example, if two `Text` elements and a `Button` element on the same form have their `NAME` attribute set to `"myField"`, an array with the elements `myField[0]`, `myField[1]`, and `myField[2]` is created. You need to be aware of this situation in your code and know whether `myField` refers to a single element or to an array of elements.

Examples In the following example, the `valueGetter` function uses a `for` loop to iterate over the array of elements on the `valueTest` form. The `msgWindow` window displays the names of all the elements on the form:

```
newWindow=window.open("http://home.netscape.com")

function valueGetter() {
   var msgWindow=window.open("")
   for (var i = 0; i < newWindow.document.valueTest.elements.length; i++) {
      msgWindow.document.write(newWindow.document.valueTest.elements[i].name + "<BR>")
   }
}
```

type

For all `Checkbox` objects, the value of the `type` property is `"checkbox"`. This property specifies the form element's type.

Property of	`Checkbox`
Read-only	
Implemented in	JavaScript 1.1

Examples The following example writes the value of the `type` property for every element on a form.

```
for (var i = 0; i < document.form1.elements.length; i++) {
   document.writeln("<BR>type is " + document.form1.elements[i].type)
}
```

value

A string that reflects the `VALUE` attribute of the checkbox.

Property of	`Checkbox`
Implemented in	JavaScript 1.0

Security **JavaScript 1.1.** This property is tainted by default. For information on data tainting, see the *Client-Side JavaScript Guide*.

See also `Checkbox.checked`, `Checkbox.defaultChecked`

Date

Lets you work with dates and times.

Core object

Implemented in	JavaScript 1.0, NES 2.0 JavaScript 1.1: added `prototype` property JavaScript 1.3: removed platform dependencies to provide a uniform behavior across platforms; added `ms_num` parameter to `Date` constructor; added `getFullYear`, `setFullYear`, `getMilliseconds`, `setMilliseconds`, `toSource`, and UTC methods (such as `getUTCDate` and `setUTCDate`).
ECMA version	ECMA-262

Created by

The `Date` constructor:

```
new Date()
new Date(milliseconds)
new Date(dateString)
new Date(yr_num, mo_num, day_num
        [, hr_num, min_num, sec_num, ms_num])
```

Versions prior to JavaScript 1.3:

```
new Date()
new Date(milliseconds)
new Date(dateString)
new Date(yr_num, mo_num, day_num[, hr_num, min_num, sec_num])
```

Parameters

`milliseconds`	Integer value representing the number of milliseconds since 1 January 1970 00:00:00.
`dateString`	String value representing a date. The string should be in a format recognized by the `Date.parse` method.
`yr_num, mo_num, day_num`	Integer values representing part of a date. As an integer value, the month is represented by 0 to 11 with 0=January and 11=December.
`hr_num, min_num, sec_num, ms_num`	Integer values representing part of a date.

Description If you supply no arguments, the constructor creates a `Date` object for today's date and time according to local time. If you supply some arguments but not others, the missing arguments are set to 0. If you supply any arguments, you must supply at least the year, month, and day. You can omit the hours, minutes, seconds, and milliseconds.

The date is measured in milliseconds since midnight 01 January, 1970 UTC. A day holds 86,400,000 milliseconds. The `Date` object range is -100,000,000 days to 100,000,000 days relative to 01 January, 1970 UTC.

The `Date` object provides uniform behavior across platforms.

The `Date` object supports a number of UTC (universal) methods, as well as local time methods. UTC, also known as Greenwich Mean Time (GMT), refers to the time as set by the World Time Standard. The local time is the time known to the computer where JavaScript is executed.

For compatibility with millennium calculations (in other words, to take into account the year 2000), you should always specify the year in full; for example, use 1998, not 98. To assist you in specifying the complete year, JavaScript includes the methods `getFullYear`, `setFullYear`, `getFullUTCYear`, and `setFullUTCYear`.

The following example returns the time elapsed between `timeA` and `timeB` in milliseconds.

```
timeA = new Date();
// Statements here to take some action.
timeB = new Date();
timeDifference = timeB - timeA;
```

Backward Compatibility **JavaScript 1.2 and earlier.** The `Date` object behaves as follows:

- Dates prior to 1970 are not allowed.
- JavaScript depends on platform-specific date facilities and behavior; the behavior of the `Date` object varies from platform to platform.

Property Summary

Property	Description
`constructor`	Specifies the function that creates an object's prototype.
`prototype`	Allows the addition of properties to a `Date` object.

Method Summary

Method	Description
`getDate`	Returns the day of the month for the specified date according to local time.
`getDay`	Returns the day of the week for the specified date according to local time.
`getFullYear`	Returns the year of the specified date according to local time.
`getHours`	Returns the hour in the specified date according to local time.
`getMilliseconds`	Returns the milliseconds in the specified date according to local time.
`getMinutes`	Returns the minutes in the specified date according to local time.
`getMonth`	Returns the month in the specified date according to local time.
`getSeconds`	Returns the seconds in the specified date according to local time.
`getTime`	Returns the numeric value corresponding to the time for the specified date according to local time.
`getTimezoneOffset`	Returns the time-zone offset in minutes for the current locale.
`getUTCDate`	Returns the day (date) of the month in the specified date according to universal time.
`getUTCDay`	Returns the day of the week in the specified date according to universal time.
`getUTCFullYear`	Returns the year in the specified date according to universal time.
`getUTCHours`	Returns the hours in the specified date according to universal time.
`getUTCMilliseconds`	Returns the milliseconds in the specified date according to universal time.
`getUTCMinutes`	Returns the minutes in the specified date according to universal time.
`getUTCMonth`	Returns the month according in the specified date according to universal time.

Method	Description
`getUTCSeconds`	Returns the seconds in the specified date according to universal time.
`getYear`	Returns the year in the specified date according to local time.
`parse`	Returns the number of milliseconds in a date string since January 1, 1970, 00:00:00, local time.
`setDate`	Sets the day of the month for a specified date according to local time.
`setFullYear`	Sets the full year for a specified date according to local time.
`setHours`	Sets the hours for a specified date according to local time.
`setMilliseconds`	Sets the milliseconds for a specified date according to local time.
`setMinutes`	Sets the minutes for a specified date according to local time.
`setMonth`	Sets the month for a specified date according to local time.
`setSeconds`	Sets the seconds for a specified date according to local time.
`setTime`	Sets the value of a Date object according to local time.
`setUTCDate`	Sets the day of the month for a specified date according to universal time.
`setUTCFullYear`	Sets the full year for a specified date according to universal time.
`setUTCHours`	Sets the hour for a specified date according to universal time.
`setUTCMilliseconds`	Sets the milliseconds for a specified date according to universal time.
`setUTCMinutes`	Sets the minutes for a specified date according to universal time.
`setUTCMonth`	Sets the month for a specified date according to universal time.

Method	Description
`setUTCSeconds`	Sets the seconds for a specified date according to universal time.
`setYear`	Sets the year for a specified date according to local time.
`toGMTString`	Converts a date to a string, using the Internet GMT conventions.
`toLocaleString`	Converts a date to a string, using the current locale's conventions.
`toSource`	Returns an object literal representing the specified Date object; you can use this value to create a new object. Overrides the `Object.toSource` method.
`toString`	Returns a string representing the specified Date object. Overrides the `Object.toString` method.
`toUTCString`	Converts a date to a string, using the universal time convention.
`UTC`	Returns the number of milliseconds in a `Date` object since January 1, 1970, 00:00:00, universal time.
`valueOf`	Returns the primitive value of a Date object. Overrides the `Object.valueOf` method.

In addition, this object inherits the `watch` and `unwatch` methods from `Object`.

Examples The following examples show several ways to assign dates:

```
today = new Date()
birthday = new Date("December 17, 1995 03:24:00")
birthday = new Date(95,11,17)
birthday = new Date(95,11,17,3,24,0)
```

constructor

Specifies the function that creates an object's prototype. Note that the value of this property is a reference to the function itself, not a string containing the function's name.

Property of	`Date`
Implemented in	JavaScript 1.1, NES 2.0
ECMA version	ECMA-262

Description See `Object.constructor`.

getDate

Returns the day of the month for the specified date according to local time.

Method of	`Date`
Implemented in	JavaScript 1.0, NES 2.0
ECMA version	ECMA-262

Syntax `getDate()`

Parameters None

Description The value returned by `getDate` is an integer between 1 and 31.

Examples The second statement below assigns the value 25 to the variable `day`, based on the value of the `Date` object `Xmas95`.

```
Xmas95 = new Date("December 25, 1995 23:15:00")
day = Xmas95.getDate()
```

See also `Date.getUTCDate`, `Date.getUTCDay`, `Date.setDate`

getDay

Returns the day of the week for the specified date according to local time.

Method of	`Date`
Implemented in	JavaScript 1.0, NES 2.0
ECMA version	ECMA-262

Syntax `getDay()`

Parameters None

Description The value returned by `getDay` is an integer corresponding to the day of the week: 0 for Sunday, 1 for Monday, 2 for Tuesday, and so on.

Examples The second statement below assigns the value 1 to `weekday`, based on the value of the `Date` object `Xmas95`. December 25, 1995, is a Monday.

```
Xmas95 = new Date("December 25, 1995 23:15:00")
weekday = Xmas95.getDay()
```

See also `Date.getUTCDay`, `Date.setDate`

getFullYear

Returns the year of the specified date according to local time.

Method of	`Date`
Implemented in	JavaScript 1.3
ECMA version	ECMA-262

Syntax `getFullYear()`

Parameters None

Description The value returned by `getFullYear` is an absolute number. For dates between the years 1000 and 9999, `getFullYear` returns a four-digit number, for example, 1995. Use this function to make sure a year is compliant with years after 2000.

Use this method instead of the `getYear` method.

Examples The following example assigns the four-digit value of the current year to the variable `yr`.

```
var yr;
Today = new Date();
yr = Today.getFullYear();
```

See also `Date.getYear`, `Date.getUTCFullYear`, `Date.setFullYear`

getHours

Returns the hour for the specified date according to local time.

Method of	`Date`
Implemented in	JavaScript 1.0, NES 2.0
ECMA version	ECMA-262

Syntax `getHours()`

Parameters None

Description The value returned by `getHours` is an integer between 0 and 23.

Examples The second statement below assigns the value 23 to the variable `hours`, based on the value of the `Date` object `Xmas95`.

```
Xmas95 = new Date("December 25, 1995 23:15:00")
hours = Xmas95.getHours()
```

See also `Date.getUTCHours`, `Date.setHours`

getMilliseconds

Returns the milliseconds in the specified date according to local time.

Method of	`Date`
Implemented in	JavaScript 1.3
ECMA version	ECMA-262

Syntax `getMilliseconds()`

Parameters None

Description The value returned by `getMilliseconds` is a number between 0 and 999.

Examples The following example assigns the milliseconds portion of the current time to the variable `ms`.

```
var ms;
Today = new Date();
ms = Today.getMilliseconds();
```

See also `Date.getUTCMilliseconds, Date.setMilliseconds`

getMinutes

Returns the minutes in the specified date according to local time.

Method of	`Date`
Implemented in	JavaScript 1.0, NES 2.0
ECMA version	ECMA-262

Syntax `getMinutes()`

Parameters None

Description The value returned by `getMinutes` is an integer between 0 and 59.

Examples The second statement below assigns the value 15 to the variable `minutes`, based on the value of the `Date` object `Xmas95`.

```
Xmas95 = new Date("December 25, 1995 23:15:00")
minutes = Xmas95.getMinutes()
```

See also `Date.getUTCMinutes`, `Date.setMinutes`

getMonth

Returns the month in the specified date according to local time.

Method of	`Date`
Implemented in	JavaScript 1.0, NES 2.0
ECMA version	ECMA-262

Syntax `getMonth()`

Parameters None

Description The value returned by `getMonth` is an integer between 0 and 11. 0 corresponds to January, 1 to February, and so on.

Examples The second statement below assigns the value 11 to the variable `month`, based on the value of the `Date` object `Xmas95`.

```
Xmas95 = new Date("December 25, 1995 23:15:00")
month = Xmas95.getMonth()
```

See also `Date.getUTCMonth`, `Date.setMonth`

getSeconds

Returns the seconds in the current time according to local time.

Method of	`Date`
Implemented in	JavaScript 1.0, NES 2.0
ECMA version	ECMA-262

Syntax `getSeconds()`

Parameters None

Description The value returned by `getSeconds` is an integer between 0 and 59.

Examples The second statement below assigns the value 30 to the variable `secs`, based on the value of the `Date` object `Xmas95`.

```
Xmas95 = new Date("December 25, 1995 23:15:30")
secs = Xmas95.getSeconds()
```

See also `Date.getUTCSeconds`, `Date.setSeconds`

getTime

Returns the numeric value corresponding to the time for the specified date according to local time.

Method of	`Date`
Implemented in	JavaScript 1.0, NES 2.0
ECMA version	ECMA-262

Syntax `getTime()`

Parameters None

Description The value returned by the `getTime` method is the number of milliseconds since 1 January 1970 00:00:00. You can use this method to help assign a date and time to another `Date` object.

Examples The following example assigns the date value of `theBigDay` to `sameAsBigDay`:

```
theBigDay = new Date("July 1, 1999")
sameAsBigDay = new Date()
sameAsBigDay.setTime(theBigDay.getTime())
```

See also `Date.getUTCHours`, `Date.setTime`

getTimezoneOffset

Returns the time-zone offset in minutes for the current locale.

Method of	`Date`
Implemented in	JavaScript 1.0, NES 2.0
ECMA version	ECMA-262

Syntax `getTimezoneOffset()`

Parameters None

Description The time-zone offset is the difference between local time and Greenwich Mean Time (GMT). Daylight savings time prevents this value from being a constant.

Examples

```
x = new Date()
currentTimeZoneOffsetInHours = x.getTimezoneOffset()/60
```

getUTCDate

Returns the day (date) of the month in the specified date according to universal time.

Method of	`Date`
Implemented in	JavaScript 1.3
ECMA version	ECMA-262

Syntax `getUTCDate()`

Parameters None

Description The value returned by `getUTCDate` is an integer between 1 and 31.

Examples The following example assigns the day portion of the current date to the variable `d`.

```
var d;
Today = new Date();
d = Today.getUTCDate();
```

See also `Date.getDate`, `Date.getUTCDay`, `Date.setUTCDate`

getUTCDay

Returns the day of the week in the specified date according to universal time.

Method of	`Date`
Implemented in	JavaScript 1.3
ECMA version	ECMA-262

Syntax `getUTCDay()`

Parameters None

Description The value returned by `getUTCDay` is an integer corresponding to the day of the week: 0 for Sunday, 1 for Monday, 2 for Tuesday, and so on.

Examples The following example assigns the weekday portion of the current date to the variable `ms`.

```
var weekday;
Today = new Date()
weekday = Today.getUTCDay()
```

See also `Date.getDay`, `Date.getUTCDate`, `Date.setUTCDate`

getUTCFullYear

Returns the year in the specified date according to universal time.

Method of	`Date`
Implemented in	JavaScript 1.3
ECMA version	ECMA-262

Syntax `getUTCFullYear()`

Parameters None

Description The value returned by `getUTCFullYear` is an absolute number that is compliant with year-2000, for example, 1995.

Examples The following example assigns the four-digit value of the current year to the variable `yr`.

```
var yr;
Today = new Date();
yr = Today.getUTCFullYear();
```

See also `Date.getFullYear`, `Date.setFullYear`

getUTCHours

Returns the hours in the specified date according to universal time.

Method of	`Date`
Implemented in	JavaScript 1.3
ECMA version	ECMA-262

Syntax `getUTCHours()`

Parameters None

Description The value returned by `getUTCHours` is an integer between 0 and 23.

Examples The following example assigns the hours portion of the current time to the variable `hrs`.

```
var hrs;
Today = new Date();
hrs = Today.getUTCHours();
```

See also `Date.getHours`, `Date.setUTCHours`

getUTCMilliseconds

Returns the milliseconds in the specified date according to universal time.

Method of	`Date`
Implemented in	JavaScript 1.3
ECMA version	ECMA-262

Syntax `getUTCMilliSeconds()`

Parameters None

Description The value returned by `getUTCMilliseconds` is an integer between 0 and 999.

Examples The following example assigns the milliseconds portion of the current time to the variable `ms`.

```
var ms;
Today = new Date();
ms = Today.getUTCMilliseconds();
```

See also `Date.getMilliseconds`, `Date.setUTCMilliseconds`

getUTCMinutes

Returns the minutes in the specified date according to universal time.

Method of	`Date`
Implemented in	JavaScript 1.3
ECMA version	ECMA-262

Syntax `getUTCMinutes()`

Parameters None

Description The value returned by `getUTCMinutes` is an integer between 0 and 59.

Examples The following example assigns the minutes portion of the current time to the variable `min`.

```
var min;
Today = new Date();
min = Today.getUTCMinutes();
```

See also `Date.getMinutes`, `Date.setUTCMinutes`

getUTCMonth

Returns the month according in the specified date according to universal time.

Method of	`Date`
Implemented in	JavaScript 1.3
ECMA version	ECMA-262

Syntax `getUTCMonth()`

Parameters None

Description The value returned by `getUTCMonth` is an integer between 0 and 11 corresponding to the month. 0 for January, 1 for February, 2 for March, and so on.

Examples The following example assigns the month portion of the current date to the variable `mon`.

```
var mon;
Today = new Date();
mon = Today.getUTCMonth();
```

See also `Date.getMonth`, `Date.setUTCMonth`

getUTCSeconds

Returns the seconds in the specified date according to universal time.

Method of	`Date`
Implemented in	JavaScript 1.3
ECMA version	ECMA-262

Syntax `getUTCSeconds()`

Parameters None

Description The value returned by `getUTCSeconds` is an integer between 0 and 59.

Examples The following example assigns the seconds portion of the current time to the variable `sec`.

```
var sec;
Today = new Date();
sec = Today.getUTCSeconds();
```

See also `Date.getSeconds`, `Date.setUTCSeconds`

getYear

Returns the year in the specified date according to local time.

Method of	`Date`
Implemented in	JavaScript 1.0, NES 2.0 JavaScript 1.3: deprecated; also, `getYear` returns the year minus 1900 regardless of the year specified
ECMA version	ECMA-262

Syntax `getYear()`

Parameters None

Description

`getYear` is no longer used and has been replaced by the `getFullYear` method.

The `getYear` method returns the year minus 1900; thus:

- For years above 2000, the value returned by `getYear` is 100 or greater. For example, if the year is 2026, `getYear` returns 126.
- For years between and including 1900 and 1999, the value returned by `getYear` is between 0 and 99. For example, if the year is 1976, `getYear` returns 76.
- For years less than 1900 or greater than 1999, the value returned by `getYear` is less than 0. For example, if the year is 1800, `getYear` returns -100.

To take into account years before and after 2000, you should use `Date.getFullYear` instead of `getYear` so that the year is specified in full.

Backward Compatibility

JavaScript 1.2 and earlier versions. The `getYear` method returns either a 2-digit or 4-digit year:

- For years between and including 1900 and 1999, the value returned by `getYear` is the year minus 1900. For example, if the year is 1976, the value returned is 76.
- For years less than 1900 or greater than 1999, the value returned by `getYear` is the four-digit year. For example, if the year is 1856, the value returned is 1856. If the year is 2026, the value returned is 2026.

Examples

Example 1. The second statement assigns the value 95 to the variable `year`.

```
Xmas = new Date("December 25, 1995 23:15:00")
year = Xmas.getYear() // returns 95
```

Example 2. The second statement assigns the value 100 to the variable `year`.

```
Xmas = new Date("December 25, 2000 23:15:00")
year = Xmas.getYear() // returns 100
```

Example 3. The second statement assigns the value -100 to the variable `year`.

```
Xmas = new Date("December 25, 1800 23:15:00")
year = Xmas.getYear() // returns -100
```

Example 4. The second statement assigns the value 95 to the variable `year`, representing the year 1995.

```
Xmas.setYear(95)
year = Xmas.getYear() // returns 95
```

See also `Date.getFullYear`, `Date.getUTCFullYear`, `Date.setYear`

parse

Returns the number of milliseconds in a date string since January 1, 1970, 00:00:00, local time.

Method of	`Date`
Static	
Implemented in	JavaScript 1.0, NES 2.0
ECMA version	ECMA-262

Syntax `Date.parse(`*`dateString`*`)`

Parameters

`dateString`	A string representing a date.

Description The `parse` method takes a date string (such as `"Dec 25, 1995"`) and returns the number of milliseconds since January 1, 1970, 00:00:00 (local time). This function is useful for setting date values based on string values, for example in conjunction with the `setTime` method and the `Date` object.

Given a string representing a time, `parse` returns the time value. It accepts the IETF standard date syntax: `"Mon, 25 Dec 1995 13:30:00 GMT"`. It understands the continental US time-zone abbreviations, but for general use, use a time-zone offset, for example, `"Mon, 25 Dec 1995 13:30:00 GMT+0430"` (4 hours, 30 minutes west of the Greenwich meridian). If you do not specify a time zone, the local time zone is assumed. GMT and UTC are considered equivalent.

Because `parse` is a static method of `Date`, you always use it as `Date.parse()`, rather than as a method of a `Date` object you created.

Examples If `IPOdate` is an existing `Date` object, then you can set it to August 9, 1995 as follows:

```
IPOdate.setTime(Date.parse("Aug 9, 1995"))
```

See also `Date.UTC`

prototype

Represents the prototype for this class. You can use the prototype to add properties or methods to all instances of a class. For information on prototypes, see `Function.prototype`.

Property of	`Date`
Implemented in	JavaScript 1.1, NES 2.0
ECMA version	ECMA-262

setDate

Sets the day of the month for a specified date according to local time.

Method of	`Date`
Implemented in	JavaScript 1.0, NES 2.0
ECMA version	ECMA-262

Syntax `setDate(`*`dayValue`*`)`

Parameters

`dayValue`	An integer from 1 to 31, representing the day of the month.

Examples The second statement below changes the day for `theBigDay` to July 24 from its original value.

```
theBigDay = new Date("July 27, 1962 23:30:00")
theBigDay.setDate(24)
```

See also `Date.getDate`, `Date.setUTCDate`

setFullYear

Sets the full year for a specified date according to local time.

Method of	`Date`
Implemented in	JavaScript 1.3
ECMA version	ECMA-262

Syntax `setFullYear(`*`yearValue`*`[, `*`monthValue`*`, `*`dayValue`*`])`

Parameters

`yearValue`	An integer specifying the numeric value of the year, for example, 1995.
`monthValue`	An integer between 0 and 11 representing the months January through December.
`dayValue`	An integer between 1 and 31 representing the day of the month. If you specify the `dayValue` parameter, you must also specify the `monthValue`.

Description If you do not specify the `monthValue` and `dayValue` parameters, the values returned from the `getMonth` and `getDate` methods are used.

If a parameter you specify is outside of the expected range, `setFullYear` attempts to update the other parameters and the date information in the `Date` object accordingly. For example, if you specify 15 for `monthValue`, the year is incremented by 1 (year + 1), and 3 is used for the month.

Examples

```
theBigDay = new Date();
theBigDay.setFullYear(1997);
```

See also `Date.getUTCFullYear`,`Date.setUTCFullYear`, `Date.setYear`

setHours

Sets the hours for a specified date according to local time.

Method of	`Date`
Implemented in	JavaScript 1.0, NES 2.0 JavaScript 1.3: Added `minutesValue`, `secondsValue`, and `msValue` parameters
ECMA version	ECMA-262

Syntax

```
setHours(hoursValue[, minutesValue, secondsValue, msValue])
```

Versions prior to JavaScript 1.3:

```
setHours(hoursValue)
```

Parameters

`hoursValue`	An integer between 0 and 23, representing the hour.
`minutesValue`	An integer between 0 and 59, representing the minutes.
`secondsValue`	An integer between 0 and 59, representing the seconds. If you specify the `secondsValue` parameter, you must also specify the `minutesValue`.
`msValue`	A number between 0 and 999, representing the milliseconds. If you specify the `msValue` parameter, you must also specify the `minutesValue` and `secondsValue`.

Description

If you do not specify the `minutesValue`, `secondsValue`, and `msValue` parameters, the values returned from the `getUTCMinutes`, `getUTCSeconds`, and `getMilliseconds` methods are used.

If a parameter you specify is outside of the expected range, `setHours` attempts to update the date information in the `Date` object accordingly. For example, if you use 100 for `secondsValue`, the minutes will be incremented by 1 (min + 1), and 40 will be used for seconds.

Examples

```
theBigDay.setHours(7)
```

See also `Date.getHours`, `Date.setUTCHours`

setMilliseconds

Sets the milliseconds for a specified date according to local time.

Method of	`Date`
Implemented in	JavaScript 1.3
ECMA version	ECMA-262

Syntax `setMilliseconds(`*`millisecondsValue`*`)`

Parameters

`millisecondsValue`	A number between 0 and 999, representing the milliseconds.

Description If you specify a number outside the expected range, the date information in the `Date` object is updated accordingly. For example, if you specify 1005, the number of seconds is incremented by 1, and 5 is used for the milliseconds.

Examples

```
theBigDay = new Date();
theBigDay.setMilliseconds(100);
```

See also `Date.getMilliseconds`, `Date.setUTCMilliseconds`

setMinutes

Sets the minutes for a specified date according to local time.

Method of	`Date`
Implemented in	JavaScript 1.0, NES 2.0
	JavaScript 1.3: Added `secondsValue` and `msValue` parameters
ECMA version	ECMA-262

Syntax `setMinutes(`*`minutesValue`*`[, `*`secondsValue`*`, `*`msValue`*`])`

Versions prior to JavaScript 1.3:

`setMinutes(minutesValue)`

Parameters

`minutesValue`	An integer between 0 and 59, representing the minutes.
`secondsValue`	An integer between 0 and 59, representing the seconds. If you specify the `secondsValue` parameter, you must also specify the `minutesValue`.
`msValue`	A number between 0 and 999, representing the milliseconds. If you specify the `msValue` parameter, you must also specify the `minutesValue` and `secondsValue`.

Examples

```
theBigDay.setMinutes(45)
```

Description If you do not specify the `secondsValue` and `msValue` parameters, the values returned from `getSeconds` and `getMilliseconds` methods are used.

If a parameter you specify is outside of the expected range, `setMinutes` attempts to update the date information in the `Date` object accordingly. For example, if you use 100 for `secondsValue`, the minutes (`minutesValue`) will be incremented by 1 (`minutesValue` + 1), and 40 will be used for seconds.

See also `Date.getMinutes`, `Date.setUTCMilliseconds`

setMonth

Sets the month for a specified date according to local time.

Method of	`Date`
Implemented in	JavaScript 1.0, NES 2.0 JavaScript 1.3: Added `dayValue` parameter
ECMA version	ECMA-262

Syntax `setMonth(`*`monthValue`*`[, `*`dayValue`*`])`

Versions prior to JavaScript 1.3:

`setMonth(`*`monthValue`*`)`

Parameters

`monthValue`	An integer between 0 and 11 (representing the months January through December).
`dayValue`	An integer from 1 to 31, representing the day of the month.

Description If you do not specify the `dayValue` parameter, the value returned from the `getDate` method is used.

If a parameter you specify is outside of the expected range, `setMonth` attempts to update the date information in the `Date` object accordingly. For example, if you use 15 for `monthValue`, the year will be incremented by 1 (year + 1), and 3 will be used for month.

Examples

```
theBigDay.setMonth(6)
```

See also `Date.getMonth`, `Date.setUTCMonth`

setSeconds

Sets the seconds for a specified date according to local time.

Method of	`Date`
Implemented in	JavaScript 1.0, NES 2.0 JavaScript 1.3: Added `msValue` parameter
ECMA version	ECMA-262

Syntax `setSeconds(`*`secondsValue`*`[, `*`msValue`*`])`

Versions prior to JavaScript 1.3:

`setSeconds(`*`secondsValue`*`)`

Parameters

`secondsValue`	An integer between 0 and 59.
`msValue`	A number between 0 and 999, representing the milliseconds.

Description If you do not specify the `msValue` parameter, the value returned from the `getMilliseconds` methods is used.

If a parameter you specify is outside of the expected range, `setSeconds` attempts to update the date information in the `Date` object accordingly. For example, if you use 100 for `secondsValue`, the minutes stored in the `Date` object will be incremented by 1, and 40 will be used for seconds.

Examples

```
theBigDay.setSeconds(30)
```

See also `Date.getSeconds`, `Date.setUTCSeconds`

setTime

Sets the value of a `Date` object according to local time.

Method of	`Date`
Implemented in	JavaScript 1.0, NES 2.0
ECMA version	ECMA-262

Syntax `setTime(`*`timevalue`*`)`

Parameters

`timevalue`	An integer representing the number of milliseconds since 1 January 1970 00:00:00.

Description Use the `setTime` method to help assign a date and time to another `Date` object.

Examples

```
theBigDay = new Date("July 1, 1999")
sameAsBigDay = new Date()
sameAsBigDay.setTime(theBigDay.getTime())
```

See also `Date.getTime`, `Date.setUTCHours`

setUTCDate

Sets the day of the month for a specified date according to universal time.

Method of	`Date`
Implemented in	JavaScript 1.3
ECMA version	ECMA-262

Syntax `setUTCDate(`*`dayValue`*`)`

Parameters

`dayValue`	An integer from 1 to 31, representing the day of the month.

Description If a parameter you specify is outside of the expected range, `setUTCDate` attempts to update the date information in the `Date` object accordingly. For example, if you use 40 for `dayValue`, and the month stored in the `Date` object is June, the day will be changed to 10 and the month will be incremented to July.

Examples

```
theBigDay = new Date();
theBigDay.setUTCDate(20);
```

See also `Date.getUTCDate`, `Date.setDate`

setUTCFullYear

Sets the full year for a specified date according to universal time.

Method of	`Date`
Implemented in	JavaScript 1.3
ECMA version	ECMA-262

Syntax `setUTCFullYear(`*`yearValue`*`[,` *`monthValue`*`,` *`dayValue`*`])`

Parameters

`yearValue`	An integer specifying the numeric value of the year, for example, 1995.
`monthValue`	An integer between 0 and 11 representing the months January through December.
`dayValue`	An integer between 1 and 31 representing the day of the month. If you specify the `dayValue` parameter, you must also specify the `monthValue`.

Description If you do not specify the `monthValue` and `dayValue` parameters, the values returned from the `getMonth` and `getDate` methods are used.

If a parameter you specify is outside of the expected range, `setUTCFullYear` attempts to update the other parameters and the date information in the `Date` object accordingly. For example, if you specify 15 for `monthValue`, the year is incremented by 1 (year + 1), and 3 is used for the month.

Examples

```
theBigDay = new Date();
theBigDay.setUTCFullYear(1997);
```

See also `Date.getUTCFullYear`, `Date.setFullYear`

setUTCHours

Sets the hour for a specified date according to universal time.

Method of	`Date`
Implemented in	JavaScript 1.3
ECMA version	ECMA-262

Syntax `setUTCHour(`*`hoursValue`*`[, `*`minutesValue`*`, `*`secondsValue`*`, `*`msValue`*`])`

Parameters

`hoursValue`	An integer between 0 and 23, representing the hour.
`minutesValue`	An integer between 0 and 59, representing the minutes.
`secondsValue`	An integer between 0 and 59, representing the seconds. If you specify the `secondsValue` parameter, you must also specify the `minutesValue`.
`msValue`	A number between 0 and 999, representing the milliseconds. If you specify the `msValue` parameter, you must also specify the `minutesValue` and `secondsValue`.

Description If you do not specify the `minutesValue`, `secondsValue`, and `msValue` parameters, the values returned from the `getUTCMinutes`, `getUTCSeconds`, and `getUTCMilliseconds` methods are used.

If a parameter you specify is outside of the expected range, `setUTCHours` attempts to update the date information in the `Date` object accordingly. For example, if you use 100 for `secondsValue`, the minutes will be incremented by 1 (min + 1), and 40 will be used for seconds.

Examples

```
theBigDay = new Date();
theBigDay.setUTCHour(8);
```

See also `Date.getUTCHours, Date.setHours`

setUTCMilliseconds

Sets the milliseconds for a specified date according to universal time.

Method of	`Date`
Implemented in	JavaScript 1.3
ECMA version	ECMA-262

Syntax `setUTCMilliseconds(`*`millisecondsValue`*`)`

Parameters

`millisecondsValue`	A number between 0 and 999, representing the milliseconds.

Description If a parameter you specify is outside of the expected range, `setUTCMilliseconds` attempts to update the date information in the `Date` object accordingly. For example, if you use 1100 for `millisecondsValue`, the seconds stored in the `Date` object will be incremented by 1, and 100 will be used for milliseconds.

Examples

```
theBigDay = new Date();
theBigDay.setUTCMilliseconds(500);
```

See also `Date.getUTCMilliseconds`, `Date.setMilliseconds`

setUTCMinutes

Sets the minutes for a specified date according to universal time.

Method of	`Date`
Implemented in	JavaScript 1.3
ECMA version	ECMA-262

Syntax `setUTCMinutes(`*`minutesValue`*`[, `*`secondsValue`*`, `*`msValue`*`])`

Parameters

`minutesValue`	An integer between 0 and 59, representing the minutes.
`secondsValue`	An integer between 0 and 59, representing the seconds. If you specify the `secondsValue` parameter, you must also specify the `minutesValue`.
`msValue`	A number between 0 and 999, representing the milliseconds. If you specify the `msValue` parameter, you must also specify the `minutesValue` and `secondsValue`.

Description If you do not specify the `secondsValue` and `msValue` parameters, the values returned from `getUTCSeconds` and `getUTCMilliseconds` methods are used.

If a parameter you specify is outside of the expected range, `setUTCMinutes` attempts to update the date information in the `Date` object accordingly. For example, if you use 100 for `secondsValue`, the minutes (`minutesValue`) will be incremented by 1 (`minutesValue` + 1), and 40 will be used for seconds.

Examples

```
theBigDay = new Date();
theBigDay.setUTCMinutes(43);
```

See also `Date.getUTCMinutes`, `Date.setMinutes`

setUTCMonth

Sets the month for a specified date according to universal time.

Method of	`Date`
Implemented in	JavaScript 1.3
ECMA version	ECMA-262

Syntax `setUTCMonth(`*`monthValue`*`[, `*`dayValue`*`])`

Parameters

`monthValue`	An integer between 0 and 11, representing the months January through December.
`dayValue`	An integer from 1 to 31, representing the day of the month.

Description If you do not specify the `dayValue` parameter, the value returned from the `getUTCDate` method is used.

If a parameter you specify is outside of the expected range, `setUTCMonth` attempts to update the date information in the `Date` object accordingly. For example, if you use 15 for `monthValue`, the year will be incremented by 1 (year + 1), and 3 will be used for month.

Examples

```
theBigDay = new Date();
theBigDay.setUTCMonth(11);
```

See also `Date.getUTCMonth`, `Date.setMonth`

setUTCSeconds

Sets the seconds for a specified date according to universal time.

Method of	`Date`
Implemented in	JavaScript 1.3
ECMA version	ECMA-262

Syntax `setUTCSeconds(`*`secondsValue`*`[, `*`msValue`*`])`

Parameters

`secondsValue`	An integer between 0 and 59.
`msValue`	A number between 0 and 999, representing the milliseconds.

Description If you do not specify the `msValue` parameter, the value returned from the `getUTCMilliseconds` methods is used.

If a parameter you specify is outside of the expected range, `setUTCSeconds` attempts to update the date information in the `Date` object accordingly. For example, if you use 100 for `secondsValue`, the minutes stored in the `Date` object will be incremented by 1, and 40 will be used for seconds.

Examples

```
theBigDay = new Date();
theBigDay.setUTCSeconds(20);
```

See also `Date.getUTCSeconds`, `Date.setSeconds`

setYear

Sets the year for a specified date according to local time.

Method of	`Date`
Implemented in	JavaScript 1.0, NES 2.0 Deprecated in JavaScript 1.3
ECMA version	ECMA-262

Syntax `setYear(`*`yearValue`*`)`

Parameters

`yearValue`	An integer.

Description `setYear` is no longer used and has been replaced by the `setFullYear` method.

If `yearValue` is a number between 0 and 99 (inclusive), then the year for `dateObjectName` is set to 1900 + `yearValue`. Otherwise, the year for `dateObjectName` is set to `yearValue`.

To take into account years before and after 2000, you should use `setFullYear` instead of `setYear` so that the year is specified in full.

Examples Note that there are two ways to set years in the 20th century.

Example 1. The year is set to 1996.

```
theBigDay.setYear(96)
```

Example 2. The year is set to 1996.

```
theBigDay.setYear(1996)
```

Example 3. The year is set to 2000.

```
theBigDay.setYear(2000)
```

See also `Date.getYear`, `Date.setFullYear`, `Date.setUTCFullYear`

toGMTString

Converts a date to a string, using the Internet GMT conventions.

Method of	`Date`
Implemented in	JavaScript 1.0, NES 2.0 Deprecated in JavaScript 1.3
ECMA version	ECMA-262

Syntax `toGMTString()`

Parameters None

Description `toGMTString` is no longer used and has been replaced by the `toUTCString` method.

The exact format of the value returned by `toGMTString` varies according to the platform.

You should use `Date.toUTCString` instead of `toGMTSTring`.

Examples In the following example, `today` is a `Date` object:

```
today.toGMTString()
```

In this example, the `toGMTString` method converts the date to GMT (UTC) using the operating system's time-zone offset and returns a string value that is similar to the following form. The exact format depends on the platform.

```
Mon, 18 Dec 1995 17:28:35 GMT
```

See also `Date.toLocaleString`, `Date.toUTCString`

toLocaleString

Converts a date to a string, using the current locale's conventions.

Method of	`Date`
Implemented in	JavaScript 1.0, NES 2.0
ECMA version	ECMA-262

Syntax `toLocaleString()`

Parameters None

Description If you pass a date using `toLocaleString`, be aware that different platforms assemble the string in different ways. Methods such as `getHours`, `getMinutes`, and `getSeconds` give more portable results.

The `toLocaleString` method relies on the underlying operating system in formatting dates. It converts the date to a string using the formatting convention of the operating system where the script is running. For example, in the United States, the month appears before the date (04/15/98), whereas in Germany the date appears before the month (15.04.98). If the operating system is not year-2000 compliant and does not use the full year for years before 1900 or over 2000, `toLocaleString` returns a string that is not year-2000 compliant. `toLocaleString` behaves similarly to `toString` when converting a year that the operating system does not properly format.

Examples In the following example, `today` is a `Date` object:

```
today = new Date(95,11,18,17,28,35) //months are represented by 0 to 11
today.toLocaleString()
```

In this example, `toLocaleString` returns a string value that is similar to the following form. The exact format depends on the platform.

```
12/18/95 17:28:35
```

See also `Date.toGMTString`, `Date.toUTCString`

toSource

Returns a string representing the source code of the object.

Method of	`Date`
Implemented in	JavaScript 1.3
ECMA version	ECMA-262

Syntax `toSource()`

Parameters None

Description The `toSource` method returns the following values:

- For the built-in `Date` object, `toSource` returns the following string indicating that the source code is not available:

```
function Date() {
   [native code]
}
```

- For instances of `Date`, `toSource` returns a string representing the source code.

This method is usually called internally by JavaScript and not explicitly in code.

See also `Object.toSource`

toString

Returns a string representing the specified Date object.

Method of	`Date`
Implemented in	JavaScript 1.1, NES 2.0
ECMA version	ECMA-262

Syntax `toString()`

Parameters None.

Description The `Date` object overrides the `toString` method of the `Object` object; it does not inherit `Object.toString`. For `Date` objects, the `toString` method returns a string representation of the object.

JavaScript calls the `toString` method automatically when a date is to be represented as a text value or when a date is referred to in a string concatenation.

Examples The following example assigns the `toString` value of a Date object to `myVar`:

```
x = new Date();
myVar=x.toString();   //assigns a value to myVar similar to:
     //Mon Sep 28 14:36:22 GMT-0700 (Pacific Daylight Time) 1998
```

See also `Object.toString`

toUTCString

Converts a date to a string, using the universal time convention.

Method of	`Date`
Implemented in	JavaScript 1.3
ECMA version	ECMA-262

Syntax `toUTCString()`

Parameters None

Description The value returned by `toUTCString` is a readable string formatted according to UTC convention. The format of the return value may vary according to the platform.

Examples

```
var UTCstring;
Today = new Date();
UTCstring = Today.toUTCString();
```

See also `Date.toLocaleString`, `Date.toUTCString`

UTC

Returns the number of milliseconds in a `Date` object since January 1, 1970, 00:00:00, universal time.

Method of	`Date`
Static	
Implemented in	JavaScript 1.0, NES 2.0
	JavaScript 1.3: added `ms` parameter
ECMA version	ECMA-262

Syntax

`Date.UTC(`*`year`*`, `*`month`*`, `*`day`*`[, `*`hrs`*`, `*`min`*`, `*`sec`*`, `*`ms`*`])`

Parameters

`year`	A year after 1900.
`month`	An integer between 0 and 11 representing the month.
`date`	An integer between 1 and 31 representing the day of the month.
`hrs`	An integer between 0 and 23 representing the hours.
`min`	An integer between 0 and 59 representing the minutes.
`sec`	An integer between 0 and 59 representing the seconds.
`ms`	An integer between 0 and 999 representing the milliseconds.

Description

`UTC` takes comma-delimited date parameters and returns the number of milliseconds between January 1, 1970, 00:00:00, universal time and the time you specified.

You should specify a full year for the year; for example, 1998. If a year between 0 and 99 is specified, the method converts the year to a year in the 20th century (1900 + year); for example, if you specify 95, the year 1995 is used.

The `UTC` method differs from the `Date` constructor in two ways.

- `Date.UTC` uses universal time instead of the local time.
- `Date.UTC` returns a time value as a number instead of creating a `Date` object.

If a parameter you specify is outside of the expected range, the `UTC` method updates the other parameters to allow for your number. For example, if you use 15 for `month`, the year will be incremented by 1 (year + 1), and 3 will be used for the month.

Because UTC is a static method of `Date`, you always use it as `Date.UTC()`, rather than as a method of a `Date` object you created.

Examples The following statement creates a `Date` object using GMT instead of local time:

```
gmtDate = new Date(Date.UTC(96, 11, 1, 0, 0, 0))
```

See also `Date.parse`

valueOf

Returns the primitive value of a Date object.

Method of	`Date`
Implemented in	JavaScript 1.1
ECMA version	ECMA-262

Syntax `valueOf()`

Parameters None

Description The `valueOf` method of `Date` returns the primitive value of a Date object as a number data type, the number of milliseconds since midnight 01 January, 1970 UTC.

This method is usually called internally by JavaScript and not explicitly in code.

Examples

```
x = new Date(56,6,17);
myVar=x.valueOf()      //assigns -424713600000 to myVar
```

See also `Object.valueOf`

document

Contains information about the current document, and provides methods for displaying HTML output to the user.

Client-side object

Implemented in	JavaScript 1.0 JavaScript 1.1: added `onBlur` and `onFocus` syntax; added `applets`, `domain`, `embeds`, `forms`, *`formName`*, `images`, and `plugins` properties. JavaScript 1.2: added `classes`, `ids`, `layers`, and `tags` properties; added `captureEvents`, `contextual`, `getSelection`, `handleEvent`, `releaseEvents`, and `routeEvent` methods.

Created by The HTML `BODY` tag. The JavaScript runtime engine creates a `document` object for each HTML page. Each `window` object has a `document` property whose value is a `document` object.

To define a `document` object, use standard HTML syntax for the `BODY` tag with the addition of JavaScript event handlers.

Event handlers The `onBlur`, `onFocus`, `onLoad`, and `onUnload` event handlers are specified in the `BODY` tag but are actually event handlers for the `window` object. The following are event handlers for the `document` object.

- `onClick`
- `onDblClick`
- `onKeyDown`
- `onKeyPress`
- `onKeyUp`
- `onMouseDown`
- `onMouseUp`

Description An HTML document consists of `HEAD` and `BODY` tags. The `HEAD` tag includes information on the document's title and base (the absolute URL base to be used for relative URL links in the document). The `BODY` tag encloses the body of a document, which is defined by the current URL. The entire body of the document (all other HTML elements for the document) goes within the `BODY` tag.

You can load a new document by setting the `window.location` property.

You can clear the document pane (and remove the text, form elements, and so on so they do not redisplay) with these statements:

```
document.close();
document.open();
document.write();
```

You can omit the `document.open` call if you are writing text or HTML, since `write` does an implicit open of that MIME type if the document stream is closed.

You can refer to the anchors, forms, and links of a document by using the `anchors`, `forms`, and `links` arrays. These arrays contain an entry for each anchor, form, or link in a document and are properties of the `document` object.

Do not use `location` as a property of the `document` object; use the `document.URL` property instead. The `document.location` property, which is a synonym for `document.URL`, is deprecated.

Property Summary

Property	Description
`alinkColor`	A string that specifies the `ALINK` attribute.
`anchors`	An array containing an entry for each anchor in the document.
`applets`	An array containing an entry for each applet in the document.
`bgColor`	A string that specifies the `BGCOLOR` attribute.
`classes`	Creates a `Style` object that can specify the styles of HTML tags with a specific CLASS attribute.
`cookie`	Specifies a cookie.
`domain`	Specifies the domain name of the server that served a document.
`embeds`	An array containing an entry for each plug-in in the document.
`fgColor`	A string that specifies the `TEXT` attribute.
`formName`	A separate property for each named form in the document.
`forms`	An array a containing an entry for each form in the document.
`height`	The height of the document, in pixels.

Property	Description
`ids`	Creates a `Style` object that can specify the style of individual HTML tags.
`images`	An array containing an entry for each image in the document.
`lastModified`	A string that specifies the date the document was last modified.
`layers`	Array containing an entry for each layer within the document.
`linkColor`	A string that specifies the `LINK` attribute.
`links`	An array containing an entry for each link in the document.
`plugins`	An array containing an entry for each plug-in in the document.
`referrer`	A string that specifies the URL of the calling document.
`tags`	Creates a `Style` object that can specify the styles of HTML tags.
`title`	A string that specifies the contents of the `TITLE` tag.
`URL`	A string that specifies the complete URL of a document.
`vlinkColor`	A string that specifies the `VLINK` attribute.
`width`	The width of the document, in pixels.

Method Summary

Method	Description
`captureEvents`	Sets the document to capture all events of the specified type.
`close`	Closes an output stream and forces data to display.
`contextual`	Uses contextual selection criteria to specify a `Style` object that can set the style of individual HTML tags.
`getSelection`	Returns a string containing the text of the current selection.
`handleEvent`	Invokes the handler for the specified event.
`open`	Opens a stream to collect the output of `write` or `writeln` methods.
`releaseEvents`	Sets the window or document to release captured events of the specified type, sending the event to objects further along the event hierarchy.
`routeEvent`	Passes a captured event along the normal event hierarchy.

Method	Description
`write`	Writes one or more HTML expressions to a document in the specified window.
`writeln`	Writes one or more HTML expressions to a document in the specified window and follows them with a newline character.

In addition, this object inherits the `watch` and `unwatch` methods from `Object`.

Examples The following example creates two frames, each with one document. The document in the first frame contains links to anchors in the document of the second frame. Each document defines its colors.

`doc0.html`, which defines the frames, contains the following code:

```
<HTML>
<HEAD>
<TITLE>Document object example</TITLE>
</HEAD>
<FRAMESET COLS="30%,70%">
<FRAME SRC="doc1.html" NAME="frame1">
<FRAME SRC="doc2.html" NAME="frame2">
</FRAMESET>
</HTML>
```

`doc1.html`, which defines the content for the first frame, contains the following code:

```
<HTML>
<SCRIPT>
</SCRIPT>
<BODY
   BGCOLOR="antiquewhite"
   TEXT="darkviolet"
   LINK="fuchsia"
   ALINK="forestgreen"
   VLINK="navy">
<P><B>Some links</B>
<LI><A HREF="doc2.html#numbers" TARGET="frame2">Numbers</A>
<LI><A HREF="doc2.html#colors" TARGET="frame2">Colors</A>
<LI><A HREF="doc2.html#musicTypes" TARGET="frame2">Music types</A>
<LI><A HREF="doc2.html#countries" TARGET="frame2">Countries</A>
</BODY>
</HTML>
```

doc2.html, which defines the content for the second frame, contains the following code:

```
<HTML>
<SCRIPT>
</SCRIPT>
<BODY
   BGCOLOR="oldlace" onLoad="alert('Hello, World.')"
   TEXT="navy">
<P><A NAME="numbers"><B>Some numbers</B></A>
<UL><LI>one
<LI>two
<LI>three
<LI>four</UL>
<P><A NAME="colors"><B>Some colors</B></A>
<UL><LI>red
<LI>orange
<LI>yellow
<LI>green</UL>
<P><A NAME="musicTypes"><B>Some music types</B></A>
<UL><LI>R&B
<LI>Jazz
<LI>Soul
<LI>Reggae</UL>
<P><A NAME="countries"><B>Some countries</B></A>
<UL><LI>Afghanistan
<LI>Brazil
<LI>Canada
<LI>Finland</UL>
</BODY>
</HTML>
```

See also Frame, window

alinkColor

A string specifying the color of an active link (after mouse-button down, but before mouse-button up).

Property of	document
Implemented in	JavaScript 1.0

Description The alinkColor property is expressed as a hexadecimal RGB triplet or as a string literal (see the *Client-Side JavaScript Guide*). This property is the JavaScript reflection of the ALINK attribute of the BODY tag.

If you express the color as a hexadecimal RGB triplet, you must use the format `rrggbb`. For example, the hexadecimal RGB values for salmon are `red=FA`, `green=80`, and `blue=72`, so the RGB triplet for salmon is `"FA8072"`.

Examples The following example sets the color of active links using a string literal:

```
document.alinkColor="aqua"
```

The following example sets the color of active links to aqua using a hexadecimal triplet:

```
document.alinkColor="00FFFF"
```

See also `document.bgColor`, `document.fgColor`, `document.linkColor`, `document.vlinkColor`

anchors

An array of objects corresponding to named anchors in source order.

Property of	`document`
Read-only	
Implemented in	JavaScript 1.0

Description You can refer to the `Anchor` objects in your code by using the `anchors` array. This array contains an entry for each `A` tag containing a `NAME` attribute in a document; these entries are in source order. For example, if a document contains three named anchors whose `NAME` attributes are `anchor1`, `anchor2`, and `anchor3`, you can refer to the anchors either as:

```
document.anchors["anchor1"]
document.anchors["anchor2"]
document.anchors["anchor3"]
```

or as:

```
document.anchors[0]
document.anchors[1]
document.anchors[2]
```

To obtain the number of anchors in a document, use the `length` property: `document.anchors.length`. If a document names anchors in a systematic way using natural numbers, you can use the `anchors` array and its `length` property to validate an anchor name before using it in operations such as setting `location.hash`.

applets

An array of objects corresponding to the applets in a document in source order.

Property of	`document`
Read-only	
Implemented in	JavaScript 1.1

Description You can refer to the applets in your code by using the `applets` array. This array contains an entry for each `Applet` object (`APPLET` tag) in a document; these entries are in source order. For example, if a document contains three applets whose `NAME` attributes are `app1`, `app2`, and `app3`, you can refer to the anchors either as:

```
document.applets["app1"]
document.applets["app2"]
document.applets["app3"]
```

or as:

```
document.applets[0]
document.applets[1]
document.applets[2]
```

To obtain the number of applets in a document, use the `length` property: `document.applets.length`.

bgColor

A string specifying the color of the document background.

Property of	`document`
Implemented in	JavaScript 1.0

Description The `bgColor` property is expressed as a hexadecimal RGB triplet or as a string literal (see the *Client-Side JavaScript Guide*). This property is the JavaScript reflection of the `BGCOLOR` attribute of the `BODY` tag. The default value of this property is set by the user with the preferences dialog box.

If you express the color as a hexadecimal RGB triplet, you must use the format `rrggbb`. For example, the hexadecimal RGB values for salmon are `red=FA`, `green=80`, and `blue=72`, so the RGB triplet for `salmon` is `"FA8072"`.

Examples The following example sets the color of the document background to aqua using a string literal:

```
document.bgColor="aqua"
```

The following example sets the color of the document background to aqua using a hexadecimal triplet:

```
document.bgColor="00FFFF"
```

See also `document.alinkColor`, `document.fgColor`, `document.linkColor`, `document.vlinkColor`

captureEvents

Sets the document to capture all events of the specified type.

Method of	`document`
Implemented in	JavaScript 1.2

Syntax `captureEvents(`*`eventType`*`)`

Parameters

`eventType`	The type of event to be captured. The available event types are listed with the `event` object.

Description When a window with frames wants to capture events in pages loaded from different locations (servers), you need to use `window.captureEvents` in a signed script and precede it with `window.enableExternalCapture`. For more information and an example, see `window.enableExternalCapture`.

`captureEvents` works in tandem with `releaseEvents`, `routeEvent`, and `handleEvent`. For more information on events, see the *Client-Side JavaScript Guide*.

classes

Creates a `Style` object that can specify the styles of HTML tags with a specific `CLASS` attribute.

Property of	`document`
Implemented in	JavaScript 1.2

Syntax `document.classes.`*`className`*`.`*`tagName`*

Parameters

`className`	The case-insensitive value of the `CLASS` attribute of the specified HTML tag in *`tagName`*.
`tagName`	The case-insensitive name of any HTML tag, such as `H1` or `BLOCKQUOTE`. If the value of *`tagName`* is `all`, *`tagName`* refers to all HTML tags.

Description Use the `classes` property to specify the style of HTML tags that have a specific `CLASS` attribute. For example, you can specify that the color of the `GreenBody` class of both the `P` or the `BLOCKQUOTE` tags is green. See the `Style` object for a description of the style properties you can specify for `classes`.

If you use the `classes` property within the `STYLE` tag (instead of within the `SCRIPT` tag), you can optionally omit `document` from the `classes` syntax. The `classes` property always applies to the current `document` object.

Examples This example sets the color of all tags using the `GreenBody` `CLASS` attribute to green:

```
<STYLE TYPE="text/javascript">
   classes.GreenBody.all.color="green"
</STYLE>
```

Notice that you can omit the document object within the `STYLE` tag. Within the `SCRIPT` tag, you must specify the document object as follows:

```
<SCRIPT LANGUAGE="JavaScript1.2">
   document.classes.GreenBody.all.color="green"
</SCRIPT>
```

In this example, text appearing within either of the following tags appears green:

```
<P CLASS="GreenBody">
<BLOCKQUOTE CLASS="GreenBody">
```

See also `document.contextual`, `document.ids`, `document.tags`, `Style`

close

Closes an output stream and forces data sent to layout to display.

Method of	`document`
Implemented in	JavaScript 1.0

Syntax `close()`

Parameters None.

Description The `close` method closes a stream opened with the `document.open` method. If the stream was opened to layout, the `close` method forces the content of the stream to display. Font style tags, such as `BIG` and `CENTER`, automatically flush a layout stream.

The `close` method also stops the "meteor shower" in the Netscape icon and displays Document: Done in the status bar.

Examples The following function calls `document.close` to close a stream that was opened with `document.open`. The `document.close` method forces the content of the stream to display in the window.

```
function windowWriter1() {
   var myString = "Hello, world!"
   msgWindow.document.open()
   msgWindow.document.write(myString + "<P>")
   msgWindow.document.close()
}
```

See also `document.open`, `document.write`, `document.writeln`

contextual

Uses contextual selection criteria to specify a `Style` object that can set the style of individual HTML tags.

Method of	`document`
Implemented in	JavaScript 1.2

Syntax `contextual(`*`context1`*`, ...[`*`contextN`*`,] `*`affectedStyle`*`)`

Parameters

`context1, ...[contextN]`	The `Style` objects, described by `document.classes` or `document.tags`, that establish the context for the affected `Style` object.
`affectedStyle`	The `Style` object whose style properties you want to change.

Description The `contextual` method provides a fine level of control for specifying styles. It lets you selectively apply a style to an HTML element that appears in a very specific context. For example, you can specify that the color of text within any `EM` tag that appears in an `H1` is blue.

You can further narrow the selection by specifying multiple contexts. For example, you can set the color of any `LI` tags with two or more `UL` parents by specifying `UL` for the first two contexts.

Examples **Example 1.** This example sets the color of text within any `EM` tag that appears in an `H1` to blue.

```
<STYLE TYPE="text/javascript">
   contextual(document.tags.H1, document.tags.EM).color="blue";
</STYLE>
```

Notice that you can omit the document object within the `STYLE` tag. Within the `SCRIPT` tag, you must specify the document object as follows:

```
<SCRIPT LANGUAGE="JavaScript1.2">
document.contextual(document.tags.H1, document.tags.EM).color="blue";
</SCRIPT>
```

In this example, text appearing within the `EM` tag is blue:

```
<H1 CLASS="Main">The following text is <EM>blue</EM></H1>
```

Example 2. This example sets the color of an `LI` element with two or more `UL` parents to red.

```
<STYLE TYPE="text/javascript">
   contextual(tags.UL, tags.UL, tags.LI).color="red";
</STYLE>
```

See also `document.classes`, `document.tags`, `Style`

cookie

String value representing all of the cookies associated with this document.

Property of	`document`
Implemented in	JavaScript 1.0

Security **JavaScript 1.1.** This property is tainted by default. For information on data tainting, see the *Client-Side JavaScript Guide*.

Description A cookie is a small piece of information stored by the web browser in the `cookies.txt` file. Use `string` methods such as `substring`, `charAt`, `indexOf`, and `lastIndexOf` to determine the value stored in the cookie. See Appendix C, "Netscape Cookies" for a complete specification of the cookie syntax.

You can set the `cookie` property at any time.

The `"expires="` component in the cookie file sets an expiration date for the cookie, so it persists beyond the current browser session. This date string is formatted as follows:

```
Wdy, DD-Mon-YY HH:MM:SS GMT
```

This format represents the following values:

- `Wdy` is a string representing the full name of the day of the week.
- `DD` is an integer representing the day of the month.
- `Mon` is a string representing the three-character abbreviation of the month.
- `YY` is an integer representing the last two digits of the year.
- `HH`, `MM`, and `SS` are 2-digit representations of hours, minutes, and seconds, respectively.

For example, a valid cookie expiration date is

```
expires=Wednesday, 09-Nov-99 23:12:40 GMT
```

The cookie date format is the same as the date returned by `toGMTString`, with the following exceptions:

- Dashes are added between the day, month, and year.
- The year is a 2-digit value for cookies.

Examples The following function uses the `cookie` property to record a reminder for users of an application. The cookie expiration date is set to one day after the date of the reminder.

```
function RecordReminder(time, expression) {
   // Record a cookie of the form "@<T>=<E>" to map
   // from <T> in milliseconds since the epoch,
   // returned by Date.getTime(), onto an encoded expression,
   // <E> (encoded to contain no white space, semicolon,
   // or comma characters)
   document.cookie = "@" + time + "=" + expression + ";"
   // set the cookie expiration time to one day
   // beyond the reminder time
   document.cookie += "expires=" + cookieDate(time + 24*60*60*1000)
   // cookieDate is a function that formats the date
   //according to the cookie spec
}
```

See also `Hidden`

domain

Specifies the domain name of the server that served a document.

Property of	`document`
Implemented in	JavaScript 1.1

Security **JavaScript 1.1.** This property is tainted by default. For information on data tainting, see the *Client-Side JavaScript Guide*.

Description **JavaScript 1.1.** The `domain` property lets scripts on multiple servers share properties when data tainting is not enabled. With tainting disabled, a script running in one window can read properties of another window only if both windows come from the same Web server. But large Web sites with multiple servers might need to share properties among servers. For example, a script on the host `www.royalairways.com` might need to share properties with a script on the host `search.royalairways.com`.

If scripts on two different servers change their `domain` property so that both scripts have the same domain name, both scripts can share properties. For example, a script loaded from `search.royalairways.com` could set its `domain` property to `"royalairways.com"`. A script from `www.royalairways.com` running in another window could also set its `domain` property to `"royalairways.com"`. Then, since both scripts have the domain `"royalairways.com"`, these two scripts can share properties, even though they did not originate from the same server.

You can change `domain` only in a restricted way. Initially, `domain` contains the hostname of the Web server from which the document was loaded. You can set `domain` only to a domain suffix of itself. For example, a script from `search.royalairways.com` can't set its `domain` property to `"search.royalairways"`. And a script from `IWantYourMoney.com` cannot set its domain to `"royalairways.com"`.

Once you change the `domain` property, you cannot change it back to its original value. For example, if you change `domain` from `"search.royalairways.com"` to `"royalairways.com"`, you cannot reset it to `"search.royalairways.com"`.

Examples The following statement changes the `domain` property to `"braveNewWorld.com"`. This statement is valid only if `"braveNewWorld.com"` is a suffix of the current domain, such as `"www.braveNewWorld.com"`.

```
document.domain="braveNewWorld.com"
```

embeds

An array containing an entry for each object embedded in the document.

Property of `document`

Read-only

Implemented in JavaScript 1.1

Description You can refer to embedded objects (created with the `EMBED` tag) in your code by using the `embeds` array. This array contains an entry for each `EMBED` tag in a document in source order. For example, if a document contains three embedded objects whose `NAME` attributes are `e1`, `e2`, and `e3`, you can refer to the objects either as:

```
document.embeds["e1"]
document.embeds["e2"]
document.embeds["e3"]
```

or as:

```
document.embeds[0]
document.embeds[1]
document.embeds[2]
```

To obtain the number of embedded objects in a document, use the `length` property: `document.embeds.length`.

Elements in the `embeds` array may have public callable functions, if they refer to a plug-in that uses LiveConnect. See the LiveConnect information in the *Client-Side JavaScript Guide*.

Use the elements in the `embeds` array to interact with the plug-in that is displaying the embedded object. If a plug-in is not Java-enabled, you cannot do anything with its element in the `embeds` array. The fields and methods of the elements in the `embeds` array vary from plug-in to plug-in; see the documentation supplied by the plug-in manufacturer.

When you use the `EMBED` tag to generate output from a plug-in application, you are not creating a `Plugin` object.

Examples The following code includes an audio plug-in in a document.

```
<EMBED SRC="train.au" HEIGHT=50 WIDTH=250>
```

See also `Plugin`

fgColor

A string specifying the color of the document (foreground) text.

Property of	`document`
Implemented in	JavaScript 1.0

Description The `fgColor` property is expressed as a hexadecimal RGB triplet or as a string literal (see the *Client-Side JavaScript Guide*). This property is the JavaScript reflection of the `TEXT` attribute of the `BODY` tag. The default value of this property is set by the user with the preferences dialog box You cannot set this property after the HTML source has been through layout.

If you express the color as a hexadecimal RGB triplet, you must use the format `rrggbb`. For example, the hexadecimal RGB values for salmon are `red=FA`, `green=80`, and `blue=72`, so the RGB triplet for `salmon` is `"FA8072"`.

You can override the value set in the `fgColor` property in either of the following ways:

- Setting the `COLOR` attribute of the `FONT` tag.
- Using the `fontcolor` method.

formName

Property of	`document`
Implemented in	JavaScript 1.1

The `document` object contains a separate property for each form in the document. The name of this property is the value of its `NAME` attribute. See `Hidden` for information on `Form` objects. You cannot add new forms to the document by creating new properties, but you can modify the form by modifying this object.

forms

An array containing an entry for each form in the document.

Property of	`document`
Read-only	
Implemented in	JavaScript 1.1

Security **JavaScript 1.1.** This property is tainted by default. For information on data tainting, see the *Client-Side JavaScript Guide*.

Description You can refer to the forms in your code by using the `forms` array (you can also use the form name). This array contains an entry for each `Form` object (`FORM` tag) in a document; these entries are in source order. For example, if a document contains three forms whose `NAME` attributes are `form1`, `form2`, and `form3`, you can refer to the objects in the `forms` array either as:

```
document.forms["form1"]
document.forms["form2"]
document.forms["form3"]
```

or as:

```
document.forms[0]
document.forms[1]
document.forms[2]
```

Additionally, the document object has a separate property for each named form, so you could refer to these forms also as:

```
document.form1
document.form2
document.form3
```

For example, you would refer to a `Text` object named `quantity` in the second form as `document.forms[1].quantity`. You would refer to the `value` property of this `Text` object as `document.forms[1].quantity.value`.

The value of each element in the `forms` array is `<object nameAttribute>`, where `nameAttribute` is the `NAME` attribute of the form.

To obtain the number of forms in a document, use the `length` property: `document.forms.length`.

getSelection

Returns a string containing the text of the current selection.

Method of	`document`
Implemented in	JavaScript 1.2

Syntax `getSelection()`

Description This method works only on the current document.

Security You cannot determine selected areas in another window.

Examples If you have a form with the following code and you click on the button, JavaScript displays an alert box containing the currently selected text from the window containing the button:

```
<INPUT TYPE="BUTTON" NAME="getstring"
   VALUE="Show highlighted text (if any)"
   onClick="alert('You have selected:\n'+document.getSelection());">
```

handleEvent

Invokes the handler for the specified event.

Method of	`document`
Implemented in	JavaScript 1.2

Syntax `handleEvent(`*`event`*`)`

Parameters

`event`	The name of an event for which the specified object has an event handler.

Description For information on handling events, see the *Client-Side JavaScript Guide*.

height

The height of a document, in pixels.

Property of	`document`
Implemented in	JavaScript 1.2

See also `document.width`

ids

Creates a `Style` object that can specify the style of individual HTML tags.

Property of	`document`
Implemented in	JavaScript 1.2

Syntax `document.ids.`*`idValue`*

Parameters

`idValue`	The case-insensitive value of the `ID` attribute of any HTML tag.

Description Use the `ids` property to specify the style of any HTML tag that has a specific `ID` attribute. For example, you can specify that the color of the `NewTopic` `ID` is green. See the `Style` object for a description of the style properties you can specify for `ids`.

The `ids` property is useful when you want to provide an exception to a class defined in the `document.classes` property.

If you use the `ids` property within the `STYLE` tag (instead of within the `SCRIPT` tag), you can optionally omit `document` from the `ids` syntax. The `ids` property always applies to the current `document` object.

Examples This example sets the Main `CLASS` attribute to 18-point bold green, but provides an exception for tags whose `ID` is `NewTopic`:

```
<STYLE TYPE="text/javascript">
   classes.Main.all.color="green"
   classes.Main.all.fontSize="18pt"
   classes.Main.all.fontWeight="bold"
   ids.NewTopic.color="blue"
</STYLE>
```

Notice that you can omit the document object within the `STYLE` tag. Within the `SCRIPT` tag, you must specify the document object as follows:

```
<SCRIPT LANGUAGE="JavaScript1.2">
   document.classes.Main.all.color="green"
   document.classes.Main.all.fontSize="18pt"
   document.classes.Main.all.fontWeight="bold"
   document.ids.NewTopic.color="blue"
</SCRIPT>
```

In this example, text appearing within the following tag is 18-point bold green:

```
<H1 CLASS="Main">Green head</H1>
```

However, text appearing within the following tag is 18-point bold blue:

```
<H1 CLASS="Main" ID="NewTopic">Blue head</H1>
```

See also `document.classes`, `document.contextual`, `document.tags`, `Style`

images

An array containing an entry for each image in the document.

Property of	`document`
Read-only	
Implemented in	JavaScript 1.1

You can refer to the images in a document by using the `images` array. This array contains an entry for each `Image` object (`IMG` tag) in a document; the entries are in source order. Images created with the `Image` constructor are not included in the `images` array. For example, if a document contains three images whose `NAME` attributes are `im1`, `im2`, and `im3`, you can refer to the objects in the `images` array either as:

```
document.images["im1"]
document.images["im2"]
document.images["im3"]
```

or as:

```
document.images[0]
document.images[1]
document.images[2]
```

To obtain the number of images in a document, use the `length` property: `document.images.length`.

lastModified

A string representing the date that a document was last modified.

Property of `document`

Read-only

Implemented in JavaScript 1.0

Security **JavaScript 1.1.** This property is tainted by default. For information on data tainting, see the *Client-Side JavaScript Guide*.

Description The `lastModified` property is derived from the HTTP header data sent by the web server. Servers generally obtain this date by examining the file's modification date.

The last modified date is not a required portion of the header, and some servers do not supply it. If the server does not return the last modified information, JavaScript receives a 0, which it displays as January 1, 1970 GMT. The following code checks the date returned by `lastModified` and prints out a value that corresponds to unknown.

```
lastmod = document.lastModified // get string of last modified date
lastmoddate = Date.parse(lastmod)// convert modified string to date
if(lastmoddate == 0){// unknown date (or January 1, 1970 GMT)
   document.writeln("Lastmodified: Unknown")
   } else {
   document.writeln("LastModified: " + lastmod)
}
```

Examples In the following example, the `lastModified` property is used in a `SCRIPT` tag at the end of an HTML file to display the modification date of the page:

```
document.write("This page updated on " + document.lastModified)
```

layers

The layers property is an array containing an entry for each layer within the document.

Property of	`document`
Implemented in	JavaScript 1.2

Description You can refer to the layers in your code by using the `layers` array. This array contains an entry for each `Layer` object (`LAYER` or `ILAYER` tag) in a document; these entries are in source order. For example, if a document contains three layers whose `NAME` attributes are `layer1`, `layer2`, and `layer3`, you can refer to the objects in the `layers` array either as:

```
document.layers["layer1"]
document.layers["layer2"]
document.layers["layer3"]
```

or as:

```
document.layers[0]
document.layers[1]
document.layers[2]
```

When accessed by integer index, array elements appear in z-order from back to front, where 0 is the bottommost layer and higher layers are indexed by consecutive integers. The index of a layer is not the same as its `zIndex` property, as the latter does not necessarily enumerate layers with consecutive integers. Adjacent layers can have the same `zIndex` property values.

These are valid ways of accessing layer objects:

```
document.layerName
document.layers[index]
document.layers["layerName"]
// example of using layers property to access nested layers:
document.layers["parentlayer"].layers["childlayer"]
```

Elements of a layers array are JavaScript objects that cannot be set by assignment, though their properties can be set. For example, the statement

```
document.layers[0]="music"
```

is invalid (and ignored) because it attempts to alter the `layers` array. However, the properties of the objects in the array readable and some are writable. For example, the statement

```
document.layers["suspect1"].left = 100;
```

is valid. This sets the layer's horizontal position to 100. The following example sets the background color to blue for the layer `bluehouse` which is nested in the layer `houses`.

```
document.layers["houses"].layers["bluehouse"].bgColor="blue";
```

To obtain the number of layers in a document, use the `length` property: `document.layers.length`.

linkColor

A string specifying the color of the document hyperlinks.

Property of	`document`
Implemented in	JavaScript 1.0

Description The `linkColor` property is expressed as a hexadecimal RGB triplet or as a string literal (see the *Client-Side JavaScript Guide*). This property is the JavaScript reflection of the `LINK` attribute of the `BODY` tag. The default value of this property is set by the user with the preferences dialog box. You cannot set this property after the HTML source has been through layout.

If you express the color as a hexadecimal RGB triplet, you must use the format `rrggbb`. For example, the hexadecimal RGB values for salmon are `red=FA`, `green=80`, and `blue=72`, so the RGB triplet for salmon is `"FA8072"`.

Examples The following example sets the color of document links to aqua using a string literal:

```
document.linkColor="aqua"
```

The following example sets the color of document links to aqua using a hexadecimal triplet:

```
document.linkColor="00FFFF"
```

See also `document.alinkColor`, `document.bgColor`, `document.fgColor`, `document.vlinkColor`

links

An array of objects corresponding to `Area` and `Link` objects in source order.

Property of	`document`
Read-only	
Implemented in	JavaScript 1.0

Security **JavaScript 1.1.** This property is tainted by default. For information on data tainting, see the *Client-Side JavaScript Guide*.

Description You can refer to the `Area` and `Link` objects in your code by using the `links` array. This array contains an entry for each `Area` (`<AREA HREF="...">` tag) and `Link` (`<A HREF="...">` tag) object in a document in source order. It also contains links created with the `link` method. For example, if a document contains three links, you can refer to them as:

```
document.links[0]
document.links[1]
document.links[2]
```

To obtain the number of links in a document, use the `length` property: `document.links.length`.

open

Opens a stream to collect the output of `write` or `writeln` methods.

Method of	`document`
Implemented in	JavaScript 1.0 JavaScript 1.1: added `"replace"` parameter; `document.open()` or `document.open("text/html")` clears the current document if it has finished loading

Syntax `open([`*`mimeType`*`, [`*`replace`*`]])`

Parameters

`mimeType`	A string specifying the type of document to which you are writing. If you do not specify `mimeType`, `text/html` is the default.
`replace`	The string `"replace"`. If you supply this parameter, `mimeType` must be `"text/html"`. Causes the new document to reuse the history entry that the previous document used.

Description Sample values for `mimeType` are:

- `text/html` specifies a document containing ASCII text with HTML formatting.
- `text/plain` specifies a document containing plain ASCII text with end-of-line characters to delimit displayed lines.
- `image/gif` specifies a document with encoded bytes constituting a GIF header and pixel data.
- `image/jpeg` specifies a document with encoded bytes constituting a JPEG header and pixel data.
- `image/x-bitmap` specifies a document with encoded bytes constituting a bitmap header and pixel data.
- `plugIn` loads the specified plug-in and uses it as the destination for `write` and `writeln` methods. For example, `"x-world/vrml"` loads the VR Scout VRML plug-in from Chaco Communications, and `"application/x-director"` loads the Macromedia Shockwave plug-in. Plug-in MIME types are only valid if the user has installed the required plug-in software.

The `open` method opens a stream to collect the output of `write` or `writeln` methods. If the `mimeType` is `text` or `image`, the stream is opened to layout; otherwise, the stream is opened to a plug-in. If a document exists in the target window, the `open` method clears it.

End the stream by using the `document.close` method. The `close` method causes text or images that were sent to layout to display. After using `document.close`, call `document.open` again when you want to begin another output stream.

In JavaScript 1.1 and later, `document.open` or `document.open("text/html")` clears the current document if it has finished loading. This is because this type of `open` call writes a default `<BASE HREF=>` tag so you can generate relative URLs based on the generating script's document base.

The `"replace"` keyword causes the new document to reuse the history entry that the previous document used. When you specify `"replace"` while opening a document, the target window's history length is not incremented even after you write and close.

`"replace"` is typically used on a window that has a blank document or an `"about:blank"` URL. After `"replace"` is specified, the `write` method typically generates HTML for the window, replacing the history entry for the blank URL. Take care when using generated HTML on a window with a blank URL. If you do not specify `"replace"`, the generated HTML has its own history entry, and the user can press the Back button and back up until the frame is empty.

After `document.open("text/html","replace")` executes, `history.current` for the target window is the URL of document that executed `document.open`.

Examples

Example 1. The following function calls `document.open` to open a stream before issuing a `write` method:

```
function windowWriter1() {
   var myString = "Hello, world!"
   msgWindow.document.open()
   msgWindow.document.write("<P>" + myString)
   msgWindow.document.close()
}
```

Example 2. The following function calls `document.open` with the `"replace"` keyword to open a stream before issuing `write` methods. The HTML code in the `write` methods is written to `msgWindow`, replacing the current history entry. The history length of `msgWindow` is not incremented.

```
function windowWriter2() {
   var myString = "Hello, world!"
   msgWindow.document.open("text/html","replace")
   msgWindow.document.write("<P>" + myString)
   msgWindow.document.write("<P>history.length is " +
      msgWindow.history.length)
   msgWindow.document.close()
}
```

The following code creates the `msgWindow` window and calls the function:

```
msgWindow=window.open('','',
   'toolbar=yes,scrollbars=yes,width=400,height=300')
windowWriter2()
```

Example 3. In the following example, the `probePlugIn` function determines whether a user has the Shockwave plug-in installed:

```
function probePlugIn(mimeType) {
   var havePlugIn = false
   var tiny = window.open("", "teensy", "width=1,height=1")
   if (tiny != null) {
      if (tiny.document.open(mimeType) != null)
         havePlugIn = true
      tiny.close()
   }
   return havePlugIn
}
```

```
var haveShockwavePlugIn = probePlugIn("application/x-director")
```

See also `document.close`, `document.write`, `document.writeln`, `Location.reload`, `Location.replace`

plugins

An array of objects corresponding to `Plugin` objects in source order.

Property of	`document`
Read-only	
Implemented in	JavaScript 1.1

You can refer to the `Plugin` objects in your code by using the `plugins` array. This array contains an entry for each `Plugin` object in a document in source order. For example, if a document contains three plugins, you can refer to them as:

```
document.plugins[0]
document.plugins[1]
document.plugins[2]
```

referrer

Specifies the URL of the calling document when a user clicks a link.

Property of	`document`
Read-only	
Implemented in	JavaScript 1.0

Security **JavaScript 1.1.** This property is tainted by default. For information on data tainting, see the *Client-Side JavaScript Guide*.

Description When a user navigates to a destination document by clicking a `Link` object on a source document, the `referrer` property contains the URL of the source document.

`referrer` is empty if the user typed a URL in the Location box, or used some other means to get to the current URL. `referrer` is also empty if the server does not provide environment variable information.

Examples In the following example, the `getReferrer` function is called from the destination document. It returns the URL of the source document.

```
function getReferrer() {
   return document.referrer
}
```

releaseEvents

Sets the document to release captured events of the specified type, sending the event to objects further along the event hierarchy.

Method of	`document`
Implemented in	JavaScript 1.2

Note If the original target of the event is a window, the window receives the event even if it is set to release that type of event.

Syntax `releaseEvents(`*`eventType`*`)`

Parameters

`eventType`	Type of event to be captured.

Description `releaseEvents` works in tandem with `captureEvents`, `routeEvent`, and `handleEvent`. For more information on events, see the *Client-Side JavaScript Guide*.

routeEvent

Passes a captured event along the normal event hierarchy.

Method of	`document`
Implemented in	JavaScript 1.2

Syntax `routeEvent(`*`event`*`)`

Parameters

`event`	Name of the event to be routed.

Description If a sub-object (document or layer) is also capturing the event, the event is sent to that object. Otherwise, it is sent to its original target.

`routeEvent` works in tandem with `captureEvents`, `releaseEvents`, and `handleEvent`. For more information on events, see the *Client-Side JavaScript Guide*.

tags

Creates a `Style` object that can specify the styles of HTML tags.

Property of	`document`
Implemented in	JavaScript 1.2

Syntax `document.tags.`*`tagName`*

Parameters

`tagName`	The case-insensitive name of any HTML tag, such as `H1` or `BLOCKQUOTE`.

Description Use the `tags` property to specify the style of HTML tags. For example, you can specify that the color of any `H1` tag is blue, and that the alignment of any `H1` or `H2` tag is centered. See the `Style` object for a description of the properties you can specify for HTML tags.

Because all HTML elements inherit from the BODY tag, you can specify a default document style by setting the style properties of BODY.

If you use the tags property within the STYLE tag (instead of within the SCRIPT tag), you can optionally omit document from the tags syntax. The tags property always applies to the current document object.

Examples **Example 1.** This example sets the color of all H1 tags to blue:

```
<STYLE TYPE="text/javascript">
   tags.H1.color="blue"
</STYLE>
```

Notice that you can omit the document object within the STYLE tag. Within the SCRIPT tag, you must specify the document object as follows:

```
<SCRIPT LANGUAGE="JavaScript1.2">
   document.tags.H1.color="blue"
</SCRIPT>
```

Example 2. This example sets a universal left margin for a document:

```
document.tags.Body.marginLeft="20pt"
```

Because all HTML tags inherit from BODY, this example sets the left margin for the entire document to 20 points.

See also document.classes, document.contextual, document.ids, Style

title

A string representing the title of a document.

Property of	document
Read-only	
Implemented in	JavaScript 1.0

Security **JavaScript 1.1.** This property is tainted by default. For information on data tainting, see the *Client-Side JavaScript Guide*.

Description The title property is a reflection of the value specified between the TITLE start and end tags. If a document does not have a title, the title property is null.

Examples In the following example, the value of the `title` property is assigned to a variable called `docTitle`:

```
var newWindow = window.open("http://home.netscape.com")
var docTitle = newWindow.document.title
```

URL

A string specifying the complete URL of the document.

Property of	`document`
Read-only	
Implemented in	JavaScript 1.0

Security **JavaScript 1.1.** This property is tainted by default. For information on data tainting, see the *Client-Side JavaScript Guide*.

Description `URL` is a string-valued property containing the full URL of the document. It usually matches what `window.location.href` is set to when you load the document, but redirection may change `location.href`.

Examples The following example displays the URL of the current document:

```
document.write("The current URL is " + document.URL)
```

See also `Location.href`

vlinkColor

A string specifying the color of visited links.

Property of	`document`
Implemented in	JavaScript 1.0

Description The `vlinkColor` property is expressed as a hexadecimal RGB triplet or as a string literal (see the *Client-Side JavaScript Guide*). This property is the JavaScript reflection of the `VLINK` attribute of the `BODY` tag. The default value of this property is set by the user with the preferences dialog box. You cannot set this property after the HTML source has been through layout.

If you express the color as a hexadecimal RGB triplet, you must use the format `rrggbb`. For example, the hexadecimal RGB values for salmon are `red=FA`, `green=80`, and `blue=72`, so the RGB triplet for salmon is `"FA8072"`.

Examples The following example sets the color of visited links to aqua using a string literal:

```
document.vlinkColor="aqua"
```

The following example sets the color of active links to aqua using a hexadecimal triplet:

```
document.vlinkColor="00FFFF"
```

See also `document.alinkColor`, `document.bgColor`, `document.fgColor`, `document.linkColor`

width

The width of a document, in pixels.

Property of	`document`
Implemented in	JavaScript 1.2

See also `document.height`

write

Writes one or more HTML expressions to a document in the specified window.

Method of	`document`
Implemented in	JavaScript 1.0

Syntax `document.write(`*`expr1`*`[, ...,`*`exprN`*`])`

Parameters

`expr1, ... exprN`	Any JavaScript expressions.

Description The `write` method displays any number of expressions in the document window. You can specify any JavaScript expression with the `write` method, including numeric, string, or logical expressions.

The `write` method is the same as the `writeln` method, except the `write` method does not append a newline character to the end of the output.

Use the `write` method within any `SCRIPT` tag or within an event handler. Event handlers execute after the original document closes, so the `write` method implicitly opens a new document of `mimeType` `text/html` if you do not explicitly issue a `document.open` method in the event handler.

You can use the `write` method to generate HTML and JavaScript code. However, the HTML parser reads the generated code as it is being written, so you might have to escape some characters. For example, the following `write` method generates a comment and writes it to `window2`:

```
window2=window.open('','window2')
beginComment="\<!--"
endComment="--\>"
window2.document.write(beginComment)
window2.document.write(" This some text inside a comment. ")
window2.document.write(endComment)
```

Printing, saving, and viewing generated HTML. In Navigator 3.0 and later, users can print and save generated HTML using the commands on the File menu.

If you choose Page Source from the Navigator View menu or View Frame Source from the right-click menu, the web browser displays the content of the HTML file with the generated HTML. (This is what would be displayed using a `wysiwyg:` URL.)

If you instead want to view the HTML source showing the scripts which generate HTML (with the `document.write` and `document.writeln` methods), do not use the Page Source or View Frame Source menu items. In this situation, use the `view-source:` protocol.

For example, assume the file `file://c|/test.html` contains this text:

```
<HTML>
<BODY>
Hello,
<SCRIPT>document.write(" there.")</SCRIPT>
</BODY>
</HTML>
```

If you load this URL into the web browser, it displays the following:

```
Hello, there.
```

If you choose View Document Source, the browser displays:

```
<HTML>
<BODY>
Hello,
 there.
</BODY>
</HTML>
```

If you load `view-source:file://c|/test.html`, the browser displays:

```
<HTML>
<BODY>
Hello,
<SCRIPT>document.write(" there.")</SCRIPT>
</BODY>
</HTML>
```

For information on specifying the `view-source:` protocol in the `location` object, see the `Location` object.

Examples In the following example, the `write` method takes several arguments, including strings, a numeric, and a variable:

```
var mystery = "world"
// Displays Hello world testing 123
msgWindow.document.write("Hello ", mystery, " testing ", 123)
```

In the following example, the `write` method takes two arguments. The first argument is an assignment expression, and the second argument is a string literal.

```
//Displays Hello world...
msgWindow.document.write(mystr = "Hello ", "world...")
```

In the following example, the `write` method takes a single argument that is a conditional expression. If the value of the variable `age` is less than 18, the method displays “Minor.” If the value of `age` is greater than or equal to 18, the method displays “Adult.”

```
msgWindow.document.write(status = (age >= 18) ? "Adult" : "Minor")
```

See also `document.close`, `document.open`, `document.writeln`

writeln

Writes one or more HTML expressions to a document in the specified window and follows them with a newline character.

Method of	`document`
Implemented in	JavaScript 1.0

Syntax `writeln(`*`expr1`*`[, ... `*`exprN`*`])`

Parameters

`expr1, ... exprN`	Any JavaScript expressions.

Description The `writeln` method displays any number of expressions in a document window. You can specify any JavaScript expression, including numeric, string, or logical expressions.

The `writeln` method is the same as the `write` method, except the `writeln` method appends a newline character to the end of the output. HTML ignores the newline character, except within certain tags such as the `PRE` tag.

Use the `writeln` method within any `SCRIPT` tag or within an event handler. Event handlers execute after the original document closes, so the `writeln` method will implicitly open a new document of `mimeType text/html` if you do not explicitly issue a `document.open` method in the event handler.

In Navigator 3.0 and later, users can print and save generated HTML using the commands on the File menu.

Examples All the examples used for the `write` method are also valid with the `writeln` method.

See also `document.close`, `document.open`, `document.write`

event

The `event` object contains properties that describe a JavaScript event, and is passed as an argument to an event handler when the event occurs.

Client-side object

Implemented in	JavaScript 1.2

In the case of a mouse-down event, for example, the `event` object contains the type of event (in this case MouseDown), the x and y position of the cursor at the time of the event, a number representing the mouse button used, and a field containing the modifier keys (Control, Alt, Meta, or Shift) that were depressed at the time of the event. The properties used within the `event` object vary from one type of event to another. This variation is provided in the descriptions of individual event handlers.

See Chapter 3, "Event Handlers," for complete information about event handlers. For more information on handling events, see the *Client-Side JavaScript Guide*.

Created by `event` objects are created by Communicator when an event occurs. You do not create them yourself.

Security Setting any property of this object requires the `UniversalBrowserWrite` privilege. In addition, getting the `data` property of the `DragDrop` event requires the `UniversalBrowserRead` privilege. For information on security, see the *Client-Side JavaScript Guide*.

Property Summary Not all of these properties are relevant to each event type. To learn which properties are used by an event, see the "Event object properties used" section of the individual event handler.

Property	Description
`data`	Returns an array of strings containing the URLs of the dropped objects. Passed with the DragDrop event.
`height`	Represents the height of the window or frame.
`layerX`	Number specifying either the object width when passed with the resize event, or the cursor's horizontal position in pixels relative to the layer in which the event occurred. Note that `layerX` is synonymous with x.

Property	Description
`layerY`	Number specifying either the object height when passed with the resize event, or the cursor's vertical position in pixels relative to the layer in which the event occurred. Note that `layerY` is synonymous with y.
`modifiers`	String specifying the modifier keys associated with a mouse or key event. Modifier key values are: ALT_MASK, CONTROL_MASK, SHIFT_MASK, and META_MASK.
`pageX`	Number specifying the cursor's horizontal position in pixels, relative to the page.
`pageY`	Number specifying the cursor's vertical position in pixels relative to the page.
`screenX`	Number specifying the cursor's horizontal position in pixels, relative to the screen.
`screenY`	Number specifying the cursor's vertical position in pixels, relative to the screen.
`target`	String representing the object to which the event was originally sent. (All events)
`type`	String representing the event type. (All events)
`which`	Number specifying either the mouse button that was pressed or the ASCII value of a pressed key. For a mouse, 1 is the left button, 2 is the middle button, and 3 is the right button.
`width`	Represents the width of the window or frame.
`x`	Synonym for `layerX`.
`y`	Synonym for `layerY`.

Method Summary This object inherits the `watch` and `unwatch` methods from `Object`.

Examples The following example uses the event object to provide the type of event to the alert message.

```
<A HREF="http://home.netscape.com" onClick='alert("Link got an event: "
+ event.type)'>Click for link event</A>
```

The following example uses the event object in an explicitly called event handler.

```
<SCRIPT>
function fun1(evnt) {
   alert ("Document got an event: " + evnt.type);
   alert ("x position is " + evnt.layerX);
   alert ("y position is " + evnt.layerY);
   if (evnt.modifiers & Event.ALT_MASK)
      alert ("Alt key was down for event.");
   return true;
   }
document.onmousedown = fun1;
</SCRIPT>
```

data

For the DragDrop event, returns an array of strings containing the URLs of the dropped objects.

Property of	`event`
Implemented in	JavaScript 1.2

Security Setting this property requires the `UniversalBrowserWrite` privilege. In addition, getting this property for the `DragDrop` event requires the `UniversalBrowserRead` privilege. For information on security, see the *Client-Side JavaScript Guide*.

height

Represents the height of the window or frame.

Property of	`event`
Implemented in	JavaScript 1.2

Security Setting this property requires the `UniversalBrowserWrite` privilege. For information on security, see the *Client-Side JavaScript Guide*.

See also `event.width`

layerX

Number specifying either the object width when passed with the resize event, or the cursor's horizontal position in pixels relative to the layer in which the event occurred.

Property of	`event`
Implemented in	JavaScript 1.2

Security Setting this property requires the `UniversalBrowserWrite` privilege. For information on security, see the *Client-Side JavaScript Guide*.

Description This property is synonymous with the `event.x` property.

See also `event.layerY`

layerY

Number specifying either the object height when passed with the resize event, or the cursor's vertical position in pixels relative to the layer in which the event occurred.

Property of	`event`
Implemented in	JavaScript 1.2

Security Setting this property requires the `UniversalBrowserWrite` privilege. For information on security, see the *Client-Side JavaScript Guide*.

Description This property is synonymous with the `event.y` property.

See also `event.layerX`

modifiers

String specifying the modifier keys associated with a mouse or key event. Modifier key values are: ALT_MASK, CONTROL_MASK, SHIFT_MASK, and META_MASK.

Property of	`event`
Implemented in	JavaScript 1.2

Security Setting this property requires the `UniversalBrowserWrite` privilege. For information on security, see the *Client-Side JavaScript Guide*.

See also `event.which`

pageX

Number specifying the cursor's horizontal position in pixels, relative to the page.

Property of	`event`
Implemented in	JavaScript 1.2

Security Setting this property requires the `UniversalBrowserWrite` privilege. For information on security, see the *Client-Side JavaScript Guide*.

See also `event.pageY`

pageY

Number specifying the cursor's vertical position in pixels relative to the page.

Property of	`event`
Implemented in	JavaScript 1.2

Security Setting this property requires the `UniversalBrowserWrite` privilege. For information on security, see the *Client-Side JavaScript Guide*.

See also `event.pageX`

screenX

Number specifying the cursor's horizontal position in pixels, relative to the screen.

Property of	`event`
Implemented in	JavaScript 1.2

Security Setting this property requires the `UniversalBrowserWrite` privilege. For information on security, see the *Client-Side JavaScript Guide*.

See also `event.screenY`

screenY

Number specifying the cursor's vertical position in pixels, relative to the screen.

Property of	`event`
Implemented in	JavaScript 1.2

Security Setting this property requires the `UniversalBrowserWrite` privilege. For information on security, see the *Client-Side JavaScript Guide*.

See also `event.screenX`

target

String representing the object to which the event was originally sent.

Property of	`event`
Implemented in	JavaScript 1.2

Security Setting this property requires the `UniversalBrowserWrite` privilege. For information on security, see the *Client-Side JavaScript Guide*.

See also `event.type`

type

String representing the event type.

Property of	`event`
Implemented in	JavaScript 1.2

Security Setting this property requires the `UniversalBrowserWrite` privilege. For information on security, see the *Client-Side JavaScript Guide*.

See also `event.target`

which

Number specifying either the mouse button that was pressed or the ASCII value of a pressed key. For a mouse, 1 is the left button, 2 is the middle button, and 3 is the right button.

Property of	`event`
Implemented in	JavaScript 1.2

Security Setting this property requires the `UniversalBrowserWrite` privilege. For information on security, see the *Client-Side JavaScript Guide*.

See also `event.modifiers`

width

Represents the width of the window or frame.

Property of	`event`
Implemented in	JavaScript 1.2

Security Setting this property requires the `UniversalBrowserWrite` privilege. For information on security, see the *Client-Side JavaScript Guide*.

See also `event.height`

x

Number specifying either the object width when passed with the resize event, or the cursor's horizontal position in pixels relative to the layer in which the event occurred.

Property of	`event`
Implemented in	JavaScript 1.2

Security Setting this property requires the `UniversalBrowserWrite` privilege. For information on security, see the *Client-Side JavaScript Guide*.

Description This property is synonymous with the `event.layerX` property.

See also `event.y`

y

Synonym for `layerY`.

Property of	`event`
Implemented in	JavaScript 1.2

Security Setting this property requires the `UniversalBrowserWrite` privilege. For information on security, see the *Client-Side JavaScript Guide*.

Description This property is synonymous with the `event.layerY` property.

See also `event.x`

FileUpload

A file upload element on an HTML form. A file upload element lets the user supply a file as input.

Client-side object

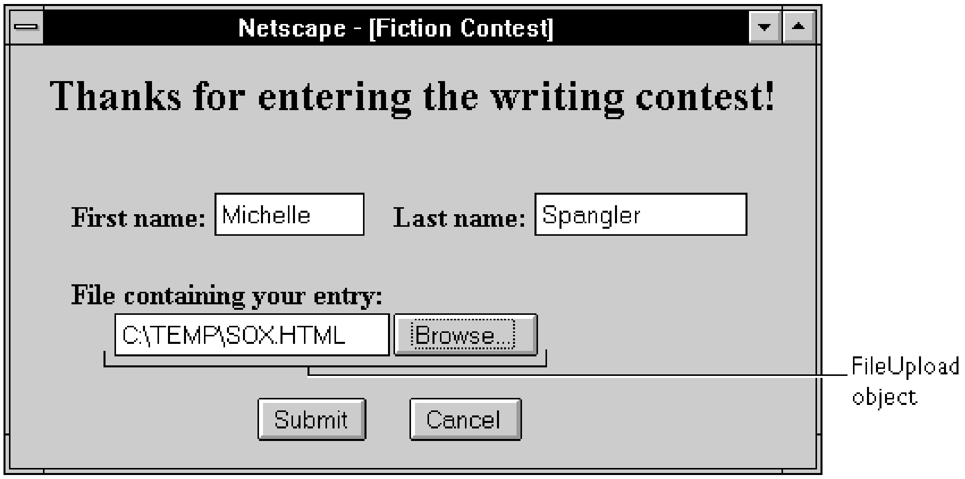
Implemented in	JavaScript 1.0
	JavaScript 1.1: added `type` property
	JavaScript 1.2: added `handleEvent` method.

Created by The HTML `INPUT` tag, with `"file"` as the value of the `TYPE` attribute. For a given form, the JavaScript runtime engine creates appropriate `FileUpload` objects and puts these objects in the `elements` array of the corresponding `Form` object. You access a `FileUpload` object by indexing this array. You can index the array either by number or, if supplied, by using the value of the `NAME` attribute.

Event handlers

- `onBlur`
- `onChange`
- `onFocus`

Description A `FileUpload` object on a form looks as follows:


Netscape - [Fiction Contest]
Thanks for entering the writing contest!
First name:
Michelle
Last name:
Spangler
File containing your entry:
C:\TEMP\SOX.HTML
Browse...
Submit
Cancel
FileUpload object

A `FileUpload` object is a form element and must be defined within a `FORM` tag.

Property Summary

Property	Description
`form`	Specifies the form containing the `FileUpload` object.
`name`	Reflects the `NAME` attribute.
`type`	Reflects the `TYPE` attribute.
`value`	Reflects the current value of the file upload element's field; this corresponds to the name of the file to upload.

Method Summary

Method	Description
`blur`	Removes focus from the object.
`focus`	Gives focus to the object.
`handleEvent`	Invokes the handler for the specified event.
`select`	Selects the input area of the file upload field.

In addition, this object inherits the `watch` and `unwatch` methods from `Object`.

Examples The following example places a `FileUpload` object on a form and provides two buttons that let the user display current values of the `name` and `value` properties.

```
<FORM NAME="form1">
File to send: <INPUT TYPE="file" NAME="myUploadObject">
<P>Get properties<BR>
<INPUT TYPE="button" VALUE="name"
   onClick="alert('name: ' + document.form1.myUploadObject.name)">
<INPUT TYPE="button" VALUE="value"
   onClick="alert('value: ' +
document.form1.myUploadObject.value)"><BR>
</FORM>
```

See also `Text`

blur

Removes focus from the object.

Method of	`FileUpload`
Implemented in	JavaScript 1.0

Syntax `blur()`

Parameters None

See also `FileUpload.focus`, `FileUpload.select`

focus

Navigates to the `FileUpload` field and give it focus.

Method of	`FileUpload`
Implemented in	JavaScript 1.0

Syntax `focus()`

Parameters None

See also `FileUpload.blur`, `FileUpload.select`

form

An object reference specifying the form containing the object.

Property of	`FileUpload`
Read-only	
Implemented in	JavaScript 1.0

Description Each form element has a `form` property that is a reference to the element's parent form. This property is especially useful in event handlers, where you might need to refer to another element on the current form.

handleEvent

Invokes the handler for the specified event.

Syntax `handleEvent(`*`event`*`)`

Method of	`FileUpload`
Implemented in	JavaScript 1.2

Parameters

`event`	The name of an event for which the object has an event handler.

Description For information on handling events, see the *Client-Side JavaScript Guide*.

name

A string specifying the name of this object.

Property of	`FileUpload`
Read-only	
Implemented in	JavaScript 1.0

Security **JavaScript 1.1.** This property is tainted by default. For information on data tainting, see the *Client-Side JavaScript Guide*.

Description The `name` property initially reflects the value of the `NAME` attribute. The `name` property is not displayed on-screen; it is used to refer to the objects programmatically.

If multiple objects on the same form have the same `NAME` attribute, an array of the given name is created automatically. Each element in the array represents an individual `Form` object. Elements are indexed in source order starting at 0. For example, if two `Text` elements and a `FileUpload` element on the same form have their `NAME` attribute set to `"myField"`, an array with the elements `myField[0]`, `myField[1]`, and `myField[2]` is created. You need to be aware of this situation in your code and know whether `myField` refers to a single element or to an array of elements.

Examples In the following example, the `valueGetter` function uses a `for` loop to iterate over the array of elements on the `valueTest` form. The `msgWindow` window displays the names of all the elements on the form:

```
newWindow=window.open("http://home.netscape.com")

function valueGetter() {
   var msgWindow=window.open("")
   for (var i = 0; i < newWindow.document.valueTest.elements.length; i++) {
      msgWindow.document.write(newWindow.document.valueTest.elements[i].name + "<BR>")
   }
}
```

select

Selects the input area of the file upload field.

Method of	`FileUpload`
Implemented in	JavaScript 1.0

Syntax `select()`

Parameters None

Description Use the `select` method to highlight the input area of a file upload field. You can use the `select` method with the `focus` method to highlight a field and position the cursor for a user response. This makes it easy for the user to replace all the text in the field.

See also `FileUpload.blur`, `FileUpload.focus`

type

For all `FileUpload` objects, the value of the `type` property is `"file"`. This property specifies the form element's type.

Property of	`FileUpload`
Read-only	
Implemented in	JavaScript 1.1

Examples The following example writes the value of the `type` property for every element on a form.

```
for (var i = 0; i < document.form1.elements.length; i++) {
   document.writeln("<BR>type is " + document.form1.elements[i].type)
}
```

value

A string that reflects the `VALUE` attribute of the object.

Property of	`FileUpload`
Read-only	
Implemented in	JavaScript 1.0

Security Setting a file upload widget requires the UniversalFileRead privilege. For information on security, see the *Client-Side JavaScript Guide*.

JavaScript 1.1. This property is tainted by default. For information on data tainting, see the *Client-Side JavaScript Guide*.

Description Use the `value` property to obtain the file name that the user typed into a `FileUpload` object.

Form

Lets users input text and make choices from `Form` elements such as checkboxes, radio buttons, and selection lists. You can also use a form to post data to a server.

Client-side object

Implemented in	JavaScript 1.0
	JavaScript 1.1: added `reset` method.
	JavaScript 1.2: added `handleEvent` method.

Created by The HTML `FORM` tag. The JavaScript runtime engine creates a `Form` object for each `FORM` tag in the document. You access `FORM` objects through the `document.forms` property and through named properties of that object.

To define a form, use standard HTML syntax with the addition of JavaScript event handlers. If you supply a value for the `NAME` attribute, you can use that value to index into the `forms` array. In addition, the associated `document` object has a named property for each named form.

Event handlers

- `onReset`
- `onSubmit`

Description Each form in a document is a distinct object. You can refer to a form's elements in your code by using the element's name (from the `NAME` attribute) or the `Form.elements` array. The `elements` array contains an entry for each element (such as a `Checkbox`, `Radio`, or `Text` object) in a form.

If multiple objects on the same form have the same `NAME` attribute, an array of the given name is created automatically. Each element in the array represents an individual `Form` object. Elements are indexed in source order starting at 0. For example, if two `Text` elements and a `Textarea` element on the same form have their `NAME` attribute set to `"myField"`, an array with the elements `myField[0]`, `myField[1]`, and `myField[2]` is created. You need to be aware of this situation in your code and know whether `myField` refers to a single element or to an array of elements.

Property Summary

Property	Description
`action`	Reflects the `ACTION` attribute.
`elements`	An array reflecting all the elements in a form.
`encoding`	Reflects the `ENCTYPE` attribute.
`length`	Reflects the number of elements on a form.
`method`	Reflects the `METHOD` attribute.
`name`	Reflects the `NAME` attribute.
`target`	Reflects the `TARGET` attribute.

Method Summary

Method	Description
`handleEvent`	Invokes the handler for the specified event.
`reset`	Simulates a mouse click on a reset button for the calling form.
`submit`	Submits a form.

In addition, this object inherits the `watch` and `unwatch` methods from `Object`.

Examples **Example 1: Named form.** The following example creates a form called `myForm` that contains text fields for first name and last name. The form also contains two buttons that change the names to all uppercase or all lowercase. The function `setCase` shows how to refer to the form by its name.

```
<HTML>
<HEAD>
<TITLE>Form object example</TITLE>
</HEAD>
<SCRIPT>
function setCase (caseSpec){
if (caseSpec == "upper") {
   document.myForm.firstName.value=document.myForm.firstName.value.toUpperCase()
   document.myForm.lastName.value=document.myForm.lastName.value.toUpperCase()}
else {
   document.myForm.firstName.value=document.myForm.firstName.value.toLowerCase()
   document.myForm.lastName.value=document.myForm.lastName.value.toLowerCase()}
}
</SCRIPT>
```

```
<BODY>
<FORM NAME="myForm">
<B>First name:</B>
<INPUT TYPE="text" NAME="firstName" SIZE=20>
<BR><B>Last name:</B>
<INPUT TYPE="text" NAME="lastName" SIZE=20>
<P><INPUT TYPE="button" VALUE="Names to uppercase" NAME="upperButton"
   onClick="setCase('upper')">
<INPUT TYPE="button" VALUE="Names to lowercase" NAME="lowerButton"
   onClick="setCase('lower')">
</FORM>
</BODY>
</HTML>
```

Example 2: forms array. The `onLoad` event handler in the following example displays the name of the first form in an Alert dialog box.

```
<BODY onLoad="alert('You are looking at the ' + document.forms[0] + '
form!')">
```

If the form name is `musicType`, the alert displays the following message:

```
You are looking at the <object musicType> form!
```

Example 3: onSubmit event handler. The following example shows an `onSubmit` event handler that determines whether to submit a form. The form contains one `Text` object where the user enters three characters. `onSubmit` calls a function, `checkData`, that returns true if there are 3 characters; otherwise, it returns false. Notice that the form's `onSubmit` event handler, not the submit button's `onClick` event handler, calls the `checkData` function. Also, the `onSubmit` handler contains a `return` statement that returns the value obtained with the function call; this prevents the form from being submitted if invalid data is specified. See `onSubmit` for more information.

```
<HTML>
<HEAD>
<TITLE>Form object/onSubmit event handler example</TITLE>
<TITLE>Form object example</TITLE>
</HEAD>
<SCRIPT>
var dataOK=false
function checkData (){
if (document.myForm.threeChar.value.length == 3) {
   return true}
   else {
      alert("Enter exactly three characters. " + document.myForm.threeChar.value +
         " is not valid.")
      return false}
}
```

```
</SCRIPT>
<BODY>
<FORM NAME="myForm" onSubmit="return checkData()">
<B>Enter 3 characters:</B>
<INPUT TYPE="text" NAME="threeChar" SIZE=3>
<P><INPUT TYPE="submit" VALUE="Done" NAME="submit1"
onClick="document.myForm.threeChar.value=document.myForm.threeChar.value.toUpperCase()">
</FORM>
</BODY>
</HTML>
```

Example 4: submit method. The following example is similar to the previous one, except it submits the form using the `submit` method instead of a `Submit` object. The form's `onSubmit` event handler does not prevent the form from being submitted. The form uses a button's `onClick` event handler to call the `checkData` function. If the value is valid, the `checkData` function submits the form by calling the form's `submit` method.

```
<HTML>
<HEAD>
<TITLE>Form object/submit method example</TITLE>
</HEAD>
<SCRIPT>
var dataOK=false
function checkData (){
if (document.myForm.threeChar.value.length == 3) {
   document.myForm.submit()}
   else {
      alert("Enter exactly three characters. " +
document.myForm.threeChar.value +
         " is not valid.")
      return false}
}
</SCRIPT>
<BODY>
<FORM NAME="myForm" onSubmit="alert('Form is being submitted.')">
<B>Enter 3 characters:</B>
<INPUT TYPE="text" NAME="threeChar" SIZE=3>
<P><INPUT TYPE="button" VALUE="Done" NAME="button1"
   onClick="checkData()">
</FORM>
</BODY>
</HTML>
```

See also `Button`, `Checkbox`, `FileUpload`, `Hidden`, `Password`, `Radio`, `Reset`, `Select`, `Submit`, `Text`, `Textarea`.

action

A string specifying a destination URL for form data that is submitted

Property of	`Form`
Implemented in	JavaScript 1.0

Security Submitting a form to a `mailto:` or `news:` URL requires the `UniversalSendMail` privilege. For information on security, see the *Client-Side JavaScript Guide*.

JavaScript 1.1. This property is tainted by default. For information on data tainting, see the *Client-Side JavaScript Guide*.

Description The `action` property is a reflection of the `ACTION` attribute of the `FORM` tag. Each section of a URL contains different information. See `Location` for a description of the URL components.

Examples The following example sets the `action` property of the `musicForm` form to the value of the variable `urlName`:

```
document.musicForm.action=urlName
```

See also `Form.encoding`, `Form.method`, `Form.target`

elements

An array of objects corresponding to form elements (such as `checkbox`, `radio`, and `Text` objects) in source order.

Property of	`Form`
Read-only	
Implemented in	JavaScript 1.0

Description You can refer to a form's elements in your code by using the `elements` array. This array contains an entry for each object (`Button`, `Checkbox`, `FileUpload`, `Hidden`, `Password`, `Radio`, `Reset`, `Select`, `Submit`, `Text`, or `Textarea` object) in a form in source order. Each radio button in a `Radio` object appears as a separate element in the `elements` array. For example, if a form called `myForm` has a text field and two checkboxes, you can refer to these elements `myForm.elements[0]`, `myForm.elements[1]`, and `myForm.elements[2]`.

Although you can also refer to a form's elements by using the element's name (from the `NAME` attribute), the `elements` array provides a way to refer to `Form` objects programmatically without using their names. For example, if the first object on the `userInfo` form is the `userName` `Text` object, you can evaluate it in either of the following ways:

```
userInfo.userName.value
userInfo.elements[0].value
```

The value of each element in the `elements` array is the full HTML statement for the object.

To obtain the number of elements in a form, use the `length` property: `myForm.elements.length`.

Examples See the examples for `window`.

encoding

A string specifying the MIME encoding of the form.

Property of	`Form`
Implemented in	JavaScript 1.0

Description The `encoding` property initially reflects the `ENCTYPE` attribute of the `FORM` tag; however, setting `encoding` overrides the `ENCTYPE` attribute.

Examples The following function returns the value of the `encoding` property of `musicForm`:

```
function getEncoding() {
   return document.musicForm.encoding
}
```

See also `Form.action`, `Form.method`, `Form.target`

handleEvent

Invokes the handler for the specified event.

Method of	`Form`
Implemented in	JavaScript 1.2

Syntax `handleEvent(`*`event`*`)`

Parameters

`event`	The name of an event for which the specified object has an event handler.

Description For information on handling events, see the *Client-Side JavaScript Guide*.

length

The number of elements in the form.

Property of	`Form`
Read-only	
Implemented in	JavaScript 1.0

Description The `form.length` property tells you how many elements are in the form. You can get the same information using `form.elements.length`.

method

A string specifying how form field input information is sent to the server.

Property of	`Form`
Implemented in	JavaScript 1.0

Description The `method` property is a reflection of the `METHOD` attribute of the `FORM` tag. The `method` property should evaluate to either `"get"` or `"post"`.

Examples The following function returns the value of the musicForm method property:

```
function getMethod() {
   return document.musicForm.method
}
```

See also Form.action, Form.encoding, Form.target

name

A string specifying the name of the form.

Property of	Form
Implemented in	JavaScript 1.0

Security **JavaScript 1.1.** This property is tainted by default. For information on data tainting, see the *Client-Side JavaScript Guide*.

Description The name property initially reflects the value of the NAME attribute. Changing the name property overrides this setting.

Examples In the following example, the valueGetter function uses a for loop to iterate over the array of elements on the valueTest form. The msgWindow window displays the names of all the elements on the form:

```
newWindow=window.open("http://home.netscape.com")

function valueGetter() {
   var msgWindow=window.open("")
   for (var i = 0; i < newWindow.document.valueTest.elements.length; i++) {
      msgWindow.document.write(newWindow.document.valueTest.elements[i].name + "<BR>")
   }
}
```

reset

Simulates a mouse click on a reset button for the calling form.

Method of	`Form`
Implemented in	JavaScript 1.1

Syntax `reset()`

Parameters None

Description The `reset` method restores a form element's default values. A reset button does not need to be defined for the form.

Examples The following example displays a `Text` object in which the user is to type "CA" or "AZ". The `Text` object's `onChange` event handler calls a function that executes the form's `reset` method if the user provides incorrect input. When the `reset` method executes, defaults are restored and the form's `onReset` event handler displays a message.

```
<SCRIPT>
function verifyInput(textObject) {
   if (textObject.value == 'CA' || textObject.value == 'AZ') {
      alert('Nice input')
   }
   else { document.myForm.reset() }
}
</SCRIPT>

<FORM NAME="myForm" onReset="alert('Please enter CA or AZ.')">
Enter CA or AZ:
<INPUT TYPE="text" NAME="state" SIZE="2" onChange=verifyInput(this)><P>
</FORM>
```

See also `onReset`, `Reset`

submit

Submits a form.

Method of	`Form`
Implemented in	JavaScript 1.0

Syntax `submit()`

Parameters None

Security Submitting a form to a `mailto:` or `news:` URL requires the `UniversalSendMail` privilege. For information on security, see the *Client-Side JavaScript Guide*.

JavaScript 1.1: The `submit` method fails without notice if the form's `action` is a `mailto:`, `news:`, or `snews:` URL. Users can submit forms with such URLs by clicking a submit button, but a confirming dialog will tell them that they are about to give away private or sensitive information.

Description The `submit` method submits the specified form. It performs the same action as a submit button.

Use the submit method to send data back to an HTTP server. The `submit` method returns the data using either "get" or "post," as specified in `Form.method`.

Examples The following example submits a form called `musicChoice`:

```
document.musicChoice.submit()
```

If `musicChoice` is the first form created, you also can submit it as follows:

```
document.forms[0].submit()
```

See also the example for `Form`.

See also `Submit`, `onSubmit`

target

A string specifying the name of the window that responses go to after a form has been submitted.

Property of	`Form`
Implemented in	JavaScript 1.0

Description The `target` property initially reflects the `TARGET` attribute of the `A`, `AREA`, and `FORM` tags; however, setting `target` overrides these attributes.

You can set `target` using a string, if the string represents a window name. The `target` property cannot be assigned the value of a JavaScript expression or variable.

Examples The following example specifies that responses to the `musicInfo` form are displayed in the `msgWindow` window:

```
document.musicInfo.target="msgWindow"
```

See also `Form.action`, `Form.encoding`, `Form.method`

Frame

A window can display multiple, independently scrollable *frames* on a single screen, each with its own distinct URL. These frames are created using the `FRAME` tag inside a `FRAMESET` tag. A series of frames makes up a page. Each frame can point to different URLs and be targeted by other URLs, all within the same page.

The `Frame` object is provided a convenience for referring to the objects that constitute frames. However, JavaScript actually represents a frame using a `window` object. Every `Frame` object is a `window` object, and has all the methods and properties of a `window` object. However, a window that is a frame differs slightly from a top-level window.

See `window` for complete information on frames.

Client-side object

Implemented in	JavaScript 1.0 JavaScript 1.1: added `blur` and `focus` methods; added `onBlur` and `onFocus` event handlers

Function

Specifies a string of JavaScript code to be compiled as a function.
Core object

Implemented in	JavaScript 1.1, NES 2.0 JavaScript 1.2: added `arity`, `arguments.callee` properties; added ability to nest functions JavaScript 1.3: added `apply`, `call`, and `toSource` methods; deprecated `arguments.caller` property
ECMA version	ECMA-262

Created by

The `Function` constructor:

```
new Function ([arg1[, arg2[, ... argN]],] functionBody)
```

The `function` statement (see “function” on page 622 for details):

```
function name([param[, param[, ... param]]]) {
   statements
}
```

Parameters

`arg1, arg2, ... argN`	Names to be used by the function as formal argument names. Each must be a string that corresponds to a valid JavaScript identifier; for example `"x"` or `"theValue"`.
`functionBody`	A string containing the JavaScript statements comprising the function definition.
`name`	The function name.
`param`	The name of an argument to be passed to the function. A function can have up to 255 arguments.
`statements`	The statements comprising the body of the function.

Description

`Function` objects created with the `Function` constructor are evaluated each time they are used. This is less efficient than declaring a function and calling it within your code, because declared functions are compiled.

To return a value, the function must have a `return` statement that specifies the value to return.

All parameters are passed to functions *by value*; the value is passed to the function, but if the function changes the value of the parameter, this change is not reflected globally or in the calling function. However, if you pass an object as a parameter to a function and the function changes the object's properties, that change is visible outside the function, as shown in the following example:

```
function myFunc(theObject) {
   theObject.make="Toyota"
}

mycar = {make:"Honda", model:"Accord", year:1998}
x=mycar.make     // returns Honda
myFunc(mycar)    // pass object mycar to the function
y=mycar.make     // returns Toyota (prop was changed by the function)
```

The `this` keyword does not refer to the currently executing function, so you must refer to `Function` objects by name, even within the function body.

Accessing a function's arguments with the arguments array. You can refer to a function's arguments within the function by using the `arguments` array. See `arguments`.

Specifying arguments with the Function constructor. The following code creates a `Function` object that takes two arguments.

```
var multiply = new Function("x", "y", "return x * y")
```

The arguments `"x"` and `"y"` are formal argument names that are used in the function body, `"return x * y"`.

The preceding code assigns a function to the variable `multiply`. To call the `Function` object, you can specify the variable name as if it were a function, as shown in the following examples.

```
var theAnswer = multiply(7,6)

var myAge = 50
if (myAge >=39) {myAge=multiply (myAge,.5)}
```

Assigning a function to a variable with the Function constructor.

Suppose you create the variable `multiply` using the `Function` constructor, as shown in the preceding section:

```
var multiply = new Function("x", "y", "return x * y")
```

This is similar to declaring the following function:

```
function multiply(x,y) {
   return x*y
}
```

Assigning a function to a variable using the `Function` constructor is similar to declaring a function with the `function` statement, but they have differences:

- When you assign a function to a variable using `var multiply = new Function("...")`, `multiply` is a variable for which the current value is a reference to the function created with `new Function()`.
- When you create a function using `function multiply() {...}`, `multiply` is not a variable, it is the name of a function.

Nesting functions. You can nest a function within a function. The nested (inner) function is private to its containing (outer) function:

- The inner function can be accessed only from statements in the outer function.
- The inner function can use the arguments and variables of the outer function. The outer function cannot use the arguments and variables of the inner function.

The following example shows nested functions:

```
function addSquares (a,b) {
   function square(x) {
      return x*x
   }
   return square(a) + square(b)
}
a=addSquares(2,3) // returns 13
b=addSquares(3,4) // returns 25
c=addSquares(4,5) // returns 41
```

When a function contains a nested function, you can call the outer function and specify arguments for both the outer and inner function:

```
function outside(x) {
   function inside(y) {
      return x+y
   }
   return inside
}
result=outside(3)(5) // returns 8
```

Specifying an event handler with a Function object. The following code assigns a function to a window's `onFocus` event handler (the event handler must be spelled in all lowercase):

```
window.onfocus = new Function("document.bgColor='antiquewhite'")
```

If a function is assigned to a variable, you can assign the variable to an event handler. The following code assigns a function to the variable `setBGColor`.

```
var setBGColor = new Function("document.bgColor='antiquewhite'")
```

You can use this variable to assign a function to an event handler in either of the following ways:

```
document.form1.colorButton.onclick=setBGColor
```

```
<INPUT NAME="colorButton" TYPE="button"
   VALUE="Change background color"
   onClick="setBGColor()">
```

Once you have a reference to a `Function` object, you can use it like a function and it will convert from an object to a function:

```
window.onfocus()
```

Event handlers do not take arguments, so you cannot declare any arguments in a `Function` constructor for an event handler. For example, you cannot call the function `multiply` by setting a button's `onclick` property as follows:

```
document.form1.button1.onclick=multFun(5,10)
```

Backward Compatibility

JavaScript 1.1 and earlier versions. You cannot nest a function statement in another statement or in itself.

Property Summary

Property	Description
`arguments`	An array corresponding to the arguments passed to a function.
`arguments.callee`	Specifies the function body of the currently executing function.
`arguments.caller`	Specifies the name of the function that invoked the currently executing function.
`arguments.length`	Specifies the number of arguments passed to the function.
`arity`	Specifies the number of arguments expected by the function.
`constructor`	Specifies the function that creates an object's prototype.
`length`	Specifies the number of arguments expected by the function.
`prototype`	Allows the addition of properties to a `Function` object.

Method Summary

Method	Description
`apply`	Allows you to apply a method of another object in the context of a different object (the calling object).
`call`	Allows you to call (execute) a method of another object in the context of a different object (the calling object).
`toSource`	Returns a string representing the source code of the function. Overrides the `Object.toSource` method.
`toString`	Returns a string representing the source code of the function. Overrides the `Object.toString` method.
`valueOf`	Returns a string representing the source code of the function. Overrides the `Object.valueOf` method.

Examples **Example 1.** The following function returns a string containing the formatted representation of a number padded with leading zeros.

```
// This function returns a string padded with leading zeros
function padZeros(num, totalLen) {
   var numStr = num.toString()             // Initialize return value
                                           // as string
   var numZeros = totalLen - numStr.length // Calculate no. of zeros
   if (numZeros > 0) {
      for (var i = 1; i <= numZeros; i++) {
         numStr = "0" + numStr
      }
   }
   return numStr
}
```

The following statements call the `padZeros` function.

```
result=padZeros(42,4) // returns "0042"
result=padZeros(42,2) // returns "42"
result=padZeros(5,4)  // returns "0005"
```

Example 2. You can determine whether a function exists by comparing the function name to null. In the following example, `func1` is called if the function `noFunc` does not exist; otherwise `func2` is called. Notice that the window name is needed when referring to the function name `noFunc`.

```
if (window.noFunc == null)
   func1()
else func2()
```

Example 3. The following example creates `onFocus` and `onBlur` event handlers for a frame. This code exists in the same file that contains the `FRAMESET` tag. Note that this is the only way to create `onFocus` and `onBlur` event handlers for a frame, because you cannot specify the event handlers in the `FRAME` tag.

```
frames[0].onfocus = new Function("document.bgColor='antiquewhite'")
frames[0].onblur = new Function("document.bgColor='lightgrey'")
```

apply

Allows you to apply a method of another object in the context of a different object (the calling object).

Method of	`Function`
Implemented in	JavaScript 1.3

Syntax `apply(`*`thisArg`*`[, `*`argArray`*`])`

Parameters

`thisArg`	Parameter for the calling object
`argArray`	An argument array for the object

Description You can assign a different `this` object when calling an existing function. `this` refers to the current object, the calling object. With `apply`, you can write a method once and then inherit it in another object, without having to rewrite the method for the new object.

`apply` is very similar to `call`, except for the type of arguments it supports. You can use an arguments array instead of a named set of parameters. With `apply`, you can use an array literal, for example, `apply(this, [name, value])`, or an `Array` object, for example, `apply(this, new Array(name, value))`.

You can also use `arguments` for the `argArray` parameter. `arguments` is a local variable of a function. It can be used for all unspecified arguments of the called object. Thus, you do not have to know the arguments of the called object when you use the `apply` method. You can use `arguments` to pass all the arguments to the called object. The called object is then responsible for handling the arguments.

Examples You can use `apply` to chain constructors for an object, similar to Java. In the following example, the constructor for the `product` object is defined with two parameters, `name` and `value`. Another object, `prod_dept`, initializes its unique variable (`dept`) and calls the constructor for `product` in its constructor to initialize the other variables. In this example, the parameter `arguments` is used for all arguments of the `product` object's constructor.

```
function product(name, value){
   this.name = name;
   if(value > 1000)
      this.value = 999;
   else
      this.value = value;
}

function prod_dept(name, value, dept){
   this.dept = dept;
   product.apply(product, arguments);
}

prod_dept.prototype = new product();

// since 5 is less than 100 value is set
cheese = new prod_dept("feta", 5, "food");

// since 5000 is above 1000, value will be 999
car = new prod_dept("honda", 5000, "auto");
```

See also `Function.call`

arguments

An array corresponding to the arguments passed to a function.

Local variable of	All function objects
Property of	`Function` (deprecated)
Implemented in	JavaScript 1.1, NES 2.0 JavaScript 1.2: added `arguments.callee` property JavaScript 1.3: deprecated `arguments.caller` property; removed support for argument names and local variable names as properties of the `arguments` array
ECMA version	ECMA-262

Description You can refer to a function's arguments within the function by using the `arguments` array. This array contains an entry for each argument passed to the function. For example, if a function is passed three arguments, you can refer to the arguments as follows:

```
arguments[0]
arguments[1]
arguments[2]
```

The `arguments` array can also be preceded by the function name:

```
myFunc.arguments[0]
myFunc.arguments[1]
myFunc.arguments[2]
```

The `arguments` array is available only within a function body. Attempting to access the `arguments` array outside a function declaration results in an error.

You can use the `arguments` array if you call a function with more arguments than it is formally declared to accept. This technique is useful for functions that can be passed a variable number of arguments. You can use `arguments.length` to determine the number of arguments passed to the function, and then process each argument by using the `arguments` array. (To determine the number of arguments declared when a function was defined, use the `Function.length` property.)

The `arguments` array has the following properties:

Property	Description
`arguments.callee`	Specifies the function body of the currently executing function.
`arguments.caller`	Specifies the name of the function that invoked the currently executing function. (Deprecated)
`arguments.length`	Specifies the number of arguments passed to the function.

Backward Compatibility

JavaScript 1.1 and 1.2. The following features that were available in JavaScript 1.1 and JavaScript 1.2 have been removed:

- Each local variable of a function is a property of the `arguments` array. For example, if a function `myFunc` has a local variable named `myLocalVar`, you can refer to the variable as `arguments.myLocalVar`.
- Each formal argument of a function is a property of the `arguments` array. For example, if a function `myFunc` has two arguments named `arg1` and `arg2`, you can refer to the arguments as `arguments.arg1` and `arguments.arg2`. (You can also refer to them as `arguments[0]` and `arguments[1]`.)

Examples

Example 1. This example defines a function that concatenates several strings. The only formal argument for the function is a string that specifies the characters that separate the items to concatenate. The function is defined as follows:

```
function myConcat(separator) {
   result="" // initialize list
   // iterate through arguments
   for (var i=1; i<arguments.length; i++) {
      result += arguments[i] + separator
   }
   return result
}
```

You can pass any number of arguments to this function, and it creates a list using each argument as an item in the list.

```
// returns "red, orange, blue, "
myConcat(", ","red","orange","blue")

// returns "elephant; giraffe; lion; cheetah;"
myConcat("; ","elephant","giraffe","lion", "cheetah")

// returns "sage. basil. oregano. pepper. parsley. "
myConcat(". ","sage","basil","oregano", "pepper", "parsley")
```

Example 2. This example defines a function that creates HTML lists. The only formal argument for the function is a string that is `"U"` if the list is to be unordered (bulleted), or `"O"` if the list is to be ordered (numbered). The function is defined as follows:

```
function list(type) {
   document.write("<" + type + "L>") // begin list
   // iterate through arguments
   for (var i=1; i<arguments.length; i++) {
      document.write("<LI>" + arguments[i])
   }
   document.write("</" + type + "L>") // end list
}
```

You can pass any number of arguments to this function, and it displays each argument as an item in the type of list indicated. For example, the following call to the function

```
list("U", "One", "Two", "Three")
```

results in this output:

```
<UL>
<LI>One
<LI>Two
<LI>Three
</UL>
```

arguments.callee

Specifies the function body of the currently executing function.

Property of	`arguments` local variable; `Function` (deprecated)
Implemented in	JavaScript 1.2
ECMA version	ECMA-262

Description The `callee` property is available only within the body of a function.

The `this` keyword does not refer to the currently executing function. Use the `callee` property to refer to a function within the function body.

Examples The following function returns the value of the function's `callee` property.

```
function myFunc() {
   return arguments.callee
}
```

The following value is returned:

```
function myFunc() { return arguments.callee; }
```

See also `Function.arguments`

arguments.caller

Specifies the name of the function that invoked the currently executing function.

Property of	`Function`
Implemented in	JavaScript 1.1, NES 2.0
	Deprecated in JavaScript 1.3

Description `caller` is no longer used.

The `caller` property is available only within the body of a function.

If the currently executing function was invoked by the top level of a JavaScript program, the value of `caller` is null.

The `this` keyword does not refer to the currently executing function, so you must refer to functions and `Function` objects by name, even within the function body.

The `caller` property is a reference to the calling function, so

- If you use it in a string context, you get the result of calling `functionName.toString`. That is, the decompiled canonical source form of the function.
- You can also call the calling function, if you know what arguments it might want. Thus, a called function can call its caller without knowing the name of the particular caller, provided it knows that all of its callers have the same form and fit, and that they will not call the called function again unconditionally (which would result in infinite recursion).

Examples The following code checks the value of a function's `caller` property.

```
function myFunc() {
   if (arguments.caller == null) {
      return ("The function was called from the top!")
   } else return ("This function's caller was " + arguments.caller)
}
```

See also `Function.arguments`

arguments.length

Specifies the number of arguments passed to the function.

Property of	`arguments` local variable; `Function` (deprecated)
Implemented in	JavaScript 1.1
ECMA version	ECMA-262

Description `arguments.length` provides the number of arguments actually passed to a function. By contrast, the `Function.length` property indicates how many arguments a function expects.

Example The following example demonstrates the use of `Function.length` and `arguments.length`.

```
function addNumbers(x,y){
   if (arguments.length == addNumbers.length) {
      return (x+y)
   }
   else return 0
}
```

If you pass more than two arguments to this function, the function returns 0:

```
result=addNumbers(3,4,5)   // returns 0
result=addNumbers(3,4)     // returns 7
result=addNumbers(103,104) // returns 207
```

See also `Function.arguments`

arity

Specifies the number of arguments expected by the function.

Property of	`Function`
Implemented in	JavaScript 1.2, NES 3.0

Description `arity` is external to the function, and indicates how many arguments a function expects. By contrast, `arguments.length` provides the number of arguments actually passed to a function.

Example The following example demonstrates the use of `arity` and `arguments.length`.

```
function addNumbers(x,y){
   if (arguments.length == addNumbers.length) {
      return (x+y)
   }
   else return 0
}
```

If you pass more than two arguments to this function, the function returns 0:

```
result=addNumbers(3,4,5)   // returns 0
result=addNumbers(3,4)     // returns 7
result=addNumbers(103,104) // returns 207
```

See also `arguments.length`, `Function.length`

call

Allows you to call (execute) a method of another object in the context of a different object (the calling object).

Method of	`Function`
Implemented in	JavaScript 1.3

Syntax `call(`*`thisArg`*`[, `*`arg1`*`[, `*`arg2`*`[, ...]]])`

Parameters

`thisArg`	Parameter for the calling object
`arg1, arg2, ...`	Arguments for the object

Description You can assign a different `this` object when calling an existing function. `this` refers to the current object, the calling object.

With `call`, you can write a method once and then inherit it in another object, without having to rewrite the method for the new object.

Examples You can use `call` to chain constructors for an object, similar to Java. In the following example, the constructor for the `product` object is defined with two parameters, `name` and `value`. Another object, `prod_dept`, initializes its unique variable (`dept`) and calls the constructor for `product` in its constructor to initialize the other variables.

```
function product(name, value){
   this.name = name;
   if(value > 1000)
      this.value = 999;
   else
      this.value = value;
}

function prod_dept(name, value, dept){
   this.dept = dept;
   product.call(this, name, value);
}

prod_dept.prototype = new product();

// since 5 is less than 100 value is set
cheese = new prod_dept("feta", 5, "food");

// since 5000 is above 1000, value will be 999
car = new prod_dept("honda", 5000, "auto");
```

See also `Function.apply`

constructor

Specifies the function that creates an object's prototype. Note that the value of this property is a reference to the function itself, not a string containing the function's name.

Property of	`Function`
Implemented in	JavaScript 1.1, NES 2.0
ECMA version	ECMA-262

Description See `Object.constructor`.

length

Specifies the number of arguments expected by the function.

Property of	`Function`
Implemented in	JavaScript 1.1
ECMA version	ECMA-262

Description `length` is external to a function, and indicates how many arguments the function expects. By contrast, `arguments.length` is local to a function and provides the number of arguments actually passed to the function.

Example See the example for `arguments.length`.

See also `arguments.length`

prototype

A value from which instances of a particular class are created. Every object that can be created by calling a constructor function has an associated `prototype` property.

Property of	`Function`
Implemented in	JavaScript 1.1, NES 2.0
ECMA version	ECMA-262

Description You can add new properties or methods to an existing class by adding them to the prototype associated with the constructor function for that class. The syntax for adding a new property or method is:

```
fun.prototype.name = value
```

where

`fun`	The name of the constructor function object you want to change.
`name`	The name of the property or method to be created.
`value`	The value initially assigned to the new property or method.

If you add a property to the prototype for an object, then all objects created with that object's constructor function will have that new property, even if the objects existed before you created the new property. For example, assume you have the following statements:

```
var array1 = new Array();
var array2 = new Array(3);
Array.prototype.description=null;
array1.description="Contains some stuff"
array2.description="Contains other stuff"
```

After you set a property for the prototype, all subsequent objects created with `Array` will have the property:

```
anotherArray=new Array()
anotherArray.description="Currently empty"
```

Example The following example creates a method, `str_rep`, and uses the statement `String.prototype.rep = str_rep` to add the method to all `String` objects. All objects created with `new String()` then have that method, even objects already created. The example then creates an alternate method and adds that to one of the `String` objects using the statement `s1.rep = fake_rep`. The `str_rep` method of the remaining `String` objects is not altered.

```
var s1 = new String("a")
var s2 = new String("b")
var s3 = new String("c")

// Create a repeat-string-N-times method for all String objects
function str_rep(n) {
   var s = "", t = this.toString()
   while (--n >= 0) s += t
   return s
}

String.prototype.rep = str_rep

s1a=s1.rep(3) // returns "aaa"
s2a=s2.rep(5) // returns "bbbbb"
s3a=s3.rep(2) // returns "cc"

// Create an alternate method and assign it to only one String variable
function fake_rep(n) {
   return "repeat " + this + " " + n + " times."
}

s1.rep = fake_rep
s1b=s1.rep(1) // returns "repeat a 1 times."
s2b=s2.rep(4) // returns "bbbb"
s3b=s3.rep(6) // returns "cccccc"
```

The function in this example also works on `String` objects not created with the `String` constructor. The following code returns `"zzz"`.

```
"z".rep(3)
```

toSource

Returns a string representing the source code of the function.

Method of	`Function`
Implemented in	JavaScript 1.3

Syntax `toSource()`

Parameters None

Description The `toSource` method returns the following values:

- For the built-in `Function` object, `toSource` returns the following string indicating that the source code is not available:

```
function Function() {
   [native code]
}
```

- For custom functions, `toSource` returns the JavaScript source that defines the object as a string.

This method is usually called internally by JavaScript and not explicitly in code. You can call toSource while debugging to examine the contents of an object.

See also `Function.toString`, `Object.valueOf`

toString

Returns a string representing the source code of the function.

Method of	`Function`
Implemented in	JavaScript 1.1, NES 2.0
ECMA version	ECMA-262

Syntax `toString()`

Parameters None.

Description The `Function` object overrides the `toString` method of the `Object` object; it does not inherit `Object.toString`. For `Function` objects, the `toString` method returns a string representation of the object.

JavaScript calls the `toString` method automatically when a `Function` is to be represented as a text value or when a `Function` is referred to in a string concatenation.

For `Function` objects, the built-in `toString` method decompiles the function back into the JavaScript source that defines the function. This string includes the `function` keyword, the argument list, curly braces, and function body.

For example, assume you have the following code that defines the `Dog` object type and creates `theDog,` an object of type `Dog`:

```
function Dog(name,breed,color,sex) {
   this.name=name
   this.breed=breed
   this.color=color
   this.sex=sex
}
```

```
theDog = new Dog("Gabby","Lab","chocolate","girl")
```

Any time `Dog` is used in a string context, JavaScript automatically calls the `toString` function, which returns the following string:

```
function Dog(name, breed, color, sex) { this.name = name; this.breed =
breed; this.color = color; this.sex = sex; }
```

See also `Object.toString`

valueOf

Returns a string representing the source code of the function.

Method of	`Function`
Implemented in	JavaScript 1.1
ECMA version	ECMA-262

Syntax `valueOf()`

Parameters None

Description The `valueOf` method returns the following values:

- For the built-in `Function` object, `valueOf` returns the following string indicating that the source code is not available:

  ```
  function Function() {
     [native code]
  }
  ```
- For custom functions, `toSource` returns the JavaScript source that defines the object as a string. The method is equivalent to the `toString` method of the function.

This method is usually called internally by JavaScript and not explicitly in code.

See also `Function.toString`, `Object.valueOf`

Hidden

A `Text` object that is suppressed from form display on an HTML form. A `Hidden` object is used for passing name/value pairs when a form submits.

Client-side object

Implemented in	JavaScript 1.0
	JavaScript 1.1: added `type` property

Created by The HTML `INPUT` tag, with `"hidden"` as the value of the `TYPE` attribute. For a given form, the JavaScript runtime engine creates appropriate `Hidden` objects and puts these objects in the `elements` array of the corresponding `Hidden` object. You access a `Hidden` object by indexing this array. You can index the array either by number or, if supplied, by using the value of the `NAME` attribute.

Description A `Hidden` object is a form element and must be defined within a `FORM` tag.

A `Hidden` object cannot be seen or modified by an end user, but you can programmatically change the value of the object by changing its `value` property. You can use `Hidden` objects for client/server communication.

Property Summary

Property	Description
`form`	Specifies the form containing the `Hidden` object.
`name`	Reflects the `NAME` attribute.
`type`	Reflects the `TYPE` attribute.
`value`	Reflects the current value of the `Hidden` object.

Method Summary This object inherits the `watch` and `unwatch` methods from `Object`.

Examples The following example uses a `Hidden` object to store the value of the last object the user clicked. The form contains a "Display hidden value" button that the user can click to display the value of the `Hidden` object in an Alert dialog box.

```
<HTML>
<HEAD>
<TITLE>Hidden object example</TITLE>
</HEAD>
<BODY>
<B>Click some of these objects, then click the "Display value" button
<BR>to see the value of the last object clicked.</B>
```

```
<FORM NAME="myForm">
<INPUT TYPE="hidden" NAME="hiddenObject" VALUE="None">
<P>
<INPUT TYPE="button" VALUE="Click me" NAME="button1"
   onClick="document.myForm.hiddenObject.value=this.value">
<P>
<INPUT TYPE="radio" NAME="musicChoice" VALUE="soul-and-r&b"
   onClick="document.myForm.hiddenObject.value=this.value"> Soul and
R&B
<INPUT TYPE="radio" NAME="musicChoice" VALUE="jazz"
   onClick="document.myForm.hiddenObject.value=this.value"> Jazz
<INPUT TYPE="radio" NAME="musicChoice" VALUE="classical"
   onClick="document.myForm.hiddenObject.value=this.value"> Classical
<P>
<SELECT NAME="music_type_single"

onFocus="document.myForm.hiddenObject.value=this.options[this.selectedI
ndex].text">
   <OPTION SELECTED> Red <OPTION> Orange <OPTION> Yellow
</SELECT>
<P><INPUT TYPE="button" VALUE="Display hidden value" NAME="button2"
   onClick="alert('Last object clicked: ' +
document.myForm.hiddenObject.value)">
</FORM>
</BODY>
</HTML>
```

See also `document.cookie`

form

An object reference specifying the form containing this object.

Property of	`Hidden`
Read-only	
Implemented in	JavaScript 1.0

Description Each form element has a `form` property that is a reference to the element's parent form. This property is especially useful in event handlers, where you might need to refer to another element on the current form.

Examples **Example 1.** In the following example, the form `myForm` contains a `Hidden` object and a button. When the user clicks the button, the value of the `Hidden` object is set to the form's name. The button's `onClick` event handler uses `this.form` to refer to the parent form, `myForm`.

```
<FORM NAME="myForm">
Form name:<INPUT TYPE="hidden" NAME="h1" VALUE="Beluga">
<P>
<INPUT NAME="button1" TYPE="button" VALUE="Store Form Name"
   onClick="this.form.h1.value=this.form.name">
</FORM>
```

Example 2. The following example uses an object reference, rather than the `this` keyword, to refer to a form. The code returns a reference to `myForm`, which is a form containing `myHiddenObject`.

```
document.myForm.myHiddenObject.form
```

See also `Hidden`

name

A string specifying the name of this object.

Property of	`Hidden`
Implemented in	JavaScript 1.0

Security **JavaScript 1.1.** This property is tainted by default. For information on data tainting, see the *Client-Side JavaScript Guide*.

type

For all `Hidden` objects, the value of the `type` property is `"hidden"`. This property specifies the form element's type.

Property of	`Hidden`
Read-only	
Implemented in	JavaScript 1.1

Examples The following example writes the value of the `type` property for every element on a form.

```
for (var i = 0; i < document.myForm.elements.length; i++) {
   document.writeln("<BR>type is " + document.myForm.elements[i].type)
}
```

value

A string that reflects the `VALUE` attribute of the object.

Property of	`Hidden`
Implemented in	JavaScript 1.0

Security **JavaScript 1.1.** This property is tainted by default. For information on data tainting, see the *Client-Side JavaScript Guide*.

Examples The following function evaluates the `value` property of a group of buttons and displays it in the `msgWindow` window:

```
function valueGetter() {
   var msgWindow=window.open("")
   msgWindow.document.write("The submit button says " +
      document.valueTest.submitButton.value + "<BR>")
   msgWindow.document.write("The reset button says " +
      document.valueTest.resetButton.value + "<BR>")
   msgWindow.document.write("The hidden field says " +
      document.valueTest.hiddenField.value + "<BR>")
   msgWindow.document.close()
}
```

This example displays the following values:

```
The submit button says Query Submit
The reset button says Reset
The hidden field says pipefish are cute.
```

The previous example assumes the buttons have been defined as follows:

```
<INPUT TYPE="submit" NAME="submitButton">
<INPUT TYPE="reset" NAME="resetButton">
<INPUT TYPE="hidden" NAME="hiddenField" VALUE="pipefish are cute.">
```

History

Contains an array of information on the URLs that the client has visited within a window. This information is stored in a history list and is accessible through the browser's Go menu.

Client-side object

Implemented in	JavaScript 1.0 JavaScript 1.1: added `current, next,` and `previous` properties.

Created by `History` objects are predefined JavaScript objects that you access through the `history` property of a `window` object.

Description To change a window's current URL without generating a history entry, you can use the `Location.replace` method. This replaces the current page with a new one without generating a history entry. See `Location.replace`.

You can refer to the history entries by using the `window.history` array. This array contains an entry for each history entry in source order. Each array entry is a string containing a URL. For example, if the history list contains three named entries, these entries are reflected as `history[0]`, `history[1]`, and `history[2]`.

If you access the `history` array without specifying an array element, the browser returns a string of HTML which displays a table of URLs, each of which is a link.

Property Summary

Property	Description
`current`	Specifies the URL of the current history entry.
`length`	Reflects the number of entries in the history list.
`next`	Specifies the URL of the next history entry.
`previous`	Specifies the URL of the previous history entry.

Method Summary

Method	Description
`back`	Loads the previous URL in the history list.
`forward`	Loads the next URL in the history list.
`go`	Loads a URL from the history list.

In addition, this object inherits the `watch` and `unwatch` methods from `Object`.

Examples

Example 1. The following example goes to the URL the user visited three clicks ago in the current window.

```
history.go(-3)
```

Example 2. You can use the `history` object with a specific window or frame. The following example causes `window2` to go back one item in its window (or session) history:

```
window2.history.back()
```

Example 3. The following example causes the second frame in a frameset to go back one item:

```
parent.frames[1].history.back()
```

Example 4. The following example causes the frame named `frame1` in a frameset to go back one item:

```
parent.frame1.history.back()
```

Example 5. The following example causes the frame named `frame2` in `window2` to go back one item:

```
window2.frame2.history.back()
```

Example 6. The following code determines whether the first entry in the `history` array contains the string `"NETSCAPE"`. If it does, the function `myFunction` is called.

```
if (history[0].indexOf("NETSCAPE") != -1) {
   myFunction(history[0])
}
```

Example 7. The following example displays the entire history list:

```
document.writeln("<B>history is</B> " + history)
```

This code displays output similar to the following:

```
history is
Welcome to Netscape http://home.netscape.com/
Sun Microsystems http://www.sun.com/
Royal Airways http://www.supernet.net/~dugbrown/
```

See also `Location`, `Location.replace`

back

Loads the previous URL in the history list.

Method of	`History`
Implemented in	JavaScript 1.0

Syntax `back()`

Parameters None

Description This method performs the same action as a user choosing the Back button in the browser. The `back` method is the same as `history.go(-1)`.

Examples The following custom buttons perform the same operation as the browser's Back button:

```
<P><INPUT TYPE="button" VALUE="< Go Back"
   onClick="history.back()">
<P><INPUT TYPE="button" VALUE="> Go Back"
   onClick="myWindow.back()">
```

See also `History.forward`, `History.go`

current

A string specifying the complete URL of the current history entry.

Property of	`History`
Read-only	
Implemented in	JavaScript 1.1

Security Getting the value of this property requires the `UniversalBrowserRead` privilege. It has no value if you do not have this privilege. For information on security, see the *Client-Side JavaScript Guide*.

JavaScript 1.1. This property is tainted by default. It has no value of data tainting is disabled. For information on data tainting, see the *Client-Side JavaScript Guide*.

Examples The following example determines whether `history.current` contains the string `"netscape.com"`. If it does, the function `myFunction` is called.

```
if (history.current.indexOf("netscape.com") != -1) {
   myFunction(history.current)
}
```

See also `History.next`, `History.previous`

forward

Loads the next URL in the history list.

Method of	`History`
Implemented in	JavaScript 1.0

Syntax `forward()`

Parameters None

Description This method performs the same action as a user choosing the Forward button in the browser. The `forward` method is the same as `history.go(1)`.

Examples The following custom buttons perform the same operation as the browser's Forward button:

```
<P><INPUT TYPE="button" VALUE="< Forward"
   onClick="history.forward()">
<P><INPUT TYPE="button" VALUE="> Forward"
   onClick="myWindow.forward()">
```

See also `History.back`, `History.go`

go

Loads a URL from the history list.

Method of	`History`
Implemented in	JavaScript 1.0

Syntax

```
go(delta)
go(location)
```

Parameters

`delta`	An integer representing a relative position in the history list.
`location`	A string representing all or part of a URL in the history list.

Description The `go` method navigates to the location in the history list determined by the specified parameter.

If the `delta` argument is 0, the browser reloads the current page. If it is an integer greater than 0, the `go` method loads the URL that is that number of entries forward in the history list; otherwise, it loads the URL that is that number of entries backward in the history list.

The `location` argument is a string. Use `location` to load the nearest history entry whose URL contains `location` as a substring. Matching the URL to the `location` parameter is case-insensitive. Each section of a URL contains different information. See `Location` for a description of the URL components.

The `go` method creates a new entry in the history list. To load a URL without creating an entry in the history list, use `Location.replace`.

Examples The following button navigates to the nearest history entry that contains the string `"home.netscape.com"`:

```
<P><INPUT TYPE="button" VALUE="Go"
   onClick="history.go('home.netscape.com')">
```

The following button navigates to the URL that is three entries backward in the history list:

```
<P><INPUT TYPE="button" VALUE="Go"
   onClick="history.go(-3)">
```

See also `History.back`, `History.forward`, `Location.reload`, `Location.replace`

length

The number of elements in the `history` array.

Property of	`History`
Read-only	
Implemented in	JavaScript 1.0

Security Getting the value of this property requires the `UniversalBrowserRead` privilege. For information on security, see the *Client-Side JavaScript Guide*.

next

A string specifying the complete URL of the next history entry.

Property of	`History`
Read-only	
Implemented in	JavaScript 1.1

Security Getting the value of this property requires the `UniversalBrowserRead` privilege. It has no value if you do not have this privilege. For information on security, see the *Client-Side JavaScript Guide*.

JavaScript 1.1. This property is tainted by default. It has no value if data tainting is disabled. For information on data tainting, see the *Client-Side JavaScript Guide*.

Description The `next` property reflects the URL that would be used if the user chose Forward from the Go menu.

Examples The following example determines whether `history.next` contains the string `"NETSCAPE.COM"`. If it does, the function `myFunction` is called.

```
if (history.next.indexOf("NETSCAPE.COM") != -1) {
   myFunction(history.next)
}
```

See also `History.current`, `History.previous`

previous

A string specifying the complete URL of the previous history entry.

Property of	`History`
Read-only	
Implemented in	JavaScript 1.1

Security Getting the value of this property requires the `UniversalBrowserRead` privilege. It has no value if you do not have this privilege. For information on security, see the *Client-Side JavaScript Guide*.

JavaScript 1.1. This property is tainted by default. It has no value of data tainting is disabled. For information on data tainting, see the *Client-Side JavaScript Guide*.

Description The `previous` property reflects the URL that would be used if the user chose Back from the Go menu.

Examples The following example determines whether `history.previous` contains the string `"NETSCAPE.COM"`. If it does, the function `myFunction` is called.

```
if (history.previous.indexOf("NETSCAPE.COM") != -1) {
   myFunction(history.previous)
}
```

See also `History.current`, `History.next`

Image

An image on an HTML form.

Client-side object

Implemented in	JavaScript 1.1
	JavaScript 1.2: added `handleEvent` method

Created by The `Image` constructor or the `IMG` tag.

The JavaScript runtime engine creates an `Image` object corresponding to each `IMG` tag in your document. It puts these objects in an array in the `document.images` property. You access an `Image` object by indexing this array.

To define an image with the `IMG` tag, use standard HTML syntax with the addition of JavaScript event handlers. If specify a value for the `NAME` attribute, you can use that name when indexing the `images` array.

To define an image with its constructor, use the following syntax:

```
new Image([width,] [height])
```

Parameters

`width`	The image width, in pixels.
`height`	The image height, in pixels.

Event handlers

- `onAbort`
- `onError`
- `onKeyDown`
- `onKeyPress`
- `onKeyUp`
- `onLoad`

To define an event handler for an `Image` object created with the `Image` constructor, set the appropriate property of the object. For example, if you have an `Image` object named `imageName` and you want to set one of its event handlers to a function whose name is `handlerFunction`, use one of the following statements:

```
imageName.onabort = handlerFunction
imageName.onerror = handlerFunction
imageName.onkeydown = handlerFunction
imageName.onkeypress = handlerFunction
imageName.onkeyup = handlerFunction
imageName.onload = handlerFunction
```

`Image` objects do not have `onClick`, `onMouseOut`, and `onMouseOver` event handlers. However, if you define an `Area` object for the image or place the `IMG` tag within a `Link` object, you can use the `Area` or `Link` object's event handlers. See `Link`.

Description The position and size of an image in a document are set when the document is displayed in the web browser and cannot be changed using JavaScript (the `width` and `height` properties are read-only for these objects). You can change which image is displayed by setting the `src` and `lowsrc` properties. (See the descriptions of `Image.src` and `Image.lowsrc`.)

You can use JavaScript to create an animation with an `Image` object by repeatedly setting the `src` property, as shown in Example 4 below. JavaScript animation is slower than GIF animation, because with GIF animation the entire animation is in one file; with JavaScript animation, each frame is in a separate file, and each file must be loaded across the network (host contacted and data transferred).

The primary use for an `Image` object created with the `Image` constructor is to load an image from the network (and decode it) before it is actually needed for display. Then when you need to display the image within an existing image cell, you can set the `src` property of the displayed image to the same value as that used for the previously fetched image, as follows.

```
myImage = new Image()
myImage.src = "seaotter.gif"
...
document.images[0].src = myImage.src
```

The resulting image will be obtained from cache, rather than loaded over the network, assuming that sufficient time has elapsed to load and decode the entire image. You can use this technique to create smooth animations, or you could display one of several images based on form input.

Property Summary

Property	Description
`border`	Reflects the `BORDER` attribute.
`complete`	Boolean value indicating whether the web browser has completed its attempt to load the image.
`height`	Reflects the `HEIGHT` attribute.
`hspace`	Reflects the `HSPACE` attribute.
`lowsrc`	Reflects the `LOWSRC` attribute.
`name`	Reflects the `NAME` attribute.
`src`	Reflects the `SRC` attribute.
`vspace`	Reflects the `VSPACE` attribute.
`width`	Reflects the `WIDTH` attribute.

Method Summary

Method	Description
`handleEvent`	Invokes the handler for the specified event.

In addition, this object inherits the `watch` and `unwatch` methods from `Object`.

Examples

Example 1: Create an image with the `IMG` tag. The following code defines an image using the `IMG` tag:

```
<IMG NAME="aircraft" SRC="f15e.gif" ALIGN="left" VSPACE="10">
```

The following code refers to the image:

```
document.aircraft.src='f15e.gif'
```

When you refer to an image by its name, you must include the form name if the image is on a form. The following code refers to the image if it is on a form:

```
document.myForm.aircraft.src='f15e.gif'
```

Example 2: Create an image with the Image constructor. The following example creates an `Image` object, `myImage`, that is 70 pixels wide and 50 pixels high. If the source URL, `seaotter.gif`, does not have dimensions of 70x50 pixels, it is scaled to that size.

```
myImage = new Image(70, 50)
myImage.src = "seaotter.gif"
```

If you omit the width and height arguments from the `Image` constructor, `myImage` is created with dimensions equal to that of the image named in the source URL.

```
myImage = new Image()
myImage.src = "seaotter.gif"
```

Example 3: Display an image based on form input. In the following example, the user selects which image is displayed. The user orders a shirt by filling out a form. The image displayed depends on the shirt color and size that the user chooses. All possible image choices are preloaded to speed response time. When the user clicks the button to order the shirt, the `allShirts` function displays the images of all the shirts.

```
<SCRIPT>
shirts = new Array()
shirts[0] = "R-S"
shirts[1] = "R-M"
shirts[2] = "R-L"
shirts[3] = "W-S"
shirts[4] = "W-M"
shirts[5] = "W-L"
shirts[6] = "B-S"
shirts[7] = "B-M"
shirts[8] = "B-L"

doneThis = 0
shirtImg = new Array()

// Preload shirt images
for(idx=0; idx < 9; idx++) {
   shirtImg[idx] = new Image()
   shirtImg[idx].src = "shirt-" + shirts[idx] + ".gif"
}
```

```
function changeShirt(form)
{
   shirtColor = form.color.options[form.color.selectedIndex].text
   shirtSize = form.size.options[form.size.selectedIndex].text

   newSrc = "shirt-" + shirtColor.charAt(0) + "-" + shirtSize.charAt(0)
+ ".gif"
   document.shirt.src = newSrc
}

function allShirts()
{
   document.shirt.src = shirtImg[doneThis].src
   doneThis++
   if(doneThis != 9)setTimeout("allShirts()", 500)
   else doneThis = 0

   return
}

</SCRIPT>

<FONT SIZE=+2><B>Netscape Polo Shirts!</FONT></B>

<TABLE CELLSPACING=20 BORDER=0>
<TR>
<TD><IMG name="shirt" SRC="shirt-W-L.gif"></TD>

<TD>
<FORM>
<B>Color</B>
<SELECT SIZE=3 NAME="color" onChange="changeShirt(this.form)">
<OPTION> Red
<OPTION SELECTED> White
<OPTION> Blue
</SELECT>

<P>
<B>Size</B>
<SELECT SIZE=3 NAME="size" onChange="changeShirt(this.form)">
<OPTION> Small
<OPTION> Medium
<OPTION SELECTED> Large
</SELECT>

<P><INPUT type="button" name="buy" value="Buy This Shirt!"
   onClick="allShirts()">
</FORM>

</TD>
</TR>
</TABLE>
```

Example 4: JavaScript animation. The following example uses JavaScript to create an animation with an `Image` object by repeatedly changing the value the `src` property. The script begins by preloading the 10 images that make up the animation (`image1.gif`, `image2.gif`, `image3.gif`, and so on). When the `Image` object is placed on the document with the `IMG` tag, `image1.gif` is displayed and the `onLoad` event handler starts the animation by calling the `animate` function. Notice that the `animate` function does not call itself after changing the `src` property of the `Image` object. This is because when the `src` property changes, the image's `onLoad` event handler is triggered and the `animate` function is called.

```
<SCRIPT>
delay = 100
imageNum = 1

// Preload animation images
theImages = new Array()
for(i = 1; i < 11; i++) {
   theImages[i] = new Image()
   theImages[i].src = "image" + i + ".gif"
}

function animate() {
   document.animation.src = theImages[imageNum].src
   imageNum++
   if(imageNum > 10) {
      imageNum = 1
   }
}

function slower() {
   delay+=10
   if(delay > 4000) delay = 4000
}

function faster() {
   delay-=10
   if(delay < 0) delay = 0
}
</SCRIPT>

<BODY BGCOLOR="white">

<IMG NAME="animation" SRC="image1.gif" ALT="[Animation]"
   onLoad="setTimeout('animate()', delay)">

<FORM>
   <INPUT TYPE="button" Value="Slower" onClick="slower()">
   <INPUT TYPE="button" Value="Faster" onClick="faster()">
</FORM>
</BODY>
```

See also the examples for the `onAbort`, `onError`, and `onLoad` event handlers.

See also `Link`, `onClick`, `onMouseOut`, `onMouseOver`

border

A string specifying the width, in pixels, of an image border.

Property of	`Image`
Read-only	
Implemented in	JavaScript 1.1

Description The `border` property reflects the `BORDER` attribute of the `IMG` tag. For images created with the `Image` constructor, the value of the `border` property is 0.

Examples The following function displays the value of an image's `border` property if the value is not 0.

```
function checkBorder(theImage) {
   if (theImage.border==0) {
      alert('The image has no border!')
   }
   else alert('The image's border is ' + theImage.border)
}
```

See also `Image.height`, `Image.hspace`, `Image.vspace`, `Image.width`

complete

A boolean value that indicates whether the web browser has completed its attempt to load an image.

Property of	`Image`
Read-only	
Implemented in	JavaScript 1.1

Examples The following example displays an image and three radio buttons. The user can click the radio buttons to choose which image is displayed. Clicking another button lets the user see the current value of the `complete` property.

```
<B>Choose an image:</B>
<BR><INPUT TYPE="radio" NAME="imageChoice" VALUE="image1" CHECKED
   onClick="document.images[0].src='f15e.gif'">F-15 Eagle
<BR><INPUT TYPE="radio" NAME="imageChoice" VALUE="image2"
   onClick="document.images[0].src='f15e2.gif'">F-15 Eagle 2
<BR><INPUT TYPE="radio" NAME="imageChoice" VALUE="image3"
   onClick="document.images[0].src='ah64.gif'">AH-64 Apache

<BR><INPUT TYPE="button" VALUE="Is the image completely loaded?"
   onClick="alert('The value of the complete property is '
      + document.images[0].complete)">
<BR>
<IMG NAME="aircraft" SRC="f15e.gif" ALIGN="left" VSPACE="10"><BR>
```

See also `Image.lowsrc`, `Image.src`

handleEvent

Invokes the handler for the specified event.

Method of	`Image`
Implemented in	JavaScript 1.2

Syntax `handleEvent(`*`event`*`)`

Parameters

`event`	The name of an event for which the specified object has an event handler.

Description For information on handling events, see the *Client-Side JavaScript Guide*.

height

A string specifying the height of an image in pixels.

Property of `Image`

Read-only

Implemented in JavaScript 1.1

Description The `height` property reflects the `HEIGHT` attribute of the `IMG` tag. For images created with the `Image` constructor, the value of the `height` property is the actual, not the displayed, height of the image.

Examples The following function displays the values of an image's `height`, `width`, `hspace`, and `vspace` properties.

```
function showImageSize(theImage) {
   alert('height=' + theImage.height+
      '; width=' + theImage.width +
      '; hspace=' + theImage.hspace +
      '; vspace=' + theImage.vspace)
}
```

See also `Image.border`, `Image.hspace`, `Image.vspace`, `Image.width`

hspace

A string specifying a margin in pixels between the left and right edges of an image and the surrounding text.

Property of `Image`

Read-only

Implemented in JavaScript 1.1

Description The `hspace` property reflects the `HSPACE` attribute of the `IMG` tag. For images created with the `Image` constructor, the value of the `hspace` property is 0.

Examples See the examples for the `height` property.

See also `Image.border`, `Image.height`, `Image.vspace`, `Image.width`

lowsrc

A string specifying the URL of a low-resolution version of an image to be displayed in a document.

Property of	`Image`
Implemented in	JavaScript 1.1

Description The `lowsrc` property initially reflects the `LOWSRC` attribute of the `IMG` tag. The web browser loads the smaller image specified by `lowsrc` and then replaces it with the larger image specified by the `src` property. You can change the `lowsrc` property at any time.

Examples See the examples for the `src` property.

See also `Image.complete`, `Image.src`

name

A string specifying the name of an object.

Property of	`Image`
Read-only	
Implemented in	JavaScript 1.1

Security **JavaScript 1.1.** This property is tainted by default. For information on data tainting, see the *Client-Side JavaScript Guide*.

Description Represents the value of the `NAME` attribute. For images created with the `Image` constructor, the value of the `name` property is null.

Examples In the following example, the `valueGetter` function uses a `for` loop to iterate over the array of elements on the `valueTest` form. The `msgWindow` window displays the names of all the elements on the form:

```
newWindow=window.open("http://home.netscape.com")

function valueGetter() {
   var msgWindow=window.open("")
   for (var i = 0; i < newWindow.document.valueTest.elements.length; i++) {
      msgWindow.document.write(newWindow.document.valueTest.elements[i].name + "<BR>")
   }
}
```

In the following example, the first statement creates a window called `netscapeWin`. The second statement displays the value `"netscapeHomePage"` in the Alert dialog box, because `"netscapeHomePage"` is the value of the `windowName` argument of `netscapeWin`.

```
netscapeWin=window.open("http://home.netscape.com","netscapeHomePage")
alert(netscapeWin.name)
```

src

A string specifying the URL of an image to be displayed in a document.

Property of	`Image`
Implemented in	JavaScript 1.1

Description The `src` property initially reflects the `SRC` attribute of the `IMG` tag. Setting the `src` property begins loading the new URL into the image area (and aborts the transfer of any image data that is already loading into the same area). Therefore, if you plan to alter the `lowsrc` property, you should do so before setting the `src` property.

If the URL in the `src` property refers to an image that is not the same size as the image cell it is loaded into, the source image is scaled to fit.

When you change the `src` property of a displayed image, the new image you specify is displayed in the area defined for the original image. For example, suppose an `Image` object originally displays the file `beluga.gif`:

```
<IMG NAME="myImage" SRC="beluga.gif" ALIGN="left">
```

If you set `myImage.src='seaotter.gif'`, the image `seaotter.gif` is scaled to fit in the same space originally used by `beluga.gif`, even if `seaotter.gif` is not the same size as `beluga.gif`.

You can change the `src` property at any time.

Examples The following example displays an image and three radio buttons. The user can click the radio buttons to choose which image is displayed. Each image also uses the `lowsrc` property to display a low-resolution image.

```
<SCRIPT>
function displayImage(lowRes,highRes) {
   document.images[0].lowsrc=lowRes
   document.images[0].src=highRes
}
</SCRIPT>

<FORM NAME="imageForm">
<B>Choose an image:</B>
<BR><INPUT TYPE="radio" NAME="imageChoice" VALUE="image1" CHECKED
   onClick="displayImage('f15el.gif','f15e.gif')">F-15 Eagle
<BR><INPUT TYPE="radio" NAME="imageChoice" VALUE="image2"
   onClick="displayImage('f15e2l.gif','f15e2.gif')">F-15 Eagle 2
<BR><INPUT TYPE="radio" NAME="imageChoice" VALUE="image3"
   onClick="displayImage('ah64l.gif','ah64.gif')">AH-64 Apache

<BR>
<IMG NAME="aircraft" SRC="f15e.gif" LOWSRC="f15el.gif" ALIGN="left" VSPACE="10"><BR>
</FORM>
```

See also `Image.complete`, `Image.lowsrc`

vspace

A string specifying a margin in pixels between the top and bottom edges of an image and the surrounding text.

Property of	`Image`
Read-only	
Implemented in	JavaScript 1.1

Description The `vspace` property reflects the `VSPACE` attribute of the `IMG` tag. For images created with the `Image` constructor, the value of the `vspace` property is 0.

Examples See the examples for the `height` property.

See also `Image.border`, `Image.height`, `Image.hspace`, `Image.width`

width

A string specifying the width of an image in pixels.

Property of	`Image`
Read-only	
Implemented in	JavaScript 1.1

Description The `width` property reflects the `WIDTH` attribute of the `IMG` tag. For images created with the `Image` constructor, the value of the `width` property is the actual, not the displayed, width of the image.

Examples See the examples for the `height` property.

See also `Image.border`, `Image.height`, `Image.hspace`, `Image.vspace`

java

A top-level object used to access any Java class in the package `java.*`.

Core object

Implemented in	JavaScript 1.1, NES 2.0

Created by The `java` object is a top-level, predefined JavaScript object. You can automatically access it without using a constructor or calling a method.

Description The `java` object is a convenience synonym for the property `Packages.java`.

See also `Packages`, `Packages.java`

JavaArray

A wrapped Java array accessed from within JavaScript code is a member of the type `JavaArray`.

Core object

Implemented in JavaScript 1.1, NES 2.0

Created by Any Java method which returns an array. In addition, you can create a `JavaArray` with an arbitrary data type using the `newInstance` method of the `Array` class:

```
public static Object newInstance(Class componentType,
   int length)
   throws NegativeArraySizeException
```

Description The `JavaArray` object is an instance of a Java array that is created in or passed to JavaScript. `JavaArray` is a wrapper for the instance; all references to the array instance are made through the `JavaArray`.

You must specify a class object, such as one returned by `java.lang.Object.forName`, for the `componentType` parameter of `newInstance` when you use this method to create an array. You cannot use a `JavaClass` object for the `componentType` parameter.

Use zero-based indexes to access the elements in a `JavaArray` object, just as you do to access elements in an array in Java. For example:

```
var javaString = new java.lang.String("Hello world!");
var byteArray  = javaString.getBytes();
byteArray[0] // returns 72
byteArray[1] // returns 101
```

Any Java data brought into JavaScript is converted to JavaScript data types. When the `JavaArray` is passed back to Java, the array is unwrapped and can be used by Java code. See the *Client-Side JavaScript Guide* for more information about data type conversions.

Property Summary

Property	Description
`length`	The number of elements in the Java array represented by `JavaArray`.

Method Summary

Method	Description
`toString`	Returns a string identifying the object as a `JavaArray`.

Examples **Example 1.** Instantiating a `JavaArray` in JavaScript.

In this example, the `JavaArray` `byteArray` is created by the `java.lang.String.getBytes` method, which returns an array.

```
var javaString = new java.lang.String("Hello world!");
var byteArray  = javaString.getBytes();
```

Example 2. Instantiating a `JavaArray` in JavaScript with the `newInstance` method.

Use a class object returned by `java.lang.Class.forName` as the argument for the `newInstance` method, as shown in the following code:

```
var dataType = java.lang.Class.forName("java.lang.String")
var dogs = java.lang.reflect.Array.newInstance(dataType, 5)
```

length

The number of elements in the Java array represented by the `JavaArray` object.

Property of	`JavaArray`
Implemented in	JavaScript 1.1, NES 2.0

Description Unlike `Array.length`, `JavaArray.length` is a read-only property. You cannot change the value of the `JavaArray.length` property because Java arrays have a fixed number of elements.

See also `Array.length`

toString

Returns a string representation of the JavaArray.

Method of	`JavaArray`
Implemented in	JavaScript 1.1, NES 2.0

Parameters None

Description The `toString` method is inherited from the `Object` object and returns the following value:

```
[object JavaArray]
```

JavaClass

A JavaScript reference to a Java class.
Core object

Implemented in JavaScript 1.1, NES 2.0

Created by A reference to the class name used with the `Packages` object:

```
Packages.JavaClass
```

where *JavaClass* is the fully-specified name of the object's Java class. The LiveConnect `java`, `sun`, and `netscape` objects provide shortcuts for commonly used Java packages and also create `JavaClass` objects.

Description A `JavaClass` object is a reference to one of the classes in a Java package, such as `netscape.javascript.JSObject`. A `JavaPackage` object is a reference to a Java package, such as `netscape.javascript`. In JavaScript, the `JavaPackage` and `JavaClass` hierarchy reflect the Java package and class hierarchy.

You must create a wrapper around an instance of `java.lang.Class` before you pass it as a parameter to a Java method—`JavaClass` objects are not automatically converted to instances of `java.lang.Class`.

Property Summary The properties of a `JavaClass` object are the static fields of the Java class.

Method Summary The methods of a `JavaClass` object are the static methods of the Java class.

Examples In the following example, `x` is a `JavaClass` object referring to `java.awt.Font`. Because `BOLD` is a static field in the `Font` class, it is also a property of the `JavaClass` object.

```
x = java.awt.Font
myFont = x("helv",x.BOLD,10) // creates a Font object
```

The previous example omits the `Packages` keyword and uses the `java` synonym because the `Font` class is in the `java` package.

See also `JavaArray`, `JavaObject`, `JavaPackage`, `Packages`

JavaObject

The type of a wrapped Java object accessed from within JavaScript code.
Core object

Implemented in	JavaScript 1.1, NES 2.0

Created by

Any Java method which returns an object type. In addition, you can explicitly construct a `JavaObject` using the object's Java constructor with the `Packages` keyword:

```
new Packages.JavaClass(parameterList)
```

where *JavaClass* is the fully-specified name of the object's Java class.

Parameters

`parameterList`	An optional list of parameters, specified by the constructor in the Java class.

Description

The `JavaObject` object is an instance of a Java class that is created in or passed to JavaScript. `JavaObject` is a wrapper for the instance; all references to the class instance are made through the `JavaObject`.

Any Java data brought into JavaScript is converted to JavaScript data types. When the `JavaObject` is passed back to Java, it is unwrapped and can be used by Java code. See the *Client-Side JavaScript Guide* for more information about data type conversions.

Property Summary

Inherits public data members from the Java class of which it is an instance as properties. It also inherits public data members from any superclass as properties.

Method Summary

Inherits public methods from the Java class of which it is an instance. The `JavaObject` also inherits methods from `java.lang.Object` and any other superclass.

Examples

Example 1. Instantiating a Java object in JavaScript.

The following code creates the `JavaObject` `theString`, which is an instance of the class `java.lang.String`:

```
var theString = new Packages.java.lang.String("Hello, world")
```

Because the `String` class is in the `java` package, you can also use the `java` synonym and omit the `Packages` keyword when you instantiate the class:

```
var theString = new java.lang.String("Hello, world")
```

Example 2. Accessing methods of a Java object.

Because the `JavaObject` `theString` is an instance of `java.lang.String`, it inherits all the public methods of `java.lang.String`. The following example uses the `startsWith` method to check whether `theString` begins with "Hello".

```
var theString = new java.lang.String("Hello, world")
theString.startsWith("Hello") // returns true
```

Example 3. Accessing inherited methods.

Because `getClass` is a method of `Object`, and `java.lang.String` extends `Object`, the `String` class inherits the `getClass` method. Consequently, `getClass` is also a method of the `JavaObject` which instantiates `String` in JavaScript.

```
var theString = new java.lang.String("Hello, world")
theString.getClass() // returns java.lang.String
```

See also `JavaArray`, `JavaClass`, `JavaPackage`, `Packages`

JavaPackage

A JavaScript reference to a Java package.

Core object

Implemented in	JavaScript 1.1, NES 2.0

Created by A reference to the package name used with the `Packages` keyword:

```
Packages.JavaPackage
```

where *JavaPackage* is the name of the object's Java package. If the package is in the `java`, `netscape`, or `sun` packages, the `Packages` keyword is optional.

Description In Java, a package is a collection of Java classes or other Java packages. For example, the `netscape` package contains the package `netscape.javascript`; the `netscape.javascript` package contains the classes `JSObject` and `JSException.`

In JavaScript, a `JavaPackage` is a reference to a Java package. For example, a reference to `netscape` is a `JavaPackage`. `netscape.javascript` is both a `JavaPackage` and a property of the `netscape` `JavaPackage`.

A `JavaClass` object is a reference to one of the classes in a package, such as `netscape.javascript.JSObject`. The `JavaPackage` and `JavaClass` hierarchy reflect the Java package and class hierarchy.

Although the packages and classes contained in a `JavaPackage` are its properties, you cannot use a `for...in` statement to enumerate them as you can enumerate the properties of other objects.

Property Summary The properties of a `JavaPackage` are the `JavaClass` objects and any other `JavaPackage` objects it contains.

Examples Suppose the Redwood corporation uses the Java `redwood` package to contain various Java classes that it implements. The following code creates the `JavaPackage` `red`:

```
var red = Packages.redwood
```

See also `JavaArray`, `JavaClass`, `JavaObject`, `Packages`

Layer

Corresponds to a layer in an HTML page and provides a means for manipulating that layer.

Client-side object

Implemented in JavaScript 1.2

Created by The HTML `LAYER` or `ILAYER` tag, or using cascading style sheet syntax. The JavaScript runtime engine creates a `Layer` object corresponding to each layer in your document. It puts these objects in an array in the `document.layers` property. You access a `Layer` object by indexing this array.

To define a layer, use standard HTML syntax. If you specify the `ID` attribute, you can use the value of that attribute to index into the `layers` array.

For a complete description of layers, see *Dynamic HTML in Netscape Communicator*.

Some `layer` properties can be directly modified by assignment; for example, "`mylayer.visibility = hide`". A layer object also has methods that can affect these properties.

Event handlers

- `onMouseOver`
- `onMouseOut`
- `onLoad`
- `onFocus`
- `onBlur`

Property Summary

Property	Description
`above`	The `layer` object above this one in z-order, among all layers in the document or the enclosing window object if this layer is topmost.
`background`	The image to use as the background for the layer's canvas.
`bgColor`	The color to use as a solid background color for the layer's canvas.
`below`	The `layer` object below this one in z-order, among all layers in the document or null if this layer is at the bottom.
`clip.bottom`	The bottom edge of the clipping rectangle (the part of the layer that is visible.)

Property	Description
`clip.height`	The height of the clipping rectangle (the part of the layer that is visible.)
`clip.left`	The left edge of the clipping rectangle (the part of the layer that is visible.)
`clip.right`	The right edge of the clipping rectangle (the part of the layer that is visible.)
`clip.top`	The top edge of the clipping rectangle (the part of the layer that is visible.)
`clip.width`	The width of the clipping rectangle (the part of the layer that is visible.)
`document`	The layer's associated document.
`left`	The horizontal position of the layer's left edge, in pixels, relative to the origin of its parent layer.
`name`	A string specifying the name assigned to the layer through the `ID` attribute in the `LAYER` tag.
`pageX`	The horizontal position of the layer, in pixels, relative to the page.
`pageY`	The vertical position of the layer, in pixels, relative to the page.
`parentLayer`	The `layer` object that contains this layer, or the enclosing `window` object if this layer is not nested in another layer.
`siblingAbove`	The `layer` object above this one in z-order, among all layers that share the same parent layer, or null if the layer has no sibling above.
`siblingBelow`	The `layer` object below this one in z-order, among all layers that share the same parent layer, or null if layer is at the bottom.
`src`	A string specifying the URL of the layer's content.
`top`	The vertical position of the layer's top edge, in pixels, relative to the origin of its parent layer.
`visibility`	Whether or not the layer is visible.
`window`	The `window` or `Frame` object that contains the layer, regardless of whether the layer is nested within another layer.
`x`	A convenience synonym for `Layer.left`.
`y`	A convenience synonym for `Layer.top`.
`zIndex`	The relative z-order of this layer with respect to its siblings.

Method Summary

Method	Description
`captureEvents`	Sets the window or document to capture all events of the specified type.
`handleEvent`	Invokes the handler for the specified event.
`load`	Changes the source of a layer to the contents of the specified file, and simultaneously changes the width at which the layer's HTML contents will be wrapped.
`moveAbove`	Stacks this layer above the layer specified in the argument, without changing either layer's horizontal or vertical position.
`moveBelow`	Stacks this layer below the specified layer, without changing either layer's horizontal or vertical position.
`moveBy`	Changes the layer position by applying the specified deltas, measured in pixels.
`moveTo`	Moves the top-left corner of the window to the specified screen coordinates.
`moveToAbsolute`	Changes the layer position to the specified pixel coordinates within the page (instead of the containing layer.)
`releaseEvents`	Sets the layer to release captured events of the specified type, sending the event to objects further along the event hierarchy.
`resizeBy`	Resizes the layer by the specified height and width values (in pixels).
`resizeTo`	Resizes the layer to have the specified height and width values (in pixels).
`routeEvent`	Passes a captured event along the normal event hierarchy.

In addition, this object inherits the `watch` and `unwatch` methods from `Object`.

Note Just as in the case of a document, if you want to define mouse click response for a layer, you must capture `onMouseDown` and `onMouseUp` events at the level of the layer and process them as you want.

For details about capturing events, see the *Client-Side JavaScript Guide*.

If an event occurs in a point where multiple layers overlap, the topmost layer gets the event, even if it is transparent. However, if a layer is hidden, it does not get events.

above

The `layer` object above this one in z-order, among all layers in the document or the enclosing window object if this layer is topmost.

Property of	`Layer`
Read-only	
Implemented in	JavaScript 1.2

background

The image to use as the background for the layer's canvas (which is the part of the layer within the clip rectangle).

Property of	`Layer`
Implemented in	JavaScript 1.2

Description Each layer has a background property, whose value is an image object, whose `src` attribute is a URL that indicates the image to use to provide a tiled backdrop. The value is null if the layer has no backdrop. For example:

```
layer.background.src = "fishbg.gif";
```

below

The `layer` object below this one in z-order, among all layers in the document or null if this layer is at the bottom.

Property of	`Layer`
Read-only	
Implemented in	JavaScript 1.2

bgColor

A string specifying the color to use as a solid background color for the layer's canvas (the part of the layer within the clip rectangle).

Property of	`Layer`
Implemented in	JavaScript 1.2

Description The `bgColor` property is expressed as a hexadecimal RGB triplet or as a string literal (see the *Client-Side JavaScript Guide*). This property is the JavaScript reflection of the `BGCOLOR` attribute of the `BODY` tag.

You can set the `bgColor` property at any time.

If you express the color as a hexadecimal RGB triplet, you must use the format `rrggbb`. For example, the hexadecimal RGB values for salmon are `red=FA`, `green=80`, and `blue=72`, so the RGB triplet for salmon is `"FA8072"`.

Examples The following example sets the background color of the `myLayer` layer's canvas to aqua using a string literal:

```
myLayer.bgColor="aqua"
```

The following example sets the background color of the `myLayer` layer's canvas to aqua using a hexadecimal triplet:

```
myLayer.bgColor="00FFFF"
```

See also `Layer.bgColor`

captureEvents

Sets the window or document to capture all events of the specified type.

Method of	`Layer`
Implemented in	JavaScript 1.2

Syntax `captureEvents(`*`eventType`*`)`

Parameters

`eventType`	Type of event to be captured. Available event types are listed in the *Client-Side JavaScript Guide*.

Description When a window with frames wants to capture events in pages loaded from different locations (servers), you need to use `captureEvents` in a signed script and precede it with `enableExternalCapture`. For more information and an example, see `enableExternalCapture`.

`captureEvents` works in tandem with `releaseEvents`, `routeEvent`, and `handleEvent`. For information on handling events, see the *Client-Side JavaScript Guide*.

clip.bottom

The bottom edge of the clipping rectangle (the part of the layer that is visible.) Any part of a layer that is outside the clipping rectangle is not displayed.

Property of	`Layer`
Implemented in	JavaScript 1.2

clip.height

The height of the clipping rectangle (the part of the layer that is visible.) Any part of a layer that is outside the clipping rectangle is not displayed.

Property of	`Layer`
Implemented in	JavaScript 1.2

clip.left

The left edge of the clipping rectangle (the part of the layer that is visible.) Any part of a layer that is outside the clipping rectangle is not displayed.

Property of	`Layer`
Implemented in	JavaScript 1.2

clip.right

The right edge of the clipping rectangle (the part of the layer that is visible.) Any part of a layer that is outside the clipping rectangle is not displayed.

Property of	`Layer`
Implemented in	JavaScript 1.2

clip.top

The top edge of the clipping rectangle (the part of the layer that is visible.) Any part of a layer that is outside the clipping rectangle is not displayed.

Property of	`Layer`
Implemented in	JavaScript 1.2

clip.width

The width of the clipping rectangle (the part of the layer that is visible.) Any part of a layer that is outside the clipping rectangle is not displayed.

Property of	`Layer`
Implemented in	JavaScript 1.2

document

The layer's associated document.

Property of	`Layer`
Read-only	
Implemented in	JavaScript 1.2

Description Each `layer` object contains its own `document` object. This object can be used to access the images, applets, embeds, links, anchors and layers that are contained within the layer. Methods of the `document` object can also be invoked to change the contents of the layer.

handleEvent

Invokes the handler for the specified event.

Method of	`Layer`
Implemented in	JavaScript 1.2

Syntax `handleEvent(`*`event`*`)`

Parameters

`event`	Name of an event for which the specified object has an event handler.

Description `handleEvent` works in tandem with `captureEvents`, `releaseEvents`, and `routeEvent`. For information on handling events, see the *Client-Side JavaScript Guide*.

left

The horizontal position of the layer's left edge, in pixels, relative to the origin of its parent layer.

Property of	`Layer`
Implemented in	JavaScript 1.2

The `Layer.x` property is a convenience synonym for the `left` property.

See also `Layer.top`

load

Changes the source of a layer to the contents of the specified file and simultaneously changes the width at which the layer's HTML contents are wrapped.

Method of	`Layer`
Implemented in	JavaScript 1.2

Syntax `load(`*`sourcestring`*`, `*`width`*`)`

Parameters

`sourcestring`	A string indicating the external file name.
`width`	The width of the layer as a pixel value.

moveAbove

Stacks this layer above the layer specified in the argument, without changing either layer's horizontal or vertical position. After re-stacking, both layers will share the same parent layer.

Method of	`Layer`
Implemented in	JavaScript 1.2

Syntax `moveAbove(`*`aLayer`*`)`

Parameters

`aLayer`	The layer above which to move the current layer.

moveBelow

Stacks this layer below the specified layer, without changing either layer's horizontal or vertical position. After re-stacking, both layers will share the same parent layer.

Method of	`Layer`
Implemented in	JavaScript 1.2

Syntax `moveBelow(`*`aLayer`*`)`

Parameters

`aLayer`	The layer below which to move the current layer.

moveBy

Changes the layer position by applying the specified deltas, measured in pixels.

Method of	`Layer`
Implemented in	JavaScript 1.2

Syntax `moveBy(`*`horizontal`*`, `*`vertical`*`)`

Parameters

`horizontal`	The number of pixels by which to move the layer horizontally.
`vertical`	The number of pixels by which to move the layer vertically.

moveTo

Moves the top-left corner of the window to the specified screen coordinates.

Method of	`Layer`
Implemented in	JavaScript 1.2

Syntax `moveTo(`*`x-coordinate`*`,` *`y-coordinate`*`)`

Parameters

`x-coordinate`	An integer representing the top edge of the window in screen coordinates.
`y-coordinate`	An integer representing the left edge of the window in screen coordinates.

Security To move a window offscreen, call the `moveTo` method in a signed script. For information on security, see the *Client-Side JavaScript Guide*.

Description Changes the layer position to the specified pixel coordinates within the containing layer. For ILayers, moves the layer relative to the natural inflow position of the layer.

See also `Layer.moveBy`

moveToAbsolute

Changes the layer position to the specified pixel coordinates within the page (instead of the containing layer.)

Method of	`Layer`
Implemented in	JavaScript 1.2

Syntax `moveToAbsolute(`*`x`*`,` *`y`*`)`

Parameters

`x`	An integer representing the top edge of the window in pixel coordinates.
`y`	An integer representing the left edge of the window in pixel coordinates.

Description This method is equivalent to setting both the `pageX` and `pageY` properties of the `layer` object.

name

A string specifying the name assigned to the layer through the `ID` attribute in the `LAYER` tag.

Property of	`Layer`
Read-only	
Implemented in	JavaScript 1.2

pageX

The horizontal position of the layer, in pixels, relative to the page.

Property of	`Layer`
Implemented in	JavaScript 1.2

pageY

The vertical position of the layer, in pixels, relative to the page.

Property of	`Layer`
Implemented in	JavaScript 1.2

parentLayer

The `layer` object that contains this layer, or the enclosing `window` object if this layer is not nested in another layer.

Property of	`Layer`
Read-only	
Implemented in	JavaScript 1.2

releaseEvents

Sets the window or document to release captured events of the specified type, sending the event to objects further along the event hierarchy.

Method of	`Layer`
Implemented in	JavaScript 1.2

Syntax `releaseEvents(`*`eventType`*`)`

Parameters

`eventType`	Type of event to be captured.

Description If the original target of the event is a window, the window receives the event even if it is set to release that type of event. `releaseEvents` works in tandem with `captureEvents`, `routeEvent`, and `handleEvent`. For more information, see the *Client-Side JavaScript Guide*.

resizeBy

Resizes the layer by the specified height and width values (in pixels).

Method of	`Layer`
Implemented in	JavaScript 1.2

Syntax `resizeBy(`*`width`*`, `*`height`*`)`

Parameters

`width`	The number of pixels by which to resize the layer horizontally.
`height`	The number of pixels by which to resize the layer vertically.

Description This does not layout any HTML contained in the layer again. Instead, the layer contents may be clipped by the new boundaries of the layer. This method has the same effect as adding `width` and `height` to `clip.width` and `clip.height`.

resizeTo

Resizes the layer to have the specified height and width values (in pixels).

Method of	`Layer`
Implemented in	JavaScript 1.2

Description This does not layout any HTML contained in the layer again. Instead, the layer contents may be clipped by the new boundaries of the layer.

Syntax `resizeTo(`*`width`*`, `*`height`*`)`

Parameters

`width`	An integer representing the layer's width in pixels.
`height`	An integer representing the layer's height in pixels.

Description This method has the same effect setting `clip.width` and `clip.height`.

routeEvent

Passes a captured event along the normal event hierarchy.

Method of	`Layer`
Implemented in	JavaScript 1.2

Syntax `routeEvent(`*`event`*`)`

Parameters

`event`	The event to route.

Description If a sub-object (document or layer) is also capturing the event, the event is sent to that object. Otherwise, it is sent to its original target.

`routeEvent` works in tandem with `captureEvents`, `releaseEvents`, and `handleEvent`. For more information, see the *Client-Side JavaScript Guide*.

siblingAbove

The layer object above this one in z-order, among all layers that share the same parent layer or null if the layer has no sibling above.

Property of	`Layer`
Read-only	
Implemented in	JavaScript 1.2

siblingBelow

The `layer` object below this one in z-order, among all layers that share the same parent layer or null if layer is at the bottom.

Property of	`Layer`
Read-only	
Implemented in	JavaScript 1.2

src

A URL string specifying the source of the layer's content. Corresponds to the `SRC` attribute.

Property of	`Layer`
Implemented in	JavaScript 1.2

top

The vertical position of the layer's left edge, in pixels, relative to the origin of its parent layer.

Property of	`Layer`
Implemented in	JavaScript 1.2

The `Layer.y` property is a convenience synonym for the `top` property.

See also `Layer.left`

visibility

Whether or not the layer is visible.

Property of	`Layer`
Implemented in	JavaScript 1.2

Description A value of `show` means show the layer; `hide` means hide the layer; `inherit` means inherit the visibility of the parent layer.

window

The `window` or `Frame` object that contains the layer, regardless of whether the layer is nested within another layer.

Property of	`Layer`
Read-only	
Implemented in	JavaScript 1.2

x

The horizontal position of the layer's left edge, in pixels, relative to the origin of its parent layer.

Property of	`Layer`
Implemented in	JavaScript 1.2

The `x` property is a convenience synonym for the `Layer.left` property.

See also `Layer.y`

y

The vertical position of the layer's left edge, in pixels, relative to the origin of its parent layer.

Property of	`Layer`
Implemented in	JavaScript 1.2

The `y` property is a convenience synonym for the `Layer.top` property.

See also `Layer.x`

zIndex

The relative z-order of this layer with respect to its siblings.

Method of	`Layer`
Implemented in	JavaScript 1.2

Description Sibling layers with lower numbered z-indexes are stacked underneath this layer. The value of `zIndex` must be 0 or a positive integer.

Link

A piece of text, an image, or an area of an image identified as a hypertext link. When the user clicks the link text, image, or area, the link hypertext reference is loaded into its target window. `Area` objects are a type of `Link` object.

Client-side object

Implemented in	JavaScript 1.0 JavaScript 1.1: added `onMouseOut` event handler; added `Area` objects; `links` array contains areas created with `<AREA HREF="...">` JavaScript 1.2: added `x` and `y` properties; added `handleEvent` method

Created by By using the HTML `A` or `AREA` tag or by a call to the `String.link` method. The JavaScript runtime engine creates a `Link` object corresponding to each `A` and `AREA` tag in your document that supplies the `HREF` attribute. It puts these objects as an array in the `document.links` property. You access a `Link` object by indexing this array.

To define a link with the `A` or `AREA` tag, use standard HTML syntax with the addition of JavaScript event handlers.

To define a link with the `String.link` method:

```
theString.link(hrefAttribute)
```

where:

`theString`	A `String` object.
`hrefAttribute`	Any string that specifies the `HREF` attribute of the `A` tag; it should be a valid URL (relative or absolute).

Event handlers `Area` objects have the following event handlers:

- `onDblClick`
- `onMouseOut`
- `onMouseOver`

`Link` objects have the following event handlers:

- `onClick`
- `onDblClick`
- `onKeyDown`
- `onKeyPress`
- `onKeyUp`
- `onMouseDown`
- `onMouseOut`
- `onMouseUp`
- `onMouseOver`

Description Each `Link` object is a `location` object and has the same properties as a `location` object.

If a `Link` object is also an `Anchor` object, the object has entries in both the `anchors` and `links` arrays.

When a user clicks a `Link` object and navigates to the destination document (specified by `HREF="locationOrURL"`), the destination document's `referrer` property contains the URL of the source document. Evaluate the `referrer` property from the destination document.

You can use a `Link` object to execute a JavaScript function rather than link to a hypertext reference by specifying the `javascript:` URL protocol for the link's `HREF` attribute. You might want to do this if the link surrounds an `Image` object and you want to execute JavaScript code when the image is clicked. Or you might want to use a link instead of a button to execute JavaScript code.

For example, when a user clicks the following links, the `slower` and `faster` functions execute:

```
<A HREF="javascript:slower()">Slower</A>
<A HREF="javascript:faster()">Faster</A>
```

You can use a `Link` object to do nothing rather than link to a hypertext reference by specifying the `javascript:void(0)` URL protocol for the link's `HREF` attribute. You might want to do this if the link surrounds an `Image` object and you want to use the link's event handlers with the image. When a user clicks the following link or image, nothing happens:

```
<A HREF="javascript:void(0)">Click here to do nothing</A>
```

```
<A HREF="javascript:void(0)">
   <IMG SRC="images\globe.gif" ALIGN="top" HEIGHT="50" WIDTH="50">
</A>
```

Property Summary

Property	Description
`hash`	Specifies an anchor name in the URL.
`host`	Specifies the host and domain name, or IP address, of a network host.
`hostname`	Specifies the `host:port` portion of the URL.
`href`	Specifies the entire URL.
`pathname`	Specifies the URL-path portion of the URL.
`port`	Specifies the communications port that the server uses.
`protocol`	Specifies the beginning of the URL, including the colon.
`search`	Specifies a query string.
`target`	Reflects the `TARGET` attribute.
`text`	A string containing the content of the corresponding `A` tag.
`x`	The horizontal position of the link's left edge, in pixels, relative to the left edge of the document.
`y`	The vertical position of the link's top edge, in pixels, relative to the top edge of the document.

Method Summary

Method	Description
`handleEvent`	Invokes the handler for the specified event.

In addition, this object inherits the `watch` and `unwatch` methods from `Object`.

Examples

Example 1. The following example creates a hypertext link to an anchor named `javascript_intro`:

```
<A HREF="#javascript_intro">Introduction to JavaScript</A>
```

Example 2. The following example creates a hypertext link to an anchor named `numbers` in the file `doc3.html` in the window `window2`. If `window2` does not exist, it is created.

```
<LI><A HREF=doc3.html#numbers TARGET="window2">Numbers</A>
```

Example 3. The following example takes the user back `x` entries in the history list:

```
<A HREF="javascript:history.go(-1 * x)">Click here</A>
```

Example 4. The following example creates a hypertext link to a URL. The user can use the set of radio buttons to choose between three URLs. The link's `onClick` event handler sets the URL (the link's `href` property) based on the selected radio button. The link also has an `onMouseOver` event handler that changes the window's `status` property. As the example shows, you must return true to set the `window.status` property in the `onMouseOver` event handler.

```
<SCRIPT>
var destHREF="http://home.netscape.com/"
</SCRIPT>
<FORM NAME="form1">
<B>Choose a destination from the following list, then click "Click me" below.</B>
<BR><INPUT TYPE="radio" NAME="destination" VALUE="netscape"
   onClick="destHREF='http://home.netscape.com/'"> Netscape home page
<BR><INPUT TYPE="radio" NAME="destination" VALUE="sun"
   onClick="destHREF='http://www.sun.com/'"> Sun home page
<BR><INPUT TYPE="radio" NAME="destination" VALUE="rfc1867"
   onClick="destHREF='http://www.ics.uci.edu/pub/ietf/html/rfc1867.txt'"> RFC 1867
<P><A HREF=""
   onMouseOver="window.status='Click this if you dare!'; return true"
   onClick="this.href=destHREF">
   <B>Click me</B></A>
</FORM>
```

Example 5: links array. In the following example, the `linkGetter` function uses the `links` array to display the value of each link in the current document. The example also defines several links and a button for running `linkGetter.`

```
function linkGetter() {
   msgWindow=window.open("","msg","width=400,height=400")
   msgWindow.document.write("links.length is " +
      document.links.length + "<BR>")
   for (var i = 0; i < document.links.length; i++) {
      msgWindow.document.write(document.links[i] + "<BR>")
   }
}
```

```
<A HREF="http://home.netscape.com">Netscape Home Page</A>
<A HREF="http://www.catalog.com/fwcfc/">China Adoptions</A>
<A HREF="http://www.supernet.net/~dugbrown/">Bad Dog Chronicles</A>
<A HREF="http://www.best.com/~doghouse/homecnt.shtml">Lab Rescue</A>
<P>
<INPUT TYPE="button" VALUE="Display links"
   onClick="linkGetter()">
```

Example 6: Refer to Area object with links array. The following code refers to the `href` property of the first `Area` object shown in Example 1.

```
document.links[0].href
```

Example 7: Area object with onMouseOver and onMouseOut event handlers. The following example displays an image, `globe.gif`. The image uses an image map that defines areas for the top half and the bottom half of the image. The `onMouseOver` and `onMouseOut` event handlers display different status bar messages depending on whether the mouse passes over or leaves the top half or bottom half of the image. The `HREF` attribute is required when using the `onMouseOver` and `onMouseOut` event handlers, but in this example the image does not need a hypertext link, so the `HREF` attribute executes `javascript:void(0)`, which does nothing.

```
<MAP NAME="worldMap">
   <AREA NAME="topWorld" COORDS="0,0,50,25" HREF="javascript:void(0)"
      onMouseOver="self.status='You are on top of the world';return true"
      onMouseOut="self.status='You have left the top of the world';return true">
   <AREA NAME="bottomWorld" COORDS="0,25,50,50" HREF="javascript:void(0)"
      onMouseOver="self.status='You are on the bottom of the world';return true"
      onMouseOut="self.status='You have left the bottom of the world';return true">
</MAP>
<IMG SRC="images\globe.gif" ALIGN="top" HEIGHT="50" WIDTH="50" USEMAP="#worldMap">
```

Example 8: Simulate an Area object's onClick using the HREF attribute. The following example uses an `Area` object's HREF attribute to execute a JavaScript function. The image displayed, `colors.gif`, shows two sample colors. The top half of the image is the color antiquewhite, and the bottom half is white. When the user clicks the top or bottom half of the image, the function `setBGColor` changes the document's background color to the color shown in the image.

```
<SCRIPT>
function setBGColor(theColor) {
   document.bgColor=theColor
}
</SCRIPT>
Click the color you want for this document's background color
<MAP NAME="colorMap">
   <AREA NAME="topColor" COORDS="0,0,50,25" HREF="javascript:setBGColor('antiquewhite')">
   <AREA NAME="bottomColor" COORDS="0,25,50,50" HREF="javascript:setBGColor('white')">
</MAP>
<IMG SRC="images\colors.gif" ALIGN="top" HEIGHT="50" WIDTH="50" USEMAP="#colorMap">
```

See also `Anchor`, `Image`, `link`

handleEvent

Invokes the handler for the specified event.

Method of	`Link`
Implemented in	JavaScript 1.2

Syntax `handleEvent(`*`event`*`)`

Parameters

`event`	The name of an event for which the specified object has an event handler.

Description For information on handling events, see the *Client-Side JavaScript Guide*.

hash

A string beginning with a hash mark (#) that specifies an anchor name in the URL.

Property of	`Link`
Implemented in	JavaScript 1.0

Security **JavaScript 1.1.** This property is tainted by default. For information on data tainting, see the *Client-Side JavaScript Guide*.

Description The `hash` property specifies a portion of the URL. This property applies to HTTP URLs only.

Be careful using this property. Assume `document.links[0]` contains:

```
http://royalairways.com/fish.htm#angel
```

Then `document.links[0].hash` returns `#angel`. Assume you have this code:

```
hash = document.links[0].hash;
document.links[0].hash = hash;
```

Now, `document.links[0].hash` returns `##angel`.

This behavior may change in a future release.

You can set the `hash` property at any time, although it is safer to set the `href` property to change a location. If the hash that you specify cannot be found in the current location, you get an error.

Setting the `hash` property navigates to the named anchor without reloading the document. This differs from the way a document is loaded when other `link` properties are set.

See RFC 1738 (`http://www.cis.ohio-state.edu/htbin/rfc/rfc1738.html`) for complete information about the hash.

See also `Link.host`, `Link.hostname`, `Link.href`, `Link.pathname`, `Link.port`, `Link.protocol`, `Link.search`

host

A string specifying the server name, subdomain, and domain name.

Property of	`Link`
Implemented in	JavaScript 1.0

Security **JavaScript 1.1.** This property is tainted by default. For information on data tainting, see the *Client-Side JavaScript Guide*.

Description The `host` property specifies a portion of a URL. The `host` property is a substring of the `hostname` property. The `hostname` property is the concatenation of the `host` and `port` properties, separated by a colon. When the `port` property is null, the `host` property is the same as the `hostname` property.

You can set the `host` property at any time, although it is safer to set the `href` property to change a location. If the host that you specify cannot be found in the current location, you get an error.

See Section 3.1 of RFC 1738 (`http://www.cis.ohio-state.edu/htbin/rfc/rfc1738.html`) for complete information about the hostname and port.

See also `Link.hash`, `Link.hostname`, `Link.href`, `Link.pathname`, `Link.port`, `Link.protocol`, `Link.search`

hostname

A string containing the full hostname of the server, including the server name, subdomain, domain, and port number.

Property of	`Link`
Implemented in	JavaScript 1.0

Security **JavaScript 1.1.** This property is tainted by default. For information on data tainting, see the *Client-Side JavaScript Guide*.

Description The `hostname` property specifies a portion of a URL. The `hostname` property is the concatenation of the `host` and `port` properties, separated by a colon. When the `port` property is 80 (the default), the `host` property is the same as the `hostname` property.

You can set the `hostname` property at any time, although it is safer to set the `href` property to change a location. If the hostname that you specify cannot be found in the current location, you get an error.

See Section 3.1 of RFC 1738 (`http://www.cis.ohio-state.edu/htbin/rfc/rfc1738.html`) for complete information about the hostname.

See also `Link.host`, `Link.hash`, `Link.href`, `Link.pathname`, `Link.port`, `Link.protocol`, `Link.search`

href

A string specifying the entire URL.

Property of	`Link`
Implemented in	JavaScript 1.0

Security **JavaScript 1.1.** This property is tainted by default. For information on data tainting, see the *Client-Side JavaScript Guide*.

Description The `href` property specifies the entire URL. Other `link` object properties are substrings of the `href` property.

You can set the `href` property at any time.

See RFC 1738 (`http://www.cis.ohio-state.edu/htbin/rfc/rfc1738.html`) for complete information about the URL.

See also `Link.hash`, `Link.host`, `Link.hostname`, `Link.pathname`, `Link.port`, `Link.protocol`, `Link.search`

pathname

A string specifying the URL-path portion of the URL.

Property of	`Link`
Implemented in	JavaScript 1.0

Security **JavaScript 1.1.** This property is tainted by default. For information on data tainting, see the *Client-Side JavaScript Guide*.

Description The `pathname` property specifies a portion of the URL. The pathname supplies the details of how the specified resource can be accessed.

You can set the `pathname` property at any time, although it is safer to set the `href` property to change a location. If the pathname that you specify cannot be found in the current location, you get an error.

See Section 3.1 of RFC 1738 (`http://www.cis.ohio-state.edu/htbin/rfc/rfc1738.html`) for complete information about the pathname.

See also `Link.host`, `Link.hostname`, `Link.hash`, `Link.href`, `Link.port`, `Link.protocol`, `Link.search`

port

A string specifying the communications port that the server uses.

Property of	`Link`
Implemented in	JavaScript 1.0

Security **JavaScript 1.1.** This property is tainted by default. For information on data tainting, see the *Client-Side JavaScript Guide*.

Description The `port` property specifies a portion of the URL. The `port` property is a substring of the `hostname` property. The `hostname` property is the concatenation of the `host` and `port` properties, separated by a colon. When the `port` property is 80 (the default), the `host` property is the same as the `hostname` property.

You can set the `port` property at any time, although it is safer to set the `href` property to change a location. If the port that you specify cannot be found in the current location, you will get an error. If the `port` property is not specified, it defaults to 80 on the server.

See Section 3.1 of RFC 1738 (`http://www.cis.ohio-state.edu/htbin/rfc/rfc1738.html`) for complete information about the port.

See also `Link.host`, `Link.hostname`, `Link.hash`, `Link.href`, `Link.pathname`, `Link.protocol`, `Link.search`

protocol

A string specifying the beginning of the URL, up to and including the first colon.

Property of	`Link`
Implemented in	JavaScript 1.0

Security **JavaScript 1.1.** This property is tainted by default. For information on data tainting, see the *Client-Side JavaScript Guide*.

Description The `protocol` property specifies a portion of the URL. The protocol indicates the access method of the URL. For example, the value `"http:"` specifies HyperText Transfer Protocol, and the value `"javascript:"` specifies JavaScript code.

You can set the `protocol` property at any time, although it is safer to set the `href` property to change a location. If the protocol that you specify cannot be found in the current location, you get an error.

The `protocol` property represents the scheme name of the URL. See Section 2.1 of RFC 1738 (`http://www.cis.ohio-state.edu/htbin/rfc/rfc1738.html`) for complete information about the protocol.

See also `Link.host`, `Link.hostname`, `Link.hash`, `Link.href`, `Link.pathname`, `Link.port`, `Link.search`

search

A string beginning with a question mark that specifies any query information in the URL.

Property of	`Link`
Implemented in	JavaScript 1.0

Security **JavaScript 1.1.** This property is tainted by default. For information on data tainting, see the *Client-Side JavaScript Guide*.

Description The `search` property specifies a portion of the URL. This property applies to http URLs only.

The `search` property contains variable and value pairs; each pair is separated by an ampersand. For example, two pairs in a search string could look like the following:

```
?x=7&y=5
```

You can set the `search` property at any time, although it is safer to set the `href` property to change a location. If the search that you specify cannot be found in the current location, you get an error.

See Section 3.3 of RFC 1738 (`http://www.cis.ohio-state.edu/htbin/rfc/rfc1738.html`) for complete information about the search.

See also `Link.host`, `Link.hostname`, `Link.hash`, `Link.href`, `Link.pathname`, `Link.port`, `Link.protocol`

target

A string specifying the name of the window that displays the content of a clicked hypertext link.

Property of	`Link`
Implemented in	JavaScript 1.0

Description The `target` property initially reflects the `TARGET` attribute of the `A` or `AREA` tags; however, setting `target` overrides this attribute.

You can set `target` using a string, if the string represents a window name. The `target` property cannot be assigned the value of a JavaScript expression or variable.

You can set the `target` property at any time.

Examples The following example specifies that responses to the `musicInfo` form are displayed in the `msgWindow` window:

```
document.musicInfo.target="msgWindow"
```

See also `Form`

text

A string containing the content of the corresponding `A` tag.

Property of	`Link`
Implemented in	JavaScript 1.2

x

The horizontal position of the link's left edge, in pixels, relative to the left edge of the document.

Property of	`Link`
Read-only	
Implemented in	JavaScript 1.2

See also `Link.y`

y

The vertical position of the link's top edge, in pixels, relative to the top edge of the document.

Property of	`Link`
Read-only	
Implemented in	JavaScript 1.2

See also `Link.x`

Location

Contains information on the current URL.

Client-side object

Implemented in	JavaScript 1.0
	JavaScript 1.1: added `reload`, `replace` methods

Created by `Location` objects are predefined JavaScript objects that you access through the `location` property of a `window` object.

Description The `location` object represents the complete URL associated with a given `window` object. Each property of the `location` object represents a different portion of the URL.

In general, a URL has this form:

`protocol//host``:`*`port/pathname#hash`*`?`*`search`*

For example:

```
http://home.netscape.com/assist/extensions.html#topic1?x=7&y=2
```

These parts serve the following purposes:

- `protocol` represents the beginning of the URL, up to and including the first colon.
- `host` represents the host and domain name, or IP address, of a network host.
- `port` represents the communications port that the server uses for communications.
- `pathname` represents the URL-path portion of the URL.
- `hash` represents an anchor name fragment in the URL, including the hash mark (#). This property applies to HTTP URLs only.
- `search` represents any query information in the URL, including the question mark (?). This property applies to HTTP URLs only. The search string contains variable and value pairs; each pair is separated by an ampersand (&).

A `Location` object has a property for each of these parts of the URL. See the individual properties for more information. A `Location` object has two other properties not shown here:

- `href` represents a complete URL.
- `hostname` represents the concatenation `host`:`port`.

If you assign a string to the `location` property of an object, JavaScript creates a `location` object and assigns that string to its `href` property. For example, the following two statements are equivalent and set the URL of the current window to the Netscape home page:

```
window.location.href="http://home.netscape.com/"
window.location="http://home.netscape.com/"
```

The `location` object is contained by the `window` object and is within its scope. If you refer to a `location` object without specifying a window, the `location` object represents the current location. If you refer to a `location` object and specify a window name, as in `windowReference.location`, the `location` object represents the location of the specified window.

In event handlers, you must specify `window.location` instead of simply using `location`. Due to the scoping of static objects in JavaScript, a call to `location` without specifying an object name is equivalent to `document.location`, which is a synonym for `document.URL`.

`Location` is not a property of the `document` object; its equivalent is the `document.URL` property. The `document.location` property, which is a synonym for `document.URL`, is deprecated.

How documents are loaded when location is set. When you set the `location` object or any of its properties except `hash,` whether a new document is loaded depends on which version of the browser you are running:

- In JavaScript 1.0, setting `location` does a conditional ("If-modified-since") HTTP GET operation, which returns no data from the server unless the document has been modified since the last version downloaded.
- In JavaScript 1.1 and later, the effect of setting `location` depends on the user's setting for comparing a document to the original over the network. The user interface option for setting this preference differs in browser versions. The user decides whether to check a document in cache every

time it is accessed, once per session, or never. The document is reloaded from cache if the user sets never or once per session; the document is reloaded from the server only if the user chooses every time.

Syntax for common URL types. When you specify a URL, you can use standard URL formats and JavaScript statements. The following table shows the syntax for specifying some of the most common types of URLs.

Table 1.1 URL syntax.

URL type	Protocol	Example
JavaScript code	javascript:	javascript:history.go(-1)
Navigator source viewer	view-source:	`view-source:wysiwyg://0/file:/c|/temp/genhtml.html`
Navigator info	about:	about:cache
World Wide Web	http:	`http://home.netscape.com/`
File	file:/	`file:///javascript/methods.html`
FTP	ftp:	`ftp://ftp.mine.com/home/mine`
MailTo	mailto:	mailto:info@netscape.com
Usenet	news:	`news://news.scruznet.com/comp.lang.javascript`
Gopher	gopher:	gopher.myhost.com

The following list explains some of the protocols:

- The `javascript:` protocol evaluates the expression after the colon (:), if there is one, and loads a page containing the string value of the expression, unless it is undefined. If the expression evaluates to undefined (by calling a void function, for example `javascript:void(0)`), no new page loads. Note that loading a new page over your script's page clears the page's variables, functions, and so on.

- The `view-source:` protocol displays HTML code that was generated with JavaScript `document.write` and `document.writeln` methods. For information on printing and saving generated HTML, see `document.write`.

- The `about:` protocol provides information on Navigator. For example:
  - — `about:` by itself is the same as choosing About Communicator from the Navigator Help menu.
  - — `about:cache` displays disk-cache statistics.
  - — `about:plugins` displays information about plug-ins you have configured. This is the same as choosing About Plug-ins from the Navigator Help menu.

Property Summary

Property	Description
`hash`	Specifies an anchor name in the URL.
`host`	Specifies the host and domain name, or IP address, of a network host.
`hostname`	Specifies the `host:port` portion of the URL.
`href`	Specifies the entire URL.
`pathname`	Specifies the URL-path portion of the URL.
`port`	Specifies the communications port that the server uses.
`protocol`	Specifies the beginning of the URL, including the colon.
`search`	Specifies a query.

Method Summary

Method	Description
`reload`	Forces a reload of the window's current document.
`replace`	Loads the specified URL over the current history entry.

In addition, this object inherits the `watch` and `unwatch` methods from `Object`.

Examples **Example 1.** The following two statements are equivalent and set the URL of the current window to the Netscape home page:

```
window.location.href="http://home.netscape.com/"
window.location="http://home.netscape.com/"
```

Example 2. The following statement sets the URL of a frame named `frame2` to the Sun home page:

```
parent.frame2.location.href="http://www.sun.com/"
```

See also the examples for `Anchor`.

See also `History`, `document.URL`

hash

A string beginning with a hash mark (#) that specifies an anchor name in the URL.

Property of	`Location`
Implemented in	JavaScript 1.0

Security **JavaScript 1.1.** This property is tainted by default. For information on data tainting, see the *Client-Side JavaScript Guide*.

Description The `hash` property specifies a portion of the URL. This property applies to HTTP URLs only.

You can set the `hash` property at any time, although it is safer to set the `href` property to change a location. If the hash that you specify cannot be found in the current location, you get an error.

Setting the `hash` property navigates to the named anchor without reloading the document. This differs from the way a document is loaded when other `location` properties are set (see "How documents are loaded when location is set" on page 252).

See RFC 1738 (`http://www.cis.ohio-state.edu/htbin/rfc/rfc1738.html`) for complete information about the hash.

Examples In the following example, the `window.open` statement creates a window called `newWindow` and loads the specified URL into it. The `document.write` statements display properties of `newWindow.location` in a window called `msgWindow`.

```
newWindow=window.open
   ("http://home.netscape.com/comprod/products/navigator/
   version_2.0/script/script_info/objects.html#checkbox_object")

msgWindow.document.write("newWindow.location.href = " +
   newWindow.location.href + "<P>")
msgWindow.document.write("newWindow.location.hash = " +
   newWindow.location.hash + "<P>")
msgWindow.document.close()
```

The previous example displays output such as the following:

```
newWindow.location.href =
   http://home.netscape.com/comprod/products/navigator/
   version_2.0/script/script_info/objects.html#checkbox_object
newWindow.location.hash = #checkbox_object
```

See also `Location.host`, `Location.hostname`, `Location.href`, `Location.pathname`, `Location.port`, `Location.protocol`, `Location.search`

host

A string specifying the server name, subdomain, and domain name.

Property of	`Location`
Implemented in	JavaScript 1.0

Security **JavaScript 1.1.** This property is tainted by default. For information on data tainting, see the *Client-Side JavaScript Guide*.

Description The `host` property specifies a portion of a URL. The `host` property is a substring of the `hostname` property. The `hostname` property is the concatenation of the `host` and `port` properties, separated by a colon. When the `port` property is null, the `host` property is the same as the `hostname` property.

You can set the `host` property at any time, although it is safer to set the `href` property to change a location. If the host that you specify cannot be found in the current location, you get an error.

See Section 3.1 of RFC 1738 (`http://www.cis.ohio-state.edu/htbin/rfc/rfc1738.html`) for complete information about the hostname and port.

Examples In the following example, the `window.open` statement creates a window called `newWindow` and loads the specified URL into it. The `document.write` statements display properties of `newWindow.location` in a window called `msgWindow`.

```
newWindow=window.open
   ("http://home.netscape.com/comprod/products/navigator/
   version_2.0/script/script_info/objects.html#checkbox_object")
```

```
msgWindow.document.write("newWindow.location.href = " +
   newWindow.location.href + "<P>")
msgWindow.document.write("newWindow.location.host = " +
   newWindow.location.host + "<P>")
msgWindow.document.close()
```

The previous example displays output such as the following:

```
newWindow.location.href =
   http://home.netscape.com/comprod/products/navigator/
   version_2.0/script/script_info/objects.html#checkbox_object
newWindow.location.host = home.netscape.com
```

See also `Location.hash`, `Location.hostname`, `Location.href`, `Location.pathname`, `Location.port`, `Location.protocol`, `Location.search`

hostname

A string containing the full hostname of the server, including the server name, subdomain, domain, and port number.

Property of	`Location`
Implemented in	JavaScript 1.0

Security **JavaScript 1.1.** This property is tainted by default. For information on data tainting, see the *Client-Side JavaScript Guide*.

Description The `hostname` property specifies a portion of a URL. The `hostname` property is the concatenation of the `host` and `port` properties, separated by a colon. When the `port` property is 80 (the default), the `host` property is the same as the `hostname` property.

You can set the hostname property at any time, although it is safer to set the href property to change a location. If the hostname that you specify cannot be found in the current location, you get an error.

See Section 3.1 of RFC 1738 (http://www.cis.ohio-state.edu/htbin/rfc/rfc1738.html) for complete information about the hostname.

Examples In the following example, the window.open statement creates a window called newWindow and loads the specified URL into it. The document.write statements display properties of newWindow.location in a window called msgWindow.

```
newWindow=window.open
   ("http://home.netscape.com/comprod/products/navigator/
   version_2.0/script/script_info/objects.html#checkbox_object")

msgWindow.document.write("newWindow.location.href = " +
   newWindow.location.href + "<P>")
msgWindow.document.write("newWindow.location.hostName = " +
   newWindow.location.hostName + "<P>")
msgWindow.document.close()
```

The previous example displays output such as the following:

```
newWindow.location.href =
   http://home.netscape.com/comprod/products/navigator/
   version_2.0/script/script_info/objects.html#checkbox_object
newWindow.location.hostName = home.netscape.com
```

See also Location.hash, Location.host, Location.href, Location.pathname, Location.port, Location.protocol, Location.search

href

A string specifying the entire URL.

Property of	Location
Implemented in	JavaScript 1.0

Security **JavaScript 1.1.** This property is tainted by default. For information on data tainting, see the *Client-Side JavaScript Guide*.

Description The `href` property specifies the entire URL. Other `location` object properties are substrings of the `href` property. If you want to change the URL associated with a window, you should do so by changing the `href` property; this correctly updates all of the other properties.

You can set the `href` property at any time.

Omitting a property name from the `location` object is equivalent to specifying `location.href`. For example, the following two statements are equivalent and set the URL of the current window to the Netscape home page:

```
window.location.href="http://home.netscape.com/"
window.location="http://home.netscape.com/"
```

See RFC 1738 (`http://www.cis.ohio-state.edu/htbin/rfc/rfc1738.html`) for complete information about the URL.

Examples In the following example, the `window.open` statement creates a window called `newWindow` and loads the specified URL into it. The `document.write` statements display all the properties of `newWindow.location` in a window called `msgWindow`.

```
newWindow=window.open
   ("http://home.netscape.com/comprod/products/navigator/
   version_2.0/script/script_info/objects.html#checkbox_object")

msgWindow.document.write("newWindow.location.href = " +
   newWindow.location.href + "<P>")
msgWindow.document.write("newWindow.location.protocol = " +
   newWindow.location.protocol + "<P>")
msgWindow.document.write("newWindow.location.host = " +
   newWindow.location.host + "<P>")
msgWindow.document.write("newWindow.location.hostName = " +
   newWindow.location.hostName + "<P>")
msgWindow.document.write("newWindow.location.port = " +
   newWindow.location.port + "<P>")
msgWindow.document.write("newWindow.location.pathname = " +
   newWindow.location.pathname + "<P>")
msgWindow.document.write("newWindow.location.hash = " +
   newWindow.location.hash + "<P>")
msgWindow.document.write("newWindow.location.search = " +
   newWindow.location.search + "<P>")
msgWindow.document.close()
```

The previous example displays output such as the following:

```
newWindow.location.href =
   http://home.netscape.com/comprod/products/navigator/
   version_2.0/script/script_info/objects.html#checkbox_object
newWindow.location.protocol = http:
newWindow.location.host = home.netscape.com
newWindow.location.hostName = home.netscape.com
newWindow.location.port =
newWindow.location.pathname =
   /comprod/products/navigator/version_2.0/script/
   script_info/objects.html
newWindow.location.hash = #checkbox_object
newWindow.location.search =
```

See also `Location.hash`, `Location.host`, `Location.hostname`, `Location.pathname`, `Location.port`, `Location.protocol`, `Location.search`

pathname

A string specifying the URL-path portion of the URL.

Property of	`Location`
Implemented in	JavaScript 1.0

Security **JavaScript 1.1.** This property is tainted by default. For information on data tainting, see the *Client-Side JavaScript Guide*.

Description The `pathname` property specifies a portion of the URL. The pathname supplies the details of how the specified resource can be accessed.

You can set the `pathname` property at any time, although it is safer to set the `href` property to change a location. If the pathname that you specify cannot be found in the current location, you get an error.

See Section 3.1 of RFC 1738 (`http://www.cis.ohio-state.edu/htbin/rfc/rfc1738.html`) for complete information about the pathname.

Examples In the following example, the `window.open` statement creates a window called `newWindow` and loads the specified URL into it. The `document.write` statements display properties of `newWindow.location` in a window called `msgWindow`.

```
newWindow=window.open
   ("http://home.netscape.com/comprod/products/navigator/
   version_2.0/script/script_info/objects.html#checkbox_object")

msgWindow.document.write("newWindow.location.href = " +
   newWindow.location.href + "<P>")
msgWindow.document.write("newWindow.location.pathname = " +
   newWindow.location.pathname + "<P>")
msgWindow.document.close()
```

The previous example displays output such as the following:

```
newWindow.location.href =
   http://home.netscape.com/comprod/products/navigator/
   version_2.0/script/script_info/objects.html#checkbox_object
newWindow.location.pathname =
   /comprod/products/navigator/version_2.0/script/
   script_info/objects.html
```

See also `Location.hash`, `Location.host`, `Location.hostname`, `Location.href`, `Location.port`, `Location.protocol`, `Location.search`

port

A string specifying the communications port that the server uses.

Property of	`Location`
Implemented in	JavaScript 1.0

Security **JavaScript 1.1.** This property is tainted by default. For information on data tainting, see the *Client-Side JavaScript Guide*.

Description The `port` property specifies a portion of the URL. The `port` property is a substring of the `hostname` property. The `hostname` property is the concatenation of the `host` and `port` properties, separated by a colon.

You can set the `port` property at any time, although it is safer to set the `href` property to change a location. If the port that you specify cannot be found in the current location, you get an error. If the `port` property is not specified, it defaults to 80.

See Section 3.1 of RFC 1738 (`http://www.cis.ohio-state.edu/htbin/rfc/rfc1738.html`) for complete information about the port.

Examples In the following example, the `window.open` statement creates a window called `newWindow` and loads the specified URL into it. The `document.write` statements display properties of `newWindow.location` in a window called `msgWindow`.

```
newWindow=window.open
   ("http://home.netscape.com/comprod/products/navigator/
   version_2.0/script/script_info/objects.html#checkbox_object")

msgWindow.document.write("newWindow.location.href = " +
   newWindow.location.href + "<P>")
msgWindow.document.write("newWindow.location.port = " +
   newWindow.location.port + "<P>")
msgWindow.document.close()
```

The previous example displays output such as the following:

```
newWindow.location.href =
   http://home.netscape.com/comprod/products/navigator/
   version_2.0/script/script_info/objects.html#checkbox_object
newWindow.location.port =
```

See also `Location.hash`, `Location.host`, `Location.hostname`, `Location.href`, `Location.pathname`, `Location.protocol`, `Location.search`

protocol

A string specifying the beginning of the URL, up to and including the first colon.

Property of	`Location`
Implemented in	JavaScript 1.0

Security **JavaScript 1.1.** This property is tainted by default. For information on data tainting, see the *Client-Side JavaScript Guide*.

Description The `protocol` property specifies a portion of the URL. The protocol indicates the access method of the URL. For example, the value `"http:"` specifies HyperText Transfer Protocol, and the value `"javascript:"` specifies JavaScript code.

You can set the `protocol` property at any time, although it is safer to set the `href` property to change a location. If the protocol that you specify cannot be found in the current location, you get an error.

The `protocol` property represents the scheme name of the URL. See Section 2.1 of RFC 1738 (`http://www.cis.ohio-state.edu/htbin/rfc/rfc1738.html`) for complete information about the protocol.

Examples In the following example, the `window.open` statement creates a window called `newWindow` and loads the specified URL into it. The `document.write` statements display properties of `newWindow.location` in a window called `msgWindow`.

```
newWindow=window.open
   ("http://home.netscape.com/comprod/products/navigator/
   version_2.0/script/script_info/objects.html#checkbox_object")

msgWindow.document.write("newWindow.location.href = " +
   newWindow.location.href + "<P>")
msgWindow.document.write("newWindow.location.protocol = " +
   newWindow.location.protocol + "<P>")
msgWindow.document.close()
```

The previous example displays output such as the following:

```
newWindow.location.href =
   http://home.netscape.com/comprod/products/navigator/
   version_2.0/script/script_info/objects.html#checkbox_object
newWindow.location.protocol = http:
```

See also `Location.hash`, `Location.host`, `Location.hostname`, `Location.href`, `Location.pathname`, `Location.port`, `Location.search`

reload

Forces a reload of the window's current document (the document specified by the `Location.href` property).

Method of	`Location`
Implemented in	JavaScript 1.1

Syntax `reload([`*`forceGet`*`])`

Parameters

`forceGet`	If you supply `true`, forces an unconditional HTTP GET of the document from the server. This should not be used unless you have reason to believe that disk and memory caches are off or broken, or the server has a new version of the document (for example, if it is generated by a CGI on each request).

Description This method uses the same policy that the browser's Reload button uses. The user interface for setting the default value of this policy varies for different browser versions.

By default, the `reload` method does not force a transaction with the server. However, if the user has set the preference to check every time, the method does a "conditional GET" request using an If-modified-since HTTP header, to ask the server to return the document only if its last-modified time is newer than the time the client keeps in its cache. In other words, `reload` reloads from the cache, unless the user has specified to check every time *and* the document has changed on the server since it was last loaded and saved in the cache.

Examples The following example displays an image and three radio buttons. The user can click the radio buttons to choose which image is displayed. Clicking another button lets the user reload the document.

```
<SCRIPT>
function displayImage(theImage) {
   document.images[0].src=theImage
}
</SCRIPT>
```

```
<FORM NAME="imageForm">
<B>Choose an image:</B>
<BR><INPUT TYPE="radio" NAME="imageChoice" VALUE="image1" CHECKED
   onClick="displayImage('seaotter.gif')">Sea otter
<BR><INPUT TYPE="radio" NAME="imageChoice" VALUE="image2"
   onClick="displayImage('orca.gif')">Killer whale
<BR><INPUT TYPE="radio" NAME="imageChoice" VALUE="image3"
   onClick="displayImage('humpback.gif')">Humpback whale

<BR>
<IMG NAME="marineMammal" SRC="seaotter.gif" ALIGN="left" VSPACE="10">

<P><INPUT TYPE="button" VALUE="Click here to reload"
   onClick="window.location.reload()">
</FORM>
```

See also `Location.replace`

replace

Loads the specified URL over the current history entry.

Method of	`Location`
Implemented in	JavaScript 1.1

Syntax `replace(`*`URL`*`)`

Parameters

`URL`	A string specifying the URL to load.

Description The `replace` method loads the specified URL over the current history entry. After calling the `replace` method, the user cannot navigate to the previous URL by using browser's Back button.

If your program will be run with JavaScript 1.0, you could put the following line in a `SCRIPT` tag early in your program. This emulates `replace`, which was introduced in JavaScript 1.1:

```
if (location.replace == null)
   location.replace = location.assign
```

The `replace` method does not create a new entry in the history list. To create an entry in the history list while loading a URL, use the `History.go` method.

Examples The following example lets the user choose among several catalogs to display. The example displays two sets of radio buttons which let the user choose a season and a category, for example the Spring/Summer Clothing catalog or the Fall/Winter Home & Garden catalog. When the user clicks the Go button, the `displayCatalog` function executes the `replace` method, replacing the current URL with the URL appropriate for the catalog the user has chosen. After invoking `displayCatalog`, the user cannot navigate to the previous URL (the list of catalogs) by using browser's Back button.

```
<SCRIPT>
function displayCatalog() {
   var seaName=""
   var catName=""

   for (var i=0; i < document.catalogForm.season.length; i++) {
      if (document.catalogForm.season[i].checked) {
         seaName=document.catalogForm.season[i].value
         i=document.catalogForm.season.length
      }
   }

   for (var i in document.catalogForm.category) {
      if (document.catalogForm.category[i].checked) {
         catName=document.catalogForm.category[i].value
         i=document.catalogForm.category.length
      }
   }
   fileName=seaName + catName + ".html"
   location.replace(fileName)
}
</SCRIPT>

<FORM NAME="catalogForm">
<B>Which catalog do you want to see?</B>

<P><B>Season</B>
<BR><INPUT TYPE="radio" NAME="season" VALUE="q1" CHECKED>Spring/Summer
<BR><INPUT TYPE="radio" NAME="season" VALUE="q3">Fall/Winter

<P><B>Category</B>
<BR><INPUT TYPE="radio" NAME="category" VALUE="clo" CHECKED>Clothing
<BR><INPUT TYPE="radio" NAME="category" VALUE="lin">Linens
<BR><INPUT TYPE="radio" NAME="category" VALUE="hom">Home & Garden

<P><INPUT TYPE="button" VALUE="Go" onClick="displayCatalog()">
</FORM>
```

See also `History`, `window.open`, `History.go`, `Location.reload`

search

A string beginning with a question mark that specifies any query information in the URL.

Property of	`Location`
Implemented in	JavaScript 1.0

Security **JavaScript 1.1.** This property is tainted by default. For information on data tainting, see the *Client-Side JavaScript Guide*.

Description The `search` property specifies a portion of the URL. This property applies to HTTP URLs only.

The `search` property contains variable and value pairs; each pair is separated by an ampersand. For example, two pairs in a search string could look as follows:

```
?x=7&y=5
```

You can set the `search` property at any time, although it is safer to set the `href` property to change a location. If the search that you specify cannot be found in the current location, you get an error.

See Section 3.3 of RFC 1738 (`http://www.cis.ohio-state.edu/htbin/rfc/rfc1738.html`) for complete information about the search.

Examples In the following example, the `window.open` statement creates a window called `newWindow` and loads the specified URL into it. The `document.write` statements display properties of `newWindow.location` in a window called `msgWindow`.

```
newWindow=window.open
   ("http://guide-p.infoseek.com/WW/NS/Titles?qt=RFC+1738+&col=WW")

msgWindow.document.write("newWindow.location.href = " +
   newWindow.location.href + "<P>")
msgWindow.document.close()
msgWindow.document.write("newWindow.location.search = " +
   newWindow.location.search + "<P>")
msgWindow.document.close()
```

The previous example displays the following output:

```
newWindow.location.href =
   http://guide-p.infoseek.com/WW/NS/Titles?qt=RFC+1738+&col=WW
newWindow.location.search = ?qt=RFC+1738+&col=WW
```

See also `Location.hash`, `Location.host`, `Location.hostname`, `Location.href`, `Location.pathname`, `Location.port`, `Location.protocol`

Math

A built-in object that has properties and methods for mathematical constants and functions. For example, the `Math` object's `PI` property has the value of pi.

Core object

Implemented in	JavaScript 1.0, NES 2.0
ECMA version	ECMA-262

Created by The `Math` object is a top-level, predefined JavaScript object. You can automatically access it without using a constructor or calling a method.

Description All properties and methods of `Math` are static. You refer to the constant PI as `Math.PI` and you call the sine function as `Math.sin(x)`, where `x` is the method's argument. Constants are defined with the full precision of real numbers in JavaScript.

It is often convenient to use the `with` statement when a section of code uses several `Math` constants and methods, so you don't have to type "Math" repeatedly. For example,

```
with (Math) {
   a = PI * r*r
   y = r*sin(theta)
   x = r*cos(theta)
}
```

Property Summary

Property	Description
`E`	Euler's constant and the base of natural logarithms, approximately 2.718.
`LN10`	Natural logarithm of 10, approximately 2.302.
`LN2`	Natural logarithm of 2, approximately 0.693.
`LOG10E`	Base 10 logarithm of E (approximately 0.434).
`LOG2E`	Base 2 logarithm of E (approximately 1.442).
`PI`	Ratio of the circumference of a circle to its diameter, approximately 3.14159.
`SQRT1_2`	Square root of 1/2; equivalently, 1 over the square root of 2, approximately 0.707.
`SQRT2`	Square root of 2, approximately 1.414.

Method Summary

Method	Description
`abs`	Returns the absolute value of a number.
`acos`	Returns the arccosine (in radians) of a number.
`asin`	Returns the arcsine (in radians) of a number.
`atan`	Returns the arctangent (in radians) of a number.
`atan2`	Returns the arctangent of the quotient of its arguments.
`ceil`	Returns the smallest integer greater than or equal to a number.
`cos`	Returns the cosine of a number.
`exp`	Returns E^{number}, where `number` is the argument, and E is Euler's constant, the base of the natural logarithms.
`floor`	Returns the largest integer less than or equal to a number.
`log`	Returns the natural logarithm (base E) of a number.
`max`	Returns the greater of two numbers.
`min`	Returns the lesser of two numbers.
`pow`	Returns `base` to the `exponent` power, that is, $\mathtt{base}^{\mathtt{exponent}}$.
`random`	Returns a pseudo-random number between 0 and 1.
`round`	Returns the value of a number rounded to the nearest integer.
`sin`	Returns the sine of a number.
`sqrt`	Returns the square root of a number.
`tan`	Returns the tangent of a number.

In addition, this object inherits the `watch` and `unwatch` methods from `Object`.

abs

Returns the absolute value of a number.

Method of	`Math`
Static	
Implemented in	JavaScript 1.0, NES 2.0
ECMA version	ECMA-262

Syntax `abs(`*`x`*`)`

Parameters

`x`	A number

Examples The following function returns the absolute value of the variable `x`:

```
function getAbs(x) {
   return Math.abs(x)
}
```

Description Because `abs` is a static method of `Math`, you always use it as `Math.abs()`, rather than as a method of a `Math` object you created.

acos

Returns the arccosine (in radians) of a number.

Method of	`Math`
Static	
Implemented in	JavaScript 1.0, NES 2.0
ECMA version	ECMA-262

Syntax `acos(`*`x`*`)`

Parameters

`x`	A number

Description The `acos` method returns a numeric value between 0 and pi radians. If the value of `number` is outside this range, it returns `NaN`.

Because `acos` is a static method of `Math`, you always use it as `Math.acos()`, rather than as a method of a `Math` object you created.

Examples The following function returns the arccosine of the variable `x`:

```
function getAcos(x) {
   return Math.acos(x)
}
```

If you pass -1 to `getAcos`, it returns 3.141592653589793; if you pass 2, it returns `NaN` because 2 is out of range.

See also `Math.asin`, `Math.atan`, `Math.atan2`, `Math.cos`, `Math.sin`, `Math.tan`

asin

Returns the arcsine (in radians) of a number.

Method of	`Math`
Static	
Implemented in	JavaScript 1.0, NES 2.0
ECMA version	ECMA-262

Syntax `asin(`*`x`*`)`

Parameters

`x`	A number

Description The `asin` method returns a numeric value between -pi/2 and pi/2 radians. If the value of `number` is outside this range, it returns `NaN`.

Because `asin` is a static method of `Math`, you always use it as `Math.asin()`, rather than as a method of a `Math` object you created.

Examples The following function returns the arcsine of the variable `x`:

```
function getAsin(x) {
   return Math.asin(x)
}
```

If you pass `getAsin` the value 1, it returns 1.570796326794897 (pi/2); if you pass it the value 2, it returns `NaN` because 2 is out of range.

See also `Math.acos`, `Math.atan`, `Math.atan2`, `Math.cos`, `Math.sin`, `Math.tan`

atan

Returns the arctangent (in radians) of a number.

Method of	`Math`
Static	
Implemented in	JavaScript 1.0, NES 2.0
ECMA version	ECMA-262

Syntax `atan(`*`x`*`)`

Parameters

`x`	A number

Description The `atan` method returns a numeric value between -pi/2 and pi/2 radians.

Because `atan` is a static method of `Math`, you always use it as `Math.atan()`, rather than as a method of a `Math` object you created.

Examples The following function returns the arctangent of the variable `x`:

```
function getAtan(x) {
   return Math.atan(x)
}
```

If you pass `getAtan` the value 1, it returns 0.7853981633974483; if you pass it the value .5, it returns 0.4636476090008061.

See also `Math.acos`, `Math.asin`, `Math.atan2`, `Math.cos`, `Math.sin`, `Math.tan`

atan2

Returns the arctangent of the quotient of its arguments.

Method of	`Math`
Static	
Implemented in	JavaScript 1.0, NES 2.0
ECMA version	ECMA-262

Syntax `atan2(`*`y`*`, `*`x`*`)`

Parameters

`y, x`	Number

Description The `atan2` method returns a numeric value between -pi and pi representing the angle theta of an (`x,y`) point. This is the counterclockwise angle, measured in radians, between the positive X axis, and the point (`x,y`). Note that the arguments to this function pass the y-coordinate first and the x-coordinate second.

`atan2` is passed separate `x` and `y` arguments, and `atan` is passed the ratio of those two arguments.

Because `atan2` is a static method of `Math`, you always use it as `Math.atan2()`, rather than as a method of a `Math` object you created.

Examples The following function returns the angle of the polar coordinate:

```
function getAtan2(x,y) {
   return Math.atan2(x,y)
}
```

If you pass `getAtan2` the values (90,15), it returns 1.4056476493802699; if you pass it the values (15,90), it returns 0.16514867741462683.

See also `Math.acos`, `Math.asin`, `Math.atan`, `Math.cos`, `Math.sin`, `Math.tan`

ceil

Returns the smallest integer greater than or equal to a number.

Method of	`Math`
Static	
Implemented in	JavaScript 1.0, NES 2.0
ECMA version	ECMA-262

Syntax `ceil(x)`

Parameters

`x`	A number

Description Because `ceil` is a static method of `Math`, you always use it as `Math.ceil()`, rather than as a method of a `Math` object you created.

Examples The following function returns the ceil value of the variable `x`:

```
function getCeil(x) {
   return Math.ceil(x)
}
```

If you pass 45.95 to `getCeil`, it returns 46; if you pass -45.95, it returns -45.

See also `Math.floor`

cos

Returns the cosine of a number.

Method of	`Math`
Static	
Implemented in	JavaScript 1.0, NES 2.0
ECMA version	ECMA-262

Syntax `cos(x)`

Parameters

`x`	A number

Description The `cos` method returns a numeric value between -1 and 1, which represents the cosine of the angle.

Because `cos` is a static method of `Math`, you always use it as `Math.cos()`, rather than as a method of a `Math` object you created.

Examples The following function returns the cosine of the variable `x`:

```
function getCos(x) {
   return Math.cos(x)
}
```

If `x` equals 2*`Math.PI`, `getCos` returns 1; if `x` equals `Math.PI`, the `getCos` method returns -1.

See also `Math.acos`, `Math.asin`, `Math.atan`, `Math.atan2`, `Math.sin`, `Math.tan`

E

Euler's constant and the base of natural logarithms, approximately 2.718.

Property of	`Math`
Static, Read-only	
Implemented in	JavaScript 1.0, NES 2.0
ECMA version	ECMA-262

Description Because `E` is a static property of `Math`, you always use it as `Math.E`, rather than as a property of a `Math` object you created.

Examples The following function returns Euler's constant:

```
function getEuler() {
   return Math.E
}
```

exp

Returns E^x, where `x` is the argument, and `E` is Euler's constant, the base of the natural logarithms.

Method of	`Math`
Static	
Implemented in	JavaScript 1.0, NES 2.0
ECMA version	ECMA-262

Syntax `exp(x)`

Parameters

`x`	A number

Description Because `exp` is a static method of `Math`, you always use it as `Math.exp()`, rather than as a method of a `Math` object you created.

Examples The following function returns the exponential value of the variable `x`:

```
function getExp(x) {
   return Math.exp(x)
}
```

If you pass `getExp` the value 1, it returns 2.718281828459045.

See also `Math.E`, `Math.log`, `Math.pow`

floor

Returns the largest integer less than or equal to a number.

Method of	`Math`
Static	
Implemented in	JavaScript 1.0, NES 2.0
ECMA version	ECMA-262

Syntax `floor(`*`x`*`)`

Parameters

`x`	A number

Description Because `floor` is a static method of `Math`, you always use it as `Math.floor()`, rather than as a method of a `Math` object you created.

Examples The following function returns the floor value of the variable `x`:

```
function getFloor(x) {
   return Math.floor(x)
}
```

If you pass 45.95 to `getFloor`, it returns 45; if you pass -45.95, it returns -46.

See also `Math.ceil`

LN10

The natural logarithm of 10, approximately 2.302.

Property of	`Math`
Static, Read-only	
Implemented in	JavaScript 1.0, NES 2.0
ECMA version	ECMA-262

Examples The following function returns the natural log of 10:

```
function getNatLog10() {
   return Math.LN10
}
```

Description Because `LN10` is a static property of `Math`, you always use it as `Math.LN10`, rather than as a property of a `Math` object you created.

LN2

The natural logarithm of 2, approximately 0.693.

Property of	`Math`
Static, Read-only	
Implemented in	JavaScript 1.0, NES 2.0
ECMA version	ECMA-262

Examples The following function returns the natural log of 2:

```
function getNatLog2() {
   return Math.LN2
}
```

Description Because `LN2` is a static property of `Math`, you always use it as `Math.LN2`, rather than as a property of a `Math` object you created.

log

Returns the natural logarithm (base `E`) of a number.

Method of	`Math`
Static	
Implemented in	JavaScript 1.0, NES 2.0
ECMA version	ECMA-262

Syntax `log(`*`x`*`)`

Parameters

`x`	A number

Description If the value of `number` is negative, the return value is always `NaN`.

Because `log` is a static method of `Math`, you always use it as `Math.log()`, rather than as a method of a `Math` object you created.

Examples The following function returns the natural log of the variable `x`:

```
function getLog(x) {
   return Math.log(x)
}
```

If you pass `getLog` the value 10, it returns 2.302585092994046; if you pass it the value 0, it returns `-Infinity`; if you pass it the value -1, it returns `NaN` because -1 is out of range.

See also `Math.exp`, `Math.pow`

LOG10E

The base 10 logarithm of E (approximately 0.434).

Property of	`Math`
Static, Read-only	
Implemented in	JavaScript 1.0, NES 2.0
ECMA version	ECMA-262

Examples The following function returns the base 10 logarithm of `E`:

```
function getLog10e() {
   return Math.LOG10E
}
```

Description Because `LOG10E` is a static property of `Math`, you always use it as `Math.LOG10E`, rather than as a property of a `Math` object you created.

LOG2E

The base 2 logarithm of E (approximately 1.442).

Property of	`Math`
Static, Read-only	
Implemented in	JavaScript 1.0, NES 2.0
ECMA version	ECMA-262

Examples The following function returns the base 2 logarithm of E:

```
function getLog2e() {
   return Math.LOG2E
}
```

Description Because `LOG2E` is a static property of `Math`, you always use it as `Math.LOG2E`, rather than as a property of a `Math` object you created.

max

Returns the larger of two numbers.

Method of	`Math`
Static	
Implemented in	JavaScript 1.0, NES 2.0
ECMA version	ECMA-262

Syntax `max(`*`x`*`,`*`y`*`)`

Parameters

`x, y`	Numbers.

Description Because `max` is a static method of `Math`, you always use it as `Math.max()`, rather than as a method of a `Math` object you created.

Examples The following function evaluates the variables `x` and `y`:

```
function getMax(x,y) {
   return Math.max(x,y)
}
```

If you pass `getMax` the values 10 and 20, it returns 20; if you pass it the values -10 and -20, it returns -10.

See also `Math.min`

min

Returns the smaller of two numbers.

Method of	`Math`
Static	
Implemented in	JavaScript 1.0, NES 2.0
ECMA version	ECMA-262

Syntax `min(`*`x`*`,`*`y`*`)`

Parameters

`x, y`	Numbers.

Description Because `min` is a static method of `Math`, you always use it as `Math.min()`, rather than as a method of a `Math` object you created.

Examples The following function evaluates the variables `x` and `y`:

```
function getMin(x,y) {
   return Math.min(x,y)
}
```

If you pass `getMin` the values 10 and 20, it returns 10; if you pass it the values -10 and -20, it returns -20.

See also `Math.max`

PI

The ratio of the circumference of a circle to its diameter, approximately 3.14159.

Property of	`Math`
Static, Read-only	
Implemented in	JavaScript 1.0, NES 2.0
ECMA version	ECMA-262

Examples The following function returns the value of pi:

```
function getPi() {
   return Math.PI
}
```

Description Because `PI` is a static property of `Math`, you always use it as `Math.PI`, rather than as a property of a `Math` object you created.

pow

Returns `base` to the `exponent` power, that is, $\texttt{base}^{\text{exponent}}$.

Method of	`Math`
Static	
Implemented in	JavaScript 1.0, NES 2.0
ECMA version	ECMA-262

Syntax `pow(`*`x`*`,`*`y`*`)`

Parameters

`base`	The base number
`exponent`	The exponent to which to raise `base`

Description Because `pow` is a static method of `Math`, you always use it as `Math.pow()`, rather than as a method of a `Math` object you created.

Examples

```
function raisePower(x,y) {
   return Math.pow(x,y)
}
```

If `x` is 7 and `y` is 2, `raisePower` returns 49 (7 to the power of 2).

See also `Math.exp`, `Math.log`

random

Returns a pseudo-random number between 0 and 1. The random number generator is seeded from the current time, as in Java.

Method of	`Math`
Static	
Implemented in	JavaScript 1.0, NES 2.0: Unix only
	JavaScript 1.1, NES 2.0: all platforms
ECMA version	ECMA-262

Syntax `random()`

Parameters None.

Description Because `random` is a static method of `Math`, you always use it as `Math.random()`, rather than as a method of a `Math` object you created.

Examples

```
//Returns a random number between 0 and 1
function getRandom() {
   return Math.random()
}
```

round

Returns the value of a number rounded to the nearest integer.

Method of	`Math`
Static	
Implemented in	JavaScript 1.0, NES 2.0
ECMA version	ECMA-262

Syntax `round(`*`x`*`)`

Parameters

`x`	A number

Description If the fractional portion of `number` is .5 or greater, the argument is rounded to the next higher integer. If the fractional portion of `number` is less than .5, the argument is rounded to the next lower integer.

Because `round` is a static method of `Math`, you always use it as `Math.round()`, rather than as a method of a `Math` object you created.

Examples

```
//Returns the value 20
x=Math.round(20.49)

//Returns the value 21
x=Math.round(20.5)

//Returns the value -20
x=Math.round(-20.5)

//Returns the value -21
x=Math.round(-20.51)
```

sin

Returns the sine of a number.

Method of	`Math`
Static	
Implemented in	JavaScript 1.0, NES 2.0
ECMA version	ECMA-262

Syntax `sin(`*`x`*`)`

Parameters

`x`	A number

Description The `sin` method returns a numeric value between -1 and 1, which represents the sine of the argument.

Because `sin` is a static method of `Math`, you always use it as `Math.sin()`, rather than as a method of a `Math` object you created.

Examples The following function returns the sine of the variable `x`:

```
function getSine(x) {
   return Math.sin(x)
}
```

If you pass `getSine` the value `Math.PI/2`, it returns 1.

See also `Math.acos`, `Math.asin`, `Math.atan`, `Math.atan2`, `Math.cos`, `Math.tan`

sqrt

Returns the square root of a number.

Method of	`Math`
Static	
Implemented in	JavaScript 1.0, NES 2.0
ECMA version	ECMA-262

Syntax `sqrt(`*`x`*`)`

Parameters

`x`	A number

Description If the value of `number` is negative, `sqrt` returns `NaN`.

Because `sqrt` is a static method of `Math`, you always use it as `Math.sqrt()`, rather than as a method of a `Math` object you created.

Examples The following function returns the square root of the variable `x`:

```
function getRoot(x) {
   return Math.sqrt(x)
}
```

If you pass `getRoot` the value 9, it returns 3; if you pass it the value 2, it returns 1.414213562373095.

SQRT1_2

The square root of 1/2; equivalently, 1 over the square root of 2, approximately 0.707.

Property of	`Math`
Static, Read-only	
Implemented in	JavaScript 1.0, NES 2.0
ECMA version	ECMA-262

Examples The following function returns 1 over the square root of 2:

```
function getRoot1_2() {
   return Math.SQRT1_2
}
```

Description Because `SQRT1_2` is a static property of `Math`, you always use it as `Math.SQRT1_2`, rather than as a property of a `Math` object you created.

SQRT2

The square root of 2, approximately 1.414.

Property of	`Math`
Static, Read-only	
Implemented in	JavaScript 1.0, NES 2.0
ECMA version	ECMA-262

Examples The following function returns the square root of 2:

```
function getRoot2() {
   return Math.SQRT2
}
```

Description Because `SQRT2` is a static property of `Math`, you always use it as `Math.SQRT2`, rather than as a property of a `Math` object you created.

tan

Returns the tangent of a number.

Method of	`Math`
Static	
Implemented in	JavaScript 1.0, NES 2.0
ECMA version	ECMA-262

Syntax `tan(`*`x`*`)`

Parameters

`x`	A number

Description The `tan` method returns a numeric value that represents the tangent of the angle.

Because `tan` is a static method of `Math`, you always use it as `Math.tan()`, rather than as a method of a `Math` object you created.

Examples The following function returns the tangent of the variable `x`:

```
function getTan(x) {
   return Math.tan(x)
}
```

See also `Math.acos`, `Math.asin`, `Math.atan`, `Math.atan2`, `Math.cos`, `Math.sin`

MimeType

A MIME type (Multipart Internet Mail Extension) supported by the client.
Client-side object

Implemented in JavaScript 1.1

Created by You do not create `MimeType` objects yourself. These objects are predefined JavaScript objects that you access through the `mimeTypes` array of the `navigator` or `Plugin` object:

```
navigator.mimeTypes[index]
```

where `index` is either an integer representing a MIME type supported by the client or a string containing the type of a `MimeType` object (from the `MimeType.type` property).

Description Each `MimeType` object is an element in a `mimeTypes` array. The `mimeTypes` array is a property of both `navigator` and `Plugin` objects. For example, the following table summarizes the values for displaying JPEG images:

Expression	Value
`navigator.mimeTypes["image/jpeg"].type`	`image/jpeg`
`navigator.mimeTypes["image/jpeg"].description`	`JPEG Image`
`navigator.mimeTypes["image/jpeg"].suffixes`	`jpeg, jpg, jpe, jfif, pjpeg, pjp`
`navigator.mimeTypes["image/jpeg"].enabledPlugins`	`null`

Property Summary

Property	Description
`description`	A description of the MIME type.
`enabledPlugin`	Reference to the `Plugin` object configured for the MIME type.
`suffixes`	A string listing possible filename extensions for the MIME type, for example `"mpeg, mpg, mpe, mpv, vbs, mpegv"`.
`type`	The name of the MIME type, for example `"video/mpeg"` or `"audio/x-wav"`.

Method Summary This object inherits the `watch` and `unwatch` methods from `Object`.

Examples The following code displays the `type`, `description`, `suffixes`, and `enabledPlugin` properties for each `MimeType` object on a client:

```
document.writeln("<TABLE BORDER=1><TR VALIGN=TOP>",
   "<TH ALIGN=left>i",
   "<TH ALIGN=left>type",
   "<TH ALIGN=left>description",
   "<TH ALIGN=left>suffixes",
   "<TH ALIGN=left>enabledPlugin.name</TR>")
for (i=0; i < navigator.mimeTypes.length; i++) {
   document.writeln("<TR VALIGN=TOP><TD>",i,
      "<TD>",navigator.mimeTypes[i].type,
      "<TD>",navigator.mimeTypes[i].description,
      "<TD>",navigator.mimeTypes[i].suffixes)
   if (navigator.mimeTypes[i].enabledPlugin==null) {
      document.writeln(
      "<TD>None",
      "</TR>")
   } else {
      document.writeln(
      "<TD>",navigator.mimeTypes[i].enabledPlugin.name,
      "</TR>")
   }
}
document.writeln("</TABLE>")
```

The preceding example displays output similar to the following:

i	type	description	suffixes	enabledPlugin.name
0	audio/aiff	AIFF	aif, aiff	LiveAudio
1	audio/wav	WAV	wav	LiveAudio
2	audio/x-midi	MIDI	mid, midi	LiveAudio
3	audio/midi	MIDI	mid, midi	LiveAudio
4	video/msvideo	Video for Windows	avi	NPAVI32 Dynamic Link Library
5	*	Netscape Default Plugin		Netscape Default Plugin
6	zz-application/zz-winassoc-TGZ		TGZ	None

See also `navigator`, `navigator.mimeTypes`, `Plugin`

description

A human-readable description of the data type described by the MIME type object.

Property of	`MimeType`
Read-only	
Implemented in	JavaScript 1.1

enabledPlugin

The `Plugin` object for the plug-in that is configured for the specified MIME type If the MIME type does not have a plug-in configured, `enabledPlugin` is null.

Property of	`MimeType`
Read-only	
Implemented in	JavaScript 1.1

Description Use the `enabledPlugin` property to determine which plug-in is configured for a specific MIME type. Each plug-in may support multiple MIME types, and each MIME type could potentially be supported by multiple plug-ins. However, only one plug-in can be configured for a MIME type. (On Macintosh and Unix, the user can configure the handler for each MIME type; on Windows, the handler is determined at browser start-up time.)

The `enabledPlugin` property is a reference to a `Plugin` object that represents the plug-in that is configured for the specified MIME type.

You might need to know which plug-in is configured for a MIME type, for example, to dynamically emit an `EMBED` tag on the page if the user has a plug-in configured for the MIME type.

Examples The following example determines whether the Shockwave plug-in is installed. If it is, a movie is displayed.

```
// Can we display Shockwave movies?
mimetype = navigator.mimeTypes["application/x-director"]
if (mimetype) {
   // Yes, so can we display with a plug-in?
   plugin = mimetype.enabledPlugin
   if (plugin)
      // Yes, so show the data in-line
      document.writeln("Here\'s a movie: <EMBED SRC=mymovie.dir HEIGHT=100 WIDTH=100>")
      else
      // No, so provide a link to the data
      document.writeln("<A HREF='mymovie.dir'>Click here</A> to see a movie.")
   } else {
   // No, so tell them so
   document.writeln("Sorry, can't show you this cool movie.")
}
```

suffixes

A string listing possible file suffixes (also known as filename extensions) for the MIME type.

Property of	`MimeType`
Read-only	
Implemented in	JavaScript 1.1

Description The `suffixes` property is a string consisting of each valid suffix (typically three letters long) separated by commas. For example, the suffixes for the `"audio/x-midi"` MIME type are `"mid, midi"`.

type

A string specifying the name of the MIME type. This string distinguishes the MIME type from all others; for example `"video/mpeg"` or `"audio/x-wav"`.

Property of	`MimeType`
Read-only	
Implemented in	JavaScript 1.1

Property of `MimeType`

navigator

Contains information about the version of Navigator in use.

Client-side object

Implemented in	JavaScript 1.0 JavaScript 1.1: added `mimeTypes` and `plugins` properties; added `javaEnabled` and `taintEnabled` methods. JavaScript 1.2: added `language` and `platform` properties; added `preference` and `savePreferences` methods.

Created by The JavaScript runtime engine on the client automatically creates the `navigator` object.

Description Use the `navigator` object to determine which version of the Navigator your users have, what MIME types the user's Navigator can handle, and what plug-ins the user has installed. All of the properties of the `navigator` object are read-only.

Property Summary

Property	Description
`appCodeName`	Specifies the code name of the browser.
`appName`	Specifies the name of the browser.
`appVersion`	Specifies version information for the Navigator.
`language`	Indicates the translation of the Navigator being used.
`mimeTypes`	An array of all MIME types supported by the client.
`platform`	Indicates the machine type for which the Navigator was compiled.
`plugins`	An array of all plug-ins currently installed on the client.
`userAgent`	Specifies the user-agent header.

Method Summary

Method	Description
`javaEnabled`	Tests whether Java is enabled.
`plugins.refresh`	Makes newly installed plug-ins available and optionally reloads open documents that contain plug-ins.
`preference`	Allows a signed script to get and set certain Navigator preferences.
`savePreferences`	Saves the Navigator preferences to the local file `prefs.js`.
`taintEnabled`	Specifies whether data tainting is enabled.

In addition, this object inherits the `watch` and `unwatch` methods from `Object`.

appCodeName

A string specifying the code name of the browser.

Property of	`navigator`
Read-only	
Implemented in	JavaScript 1.0

Examples The following example displays the value of the `appCodeName` property:

```
document.write("The value of navigator.appCodeName is " +
   navigator.appCodeName)
```

For Navigator 2.0 and later, this displays the following:

```
The value of navigator.appCodeName is Mozilla
```

appName

A string specifying the name of the browser.

Property of `navigator`

Read-only

Implemented in JavaScript 1.0

Examples The following example displays the value of the `appName` property:

```
document.write("The value of navigator.appName is " +
   navigator.appName)
```

For Navigator 2.0 and 3.0, this displays the following:

```
The value of navigator.appName is Netscape
```

appVersion

A string specifying version information for the Navigator.

Property of `navigator`

Read-only

Implemented in JavaScript 1.0

Description The `appVersion` property specifies version information in the following format:

`releaseNumber` (`platform`; `country`)

The values contained in this format are the following:

- `releaseNumber` is the version number of the Navigator. For example, `"2.0b4"` specifies Navigator 2.0, beta 4.
- `platform` is the platform upon which the Navigator is running. For example, `"Win16"` specifies a 16-bit version of Windows such as Windows 3.1.
- `country` is either `"I"` for the international release, or `"U"` for the domestic U.S. release. The domestic release has a stronger encryption feature than the international release.

Examples **Example 1.** The following example displays version information for the Navigator:

```
document.write("The value of navigator.appVersion is " +
   navigator.appVersion)
```

For Navigator 2.0 on Windows 95, this displays the following:

```
The value of navigator.appVersion is 2.0 (Win95, I)
```

For Navigator 3.0 on Windows NT, this displays the following:

```
The value of navigator.appVersion is 3.0 (WinNT, I)
```

Example 2. The following example populates a `Textarea` object with newline characters separating each line. Because the newline character varies from platform to platform, the example tests the `appVersion` property to determine whether the user is running Windows (`appVersion` contains `"Win"` for all versions of Windows). If the user is running Windows, the newline character is set to \r\n; otherwise, it's set to \n, which is the newline character for Unix and Macintosh.

Note This code is needed only for JavaScript 1.0. JavaScript versions 1.1 and later check for all newline characters before setting a string-valued property and translate them as needed for the user's platform.

```
<SCRIPT>
var newline=null
function populate(textareaObject){
   if (navigator.appVersion.lastIndexOf('Win') != -1)
      newline="\r\n"
      else newline="\n"
   textareaObject.value="line 1" + newline + "line 2" + newline
   + "line 3"
}
</SCRIPT>
<FORM NAME="form1">
<BR><TEXTAREA NAME="testLines" ROWS=8 COLS=55></TEXTAREA>
<P><INPUT TYPE="button" VALUE="Populate the Textarea object"
   onClick="populate(document.form1.testLines)">
</TEXTAREA>
</FORM>
```

javaEnabled

Tests whether Java is enabled.

Method of `navigator`

Static

Implemented in JavaScript 1.1

Syntax `javaEnabled()`

Parameters None.

Description `javaEnabled` returns true if Java is enabled; otherwise, false. The user can enable or disable Java by through user preferences.

Examples The following code executes `function1` if Java is enabled; otherwise, it executes `function2`.

```
if (navigator.javaEnabled()) {
   function1()
}
else function2()
```

See also `navigator.appCodeName`, `navigator.appName`, `navigator.userAgent`

language

Indicates the translation of the Navigator being used.

Property of `navigator`

Read-only

Implemented in JavaScript 1.2

Description The value for language is usually a 2-letter code, such as "en" and occasionally a five-character code to indicate a language subtype, such as "zh_CN".

Use this property to determine the language of the Navigator client software being used. For example you might want to display translated text for the user.

mimeTypes

An array of all MIME types supported by the client.

Property of	`navigator`
Read-only	
Implemented in	JavaScript 1.1

The `mimeTypes` array contains an entry for each MIME type supported by the client (either internally, via helper applications, or by plug-ins). For example, if a client supports three MIME types, these MIME types are reflected as `navigator.mimeTypes[0]`, `navigator.mimeTypes[1]`, and `navigator.mimeTypes[2]`.

Each element of the `mimeTypes` array is a `MimeType` object.

To obtain the number of supported mime types, use the `length` property: `navigator.mimeTypes.length`.

See also `MimeType`

platform

Indicates the machine type for which the Navigator was compiled.

Property of	`navigator`
Read-only	
Implemented in	JavaScript 1.2

Description Platform values are Win32, Win16, Mac68k, MacPPC and various Unix.

The machine type the Navigator was compiled for may differ from the actual machine type due to version differences, emulators, or other reasons.

If you use SmartUpdate to download software to a user's machine, you can use this property to ensure that the trigger downloads the appropriate JAR files. The triggering page checks the Navigator version before checking the platform property. For information on using SmartUpdate, see *Using JAR Installation Manager for SmartUpdate*.

plugins

An array of all plug-ins currently installed on the client.

Property of	`navigator`
Read-only	
Implemented in	JavaScript 1.1

You can refer to the `Plugin` objects installed on the client by using this array. Each element of the `plugins` array is a `Plugin` object. For example, if three plug-ins are installed on the client, these plug-ins are reflected as `navigator.plugins[0]`, `navigator.plugins[1]`, and `navigator.plugins[2]`.

To use the `plugins` array:

```
1. navigator.plugins[index]
2. navigator.plugins[index][mimeTypeIndex]
```

`index` is an integer representing a plug-in installed on the client or a string containing the name of a `Plugin` object (from the `name` property). The first form returns the `Plugin` object stored at the specified location in the plugins array. The second form returns the `MimeType` object at the specified index in that `Plugin` object.

To obtain the number of plug-ins installed on the client, use the `length` property: `navigator.plugins.length`.

plugins.refresh. The `plugins` array has its own method, `refresh`. This method makes newly installed plug-ins available, updates related arrays such as the `plugins` array, and optionally reloads open documents that contain plug-ins. You call this method with one of the following statements:

```
navigator.plugins.refresh(true)
navigator.plugins.refresh(false)
```

If you supply true, `refresh` refreshes the `plugins` array to make newly installed plug-ins available and reloads all open documents that contain embedded objects (`EMBED` tag). If you supply false, it refreshes the `plugins` array, but does not reload open documents.

When the user installs a plug-in, that plug-in is not available until `refresh` is called or the user closes and restarts Navigator.

Examples The following code refreshes arrays and reloads open documents containing embedded objects:

```
navigator.plugins.refresh(true)
```

See also the examples for the `Plugin` object.

preference

Allows a signed script to get and set certain Navigator preferences.

Method of	`navigator`
Static	
Implemented in	JavaScript 1.2

Syntax `preference(`*`prefName`*`[, `*`setValue`*`])`

Parameters

`prefName`	A string representing the name of the preference you want to get or set. Allowed preferences are listed below.
`setValue`	The value you want to assign to the preference. This can be a string, number, or Boolean.

Description This method returns the value of the preference. If you use the method to set the value, it returns the new value.

With permission, you can get and set the preferences shown in the following table.

Table 1.2 Preferences.

To do this...	Set this preference...	To this value...
Automatically load images	`general.always_load_images`	true or false
Enable Java	`security.enable_java`	true or false
Enable JavaScript	`javascript.enabled`	true or false
Enable style sheets	`browser.enable_style_sheets`	true or false
Enable SmartUpdate	`autoupdate.enabled`	true or false
Accept all cookies	`network.cookie.cookieBehavior`	0

Table 1.2 Preferences. (Continued)

To do this...	Set this preference...	To this value...
Accept only cookies that get sent back to the originating server	`network.cookie.cookieBehavior`	1
Disable cookies	`network.cookie.cookieBehavior`	2
Warn before accepting cookie	`network.cookie.warnAboutCookies`	true or false

Security Reading a preference with the `preference` method requires the `UniversalPreferencesRead` privilege. Setting a preference with this method requires the `UniversalPreferencesWrite` privilege. For information on security, see the *Client-Side JavaScript Guide*.

See also `savePreferences`

savePreferences

Saves the Navigator preferences to the local file `prefs.js`.

Method of	`navigator`
Static	
Implemented in	JavaScript 1.2

Security Saving user preferences requires the `UniversalPreferencesWrite` privilege. For information on security, see the *Client-Side JavaScript Guide*.

Syntax `SavePreferences()`

Description This method immediately saves the current Navigator preferences to the user's `prefs.js` settings file. Navigator also saves preferences automatically when it exits.

See also `preference`

taintEnabled

Specifies whether data tainting is enabled.

Method of	`navigator`
Static	
Implemented in	JavaScript 1.1
	JavaScript 1.2: removed

Syntax `navigator.taintEnabled()`

Description Tainting prevents other scripts from passing information that should be secure and private, such as directory structures or user session history. JavaScript cannot pass tainted values on to any server without the end user's permission.

Use `taintEnabled` to determine if data tainting is enabled. `taintEnabled` returns true if data tainting is enabled, false otherwise. The user enables or disables data tainting by using the environment variable `NS_ENABLE_TAINT`.

Examples The following code executes function1 if data tainting is enabled; otherwise it executes function2.

```
if (navigator.taintEnabled()) {
   function1()
   }
else function2()
```

See also `taint`, `untaint`

userAgent

A string representing the value of the user-agent header sent in the HTTP protocol from client to server.

Property of	`navigator`
Read-only	
Implemented in	JavaScript 1.0

Description Servers use the value sent in the user-agent header to identify the client.

Examples The following example displays `userAgent` information for the Navigator:

```
document.write("The value of navigator.userAgent is " +
   navigator.userAgent)
```

For Navigator 2.0, this displays the following:

```
The value of navigator.userAgent is Mozilla/2.0 (Win16; I)
```

netscape

A top-level object used to access any Java class in the package `netscape.*`.

Core object

Implemented in JavaScript 1.1, NES 2.0

Created by The `netscape` object is a top-level, predefined JavaScript object. You can automatically access it without using a constructor or calling a method.

Description The `netscape` object is a convenience synonym for the property `Packages.netscape`.

See also `Packages`, `Packages.netscape`

Number

Lets you work with numeric values. The `Number` object is an object wrapper for primitive numeric values.

Core object

Implemented in	JavaScript 1.1, NES 2.0
	JavaScript 1.2: modified behavior of `Number` constructor
	JavaScript 1.3: added `toSource` method
ECMA version	ECMA-262

Created by The `Number` constructor:

```
new Number(value)
```

Parameters

`value`	The numeric value of the object being created.

Description The primary uses for the `Number` object are:

- To access its constant properties, which represent the largest and smallest representable numbers, positive and negative infinity, and the Not-a-Number value.
- To create numeric objects that you can add properties to. Most likely, you will rarely need to create a `Number` object.

The properties of `Number` are properties of the class itself, not of individual `Number` objects.

JavaScript 1.2: `Number(x)` now produces `NaN` rather than an error if `x` is a string that does not contain a well-formed numeric literal. For example,

```
x=Number("three");
document.write(x + "<BR>");
```

prints `NaN`

You can convert any object to a number using the top-level `Number` function.

Property Summary

Property	Description
`constructor`	Specifies the function that creates an object's prototype.
`MAX_VALUE`	The largest representable number.
`MIN_VALUE`	The smallest representable number.
`NaN`	Special "not a number" value.
`NEGATIVE_INFINITY`	Special value representing negative infinity; returned on overflow.
`POSITIVE_INFINITY`	Special value representing infinity; returned on overflow.
`prototype`	Allows the addition of properties to a `Number` object.

Method Summary

Method	Description
`toSource`	Returns an object literal representing the specified Number object; you can use this value to create a new object. Overrides the `Object.toSource` method.
`toString`	Returns a string representing the specified object. Overrides the `Object.toString` method.
`valueOf`	Returns the primitive value of the specified object. Overrides the `Object.valueOf` method.

In addition, this object inherits the `watch` and `unwatch` methods from `Object`.

Examples

Example 1. The following example uses the `Number` object's properties to assign values to several numeric variables:

```
biggestNum = Number.MAX_VALUE
smallestNum = Number.MIN_VALUE
infiniteNum = Number.POSITIVE_INFINITY
negInfiniteNum = Number.NEGATIVE_INFINITY
notANum = Number.NaN
```

Example 2. The following example creates a `Number` object, `myNum`, then adds a `description` property to all `Number` objects. Then a value is assigned to the `myNum` object's `description` property.

```
myNum = new Number(65)
Number.prototype.description=null
myNum.description="wind speed"
```

constructor

Specifies the function that creates an object's prototype. Note that the value of this property is a reference to the function itself, not a string containing the function's name.

Property of	`Number`
Implemented in	JavaScript 1.1, NES 2.0
ECMA version	ECMA-262

Description See `Object.constructor`.

MAX_VALUE

The maximum numeric value representable in JavaScript.

Property of	`Number`
Static, Read-only	
Implemented in	JavaScript 1.1, NES 2.0
ECMA version	ECMA-262

Description The `MAX_VALUE` property has a value of approximately 1.79E+308. Values larger than `MAX_VALUE` are represented as `"Infinity"`.

Because `MAX_VALUE` is a static property of `Number`, you always use it as `Number.MAX_VALUE`, rather than as a property of a `Number` object you created.

Examples The following code multiplies two numeric values. If the result is less than or equal to `MAX_VALUE`, the `func1` function is called; otherwise, the `func2` function is called.

```
if (num1 * num2 <= Number.MAX_VALUE)
   func1()
else
   func2()
```

MIN_VALUE

The smallest positive numeric value representable in JavaScript.

Property of	`Number`
Static, Read-only	
Implemented in	JavaScript 1.1, NES 2.0
ECMA version	ECMA-262

Description The `MIN_VALUE` property is the number closest to 0, not the most negative number, that JavaScript can represent.

`MIN_VALUE` has a value of approximately 5e-324. Values smaller than `MIN_VALUE` ("underflow values") are converted to 0.

Because `MIN_VALUE` is a static property of `Number`, you always use it as `Number.MIN_VALUE`, rather than as a property of a `Number` object you created.

Examples The following code divides two numeric values. If the result is greater than or equal to `MIN_VALUE`, the `func1` function is called; otherwise, the `func2` function is called.

```
if (num1 / num2 >= Number.MIN_VALUE)
   func1()
else
   func2()
```

NaN

A special value representing Not-A-Number. This value is represented as the unquoted literal NaN.

Property of	`Number`
Read-only	
Implemented in	JavaScript 1.1, NES 2.0
ECMA version	ECMA-262

Description JavaScript prints the value `Number.NaN` as `NaN`.

`NaN` is always unequal to any other number, including NaN itself; you cannot check for the not-a-number value by comparing to `Number.NaN`. Use the `isNaN` function instead.

You might use the `NaN` property to indicate an error condition for a function that should return a valid number.

Examples In the following example, if `month` has a value greater than 12, it is assigned NaN, and a message is displayed indicating valid values.

```
var month = 13
if (month < 1 || month > 12) {
   month = Number.NaN
   alert("Month must be between 1 and 12.")
}
```

See also `NaN`, `isNaN`, `parseFloat`, `parseInt`

NEGATIVE_INFINITY

A special numeric value representing negative infinity. This value is represented as the unquoted literal `"-Infinity"`.

Property of	`Number`
Static, Read-only	
Implemented in	JavaScript 1.1, NES 2.0
ECMA version	ECMA-262

Description This value behaves slightly differently than mathematical infinity:

- Any positive value, including `POSITIVE_INFINITY`, multiplied by `NEGATIVE_INFINITY` is `NEGATIVE_INFINITY`.
- Any negative value, including `NEGATIVE_INFINITY`, multiplied by `NEGATIVE_INFINITY` is `POSITIVE_INFINITY`.
- Zero multiplied by `NEGATIVE_INFINITY` is `NaN`.
- `NaN` multiplied by `NEGATIVE_INFINITY` is `NaN`.
- `NEGATIVE_INFINITY`, divided by any negative value except `NEGATIVE_INFINITY`, is `POSITIVE_INFINITY`.
- `NEGATIVE_INFINITY`, divided by any positive value except `POSITIVE_INFINITY`, is `NEGATIVE_INFINITY`.

- NEGATIVE_INFINITY, divided by either NEGATIVE_INFINITY or POSITIVE_INFINITY, is NaN.
- Any number divided by NEGATIVE_INFINITY is Zero.

Because NEGATIVE_INFINITY is a static property of Number, you always use it as Number.NEGATIVE_INFINITY, rather than as a property of a Number object you created.

Examples In the following example, the variable smallNumber is assigned a value that is smaller than the minimum value. When the if statement executes, smallNumber has the value "-Infinity", so the func1 function is called.

```
var smallNumber = -Number.MAX_VALUE*10
if (smallNumber == Number.NEGATIVE_INFINITY)
   func1()
else
   func2()
```

See also Infinity, isFinite

POSITIVE_INFINITY

A special numeric value representing infinity. This value is represented as the unquoted literal "Infinity".

Property of	Number
Static, Read-only	
Implemented in	JavaScript 1.1, NES 2.0
ECMA version	ECMA-262

Description This value behaves slightly differently than mathematical infinity:

- Any positive value, including POSITIVE_INFINITY, multiplied by POSITIVE_INFINITY is POSITIVE_INFINITY.
- Any negative value, including NEGATIVE_INFINITY, multiplied by POSITIVE_INFINITY is NEGATIVE_INFINITY.
- Zero multiplied by POSITIVE_INFINITY is NaN.
- NaN multiplied by POSITIVE_INFINITY is NaN.
- POSITIVE_INFINITY, divided by any negative value except NEGATIVE_INFINITY, is NEGATIVE_INFINITY.
- POSITIVE_INFINITY, divided by any positive value except POSITIVE_INFINITY, is POSITIVE_INFINITY.

- `POSITIVE_INFINITY`, divided by either `NEGATIVE_INFINITY` or `POSITIVE_INFINITY`, is `NaN`.
- Any number divided by `POSITIVE_INFINITY` is Zero.

Because `POSITIVE_INFINITY` is a static property of `Number`, you always use it as `Number.POSITIVE_INFINITY`, rather than as a property of a `Number` object you created.

Examples In the following example, the variable `bigNumber` is assigned a value that is larger than the maximum value. When the `if` statement executes, `bigNumber` has the value `"Infinity"`, so the `func1` function is called.

```
var bigNumber = Number.MAX_VALUE * 10
if (bigNumber == Number.POSITIVE_INFINITY)
   func1()
else
   func2()
```

See also `Infinity`, `isFinite`

prototype

Represents the prototype for this class. You can use the prototype to add properties or methods to all instances of a class. For information on prototypes, see `Function.prototype`.

Property of	`Number`
Implemented in	JavaScript 1.1, NES 2.0
ECMA version	ECMA-262

toSource

Returns a string representing the source code of the object.

Method of	`Number`
Implemented in	JavaScript 1.3

Syntax `toSource()`

Parameters None

Description The `toSource` method returns the following values:

- For the built-in `Number` object, `toSource` returns the following string indicating that the source code is not available:

```
function Number() {
   [native code]
}
```

- For instances of `Number`, `toSource` returns a string representing the source code.

This method is usually called internally by JavaScript and not explicitly in code.

See also `Object.toSource`

toString

Returns a string representing the specified Number object.

Method of	`Number`
Implemented in	JavaScript 1.1
ECMA version	ECMA-262

Syntax

```
toString()
toString([radix])
```

Parameters

`radix`	An integer between 2 and 36 specifying the base to use for representing numeric values.

Description The `Number` object overrides the `toString` method of the `Object` object; it does not inherit `Object.toString`. For `Number` objects, the `toString` method returns a string representation of the object.

JavaScript calls the `toString` method automatically when a number is to be represented as a text value or when a number is referred to in a string concatenation.

For `Number` objects and values, the built-in `toString` method returns the string representing the value of the number.

You can use `toString` on numeric values, but not on numeric literals:

```
// The next two lines are valid
var howMany=10
alert("howMany.toString() is " + howMany.toString())

// The next line causes an error
alert("45.toString() is " + 45.toString())
```

valueOf

Returns the primitive value of a Number object.

Method of	`Number`
Implemented in	JavaScript 1.1
ECMA version	ECMA-262

Syntax `valueOf()`

Parameters None

Description The `valueOf` method of `Number` returns the primitive value of a Number object as a number data type.

This method is usually called internally by JavaScript and not explicitly in code.

Examples

```
x = new Number();
alert(x.valueOf())      //displays 0
```

See also `Object.valueOf`

Object

`Object` is the primitive JavaScript object type. All JavaScript objects are descended from `Object`. That is, all JavaScript objects have the methods defined for `Object`.

Core object

Implemented in	JavaScript 1.0: `toString` method JavaScript 1.1, NES 2.0: added `eval` and `valueOf` methods; `constructor` property JavaScript 1.2: deprecated `eval` method JavaScript 1.3: added `toSource` method
ECMA version	ECMA-262

Created by

The `Object` constructor:

```
new Object()
```

Parameters

None

Property Summary

Property	Description
`constructor`	Specifies the function that creates an object's prototype.
`prototype`	Allows the addition of properties to all objects.

Method Summary

Method	Description
`eval`	Deprecated. Evaluates a string of JavaScript code in the context of the specified object.
`toSource`	Returns an object literal representing the specified object; you can use this value to create a new object.
`toString`	Returns a string representing the specified object.
`unwatch`	Removes a watchpoint from a property of the object.
`valueOf`	Returns the primitive value of the specified object.
`watch`	Adds a watchpoint to a property of the object.

constructor

Specifies the function that creates an object's prototype. Note that the value of this property is a reference to the function itself, not a string containing the function's name.

Property of	`Object`
Implemented in	JavaScript 1.1, NES 2.0
ECMA version	ECMA-262

Description All objects inherit a `constructor` property from their `prototype`:

```
o = new Object  // or o = {} in JavaScript 1.2
o.constructor == Object
a = new Array   // or a = [] in JavaScript 1.2
a.constructor == Array
n = new Number(3)
n.constructor == Number
```

Even though you cannot construct most HTML objects, you can do comparisons. For example,

```
document.constructor == Document
document.form3.constructor == Form
```

Examples The following example creates a prototype, `Tree`, and an object of that type, `theTree`. The example then displays the `constructor` property for the object `theTree`.

```
function Tree(name) {
   this.name=name
}
theTree = new Tree("Redwood")
document.writeln("<B>theTree.constructor is</B> " +
   theTree.constructor + "<P>")
```

This example displays the following output:

```
theTree.constructor is function Tree(name) { this.name = name; }
```

eval

Deprecated. Evaluates a string of JavaScript code in the context of an object.

Method of	`Object`
Implemented in	JavaScript 1.1, NES 2.0 JavaScript 1.2, NES 3.0: deprecated as method of objects; retained as top-level function

Syntax `eval(`*`string`*`)`

Parameters

`string`	Any string representing a JavaScript expression, statement, or sequence of statements. The expression can include variables and properties of existing objects.

Description `eval` as a method of Object and every object derived from Object is deprecated. Use the top-level `eval` function.

Backward Compatibility **JavaScript 1.1.** `eval` is a method of Object and every object derived from Object.

See also `eval`

prototype

Represents the prototype for this class. You can use the prototype to add properties or methods to all instances of a class. For more information, see `Function.prototype`.

Property of	`Object`
Implemented in	JavaScript 1.1
ECMA version	ECMA-262

toSource

Returns a string representing the source code of the object.

Method of	`Object`
Implemented in	JavaScript 1.3

Syntax `toSource()`

Parameters None

Description The `toSource` method returns the following values:

- For the built-in `Object` object, `toSource` returns the following string indicating that the source code is not available:

  ```
  function Object() {
     [native code]
  }
  ```

- For instances of `Object`, `toSource` returns a string representing the source code.
- For custom objects, `toSource` returns the JavaScript source that defines the object as a string.

This method is usually called internally by JavaScript and not explicitly in code. You can call `toSource` while debugging to examine the contents of an object.

Examples The following code defines the `Dog` object type and creates `theDog,` an object of type `Dog`:

```
function Dog(name,breed,color,sex) {
   this.name=name
   this.breed=breed
   this.color=color
   this.sex=sex
}
theDog = new Dog("Gabby","Lab","chocolate","girl")
```

Calling the `toSource` method of `theDog` displays the JavaScript source that defines the object:

```
theDog.toSource()
//returns "{name:"Gabby", breed:"Lab", color:"chocolate", sex:"girl"}
```

See also `Object.toString`

toString

Returns a string representing the specified object.

Method of	`Object`
Implemented in	JavaScript 1.0
ECMA version	ECMA-262

Syntax `toString()`

Security JavaScript 1.1: This method is tainted by default for the following objects: `Button`, `Checkbox`, `FileUpload`, `Hidden`, `History`, `Link`, `Location`, `Password`, `Radio`, `Reset`, `Select`, `Submit`, `Text`, and `Textarea`. For information on data tainting, see the *Client-Side JavaScript Guide*.

Description Every object has a `toString` method that is automatically called when it is to be represented as a text value or when an object is referred to in a string concatenation. For example, the following examples require `theDog` to be represented as a string:

```
document.write(theDog)
document.write("The dog is " + theDog)
```

By default, the `toString` method is inherited by every object descended from `Object`. You can override this method for custom objects that you create. If you do not override `toString` in a custom object, `toString` returns `[object` *`type`*`]`, where *`type`* is the object type or the name of the constructor function that created the object.

For example:

```
var o = new Object()
o.toString // returns [object Object]
```

Built-in toString methods. Every built-in core JavaScript object overrides the `toString` method of `Object` to return an appropriate value. JavaScript calls this method whenever it needs to convert an object to a string.

Some built-in client-side and server-side JavaScript objects do not override the toString method of Object. For example, for an `Image` object named `sealife` defined as shown below, `sealife.toString()` returns `[object Image]`.

```
<IMG NAME="sealife" SRC="images\seaotter.gif" ALIGN="left" VSPACE="10">
```

Overriding the default toString method. You can create a function to be called in place of the default `toString` method. The `toString` method takes no arguments and should return a string. The `toString` method you create can be any value you want, but it will be most useful if it carries information about the object.

The following code defines the `Dog` object type and creates `theDog,` an object of type `Dog`:

```
function Dog(name,breed,color,sex) {
   this.name=name
   this.breed=breed
   this.color=color
   this.sex=sex
}
```

```
theDog = new Dog("Gabby","Lab","chocolate","girl")
```

If you call the `toString` method on this custom object, it returns the default value inherited from `Object`:

```
theDog.toString() //returns [object Object]
```

The following code creates `dogToString,` the function that will be used to override the default `toString` method. This function generates a string containing each property, of the form `"property = value;"`.

```
function dogToString() {
   var ret = "Dog " + this.name + " is [\n"
   for (var prop in this)
      ret += "  " + prop + " is " + this[prop] + ";\n"
   return ret + "]"
}
```

The following code assigns the user-defined function to the object's `toString` method:

```
Dog.prototype.toString = dogToString
```

With the preceding code in place, any time `theDog` is used in a string context, JavaScript automatically calls the `dogToString` function, which returns the following string:

```
Dog Gabby is [
  name is Gabby;
  breed is Lab;
  color is chocolate;
  sex is girl;
]
```

An object's `toString` method is usually invoked by JavaScript, but you can invoke it yourself as follows:

```
var dogString = theDog.toString()
```

Backward Compatibility

JavaScript 1.2. The behavior of the `toString` method depends on whether you specify `LANGUAGE="JavaScript1.2"` in the `<SCRIPT>` tag:

- If you specify `LANGUAGE="JavaScript1.2"` in the `<SCRIPT>` tag, the `toString` method returns an object literal.
- If you do not specify `LANGUAGE="JavaScript1.2"` in the `<SCRIPT>` tag, the `toString` method returns `[object` *`type`*`]`, as with other JavaScript versions.

Examples

Example 1: The location object. The following example prints the string equivalent of the current location.

```
document.write("location.toString() is " + location.toString() + "<BR>")
```

The output is as follows:

```
location.toString() is file:///C|/TEMP/myprog.html
```

Example 2: Object with no string value. Assume you have an `Image` object named `sealife` defined as follows:

```
<IMG NAME="sealife" SRC="images\seaotter.gif" ALIGN="left" VSPACE="10">
```

Because the `Image` object itself has no special `toString` method, `sealife.toString()` returns the following:

```
[object Image]
```

Example 3: The radix parameter. The following example prints the string equivalents of the numbers 0 through 9 in decimal and binary.

```
for (x = 0; x < 10; x++) {
   document.write("Decimal: ", x.toString(10), " Binary: ",
      x.toString(2), "<BR>")
}
```

The preceding example produces the following output:

```
Decimal: 0 Binary: 0
Decimal: 1 Binary: 1
Decimal: 2 Binary: 10
Decimal: 3 Binary: 11
Decimal: 4 Binary: 100
Decimal: 5 Binary: 101
Decimal: 6 Binary: 110
Decimal: 7 Binary: 111
Decimal: 8 Binary: 1000
Decimal: 9 Binary: 1001
```

See also `Object.toSource`, `Object.valueOf`

unwatch

Removes a watchpoint set with the `watch` method.

Method of	`Object`
Implemented in	JavaScript 1.2, NES 3.0

Syntax `unwatch(`*`prop`*`)`

Parameters

`prop`	The name of a property of the object.

Description The JavaScript debugger has functionality similar to that provided by this method, as well as other debugging options. For information on the debugger, see *Getting Started with Netscape JavaScript Debugger*.

By default, this method is inherited by every object descended from `Object`.

Example See `watch`.

valueOf

Returns the primitive value of the specified object.

Method of	`Object`
Implemented in	JavaScript 1.1
ECMA version	ECMA-262

Syntax `valueOf()`

Parameters None

Description JavaScript calls the `valueOf` method to convert an object to a primitive value. You rarely need to invoke the `valueOf` method yourself; JavaScript automatically invokes it when encountering an object where a primitive value is expected.

By default, the `valueOf` method is inherited by every object descended from `Object`. Every built-in core object overrides this method to return an appropriate value. If an object has no primitive value, `valueOf` returns the object itself, which is displayed as:

```
[object Object]
```

You can use `valueOf` within your own code to convert a built-in object into a primitive value. When you create a custom object, you can override `Object.valueOf` to call a custom method instead of the default `Object` method.

Overriding valueOf for custom objects. You can create a function to be called in place of the default `valueOf` method. Your function must take no arguments.

Suppose you have an object type `myNumberType` and you want to create a `valueOf` method for it. The following code assigns a user-defined function to the object's `valueOf` method:

```
myNumberType.prototype.valueOf = new Function(functionText)
```

With the preceding code in place, any time an object of type `myNumberType` is used in a context where it is to be represented as a primitive value, JavaScript automatically calls the function defined in the preceding code.

An object's `valueOf` method is usually invoked by JavaScript, but you can invoke it yourself as follows:

```
myNumber.valueOf()
```

Note Objects in string contexts convert via the `toString` method, which is different from `String` objects converting to string primitives using `valueOf`. All string objects have a string conversion, if only `"[object type]"`. But many objects do not convert to number, boolean, or function.

See also `parseInt`, `Object.toString`

watch

Watches for a property to be assigned a value and runs a function when that occurs.

Method of	`Object`
Implemented in	JavaScript 1.2, NES 3.0

Syntax `watch(`*`prop`*`, `*`handler`*`)`

Parameters

`prop`	The name of a property of the object.
`handler`	A function to call.

Description Watches for assignment to a property named `prop` in this object, calling `handler(prop, oldval, newval)` whenever `prop` is set and storing the return value in that property. A watchpoint can filter (or nullify) the value assignment, by returning a modified `newval` (or `oldval`).

If you delete a property for which a watchpoint has been set, that watchpoint does not disappear. If you later recreate the property, the watchpoint is still in effect.

To remove a watchpoint, use the `unwatch` method. By default, the `watch` method is inherited by every object descended from `Object`.

The JavaScript debugger has functionality similar to that provided by this method, as well as other debugging options. For information on the debugger, see *Getting Started with Netscape JavaScript Debugger*.

Example

```
<script language="JavaScript1.2">
o = {p:1}
o.watch("p",
   function (id,oldval,newval) {
      document.writeln("o." + id + " changed from "
         + oldval + " to " + newval)
      return newval
   })

o.p = 2
o.p = 3
delete o.p
o.p = 4

o.unwatch('p')
o.p = 5

</script>
```

This script displays the following:

```
o.p changed from 1 to 2
o.p changed from 2 to 3
o.p changed from 3 to 4
```

Option

An option in a selection list.

Client-side object

Implemented in	JavaScript 1.0
	JavaScript 1.1: added `defaultSelected` property; `text` property can be changed to change the text of an option

Created by The `Option` constructor or the HTML `OPTION` tag. To create an `Option` object with its constructor:

```
new Option([text[, value[, defaultSelected[, selected]]]])
```

Once you've created an Option object, you can add it to a selection list using the `Select.options` array.

Parameters

`text`	Specifies the text to display in the select list.
`value`	Specifies a value that is returned to the server when the option is selected and the form is submitted.
`defaultSelected`	Specifies whether the option is initially selected (true or false).
`selected`	Specifies the current selection state of the option (true or false).

Property Summary

Property	Description
`defaultSelected`	Specifies the initial selection state of the option
`index`	The zero-based index of an element in the `Select.options` array.
`length`	The number of elements in the `Select.options` array.
`selected`	Specifies the current selection state of the option
`text`	Specifies the text for the option
`value`	Specifies the value that is returned to the server when the option is selected and the form is submitted

Method Summary This object inherits the `watch` and `unwatch` methods from `Object`.

Description Usually you work with `Option` objects in the context of a selection list (a `Select` object). When JavaScript creates a `Select` object for each `SELECT` tag in the document, it creates `Option` objects for the `OPTION` tags inside the `SELECT` tag and puts those objects in the `options` array of the `Select` object.

In addition, you can create new options using the `Option` constructor and add those to a selection list. After you create an option and add it to the `Select` object, you must refresh the document by using `history.go(0)`. This statement must be last. When the document reloads, variables are lost if not saved in cookies or form element values.

You can use the `Option.selected` and `Select.selectedIndex` properties to change the selection state of an option.

- The `Select.selectedIndex` property is an integer specifying the index of the selected option. This is most useful for `Select` objects that are created without the `MULTIPLE` attribute. The following statement sets a `Select` object's `selectedIndex` property:

  ```
  document.myForm.musicTypes.selectedIndex = i
  ```

- The `Option.selected` property is a Boolean value specifying the current selection state of the option in a `Select` object. If an option is selected, its `selected` property is true; otherwise it is false. This is more useful for `Select` objects that are created with the `MULTIPLE` attribute. The following statement sets an option's `selected` property to true:

  ```
  document.myForm.musicTypes.options[i].selected = true
  ```

To change an option's text, use is `Option.text` property. For example, suppose a form has the following `Select` object:

```
<SELECT name="userChoice">
   <OPTION>Choice 1
   <OPTION>Choice 2
   <OPTION>Choice 3
</SELECT>
```

You can set the text of the `i`th item in the selection based on text entered in a text field named `whatsNew` as follows:

```
myform.userChoice.options[i].text = myform.whatsNew.value
```

You do not need to reload or refresh after changing an option's text.

Examples The following example creates two `Select` objects, one with and one without the `MULTIPLE` attribute. No options are initially defined for either object. When the user clicks a button associated with the `Select` object, the `populate` function creates four options for the `Select` object and selects the first option.

```
<SCRIPT>
function populate(inForm) {
   colorArray = new Array("Red", "Blue", "Yellow", "Green")

   var option0 = new Option("Red", "color_red")
   var option1 = new Option("Blue", "color_blue")
   var option2 = new Option("Yellow", "color_yellow")
   var option3 = new Option("Green", "color_green")

   for (var i=0; i < 4; i++) {
      eval("inForm.selectTest.options[i]=option" + i)
      if (i==0) {
         inForm.selectTest.options[i].selected=true
      }
   }

   history.go(0)
}
</SCRIPT>


<H3>Select Option() constructor</H3>
<FORM>
<SELECT NAME="selectTest"></SELECT><P>
<INPUT TYPE="button" VALUE="Populate Select List" onClick="populate(this.form)">
<P>
</FORM>

<HR>
<H3>Select-Multiple Option() constructor</H3>
<FORM>
<SELECT NAME="selectTest" multiple></SELECT><P>
<INPUT TYPE="button" VALUE="Populate Select List" onClick="populate(this.form)">
</FORM>
```

defaultSelected

A Boolean value indicating the default selection state of an option in a selection list.

Property of	`Option`
Implemented in	JavaScript 1.1

Security **JavaScript 1.1.** This property is tainted by default. For information on data tainting, see the *Client-Side JavaScript Guide*.

Description If an option is selected by default, the value of the `defaultSelected` property is true; otherwise, it is false. `defaultSelected` initially reflects whether the `SELECTED` attribute is used within an `OPTION` tag; however, setting `defaultSelected` overrides the `SELECTED` attribute.

You can set the `defaultSelected` property at any time. The display of the corresponding `Select` object does not update when you set the `defaultSelected` property of an option, only when you set the `Option.selected` or `Select.selectedIndex` properties.

A `Select` object created without the `MULTIPLE` attribute can have only one option selected by default. When you set `defaultSelected` in such an object, any previous default selections, including defaults set with the `SELECTED` attribute, are cleared. If you set `defaultSelected` in a `Select` object created with the `MULTIPLE` attribute, previous default selections are not affected.

Examples In the following example, the `restoreDefault` function returns the `musicType` `Select` object to its default state. The `for` loop uses the `options` array to evaluate every option in the `Select` object. The `if` statement sets the `selected` property if `defaultSelected` is true.

```
function restoreDefault() {
   for (var i = 0; i < document.musicForm.musicType.length; i++) {
      if (document.musicForm.musicType.options[i].defaultSelected == true) {
         document.musicForm.musicType.options[i].selected=true
      }
   }
}
```

The previous example assumes that the `Select` object is similar to the following:

```
<SELECT NAME="musicType">
   <OPTION SELECTED> R&B
   <OPTION> Jazz
   <OPTION> Blues
   <OPTION> New Age
</SELECT>
```

See also `Option.selected`, `Select.selectedIndex`

index

The zero-based index of an element in the `Select.options` array.

Property of	`Option`
Implemented in	JavaScript 1.0

Description The `index` property specifies the position of an element in the `Select.options` array, starting with 0.

Examples In the following example, the `getChoice` function returns the value of the `index` property for the selected option. The `for` loop evaluates every option in the `musicType` `Select` object. The `if` statement finds the option that is selected.

```
function getChoice() {
   for (var i = 0; i < document.musicForm.musicType.length; i++) {
      if (document.musicForm.musicType.options[i].selected == true) {
         return document.musicForm.musicType.options[i].index
      }
   }
   return null
}
```

The previous example assumes that the `Select` object is similar to the following:

```
<SELECT NAME="musicType">
   <OPTION SELECTED> R&B
   <OPTION> Jazz
   <OPTION> Blues
   <OPTION> New Age
</SELECT>
```

Note that you can also determine the index of the selected option in this example by using `document.musicForm.musicType.selectedIndex.`

length

The number of elements in the `Select.options` array.

Property of	`Option`
Read-only	
Implemented in	JavaScript 1.0

Description This value of this property is the same as the value of `Select.length`.

Examples See `Option.index` for an example of the `length` property.

selected

A Boolean value indicating whether an option in a `Select` object is selected.

Property of	`Option`
Implemented in	JavaScript 1.0

Security **JavaScript 1.1.** This property is tainted by default. For information on data tainting, see the *Client-Side JavaScript Guide*.

Description If an option in a `Select` object is selected, the value of its `selected` property is true; otherwise, it is false. You can set the `selected` property at any time. The display of the associated `Select` object updates immediately when you set the `selected` property for one of its options.

In general, the `Option.selected` property is more useful than the `Select.selectedIndex` property for `Select` objects that are created with the `MULTIPLE` attribute. With the `Option.selected` property, you can evaluate every option in the `Select.options` array to determine multiple selections, and you can select individual options without clearing the selection of other options.

Examples See the examples for `defaultSelected`.

See also `Option.defaultSelected`, `Select.selectedIndex`

text

A string specifying the text of an option in a selection list.

Property of	`Option`
Implemented in	JavaScript 1.0 JavaScript 1.1: The `text` property can be changed to updated the selection option. In previous releases, you could set the `text` property but the new value was not reflected in the `Select` object.

Security **JavaScript 1.1.** This property is tainted by default. For information on data tainting, see the *Client-Side JavaScript Guide*.

Description The `text` property initially reflects the text that follows an `OPTION` tag of a `SELECT` tag. You can set the `text` property at any time and the text displayed by the option in the selection list changes.

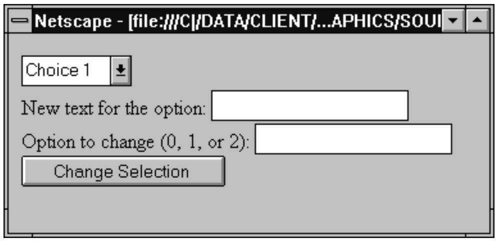
Examples **Example 1.** In the following example, the `getChoice` function returns the value of the `text` property for the selected option. The `for` loop evaluates every option in the `musicType` `Select` object. The `if` statement finds the option that is selected.

```
function getChoice() {
   for (var i = 0; i < document.musicForm.musicType.length; i++) {
      if (document.musicForm.musicType.options[i].selected == true) {
         return document.musicForm.musicType.options[i].text
      }
   }
   return null
}
```

The previous example assumes that the `Select` object is similar to the following:

```
<SELECT NAME="musicType">
   <OPTION SELECTED> R&B
   <OPTION> Jazz
   <OPTION> Blues
   <OPTION> New Age
</SELECT>
```

Example 2. In the following form, the user can enter some text in the first text field and then enter a number between 0 and 2 (inclusive) in the second text field. When the user clicks the button, the text is substituted for the indicated option number and that option is selected.



New text for the option:

Option to change (0, 1, or 2):

The code for this example looks as follows:

```
<SCRIPT>
function updateList(theForm, i) {
   theForm.userChoice.options[i].text = theForm.whatsNew.value
   theForm.userChoice.options[i].selected = true
}
</SCRIPT>
<FORM>
<SELECT name="userChoice">
   <OPTION>Choice 1
   <OPTION>Choice 2
   <OPTION>Choice 3
</SELECT>
<BR>
New text for the option: <INPUT TYPE="text" NAME="whatsNew">
<BR>
Option to change (0, 1, or 2): <INPUT TYPE="text" NAME="idx">
<BR>
<INPUT TYPE="button" VALUE="Change Selection"
onClick="updateList(this.form, this.form.idx.value)">
</FORM>
```

value

A string that reflects the `VALUE` attribute of the option.

Property of	`Option`
Read-only	
Implemented in	JavaScript 1.0

Security **JavaScript 1.1.** This property is tainted by default. For information on data tainting, see the *Client-Side JavaScript Guide*.

Description When a `VALUE` attribute is specified in HTML, the `value` property is a string that reflects it. When a `VALUE` attribute is not specified in HTML, the `value` property is the empty string. The `value` property is not displayed on the screen but is returned to the server if the option is selected.

Do not confuse the property with the selection state of the option or the text that is displayed next to it. The `selected` property determines the selection state of the object, and the `defaultSelected` property determines the default selection state. The text that is displayed is specified following the `OPTION` tag and corresponds to the `text` property.

Packages

A top-level object used to access Java classes from within JavaScript code.

Core object

Implemented in JavaScript 1.1, NES 2.0

Created by The `Packages` object is a top-level, predefined JavaScript object. You can automatically access it without using a constructor or calling a method.

Description The `Packages` object lets you access the public methods and fields of an arbitrary Java class from within JavaScript. The `java`, `netscape`, and `sun` properties represent the packages java.*, netscape.*, and sun.* respectively. Use standard Java dot notation to access the classes, methods, and fields in these packages. For example, you can access a constructor of the `Frame` class as follows:

```
var theFrame = new Packages.java.awt.Frame();
```

For convenience, JavaScript provides the top-level `netscape`, `sun`, and `java` objects that are synonyms for the `Packages` properties with the same names. Consequently, you can access Java classes in these packages without the Packages keyword, as follows:

```
var theFrame = new java.awt.Frame();
```

The `className` property represents the fully qualified path name of any other Java class that is available to JavaScript. You must use the `Packages` object to access classes outside the `netscape`, `sun`, and `java` packages.

Property Summary

Property	Description
`className`	The fully qualified name of a Java class in a package other than netscape, java, or sun that is available to JavaScript.
`java`	Any class in the Java package java.*.
`netscape`	Any class in the Java package netscape.*.
`sun`	Any class in the Java package sun.*.

Examples The following JavaScript function creates a Java dialog box:

```
function createWindow() {
   var theOwner = new Packages.java.awt.Frame();
   var theWindow = new Packages.java.awt.Dialog(theOwner);
   theWindow.setSize(350,200);
   theWindow.setTitle("Hello, World");
   theWindow.setVisible(true);
}
```

In the previous example, the function instantiates `theWindow` as a new `Packages` object. The `setSize`, `setTitle`, and `setVisible` methods are all available to JavaScript as public methods of `java.awt.Dialog`.

className

The fully qualified name of a Java class in a package other than `netscape`, `java`, or `sun` that is available to JavaScript.

Property of	`Packages`
Implemented in	JavaScript 1.1, NES 2.0

Syntax `Packages.`*`className`*

where *`classname`* is the fully qualified name of a Java class.

Description You must use the *`className`* property of the `Packages` object to access classes outside the `netscape`, `sun`, and `java` packages.

Examples The following code accesses the constructor of the `CorbaObject` class in the `myCompany` package from JavaScript:

```
var theObject = new Packages.myCompany.CorbaObject()
```

In the previous example, the value of the *`className`* property is `myCompany.CorbaObject,` the fully qualified path name of the `CorbaObject` class.

java

Any class in the Java package `java.*`.

Property of	`Packages`
Implemented in	JavaScript 1.1, NES 2.0

Syntax `Packages.java`

Description Use the `java` property to access any class in the `java` package from within JavaScript. Note that the top-level object `java` is a synonym for `Packages.java`.

Examples The following code accesses the constructor of the `java.awt.Frame` class:

```
var theOwner = new Packages.java.awt.Frame();
```

You can simplify this code by using the top-level java object to access the constructor as follows:

```
var theOwner = new java.awt.Frame();
```

netscape

Any class in the Java package `netscape.*`.

Property of	`Packages`
Implemented in	JavaScript 1.1, NES 2.0

Syntax `Packages.netscape`

Description Use the `netscape` property to access any class in the `netscape` package from within JavaScript. Note that the top-level object `netscape` is a synonym for `Packages.netscape`.

Examples See the example for .`Packages.java`

sun

Any class in the Java package `sun.*`.

Property of	`Packages`
Implemented in	JavaScript 1.1, NES 2.0

Syntax `Packages.sun`

Description Use the `sun` property to access any class in the `sun` package from within JavaScript. Note that the top-level object `sun` is a synonym for `Packages.sun`.

Examples See the example for .`Packages.java`

Password

A text field on an HTML form that conceals its value by displaying asterisks (*). When the user enters text into the field, asterisks (*) hide entries from view.

Client-side object

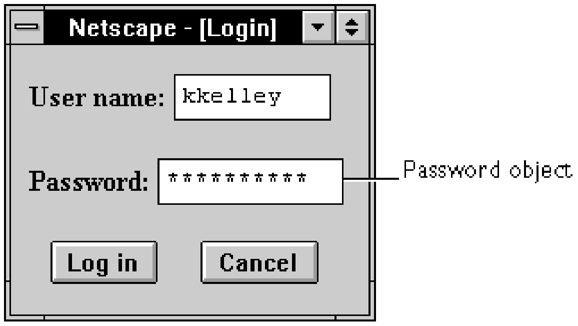
Implemented in	JavaScript 1.0 JavaScript 1.1: added `type` property; added onBlur and onFocus event handlers JavaScript 1.2: added `handleEvent` method.

Created by The HTML `INPUT` tag, with `"password"` as the value of the `TYPE` attribute. For a given form, the JavaScript runtime engine creates appropriate `Password` objects and puts these objects in the `elements` array of the corresponding `Form` object. You access a `Password` object by indexing this array. You can index the array either by number or, if supplied, by using the value of the `NAME` attribute.

Event handlers

- `onBlur`
- `onFocus`

Description A `Password` object on a form looks as follows:



A `Password` object is a form element and must be defined within a `FORM` tag.

Security **JavaScript versions 1.2 and later.** The `value` property is returned in plain text and has no security associated with it. Take care when using this property, and avoid storing its value in a cookie.

JavaScript 1.1. If a user interactively modifies the value in a password field, you cannot evaluate it accurately unless data tainting is enabled. For information on data tainting, see the *Client-Side JavaScript Guide*.

Property Summary

Property	Description
`defaultValue`	Reflects the `VALUE` attribute.
`form`	Specifies the form containing the `Password` object.
`name`	Reflects the `NAME` attribute.
`type`	Reflects the `TYPE` attribute.
`value`	Reflects the current value of the `Password` object's field.

Method Summary

Method	Description
`blur`	Removes focus from the object.
`focus`	Gives focus to the object.
`handleEvent`	Invokes the handler for the specified event.
`select`	Selects the input area of the object.

In addition, this object inherits the `watch` and `unwatch` methods from `Object`.

Examples The following example creates a `Password` object with no default value:

```
<B>Password:</B>
<INPUT TYPE="password" NAME="password" VALUE="" SIZE=25>
```

See also `Form`, `Text`

blur

Removes focus from the object.

Method of	`Password`
Implemented in	JavaScript 1.0

Syntax `blur()`

Parameters None

Examples The following example removes focus from the password element `userPass:`

```
userPass.blur()
```

This example assumes that the password is defined as

```
<INPUT TYPE="password" NAME="userPass">
```

See also `Password.focus`, `Password.select`

defaultValue

A string indicating the default value of a `Password` object.

Property of	`Password`
Implemented in	JavaScript 1.0

Security **JavaScript 1.1.** This property is tainted by default. For information on data tainting, see the *Client-Side JavaScript Guide*.

Description The initial value of `defaultValue` is null (for security reasons), regardless of the value of the `VALUE` attribute.

Setting `defaultValue` programmatically overrides the initial setting. If you programmatically set `defaultValue` for the `Password` object and then evaluate it, JavaScript returns the current value.

You can set the `defaultValue` property at any time. The display of the related object does not update when you set the `defaultValue` property, only when you set the `value` property.

See also `Password.value`

focus

Gives focus to the password object.

Method of `Password`

Implemented in JavaScript 1.0

Syntax `focus()`

Parameters None

Description Use the `focus` method to navigate to the password field and give it focus. You can then either programmatically enter a value in the field or let the user enter a value.

Examples In the following example, the `checkPassword` function confirms that a user has entered a valid password. If the password is not valid, the `focus` method returns focus to the `Password` object and the `select` method highlights it so the user can reenter the password.

```
function checkPassword(userPass) {
   if (badPassword) {
      alert("Please enter your password again.")
      userPass.focus()
      userPass.select()
   }
}
```

This example assumes that the `Password` object is defined as

```
<INPUT TYPE="password" NAME="userPass">
```

See also `Password.blur`, `Password.select`

form

An object reference specifying the form containing this object.

Property of `Password`

Read-only

Implemented in JavaScript 1.0

Description Each form element has a `form` property that is a reference to the element's parent form. This property is especially useful in event handlers, where you might need to refer to another element on the current form.

handleEvent

Invokes the handler for the specified event.

Method of	`Password`
Implemented in	JavaScript 1.2

Syntax `handleEvent(`*`event`*`)`

Parameters

`event`	The name of an event for which the object has an event handler.

Description For information on handling events, see the *Client-Side JavaScript Guide*.

name

A string specifying the name of this object.

Property of	`Password`
Implemented in	JavaScript 1.0

Security **JavaScript 1.1.** This property is tainted by default. For information on data tainting, see the *Client-Side JavaScript Guide*.

Description The `name` property initially reflects the value of the `NAME` attribute. Changing the `name` property overrides this setting. The `name` property is not displayed on-screen; it is used to refer to the objects programmatically.

If multiple objects on the same form have the same `NAME` attribute, an array of the given name is created automatically. Each element in the array represents an individual `Form` object. Elements are indexed in source order starting at 0. For example, if two `Text` elements and a `Password` element on the same form have their `NAME` attribute set to `"myField"`, an array with the elements `myField[0]`, `myField[1]`, and `myField[2]` is created. You need to be aware of this situation in your code and know whether `myField` refers to a single element or to an array of elements.

Examples In the following example, the `valueGetter` function uses a `for` loop to iterate over the array of elements on the `valueTest` form. The `msgWindow` window displays the names of all the elements on the form:

```
newWindow=window.open("http://home.netscape.com")

function valueGetter() {
   var msgWindow=window.open("")
   for (var i = 0; i < newWindow.document.valueTest.elements.length; i++) {
      msgWindow.document.write(newWindow.document.valueTest.elements[i].name + "<BR>")
   }
}
```

select

Selects the input area of the password field.

Method of	`Password`
Implemented in	JavaScript 1.0

Syntax `select()`

Parameters None

Description Use the `select` method to highlight the input area of the password field. You can use the `select` method with the `focus` method to highlight a field and position the cursor for a user response.

Examples In the following example, the `checkPassword` function confirms that a user has entered a valid password. If the password is not valid, the `select` method highlights the password field and the `focus` method returns focus to it so the user can reenter the password.

```
function checkPassword(userPass) {
   if (badPassword) {
      alert("Please enter your password again.")
      userPass.focus()
      userPass.select()
   }
}
```

This example assumes that the password is defined as

```
<INPUT TYPE="password" NAME="userPass">
```

See also `Password.blur`, `Password.focus`

type

For all `Password` objects, the value of the `type` property is `"password"`. This property specifies the form element's type.

Property of	`Password`
Read-only	
Implemented in	JavaScript 1.1

Examples The following example writes the value of the `type` property for every element on a form.

```
for (var i = 0; i < document.form1.elements.length; i++) {
   document.writeln("<BR>type is " + document.form1.elements[i].type)
}
```

value

A string that initially reflects the `VALUE` attribute.

Property of	`Password`
Implemented in	JavaScript 1.0

Security **JavaScript versions 1.2 and later.** This property is returned in plain text and has no security associated with it. Take care when using this property, and avoid storing its value in a cookie.

JavaScript 1.1. This property is tainted by default. If you programmatically set the `value` property and then evaluate it, JavaScript returns the current value. If a user interactively modifies the value in the password field, you cannot evaluate it accurately unless data tainting is enabled. For information on data tainting, see the *Client-Side JavaScript Guide*.

Description This string is represented by asterisks in the `Password` object field. The value of this property changes when a user or a program modifies the field, but the value is always displayed as asterisks.

See also `Password.defaultValue`

Plugin

A plug-in module installed on the client.

Client-side object

Implemented in	JavaScript 1.1

Created by `Plugin` objects are predefined JavaScript objects that you access through the `navigator.plugins` array.

Description A `Plugin` object is a plug-in installed on the client. A plug-in is a software module that the browser can invoke to display specialized types of embedded data within the browser. The user can obtain a list of installed plug-ins by choosing About Plug-ins from the Help menu.

Each `Plugin` object is itself array containing one element for each MIME type supported by the plug-in. Each element of the array is a `MimeType` object. For example, the following code displays the `type` and `description` properties of the first `Plugin` object's first `MimeType` object.

```
myPlugin=navigator.plugins[0]
myMimeType=myPlugin[0]
document.writeln('myMimeType.type is ',myMimeType.type,"<BR>")
document.writeln('myMimeType.description is ',myMimeType.description)
```

The preceding code displays output similar to the following:

```
myMimeType.type is video/quicktime
myMimeType.description is QuickTime for Windows
```

The `Plugin` object lets you dynamically determine which plug-ins are installed on the client. You can write scripts to display embedded plug-in data if the appropriate plug-in is installed, or display some alternative information such as images or text if not.

Plug-ins can be platform dependent and configurable, so a `Plugin` object's array of `MimeType` objects can vary from platform to platform, and from user to user.

Each `Plugin` object is an element in the `plugins` array.

When you use the `EMBED` tag to generate output from a plug-in application, you are not creating a `Plugin` object. Use the `document.embeds` array to refer to plug-in instances created with `EMBED` tags. See the `document.embeds` array.

Property Summary

Property	Description
`description`	A description of the plug-in.
`filename`	Name of the plug-in file on disk.
`length`	Number of elements in the plug-in's array of `MimeType` objects.
`name`	Name of the plug-in.

Method Summary This object inherits the `watch` and `unwatch` methods from `Object`.

Examples **Example 1.** The user can obtain a list of installed plug-ins by choosing About Plug-ins from the Help menu. To see the code the browser uses for this report, choose About Plug-ins from the Help menu, then choose Page Source from the View menu.

Example 2. The following code assigns shorthand variables for the predefined LiveAudio properties.

```
var myPluginName = navigator.plugins["LiveAudio"].name
var myPluginFile = navigator.plugins["LiveAudio"].filename
var myPluginDesc = navigator.plugins["LiveAudio"].description
```

Example 3. The following code displays the message "LiveAudio is configured for audio/wav" if the LiveAudio plug-in is installed and is enabled for the `"audio/wav"` MIME type:

```
var myPlugin = navigator.plugins["LiveAudio"]
var myType = myPlugin["audio/wav"]
if (myType && myType.enabledPlugin == myPlugin)
   document.writeln("LiveAudio is configured for audio/wav")
```

Example 4. The following expression represents the number of MIME types that Shockwave can display:

```
navigator.plugins["Shockwave"].length
```

Example 5. The following code displays the `name`, `filename`, `description`, and `length` properties for each `Plugin` object on a client:

```
document.writeln("<TABLE BORDER=1><TR VALIGN=TOP>",
   "<TH ALIGN=left>i",
   "<TH ALIGN=left>name",
   "<TH ALIGN=left>filename",
   "<TH ALIGN=left>description",
   "<TH ALIGN=left># of types</TR>")
for (i=0; i < navigator.plugins.length; i++) {
   document.writeln("<TR VALIGN=TOP><TD>",i,
      "<TD>",navigator.plugins[i].name,
      "<TD>",navigator.plugins[i].filename,
      "<TD>",navigator.plugins[i].description,
      "<TD>",navigator.plugins[i].length,
      "</TR>")
}
document.writeln("</TABLE>")
```

The preceding example displays output similar to the following:

i	name	filename	description	# of types
0	QuickTime Plug-In	d:\nettools\netscape\nav30\Program\plugins\NPQTW32.DLL	QuickTime Plug-In for Win32 v.1.0.0	1
1	LiveAudio	d:\nettools\netscape\nav30\Program\plugins\NPAUDIO.DLL	LiveAudio—Netscape Navigator sound playing component	7
2	NPAVI32 Dynamic Link Library	d:\nettools\netscape\nav30\Program\plugins\npavi32.dll	NPAVI32, avi plugin DLL	2
3	Netscape Default Plugin	d:\nettools\netscape\nav30\Program\plugins\npnul32.dll	Null Plugin	1

See also `MimeType`, `document.embeds`

description

A human-readable description of the plug-in. The text is provided by the plug-in developers.

Property of	`Plugin`
Read-only	
Implemented in	JavaScript 1.1

filename

The name of a plug-in file on disk.

Property of	`Plugin`
Read-only	
Implemented in	JavaScript 1.1

Description The `filename` property is the plug-in program's file name and is supplied by the plug-in itself. This name may vary from platform to platform.

Examples See the examples for `Plugin`.

length

The number of elements in the plug-in's array of `MimeType` objects.

Property of	`Plugin`
Read-only	
Implemented in	JavaScript 1.1

name

A string specifying the plug-in's name.

Property of	`Plugin`
Read-only	
Implemented in	JavaScript 1.1

Security **JavaScript 1.1.** This property is tainted by default. For information on data tainting, see the *Client-Side JavaScript Guide*.

Description The plug-in's name, supplied by the plug-in itself. Each plug-in should have a name that uniquely identifies it.

Radio

An individual radio button in a set of radio buttons on an HTML form. The user can use a set of radio buttons to choose one item from a list.

Client-side object

Implemented in	JavaScript 1.0 JavaScript 1.1: added `type` property; added `blur` and `focus` methods. JavaScript 1.2: added `handleEvent` method.

Created by The HTML `INPUT` tag, with `"radio"` as the value of the `TYPE` attribute. All the radio buttons in a single group must have the same value for the `NAME` attribute. This allows them to be accessed as a single group.

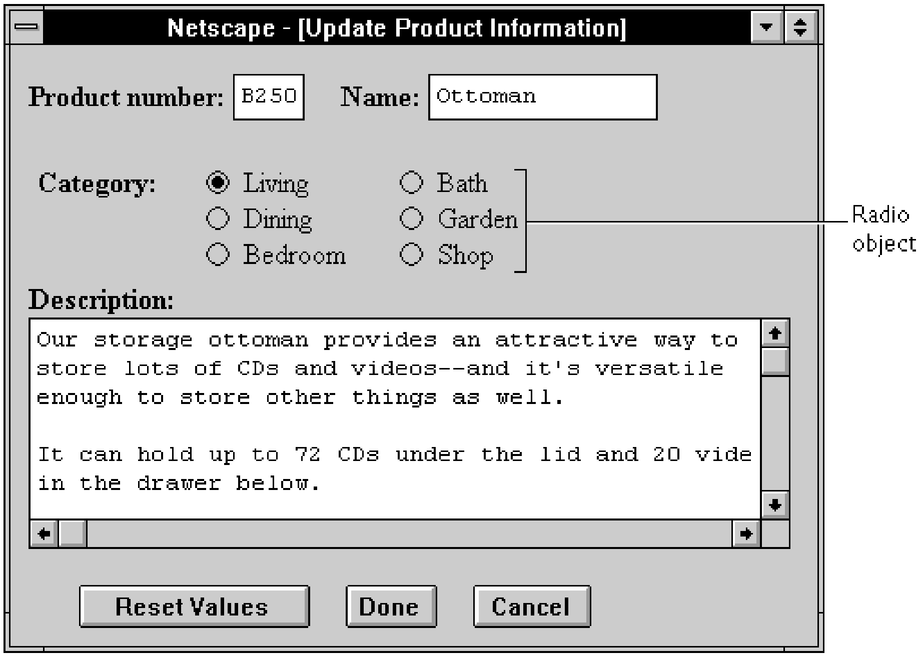
For a given form, the JavaScript runtime engine creates an individual `Radio` object for each radio button in that form. It puts in a single array all the `Radio` objects that have the same value for the `NAME` attribute. It puts that array in the `elements` array of the corresponding `Form` object. If a single form has multiple sets of radio buttons, the `elements` array has multiple `Radio` objects.

You access a set of buttons by accessing the `Form.elements` array (either by number or by using the value of the `NAME` attribute). To access the individual radio buttons in that set, you use the returned object array. For example, if your document has a form called `emp` with a set of radio buttons whose `NAME` attribute is `"dept"`, you would access the individual buttons as `document.emp.dept[0]`, `document.emp.dept[1]`, and so on.

Event handlers

- `onBlur`
- `onClick`
- `onFocus`

Description A `Radio` object on a form looks as follows:



```
Our storage ottoman provides an attractive way to
store lots of CDs and videos--and it's versatile
enough to store other things as well.

It can hold up to 72 CDs under the lid and 20 vide
in the drawer below.
```

A `Radio` object is a form element and must be defined within a `FORM` tag.

Property Summary

Property	Description
`checked`	Lets you programmatically select a radio button (property of the individual button).
`defaultChecked`	Reflects the `CHECKED` attribute (property of the individual button).
`form`	Specifies the form containing the `Radio` object (property of the array of buttons).
`name`	Reflects the `NAME` attribute (property of the array of buttons).
`type`	Reflects the `TYPE` attribute (property of the array of buttons).
`value`	Reflects the `VALUE` attribute (property of the array of buttons).

Method Summary

Method	Description
`blur`	Removes focus from the radio button.
`click`	Simulates a mouse-click on the radio button.
`focus`	Gives focus to the radio button.
`handleEvent`	Invokes the handler for the specified event.

In addition, this object inherits the `watch` and `unwatch` methods from `Object`.

Examples

Example 1. The following example defines a radio button group to choose among three music catalogs. Each radio button is given the same name, `NAME="musicChoice"`, forming a group of buttons for which only one choice can be selected. The example also defines a text field that defaults to what was chosen via the radio buttons but that allows the user to type a nonstandard catalog name as well. The `onClick` event handler sets the catalog name input field when the user clicks a radio button.

```
<INPUT TYPE="text" NAME="catalog" SIZE="20">
<INPUT TYPE="radio" NAME="musicChoice" VALUE="soul-and-r&b"
   onClick="musicForm.catalog.value = 'soul-and-r&b'"> Soul and R&B
<INPUT TYPE="radio" NAME="musicChoice" VALUE="jazz"
   onClick="musicForm.catalog.value = 'jazz'"> Jazz
<INPUT TYPE="radio" NAME="musicChoice" VALUE="classical"
   onClick="musicForm.catalog.value = 'classical'"> Classical
```

Example 2. The following example contains a form with three text boxes and three radio buttons. The radio buttons let the user choose whether the text fields are converted to uppercase or lowercase, or not converted at all. Each text field has an `onChange` event handler that converts the field value depending on which radio button is checked. The radio buttons for uppercase and lowercase have `onClick` event handlers that convert all fields when the user clicks the radio button.

```
<HTML>
<HEAD>
<TITLE>Radio object example</TITLE>
</HEAD>
<SCRIPT>
function convertField(field) {
   if (document.form1.conversion[0].checked) {
      field.value = field.value.toUpperCase()}
   else {
   if (document.form1.conversion[1].checked) {
      field.value = field.value.toLowerCase()}
   }
}
function convertAllFields(caseChange) {
   if (caseChange=="upper") {
   document.form1.lastName.value = document.form1.lastName.value.toUpperCase()
   document.form1.firstName.value = document.form1.firstName.value.toUpperCase()
   document.form1.cityName.value = document.form1.cityName.value.toUpperCase()}
   else {
   document.form1.lastName.value = document.form1.lastName.value.toLowerCase()
   document.form1.firstName.value = document.form1.firstName.value.toLowerCase()
   document.form1.cityName.value = document.form1.cityName.value.toLowerCase()
   }
}
</SCRIPT>
<BODY>
<FORM NAME="form1">
<B>Last name:</B>
<INPUT TYPE="text" NAME="lastName" SIZE=20 onChange="convertField(this)">
<BR><B>First name:</B>
<INPUT TYPE="text" NAME="firstName" SIZE=20 onChange="convertField(this)">
<BR><B>City:</B>
<INPUT TYPE="text" NAME="cityName" SIZE=20 onChange="convertField(this)">
<P><B>Convert values to:</B>
<BR><INPUT TYPE="radio" NAME="conversion" VALUE="upper"
   onClick="if (this.checked) {convertAllFields('upper')}"> Upper case
<BR><INPUT TYPE="radio" NAME="conversion" VALUE="lower"
   onClick="if (this.checked) {convertAllFields('lower')}"> Lower case
<BR><INPUT TYPE="radio" NAME="conversion" VALUE="noChange"> No conversion
</FORM>
</BODY>
</HTML>
```

See also the example for `Link`.

See also `Checkbox`, `Form`, `Select`

blur

Removes focus from the radio button.

Method of `Radio`

Implemented in JavaScript 1.0

Syntax `blur()`

Parameters None

See also `Radio.focus`

checked

A Boolean value specifying the selection state of a radio button.

Property of `Radio`

Implemented in JavaScript 1.0

Security **JavaScript 1.1.** This property is tainted by default. For information on data tainting, see the *Client-Side JavaScript Guide*.

Description If a radio button is selected, the value of its `checked` property is true; otherwise, it is false. You can set the `checked` property at any time. The display of the radio button updates immediately when you set the `checked` property.

At any given time, only one button in a set of radio buttons can be checked. When you set the `checked` property for one radio button in a group to true, that property for all other buttons in the group becomes false.

Examples The following example examines an array of radio buttons called `musicType` on the `musicForm` form to determine which button is selected. The `VALUE` attribute of the selected button is assigned to the `checkedButton` variable.

```
function stateChecker() {
   var checkedButton = ""
   for (var i in document.musicForm.musicType) {
      if (document.musicForm.musicType[i].checked=="1") {
         checkedButton=document.musicForm.musicType[i].value
      }
   }
}
```

See also `Radio.defaultChecked`

click

Simulates a mouse-click on the radio button, but does *not* trigger the button's `onClick` event handler.

Method of	`Radio`
Implemented in	JavaScript 1.0

Syntax `click()`

Parameters None

Examples The following example toggles the selection status of the first radio button in the `musicType` `Radio` object on the `musicForm` form:

```
document.musicForm.musicType[0].click()
```

The following example toggles the selection status of the `newAge` checkbox on the `musicForm` form:

```
document.musicForm.newAge.click()
```

defaultChecked

A Boolean value indicating the default selection state of a radio button.

Property of	`Radio`
Implemented in	JavaScript 1.0

Security **JavaScript 1.1.** This property is tainted by default. For information on data tainting, see the *Client-Side JavaScript Guide*.

Description If a radio button is selected by default, the value of the `defaultChecked` property is true; otherwise, it is false. `defaultChecked` initially reflects whether the `CHECKED` attribute is used within an `INPUT` tag; however, setting `defaultChecked` overrides the `CHECKED` attribute.

Unlike for the `checked` property, changing the value of `defaultChecked` for one button in a radio group does not change its value for the other buttons in the group.

You can set the `defaultChecked` property at any time. The display of the radio button does not update when you set the `defaultChecked` property, only when you set the `checked` property.

Examples The following example resets an array of radio buttons called `musicType` on the `musicForm` form to the default selection state:

```
function radioResetter() {
   var i=""
   for (i in document.musicForm.musicType) {
      if (document.musicForm.musicType[i].defaultChecked==true) {
         document.musicForm.musicType[i].checked=true
      }
   }
}
```

See also `Radio.checked`

focus

Gives focus to the radio button.

Method of `Radio`

Implemented in JavaScript 1.0

Syntax `focus()`

Parameters None

Description Use the `focus` method to navigate to the radio button and give it focus. The user can then easily toggle that button.

See also `Radio.blur`

form

An object reference specifying the form containing the radio button.

Property of `Radio`

Read-only

Implemented in JavaScript 1.0

Description Each form element has a `form` property that is a reference to the element's parent form. This property is especially useful in event handlers, where you might need to refer to another element on the current form.

handleEvent

Invokes the handler for the specified event.

Method of	`Radio`
Implemented in	JavaScript 1.2

Syntax `handleEvent(`*`event`*`)`

Parameters

`event`	The name of an event for which the specified object has an event handler.

name

A string specifying the name of the set of radio buttons with which this button is associated.

Property of	`Radio`
Implemented in	JavaScript 1.0

Security **JavaScript 1.1.** This property is tainted by default. For information on data tainting, see the *Client-Side JavaScript Guide*.

Description The `name` property initially reflects the value of the `NAME` attribute. Changing the `name` property overrides this setting.

All radio buttons that have the same value for their `name` property are in the same group and are treated together. If you change the `name` of a single radio button, you change which group of buttons it belongs to.

Do not confuse the name property with the label displayed on a Button. The `value` property specifies the label for the button. The `name` property is not displayed onscreen; it is used to refer programmatically to the button.

Examples In the following example, the `valueGetter` function uses a `for` loop to iterate over the array of elements on the `valueTest` form. The `msgWindow` window displays the names of all the elements on the form:

```
newWindow=window.open("http://home.netscape.com")

function valueGetter() {
   var msgWindow=window.open("")
   for (var i = 0; i < newWindow.document.valueTest.elements.length; i++) {
      msgWindow.document.write(newWindow.document.valueTest.elements[i].name + "<BR>")
   }
}
```

type

For all `Radio` objects, the value of the `type` property is `"radio"`. This property specifies the form element's type.

Property of	`Radio`
Read-only	
Implemented in	JavaScript 1.1

Examples The following example writes the value of the `type` property for every element on a form.

```
for (var i = 0; i < document.form1.elements.length; i++) {
   document.writeln("<BR>type is " + document.form1.elements[i].type)
}
```

value

A string that reflects the `VALUE` attribute of the radio button.

Property of	`Radio`
Read-only	
Implemented in	JavaScript 1.0

Security **JavaScript 1.1.** This property is tainted by default. For information on data tainting, see the *Client-Side JavaScript Guide*.

Description When a `VALUE` attribute is specified in HTML, the `value` property is a string that reflects it. When a `VALUE` attribute is not specified in HTML, the `value` property is a string that evaluates to `"on"`. The `value` property is not displayed on the screen but is returned to the server if the radio button or checkbox is selected.

Do not confuse the property with the selection state of the radio button or the text that is displayed next to the button. The `checked` property determines the selection state of the object, and the `defaultChecked` property determines the default selection state. The text that is displayed is specified following the `INPUT` tag.

Examples The following function evaluates the `value` property of a group of radio buttons and displays it in the `msgWindow` window:

```
function valueGetter() {
   var msgWindow=window.open("")
   for (var i = 0; i < document.valueTest.radioObj.length; i++) {
       msgWindow.document.write
          ("The value of radioObj[" + i + "] is " +
          document.valueTest.radioObj[i].value +"<BR>")
   }
   msgWindow.document.close()
}
```

This example displays the following values:

```
on
on
on
on
```

The previous example assumes the buttons have been defined as follows:

```
<BR><INPUT TYPE="radio" NAME="radioObj">R&B
<BR><INPUT TYPE="radio" NAME="radioObj" CHECKED>Soul
<BR><INPUT TYPE="radio" NAME="radioObj">Rock and Roll
<BR><INPUT TYPE="radio" NAME="radioObj">Blues
```

See also `Radio.checked`, `Radio.defaultChecked`

RegExp

A regular expression object contains the pattern of a regular expression. It has properties and methods for using that regular expression to find and replace matches in strings.

In addition to the properties of an individual regular expression object that you create using the `RegExp` constructor function, the predefined `RegExp` object has static properties that are set whenever any regular expression is used.
Core object

Implemented in	JavaScript 1.2, NES 3.0 JavaScript 1.3: added `toSource` method

Created by A literal text format or the `RegExp` constructor function.

The literal format is used as follows:

```
/pattern/flags
```

The constructor function is used as follows:

```
new RegExp("pattern"[, "flags"])
```

Parameters

`pattern`	The text of the regular expression.
`flags`	If specified, flags can have one of the following values: • `g`: global match • `i`: ignore case • `gi`: both global match and ignore case

Notice that the parameters to the literal format do not use quotation marks to indicate strings, while the parameters to the constructor function do use quotation marks. So the following expressions create the same regular expression:

```
/ab+c/i
new RegExp("ab+c", "i")
```

Description When using the constructor function, the normal string escape rules (preceding special characters with \ when included in a string) are necessary. For example, the following are equivalent:

```
re = new RegExp("\\w+")
re = /\w+/
```

The following table provides a complete list and description of the special characters that can be used in regular expressions.

Table 1.3 Special characters in regular expressions.

Character	Meaning
`\`	For characters that are usually treated literally, indicates that the next character is special and not to be interpreted literally. For example, `/b/` matches the character 'b'. By placing a backslash in front of b, that is by using `/\b/`, the character becomes special to mean match a word boundary. -or- For characters that are usually treated specially, indicates that the next character is not special and should be interpreted literally. For example, `*` is a special character that means 0 or more occurrences of the preceding character should be matched; for example, `/a*/` means match 0 or more a's. To match `*` literally, precede the it with a backslash; for example, `/a\*/` matches 'a*'.
`^`	Matches beginning of input or line. For example, `/^A/` does not match the 'A' in "an A," but does match it in "An A."
`$`	Matches end of input or line. For example, `/t$/` does not match the 't' in "eater", but does match it in "eat"
`*`	Matches the preceding character 0 or more times. For example, `/bo*/` matches 'boooo' in "A ghost booooed" and 'b' in "A bird warbled", but nothing in "A goat grunted".
`+`	Matches the preceding character 1 or more times. Equivalent to `{1,}`. For example, `/a+/` matches the 'a' in "candy" and all the a's in "caaaaaaandy."
`?`	Matches the preceding character 0 or 1 time. For example, `/e?le?/` matches the 'el' in "angel" and the 'le' in "angle."

Table 1.3 Special characters in regular expressions. (Continued)

Character	Meaning
`.`	(The decimal point) matches any single character except the newline character. For example, `/.n/` matches 'an' and 'on' in "nay, an apple is on the tree", but not 'nay'.
`(x)`	Matches 'x' and remembers the match. For example, `/(foo)/` matches and remembers 'foo' in "foo bar." The matched substring can be recalled from the resulting array's elements `[1]`, ..., `[n]`, or from the predefined `RegExp` object's properties `$1`, ..., `$9`.
`x|y`	Matches either 'x' or 'y'. For example, `/green|red/` matches 'green' in "green apple" and 'red' in "red apple."
`{n}`	Where `n` is a positive integer. Matches exactly `n` occurrences of the preceding character. For example, `/a{2}/` doesn't match the 'a' in "candy," but it matches all of the a's in "caandy," and the first two a's in "caaandy."
`{n,}`	Where `n` is a positive integer. Matches at least `n` occurrences of the preceding character. For example, `/a{2,}` doesn't match the 'a' in "candy", but matches all of the a's in "caandy" and in "caaaaaaandy."
`{n,m}`	Where `n` and `m` are positive integers. Matches at least `n` and at most `m` occurrences of the preceding character. For example, `/a{1,3}/` matches nothing in "cndy", the 'a' in "candy," the first two a's in "caandy," and the first three a's in "caaaaaaandy" Notice that when matching "caaaaaaandy", the match is "aaa", even though the original string had more a's in it.
`[xyz]`	A character set. Matches any one of the enclosed characters. You can specify a range of characters by using a hyphen. For example, `[abcd]` is the same as `[a-c]`. They match the 'b' in "brisket" and the 'c' in "ache"`.`
`[^xyz]`	A negated or complemented character set. That is, it matches anything that is not enclosed in the brackets. You can specify a range of characters by using a hyphen. For example, `[^abc]` is the same as `[^a-c]`. They initially match 'r' in "brisket" and 'h' in "chop."
`[\b]`	Matches a backspace. (Not to be confused with `\b`.)

Table 1.3 Special characters in regular expressions. (Continued)

Character	Meaning
`\b`	Matches a word boundary, such as a space. (Not to be confused with `[\b]`.) For example, `/\bn\w/` matches the 'no' in "noonday";`/\wy\b/` matches the 'ly' in "possibly yesterday."
`\B`	Matches a non-word boundary. For example, `/\w\Bn/` matches 'on' in "noonday", and `/y\B\w/` matches 'ye' in "possibly yesterday."
`\c`*X*	Where *X* is a control character. Matches a control character in a string. For example, `/\cM/` matches control-M in a string.
`\d`	Matches a digit character. Equivalent to `[0-9]`. For example, `/\d/` or `/[0-9]/` matches '2' in "B2 is the suite number."
`\D`	Matches any non-digit character. Equivalent to `[^0-9]`. For example, `/\D/` or `/[^0-9]/` matches 'B' in "B2 is the suite number."
`\f`	Matches a form-feed.
`\n`	Matches a linefeed.
`\r`	Matches a carriage return.
`\s`	Matches a single white space character, including space, tab, form feed, line feed. Equivalent to `[ \f\n\r\t\v]`. for example, `/\s\w*/` matches ' bar' in "foo bar."
`\S`	Matches a single character other than white space. Equivalent to `[^ \f\n\r\t\v]`. For example, `/\S/\w*` matches 'foo' in "foo bar."
`\t`	Matches a tab
`\v`	Matches a vertical tab.
`\w`	Matches any alphanumeric character including the underscore. Equivalent to `[A-Za-z0-9_]`. For example, `/\w/` matches 'a' in "apple," '5' in "$5.28," and '3' in "3D."
`\W`	Matches any non-word character. Equivalent to `[^A-Za-z0-9_]`. For example, `/\W/` or `/[^$A-Za-z0-9_]/` matches '%' in "50%."

Table 1.3 Special characters in regular expressions. (Continued)

Character	Meaning
`\`*`n`*	Where *n* is a positive integer. A back reference to the last substring matching the *n* parenthetical in the regular expression (counting left parentheses). For example, `/apple(,)\sorange\1/` matches 'apple, orange', in "apple, orange, cherry, peach." A more complete example follows this table. **Note:** If the number of left parentheses is less than the number specified in \n, the \n is taken as an octal escape as described in the next row.

`\o`*`octal`*
`\x`*`hex`*
	Where `\o`*`octal`* is an octal escape value or `\x`*`hex`* is a hexadecimal escape value. Allows you to embed ASCII codes into regular expressions.

The literal notation provides compilation of the regular expression when the expression is evaluated. Use literal notation when the regular expression will remain constant. For example, if you use literal notation to construct a regular expression used in a loop, the regular expression won't be recompiled on each iteration.

The constructor of the regular expression object, for example, `new RegExp("ab+c")`, provides runtime compilation of the regular expression. Use the constructor function when you know the regular expression pattern will be changing, or you don't know the pattern and are getting it from another source, such as user input. Once you have a defined regular expression, and if the regular expression is used throughout the script and may change, you can use the `compile` method to compile a new regular expression for efficient reuse.

A separate predefined `RegExp` object is available in each window; that is, each separate thread of JavaScript execution gets its own `RegExp` object. Because each script runs to completion without interruption in a thread, this assures that different scripts do not overwrite values of the `RegExp` object.

The predefined `RegExp` object contains the static properties `input`, `multiline`, `lastMatch`, `lastParen`, `leftContext`, `rightContext`, and `$1` through `$9`. The `input` and `multiline` properties can be preset. The values for the other static properties are set after execution of the `exec` and `test` methods of an individual regular expression object, and after execution of the `match` and `replace` methods of `String`.

Property Summary

Note that several of the `RegExp` properties have both long and short (Perl-like) names. Both names always refer to the same value. Perl is the programming language from which JavaScript modeled its regular expressions.

Property	Description
`$1, ..., $9`	Parenthesized substring matches, if any.
`$_`	See `input`.
`$*`	See `multiline`.
`$&`	See `lastMatch`.
`$+`	See `lastParen`.
`$‘`	See `leftContext`.
`$’`	See `rightContext`.
`constructor`	Specifies the function that creates an object’s prototype.
`global`	Whether or not to test the regular expression against all possible matches in a string, or only against the first.
`ignoreCase`	Whether or not to ignore case while attempting a match in a string.
`input`	The string against which a regular expression is matched.
`lastIndex`	The index at which to start the next match.
`lastMatch`	The last matched characters.
`lastParen`	The last parenthesized substring match, if any.
`leftContext`	The substring preceding the most recent match.
`multiline`	Whether or not to search in strings across multiple lines.
`prototype`	Allows the addition of properties to all objects.
`rightContext`	The substring following the most recent match.
`source`	The text of the pattern.

Method Summary

Method	Description
`compile`	Compiles a regular expression object.
`exec`	Executes a search for a match in its string parameter.

Method	Description
`test`	Tests for a match in its string parameter.
`toSource`	Returns an object literal representing the specified object; you can use this value to create a new object. Overrides the `Object.toSource` method.
`toString`	Returns a string representing the specified object. Overrides the `Object.toString` method.
`valueOf`	Returns the primitive value of the specified object. Overrides the `Object.valueOf` method.

In addition, this object inherits the `watch` and `unwatch` methods from `Object`.

Examples

Example 1. The following script uses the `replace` method to switch the words in the string. For the replacement text, the script uses the values of the `$1` and `$2` properties of the global `RegExp` object. Note that the `RegExp` object name is not be prepended to the `$` properties when they are passed as the second argument to the `replace` method.

```
<SCRIPT LANGUAGE="JavaScript1.2">
re = /(\w+)\s(\w+)/;
str = "John Smith";
newstr=str.replace(re, "$2, $1");
document.write(newstr)
</SCRIPT>
```

This displays "Smith, John".

Example 2. In the following example, `RegExp.input` is set by the Change event. In the `getInfo` function, the `exec` method uses the value of `RegExp.input` as its argument. Note that `RegExp` is prepended to the `$` properties.

```
<HTML>

<SCRIPT LANGUAGE="JavaScript1.2">
function getInfo() {
   re = /(\w+)\s(\d+)/;
   re.exec();
   window.alert(RegExp.$1 + ", your age is " + RegExp.$2);
}
</SCRIPT>

Enter your first name and your age, and then press Enter.
```

```
<FORM>
<INPUT TYPE:"TEXT" NAME="NameAge" onChange="getInfo(this);">
</FORM>

</HTML>
```

$1, ..., $9

Properties that contain parenthesized substring matches, if any.

Property of	`RegExp`
Static, Read-only	
Implemented in	JavaScript 1.2, NES 3.0

Description Because `input` is static, it is not a property of an individual regular expression object. Instead, you always use it as `RegExp.input`.

The number of possible parenthesized substrings is unlimited, but the predefined `RegExp` object can only hold the last nine. You can access all parenthesized substrings through the returned array's indexes.

These properties can be used in the replacement text for the `String.replace` method. When used this way, do not prepend them with `RegExp`. The example below illustrates this. When parentheses are not included in the regular expression, the script interprets *`$n`*'s literally (where *`n`* is a positive integer).

Examples The following script uses the `replace` method to switch the words in the string. For the replacement text, the script uses the values of the `$1` and `$2` properties of the global `RegExp` object. Note that the `RegExp` object name is not be prepended to the `$` properties when they are passed as the second argument to the `replace` method.

```
<SCRIPT LANGUAGE="JavaScript1.2">
re = /(\w+)\s(\w+)/;
str = "John Smith";
newstr=str.replace(re, "$2, $1");
document.write(newstr)
</SCRIPT>
```

This displays "Smith, John".

$_

See `input`.

$*

See `multiline`.

$&

See `lastMatch`.

$+

See `lastParen`.

$‘

See `leftContext`.

$’

See `rightContext`.

compile

Compiles a regular expression object during execution of a script.

Method of	`RegExp`
Implemented in	JavaScript 1.2, NES 3.0

Syntax

```
regexp.compile(pattern[, flags])
```

Parameters

`regexp`	`The` name of the regular expression. It can be a variable name or a literal.
`pattern`	A string containing the text of the regular expression.
`flags`	If specified, flags can have one of the following values: • `"g"`: global match • `"i"`: ignore case • `"gi"`: both global match and ignore case

Description Use the `compile` method to compile a regular expression created with the `RegExp` constructor function. This forces compilation of the regular expression once only which means the regular expression isn't compiled each time it is encountered. Use the `compile` method when you know the regular expression will remain constant (after getting its pattern) and will be used repeatedly throughout the script.

You can also use the `compile` method to change the regular expression during execution. For example, if the regular expression changes, you can use the `compile` method to recompile the object for more efficient repeated use.

Calling this method changes the value of the regular expression's `source`, `global`, and `ignoreCase` properties.

constructor

Specifies the function that creates an object's prototype. Note that the value of this property is a reference to the function itself, not a string containing the function's name.

Property of	`RegExp`
Implemented in	JavaScript 1.1, NES 2.0
ECMA version	ECMA-262

Description See `Object.constructor`.

exec

Executes the search for a match in a specified string. Returns a result array.

Method of	`RegExp`
Implemented in	JavaScript 1.2, NES 3.0

Syntax

```
regexp.exec([str])
regexp([str])
```

Parameters

`regexp`	`The` name of the regular expression. It can be a variable name or a literal.
`str`	The string against which to match the regular expression. If omitted, the value of `RegExp.input` is used.

Description As shown in the syntax description, a regular expression's `exec` method can be called either directly, (with `regexp.exec(str)`) or indirectly (with `regexp(str)`).

If you are executing a match simply to find `true` or `false`, use the `test` method or the `String` `search` method.

If the match succeeds, the `exec` method returns an array and updates properties of the regular expression object and the predefined regular expression object, `RegExp`. If the match fails, the `exec` method returns `null`.

Consider the following example:

```
<SCRIPT LANGUAGE="JavaScript1.2">
//Match one d followed by one or more b's followed by one d
//Remember matched b's and the following d
//Ignore case
myRe=/d(b+)(d)/ig;
myArray = myRe.exec("cdbBdbsbz");
</SCRIPT>
```

The following table shows the results for this script:

Object	Property/Index	Description	Example
`myArray`		The contents of `myArray`	`["dbBd", "bB", "d"]`
	`index`	The 0-based index of the match in the string	`1`
	`input`	The original string	`cdbBdbsbz`
	`[0]`	The last matched characters	`dbBd`

	`[1], ...[`*n*`]`	The parenthesized substring matches, if any. The number of possible parenthesized substrings is unlimited.	`[1] = bB`
			`[2] = d`

`myRe`	`lastIndex`	The index at which to start the next match.	`5`
	`ignoreCase`	Indicates if the `"i"` flag was used to ignore case	`true`
	`global`	Indicates if the `"g"` flag was used for a global match	`true`
	`source`	The text of the pattern	`d(b+)(d)`
`RegExp`	`lastMatch` `$&`	The last matched characters	`dbBd`

	`leftContext`
	`` $` ``
		The substring preceding the most recent match	`c`

	`rightContext` `$'`	The substring following the most recent match	`bsbz`

	`$1, ...$9`	The parenthesized substring matches, if any. The number of possible parenthesized substrings is unlimited, but `RegExp` can only hold the last nine.	`$1 = bB`
			`$2 = d`

	`lastParen` `$+`	The last parenthesized substring match, if any.	`d`

If your regular expression uses the `"g"` flag, you can use the `exec` method multiple times to find successive matches in the same string. When you do so, the search starts at the substring of `str` specified by the regular expression's `lastIndex` property. For example, assume you have this script:

```
<SCRIPT LANGUAGE="JavaScript1.2">
myRe=/ab*/g;
str = "abbcdefabh"
myArray = myRe.exec(str);
document.writeln("Found " + myArray[0] +
   ". Next match starts at " + myRe.lastIndex)
mySecondArray = myRe.exec(str);
document.writeln("Found " + mySecondArray[0] +
   ". Next match starts at " + myRe.lastIndex)
</SCRIPT>
```

This script displays the following text:

Found `abb`. Next match starts at 3
Found ab. Next match starts at 9

Examples In the following example, the user enters a name and the script executes a match against the input. It then cycles through the array to see if other names match the user's name.

This script assumes that first names of registered party attendees are preloaded into the array A, perhaps by gathering them from a party database.

```
<HTML>

<SCRIPT LANGUAGE="JavaScript1.2">
A = ["Frank", "Emily", "Jane", "Harry", "Nick", "Beth", "Rick",
      "Terrence", "Carol", "Ann", "Terry", "Frank", "Alice", "Rick",
      "Bill", "Tom", "Fiona", "Jane", "William", "Joan", "Beth"]
```

```
function lookup() {
   firstName = /\w+/i();
   if (!firstName)
      window.alert (RegExp.input + " isn't a name!");
   else {
      count = 0;
      for (i=0; i<A.length; i++)
         if (firstName[0].toLowerCase() == A[i].toLowerCase()) count++;
      if (count ==1)
         midstring = " other has ";
      else
         midstring = " others have ";
      window.alert ("Thanks, " + count + midstring + "the same name!")
   }
}

</SCRIPT>

Enter your first name and then press Enter.

<FORM> <INPUT TYPE:"TEXT" NAME="FirstName" onChange="lookup(this);"> </
FORM>

</HTML>
```

global

Whether or not the `"g"` flag is used with the regular expression.

Property of	`RegExp`
Read-only	
Implemented in	JavaScript 1.2, NES 3.0

Description `global` is a property of an individual regular expression object.

The value of `global` is `true` if the `"g"` flag was used; otherwise, `false`. The `"g"` flag indicates that the regular expression should be tested against all possible matches in a string.

You cannot change this property directly. However, calling the `compile` method changes the value of this property.

ignoreCase

Whether or not the `"i"` flag is used with the regular expression.

Property of	`RegExp`
Read-only	
Implemented in	JavaScript 1.2, NES 3.0

Description `ignoreCase` is a property of an individual regular expression object.

The value of `ignoreCase` is `true` if the `"i"` flag was used; otherwise, `false`. The `"i"` flag indicates that case should be ignored while attempting a match in a string.

You cannot change this property directly. However, calling the `compile` method changes the value of this property.

input

The string against which a regular expression is matched. `$_` is another name for the same property.

Property of	`RegExp`
Static	
Implemented in	JavaScript 1.2, NES 3.0

Description Because `input` is static, it is not a property of an individual regular expression object. Instead, you always use it as `RegExp.input`.

If no string argument is provided to a regular expression's `exec` or `test` methods, and if `RegExp.input` has a value, its value is used as the argument to that method.

The script or the browser can preset the `input` property. If preset and if no string argument is explicitly provided, the value of `input` is used as the string argument to the `exec` or `test` methods of the regular expression object. `input` is set by the browser in the following cases:

- When an event handler is called for a `TEXT` form element, `input` is set to the value of the contained text.
- When an event handler is called for a `TEXTAREA` form element, `input` is set to the value of the contained text. Note that `multiline` is also set to `true` so that the match can be executed over the multiple lines of text.
- When an event handler is called for a `SELECT` form element, `input` is set to the value of the selected text.
- When an event handler is called for a `Link` object, `input` is set to the value of the text between `<A HREF=...>` and `</A>`.

The value of the `input` property is cleared after the event handler completes.

lastIndex

A read/write integer property that specifies the index at which to start the next match.

Property of	`RegExp`
Implemented in	JavaScript 1.2, NES 3.0

Description `lastIndex` is a property of an individual regular expression object.

This property is set only if the regular expression used the `"g"` flag to indicate a global search. The following rules apply:

- If `lastIndex` is greater than the length of the string, `regexp.test` and `regexp.exec` fail, and `lastIndex` is set to `0`.
- If `lastIndex` is equal to the length of the string and if the regular expression matches the empty string, then the regular expression matches input starting at `lastIndex`.
- If `lastIndex` is equal to the length of the string and if the regular expression does not match the empty string, then the regular expression mismatches input, and `lastIndex` is reset to 0.

- Otherwise, `lastIndex` is set to the next position following the most recent match.

For example, consider the following sequence of statements:

`re = /(hi)?/g`	Matches the empty string.
`re("hi")`	Returns `["hi", "hi"]` with `lastIndex` equal to `2`.
`re("hi")`	Returns `[""]`, an empty array whose zeroth element is the match string. In this case, the empty string because `lastIndex` was 2 (and still is 2) and `"hi"` has length 2.

lastMatch

The last matched characters. `$&` is another name for the same property.

Property of	`RegExp`
Static, Read-only	
Implemented in	JavaScript 1.2, NES 3.0

Description Because `lastMatch` is static, it is not a property of an individual regular expression object. Instead, you always use it as `RegExp.lastMatch`.

lastParen

The last parenthesized substring match, if any. `$+` is another name for the same property.

Property of	`RegExp`
Static, Read-only	
Implemented in	JavaScript 1.2, NES 3.0

Description Because `lastParen` is static, it is not a property of an individual regular expression object. Instead, you always use it as `RegExp.lastParen`.

leftContext

The substring preceding the most recent match. `$‘` is another name for the same property.

Property of	`RegExp`
Static, Read-only	
Implemented in	JavaScript 1.2, NES 3.0

Description Because `leftContext` is static, it is not a property of an individual regular expression object. Instead, you always use it as `RegExp.leftContext`.

multiline

Reflects whether or not to search in strings across multiple lines. $* is another name for the same property.

Property of	`RegExp`
Static	
Implemented in	JavaScript 1.2, NES 3.0

Description Because `multiline` is static, it is not a property of an individual regular expression object. Instead, you always use it as `RegExp.multiline`.

The value of `multiline` is `true` if multiple lines are searched, `false` if searches must stop at line breaks.

The script or the browser can preset the `multiline` property. When an event handler is called for a `TEXTAREA` form element, the browser sets `multiline` to `true`. `multiline` is cleared after the event handler completes. This means that, if you've preset multiline to `true`, it is reset to `false` after the execution of any event handler.

prototype

Represents the prototype for this class. You can use the prototype to add properties or methods to all instances of a class. For information on prototypes, see `Function.prototype`.

Property of	`RegExp`
Implemented in	JavaScript 1.1, NES 2.0
ECMA version	ECMA-262

rightContext

The substring following the most recent match. `$'` is another name for the same property.

Property of	`RegExp`
Static, Read-only	
Implemented in	JavaScript 1.2, NES 3.0

Description Because `rightContext` is static, it is not a property of an individual regular expression object. Instead, you always use it as `RegExp.rightContext`.

source

A read-only property that contains the text of the pattern, excluding the forward slashes and `"g"` or `"i"` flags.

Property of	`RegExp`
Read-only	
Implemented in	JavaScript 1.2, NES 3.0

Description `source` is a property of an individual regular expression object.

You cannot change this property directly. However, calling the `compile` method changes the value of this property.

test

Executes the search for a match between a regular expression and a specified string. Returns `true` or `false`.

Method of	`RegExp`
Implemented in	JavaScript 1.2, NES 3.0

Syntax *regexp*.test([*str*])

Parameters

`regexp`	The name of the regular expression. It can be a variable name or a literal.
`str`	The string against which to match the regular expression. If omitted, the value of `RegExp.input` is used.

Description When you want to know whether a pattern is found in a string use the `test` method (similar to the `String.search` method); for more information (but slower execution) use the `exec` method (similar to the `String.match` method).

Example The following example prints a message which depends on the success of the test:

```
function testinput(re, str){
   if (re.test(str))
      midstring = " contains ";
   else
      midstring = " does not contain ";
   document.write (str + midstring + re.source);
}
```

toSource

Returns a string representing the source code of the object.

Method of	`RegExp`
Implemented in	JavaScript 1.3

Syntax `toSource()`

Parameters None

Description The `toSource` method returns the following values:

- For the built-in `RegExp` object, `toSource` returns the following string indicating that the source code is not available:

```
function Boolean() {
   [native code]
}
```

- For instances of `RegExp`, `toSource` returns a string representing the source code.

This method is usually called internally by JavaScript and not explicitly in code.

See also `Object.toSource`

toString

Returns a string representing the specified object.

Method of	`RegExp`
Implemented in	JavaScript 1.1, NES 2.0
ECMA version	ECMA-262

Syntax `toString()`

Parameters None.

Description The `RegExp` object overrides the `toString` method of the `Object` object; it does not inherit `Object.toString`. For `RegExp` objects, the `toString` method returns a string representation of the object.

Examples The following example displays the string value of a RegExp object:

```
myExp = new RegExp("a+b+c");
alert(myExp.toString())       displays "/a+b+c/"
```

See also `Object.toString`

valueOf

Returns the primitive value of a RegExp object.

Method of	`RegExp`
Implemented in	JavaScript 1.1
ECMA version	ECMA-262

Syntax `valueOf()`

Parameters None

Description The `valueOf` method of `RegExp` returns the primitive value of a RegExp object as a string data type. This value is equivalent to `RegExp.toString`.

This method is usually called internally by JavaScript and not explicitly in code.

Examples

```
myExp = new RegExp("a+b+c");
alert(myExp.valueOf())        displays "/a+b+c/"
```

See also `RegExp.toString`, `Object.valueOf`

Reset

A reset button on an HTML form. A reset button resets all elements in a form to their defaults.

Client-side object

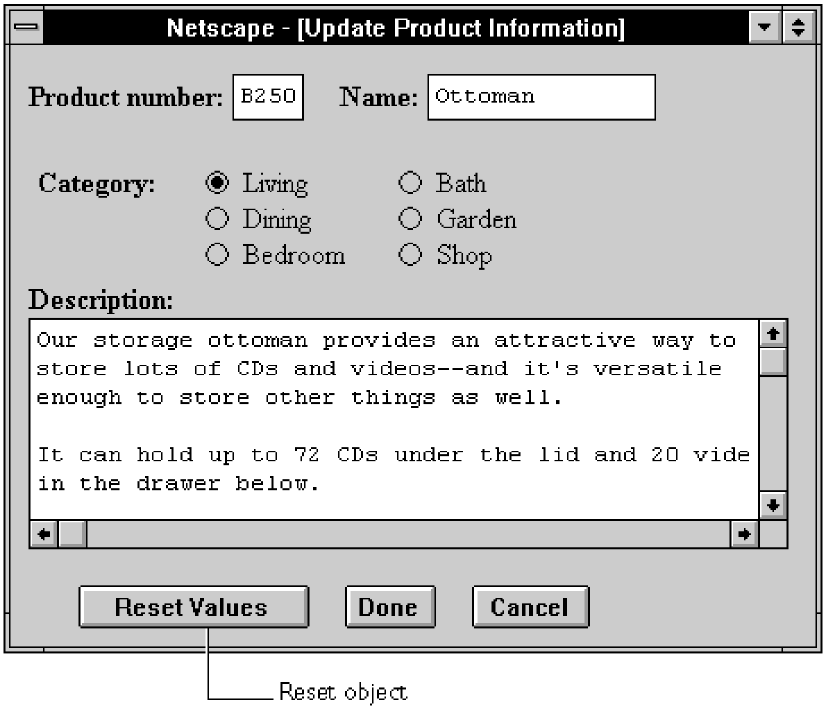
Implemented in	JavaScript 1.0
	JavaScript 1.1: added `type` property; added `onBlur` and `onFocus` event handlers; added `blur` and `focus` methods
	JavaScript 1.2: added `handleEvent` method

Created by The HTML `INPUT` tag, with `"reset"` as the value of the `TYPE` attribute. For a given form, the JavaScript runtime engine creates an appropriate `Reset` object and puts it in the `elements` array of the corresponding `Form` object. You access a `Reset` object by indexing this array. You can index the array either by number or, if supplied, by using the value of the `NAME` attribute.

Event handlers

- `onBlur`
- `onClick`
- `onFocus`

Description A `Reset` object on a form looks as follows:



Product number: B250 Name: Ottoman

Category: [✓ Living] Dining Bedroom Bath Garden Shop

Description:

```
Our storage ottoman provides an attractive way to
store lots of CDs and videos--and it's versatile
enough to store other things as well.

It can hold up to 72 CDs under the lid and 20 vide
in the drawer below.
```

A `Reset` object is a form element and must be defined within a `FORM` tag.

The reset button's `onClick` event handler cannot prevent a form from being reset; once the button is clicked, the reset cannot be canceled.

Property Summary

Property	Description
`form`	Specifies the form containing the `Reset` object.
`name`	Reflects the `NAME` attribute.
`type`	Reflects the `TYPE` attribute.
`value`	Reflects the `VALUE` attribute.

Method Summary

Method	Description
`blur`	Removes focus from the reset button.
`click`	Simulates a mouse-click on the reset button.
`focus`	Gives focus to the reset button.
`handleEvent`	Invokes the handler for the specified event.

In addition, this object inherits the `watch` and `unwatch` methods from `Object`.

Examples

Example 1. The following example displays a `Text` object with the default value "CA" and a reset button with the text "Clear Form" displayed on its face. If the user types a state abbreviation in the `Text` object and then clicks the Clear Form button, the original value of "CA" is restored.

```
<B>State: </B><INPUT TYPE="text" NAME="state" VALUE="CA" SIZE="2">
<P><INPUT TYPE="reset" VALUE="Clear Form">
```

Example 2. The following example displays two `Text` objects, a `Select` object, and three radio buttons; all of these objects have default values. The form also has a reset button with the text "Defaults" on its face. If the user changes the value of any of the objects and then clicks the Defaults button, the original values are restored.

```
<HTML>
<HEAD>
<TITLE>Reset object example</TITLE>
</HEAD>
<BODY>
<FORM NAME="form1">
<BR><B>City: </B><INPUT TYPE="text" NAME="city" VALUE="Santa Cruz" SIZE="20">
<B>State: </B><INPUT TYPE="text" NAME="state" VALUE="CA" SIZE="2">
<P><SELECT NAME="colorChoice">
   <OPTION SELECTED> Blue
   <OPTION> Yellow
   <OPTION> Green
   <OPTION> Red
</SELECT>
```

```
<P><INPUT TYPE="radio" NAME="musicChoice" VALUE="soul-and-r&b"
   CHECKED> Soul and R&B
<BR><INPUT TYPE="radio" NAME="musicChoice" VALUE="jazz">
   Jazz
<BR><INPUT TYPE="radio" NAME="musicChoice" VALUE="classical">
   Classical
<P><INPUT TYPE="reset" VALUE="Defaults" NAME="reset1">
</FORM>
</BODY>
</HTML>
```

See also `Button`, `Form`, `onReset`, `Form.reset`, `Submit`

blur

Removes focus from the reset button.

Method of	`Reset`
Implemented in	JavaScript 1.0

Syntax `blur()`

Parameters None

Examples The following example removes focus from the reset button `userReset:`

```
userReset.blur()
```

This example assumes that the button is defined as

```
<INPUT TYPE="reset" NAME="userReset">
```

See also `Reset.focus`

click

Simulates a mouse-click on the reset button, but does *not* trigger an object's `onClick` event handler.

Method of	`Reset`
Implemented in	JavaScript 1.0

Syntax `click()`

Parameters None

focus

Navigates to the reset button and gives it focus.

Method of	`Reset`
Implemented in	JavaScript 1.0

Syntax `focus()`

Parameters None

See also `Reset.blur`

form

An object reference specifying the form containing the reset button.

Property of	`Reset`
Read-only	
Implemented in	JavaScript 1.0

Description Each form element has a `form` property that is a reference to the element's parent form. This property is especially useful in event handlers, where you might need to refer to another element on the current form.

See also `Form`

handleEvent

Invokes the handler for the specified event.

Method of	`Reset`
Implemented in	JavaScript 1.2

Syntax `handleEvent(`*`event`*`)`

Parameters

`event`	The name of an event for which the specified object has an event handler.

name

A string specifying the name of the reset button.

Property of `Reset`

Implemented in JavaScript 1.0

Security **JavaScript 1.1.** This property is tainted by default. For information on data tainting, see the *Client-Side JavaScript Guide*.

Description The value of the `name` property initially reflects the value of the `NAME` attribute. Changing the `name` property overrides this setting.

Do not confuse the `name` property with the label displayed on the reset button. The `value` property specifies the label for this button. The `name` property is not displayed on the screen; it is used to refer programmatically to the button.

If multiple objects on the same form have the same `NAME` attribute, an array of the given name is created automatically. Each element in the array represents an individual `Form` object. Elements are indexed in source order starting at 0. For example, if two `Text` elements and a `Reset` element on the same form have their `NAME` attribute set to `"myField"`, an array with the elements `myField[0]`, `myField[1]`, and `myField[2]` is created. You need to be aware of this situation in your code and know whether `myField` refers to a single element or to an array of elements.

Examples In the following example, the `valueGetter` function uses a `for` loop to iterate over the array of elements on the `valueTest` form. The `msgWindow` window displays the names of all the elements on the form:

```
newWindow=window.open("http://home.netscape.com")

function valueGetter() {
   var msgWindow=window.open("")
   for (var i = 0; i < newWindow.document.valueTest.elements.length; i++) {
      msgWindow.document.write(newWindow.document.valueTest.elements[i].name + "<BR>")
   }
}
```

See also `Reset.value`

type

For all `Reset` objects, the value of the `type` property is `"reset"`. This property specifies the form element's type.

Property of	`Reset`
Read-only	
Implemented in	JavaScript 1.1

Examples The following example writes the value of the `type` property for every element on a form.

```
for (var i = 0; i < document.form1.elements.length; i++) {
   document.writeln("<BR>type is " + document.form1.elements[i].type)
}
```

value

A string that reflects the reset button's `VALUE` attribute.

Property of	`Reset`
Read-only	
Implemented in	JavaScript 1.0

Security **JavaScript 1.1.** This property is tainted by default. For information on data tainting, see the *Client-Side JavaScript Guide*.

Description This string is displayed on the face of the button. When a `VALUE` attribute is not specified in HTML, the `value` property is the string `"Reset"`.

Do not confuse the `value` property with the `name` property. The `name` property is not displayed on the screen; it is used to refer programmatically to the button.

Examples The following function evaluates the `value` property of a group of buttons and displays it in the `msgWindow` window:

```
function valueGetter() {
   var msgWindow=window.open("")
   msgWindow.document.write("submitButton.value is " +
      document.valueTest.submitButton.value + "<BR>")
   msgWindow.document.write("resetButton.value is " +
      document.valueTest.resetButton.value + "<BR>")
   msgWindow.document.write("helpButton.value is " +
      document.valueTest.helpButton.value + "<BR>")
   msgWindow.document.close()
}
```

This example displays the following values:

```
Query Submit
Reset
Help
```

The previous example assumes the buttons have been defined as follows:

```
<INPUT TYPE="submit" NAME="submitButton">
<INPUT TYPE="reset" NAME="resetButton">
<INPUT TYPE="button" NAME="helpButton" VALUE="Help">
```

See also `Reset.name`

screen

Contains properties describing the display screen and colors.
Client-side object

Implemented in JavaScript 1.2

Created by The JavaScript runtime engine creates the `screen` object for you. You can access its properties automatically.

Description This object contains read-only properties that allow you to get information about the user's display.

Property Summary

Method	Description
`availHeight`	Specifies the height of the screen, in pixels, minus permanent or semipermanent user interface features displayed by the operating system, such as the Taskbar on Windows.
`availLeft`	Specifies the x-coordinate of the first pixel that is not allocated to permanent or semipermanent user interface features.
`availTop`	Specifies the y-coordinate of the first pixel that is not allocated to permanent or semipermanent user interface features.
`availWidth`	Specifies the width of the screen, in pixels, minus permanent or semipermanent user interface features displayed by the operating system, such as the Taskbar on Windows.
`colorDepth`	The bit depth of the color palette, if one is in use; otherwise, the value is derived from `screen.pixelDepth`.
`height`	Display screen height.
`pixelDepth`	Display screen color resolution (bits per pixel).
`width`	Display screen width.

Method Summary This object inherits the `watch` and `unwatch` methods from `Object`.

Examples The following function creates a string containing the current display properties:

```
function screen_properties() {
   document.examples.results.value = "("+screen.width+" x
      "+screen.height+") pixels, "+
      screen.pixelDepth +" bit depth, "+
      screen.colorDepth +" bit color palette depth.";
} // end function screen_properties
```

availHeight

Specifies the height of the screen, in pixels, minus permanent or semipermanent user interface features displayed by the operating system, such as the Taskbar on Windows.

Property of	`screen`
Implemented in	JavaScript 1.2

See also `screen.availTop`

availLeft

Specifies the x-coordinate of the first pixel that is not allocated to permanent or semipermanent user interface features.

Property of	`screen`
Implemented in	JavaScript 1.2

See also `screen.availWidth`

availTop

Specifies the y-coordinate of the first pixel that is not allocated to permanent or semipermanent user interface features.

Property of	`screen`
Implemented in	JavaScript 1.2

See also `screen.availHeight`

availWidth

Specifies the width of the screen, in pixels, minus permanent or semipermanent user interface features displayed by the operating system, such as the Taskbar on Windows.

Property of	`screen`
Implemented in	JavaScript 1.2

See also `screen.availLeft`

colorDepth

The bit depth of the color palette in bits per pixel, if a color palette is in use. Otherwise, this property is derived from `screen.pixelDepth`.

Property of	`screen`
Implemented in	JavaScript 1.2

height

Display screen height, in pixels.

Property of	`screen`
Implemented in	JavaScript 1.2

pixelDepth

Display screen color resolution, in bits per pixel.

Property of	`screen`
Implemented in	JavaScript 1.2

width

Display screen width, in pixels.

Property of	`screen`
Implemented in	JavaScript 1.2

Select

A selection list on an HTML form. The user can choose one or more items from a selection list, depending on how the list was created.

Client-side object

Implemented in	JavaScript 1.0
	JavaScript 1.1: added `type` property; added the ability to add and delete options.
	JavaScript 1.2: added `handleEvent` method.

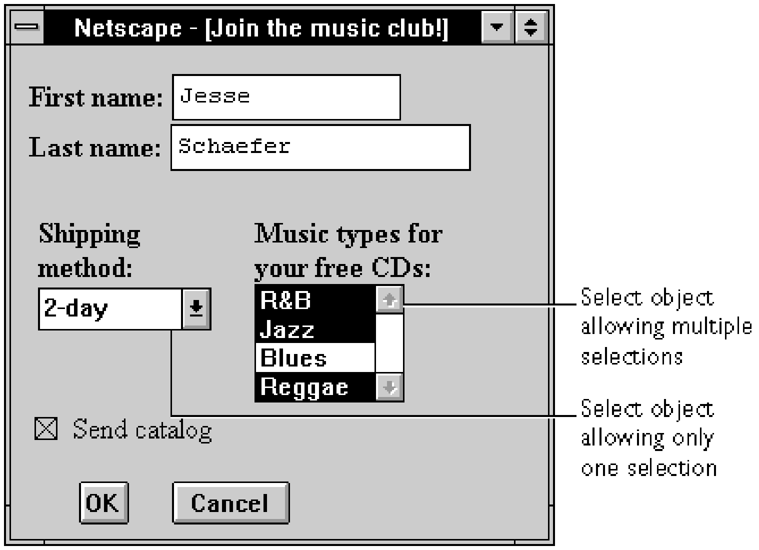
Created by The HTML `SELECT` tag. For a given form, the JavaScript runtime engine creates appropriate `Select` objects for each selection list and puts these objects in the `elements` array of the corresponding `Form` object. You access a `Select` object by indexing this array. You can index the array either by number or, if supplied, by using the value of the `NAME` attribute.

The runtime engine also creates `Option` objects for each `OPTION` tag inside the `SELECT` tag.

Event handlers

- `onBlur`
- `onChange`
- `onFocus`

Description The following figure shows a form containing two selection lists. The user can choose one item from the list on the left and can choose multiple items from the list on the right:


Netscape - [Join the music club!]
First name:
Jesse
Last name:
Schaefer
Shipping method:
2-day
Music types for your free CDs:
R&B
Jazz
Blues
Reggae
Send catalog
OK
Cancel
Select object allowing multiple selections
Select object allowing only one selection

A `Select` object is a form element and must be defined within a `FORM` tag.

Property Summary

Property	Description
`form`	Specifies the form containing the selection list.
`length`	Reflects the number of options in the selection list.
`name`	Reflects the `NAME` attribute.
`options`	Reflects the `OPTION` tags.
`selectedIndex`	Reflects the index of the selected option (or the first selected option, if multiple options are selected).
`type`	Specifies that the object is represents a selection list and whether it can have one or more selected options.

Method Summary

Method	Description
`blur`	Removes focus from the selection list.
`focus`	Gives focus to the selection list.
`handleEvent`	Invokes the handler for the specified event.

In addition, this object inherits the `watch` and `unwatch` methods from `Object`.

Examples **Example 1.** The following example displays two selection lists. In the first list, the user can select only one item; in the second list, the user can select multiple items.

```
Choose the music type for your free CD:
<SELECT NAME="music_type_single">
   <OPTION SELECTED> R&B
   <OPTION> Jazz
   <OPTION> Blues
   <OPTION> New Age
</SELECT>
<P>Choose the music types for your free CDs:
<BR><SELECT NAME="music_type_multi" MULTIPLE>
   <OPTION SELECTED> R&B
   <OPTION> Jazz
   <OPTION> Blues
   <OPTION> New Age
</SELECT>
```

Example 2. The following example displays two selection lists that let the user choose a month and day. These selection lists are initialized to the current date. The user can change the month and day by using the selection lists or by choosing preset dates from radio buttons. Text fields on the form display the values of the `Select` object's properties and indicate the date chosen and whether it is Cinco de Mayo.

```
<HTML>
<HEAD>
<TITLE>Select object example</TITLE>
</HEAD>
<BODY>
<SCRIPT>
var today = new Date()
//---------------
function updatePropertyDisplay(monthObj,dayObj) {
   // Get date strings
   var monthInteger, dayInteger, monthString, dayString
   monthInteger=monthObj.selectedIndex
   dayInteger=dayObj.selectedIndex
   monthString=monthObj.options[monthInteger].text
   dayString=dayObj.options[dayInteger].text
   // Display property values
   document.selectForm.textFullDate.value=monthString + " " + dayString
   document.selectForm.textMonthLength.value=monthObj.length
   document.selectForm.textDayLength.value=dayObj.length
   document.selectForm.textMonthName.value=monthObj.name
   document.selectForm.textDayName.value=dayObj.name
   document.selectForm.textMonthIndex.value=monthObj.selectedIndex
   document.selectForm.textDayIndex.value=dayObj.selectedIndex
   // Is it Cinco de Mayo?
   if (monthObj.options[4].selected && dayObj.options[4].selected)
      document.selectForm.textCinco.value="Yes!"
   else
      document.selectForm.textCinco.value="No"
}
</SCRIPT>
<!--------------->
<FORM NAME="selectForm">
<P><B>Choose a month and day:</B>
Month: <SELECT NAME="monthSelection"
   onChange="updatePropertyDisplay(this,document.selectForm.daySelection)">
   <OPTION> January <OPTION> February <OPTION> March
   <OPTION> April <OPTION> May <OPTION> June
   <OPTION> July <OPTION> August <OPTION> September
   <OPTION> October <OPTION> November <OPTION> December
</SELECT>
Day: <SELECT NAME="daySelection"
   onChange="updatePropertyDisplay(document.selectForm.monthSelection,this)">
   <OPTION> 1 <OPTION> 2 <OPTION> 3 <OPTION> 4 <OPTION> 5
   <OPTION> 6 <OPTION> 7 <OPTION> 8 <OPTION> 9 <OPTION> 10
   <OPTION> 11 <OPTION> 12 <OPTION> 13 <OPTION> 14 <OPTION> 15
   <OPTION> 16 <OPTION> 17 <OPTION> 18 <OPTION> 19 <OPTION> 20
   <OPTION> 21 <OPTION> 22 <OPTION> 23 <OPTION> 24 <OPTION> 25
   <OPTION> 26 <OPTION> 27 <OPTION> 28 <OPTION> 29 <OPTION> 30
   <OPTION> 31
</SELECT>
```

```
<P><B>Set the date to: </B>
<INPUT TYPE="radio" NAME="dateChoice"
   onClick="
      monthSelection.selectedIndex=0;
      daySelection.selectedIndex=0;
      updatePropertyDisplay
         document.selectForm.monthSelection,document.selectForm.daySelection)">
   New Year's Day
<INPUT TYPE="radio" NAME="dateChoice"
   onClick="
      monthSelection.selectedIndex=4;
      daySelection.selectedIndex=4;
      updatePropertyDisplay
         (document.selectForm.monthSelection,document.selectForm.daySelection)">
   Cinco de Mayo
<INPUT TYPE="radio" NAME="dateChoice"
   onClick="
      monthSelection.selectedIndex=5;
      daySelection.selectedIndex=20;
      updatePropertyDisplay
         (document.selectForm.monthSelection,document.selectForm.daySelection)">
   Summer Solstice
<P><B>Property values:</B>
<BR>Date chosen: <INPUT TYPE="text" NAME="textFullDate" VALUE="" SIZE=20">
<BR>monthSelection.length<INPUT TYPE="text" NAME="textMonthLength" VALUE="" SIZE=20">
<BR>daySelection.length<INPUT TYPE="text" NAME="textDayLength" VALUE="" SIZE=20">
<BR>monthSelection.name<INPUT TYPE="text" NAME="textMonthName" VALUE="" SIZE=20">
<BR>daySelection.name<INPUT TYPE="text" NAME="textDayName" VALUE="" SIZE=20">
<BR>monthSelection.selectedIndex
   <INPUT TYPE="text" NAME="textMonthIndex" VALUE="" SIZE=20">
<BR>daySelection.selectedIndex<INPUT TYPE="text" NAME="textDayIndex" VALUE="" SIZE=20">
<BR>Is it Cinco de Mayo? <INPUT TYPE="text" NAME="textCinco" VALUE="" SIZE=20">
<SCRIPT>
document.selectForm.monthSelection.selectedIndex=today.getMonth()
document.selectForm.daySelection.selectedIndex=today.getDate()-1
updatePropertyDisplay(document.selectForm.monthSelection,document.selectForm.daySelection)
</SCRIPT>
</FORM>
</BODY>
</HTML>
```

Example 3. Add an option with the Option constructor. The following example creates two `Select` objects, one with and one without the `MULTIPLE` attribute. No options are initially defined for either object. When the user clicks a button associated with the `Select` object, the `populate` function creates four options for the `Select` object and selects the first option.

```
<SCRIPT>
function populate(inForm) {
   colorArray = new Array("Red", "Blue", "Yellow", "Green")

   var option0 = new Option("Red", "color_red")
   var option1 = new Option("Blue", "color_blue")
   var option2 = new Option("Yellow", "color_yellow")
   var option3 = new Option("Green", "color_green")

   for (var i=0; i < 4; i++) {
      eval("inForm.selectTest.options[i]=option" + i)
      if (i==0) {
         inForm.selectTest.options[i].selected=true
      }
   }

   history.go(0)
}
</SCRIPT>


<H3>Select Option() constructor</H3>
<FORM>
<SELECT NAME="selectTest"></SELECT><P>
<INPUT TYPE="button" VALUE="Populate Select List" onClick="populate(this.form)">
<P>
</FORM>

<HR>
<H3>Select-Multiple Option() constructor</H3>
<FORM>
<SELECT NAME="selectTest" multiple></SELECT><P>
<INPUT TYPE="button" VALUE="Populate Select List" onClick="populate(this.form)">
</FORM>
```

Example 4. Delete an option. The following function removes an option from a `Select` object.

```
function deleteAnItem(theList,itemNo) {
   theList.options[itemNo]=null
   history.go(0)
}
```

See also `Form`, `Radio`

blur

Removes focus from the selection list.

Method of	`Select`
Implemented in	JavaScript 1.0

Syntax `blur()`

Parameters None

See also `Select.focus`

focus

Navigates to the selection list and gives it focus.

Method of	`Select`
Implemented in	JavaScript 1.0

Syntax `focus()`

Parameters None

Description Use the `focus` method to navigate to a selection list and give it focus. The user can then make selections from the list.

See also `Select.blur`

form

An object reference specifying the form containing the selection list.

Property of	`Select`
Read-only	
Implemented in	JavaScript 1.0

Description Each form element has a `form` property that is a reference to the element's parent form. This property is especially useful in event handlers, where you might need to refer to another element on the current form.

See also `Form`

handleEvent

Invokes the handler for the specified event.

Method of	`Select`
Implemented in	JavaScript 1.2

Syntax `handleEvent(`*`event`*`)`

Parameters

`event`	The name of an event for which the object has an event handler.

length

The number of options in the selection list.

Property of	`Select`
Read-only	
Implemented in	JavaScript 1.0

Description This value of this property is the same as the value of `Option.length`.

name

A string specifying the name of the selection list.

Property of	`Select`
Implemented in	JavaScript 1.0

Security **JavaScript 1.1.** This property is tainted by default. For information on data tainting, see the *Client-Side JavaScript Guide*.

Description The `name` property initially reflects the value of the `NAME` attribute. Changing the `name` property overrides this setting. The `name` property is not displayed on the screen; it is used to refer to the list programmatically.

If multiple objects on the same form have the same `NAME` attribute, an array of the given name is created automatically. Each element in the array represents an individual `Form` object. Elements are indexed in source order starting at 0. For example, if two `Text` elements and a `Select` element on the same form have their `NAME` attribute set to `"myField"`, an array with the elements `myField[0]`, `myField[1]`, and `myField[2]` is created. You need to be aware of this situation in your code and know whether `myField` refers to a single element or to an array of elements.

Examples In the following example, the `valueGetter` function uses a `for` loop to iterate over the array of elements on the `valueTest` form. The `msgWindow` window displays the names of all the elements on the form:

```
newWindow=window.open("http://home.netscape.com")

function valueGetter() {
   var msgWindow=window.open("")
   for (var i = 0; i < newWindow.document.valueTest.elements.length; i++) {
      msgWindow.document.write(newWindow.document.valueTest.elements[i].name + "<BR>")
   }
}
```

options

An array corresponding to options in a `Select` object in source order.

Property of	`Select`
Read-only	
Implemented in	JavaScript 1.0

Description You can refer to the options of a `Select` object by using the `options` array. This array contains an entry for each option in a `Select` object (`OPTION` tag) in source order. For example, if a `Select` object named `musicStyle` contains three options, you can access these options as `musicStyle.options[0]`, `musicStyle.options[1]`, and `musicStyle.options[2]`.

To obtain the number of options in the selection list, you can use either `Select.length` or the `length` property of the `options` array. For example, you can get the number of options in the `musicStyle` selection list with either of these expressions:

```
musicStyle.length
musicStyle.options.length
```

You can add or remove options from a selection list using this array. To add or replace an option to an existing `Select` object, you assign a new `Option` object to a place in the array. For example, to create a new `Option` object called `jeans` and add it to the end of the selection list named `myList`, you could use the following code:

```
jeans = new Option("Blue Jeans", "jeans", false, false);
myList.options[myList.length] = jeans;
```

To delete an option from a `Select` object, you set the appropriate index of the `options` array to null. Removing an option compresses the options array. For example, assume that `myList` has 5 elements in it, the value of the fourth element is `"foo"`, and you execute this statement:

```
myList.options[1] = null
```

Now, `myList` has 4 elements in it and the value of the *third* element is `"foo"`.

After you delete an option, you must refresh the document by using `history.go(0)`. This statement must be last. When the document reloads, variables are lost if not saved in cookies or form element values.

You can determine which option in a selection list is currently selected by using either the `selectedIndex` property of the `options` array or of the `Select` object itself. That is, the following expressions return the same value:

```
musicStyle.selectedIndex
musicStyle.options.selectedIndex
```

For more information about this property, see `Select.selectedIndex`.

For `Select` objects that can have multiple selections (that is, the `SELECT` tag has the `MULTIPLE` attribute), the `selectedIndex` property is not very useful. In this case, it returns the index of the first selection. To find all the selected options, you have to loop and test each option individually. For example, to print a list of all selected options in a selection list named `mySelect`, you could use code such as this:

```
document.write("You've selected the following options:\n")
for (var i = 0; i < mySelect.options.length; i++) {
   if (mySelect.options[i].selected)
      document.write(" mySelect.options[i].text\n")
}
```

In general, to work with individual options in a selection list, you work with the appropriate `Option` object.

selectedIndex

An integer specifying the index of the selected option in a `Select` object.

Property of	`Select`
Implemented in	JavaScript 1.0

Security **JavaScript 1.1.** This property is tainted by default. For information on data tainting, see the *Client-Side JavaScript Guide*.

Description Options in a `Select` object are indexed in the order in which they are defined, starting with an index of 0. You can set the `selectedIndex` property at any time. The display of the `Select` object updates immediately when you set the `selectedIndex` property.

If no option is selected, `selectedIndex` has a value of -1.

In general, the `selectedIndex` property is more useful for `Select` objects that are created without the `MULTIPLE` attribute. If you evaluate `selectedIndex` when multiple options are selected, the `selectedIndex` property specifies the index of the first option only. Setting `selectedIndex` clears any other options that are selected in the `Select` object.

The `Option.selected` property is more useful in conjunction with `Select` objects that are created with the `MULTIPLE` attribute. With the `Option.selected` property, you can evaluate every option in the `options` array to determine multiple selections, and you can select individual options without clearing the selection of other options.

Examples In the following example, the `getSelectedIndex` function returns the selected index in the `musicType` `Select` object:

```
function getSelectedIndex() {
   return document.musicForm.musicType.selectedIndex
}
```

The previous example assumes that the `Select` object is similar to the following:

```
<SELECT NAME="musicType">
   <OPTION SELECTED> R&B
   <OPTION> Jazz
   <OPTION> Blues
   <OPTION> New Age
</SELECT>
```

See also `Option.defaultSelected`, `Option.selected`

type

For all `Select` objects created with the `MULTIPLE` keyword, the value of the `type` property is `"select-multiple"`. For `Select` objects created without this keyword, the value of the `type` property is `"select-one"`. This property specifies the form element's type.

Property of	`Select`
Read-only	
Implemented in	JavaScript 1.1

Examples The following example writes the value of the `type` property for every element on a form.

```
for (var i = 0; i < document.form1.elements.length; i++) {
   document.writeln("<BR>type is " + document.form1.elements[i].type)
}
```

String

An object representing a series of characters in a string.

Core object

Implemented in	JavaScript 1.0: Create a `String` object only by quoting characters. JavaScript 1.1, NES 2.0: added `String` constructor; added `prototype` property; added `split` method; added ability to pass strings among scripts in different windows or frames (in previous releases, you had to add an empty string to another window's string to refer to it) JavaScript 1.2, NES 3.0: added `concat`, `match`, `replace`, `search`, `slice`, and `substr` methods. JavaScript 1.3: added `toSource` method; changed `charCodeAt`, `fromCharCode`, and `replace` methods
ECMA version	ECMA-262

Created by The `String` constructor:

```
new String(string)
```

Parameters

`string`	Any string.

Description The `String` object is a wrapper around the string primitive data type. Do not confuse a string literal with the `String` object. For example, the following code creates the string literal `s1` and also the `String` object `s2`:

```
s1 = "foo" // creates a string literal value
s2 = new String("foo") // creates a String object
```

You can call any of the methods of the `String` object on a string literal value—JavaScript automatically converts the string literal to a temporary `String` object, calls the method, then discards the temporary `String` object. You can also use the `String.length` property with a string literal.

You should use string literals unless you specifically need to use a `String` object, because `String` objects can have counterintuitive behavior. For example:

```
s1 = "2 + 2" // creates a string literal value
s2 = new String("2 + 2") // creates a String object
eval(s1)     // returns the number 4
eval(s2)     // returns the string "2 + 2"
```

A string can be represented as a literal enclosed by single or double quotation marks; for example, “Netscape” or ‘Netscape’.

You can convert the value of any object into a string using the top-level `String` function.

Property Summary

Property	Description
`constructor`	Specifies the function that creates an object’s prototype.
`length`	Reflects the length of the string.
`prototype`	Allows the addition of properties to a `String` object.

Method Summary

Method	Description
`anchor`	Creates an HTML anchor that is used as a hypertext target.
`big`	Causes a string to be displayed in a big font as if it were in a `BIG` tag.
`blink`	Causes a string to blink as if it were in a `BLINK` tag.
`bold`	Causes a string to be displayed as if it were in a `B` tag.
`charAt`	Returns the character at the specified `index`.
`charCodeAt`	Returns a number indicating the Unicode value of the character at the given index.
`concat`	Combines the text of two strings and returns a new string.
`fixed`	Causes a string to be displayed in fixed-pitch font as if it were in a `TT` tag.
`fontcolor`	Causes a string to be displayed in the specified color as if it were in a `<FONT COLOR=color>` tag.

Method	Description
`fontsize`	Causes a string to be displayed in the specified font size as if it were in a `<FONT SIZE=size>` tag.
`fromCharCode`	Returns a string created by using the specified sequence of Unicode values.
`indexOf`	Returns the index within the calling `String` object of the first occurrence of the specified value, or -1 if not found.
`italics`	Causes a string to be italic, as if it were in an `I` tag.
`lastIndexOf`	Returns the index within the calling `String` object of the last occurrence of the specified value, or -1 if not found.
`link`	Creates an HTML hypertext link that requests another URL.
`match`	Used to match a regular expression against a string.
`replace`	Used to find a match between a regular expression and a string, and to replace the matched substring with a new substring.
`search`	Executes the search for a match between a regular expression and a specified string.
`slice`	Extracts a section of a string and returns a new string.
`small`	Causes a string to be displayed in a small font, as if it were in a `SMALL` tag.
`split`	Splits a `String` object into an array of strings by separating the string into substrings.
`strike`	Causes a string to be displayed as struck-out text, as if it were in a `STRIKE` tag.
`sub`	Causes a string to be displayed as a subscript, as if it were in a `SUB` tag.
`substr`	Returns the characters in a string beginning at the specified location through the specified number of characters.
`substring`	Returns the characters in a string between two indexes into the string.
`sup`	Causes a string to be displayed as a superscript, as if it were in a `SUP` tag.
`toLowerCase`	Returns the calling string value converted to lowercase.

Method	Description
`toSource`	Returns an object literal representing the specified object; you can use this value to create a new object. Overrides the `Object.toSource` method.
`toString`	Returns a string representing the specified object. Overrides the `Object.toString` method.
`toUpperCase`	Returns the calling string value converted to uppercase.
`valueOf`	Returns the primitive value of the specified object. Overrides the `Object.valueOf` method.

In addition, this object inherits the `watch` and `unwatch` methods from `Object`.

Examples **Example 1: String literal.** The following statement creates a string literal:

```
var last_name = "Schaefer"
```

Example 2: String literal properties. The following statements evaluate to 8, `"SCHAEFER,"` and `"schaefer"`:

```
last_name.length
last_name.toUpperCase()
last_name.toLowerCase()
```

Example 3: Accessing individual characters in a string. You can think of a string as an array of characters. In this way, you can access the individual characters in the string by indexing that array. For example, the following code displays "The first character in the string is H":

```
var myString = "Hello"
myString[0] // returns "H"
```

Example 4: Pass a string among scripts in different windows or frames. The following code creates two string variables and opens a second window:

```
var lastName = "Schaefer"
var firstName = "Jesse"
empWindow=window.open('string2.html','window1','width=300,height=300')
```

If the HTML source for the second window (`string2.html`) creates two string variables, `empLastName` and `empFirstName`, the following code in the first window assigns values to the second window's variables:

```
empWindow.empFirstName=firstName
empWindow.empLastName=lastName
```

The following code in the first window displays the values of the second window's variables:

```
alert('empFirstName in empWindow is ' + empWindow.empFirstName)
alert('empLastName in empWindow is ' + empWindow.empLastName)
```

anchor

Creates an HTML anchor that is used as a hypertext target.

Method of	`String`
Implemented in	JavaScript 1.0, NES 2.0

Syntax `anchor(`*`nameAttribute`*`)`

Parameters

`nameAttribute`	A string.

Description Use the `anchor` method with the `document.write` or `document.writeln` methods to programmatically create and display an anchor in a document. Create the anchor with the `anchor` method, and then call `write` or `writeln` to display the anchor in a document. In server-side JavaScript, use the `write` function to display the anchor.

In the syntax, the `text` string represents the literal text that you want the user to see. The `nameAttribute` string represents the `NAME` attribute of the `A` tag.

Anchors created with the `anchor` method become elements in the `document.anchors` array.

Examples The following example opens the `msgWindow` window and creates an anchor for the table of contents:

```
var myString="Table of Contents"
msgWindow.document.writeln(myString.anchor("contents_anchor"))
```

The previous example produces the same output as the following HTML:

```
<A NAME="contents_anchor">Table of Contents</A>
```

See also `String.link`

big

Causes a string to be displayed in a big font as if it were in a `BIG` tag.

Method of	`String`
Implemented in	JavaScript 1.0, NES 2.0

Syntax `big()`

Parameters None

Description Use the `big` method with the `write` or `writeln` methods to format and display a string in a document. In server-side JavaScript, use the `write` function to display the string.

Examples The following example uses `string` methods to change the size of a string:

```
var worldString="Hello, world"

document.write(worldString.small())
document.write("<P>" + worldString.big())
document.write("<P>" + worldString.fontsize(7))
```

The previous example produces the same output as the following HTML:

```
<SMALL>Hello, world</SMALL>
<P><BIG>Hello, world</BIG>
<P><FONTSIZE=7>Hello, world</FONTSIZE>
```

See also `String.fontsize`, `String.small`

blink

Causes a string to blink as if it were in a `BLINK` tag.

Method of	`String`
Implemented in	JavaScript 1.0, NES 2.0

Syntax `blink()`

Parameters None

Description Use the `blink` method with the `write` or `writeln` methods to format and display a string in a document. In server-side JavaScript, use the `write` function to display the string.

Examples The following example uses `string` methods to change the formatting of a string:

```
var worldString="Hello, world"

document.write(worldString.blink())
document.write("<P>" + worldString.bold())
document.write("<P>" + worldString.italics())
document.write("<P>" + worldString.strike())
```

The previous example produces the same output as the following HTML:

```
<BLINK>Hello, world</BLINK>
<P><B>Hello, world</B>
<P><I>Hello, world</I>
<P><STRIKE>Hello, world</STRIKE>
```

See also `String.bold`, `String.italics`, `String.strike`

bold

Causes a string to be displayed as bold as if it were in a `B` tag.

Method of	`String`
Implemented in	JavaScript 1.0, NES 2.0

Syntax `bold()`

Parameters None

Description Use the `bold` method with the `write` or `writeln` methods to format and display a string in a document. In server-side JavaScript, use the `write` function to display the string.

Examples The following example uses `string` methods to change the formatting of a string:

```
var worldString="Hello, world"
document.write(worldString.blink())
document.write("<P>" + worldString.bold())
document.write("<P>" + worldString.italics())
document.write("<P>" + worldString.strike())
```

The previous example produces the same output as the following HTML:

```
<BLINK>Hello, world</BLINK>
<P><B>Hello, world</B>
<P><I>Hello, world</I>
<P><STRIKE>Hello, world</STRIKE>
```

See also `String.blink`, `String.italics`, `String.strike`

charAt

Returns the specified character from the string.

Method of	`String`
Implemented in	JavaScript 1.0, NES 2.0
ECMA version	ECMA-262

Syntax `charAt(`*`index`*`)`

Parameters

`index`	An integer between 0 and 1 less than the length of the string.

Description Characters in a string are indexed from left to right. The index of the first character is 0, and the index of the last character in a string called `stringName` is `stringName.length - 1`. If the `index` you supply is out of range, JavaScript returns an empty string.

Examples The following example displays characters at different locations in the string `"Brave new world"`:

```
var anyString="Brave new world"

document.writeln("The character at index 0 is " + anyString.charAt(0))
document.writeln("The character at index 1 is " + anyString.charAt(1))
document.writeln("The character at index 2 is " + anyString.charAt(2))
document.writeln("The character at index 3 is " + anyString.charAt(3))
document.writeln("The character at index 4 is " + anyString.charAt(4))
```

These lines display the following:

```
The character at index 0 is B
The character at index 1 is r
The character at index 2 is a
The character at index 3 is v
The character at index 4 is e
```

See also `String.indexOf`, `String.lastIndexOf`, `String.split`

charCodeAt

Returns a number indicating the Unicode value of the character at the given index.

Method of	`String`
Implemented in	JavaScript 1.2, NES 3.0 JavaScript 1.3: returns a Unicode value rather than an ISO-Latin-1 value
ECMA version	ECMA-262

Syntax `charCodeAt([`*`index`*`])`

Parameters

`index`	An integer between 0 and 1 less than the length of the string. The default value is 0.

Description Unicode values range from 0 to 65,535. The first 128 Unicode values are a direct match of the ASCII character set. For information on Unicode, see the *Client-Side JavaScript Guide*.

Backward Compatibility **JavaScript 1.2.** The `charCodeAt` method returns a number indicating the ISO-Latin-1 codeset value of the character at the given index. The ISO-Latin-1 codeset ranges from 0 to 255. The first 0 to 127 are a direct match of the ASCII character set.

Example **Example 1.** The following example returns 65, the Unicode value for A.

```
"ABC".charCodeAt(0) // returns 65
```

Example 2. The following example enables the creation of an event used to simulate a key press. A KeyPress event has a `which` property that represents the ASCII value of the pressed key. If you know the letter, number, or symbol, you can use `charCodeAt` to supply the ASCII value to `which`.

```
//create an event object with appropriate property values
ev = new Event()
ev.type = KeyPress
ev.layerX = 150
//assign values to layerY, pageX, pageY, screenX, and screenY
...
//assign the ASCII value to the which property
ev.which = "v".charCodeAt(0)
//assign modifier property
ev.modifiers = <FONT COLOR="#FF0080">How do I do this?</FONT>
```

concat

Combines the text of two or more strings and returns a new string.

Method of	`String`
Implemented in	JavaScript 1.2, NES 3.0

Syntax `concat(`*`string2`*`, `*`string3`*`[, ..., `*`stringN`*`])`

Parameters

`string2...`
`string`*`N`*
	Strings to concatenate to this string.

Description `concat` combines the text from two strings and returns a new string. Changes to the text in one string do not affect the other string.

Example The following example combines two strings into a new string.

```
s1="Oh "
s2="what a beautiful "
s3="mornin'."
s4=s1.concat(s2,s3) // returns "Oh what a beautiful mornin'."
```

constructor

Specifies the function that creates an object's prototype. Note that the value of this property is a reference to the function itself, not a string containing the function's name.

Property of	`String`
Implemented in	JavaScript 1.1, NES 2.0
ECMA version	ECMA-262

Description See `Object.constructor`.

fixed

Causes a string to be displayed in fixed-pitch font as if it were in a `TT` tag.

Method of	`String`
Implemented in	JavaScript 1.0, NES 2.0

Syntax `fixed()`

Parameters None

Description Use the `fixed` method with the `write` or `writeln` methods to format and display a string in a document. In server-side JavaScript, use the `write` function to display the string.

Examples The following example uses the `fixed` method to change the formatting of a string:

```
var worldString="Hello, world"
document.write(worldString.fixed())
```

The previous example produces the same output as the following HTML:

```
<TT>Hello, world</TT>
```

fontcolor

Causes a string to be displayed in the specified color as if it were in a `<FONT COLOR=color>` tag.

Method of	`String`
Implemented in	JavaScript 1.0, NES 2.0

Syntax `fontcolor(`*`color`*`)`

Parameters

`color`	A string expressing the color as a hexadecimal RGB triplet or as a string literal. String literals for color names are listed in the *Client-Side JavaScript Guide*.

Description Use the `fontcolor` method with the `write` or `writeln` methods to format and display a string in a document. In server-side JavaScript, use the `write` function to display the string.

If you express `color` as a hexadecimal RGB triplet, you must use the format `rrggbb`. For example, the hexadecimal RGB values for salmon are `red=FA`, `green=80`, and `blue=72`, so the RGB triplet for `salmon` is `"FA8072"`.

The `fontcolor` method overrides a value set in the `fgColor` property.

Examples The following example uses the `fontcolor` method to change the color of a string:

```
var worldString="Hello, world"

document.write(worldString.fontcolor("maroon") +
   " is maroon in this line")
document.write("<P>" + worldString.fontcolor("salmon") +
   " is salmon in this line")
document.write("<P>" + worldString.fontcolor("red") +
   " is red in this line")

document.write("<P>" + worldString.fontcolor("8000") +
   " is maroon in hexadecimal in this line")
document.write("<P>" + worldString.fontcolor("FA8072") +
   " is salmon in hexadecimal in this line")
document.write("<P>" + worldString.fontcolor("FF00") +
   " is red in hexadecimal in this line")
```

The previous example produces the same output as the following HTML:

```
<FONT COLOR="maroon">Hello, world</FONT> is maroon in this line
<P><FONT COLOR="salmon">Hello, world</FONT> is salmon in this line
<P><FONT COLOR="red">Hello, world</FONT> is red in this line
```

```
<FONT COLOR="8000">Hello, world</FONT>
is maroon in hexadecimal in this line
<P><FONT COLOR="FA8072">Hello, world</FONT>
is salmon in hexadecimal in this line
<P><FONT COLOR="FF00">Hello, world</FONT>
is red in hexadecimal in this line
```

fontsize

Causes a string to be displayed in the specified font size as if it were in a `<FONT SIZE=size>` tag.

Method of	`String`
Implemented in	JavaScript 1.0, NES 2.0

Syntax `fontsize(`*`size`*`)`

Parameters

`size`	An integer between 1 and 7, a string representing a signed integer between 1 and 7.

Description Use the `fontsize` method with the `write` or `writeln` methods to format and display a string in a document. In server-side JavaScript, use the `write` function to display the string.

When you specify `size` as an integer, you set the size of `stringName` to one of the 7 defined sizes. When you specify `size` as a string such as `"-2"`, you adjust the font size of `stringName` relative to the size set in the `BASEFONT` tag.

Examples The following example uses `string` methods to change the size of a string:

```
var worldString="Hello, world"

document.write(worldString.small())
document.write("<P>" + worldString.big())
document.write("<P>" + worldString.fontsize(7))
```

The previous example produces the same output as the following HTML:

```
<SMALL>Hello, world</SMALL>
<P><BIG>Hello, world</BIG>
<P><FONTSIZE=7>Hello, world</FONTSIZE>
```

See also `String.big`, `String.small`

fromCharCode

Returns a string created by using the specified sequence of Unicode values.

Method of	`String`
Static	
Implemented in	JavaScript 1.2, NES 3.0 JavaScript 1.3: uses a Unicode value rather than an ISO-Latin-1 value
ECMA version	ECMA-262

Syntax `fromCharCode(`*`num1`*`, ..., `*`numN`*`)`

Parameters

`num1, ..., numN`	A sequence of numbers that are Unicode values.

Description This method returns a string and not a `String` object.

Because `fromCharCode` is a static method of `String`, you always use it as `String.fromCharCode()`, rather than as a method of a `String` object you created.

Backward Compatibility **JavaScript 1.2.** The `fromCharCode` method returns a string created by using the specified sequence of ISO-Latin-1 codeset values.

Examples **Example 1**. The following example returns the string "ABC".

```
String.fromCharCode(65,66,67)
```

Example 2. The `which` property of the `KeyDown`, `KeyPress`, and `KeyUp` events contains the ASCII value of the key pressed at the time the event occurred. If you want to get the actual letter, number, or symbol of the key, you can use `fromCharCode`. The following example returns the letter, number, or symbol of the KeyPress event's `which` property.

```
String.fromCharCode(KeyPress.which)
```

indexOf

Returns the index within the calling `String` object of the first occurrence of the specified value, starting the search at `fromIndex`, or -1 if the value is not found.

Method of	`String`
Implemented in	JavaScript 1.0, NES 2.0
ECMA version	ECMA-262

Syntax `indexOf(`*`searchValue`*`[, `*`fromIndex`*`])`

Parameters

`searchValue`	A string representing the value to search for.
`fromIndex`	The location within the calling string to start the search from. It can be any integer between 0 and the length of the string. The default value is 0.

Description Characters in a string are indexed from left to right. The index of the first character is 0, and the index of the last character of a string called `stringName` is `stringName.length - 1`.

```
"Blue Whale".indexOf("Blue")    // returns 0
"Blue Whale".indexOf("Blute")   // returns -1
"Blue Whale".indexOf("Whale",0) // returns 5
"Blue Whale".indexOf("Whale",5) // returns 5
"Blue Whale".indexOf("",9)      // returns 9
"Blue Whale".indexOf("",10)     // returns 10
"Blue Whale".indexOf("",11)     // returns 10
```

The `indexOf` method is case sensitive. For example, the following expression returns -1:

```
"Blue Whale".indexOf("blue")
```

Examples

Example 1. The following example uses `indexOf` and `lastIndexOf` to locate values in the string `"Brave new world."`

```
var anyString="Brave new world"

// Displays 8
document.write("<P>The index of the first w from the beginning is " +
   anyString.indexOf("w"))
// Displays 10
document.write("<P>The index of the first w from the end is " +
   anyString.lastIndexOf("w"))
// Displays 6
document.write("<P>The index of 'new' from the beginning is " +
   anyString.indexOf("new"))
// Displays 6
document.write("<P>The index of 'new' from the end is " +
   anyString.lastIndexOf("new"))
```

Example 2. The following example defines two string variables. The variables contain the same string except that the second string contains uppercase letters. The first `writeln` method displays 19. But because the `indexOf` method is case sensitive, the string `"cheddar"` is not found in `myCapString`, so the second `writeln` method displays -1.

```
myString="brie, pepper jack, cheddar"
myCapString="Brie, Pepper Jack, Cheddar"
document.writeln('myString.indexOf("cheddar") is ' +
   myString.indexOf("cheddar"))
document.writeln('<P>myCapString.indexOf("cheddar") is ' +
   myCapString.indexOf("cheddar"))
```

Example 3. The following example sets `count` to the number of occurrences of the letter `x` in the string `str`:

```
count = 0;
pos = str.indexOf("x");
while ( pos != -1 ) {
   count++;
   pos = str.indexOf("x",pos+1);
}
```

See also `String.charAt`, `String.lastIndexOf`, `String.split`

italics

Causes a string to be italic, as if it were in an `<I>` tag.

Method of	`String`
Implemented in	JavaScript 1.0, NES 2.0

Syntax `italics()`

Parameters None

Description Use the `italics` method with the `write` or `writeln` methods to format and display a string in a document. In server-side JavaScript, use the `write` function to display the string.

Examples The following example uses `string` methods to change the formatting of a string:

```
var worldString="Hello, world"

document.write(worldString.blink())
document.write("<P>" + worldString.bold())
document.write("<P>" + worldString.italics())
document.write("<P>" + worldString.strike())
```

The previous example produces the same output as the following HTML:

```
<BLINK>Hello, world</BLINK>
<P><B>Hello, world</B>
<P><I>Hello, world</I>
<P><STRIKE>Hello, world</STRIKE>
```

See also `String.blink`, `String.bold`, `String.strike`

lastIndexOf

Returns the index within the calling `String` object of the last occurrence of the specified value, or -1 if not found. The calling string is searched backward, starting at `fromIndex`.

Method of	`String`
Implemented in	JavaScript 1.0, NES 2.0
ECMA version	ECMA-262

Syntax `lastIndexOf(`*`searchValue`*`[, `*`fromIndex`*`])`

Parameters

`searchValue`	A string representing the value to search for.
`fromIndex`	The location within the calling string to start the search from. It can be any integer between 0 and the length of the string. The default value is the length of the string.

Description Characters in a string are indexed from left to right. The index of the first character is 0, and the index of the last character is `stringName`.length - 1.

```
"canal".lastIndexOf("a")   // returns 3
"canal".lastIndexOf("a",2) // returns 1
"canal".lastIndexOf("a",0) // returns -1
"canal".lastIndexOf("x")   // returns -1
```

The `lastIndexOf` method is case sensitive. For example, the following expression returns -1:

```
"Blue Whale, Killer Whale".lastIndexOf("blue")
```

Examples The following example uses `indexOf` and `lastIndexOf` to locate values in the string `"Brave new world."`

```
var anyString="Brave new world"

// Displays 8
document.write("<P>The index of the first w from the beginning is " +
   anyString.indexOf("w"))
// Displays 10
document.write("<P>The index of the first w from the end is " +
   anyString.lastIndexOf("w"))
// Displays 6
document.write("<P>The index of 'new' from the beginning is " +
   anyString.indexOf("new"))
// Displays 6
document.write("<P>The index of 'new' from the end is " +
   anyString.lastIndexOf("new"))
```

See also `String.charAt`, `String.indexOf`, `String.split`

length

The length of the string.

Property of	`String`
Read-only	
Implemented in	JavaScript 1.0, NES 2.0
ECMA version	ECMA-262

Description For a null string, length is 0.

Examples The following example displays 8 in an Alert dialog box:

```
var x="Netscape"
alert("The string length is " + x.length)
```

link

Creates an HTML hypertext link that requests another URL.

Method of	`String`
Implemented in	JavaScript 1.0, NES 2.0

Syntax `link(`*`hrefAttribute`*`)`

Parameters

`hrefAttribute`	Any string that specifies the `HREF` attribute of the `A` tag; it should be a valid URL (relative or absolute).

Description Use the `link` method to programmatically create a hypertext link, and then call `write` or `writeln` to display the link in a document. In server-side JavaScript, use the `write` function to display the link.

Links created with the `link` method become elements in the `links` array of the `document` object. See `document.links`.

Examples The following example displays the word “Netscape” as a hypertext link that returns the user to the Netscape home page:

```
var hotText="Netscape"
var URL="http://home.netscape.com"

document.write("Click to return to " + hotText.link(URL))
```

The previous example produces the same output as the following HTML:

```
Click to return to <A HREF="http://home.netscape.com">Netscape</A>
```

See also `Anchor`

match

Used to match a regular expression against a string.

Method of	`String`
Implemented in	JavaScript 1.2

Syntax `match(`*`regexp`*`)`

Parameters

`regexp`	Name of the regular expression. It can be a variable name or a literal.

Description If you want to execute a global match, or a case insensitive match, include the `g` (for global) and `i` (for ignore case) flags in the regular expression. These can be included separately or together. The following two examples below show how to use these flags with `match`.

Note If you execute a match simply to find true or false, use `String.search` or the regular expression `test` method.

Examples **Example 1**. In the following example, `match` is used to find 'Chapter' followed by 1 or more numeric characters followed by a decimal point and numeric character 0 or more times. The regular expression includes the `i` flag so that case will be ignored.

```
<SCRIPT>
str = "For more information, see Chapter 3.4.5.1";
re = /(chapter \d+(\.\d)*)/i;
found = str.match(re);
document.write(found);
</SCRIPT>
```

This returns the array containing Chapter 3.4.5.1,Chapter 3.4.5.1,.1

`'Chapter 3.4.5.1'` is the first match and the first value remembered from `(Chapter \d+(\.\d)*)`.

`'.1'` is the second value remembered from `(\.\d)`.

Example 2. The following example demonstrates the use of the global and ignore case flags with `match`.

```
<SCRIPT>
str = "abcDdcba";
newArray = str.match(/d/gi);
document.write(newArray);
</SCRIPT>
```

The returned array contains D, d.

prototype

Represents the prototype for this class. You can use the prototype to add properties or methods to all instances of a class. For information on prototypes, see `Function.prototype`.

Property of	`String`
Implemented in	JavaScript 1.1, NES 3.0
ECMA version	ECMA-262

replace

Finds a match between a regular expression and a string, and replaces the matched substring with a new substring.

Method of	`String`
Implemented in	JavaScript 1.2 JavaScript 1.3: supports the nesting of a function in place of the second argument

Syntax

```
replace(regexp, newSubStr)
replace(regexp, function)
```

Versions prior to JavaScript 1.3:

```
replace(regexp, newSubStr)
```

Parameters

`regexp`	The name of the regular expression. It can be a variable name or a literal.
`newSubStr`	The string to put in place of the string found with `regexp`. This string can include the `RegExp` properties `$1, ..., $9`, `lastMatch`, `lastParen`, `leftContext`, and `rightContext`.
`function`	A function to be invoked after the match has been performed.

Description

This method does not change the `String` object it is called on; it simply returns a new string.

If you want to execute a global search and replace, or a case insensitive search, include the `g` (for global) and `i` (for ignore case) flags in the regular expression. These can be included separately or together. The following two examples below show how to use these flags with `replace`.

Specifying a function as a parameter. When you specify a function as the second parameter, the function is invoked after the match has been performed. (The use of a function in this manner is often called a lambda expression.)

In your function, you can dynamically generate the string that replaces the matched substring. The result of the function call is used as the replacement value.

The nested function can use the matched substrings to determine the new string (`newSubStr`) that replaces the found substring. You get the matched substrings through the parameters of your function. The first parameter of your function holds the complete matched substring. Other parameters can be used for parenthetical matches, remembered submatch strings. For example, the following `replace` method returns XX.zzzz - XX , zzzz.

```
"XXzzzz".replace(/(X*)(z*)/,
                   function (str, p1, p2) {
                      return str + " - " + p1 + " , " + p2;
                   }
                )
```

The array returned from the `exec` method of the `RegExp` object and the subsequent match is available to your function. You can use the content of the array plus the `input` and the `index` (index of match in the input string) properties of the array to perform additional tasks before the method replaces the substring.

Examples

Example 1. In the following example, the regular expression includes the global and ignore case flags which permits `replace` to replace each occurrence of 'apples' in the string with 'oranges.'

```
<SCRIPT>
re = /apples/gi;
str = "Apples are round, and apples are juicy.";
newstr=str.replace(re, "oranges");
document.write(newstr)
</SCRIPT>
```

This prints "oranges are round, and oranges are juicy."

Example 2. In the following example, the regular expression is defined in `replace` and includes the ignore case flag.

```
<SCRIPT>
str = "Twas the night before Xmas...";
newstr=str.replace(/xmas/i, "Christmas");
document.write(newstr)
</SCRIPT>
```

This prints "Twas the night before Christmas..."

Example 3. The following script switches the words in the string. For the replacement text, the script uses the values of the `$1` and `$2` properties.

```
<SCRIPT LANGUAGE="JavaScript1.2">
re = /(\w+)\s(\w+)/;
str = "John Smith";
newstr = str.replace(re, "$2, $1");
document.write(newstr)
</SCRIPT>
```

This prints "Smith, John".

Example 4. The following example replaces a Fahrenheit degree with its equivalent Celsius degree. The Fahrenheit degree should be a number ending with F. The function returns the Celsius number ending with C. For example, if the input number is 212F, the function returns 100C. If the number is 0F, the function returns -17.77777777777778C.

The regular expression `test` checks for any number that ends with F. The number of Fahrenheit degree is accessible to your function through the parameter `$1`. The function sets the Celsius number based on the Fahrenheit degree passed in a string to the `f2c` function. `f2c` then returns the Celsius number. This function approximates Perl's s///e flag.

```
function f2c(x) {
   var s = String(x)
   var test = /(\d+(\.\d*)?)F\b/g
   return s.replace
      (test,
         myfunction ($0,$1,$2) {
            return (($1-32) * 5/9) + "C";
         }
      )
}
```

search

Executes the search for a match between a regular expression and this `String` object.

Method of	`String`
Implemented in	JavaScript 1.2

Syntax `search(`*`regexp`*`)`

Parameters

`regexp`	Name of the regular expression. It can be a variable name or a literal.

Description If successful, `search` returns the index of the regular expression inside the string. Otherwise, it returns -1.

When you want to know whether a pattern is found in a string use `search` (similar to the regular expression `test` method); for more information (but slower execution) use `match` (similar to the regular expression `exec` method).

Example The following example prints a message which depends on the success of the test.

```
function testinput(re, str){
   if (str.search(re) != -1)
      midstring = " contains ";
   else
      midstring = " does not contain ";
   document.write (str + midstring + re.source);
}
```

slice

Extracts a section of a string and returns a new string.

Method of	`String`
Implemented in	JavaScript 1.0, NES 2.0

Syntax `slice(`*`beginslice`*`[, `*`endSlice`*`])`

Parameters

`beginSlice`	The zero-based index at which to begin extraction.
`endSlice`	The zero-based index at which to end extraction. If omitted, `slice` extracts to the end of the string.

Description `slice` extracts the text from one string and returns a new string. Changes to the text in one string do not affect the other string.

`slice` extracts up to but not including `endSlice`. `string.slice(1,4)` extracts the second character through the fourth character (characters indexed 1, 2, and 3).

As a negative index, `endSlice` indicates an offset from the end of the string. `string.slice(2,-1)` extracts the third character through the second to last character in the string.

Example The following example uses `slice` to create a new string.

```
<SCRIPT>
str1="The morning is upon us. "
str2=str1.slice(3,-5)
document.write(str2)
</SCRIPT>
```

This writes:

morning is upon

small

Causes a string to be displayed in a small font, as if it were in a `<SMALL>` tag.

Method of	`String`
Implemented in	JavaScript 1.0, NES 2.0

Syntax `small()`

Parameters None

Description Use the `small` method with the `write` or `writeln` methods to format and display a string in a document. In server-side JavaScript, use the `write` function to display the string.

Examples The following example uses `string` methods to change the size of a string:

```
var worldString="Hello, world"

document.write(worldString.small())
document.write("<P>" + worldString.big())
document.write("<P>" + worldString.fontsize(7))
```

The previous example produces the same output as the following HTML:

```
<SMALL>Hello, world</SMALL>
<P><BIG>Hello, world</BIG>
<P><FONTSIZE=7>Hello, world</FONTSIZE>
```

See also `String.big`, `String.fontsize`

split

Splits a `String` object into an array of strings by separating the string into substrings.

Method of	`String`
Implemented in	JavaScript 1.1, NES 2.0
ECMA version	ECMA-262

Syntax `split([`*`separator`*`][, `*`limit`*`])`

Parameters

`separator`	Specifies the character to use for separating the string. The `separator` is treated as a string. If `separator` is omitted, the array returned contains one element consisting of the entire string.
`limit`	Integer specifying a limit on the number of splits to be found.

Description The `split` method returns the new array.

When found, `separator` is removed from the string and the substrings are returned in an array. If `separator` is omitted, the array contains one element consisting of the entire string.

In JavaScript 1.2, `split` has the following additions:

- It can take a regular expression argument, as well as a fixed string, by which to split the object string. If `separator` is a regular expression, any included parenthesis cause submatches to be included in the returned array.
- It can take a limit count so that the resulting array does not include trailing empty elements.
- If you specify `LANGUAGE="JavaScript1.2"` in the `SCRIPT` tag, `string.split(" ")` splits on any run of 1 or more white space characters including spaces, tabs, line feeds, and carriage returns. For this behavior, `LANGUAGE="JavaScript1.2"` must be specified in the `<SCRIPT>` tag.

Examples **Example 1**. The following example defines a function that splits a string into an array of strings using the specified separator. After splitting the string, the function displays messages indicating the original string (before the split), the separator used, the number of elements in the array, and the individual array elements.

```
function splitString (stringToSplit,separator) {
   arrayOfStrings = stringToSplit.split(separator)
   document.write ('<P>The original string is: "' + stringToSplit + '"')
   document.write ('<BR>The separator is: "' + separator + '"')
   document.write ("<BR>The array has " + arrayOfStrings.length + " elements: ")

   for (var i=0; i < arrayOfStrings.length; i++) {
      document.write (arrayOfStrings[i] + " / ")
   }
}
```

```
var tempestString="Oh brave new world that has such people in it."
var monthString="Jan,Feb,Mar,Apr,May,Jun,Jul,Aug,Sep,Oct,Nov,Dec"

var space=" "
var comma=","

splitString(tempestString,space)
splitString(tempestString)
splitString(monthString,comma)
```

This example produces the following output:

```
The original string is: "Oh brave new world that has such people in it."
The separator is: " "
The array has 10 elements: Oh / brave / new / world / that / has / such / people / in / it.
/

The original string is: "Oh brave new world that has such people in it."
The separator is: "undefined"
The array has 1 elements: Oh brave new world that has such people in it. /

The original string is: "Jan,Feb,Mar,Apr,May,Jun,Jul,Aug,Sep,Oct,Nov,Dec"
The separator is: ","
The array has 12 elements: Jan / Feb / Mar / Apr / May / Jun / Jul / Aug / Sep / Oct / Nov
/ Dec /
```

Example 2. Consider the following script:

```
<SCRIPT LANGUAGE="JavaScript1.2">
str="She sells    seashells \nby   the\n seashore"
document.write(str + "<BR>")
a=str.split(" ")
document.write(a)
</SCRIPT>
```

Using `LANGUAGE="JavaScript1.2"`, this script produces

```
"She", "sells", "seashells", "by", "the", "seashore"
```

Without `LANGUAGE="JavaScript1.2"`, this script splits only on single space characters, producing

```
"She", "sells", , , , "seashells", "by", , , "the", "seashore"
```

Example 3. In the following example, `split` looks for 0 or more spaces followed by a semicolon followed by 0 or more spaces and, when found, removes the spaces from the string. `nameList` is the array returned as a result of `split`.

```
<SCRIPT>
names = "Harry  Trump  ;Fred Barney; Helen   Rigby ;  Bill Abel ;Chris Hand ";
document.write (names + "<BR>" + "<BR>");
re = /\s*;\s*/;
nameList = names.split (re);
document.write(nameList);
</SCRIPT>
```

This prints two lines; the first line prints the original string, and the second line prints the resulting array.

Harry Trump ;Fred Barney; Helen Rigby ; Bill Abel ;Chris Hand
Harry Trump,Fred Barney,Helen Rigby,Bill Abel,Chris Hand

Example 4. In the following example, `split` looks for 0 or more spaces in a string and returns the first 3 splits that it finds.

```
<SCRIPT LANGUAGE="JavaScript1.2">
myVar = "  Hello World. How are you doing?    ";
splits = myVar.split(" ", 3);
document.write(splits)
</SCRIPT>
```

This script displays the following:

```
["Hello", "World.", "How"]
```

See also `String.charAt`, `String.indexOf`, `String.lastIndexOf`

strike

Causes a string to be displayed as struck-out text, as if it were in a `<STRIKE>` tag.

Method of	`String`
Implemented in	JavaScript 1.0, NES 2.0

Syntax `strike()`

Parameters None

Description Use the `strike` method with the `write` or `writeln` methods to format and display a string in a document. In server-side JavaScript, use the `write` function to display the string.

Examples The following example uses `string` methods to change the formatting of a string:

```
var worldString="Hello, world"

document.write(worldString.blink())
document.write("<P>" + worldString.bold())
document.write("<P>" + worldString.italics())
document.write("<P>" + worldString.strike())
```

The previous example produces the same output as the following HTML:

```
<BLINK>Hello, world</BLINK>
<P><B>Hello, world</B>
<P><I>Hello, world</I>
<P><STRIKE>Hello, world</STRIKE>
```

See also `String.blink`, `String.bold`, `String.italics`

sub

Causes a string to be displayed as a subscript, as if it were in a `<SUB>` tag.

Method of	`String`
Implemented in	JavaScript 1.0, NES 2.0

Syntax `sub()`

Parameters None

Description Use the `sub` method with the `write` or `writeln` methods to format and display a string in a document. In server-side JavaScript, use the `write` function to generate the HTML.

Examples The following example uses the `sub` and `sup` methods to format a string:

```
var superText="superscript"
var subText="subscript"

document.write("This is what a " + superText.sup() + " looks like.")
document.write("<P>This is what a " + subText.sub() + " looks like.")
```

The previous example produces the same output as the following HTML:

```
This is what a <SUP>superscript</SUP> looks like.
<P>This is what a <SUB>subscript</SUB> looks like.
```

See also `String.sup`

substr

Returns the characters in a string beginning at the specified location through the specified number of characters.

Method of	`String`
Implemented in	JavaScript 1.0, NES 2.0

Syntax `substr(`*`start`*`[, `*`length`*`])`

Parameters

`start`	Location at which to begin extracting characters.
`length`	The number of characters to extract

Description `start` is a character index. The index of the first character is 0, and the index of the last character is 1 less than the length of the string. `substr` begins extracting characters at `start` and collects `length` number of characters.

If `start` is positive and is the length of the string or longer, `substr` returns no characters.

If `start` is negative, `substr` uses it as a character index from the end of the string. If `start` is negative and `abs(start)` is larger than the length of the string, `substr` uses 0 is the start index.

If `length` is 0 or negative, `substr` returns no characters. If `length` is omitted, `start` extracts characters to the end of the string.

Example Consider the following script:

```
<SCRIPT LANGUAGE="JavaScript1.2">

str = "abcdefghij"
document.writeln("(1,2): ", str.substr(1,2))
document.writeln("(-2,2): ", str.substr(-2,2))
document.writeln("(1): ", str.substr(1))
document.writeln("(-20, 2): ", str.substr(1,20))
document.writeln("(20, 2): ", str.substr(20,2))

</SCRIPT>
```

This script displays:

```
(1,2): bc
(-2,2): ij
(1): bcdefghij
(-20, 2): bcdefghij
(20, 2):
```

See also `substring`

substring

Returns a subset of a `String` object.

Method of	`String`
Implemented in	JavaScript 1.0, NES 2.0
ECMA version	ECMA-262

Syntax `substring(`*`indexA`*`, `*`indexB`*`)`

Parameters

`indexA`	An integer between 0 and 1 less than the length of the string.
`indexB`	An integer between 0 and 1 less than the length of the string.

Description `substring` extracts characters from `indexA` up to but not including `indexB`. In particular:

- If `indexA` is less than 0, `indexA` is treated as if it were 0.
- If `indexB` is greater than `stringName.length`, `indexB` is treated as if it were `stringName.length`.
- If `indexA` equals `indexB`, `substring` returns an empty string.
- If `indexB` is omitted, `indexA` extracts characters to the end of the string.

In JavaScript 1.2, using `LANGUAGE="JavaScript1.2"` in the `SCRIPT` tag,

- If `indexA` is greater than `indexB`, JavaScript produces a runtime error (out of memory).

In JavaScript 1.2, without `LANGUAGE="JavaScript1.2"` in the `SCRIPT` tag,

- If `indexA` is greater than `indexB`, JavaScript returns a substring beginning with `indexB` and ending with `indexA - 1`.

Examples **Example 1.** The following example uses `substring` to display characters from the string `"Netscape"`:

```
var anyString="Netscape"

// Displays "Net"
document.write(anyString.substring(0,3))
document.write(anyString.substring(3,0))
// Displays "cap"
document.write(anyString.substring(4,7))
document.write(anyString.substring(7,4))
// Displays "Netscap"
document.write(anyString.substring(0,7))
// Displays "Netscape"
document.write(anyString.substring(0,8))
document.write(anyString.substring(0,10))
```

Example 2. The following example replaces a substring within a string. It will replace both individual characters and substrings. The function call at the end of the example changes the string `"Brave New World"` into `"Brave New Web"`.

```
function replaceString(oldS,newS,fullS) {
// Replaces oldS with newS in the string fullS
   for (var i=0; i<fullS.length; i++) {
      if (fullS.substring(i,i+oldS.length) == oldS) {
         fullS = fullS.substring(0,i)+newS+fullS.substring(i+oldS.length,fullS.length)
      }
   }
   return fullS
}

replaceString("World","Web","Brave New World")
```

Example 3. In JavaScript 1.2, using `LANGUAGE="JavaScript1.2"`, the following script produces a runtime error (out of memory).

```
<SCRIPT LANGUAGE="JavaScript1.2">
str="Netscape"
document.write(str.substring(0,3);
document.write(str.substring(3,0);
</SCRIPT>
```

Without `LANGUAGE="JavaScript1.2"`, the above script prints the following:

Net Net

In the second `write`, the index numbers are swapped.

See also `substr`

sup

Causes a string to be displayed as a superscript, as if it were in a `<SUP>` tag.

Method of	`String`
Implemented in	JavaScript 1.0, NES 2.0

Syntax `sup()`

Parameters None

Description Use the `sup` method with the `write` or `writeln` methods to format and display a string in a document. In server-side JavaScript, use the `write` function to generate the HTML.

Examples The following example uses the `sub` and `sup` methods to format a string:

```
var superText="superscript"
var subText="subscript"

document.write("This is what a " + superText.sup() + " looks like.")
document.write("<P>This is what a " + subText.sub() + " looks like.")
```

The previous example produces the same output as the following HTML:

```
This is what a <SUP>superscript</SUP> looks like.
<P>This is what a <SUB>subscript</SUB> looks like.
```

See also `String.sub`

toLowerCase

Returns the calling string value converted to lowercase.

Method of	`String`
Implemented in	JavaScript 1.0, NES 2.0
ECMA version	ECMA-262

Syntax `toLowerCase()`

Parameters None

Description The `toLowerCase` method returns the value of the string converted to lowercase. `toLowerCase` does not affect the value of the string itself.

Examples The following example displays the lowercase string `"alphabet"`:

```
var upperText="ALPHABET"
document.write(upperText.toLowerCase())
```

See also `String.toUpperCase`

toSource

Returns a string representing the source code of the object.

Method of	`String`
Implemented in	JavaScript 1.3

Syntax `toSource()`

Parameters None

Description The `toSource` method returns the following values:

- For the built-in `String` object, `toSource` returns the following string indicating that the source code is not available:

```
function String() {
   [native code]
}
```

- For instances of `String` or string literals, `toSource` returns a string representing the source code.

This method is usually called internally by JavaScript and not explicitly in code.

toString

Returns a string representing the specified object.

Method of	`String`
Implemented in	JavaScript 1.1, NES 2.0
ECMA version	ECMA-262

Syntax `toString()`

Parameters None.

Description The `String` object overrides the `toString` method of the `Object` object; it does not inherit `Object.toString`. For `String` objects, the `toString` method returns a string representation of the object.

Examples The following example displays the string value of a String object:

```
x = new String("Hello world");
alert(x.toString())      // Displays "Hello world"
```

See also `Object.toString`

toUpperCase

Returns the calling string value converted to uppercase.

Method of	`String`
Implemented in	JavaScript 1.0, NES 2.0
ECMA version	ECMA-262

Syntax `toUpperCase()`

Parameters None

Description The `toUpperCase` method returns the value of the string converted to uppercase. `toUpperCase` does not affect the value of the string itself.

Examples The following example displays the string `"ALPHABET"`:

```
var lowerText="alphabet"
document.write(lowerText.toUpperCase())
```

See also `String.toLowerCase`

valueOf

Returns the primitive value of a String object.

Method of	`String`
Implemented in	JavaScript 1.1
ECMA version	ECMA-262

Syntax `valueOf()`

Parameters None

Description The `valueOf` method of `String` returns the primitive value of a String object as a string data type. This value is equivalent to `String.toString`.

This method is usually called internally by JavaScript and not explicitly in code.

Examples

```
x = new String("Hello world");
alert(x.valueOf())          // Displays "Hello world"
```

See also `String.toString`, `Object.valueOf`

Style

An object that specifies the style of HTML elements.

Client-side object

Implemented in JavaScript 1.2

Created by Any of the following properties or methods of the `document` object:

- `document.classes`
- `document.contextual`
- `document.ids`
- `document.tags`

Description The `Style` object lets you implement dynamic HTML style sheets in JavaScript. The methods and properties of the Style object implement the cascading style sheet style properties of HTML in JavaScript.

For a complete description of style sheets, see *Dynamic HTML in Netscape Communicator*.

Property Summary

Property	Description
`align`	Specifies the alignment of an HTML element within its parent.
`backgroundColor`	Specifies a solid background color for an element.
`backgroundImage`	Specifies a background image for an HTML element.
`borderBottomWidth`	Specifies the width of the bottom border of an HTML element.
`borderColor`	Specifies the color of the border of an HTML element.
`borderLeftWidth`	Specifies the width of the left border of an HTML element.
`borderRightWidth`	Specifies the width of the right border of an HTML element.
`borderStyle`	Specifies the style of border, such as solid or double, around a block-level HTML element.
`borderTopWidth`	Specifies the width of the top border of an HTML element.
`clear`	Specifies the sides of an HTML element that allow floating elements.
`color`	Specifies the color of the text in an HTML element.

Property	Description
`display`	Overrides the usual display of an element and specifies whether the element appears in line, as a block-level element, or as a block-level list item.
`fontFamily`	Specifies the font family, such as Helvetica or Arial, for an HTML text element.
`fontSize`	Specifies the font size for an HTML text element.
`fontStyle`	Specifies the style of the font of an HTML element.
`fontWeight`	Specifies the weight of the font of an HTML element.
`lineHeight`	Specifies the distance between the baselines of two adjacent lines of block-level type.
`listStyleType`	Specifies the style of bullet displayed for list items.
`marginBottom`	Specifies the minimal distance between the bottom of an HTML element and the top of an adjacent element.
`marginLeft`	Specifies the minimal distance between the left side of an HTML element and the right side of an adjacent element.
`marginRight`	Specifies the minimal distance between the right side of an HTML element and the left side of an adjacent element.
`marginTop`	Specifies the minimal distance between the top of an HTML element and the bottom of an adjacent element.
`paddingBottom`	Specifies how much space to insert between the bottom of an element and its content, such as text or an image.
`paddingLeft`	Specifies how much space to insert between the left side of an element and its content, such as text or an image.
`paddingRight`	Specifies how much space to insert between the right side of an element and its content, such as text or an image.
`paddingTop`	Specifies how much space to insert between the top of an element and its content, such as text or an image.
`textAlign`	Specifies the alignment of an HTML block-level text element.
`textDecoration`	Specifies special effects, such as blinking, strike-outs, and underlines, added to an HTML text element.
`textIndent`	Specifies the length of indentation appearing before the first formatted line of a block-level HTML text element.

Property	Description
`textTransform`	Specifies the case of an HTML text element.
`whiteSpace`	Specifies whether or not white space within an HTML element should be collapsed.
`width`	Specifies the width of a block-level HTML element.

Method Summary

Method	Description
`borderWidths`	Specifies the width of the borders of an HTML element.
`margins`	Specifies the minimal distance between the sides of an HTML element and the sides of adjacent elements.
`paddings`	Specifies how much space to insert between the sides of an element and its content, such as text or an image.

In addition, this object inherits the `watch` and `unwatch` methods from `Object`.

align

Specifies the alignment of an HTML element within its parent.

Property of	`Style`
Implemented in	JavaScript 1.2

Syntax `*styleObject*.align = {left | right | none}`

Parameters

`styleObject`	A `Style` object.

Do not confuse `align` with `textAlign`, which specifies the alignment of the content of text elements.

The `align` property is a reflection of the cascading style sheet `float` property.

backgroundColor

Specifies a solid background color for an element.

Property of	`Style`
Implemented in	JavaScript 1.2

Syntax *`styleObject`*`.backgroundColor =` *`colorValue`*

Parameters

`styleObject`	A `Style` object.
`colorValue`	A string evaluating to a color value, as described in Appendix B, "Color Values."

The `backgroundColor` property is a reflection of the cascading style sheet `background-color` property.

backgroundImage

Specifies a background image for an HTML element.

Property of	`Style`
Implemented in	JavaScript 1.2

Syntax *`styleObject`*`.backgroundImage =` *`url`*

Parameters

`styleObject`	A `Style` object.
`url`	A string evaluating to either a full URL or a partial URL relative to the source of the style sheet.

The `backgroundImage` property is a reflection of the cascading style sheet `background-image` property.

borderBottomWidth

Specifies the width of the bottom border of an HTML element.

Property of	`Style`
Implemented in	JavaScript 1.2

Syntax `*styleObject*.borderBottomWidth = *length*`

Parameters

`styleObject`	A `Style` object.
`length`	A string evaluating to a size followed by a unit of measurement; for example, `10pt`.

The `borderBottomWidth` property is a reflection of the cascading style sheet `border-bottom-width` property.

See also `Style.borderLeftWidth`, `Style.borderRightWidth`, `Style.borderTopWidth`, `Style.borderWidths`

borderColor

Specifies the color of the border of an HTML element.

Property of	`Style`
Implemented in	JavaScript 1.2

Syntax `*styleObject*.borderColor = {none | *colorValue*}`

Parameters

`styleObject`	A `Style` object.
`colorValue`	A string evaluating to a color value, as described in Appendix B, "Color Values."

The `borderColor` property is a reflection of the cascading style sheet `border-color` property.

borderLeftWidth

Specifies the width of the left border of an HTML element.

Property of	`Style`
Implemented in	JavaScript 1.2

Syntax `styleObject.borderLeftWidth = length`

Parameters

`styleObject`	A `Style` object.
`length`	A string evaluating to a size followed by a unit of measurement; for example, `10pt`.

The `borderLeftWidth` property is a reflection of the cascading style sheet `border-left-width` property.

See also `Style.borderBottomWidth`, `Style.borderRightWidth`, `Style.borderTopWidth`, `Style.borderWidths`

borderRightWidth

Specifies the width of the right border of an HTML element.

Property of	`Style`
Implemented in	JavaScript 1.2

Syntax `styleObject.borderRightWidth = length`

Parameters

`styleObject`	A `Style` object.
`length`	A string evaluating to a size followed by a unit of measurement; for example, `10pt`.

The `borderRightWidth` property is a reflection of the cascading style sheet `border-right-width` property.

See also `Style.borderBottomWidth`, `Style.borderLeftWidth`, `Style.borderTopWidth`, `Style.borderWidths`

borderStyle

Specifies the style of border, such as solid or double, around a block-level HTML element.

Property of	`Style`
Implemented in	JavaScript 1.2

Syntax *`styleObject`*`.borderStyle = `*`styleType`*

Parameters

`styleObject`	A `Style` object.
`styleType`	A string evaluating to any of the following keywords:

- `none`
- `solid`
- `double`
- `inset`
- `outset`
- `groove`
- `ridge`

You must also specify a border width for the border to be visible.

The `borderStyle` property is a reflection of the cascading style sheet `border-style` property.

borderTopWidth

Specifies the width of the top border of an HTML element.

Property of	`Style`
Implemented in	JavaScript 1.2

Syntax *`styleObject`*`.borderTopWidth = `*`length`*

Parameters

`styleObject`	A `Style` object.
`length`	A string evaluating to a size followed by a unit of measurement; for example, `10pt`.

The `borderTopWidth` property is a reflection of the cascading style sheet `border-top-width` property.

See also `Style.borderBottomWidth`, `Style.borderLeftWidth`, `Style.borderRightWidth`, `Style.borderWidths`

borderWidths

Specifies the width of the borders of an HTML element.

Method of	`Style`
Implemented in	JavaScript 1.2

Syntax `borderWidths(`*`top, right, bottom, left`*`)`

Parameters

`top`	A string specifying the value of the `Style.borderTopWidth` property.
`right`	A string specifying the value of the `Style.borderRightWidth` property.
`bottom`	A string specifying the value of the `Style.borderBottomWidth` property.
`left`	A string specifying the value of the `Style.borderLeftWidth` property.

Description The `borderWidths` method is a convenience shortcut for setting all the border width properties.

See also `Style.borderBottomWidth`, `Style.borderLeftWidth`, `Style.borderRightWidth`, `Style.borderTopWidth`

clear

Specifies the sides of an HTML element that allow floating elements.

Property of	`Style`
Implemented in	JavaScript 1.2

Syntax

```
styleObject.clear = {left | right | both | none}
```

Parameters

`styleObject`	A `Style` object.

The `clear` property is a reflection of the cascading style sheet `clear` property.

color

Specifies the color of the text in an HTML element.

Property of	`Style`
Implemented in	JavaScript 1.2

Syntax

```
styleObject.color = colorValue
```

Parameters

`styleObject`	A `Style` object.
`colorValue`	A string evaluating to a color value, as described in Appendix B, “Color Values.”

The `color` property is a reflection of the cascading style `color` property.

display

Overrides the usual display of an element and specifies whether the element appears in line, as a block-level element, or as a block-level list item.

Property of	`Style`
Implemented in	JavaScript 1.2

Syntax *`styleObject`*`.display =` *`styleType`*

Parameters

`styleObject`	A `Style` object.
`styleType`	A string evaluating to any of the following keywords: • `none` • `block` • `inline` • `list-item`

The `display` property is a reflection of the cascading style `display` property.

fontFamily

Specifies the font family, such as Helvetica or Arial, for an HTML text element.

Property of	`Style`
Implemented in	JavaScript 1.2

Syntax *`styleObject`*`.fontFamily = {`*`specificFamily`*` | `*`genericFamily`*`}`

Parameters

`styleObject`	A `Style` object.
`specificFamily`	A string evaluating to a comma-separated list of specific font families, such as `Helvetica` or `Arial`.
`genericFamily`	A string evaluating to any of the following keywords: • `serif` • `sans-serif` • `cursive` • `monospace` • `fantasy`

The `fontFamily` property is a reflection of the cascading style sheet `font-family` property. The *genericFamily* keywords are available for all platforms, but the specific font displayed varies on each platform.

You can mix the *specificFamily* and *genericFamily* keywords in the same value. For example, the following code displays text in Helvetica if that font is available; otherwise, the text displays in a sans-serif font determined by the operating system:

```
document.tags.H1.fontFamily = "Helvetica, sans-serif"
```

You can also link to a font definition file and download it when a browser loads the web page, guaranteeing that all the fonts are available on a user's system. See *Dynamic HTML in Netscape Communicator*.

fontSize

Specifies the font size for an HTML text element.

Property of	`Style`
Implemented in	JavaScript 1.2

Syntax

```
styleObject.fontSize =
   {absoluteSize | relativeSize | length | percentage}
```

Parameters

`styleObject`	A `Style` object.
`absoluteSize`	A string evaluating to any of the following keywords: • `xx-small` • `x-small` • `small` • `medium` • `large` • `x-large` • `xx-large`
`relativeSize`	A string evaluating to a size relative to the size of the parent element, indicated by either of the following keywords: • `smaller` • `larger`
`length`	A string evaluating to a size followed by a unit of measurement; for example, `18pt`.
`percentage`	A string evaluating to a percent of the size of the parent element; for example, `50%`.

The `fontSize` property is a reflection of the cascading style sheet `font-size` property. By default, the initial value is `medium`.

fontStyle

Specifies the style of the font of an HTML element.

Property of	`Style`
Implemented in	JavaScript 1.2

Syntax `styleObject.fontStyle = styleType`

Parameters

`styleObject`	A `Style` object.
`styleType`	A string evaluating to either of the following keywords: • `normal` • `italic`

The `fontStyle` property is a reflection of the cascading style sheet `font-style` property.

fontWeight

Specifies the weight of the font of an HTML element.

Property of	`Style`
Implemented in	JavaScript 1.2

Syntax `*styleObject*.fontWeight = {*absolute* | *relative* | *numeric*}`

Parameters

`styleObject`	A `Style` object.
`absolute`	A string evaluating to either of the following keywords: • `normal` • `bold`
`relative`	A string evaluating to a weight relative to the weight of the parent element, indicated by either of the following keywords: • `bolder` • `lighter`
`numeric`	A string evaluating to a numeric value between 100 and 900, where 100 indicates the lightest weight and 900 indicates the heaviest weight.

The `fontWeight` property is a reflection of the cascading style sheet `font-weight` property.

lineHeight

Specifies the distance between the baselines of two adjacent lines of block-level type.

Property of	`Style`
Implemented in	JavaScript 1.2

Syntax `*styleObject*.lineHeight = {*number* | *length* | *percentage* | normal}`

Parameters

`styleObject`	A `Style` object.
`number`	A string evaluating to a size without a unit of measurement; for example, `1.2`.
`length`	A string evaluating to a size followed by a unit of measurement; for example, `10pt`.
`percentage`	A string evaluating to a percentage of the parent element's width; for example, `20%`.
`normal`	The string normal, indicating that the line height is determined automatically by Navigator.

The `lineHeight` property is a reflection of the cascading style sheet `line-height` property.

When you set the `lineHeight` property by specifying ***`number`***, Navigator calculates the line height by multiplying the font size of the current element by *`number`*. For example, if `lineHeight` is set to 1.2 in a paragraph using a 10-point font, the line height is 12 points.

When you set `lineHeight` with *`number`*, children of the current paragraph inherit the line height *factor*; when you set `lineHeight` with *`length`* or *`percentage`*, children inherit the *resulting value*.

listStyleType

Specifies the style of bullet displayed for list items.

Property of	`Style`
Implemented in	JavaScript 1.2

Syntax *`styleObject`*`.listStyleType = `*`styleType`*

Parameters

`styleObject`	A `Style` object.
`styleType`	A string evaluating to any of the following keywords:

- `disc`
- `circle`
- `square`
- `decimal`
- `lower-roman`
- `upper-roman`
- `lower-alpha`
- `upper-alpha`
- `none`

The `listStyleType` property is a reflection of the cascading style sheet `list-style-type` property.

marginBottom

Specifies the minimal distance between the bottom of an HTML element and the top of an adjacent element.

Property of	`Style`
Implemented in	JavaScript 1.2

Syntax `styleObject.marginBottom = {length | percentage | auto}`

Parameters

`styleObject`	A `Style` object.
`length`	A string evaluating to a size followed by a unit of measurement; for example, `10pt`.
`percentage`	A string evaluating to a percentage of the parent element's width; for example, `20%`.
`auto`	The string auto, indicating that the margin is determined automatically by Navigator.

The `marginBottom` property is a reflection of the cascading style sheet `margin-bottom` property.

See also `Style.marginLeft`, `Style.marginRight`, `Style.marginTop`, `Style.margins`

marginLeft

Specifies the minimal distance between the left side of an HTML element and the right side of an adjacent element.

Property of	`Style`
Implemented in	JavaScript 1.2

Syntax `styleObject.marginLeft = {length | percentage | auto}`

Parameters

`styleObject`	A `Style` object.
`length`	A string evaluating to a size followed by a unit of measurement; for example, `10pt`.
`percentage`	A string evaluating to a percentage of the parent element's width; for example, `20%`.
`auto`	The string auto, indicating that the margin is determined automatically by Navigator.

The `marginLeft` property is a reflection of the cascading style sheet `margin-left` property.

See also `Style.marginBottom`, `Style.marginRight`, `Style.marginTop`, `Style.margins`

marginRight

Specifies the minimal distance between the right side of an HTML element and the left side of an adjacent element.

Property of	`Style`
Implemented in	JavaScript 1.2

Syntax *`styleObject`*`.marginRight = {`*`length`*` | `*`percentage`*` | auto}`

Parameters

`styleObject`	A `Style` object.
`length`	A string evaluating to a size followed by a unit of measurement; for example, `10pt`.
`percentage`	A string evaluating to a percentage of the parent element's width; for example, `20%`.
`auto`	The string auto, indicating that the margin is determined automatically by Navigator.

The `marginRight` property is a reflection of the cascading style sheet `margin-right` property.

See also `Style.marginBottom`, `Style.marginLeft`, `Style.marginTop`, `Style.margins`

margins

Specifies the minimal distance between the sides of an HTML element and the sides of adjacent elements.

Method of	`Style`
Implemented in	JavaScript 1.2

Syntax `margins(`*`top, right, bottom, left`*`)`

Parameters

`top`	A string specifying the value of the `Style.marginTop` property.
`right`	A string specifying the value of the `Style.marginRight` property.
`bottom`	A string specifying the value of the `Style.marginBottom` property.
`left`	A string specifying the value of the `Style.marginLeft` property.

Description The `margins` method is a convenience shortcut for setting all the margin properties.

See also `Style.marginBottom`, `Style.marginLeft`, `Style.marginRight`, `Style.marginTop`

marginTop

Specifies the minimal distance between the top of an HTML element and the bottom of an adjacent element.

Property of	`Style`
Implemented in	JavaScript 1.2

Syntax `styleObject.marginTop = {length | percentage | auto}`

Parameters

`styleObject`	A `Style` object.
`length`	A string evaluating to a size followed by a unit of measurement; for example, `10pt`.
`percentage`	A string evaluating to a percentage of the parent element's width; for example, `20%`.
`auto`	The string auto, indicating that the margin is determined automatically by Navigator.

The `marginTop` property is a reflection of the cascading style sheet `margin-top` property.

See also `Style.marginBottom`, `Style.marginLeft`, `Style.marginRight`, `Style.margins`

paddingBottom

Specifies how much space to insert between the bottom of an element and its content, such as text or an image.

Property of	`Style`
Implemented in	JavaScript 1.2

Syntax `styleObject.paddingBottom = {length | percentage}`

Parameters

`styleObject`	A `Style` object.
`length`	A string evaluating to a size followed by a unit of measurement; for example, `10pt`.
`percentage`	A string evaluating to a percentage of the parent element's width; for example, `20%`.

The `paddingBottom` property is a reflection of the cascading style sheet `padding-bottom` property.

See also `Style.paddingLeft`, `Style.paddingRight`, `Style.paddingTop`, `Style.paddings`

paddingLeft

Specifies how much space to insert between the left side of an element and its content, such as text or an image.

Property of	`Style`
Implemented in	JavaScript 1.2

Syntax `*styleObject*.paddingLeft = {*length* | *percentage*}`

Parameters

`styleObject`	A `Style` object.
`length`	A string evaluating to a size followed by a unit of measurement; for example, `10pt`.
`percentage`	A string evaluating to a percentage of the parent element's width; for example, `20%`.

The `paddingLeft` property is a reflection of the cascading style sheet `padding-left` property.

See also `Style.paddingBottom`, `Style.paddingRight`, `Style.paddingTop`, `Style.paddings`

paddingRight

Specifies how much space to insert between the right side of an element and its content, such as text or an image.

Property of	`Style`
Implemented in	JavaScript 1.2

Syntax `*styleObject*.paddingRight = {*length* | *percentage*}`

Parameters

`styleObject`	A `Style` object.
`length`	A string evaluating to a size followed by a unit of measurement; for example, `10pt`.
`percentage`	A string evaluating to a percentage of the parent element's width; for example, `20%`.

The `paddingRight` property is a reflection of the cascading style sheet `padding-right` property.

See also `Style.paddingBottom`, `Style.paddingLeft`, `Style.paddingTop`, `Style.paddings`

paddings

Specifies how much space to insert between the sides of an element and its content, such as text or an image.

Method of	`Style`
Implemented in	JavaScript 1.2

Syntax `paddings(*top, right, bottom, left*)`

Parameters

`top`	A string specifying the value of the `Style.paddingTop` property.
`right`	A string specifying the value of the `Style.paddingRight` property.
`bottom`	A string specifying the value of the `Style.paddingBottom` property.
`left`	A string specifying the value of the `Style.paddingLeft` property.

Description The `paddings` method is a convenience shortcut for setting all the padding properties.

See also `Style.paddingBottom`, `Style.paddingLeft`, `Style.paddingRight`, `Style.paddingTop`

paddingTop

Specifies how much space to insert between the top of an element and its content, such as text or an image.

Property of	`Style`
Implemented in	JavaScript 1.2

Syntax

```
styleObject.paddingTop = {length | percentage}
```

Parameters

`styleObject`	A `Style` object.
`length`	A string evaluating to a size followed by a unit of measurement; for example, `10pt`.
`percentage`	A string evaluating to a percentage of the parent element's width; for example, `20%`.

The `paddingTop` property is a reflection of the cascading style sheet `padding-top` property.

See also `Style.paddingBottom`, `Style.paddingLeft`, `Style.paddingRight`, `Style.paddings`

textAlign

Specifies the alignment of an HTML block-level text element.

Property of	`Style`
Implemented in	JavaScript 1.2

Syntax `*styleObject*.textAlign = *alignment*`

Parameters

`styleObject`	A `Style` object.
`alignment`	A string evaluating to any of the following keywords: • `left` • `right` • `center` • `justify`

Do not confuse `textAlign` with `align`, which specifies the alignment of an HTML element within its parent.

The `textAlign` property is a reflection of the cascading style sheet `text-align` property.

textDecoration

Specifies special effects, such as blinking, strike-outs, and underlines, added to an HTML text element.

Property of	`Style`
Implemented in	JavaScript 1.2

Syntax *`styleObject`*`.textDecoration = `*`decoration`*

Parameters

`styleObject`	A `Style` object.
`decoration`	A string evaluating to any of the following keywords:

- `none`
- `underline`
- `line-through`
- `blink`

The `textDecoration` property is a reflection of the cascading style sheet `text-decoration` property.

textIndent

Specifies the length of indentation appearing before the first formatted line of a block-level HTML text element.

Property of	`Style`
Implemented in	JavaScript 1.2

Syntax *`styleObject`*`.textIndent = {`*`length`*` | `*`percentage`*`}`

Parameters

`styleObject`	A `Style` object.
`length`	A string evaluating to a size followed by a unit of measurement; for example, `18pt`.
`percentage`	A string evaluating to a percentage of the parent element's width; for example, `20%`.

The `textIndent` property is a reflection of the cascading style sheet `text-indent` property.

textTransform

Specifies the case of an HTML text element.

Property of	`Style`
Implemented in	JavaScript 1.2

Syntax *`styleObject`*`.textTransform = `*`transformation`*

Parameters

`styleObject`	A `Style` object.
`transformation`	A string evaluating to any of the following keywords:

- `none`
- `capitalize`
- `uppercase`
- `lowercase`

The `textTransform` property is a reflection of the cascading style sheet `text-transform` property.

whiteSpace

Specifies whether or not white space within an HTML element should be collapsed.

Property of	`Style`
Implemented in	JavaScript 1.2

Syntax *`styleObject`*`.whiteSpace = {normal | pre}`

Parameters

`styleObject`	A `Style` object.

The `whiteSpace` property is a reflection of the cascading style sheet `white-space` property.

width

Specifies the width of a block-level HTML element.

Property of	`Style`
Implemented in	JavaScript 1.2

Syntax `*styleObject*.width = {*length* | *percentage* | auto}`

Parameters

`styleObject`	A `Style` object.
`length`	A string evaluating to a size followed by a unit of measurement; for example, `10pt`.
`percentage`	A string evaluating to a percentage of the parent element's width; for example, `20%`.
`auto`	The string auto, indicating that the width is determined automatically by Navigator.

The `width` property is a reflection of the cascading style sheet `width` property.

The `Style.marginLeft` and `Style.marginRight` properties take precedence over the `Style.width` property. For example, if `marginLeft` is set to 25%, `marginRight` is set to 10%, and `width` is set to 100%, Navigator ignores the `width` value and uses 65% for the `width` setting.

Submit

A submit button on an HTML form. A submit button causes a form to be submitted.

Client-side object

Implemented in	JavaScript 1.0
	JavaScript 1.1: added `type` property; added onBlur and onFocus event handlers; added `blur` and `focus` methods
	JavaScript 1.2: added `handleEvent` method

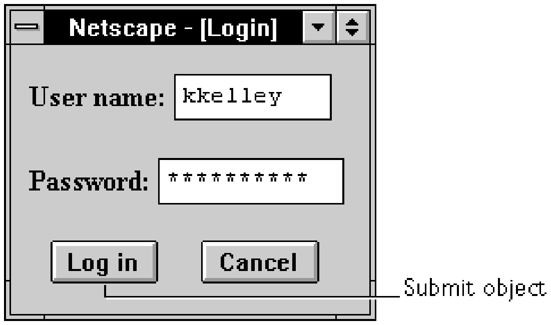
Created by The HTML `INPUT` tag, with `"submit"` as the value of the `TYPE` attribute. For a given form, the JavaScript runtime engine creates an appropriate `Submit` object and puts it in the `elements` array of the corresponding `Form` object. You access a `Submit` object by indexing this array. You can index the array either by number or, if supplied, by using the value of the `NAME` attribute.

Event handlers

- `onBlur`
- `onClick`
- `onFocus`

Security Submitting a form to a `mailto:` or `news:` URL requires the `UniversalSendMail` privilege. For information on security, see the *Client-Side JavaScript Guide*.

Description A `Submit` object on a form looks as follows:



A `Submit` object is a form element and must be defined within a `FORM` tag.

Clicking a submit button submits a form to the URL specified by the form's `action` property. This action always loads a new page into the client; it may be the same as the current page, if the action so specifies or is not specified.

The submit button's `onClick` event handler cannot prevent a form from being submitted; instead, use the form's `onSubmit` event handler or use the `submit` method instead of a `Submit` object. See the examples for the `Form` object.

Property Summary

Property	Description
`form`	Specifies the form containing the `Submit` object.
`name`	Reflects the `NAME` attribute.
`type`	Reflects the `TYPE` attribute.
`value`	Reflects the `VALUE` attribute.

Method Summary

Method	Description
`blur`	Removes focus from the submit button.
`click`	Simulates a mouse-click on the submit button.
`focus`	Gives focus to the submit button.
`handleEvent`	Invokes the handler for the specified event.

In addition, this object inherits the `watch` and `unwatch` methods from `Object`.

Examples The following example creates a `Submit` object called `submitButton`. The text "Done" is displayed on the face of the button.

```
<INPUT TYPE="submit" NAME="submitButton" VALUE="Done">
```

See also the examples for the `Form`.

See also `Button`, `Form`, `Reset`, `Form.submit`, `onSubmit`

blur

Removes focus from the submit button.

Method of	`Submit`
Implemented in	JavaScript 1.0

Syntax `blur()`

Parameters None

See also `Submit.focus`

click

Simulates a mouse-click on the submit button, but does *not* trigger an object's `onClick` event handler.

Method of	`Submit`
Implemented in	JavaScript 1.0

Syntax `click()`

Parameters None

focus

Navigates to the submit button and gives it focus.

Method of	`Submit`
Implemented in	JavaScript 1.0

Syntax `focus()`

Parameters None

See also `Submit.blur`

form

An object reference specifying the form containing the submit button.

Property of	`Submit`
Read-only	
Implemented in	JavaScript 1.0

Description Each form element has a `form` property that is a reference to the element's parent form. This property is especially useful in event handlers, where you might need to refer to another element on the current form.

Examples The following example shows a form with several elements. When the user clicks `button2`, the function `showElements` displays an alert dialog box containing the names of each element on the form `myForm`.

```
<SCRIPT>
function showElements(theForm) {
   str = "Form Elements of form " + theForm.name + ": \n "
   for (i = 0; i < theForm.length; i++)
      str += theForm.elements[i].name + "\n"
   alert(str)
}
</SCRIPT>
<FORM NAME="myForm">
Form name:<INPUT TYPE="text" NAME="text1" VALUE="Beluga">
<P>
<INPUT NAME="button1" TYPE="button" VALUE="Show Form Name"
   onClick="this.form.text1.value=this.form.name">
<INPUT NAME="button2" TYPE="submit" VALUE="Show Form Elements"
   onClick="showElements(this.form)">
</FORM>
```

The alert dialog box displays the following text:

```
Form Elements of form myForm:
text1
button1
button2
```

See also `Form`

handleEvent

Invokes the handler for the specified event.

Method of	`Submit`
Implemented in	JavaScript 1.2

Syntax `handleEvent(`*`event`*`)`

Parameters

`event`	The name of an event for which the specified object has an event handler.

Description For information on handling events, see the *Client-Side JavaScript Guide*.

name

A string specifying the submit button's name.

Property of	`Submit`
Implemented in	JavaScript 1.0

Security **JavaScript 1.1.** This property is tainted by default. For information on data tainting, see the *Client-Side JavaScript Guide*.

Description The `name` property initially reflects the value of the `NAME` attribute. Changing the `name` property overrides this setting.

Do not confuse the `name` property with the label displayed on the `Submit` button. The `value` property specifies the label for this button. The `name` property is not displayed on the screen; it is used to refer programmatically to the button.

If multiple objects on the same form have the same `NAME` attribute, an array of the given name is created automatically. Each element in the array represents an individual `Form` object. Elements are indexed in source order starting at 0. For example, if two `Text` elements and a `Submit` element on the same form have their `NAME` attribute set to `"myField"`, an array with the elements `myField[0]`, `myField[1]`, and `myField[2]` is created. You need to be aware of this situation in your code and know whether `myField` refers to a single element or to an array of elements.

Examples In the following example, the `valueGetter` function uses a `for` loop to iterate over the array of elements on the `valueTest` form. The `msgWindow` window displays the names of all the elements on the form:

```
newWindow=window.open("http://home.netscape.com")

function valueGetter() {
   var msgWindow=window.open("")
   for (var i = 0; i < newWindow.document.valueTest.elements.length; i++) {
      msgWindow.document.write(newWindow.document.valueTest.elements[i].name + "<BR>")
   }
}
```

See also `Submit.value`

type

For all `Submit` objects, the value of the `type` property is `"submit"`. This property specifies the form element's type.

Property of	`Submit`
Read-only	
Implemented in	JavaScript 1.1

Examples The following example writes the value of the `type` property for every element on a form.

```
for (var i = 0; i < document.form1.elements.length; i++) {
   document.writeln("<BR>type is " + document.form1.elements[i].type)
}
```

value

A string that reflects the submit button's `VALUE` attribute.

Property of	`Submit`
Read-only	
Implemented in	JavaScript 1.0

Security **JavaScript 1.1.** This property is tainted by default. For information on data tainting, see the *Client-Side JavaScript Guide*.

Description When a `VALUE` attribute is specified in HTML, the `value` property is that string and is displayed on the face of the button. When a `VALUE` attribute is not specified in HTML, the `value` property for the button is the string "Submit Query."

Do not confuse the `value` property with the `name` property. The `name` property is not displayed on the screen; it is used to refer programmatically to the button.

Examples The following function evaluates the `value` property of a group of buttons and displays it in the `msgWindow` window:

```
function valueGetter() {
   var msgWindow=window.open("")
   msgWindow.document.write("submitButton.value is " +
      document.valueTest.submitButton.value + "<BR>")
   msgWindow.document.write("resetButton.value is " +
      document.valueTest.resetButton.value + "<BR>")
   msgWindow.document.write("helpButton.value is " +
      document.valueTest.helpButton.value + "<BR>")
   msgWindow.document.close()
}
```

This example displays the following values:

```
Query Submit
Reset
Help
```

The previous example assumes the buttons have been defined as follows:

```
<INPUT TYPE="submit" NAME="submitButton">
<INPUT TYPE="reset" NAME="resetButton">
<INPUT TYPE="button" NAME="helpButton" VALUE="Help">
```

See also `Submit.name`

sun

A top-level object used to access any Java class in the package `sun.*`.
Core object

Implemented in JavaScript 1.1, NES 2.0

Created by The `sun` object is a top-level, predefined JavaScript object. You can automatically access it without using a constructor or calling a method.

Description The `sun` object is a convenience synonym for the property `Packages.sun`.

See also `Packages`, `Packages.sun`

Text

A text input field on an HTML form. The user can enter a word, phrase, or series of numbers in a text field.

Client-side object

Implemented in	JavaScript 1.0
	JavaScript 1.1: added `type` property
	JavaScript 1.2: added `handleEvent` method

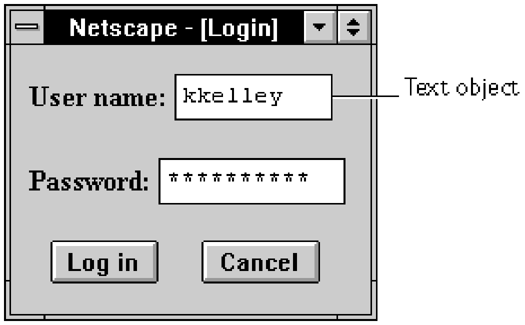
Created by The HTML `INPUT` tag, with `"text"` as the value of the `TYPE` attribute. For a given form, the JavaScript runtime engine creates appropriate `Text` objects and puts these objects in the `elements` array of the corresponding `Form` object. You access a `Text` object by indexing this array. You can index the array either by number or, if supplied, by using the value of the `NAME` attribute.

To define a `Text` object, use standard HTML syntax with the addition of JavaScript event handlers.

Event handlers

- `onBlur`
- `onChange`
- `onFocus`
- `onSelect`

Description A `Text` object on a form looks as follows:



A `Text` object is a form element and must be defined within a `FORM` tag.

`Text` objects can be updated (redrawn) dynamically by setting the `value` property (`this.value`).

Property Summary

Property	Description
`defaultValue`	Reflects the `VALUE` attribute.
`form`	Specifies the form containing the `Text` object.
`name`	Reflects the `NAME` attribute.
`type`	Reflects the `TYPE` attribute.
`value`	Reflects the current value of the `Text` object's field.

Method Summary

Method	Description
`blur`	Removes focus from the object.
`focus`	Gives focus to the object.
`handleEvent`	Invokes the handler for the specified event.
`select`	Selects the input area of the object.

In addition, this object inherits the `watch` and `unwatch` methods from `Object`.

Examples **Example 1.** The following example creates a `Text` object that is 25 characters long. The text field appears immediately to the right of the words "Last name:". The text field is blank when the form loads.

```
<B>Last name:</B> <INPUT TYPE="text" NAME="last_name" VALUE="" SIZE=25>
```

Example 2. The following example creates two `Text` objects on a form. Each object has a default value. The `city` object has an `onFocus` event handler that selects all the text in the field when the user tabs to that field. The `state` object has an `onChange` event handler that forces the value to uppercase.

```
<FORM NAME="form1">
<BR><B>City: </B><INPUT TYPE="text" NAME="city" VALUE="Anchorage"
   SIZE="20" onFocus="this.select()">
<B>State: </B><INPUT TYPE="text" NAME="state" VALUE="AK" SIZE="2"
   onChange="this.value=this.value.toUpperCase()">
</FORM>
```

See also the examples for the `onBlur`, `onChange`, `onFocus`, and `onSelect`.

See also `Text`, `Form`, `Password`, `String`, `Textarea`

blur

Removes focus from the text field.

Method of	`Text`
Implemented in	JavaScript 1.0

Syntax `blur()`

Parameters None

Examples The following example removes focus from the text element `userText`:

```
userText.blur()
```

This example assumes that the text element is defined as

```
<INPUT TYPE="text" NAME="userText">
```

See also `Text.focus`, `Text.select`

defaultValue

A string indicating the default value of a `Text` object.

Property of	`Text`
Implemented in	JavaScript 1.0

Security **JavaScript 1.1.** This property is tainted by default. For information on data tainting, see the *Client-Side JavaScript Guide*.

Description The initial value of `defaultValue` reflects the value of the `VALUE` attribute. Setting `defaultValue` programmatically overrides the initial setting.

You can set the `defaultValue` property at any time. The display of the related object does not update when you set the `defaultValue` property, only when you set the `value` property.

Examples The following function evaluates the `defaultValue` property of objects on the `surfCity` form and displays the values in the `msgWindow` window:

```
function defaultGetter() {
   msgWindow=window.open("")
   msgWindow.document.write("hidden.defaultValue is " +
      document.surfCity.hiddenObj.defaultValue + "<BR>")
   msgWindow.document.write("password.defaultValue is " +
      document.surfCity.passwordObj.defaultValue + "<BR>")
   msgWindow.document.write("text.defaultValue is " +
      document.surfCity.textObj.defaultValue + "<BR>")
   msgWindow.document.write("textarea.defaultValue is " +
      document.surfCity.textareaObj.defaultValue + "<BR>")
   msgWindow.document.close()
}
```

See also `Text.value`

focus

Navigates to the text field and gives it focus.

Method of	`Text`
Implemented in	JavaScript 1.0

Syntax `focus()`

Parameters None

Description Use the `focus` method to navigate to a text field and give it focus. You can then either programmatically enter a value in the field or let the user enter a value. If you use this method without the `select` method, the cursor is positioned at the beginning of the field.

Examples See example for `select`.

See also `Text.blur`, `Text.select`

form

An object reference specifying the form containing this object.

Property of	`Text`
Read-only	
Implemented in	JavaScript 1.0

Description Each form element has a `form` property that is a reference to the element's parent form. This property is especially useful in event handlers, where you might need to refer to another element on the current form.

Examples **Example 1.** In the following example, the form `myForm` contains a `Text` object and a button. When the user clicks the button, the value of the `Text` object is set to the form's name. The button's `onClick` event handler uses `this.form` to refer to the parent form, `myForm`.

```
<FORM NAME="myForm">
Form name:<INPUT TYPE="text" NAME="text1" VALUE="Beluga">
<P>
<INPUT NAME="button1" TYPE="button" VALUE="Show Form Name"
   onClick="this.form.text1.value=this.form.name">
</FORM>
```

Example 2. The following example shows a form with several elements. When the user clicks `button2`, the function `showElements` displays an alert dialog box containing the names of each element on the form `myForm`.

```
function showElements(theForm) {
   str = "Form Elements of form " + theForm.name + ": \n "
   for (i = 0; i < theForm.length; i++)
      str += theForm.elements[i].name + "\n"
   alert(str)
}
</script>
<FORM NAME="myForm">
Form name:<INPUT TYPE="text" NAME="text1" VALUE="Beluga">
<P>
<INPUT NAME="button1" TYPE="button" VALUE="Show Form Name"
   onClick="this.form.text1.value=this.form.name">
<INPUT NAME="button2" TYPE="button" VALUE="Show Form Elements"
   onClick="showElements(this.form)">
</FORM>
```

The alert dialog box displays the following text:

```
JavaScript Alert:
Form Elements of form myForm:
text1
button1
button2
```

Example 3. The following example uses an object reference, rather than the `this` keyword, to refer to a form. The code returns a reference to `myForm`, which is a form containing `myTextObject`.

```
document.myForm.myTextObject.form
```

See also `Form`

handleEvent

Invokes the handler for the specified event.

Method of	`Text`
Implemented in	JavaScript 1.2

Syntax `handleEvent(`*`event`*`)`

Parameters

`event`	The name of an event for which the specified object has an event handler.

name

A string specifying the name of this object.

Property of	`Text`
Implemented in	JavaScript 1.0

Security **JavaScript 1.1.** This property is tainted by default. For information on data tainting, see the *Client-Side JavaScript Guide*.

Description The `name` property initially reflects the value of the `NAME` attribute. Changing the `name` property overrides this setting. The `name` property is not displayed on-screen; it is used to refer to the objects programmatically.

If multiple objects on the same form have the same `NAME` attribute, an array of the given name is created automatically. Each element in the array represents an individual `Form` object. Elements are indexed in source order starting at 0. For example, if two `Text` elements and a `Textarea` element on the same form have their `NAME` attribute set to `"myField"`, an array with the elements `myField[0]`, `myField[1]`, and `myField[2]` is created. You need to be aware of this situation in your code and know whether `myField` refers to a single element or to an array of elements.

Examples In the following example, the `valueGetter` function uses a `for` loop to iterate over the array of elements on the `valueTest` form. The `msgWindow` window displays the names of all the elements on the form:

```
newWindow=window.open("http://home.netscape.com")

function valueGetter() {
   var msgWindow=window.open("")
   for (var i = 0; i < newWindow.document.valueTest.elements.length; i++) {
      msgWindow.document.write(newWindow.document.valueTest.elements[i].name + "<BR>")
   }
}
```

select

Selects the input area of the text field.

Method of	`Text`
Implemented in	JavaScript 1.0

Syntax `select()`

Parameters None

Description Use the `select` method to highlight the input area of a text field. You can use the `select` method with the `focus` method to highlight a field and position the cursor for a user response. This makes it easy for the user to replace all the text in the field.

Examples The following example uses an `onClick` event handler to move the focus to a text field and select that field for changing:

```
<FORM NAME="myForm">
<B>Last name: </B><INPUT TYPE="text" NAME="lastName" SIZE=20 VALUE="Pigman">
<BR><B>First name: </B><INPUT TYPE="text" NAME="firstName" SIZE=20 VALUE="Victoria">
<BR><BR>
<INPUT TYPE="button" VALUE="Change last name"
   onClick="this.form.lastName.select();this.form.lastName.focus();">
</FORM>
```

See also `Text.blur`, `Text.focus`

type

For all `Text` objects, the value of the `type` property is `"text"`. This property specifies the form element's type.

Property of	`Text`
Read-only	
Implemented in	JavaScript 1.1

Examples The following example writes the value of the `type` property for every element on a form.

```
for (var i = 0; i < document.form1.elements.length; i++) {
   document.writeln("<BR>type is " + document.form1.elements[i].type)
}
```

value

A string that reflects the `VALUE` attribute of the object.

Property of	`Text`
Implemented in	JavaScript 1.0

Security **JavaScript 1.1.** This property is tainted by default. For information on data tainting, see the *Client-Side JavaScript Guide*.

Description The `value` property is a string that initially reflects the `VALUE` attribute. This string is displayed in the text field. The value of this property changes when a user or a program modifies the field.

You can set the `value` property at any time. The display of the `Text` object updates immediately when you set the `value` property.

Examples The following function evaluates the `value` property of a group of buttons and displays it in the `msgWindow` window:

```
function valueGetter() {
   var msgWindow=window.open("")
   msgWindow.document.write("submitButton.value is " +
      document.valueTest.submitButton.value + "<BR>")
   msgWindow.document.write("resetButton.value is " +
      document.valueTest.resetButton.value + "<BR>")
   msgWindow.document.write("myText.value is " +
      document.valueTest.myText.value + "<BR>")
   msgWindow.document.close()
}
```

This example displays the following:

```
submitButton.value is Query Submit
resetButton.value is Reset
myText.value is Stonefish are dangerous.
```

The previous example assumes the buttons have been defined as follows:

```
<INPUT TYPE="submit" NAME="submitButton">
<INPUT TYPE="reset" NAME="resetButton">
<INPUT TYPE="text" NAME="myText" VALUE="Stonefish are dangerous.">
```

See also `Text.defaultValue`

Textarea

A multiline input field on an HTML form. The user can use a textarea field to enter words, phrases, or numbers.

Client-side object

Implemented in	JavaScript 1.0
	JavaScript 1.1: added `type` property
	JavaScript 1.2: added `handleEvent` method

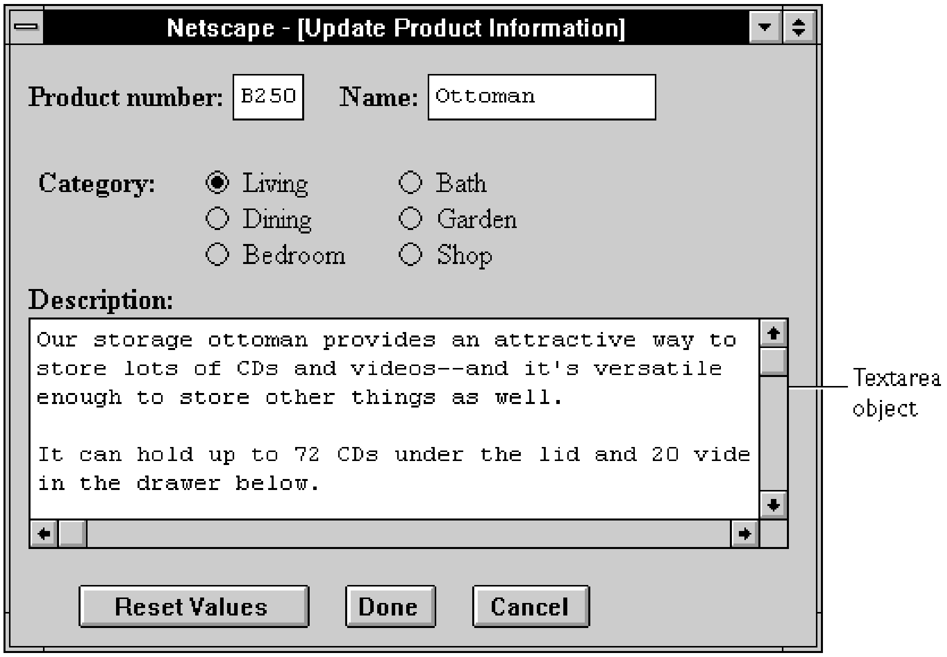
Created by The HTML `TEXTAREA` tag. For a given form, the JavaScript runtime engine creates appropriate `Textarea` objects and puts these objects in the `elements` array of the corresponding `Form` object. You access a `Textarea` object by indexing this array. You can index the array either by number or, if supplied, by using the value of the `NAME` attribute.

To define a text area, use standard HTML syntax with the addition of JavaScript event handlers.

Event handlers

- `onBlur`
- `onChange`
- `onFocus`
- `onKeyDown`
- `onKeyPress`
- `onKeyUp`
- `onSelect`

Description A `Textarea` object on a form looks as follows:



```
Our storage ottoman provides an attractive way to
store lots of CDs and videos--and it's versatile
enough to store other things as well.

It can hold up to 72 CDs under the lid and 20 vide
in the drawer below.
```

A `Textarea` object is a form element and must be defined within a `FORM` tag.

`Textarea` objects can be updated (redrawn) dynamically by setting the `value` property (`this.value`).

To begin a new line in a `Textarea` object, you can use a newline character. Although this character varies from platform to platform (Unix is `\n`, Windows is `\r`, and Macintosh is `\n`), JavaScript checks for all newline characters before setting a string-valued property and translates them as needed for the user's platform. You could also enter a newline character programmatically—one way is to test the `navigator.appVersion` property to determine the current platform, then set the newline character accordingly. See `navigator.appVersion` for an example.

Property Summary

Property	Description
`defaultValue`	Reflects the `VALUE` attribute.
`form`	Specifies the form containing the `Textarea` object.
`name`	Reflects the `NAME` attribute.
`type`	Specifies that the object is a `Textarea` object.
`value`	Reflects the current value of the `Textarea` object.

Method Summary

Method	Description
`blur`	Removes focus from the object.
`focus`	Gives focus to the object.
`handleEvent`	Invokes the handler for the specified event.
`select`	Selects the input area of the object.

In addition, this object inherits the `watch` and `unwatch` methods from `Object`.

Examples **Example 1.** The following example creates a `Textarea` object that is six rows long and 55 columns wide. The textarea field appears immediately below the word "Description:". When the form loads, the `Textarea` object contains several lines of data, including one blank line.

```
<B>Description:</B>
<BR><TEXTAREA NAME="item_description" ROWS=6 COLS=55>
Our storage ottoman provides an attractive way to
store lots of CDs and videos--and it's versatile
enough to store other things as well.

It can hold up to 72 CDs under the lid and 20 videos
in the drawer below.
</TEXTAREA>
```

Example 2. The following example creates a string variable containing newline characters for different platforms. When the user clicks the button, the `Textarea` object is populated with the value from the string variable. The result is three lines of text in the `Textarea` object.

```
<SCRIPT>
myString="This is line one.\nThis is line two.\rThis is line three."
</SCRIPT>
<FORM NAME="form1">
<INPUT TYPE="button" Value="Populate the textarea"
onClick="document.form1.textarea1.value=myString">
   <P>
<TEXTAREA NAME="textarea1" ROWS=6 COLS=55></TEXTAREA>
```

See also `Form`, `Password`, `String`, `Text`

blur

Removes focus from the object.

Method of	`Textarea`
Implemented in	JavaScript 1.0

Syntax `blur()`

Parameters None

Examples The following example removes focus from the textarea element `userText:`

```
userText.blur()
```

This example assumes that the textarea is defined as

```
<TEXTAREA NAME="userText">
Initial text for the text area.
</TEXTAREA>
```

See also `Textarea.focus`, `Textarea.select`

defaultValue

A string indicating the default value of a `Textarea` object.

Property of	`Textarea`
Implemented in	JavaScript 1.0

Security **JavaScript 1.1.** This property is tainted by default. For information on data tainting, see the *Client-Side JavaScript Guide*.

Description The initial value of `defaultValue` reflects the value specified between the `TEXTAREA` start and end tags. Setting `defaultValue` programmatically overrides the initial setting.

You can set the `defaultValue` property at any time. The display of the related object does not update when you set the `defaultValue` property, only when you set the `value` property.

Examples The following function evaluates the `defaultValue` property of objects on the `surfCity` form and displays the values in the `msgWindow` window:

```
function defaultGetter() {
   msgWindow=window.open("")
   msgWindow.document.write("hidden.defaultValue is " +
      document.surfCity.hiddenObj.defaultValue + "<BR>")
   msgWindow.document.write("password.defaultValue is " +
      document.surfCity.passwordObj.defaultValue + "<BR>")
   msgWindow.document.write("text.defaultValue is " +
      document.surfCity.textObj.defaultValue + "<BR>")
   msgWindow.document.write("textarea.defaultValue is " +
      document.surfCity.textareaObj.defaultValue + "<BR>")
   msgWindow.document.close()
}
```

See also `Textarea.value`

focus

Navigates to the textarea field and gives it focus.

Method of	`Textarea`
Implemented in	JavaScript 1.0

Syntax `focus()`

Parameters None

Description Use the `focus` method to navigate to the textarea field and give it focus. You can then either programmatically enter a value in the field or let the user enter a value. If you use this method without the `select` method, the cursor is positioned at the beginning of the field.

See also `Textarea.blur`, `Textarea.select`

Examples See example for `Textarea.select`.

form

An object reference specifying the form containing this object.

Property of	`Textarea`
Read-only	
Implemented in	JavaScript 1.0

Description Each form element has a `form` property that is a reference to the element's parent form. This property is especially useful in event handlers, where you might need to refer to another element on the current form.

Examples **Example 1.** The following example shows a form with several elements. When the user clicks `button2`, the function `showElements` displays an alert dialog box containing the names of each element on the form `myForm`.

```
function showElements(theForm) {
   str = "Form Elements of form " + theForm.name + ": \n "
   for (i = 0; i < theForm.length; i++)
      str += theForm.elements[i].name + "\n"
   alert(str)
}
</script>
```

```
<FORM NAME="myForm">
Form name:<INPUT TYPE="textarea" NAME="text1" VALUE="Beluga">
<P>
<INPUT NAME="button1" TYPE="button" VALUE="Show Form Name"
   onClick="this.form.text1.value=this.form.name">
<INPUT NAME="button2" TYPE="button" VALUE="Show Form Elements"
   onClick="showElements(this.form)">
</FORM>
```

The alert dialog box displays the following text:

```
JavaScript Alert:
Form Elements of form myForm:
text1
button1
button2
```

Example 2. The following example uses an object reference, rather than the `this` keyword, to refer to a form. The code returns a reference to `myForm`, which is a form containing `myTextareaObject`.

```
document.myForm.myTextareaObject.form
```

See also `Form`

handleEvent

Invokes the handler for the specified event.

Method of	`Textarea`
Implemented in	JavaScript 1.2

Syntax `handleEvent(`*`event`*`)`

Parameters

`event`	The name of an event for which the object has an event handler.

Description For information on handling events, see the *Client-Side JavaScript Guide*.

name

A string specifying the name of this object.

Property of	`Textarea`
Implemented in	JavaScript 1.0

Security **JavaScript 1.1.** This property is tainted by default. For information on data tainting, see the *Client-Side JavaScript Guide*.

Description The `name` property initially reflects the value of the `NAME` attribute. Changing the `name` property overrides this setting. The `name` property is not displayed on-screen; it is used to refer to the objects programmatically.

If multiple objects on the same form have the same `NAME` attribute, an array of the given name is created automatically. Each element in the array represents an individual `Form` object. Elements are indexed in source order starting at 0. For example, if two `Text` elements and a `Textarea` element on the same form have their `NAME` attribute set to `"myField"`, an array with the elements `myField[0]`, `myField[1]`, and `myField[2]` is created. You need to be aware of this situation in your code and know whether `myField` refers to a single element or to an array of elements.

Examples In the following example, the `valueGetter` function uses a `for` loop to iterate over the array of elements on the `valueTest` form. The `msgWindow` window displays the names of all the elements on the form:

```
newWindow=window.open("http://home.netscape.com")

function valueGetter() {
   var msgWindow=window.open("")
   for (var i = 0; i < newWindow.document.valueTest.elements.length; i++) {
      msgWindow.document.write(newWindow.document.valueTest.elements[i].name + "<BR>")
   }
}
```

select

Selects the input area of the object.

Method of	`Textarea`
Implemented in	JavaScript 1.0

Syntax `select()`

Parameters None

Description Use the `select` method to highlight the input area of a textarea field. You can use the `select` method with the `focus` method to highlight the field and position the cursor for a user response. This makes it easy for the user to replace all the text in the field.

Examples The following example uses an `onClick` event handler to move the focus to a textarea field and select that field for changing:

```
<FORM NAME="myForm">
<B>Last name: </B><INPUT TYPE="text" NAME="lastName" SIZE=20 VALUE="Pigman">
<BR><B>First name: </B><INPUT TYPE="text" NAME="firstName" SIZE=20 VALUE="Victoria">
<BR><B>Description:</B>
<BR><TEXTAREA NAME="desc" ROWS=3 COLS=40>An avid scuba diver.</TEXTAREA>
<BR><BR>
<INPUT TYPE="button" VALUE="Change description"
   onClick="this.form.desc.select();this.form.desc.focus();">
</FORM>
```

See also `Textarea.blur`, `Textarea.focus`

type

For all `Textarea` objects, the value of the `type` property is `"textarea"`. This property specifies the form element's type.

Property of	`Textarea`
Read-only	
Implemented in	JavaScript 1.1

Examples The following example writes the value of the `type` property for every element on a form.

```
for (var i = 0; i < document.form1.elements.length; i++) {
   document.writeln("<BR>type is " + document.form1.elements[i].type)
}
```

value

A string that initially reflects the `VALUE` attribute.

Property of	`Textarea`
Implemented in	JavaScript 1.0

Security **JavaScript 1.1.** This property is tainted by default. For information on data tainting, see the *Client-Side JavaScript Guide.*

Description This string is displayed in the textarea field. The value of this property changes when a user or a program modifies the field.

You can set the `value` property at any time. The display of the `Textarea` object updates immediately when you set the `value` property.

Examples The following function evaluates the `value` property of a group of buttons and displays it in the `msgWindow` window:

```
function valueGetter() {
   var msgWindow=window.open("")
   msgWindow.document.write("submitButton.value is " +
      document.valueTest.submitButton.value + "<BR>")
   msgWindow.document.write("resetButton.value is " +
      document.valueTest.resetButton.value + "<BR>")
   msgWindow.document.write("blurb.value is " +
      document.valueTest.blurb.value + "<BR>")
   msgWindow.document.close()
}
```

This example displays the following:

```
submitButton.value is Query Submit
resetButton.value is Reset
blurb.value is Tropical waters contain all sorts of cool fish,
such as the harlequin ghost pipefish, dragonet, and cuttlefish.
A cuttlefish looks much like a football wearing a tutu and a mop.
```

The previous example assumes the buttons have been defined as follows:

```
<INPUT TYPE="submit" NAME="submitButton">
<INPUT TYPE="reset" NAME="resetButton">
<TEXTAREA NAME="blurb" rows=3 cols=60>
Tropical waters contain all sorts of cool fish,
such as the harlequin ghost pipefish, dragonet, and cuttlefish.
A cuttlefish looks much like a football wearing a tutu and a mop.
</TEXTAREA>
```

See also `Textarea.defaultValue`

window

Represents a browser window or frame. This is the top-level object for each `document`, `Location`, and `History` object group.

Client-side object.

Implemented in	JavaScript 1.0 JavaScript 1.1: added `closed`, `history`, and `opener` properties; added `blur`, `focus`, and `scroll` methods; added `onBlur`, `onError`, and `onFocus` event handlers JavaScript 1.2: added `crypto`, `innerHeight`, `innerWidth`, `locationbar`, `menubar`, `offscreenBuffering`, `outerHeight`, `outerWidth`, `pageXOffset`, `pageYOffset`, `personalbar`, `screenX`, `screenY`, `scrollbars`, `statusbar`, and `toolbar` properties; added `atob`, `back`, `btoa`, `captureEvents`, `clearInterval`, `crypto.random`, `crypto.signText`, `disableExternalCapture`, `enableExternalCapture`, `find`, `forward`, `handleEvent`, `home`, `moveBy`, `moveTo`, `releaseEvents`, `resizeBy`, `resizeTo`, `routeEvent`, `scrollBy`, `scrollTo`, `setHotKeys`, `setInterval`, `setResizable`, `setZOptions`, and `stop` methods; deprecated `scroll` method

Created by The JavaScript runtime engine creates a `window` object for each `BODY` or `FRAMESET` tag. It also creates a `window` object to represent each frame defined in a `FRAME` tag. In addition, you can create other windows by calling the `window.open` method. For details on defining a window, see `open`.

Event handlers

- `onBlur`
- `onDragDrop`
- `onError`
- `onFocus`
- `onLoad`
- `onMove`
- `onResize`
- `onUnload`

In JavaScript 1.1, on some platforms, placing an `onBlur` or `onFocus` event handler in a `FRAMESET` tag has no effect.

Description The `window` object is the top-level object in the JavaScript client hierarchy. A `window` object can represent either a top-level window or a frame inside a frameset. As a matter of convenience, you can think about a `Frame` object as a `window` object that isn't a top-level window. However, there is not really a separate `Frame` class; these objects really are `window` objects, with a very few minor differences:

- For a top-level window, the `parent` and `top` properties are references to the window itself. For a frame, the `top` refers to the topmost browser window, and `parent` refers to the parent window of the current window.
- For a top-level window, setting the `defaultStatus` or `status` property sets the text appearing in the browser status line. For a frame, setting these properties only sets the status line text when the cursor is over the frame.
- The `close` method is not useful for windows that are frames.
- To create an `onBlur` or `onFocus` event handler for a frame, you must set the `onblur` or `onfocus` property and specify it in all lowercase (you cannot specify it in HTML).
- If a `FRAME` tag contains `SRC` and `NAME` attributes, you can refer to that frame from a sibling frame by using `parent.frameName` or `parent.frames[index]`. For example, if the fourth frame in a set has `NAME="homeFrame"`, sibling frames can refer to that frame using `parent.homeFrame` or `parent.frames[3]`.

For all windows, the `self` and `window` properties of a `window` object are synonyms for the current window, and you can optionally use them to refer to the current window. For example, you can close the current window by calling the `close` method of either `window` or `self`. You can use these properties to make your code more readable or to disambiguate the property reference `self.status` from a form called `status`. See the properties and methods listed below for more examples.

Because the existence of the current window is assumed, you do not have to refer to the name of the window when you call its methods and assign its properties. For example, `status="Jump to a new location"` is a valid property assignment, and `close()` is a valid method call.

However, when you open or close a window within an event handler, you must specify `window.open()` or `window.close()` instead of simply using `open()` or `close()`. Due to the scoping of static objects in JavaScript, a call to `close()` without specifying an object name is equivalent to `document.close()`.

For the same reason, when you refer to the `location` object within an event handler, you must specify `window.location` instead of simply using `location`. A call to `location` without specifying an object name is equivalent to `document.location`, which is a synonym for `document.URL`.

You can refer to a window's `Frame` objects in your code by using the `frames` array. In a window with a `FRAMESET` tag, the `frames` array contains an entry for each frame.

A windows lacks event handlers until HTML that contains a `BODY` or `FRAMESET` tag is loaded into it.

Property Summary

Property	Description
`closed`	Specifies whether a window has been closed.
`crypto`	An object which allows access Navigator's encryption features.
`defaultStatus`	Reflects the default message displayed in the window's status bar.
`document`	Contains information on the current document, and provides methods for displaying HTML output to the user.
`frames`	An array reflecting all the frames in a window.
`history`	Contains information on the URLs that the client has visited within a window.
`innerHeight`	Specifies the vertical dimension, in pixels, of the window's content area.
`innerWidth`	Specifies the horizontal dimension, in pixels, of the window's content area.
`length`	The number of frames in the window.
`location`	Contains information on the current URL.
`locationbar`	Represents the browser window's location bar.
`menubar`	Represents the browser window's menu bar.

Property	Description
`name`	A unique name used to refer to this window.
`offscreenBuffering`	Specifies whether updates to a window are performed in an offscreen buffer.
`opener`	Specifies the window name of the calling document when a window is opened using the `open` method
`outerHeight`	Specifies the vertical dimension, in pixels, of the window's outside boundary.
`outerWidth`	Specifies the horizontal dimension, in pixels, of the window's outside boundary.
`pageXOffset`	Provides the current x-position, in pixels, of a window's viewed page.
`pageYOffset`	Provides the current y-position, in pixels, of a window's viewed page.
`parent`	A synonym for a window or frame whose frameset contains the current frame.
`personalbar`	Represents the browser window's personal bar (also called the directories bar).
`screenX`	Specifies the x-coordinate of the left edge of a window.
`screenY`	Specifies the y-coordinate of the top edge of a window.
`scrollbars`	Represents the browser window's scroll bars.
`self`	A synonym for the current window.
`status`	Specifies a priority or transient message in the window's status bar.
`statusbar`	Represents the browser window's status bar.
`toolbar`	Represents the browser window's toolbar.
`top`	A synonym for the topmost browser window.
`window`	A synonym for the current window.

Method Summary

Method	Description
`alert`	Displays an Alert dialog box with a message and an OK button.
`atob`	Decodes a string of data which has been encoded using base-64 encoding.
`back`	Undoes the last history step in any frame within the top-level window.
`blur`	Removes focus from the specified object.
`btoa`	Creates a base-64 encoded string.
`captureEvents`	Sets the window or document to capture all events of the specified type.
`clearInterval`	Cancels a timeout that was set with the `setInterval` method.
`clearTimeout`	Cancels a timeout that was set with the `setTimeout` method.
`close`	Closes the specified window.
`confirm`	Displays a Confirm dialog box with the specified message and OK and Cancel buttons.
`crypto.random`	Returns a pseudo-random string whose length is the specified number of bytes.
`crypto.signText`	Returns a string of encoded data which represents a signed object.
`disableExternalCapture`	Disables external event capturing set by the `enableExternalCapture` method.
`enableExternalCapture`	Allows a window with frames to capture events in pages loaded from different locations (servers).
`find`	Finds the specified text string in the contents of the specified window.
`focus`	Gives focus to the specified object.
`forward`	Loads the next URL in the history list.
`handleEvent`	Invokes the handler for the specified event.
`home`	Points the browser to the URL specified in preferences as the user's home page.

Method	Description
moveBy	Moves the window by the specified amounts.
moveTo	Moves the top-left corner of the window to the specified screen coordinates.
open	Opens a new web browser window.
print	Prints the contents of the window or frame.
prompt	Displays a Prompt dialog box with a message and an input field.
releaseEvents	Sets the window to release captured events of the specified type, sending the event to objects further along the event hierarchy.
resizeBy	Resizes an entire window by moving the window's bottom-right corner by the specified amount.
resizeTo	Resizes an entire window to the specified outer height and width.
routeEvent	Passes a captured event along the normal event hierarchy.
scroll	Scrolls a window to a specified coordinate.
scrollBy	Scrolls the viewing area of a window by the specified amount.
scrollTo	Scrolls the viewing area of the window to the specified coordinates, such that the specified point becomes the top-left corner.
setHotKeys	Enables or disables hot keys in a window which does not have menus.
setInterval	Evaluates an expression or calls a function every time a specified number of milliseconds elapses.
setResizable	Specifies whether a user is permitted to resize a window.
setTimeout	Evaluates an expression or calls a function once after a specified number of milliseconds has elapsed.
setZOptions	Specifies the z-order stacking behavior of a window.
stop	Stops the current download.

In addition, this object inherits the `watch` and `unwatch` methods from `Object`.

Examples **Example 1. Windows opening other windows.** In the following example, the document in the top window opens a second window, `window2`, and defines push buttons that open a message window, write to the message window, close the message window, and close `window2`. The `onLoad` and `onUnload` event handlers of the document loaded into `window2` display alerts when the window opens and closes.

`win1.html`, which defines the frames for the first window, contains the following code:

```
<HTML>
<HEAD>
<TITLE>window object example: Window 1</TITLE>
</HEAD>
<BODY BGCOLOR="antiquewhite">
<SCRIPT>
window2=open("win2.html","secondWindow",
   "scrollbars=yes,width=250, height=400")
document.writeln("<B>The first window has no name: "
   + window.name + "</B>")
document.writeln("<BR><B>The second window is named: "
   + window2.name + "</B>")
</SCRIPT>
<FORM NAME="form1">
<P><INPUT TYPE="button" VALUE="Open a message window"
   onClick = "window3=window.open('','messageWindow',
   'scrollbars=yes,width=175, height=300')">
<P><INPUT TYPE="button" VALUE="Write to the message window"
   onClick="window3.document.writeln('Hey there');
   window3.document.close()">
<P><INPUT TYPE="button" VALUE="Close the message window"
   onClick="window3.close()">
<P><INPUT TYPE="button" VALUE="Close window2"
   onClick="window2.close()">
</FORM>
</BODY>
</HTML>
```

`win2.html`, which defines the content for `window2`, contains the following code:

```
<HTML>
<HEAD>
<TITLE>window object example: Window 2</TITLE>
</HEAD>
<BODY BGCOLOR="oldlace"
   onLoad="alert('Message from ' + window.name + ': Hello, World.')"
   onUnload="alert('Message from ' + window.name + ': I\'m closing')">
<B>Some numbers</B>
<UL><LI>one
<LI>two
<LI>three
<LI>four</UL>
</BODY>
</HTML>
```

Example 2. Creating frames. The following example creates two windows, each with four frames. In the first window, the first frame contains push buttons that change the background colors of the frames in both windows. `framset1.html`, which defines the frames for the first window, contains the following code:

```
<HTML>
<HEAD>
<TITLE>Frames and Framesets: Window 1</TITLE>
</HEAD>
<FRAMESET ROWS="50%,50%" COLS="40%,60%"
   onLoad="alert('Hello, World.')">
<FRAME SRC=framcon1.html NAME="frame1">
<FRAME SRC=framcon2.html NAME="frame2">
<FRAME SRC=framcon2.html NAME="frame3">
<FRAME SRC=framcon2.html NAME="frame4">
</FRAMESET>
</HTML>
```

`framset2.html`, which defines the frames for the second window, contains the following code:

```
<HTML>
<HEAD>
<TITLE>Frames and Framesets: Window 2</TITLE>
</HEAD>
<FRAMESET ROWS="50%,50%" COLS="40%,60%">
<FRAME SRC=framcon2.html NAME="frame1">
<FRAME SRC=framcon2.html NAME="frame2">
<FRAME SRC=framcon2.html NAME="frame3">
<FRAME SRC=framcon2.html NAME="frame4">
</FRAMESET>
</HTML>
```

`framcon1.html`, which defines the content for the first frame in the first window, contains the following code:

```
<HTML>
<BODY>
<A NAME="frame1"><H1>Frame1</H1></A>
<P><A HREF="framcon3.htm" target=frame2>Click here</A>
   to load a different file into frame 2.
<SCRIPT>
window2=open("framset2.htm","secondFrameset")
</SCRIPT>
<FORM>
<P><INPUT TYPE="button" VALUE="Change frame2 to teal"
   onClick="parent.frame2.document.bgColor='teal'">
<P><INPUT TYPE="button" VALUE="Change frame3 to slateblue"
   onClick="parent.frames[2].document.bgColor='slateblue'">
<P><INPUT TYPE="button" VALUE="Change frame4 to darkturquoise"
   onClick="top.frames[3].document.bgColor='darkturquoise'">

<P><INPUT TYPE="button" VALUE="window2.frame2 to violet"
   onClick="window2.frame2.document.bgColor='violet'">
<P><INPUT TYPE="button" VALUE="window2.frame3 to fuchsia"
   onClick="window2.frames[2].document.bgColor='fuchsia'">
<P><INPUT TYPE="button" VALUE="window2.frame4 to deeppink"
   onClick="window2.frames[3].document.bgColor='deeppink'">
</FORM>
</BODY>
</HTML>
```

`framcon2.html`, which defines the content for the remaining frames, contains the following code:

```
<HTML>
<BODY>
<P>This is a frame.
</BODY>
</HTML>
```

`framcon3.html`, which is referenced in a `Link` object in `framcon1.html`, contains the following code:

```
<HTML>
<BODY>
<P>This is a frame. What do you think?
</BODY>
</HTML>
```

See also `document`, `Frame`

alert

Displays an Alert dialog box with a message and an OK button.

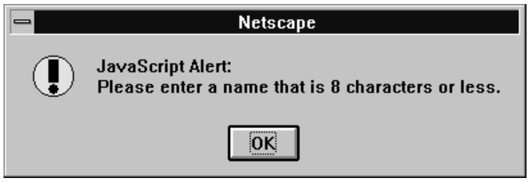
Method of	`window`
Implemented in	JavaScript 1.0

Syntax `alert(`*`message`*`)`

Parameters

`message`	A string.

Description An alert dialog box looks as follows:


Netscape

JavaScript Alert:
Please enter a name that is 8 characters or less.

Use the `alert` method to display a message that does not require a user decision. The `message` argument specifies a message that the dialog box contains.

You cannot specify a title for an alert dialog box, but you can use the `open` method to create your own alert dialog box. See `open`.

Examples In the following example, the `testValue` function checks the name entered by a user in the `Text` object of a form to make sure that it is no more than eight characters in length. This example uses the `alert` method to prompt the user to enter a valid value.

```
function testValue(textElement) {
   if (textElement.length > 8) {
      alert("Please enter a name that is 8 characters or less")
   }
}
```

You can call the `testValue` function in the `onBlur` event handler of a form's `Text` object, as shown in the following example:

```
Name: <INPUT TYPE="text" NAME="userName"
   onBlur="testValue(userName.value)">
```

See also `window.confirm`, `window.prompt`

atob

Decodes a string of data which has been encoded using base-64 encoding.

Method of	`window`
Implemented in	JavaScript 1.2

Syntax `atob(`*`encodedData`*`)`

Parameters

`encodedData`	A string of data which has been created using base-64 encoding.

Description This method decodes a string of data which has been encoded using base-64 encoding. For example, the `window.btoa` method takes a binary string as a parameter and returns a base-64 encoded string.

You can use the `window.btoa` method to encode and transmit data which may otherwise cause communication problems, then transmit it and use the `window.atob` method to decode the data again. For example, you can encode, transmit, and decode characters such as ASCII values 0 through 31.

Examples The following example encodes and decodes the string “Hello, world”.

```
// encode a string
encodedData = btoa("Hello, world");

// decode the string
decodedData = atob(encodedData);
```

See also `window.btoa`

back

Undoes the last history step in any frame within the top-level window; equivalent to the user pressing the browser’s Back button.

Method of	`window`
Implemented in	JavaScript 1.2

Syntax `back()`

Parameters None

Description Calling the `back` method is equivalent to the user pressing the browser’s Back button. That is, `back` undoes the last step anywhere within the top-level window, whether it occurred in the same frame or in another frame in the tree of frames loaded from the top-level window. In contrast, the `history` object's `back` method backs up the *current* window or frame history one step.

For example, consider the following scenario. While in Frame A, you click the Forward button to change Frame A’s content. You then move to Frame B and click the Forward button to change Frame B’s content. If you move back to Frame A and call `FrameA.back()`, the content of Frame B changes (clicking the Back button behaves the same).

If you want to navigate Frame A separately, use `FrameA.history.back()`.

Examples The following custom buttons perform the same operation as the browser’s Back button:

```
<P><INPUT TYPE="button" VALUE="< Go Back"
   onClick="history.back()">
<P><INPUT TYPE="button" VALUE="> Go Back"
   onClick="myWindow.back()">
```

See also `window.forward`, `History.back`

blur

Removes focus from the specified object.

Method of	`window`
Implemented in	JavaScript 1.0

Syntax `blur()`

Parameters None

Description Use the `blur` method to remove focus from a specific window or frame. Removing focus from a window sends the window to the background in most windowing systems.

See also `window.focus`

btoa

Creates a base-64 encoded ASCII string from a string of binary data.

Method of	`window`
Implemented in	JavaScript 1.2

Syntax `btoa(`*`stringToEncode`*`)`

Parameters

`stringToEncode`	An arbitrary binary string to be encoded.

Description This method takes a binary ASCII string as a parameter and returns another ASCII string which has been encoded using base-64 encoding.

You can use this method to encode data which may otherwise cause communication problems, transmit it, then use the `window.atob` method to decode the data again. For example, you can encode characters such as ASCII values 0 through 31.

Examples See `window.atob`.

See also `window.atob`

captureEvents

Sets the window to capture all events of the specified type.

Method of	`window`
Implemented in	JavaScript 1.2

Syntax `captureEvents(`*`eventType1`*` [|`*`eventTypeN...`*`])`

Parameters

`eventType1...` `eventTypeN`	The type of event to be captured. The available event types are discussed in Chapter 3, "Event Handlers."

Security When a window with frames wants to capture events in pages loaded from different locations (servers), you need to use `captureEvents` in a signed script and precede it with `enableExternalCapture`. You must have the `UniversalBrowserWrite` privilege. For more information and an example, see `enableExternalCapture`. For information on security, see the *Client-Side JavaScript Guide*.

See also `captureEvents` works in tandem with `releaseEvents`, `routeEvent`, and `handleEvent`. For more information, see the *Client-Side JavaScript Guide*.

clearInterval

Cancels a timeout that was set with the `setInterval` method.

Method of	`window`
Implemented in	JavaScript 1.2

Syntax `clearInterval(`*`intervalID`*`)`

Parameters

`intervalID`	Timeout setting that was returned by a previous call to the `setInterval` method.

Description See `setInterval`.

Examples See `setInterval`.

See also `window.setInterval`

clearTimeout

Cancels a timeout that was set with the `setTimeout` method.

Method of	`window`
Implemented in	JavaScript 1.0

Syntax `clearTimeout(`*`timeoutID`*`)`

Parameters

`timeoutID`	A timeout setting that was returned by a previous call to the `setTimeout` method.

Description See `setTimeout`.

Examples See `setTimeout`.

See also `window.clearInterval`, `window.setTimeout`

close

Closes the specified window.

Method of	`window`
Implemented in	JavaScript 1.0: closes any window JavaScript 1.1: closes only windows opened by JavaScript JavaScript 1.2: must use signed scripts to unconditionally close a window

Syntax `close()`

Parameters None

Security To unconditionally close a window, you need the `UniversalBrowserWrite` privilege. For information on security, see the *Client-Side JavaScript Guide*.

Description The `close` method closes the specified window. If you call `close` without specifying a `windowReference`, JavaScript closes the current window.

The `close` method closes only windows opened by JavaScript using the `open` method. If you attempt to close any other window, a confirm is generated, which lets the user choose whether the window closes. This is a security feature to prevent "mail bombs" containing `self.close()`. However, if the window has only one document (the current one) in its session history, the close is allowed without any confirm. This is a special case for one-off windows that need to open other windows and then dispose of themselves.

In event handlers, you must specify `window.close()` instead of simply using `close()`. Due to the scoping of static objects in JavaScript, a call to `close()` without specifying an object name is equivalent to `document.close()`.

Examples **Example 1.** Any of the following examples closes the current window:

```
window.close()
self.close()
close()
```

Example 2: Close the main browser window. The following code closes the main browser window.

```
top.opener.close()
```

Example 3. The following example closes the `messageWin` window:

```
messageWin.close()
```

This example assumes that the window was opened in a manner similar to the following:

```
messageWin=window.open("")
```

See also `window.closed`, `window.open`

closed

Specifies whether a window is closed.

Property of	`window`
Read-only	
Implemented in	JavaScript 1.1

Description The `closed` property is a boolean value that specifies whether a window has been closed. When a window closes, the `window` object that represents it continues to exist, and its `closed` property is set to true.

Use `closed` to determine whether a window that you opened, and to which you still hold a reference (from the return value of `window.open`), is still open. Once a window is closed, you should not attempt to manipulate it.

Examples **Example 1.** The following code opens a window, `win1`, then later checks to see if that window has been closed. A function is called depending on whether `win1` is closed.

```
win1=window.open('opener1.html','window1','width=300,height=300')
...
if (win1.closed)
   function1()
   else
   function2()
```

Example 2. The following code determines if the current window's opener window is still closed, and calls the appropriate function.

```
if (window.opener.closed)
   function1()
   else
   function2()
```

See also `window.close`, `window.open`

confirm

Displays a Confirm dialog box with the specified message and OK and Cancel buttons.

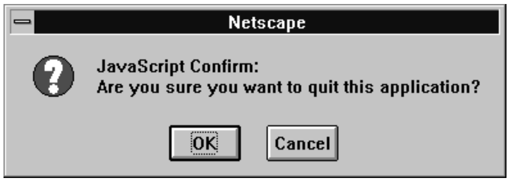
Method of	`window`
Implemented in	JavaScript 1.0

Syntax `confirm(`*`message`*`)`

Parameters

`message`	A string.

Description A confirm dialog box looks as follows:



JavaScript Confirm:
Are you sure you want to quit this application?

Use the `confirm` method to ask the user to make a decision that requires either an OK or a Cancel. The `message` argument specifies a message that prompts the user for the decision. The `confirm` method returns true if the user chooses OK and false if the user chooses Cancel.

You cannot specify a title for a confirm dialog box, but you can use the `open` method to create your own confirm dialog. See `open`.

Examples This example uses the `confirm` method in the `confirmCleanUp` function to confirm that the user of an application really wants to quit. If the user chooses OK, the custom `cleanUp` function closes the application.

```
function confirmCleanUp() {
   if (confirm("Are you sure you want to quit this application?")) {
      cleanUp()
   }
}
```

You can call the `confirmCleanUp` function in the `onClick` event handler of a form's push button, as shown in the following example:

```
<INPUT TYPE="button" VALUE="Quit" onClick="confirmCleanUp()">
```

See also `window.alert`, `window.prompt`

crypto

An object which allows access Navigator's encryption features.

Property of	`window`
Read-only	
Implemented in	JavaScript 1.2

Description The `crypto` object is only available as a property of `window`; it provides access to methods which support Navigator's encryption features.

See also `window.crypto.random`, `window.crypto.signText`

crypto.random

Returns a pseudo-random string whose length is the specified number of bytes.

Method of	`window`
Static	
Implemented in	JavaScript 1.2

Syntax `crypto.random(`*`numberOfBytes`*`)`

Parameters

`numberOfBytes`	The number of bytes of pseudo-random data the method will return.

Description This method generates a random string of data whose length is specified by the *`numberOfBytes`* parameter.

Examples The following function returns a string whose length is 16 bytes.

```
function getRandom() {
   return crypto.random(16)
}
```

See also `Math.random`

crypto.signText

Returns a string of encoded data which represents a signed object.

Method of	`window`
Static	
Implemented in	JavaScript 1.2

Syntax

```
crypto.signText
   (text, selectionStyle [, authority1 [, ... authorityN]])
```

Parameters

`text`	A string evaluating to the text you want a user to sign.
`selectionStyle`	A string evaluating to either of the following: • `ask` specifies that a dialog box will present a user with a list of possible certificates. • `auto` specifies that Navigator automatically selects a certificate from *authority1* through *authorityN*.
`authority1...` `authorityN`	Optional strings evaluating to Certificate Authorities accepted by the server using the signed text.

Description The `signText` method asks a user to validate a *`text`* string by attaching a digital signature to it. If the *`selectionStyle`* parameter is set to `ask`, `signText` displays a dialog box, and a user must interactively select a certificate to validate the text. If *`selectionStyle`* is set to `auto`, Navigator attempts to automatically select a certificate.

Use the `signText` method to submit an encoded signature to a server; the server decodes the signature and verifies it. If `signText` fails, it returns one of the following error codes:

- `error:noMatchingCert` specifies that the user's certificate does not match one of the certificates required by *`authority1`* through *`authorityN`*.
- `error:userCancel` specifies that the user cancelled the signature dialog box without submitting a certificate.
- `error:internalError` specifies that an internal error occurred.

defaultStatus

The default message displayed in the status bar at the bottom of the window.

Property of	`window`
Implemented in	JavaScript 1.0

Security **JavaScript 1.1.** This property is tainted by default. For information on data tainting, see the *Client-Side JavaScript Guide*.

Description The `defaultStatus` message appears when nothing else is in the status bar. Do not confuse the `defaultStatus` property with the `status` property. The `status` property reflects a priority or transient message in the status bar, such as the message that appears when a `mouseOver` event occurs over an anchor.

You can set the `defaultStatus` property at any time. You must return true if you want to set the `defaultStatus` property in the `onMouseOut` or `onMouseOver` event handlers.

Examples In the following example, the `statusSetter` function sets both the `status` and `defaultStatus` properties in an `onMouseOver` event handler:

```
function statusSetter() {
   window.defaultStatus = "Click the link for the Netscape home page"
   window.status = "Netscape home page"
}

<A HREF="http://home.netscape.com"
   onMouseOver = "statusSetter(); return true">Netscape</A>
```

In the previous example, notice that the `onMouseOver` event handler returns a value of true. You must return true to set `status` or `defaultStatus` in an event handler.

See also `window.status`

disableExternalCapture

Disables external event capturing set by the `enableExternalCapture` method.

Method of	`window`
Implemented in	JavaScript 1.2

Syntax `disableExternalCapture()`

Parameters None

Description See `enableExternalCapture`.

document

Contains information on the current document, and provides methods for displaying HTML output to the user.

Property of	`window`
Implemented in	JavaScript 1.0

Description The value of this property is the window's associated `document` object.

enableExternalCapture

Allows a window with frames to capture events in pages loaded from different locations (servers).

Method of	`window`
Implemented in	JavaScript 1.2

Syntax `enableExternalCapture()`

Parameters None

Description Use this method in a signed script requesting `UniversalBrowserWrite` privileges, and use it before calling the `captureEvents` method.

If Communicator sees additional scripts that cause the set of principals in effect for the container to be downgraded, it disables external capture of events. Additional calls to `enableExternalCapture` (after acquiring the `UniversalBrowserWrite` privilege under the reduced set of principals) can be made to enable external capture again.

Examples In the following example, the window is able to capture all `Click` events that occur across its frames.

```
<SCRIPT ARCHIVE="myArchive.jar" ID="2">
function captureClicks() {
   netscape.security.PrivilegeManager.enablePrivilege(
      "UniversalBrowserWrite");
   enableExternalCapture();
   captureEvents(Event.CLICK);
   ...
}
</SCRIPT>
```

See also `window.disableExternalCapture`, `window.captureEvents`

find

Finds the specified text string in the contents of the specified window.

Method of	`window`
Implemented in	JavaScript 1.2

Syntax `find([`*`string`*`[, `*`caseSensitive`*`, `*`backward`*`]])`

Parameters

`string`	The text string for which to search.
`caseSensitive`	Boolean value. If true, specifies a case-sensitive search. If you supply this parameter, you must also supply `backward`.
`backward`	Boolean. If true, specifies a backward search. If you supply this parameter, you must also supply `casesensitive`.

Returns true if the string is found; otherwise, false.

Description When a string is specified, the browser performs a case-insensitive, forward search. If a string is not specified, the method displays the Find dialog box, allowing the user to enter a search string.

focus

Gives focus to the specified object.

Method of	`window`
Implemented in	JavaScript 1.1

Syntax `focus()`

Parameters None

Description Use the `focus` method to navigate to a specific window or frame, and give it focus. Giving focus to a window brings the window forward in most windowing systems.

In JavaScript 1.1, on some platforms, the `focus` method gives focus to a frame but the focus is not visually apparent (for example, the frame's border is not darkened).

Examples In the following example, the `checkPassword` function confirms that a user has entered a valid password. If the password is not valid, the `focus` method returns focus to the `Password` object and the `select` method highlights it so the user can reenter the password.

```
function checkPassword(userPass) {
   if (badPassword) {
      alert("Please enter your password again.")
      userPass.focus()
      userPass.select()
   }
}
```

This example assumes that the `Password` object is defined as

```
<INPUT TYPE="password" NAME="userPass">
```

See also `window.blur`

forward

Points the browser to the next URL in the current history list; equivalent to the user pressing the browser's Forward button

Method of	`window`
Implemented in	JavaScript 1.2

Syntax

```
history.forward()

forward()
```

Parameters None

Description This method performs the same action as a user choosing the Forward button in the browser. The `forward` method is the same as `history.go(1)`.

When used with the Frame object, `forward` behaves as follows: While in Frame A, you click the Back button to change Frame A's content. You then move to Frame B and click the Back button to change Frame B's content. If you move back to Frame A and call `FrameA.forward()`, the content of Frame B changes (clicking the Forward button behaves the same). If you want to navigate Frame A separately, use `FrameA.history.forward()`.

Examples The following custom buttons perform the same operation as the browser's Forward button:

```
<P><INPUT TYPE="button" VALUE="< Go Forth"
   onClick="history.forward()">
<P><INPUT TYPE="button" VALUE="> Go Forth"
   onClick="myWindow.forward()">
```

See also `window.back`

frames

An array of objects corresponding to child frames (created with the `FRAME` tag) in source order.

Property of	`window`
Read-only	
Implemented in	JavaScript 1.0

You can refer to the child frames of a window by using the `frames` array. This array contains an entry for each child frame (created with the `FRAME` tag) in a window containing a `FRAMESET` tag; the entries are in source order. For example, if a window contains three child frames whose `NAME` attributes are `fr1`, `fr2`, and `fr3`, you can refer to the objects in the `images` array either as:

```
parent.frames["fr1"]
parent.frames["fr2"]
parent.frames["fr3"]
```

or as:

```
parent.frames[0]
parent.frames[1]
parent.frames[2]
```

You can find out how many child frames the window has by using the `length` property of the `window` itself or of the `frames` array.

The value of each element in the `frames` array is `<object nameAttribute>`, where `nameAttribute` is the `NAME` attribute of the frame.

handleEvent

Invokes the handler for the specified event.

Method of	`window`
Implemented in	JavaScript 1.2

Syntax `handleEvent(`*`event`*`)`

Parameters

`event`	The name of an event for which the specified object has an event handler.

Description `handleEvent` works in tandem with `captureEvents`, `releaseEvents`, and `routeEvent`. For more information, see the *Client-Side JavaScript Guide*.

history

Contains information on the URLs that the client has visited within a window.

Property of	`window`
Implemented in	JavaScript 1.1

Description The value of this property is the window's associated `History` object.

home

Points the browser to the URL specified in preferences as the user's home page; equivalent to the user pressing the browser's Home button.

Method of	`window`
Implemented in	JavaScript 1.2

Syntax `home()`

Parameters None

Description This method performs the same action as a user choosing the Home button in the browser.

innerHeight

Specifies the vertical dimension, in pixels, of the window's content area.

Property of	`window`
Implemented in	JavaScript 1.2

Description To create a window smaller than 100 x 100 pixels, set this property in a signed script.

Security To set the inner height of a window to a size smaller than 100 x 100 or larger than the screen can accommodate, you need the `UniversalBrowserWrite` privilege. For information on security, see the *Client-Side JavaScript Guide*.

See also `window.innerWidth`, `window.outerHeight`, `window.outerWidth`

innerWidth

Specifies the horizontal dimension, in pixels, of the window's content area.

Property of	`window`
Implemented in	JavaScript 1.2

Description To create a window smaller than 100 x 100 pixels, set this property in a signed script.

Security To set the inner width of a window to a size smaller than 100 x 100 or larger than the screen can accommodate, you need the `UniversalBrowserWrite` privilege. For information on security, see the *Client-Side JavaScript Guide*.

See also `window.innerHeight`, `window.outerHeight`, `window.outerWidth`

length

The number of child frames in the window.

Property of	`window`
Read-only	
Implemented in	JavaScript 1.0

Description This property gives you the same result as using the `length` property of the `frames` array.

location

Contains information on the current URL.

Property of `window`

Implemented in JavaScript 1.0

Description The value of this property is the window's associated `Location` object.

locationbar

Represents the browser window's location bar (the region containing the bookmark and URL areas).

Property of `window`

Implemented in JavaScript 1.2

Description The value of the `locationbar` property itself has one property, `visible`. If true, the location bar is visible; if false, it is hidden.

Security Setting the value of the location bar's `visible` property requires the `UniversalBrowserWrite` privilege. For information on security, see the *Client-Side JavaScript Guide*.

Examples The following example would make the referenced window "chromeless" (chromeless windows lack toolbars, scrollbars, status areas, and so on, much like a dialog box) by hiding most of the user interface toolbars:

```
self.menubar.visible=false;
self.toolbar.visible=false;
self.locationbar.visible=false;
self.personalbar.visible=false;
self.scrollbars.visible=false;
self.statusbar.visible=false;
```

menubar

Represents the browser window's menu bar. This region contains the browser's drop-down menus such as File, Edit, View, Go, Communicator, and so on.

Property of	`window`
Implemented in	JavaScript 1.2

Description The value of the `menubar` property itself has one property, `visible`. If true, the menu bar is visible; if false, it is hidden.

Security Setting the value of the menu bar's `visible` property requires the `UniversalBrowserWrite` privilege. For information on security, see the *Client-Side JavaScript Guide*.

Examples The following example would make the referenced window "chromeless" (chromeless windows lack toolbars, scrollbars, status areas, and so on, much like a dialog box) by hiding most of the user interface toolbars:

```
self.menubar.visible=false;
self.toolbar.visible=false;
self.locationbar.visible=false;
self.personalbar.visible=false;
self.scrollbars.visible=false;
self.statusbar.visible=false;
```

moveBy

Moves the window relative to its current position, moving the specified number of pixels.

Method of	`window`
Implemented in	JavaScript 1.2

Syntax `moveBy(`*`horizontal`*`, `*`vertical`*`)`

Parameters

`horizontal`	The number of pixels by which to move the window horizontally.
`vertical`	The number of pixels by which to move the window vertically.

Description This method moves the window by adding or subtracting the specified number of pixels to the current location.

Security Exceeding any of the boundaries of the screen (to hide some or all of a window) requires signed JavaScript, so a window won't move past the screen boundaries. You need the `UniversalBrowserWrite` privilege for this. For information on security, see the *Client-Side JavaScript Guide*.

Examples: To move the current window 5 pixels up towards the top of the screen (x-axis), and 10 pixels towards the right (y-axis) of the current window position, use this statement:

```
self.moveBy(-5,10); // relative positioning
```

See also `window.moveTo`

moveTo

Moves the top-left corner of the window to the specified screen coordinates.

Method of	`window`
Implemented in	JavaScript 1.2

Syntax `moveTo(`*`x-coordinate`*`, `*`y-coordinate`*`)`

Parameters

`x-coordinate`	The left edge of the window in screen coordinates.
`y-coordinate`	The top edge of the window in screen coordinates.

Description This method moves the window to the absolute pixel location indicated by its parameters. The origin of the axes is at absolute position (0,0); this is the upper left-hand corner of the display.

Security Exceeding any of the boundaries of the screen (to hide some or all of a window) requires signed JavaScript, so a window won't move past the screen boundaries. You need the `UniversalBrowserWrite` privilege for this. For information on security, see the *Client-Side JavaScript Guide*.

Examples: To move the current window to 25 pixels from the top boundary of the screen (x-axis), and 10 pixels from the left boundary of the screen (y-axis), use this statement:

```
self.moveTo(25,10); // absolute positioning
```

See also `window.moveBy`

name

A string specifying the window's name.

Property of `window`

Read-only (2.0); Modifiable (later versions)

Implemented in JavaScript 1.0

Security **JavaScript 1.1.** This property is tainted by default. For information on data tainting, see the *Client-Side JavaScript Guide*.

Description In JavaScript 1.0, `NAME` was a read-only property. In later versions, this property is modifiable by your code. This allows you to assign a name to a top-level window.

Examples In the following example, the first statement creates a window called `netscapeWin`. The second statement displays the value `"netscapeHomePage"` in the Alert dialog box, because `"netscapeHomePage"` is the value of the `windowName` argument of `netscapeWin`.

```
netscapeWin=window.open("http://home.netscape.com","netscapeHomePage")
alert(netscapeWin.name)
```

offscreenBuffering

Specifies whether window updates are performed in an offscreen buffer.

Property of `window`

Implemented in JavaScript 1.2

Description By default, Navigator automatically determines whether updates to a window are performed in an offscreen buffer and then displayed in a window. You can either prevent buffering completely or require Navigator to buffer updates by setting `offscreenBuffering` to either `false` or `true`, respectively.

Buffering can reduce the flicker that occurs during window updates, but it requires additional system resources.

open

Opens a new web browser window.

Method of	`window`
Implemented in	JavaScript 1.0 JavaScript 1.2: added several new `windowFeatures`

Syntax `open(`*`URL`*`, `*`windowName`*`[, `*`windowFeatures`*`])`

Parameters

`URL`	A string specifying the URL to open in the new window. See the `Location` object for a description of the URL components.
`windowName`	A string specifying the window name to use in the `TARGET` attribute of a `FORM` or `A` tag. `windowName` can contain only alphanumeric or underscore (_) characters.
`windowFeatures`	A string containing a comma-separated list determining whether or not to create various standard window features. These options are described in the following section.

Description In event handlers, you must specify `window.open()` instead of simply using `open()`. Due to the scoping of static objects in JavaScript, a call to `open()` without specifying an object name is equivalent to `document.open()`.

The `open` method opens a new Web browser window on the client, similar to choosing New, then Navigator Window from the Navigator File menu. The `URL` argument specifies the URL contained by the new window. If `URL` is an empty string, a new, empty window is created.

You can use `open` on an existing window, and if you pass the empty string for the URL, you will get a reference to the existing window, but not load anything into it. You can, for example, then look for properties in the window.

`windowFeatures` is an optional string containing a comma-separated list of options for the new window (do not include any spaces in this list). After a window is open, you cannot use JavaScript to change the `windowFeatures.` You can specify the following features:

Table 1.4 Optional features to specify for a new window.

windowFeatures	Description
`alwaysLowered`	(JavaScript 1.2) If yes, creates a new window that floats below other windows, whether it is active or not. This is a secure feature and must be set in signed scripts.
`alwaysRaised`	(JavaScript 1.2) If yes, creates a new window that floats on top of other windows, whether it is active or not. This is a secure feature and must be set in signed scripts.
`dependent`	(JavaScript 1.2) If yes, creates a new window as a child of the current window. A dependent window closes when its parent window closes. On Windows platforms, a dependent window does not show on the task bar.
`directories`	If yes, creates the standard browser directory buttons, such as What's New and What's Cool.
`height`	(JavaScript 1.0 and 1.1) Specifies the height of the window in pixels.
`hotkeys`	(JavaScript 1.2) If no (or 0), disables most hotkeys in a new window that has no menu bar. The security and quit hotkeys remain enabled.
`innerHeight`	(JavaScript 1.2) Specifies the height, in pixels, of the window's content area. To create a window smaller than 100 x 100 pixels, set this feature in a signed script. This feature replaces `height`, which remains for backwards compatibility.
`innerWidth`	(JavaScript 1.2) Specifies the width, in pixels, of the window's content area. To create a window smaller than 100 x 100 pixels, set this feature in a signed script. This feature replaces `width`, which remains for backwards compatibility.
`location`	If yes, creates a Location entry field.
`menubar`	If yes, creates the menu at the top of the window.
`outerHeight`	(JavaScript 1.2) Specifies the vertical dimension, in pixels, of the outside boundary of the window. To create a window smaller than 100 x 100 pixels, set this feature in a signed script.
`personalbar`	(JavaScript 1.2) If yes, creates the Personal Toolbar, which displays buttons from the user's Personal Toolbar bookmark folder.
`resizable`	If yes, allows a user to resize the window.

Table 1.4 Optional features to specify for a new window.

windowFeatures	Description
`screenX`	(JavaScript 1.2) Specifies the distance the new window is placed from the left side of the screen. To place a window offscreen, set this feature in a signed scripts.
`screenY`	(JavaScript 1.2) Specifies the distance the new window is placed from the top of the screen. To place a window offscreen, set this feature in a signed scripts.
`scrollbars`	If yes, creates horizontal and vertical scrollbars when the Document grows larger than the window dimensions.
`status`	If yes, creates the status bar at the bottom of the window.
`titlebar`	(JavaScript 1.2) If yes, creates a window with a title bar. To set the titlebar to no, set this feature in a signed script.
`toolbar`	If yes, creates the standard browser toolbar, with buttons such as Back and Forward.
`width`	(JavaScript 1.0 and 1.1) Specifies the width of the window in pixels.
`z-lock`	(JavaScript 1.2) If yes, creates a new window that does not rise above other windows when activated. This is a secure feature and must be set in signed scripts.

Many of these features (as noted above) can either be yes or no. For these features, you can use 1 instead of yes and 0 instead of no. If you want to turn a feature on, you can also simply list the feature name in the `windowFeatures` string.

If `windowName` does not specify an existing window and you do not supply the `windowFeatures` parameter, all of the features which have a yes/no choice are yes by default. However, if you do supply the `windowFeatures` parameter, then the `titlebar` and `hotkeys` are still yes by default, but the other features which have a yes/no choice are no by default.

For example, all of the following statements turn on the toolbar option and turn off all other Boolean options:

```
open("", "messageWindow", "toolbar")
open("", "messageWindow", "toolbar=yes")
open("", "messageWindow", "toolbar=1")
```

The following statement turn on the location and directories options and turns off all other Boolean options:

```
open("", "messageWindow", "toolbar,directories=yes")
```

How the `alwaysLowered`, `alwaysRaised`, and `z-lock` features behave depends on the windowing hierarchy of the platform. For example, on Windows, an `alwaysLowered` or `z-locked` browser window is below all windows in all open applications. On Macintosh, an `alwaysLowered` browser window is below all browser windows, but not necessarily below windows in other open applications. Similarly for an `alwaysRaised` window.

You may use `open` to open a new window and then use `open` on that window to open another window, and so on. In this way, you can end up with a chain of opened windows, each of which has an `opener` property pointing to the window that opened it.

Communicator allows a maximum of 100 windows to be around at once. If you open `window2` from `window1` and then are done with `window1`, be sure to set the `opener` property of `window2` to `null`. This allows JavaScript to garbage collect `window1`. If you do not set the `opener` property to `null`, the `window1` object remains, even though it's no longer really needed.

Security To perform the following operations, you need the `UniversalBrowserWrite` privilege:

- To create a window smaller than 100 x 100 pixels or larger than the screen can accommodate by using `innerWidth`, `innerHeight`, `outerWidth`, and `outerHeight`.
- To place a window off screen by using `screenX` and `screenY`.
- To create a window without a titlebar by using `titlebar`.
- To use `alwaysRaised`, `alwaysLowered`, or `z-lock` for any setting.

For information on security, see the *Client-Side JavaScript Guide*.

Examples **Example 1.** In the following example, the `windowOpener` function opens a window and uses `write` methods to display a message:

```
function windowOpener() {
   msgWindow=window.open("","displayWindow","menubar=yes")
   msgWindow.document.write
      ("<HEAD><TITLE>Message window</TITLE></HEAD>")
   msgWindow.document.write
      ("<CENTER><BIG><B>Hello, world!</B></BIG></CENTER>")
}
```

Example 2. The following is an `onClick` event handler that opens a new client window displaying the content specified in the file `sesame.html`. The window opens with the specified option settings; all other options are false because they are not specified.

```
<FORM NAME="myform">
<INPUT TYPE="button" NAME="Button1" VALUE="Open Sesame!"
   onClick="window.open ('sesame.html', 'newWin',
   'scrollbars=yes,status=yes,width=300,height=300')">
</FORM>
```

See also `window.close`

opener

Specifies the window of the calling document when a window is opened using the `open` method.

Property of	`window`
Implemented in	JavaScript 1.1

Description When a source document opens a destination window by calling the `open` method, the `opener` property specifies the window of the source document. Evaluate the `opener` property from the destination window.

This property persists across document unload in the opened window.

You can change the `opener` property at any time.

You may use `window.open` to open a new window and then use `window.open` on that window to open another window, and so on. In this way, you can end up with a chain of opened windows, each of which has an `opener` property pointing to the window that opened it.

Communicator allows a maximum of 100 windows to be around at once. If you open `window2` from `window1` and then are done with `window1`, be sure to set the `opener` property of `window2` to `null`. This allows JavaScript to garbage collect `window1`. If you do not set the `opener` property to `null`, the `window1` object remains, even though it's no longer really needed.

Examples **Example 1: Close the opener.** The following code closes the window that opened the current window. When the opener window closes, `opener` is unchanged. However, `window.opener.name` then evaluates to undefined.

```
window.opener.close()
```

Example 2: Close the main browser window.

```
top.opener.close()
```

Example 3: Evaluate the name of the opener. A window can determine the name of its opener as follows:

```
document.write("<BR>opener property is " + window.opener.name)
```

Example 4: Change the value of opener. The following code changes the value of the `opener` property to null. After this code executes, you cannot close the opener window as shown in Example 1.

```
window.opener=null
```

Example 5: Change a property of the opener. The following code changes the background color of the window specified by the `opener` property.

```
window.opener.document.bgColor='bisque'
```

See also `window.close`, `window.open`

outerHeight

Specifies the vertical dimension, in pixels, of the window's outside boundary.

Property of	`window`
Implemented in	JavaScript 1.2

Description The outer boundary includes the scroll bars, the status bar, the toolbars, and other "chrome" (window border user interface elements). To create a window smaller than 100 x 100 pixels, set this property in a signed script.

See also `window.innerWidth`, `window.innerHeight`, `window.outerWidth`

outerWidth

Specifies the horizontal dimension, in pixels, of the window's outside boundary.

Property of	`window`
Implemented in	JavaScript 1.2

Description The outer boundary includes the scroll bars, the status bar, the toolbars, and other "chrome" (window border user interface elements). To create a window smaller than 100 x 100 pixels, set this property in a signed script.

See also `window.innerWidth`, `window.innerHeight`, `window.outerHeight`

pageXOffset

Provides the current x-position, in pixels, of a window's viewed page.

Property of	`window`
Read-only	
Implemented in	JavaScript 1.2

Description The `pageXOffset` property provides the current x-position of a page as it relates to the upper-left corner of the window's content area. This property is useful when you need to find the current location of the scrolled page before using `scrollTo` or `scrollBy`.

Examples The following example returns the x-position of the viewed page.

```
x = myWindow.pageXOffset
```

See Also `window.pageYOffset`

pageYOffset

Provides the current y-position, in pixels, of a window's viewed page.

Property of	`window`
Read-only	
Implemented in	JavaScript 1.2

Description The `pageYOffset` property provides the current y-position of a page as it relates to the upper-left corner of the window's content area. This property is useful when you need to find the current location of the scrolled page before using `scrollTo` or `scrollBy`.

Examples The following example returns the y-position of the viewed page.

```
x = myWindow.pageYOffset
```

See also `window.pageXOffset`

parent

The `parent` property is the window or frame whose frameset contains the current frame.

Property of	`window`
Read-only	
Implemented in	JavaScript 1.0

Description This property is only meaningful for frames; that is, windows that are not top-level windows.

The `parent` property refers to the `FRAMESET` window of a frame. Child frames within a frameset refer to sibling frames by using `parent` in place of the window name in one of the following ways:

```
parent.frameName
parent.frames[index]
```

For example, if the fourth frame in a set has `NAME="homeFrame"`, sibling frames can refer to that frame using `parent.homeFrame` or `parent.frames[3]`.

You can use `parent.parent` to refer to the "grandparent" frame or window when a `FRAMESET` tag is nested within a child frame.

The value of the `parent` property is

```
<object nameAttribute>
```

where `nameAttribute` is the `NAME` attribute if the parent is a frame, or an internal reference if the parent is a window.

Examples See examples for `Frame`.

personalbar

Represents the browser window's personal bar (also called the directories bar). This is the region the user can use for easy access to certain bookmarks.

Property of	`window`
Implemented in	JavaScript 1.2

Description The value of the `personalbar` property itself has one property, `visible`. If true, the personal bar is visible; if false, it is hidden.

Security Setting the value of the personal bar's `visible` property requires the `UniversalBrowserWrite` privilege. For information on security, see the *Client-Side JavaScript Guide*.

Examples The following example would make the referenced window "chromeless" (chromeless windows lack toolbars, scrollbars, status areas, and so on, much like a dialog box) by hiding most of the user interface toolbars:

```
self.menubar.visible=false;
self.toolbar.visible=false;
self.locationbar.visible=false;
self.personalbar.visible=false;
self.scrollbars.visible=false;
self.statusbar.visible=false;
```

print

Prints the contents of the window.

Method of	`window`
Implemented in	JavaScript 1.2

Syntax `print()`

Parameters None

prompt

Displays a Prompt dialog box with a message and an input field.

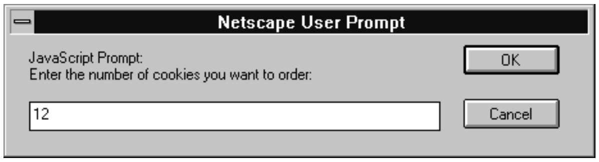
Method of	`window`
Implemented in	JavaScript 1.0

Syntax `prompt(`*`message`*`[, `*`inputDefault`*`])`

Parameters

`message`	A string to be displayed as the message.
`inputDefault`	A string or integer representing the default value of the input field.

Description A prompt dialog box looks as follows:



Use the `prompt` method to display a dialog box that receives user input. If you do not specify an initial value for `inputDefault`, the dialog box displays `<undefined>`.

You cannot specify a title for a prompt dialog box, but you can use the `open` method to create your own prompt dialog. See `open`.

Examples

```
prompt("Enter the number of cookies you want to order:", 12)
```

See also `window.alert`, `window.confirm`

releaseEvents

Sets the window or document to release captured events of the specified type, sending the event to objects further along the event hierarchy.

Method of	`window`
Implemented in	JavaScript 1.2

Note If the original target of the event is a window, the window receives the event even if it is set to release that type of event.

Syntax `releaseEvents(eventType1 [|eventTypeN...])`

Parameters

`eventType1...` `eventTypeN`	The type of event to be captured. The available event types are discussed in Chapter 3, "Event Handlers."

Description `releaseEvents` works in tandem with `captureEvents`, `routeEvent`, and `handleEvent`. For more information, see the *Client-Side JavaScript Guide*.

resizeBy

Resizes an entire window by moving the window's bottom-right corner by the specified amount.

Method of	`window`
Implemented in	JavaScript 1.2

Syntax `resizeBy(horizontal, vertical)`

Parameters

`horizontal`	The number of pixels by which to resize the window horizontally.
`vertical`	The number of pixels by which to resize the window vertically.

Description This method changes the window's dimensions by setting its `outerWidth` and `outerHeight` properties. The upper left-hand corner remains anchored and the lower right-hand corner moves. `resizeBy` moves the window by adding or subtracting the specified number of pixels to that corner's current location.

Security Exceeding any of the boundaries of the screen (to hide some or all of a window) requires signed JavaScript, so a window won't move past the screen boundaries. In addition, windows have an enforced minimum size of 100 x 100 pixels; resizing a window to be smaller than this minimum requires signed JavaScript. You need the `UniversalBrowserWrite` privilege for this. For information on security, see the *Client-Side JavaScript Guide*.

Examples To make the current window 5 pixels narrower and 10 pixels taller than its current dimensions, use this statement:

```
self.resizeBy(-5,10); // relative positioning
```

See also `window.resizeTo`

resizeTo

Resizes an entire window to the specified pixel dimensions.

Method of	`window`
Implemented in	JavaScript 1.2

Syntax `resizeTo(`*`outerWidth`*`, `*`outerHeight`*`)`

Parameters

`outerWidth`	An integer representing the window's width in pixels.
`outerHeight`	An integer representing the window's height in pixels.

Description This method changes the window's dimensions by setting its `outerWidth` and `outerHeight` properties. The upper left-hand corner remains anchored and the lower right-hand corner moves. `resizeBy` moves to the specified position. The origin of the axes is at absolute position (0,0); this is the upper left-hand corner of the display.

Security Exceeding any of the boundaries of the screen (to hide some or all of a window) requires signed JavaScript, so a window won't move past the screen boundaries. In addition, windows have an enforced minimum size of 100 x 100 pixels; resizing a window to be smaller than this minimum requires signed JavaScript. You need the `UniversalBrowserWrite` privilege for this. For information on security, see the *Client-Side JavaScript Guide*.

Examples To make the window 225 pixels wide and 200 pixels tall, use this statement:

```
self.resizeTo(225,200); // absolute positioning
```

See also `window.resizeBy`

routeEvent

Passes a captured event along the normal event hierarchy.

Method of	`window`
Implemented in	JavaScript 1.2

Syntax `routeEvent(`*`event`*`)`

Parameters

`event`	Name of the event to be routed.

Description If a sub-object (document or layer) is also capturing the event, the event is sent to that object. Otherwise, it is sent to its original target.

`routeEvent` works in tandem with `captureEvents`, `releaseEvents`, and `handleEvent`. For more information, see the *Client-Side JavaScript Guide*.

screenX

Specifies the x-coordinate of the left edge of a window.

Property of	`window`
Implemented in	JavaScript 1.2

Security Setting the value of the `screenX` property requires the `UniversalBrowserWrite` privilege. For information on security, see the *Client-Side JavaScript Guide*.

See also `window.screenY`

screenY

Specifies the y-coordinate of the top edge of a window.

Property of	`window`
Implemented in	JavaScript 1.2

Security Setting the value of the `screenY` property requires the `UniversalBrowserWrite` privilege. For information on security, see the *Client-Side JavaScript Guide*.

See also `window.screenX`

scroll

Scrolls a window to a specified coordinate.

Method of	`window`
Implemented in	JavaScript 1.1 JavaScript 1.2: deprecated

Description In JavaScript 1.2, `scroll` is no longer used and has been replaced by `scrollTo`. `scrollTo` extends the capabilities of `scroll`. `scroll` remains for backward compatibility.

scrollbars

Represents the browser window's vertical and horizontal scroll bars for the document area.

Property of	`window`
Implemented in	JavaScript 1.2

Description The value of the `scrollbars` property itself has one property, `visible`. If true, both scrollbars are visible; if false, they are hidden.

Security Setting the value of the scrollbars' `visible` property requires the `UniversalBrowserWrite` privilege. For information on security, see the *Client-Side JavaScript Guide*.

Examples The following example would make the referenced window "chromeless" (chromeless windows lack toolbars, scrollbars, status areas, and so on, much like a dialog box) by hiding most of the user interface toolbars:

```
self.menubar.visible=false;
self.toolbar.visible=false;
self.locationbar.visible=false;
self.personalbar.visible=false;
self.scrollbars.visible=false;
self.statusbar.visible=false;
```

scrollBy

Scrolls the viewing area of a window by the specified amount.

Method of	`window`
Implemented in	JavaScript 1.2

Syntax `scrollBy(`*`horizontal`*`, `*`vertical`*`)`

Parameters

`horizontal`	The number of pixels by which to scroll the viewing area horizontally.
`vertical`	The number of pixels by which to scroll the viewing area vertically.

Description This method scrolls the content in the window if portions that can't be seen exist outside of the window. `scrollBy` scrolls the window by adding or subtracting the specified number of pixels to the current scrolled location.

For this method to have an effect the `visible` property of `window.scrollbars` must be true.

Examples To scroll the current window 5 pixels towards the left and 30 pixels down from the current position, use this statement:

```
self.scrollBy(-5,30); // relative positioning
```

See also `window.scrollTo`

scrollTo

Scrolls the viewing area of the window so that the specified point becomes the top-left corner.

Method of	`window`
Implemented in	JavaScript 1.2

Syntax `scrollTo(`*`x-coordinate`*`, `*`y-coordinate`*`)`

Parameters

`x-coordinate`	An integer representing the x-coordinate of the viewing area in pixels.
`y-coordinate`	An integer representing the y-coordinate of the viewing area in pixels.

Description `scrollTo` replaces `scroll`. `scroll` remains for backward compatibility.

The `scrollTo` method scrolls the content in the window if portions that can't be seen exist outside of the window. For this method to have an effect the `visible` property of `window.scrollbars` must be true.

Examples **Example 1: Scroll the current viewing area.** To scroll the current window to the leftmost boundary and 20 pixels down from the top of the window, use this statement:

```
self.scrollTo(0,20); // absolute positioning
```

Example 2: Scroll a different viewing area. The following code, which exists in one frame, scrolls the viewing area of a second frame. Two `Text` objects let the user specify the x and y coordinates. When the user clicks the Go button, the document in `frame2` scrolls to the specified coordinates.

```
<SCRIPT>
function scrollIt(form) {
   var x = parseInt(form.x.value)
   var y = parseInt(form.y.value)
   parent.frame2.scrollTo(x, y)
}
</SCRIPT>
<BODY>

<FORM NAME="myForm">
<P><B>Specify the coordinates to scroll to:</B>
<BR>Horizontal:
<INPUT TYPE="text" NAME=x VALUE="0" SIZE=4>
<BR>Vertical:
<INPUT TYPE="text" NAME=y VALUE="0" SIZE=4>
<BR><INPUT TYPE="button" VALUE="Go"
   onClick="scrollIt(document.myForm)">
</FORM>
```

See also `window.scrollBy`

self

The `self` property is a synonym for the current window.

Property of	`window`
Read-only	
Implemented in	JavaScript 1.0

Description The `self` property refers to the current window. That is, the value of this property is a synonym for the object itself.

Use the `self` property to disambiguate a `window` property from a form or form element of the same name. You can also use the `self` property to make your code more readable.

The value of the `self` property is

```
<object nameAttribute>
```

where `nameAttribute` is the `NAME` attribute if `self` refers to a frame, or an internal reference if `self` refers to a window.

Examples In the following example, `self.status` is used to set the `status` property of the current window. This usage disambiguates the `status` property of the current window from a form or form element called `status` within the current window.

```
<A HREF=""
   onClick="this.href=pickRandomURL()"
   onMouseOver="self.status='Pick a random URL' ; return true">
Go!</A>
```

setHotKeys

Enables or disables hot keys in a window which does not have menus.

Method of	`window`
Implemented in	JavaScript 1.2

Syntax `setHotKeys(`*`trueOrFalse`*`)`

Parameters

`trueOrFalse`	A Boolean value specifying whether hot keys are enabled: • true enables hot keys • false disables hot keys

Security To enable or disable hot keys, you need the `UniversalBrowserWrite` privilege. For information on security, see the *Client-Side JavaScript Guide*.

Description By default, hot keys are disabled in a window which does not display a menu. With the `setHotKeys` method, you can explicitly enable or disable all hot keys except security and quit, which are always enabled.

You can also specify whether to enable hot keys at the time you create a window when you use the `window.open` method.

See also `window.open`

setInterval

Evaluates an expression or calls a function every time a specified number of milliseconds elapses, until canceled by a call to `clearInterval`.

Method of	`window`
Implemented in	JavaScript 1.2

Syntax

`setInterval(`*`expression`*`, `*`msec`*`)`
`setInterval(`*`function`*`, `*`msec`*`[, `*`arg1`*`[, ..., `*`argN`*`]])`

Parameters

`function`	Any function.
`expression`	A string containing a JavaScript expression. The expression must be quoted; otherwise, `setInterval` calls it immediately. For example, `setInterval("calcnum(3, 2)", 25)`.
`msec`	A numeric value or numeric string, in millisecond units.
`arg1, ..., argn`	The arguments, if any, passed to `function`.

Description The timeouts continue to fire until the associated window or frame is destroyed or the interval is canceled using the `clearInterval` method.

`setInterval` does not stall the script. The script continues immediately (not waiting for the interval to elapse). The call simply schedules a future event.

Examples The following code displays the current time in a `Text` object. In the `startclock` function, the call to the `setInterval` method causes the `showtime` function to be called every second to update the clock. Notice that the `startclock` function and `setInterval` method are each called only one time.

```
<SCRIPT LANGUAGE="JavaScript">
var timerID = null
var timerRunning = false

function stopclock(){
   if(timerRunning)
      clearInterval(timerID)
   timerRunning = false
}
```

```
function startclock(){
   // Make sure the clock is stopped
   stopclock()
   timerID = setInterval("showtime()",1000)
   timerRunning = true
}

function showtime(){
   var now = new Date()
   var hours = now.getHours()
   var minutes = now.getMinutes()
   var seconds = now.getSeconds()
   var timeValue = "" + ((hours > 12) ? hours - 12 : hours)
   timeValue += ((minutes < 10) ? ":0" : ":") + minutes
   timeValue += ((seconds < 10) ? ":0" : ":") + seconds
   timeValue += (hours >= 12) ? " P.M." : " A.M."
   document.clock.face.value = timeValue
}
</SCRIPT>

<BODY onLoad="startclock()">
<FORM NAME="clock" onSubmit="0">
   <INPUT TYPE="text" NAME="face" SIZE=12 VALUE ="">
</FORM>
</BODY>
```

See also `window.clearInterval`, `window.setTimeout`

setResizable

Specifies whether a user is permitted to resize a window.

Method of	`window`
Implemented in	JavaScript 1.2

Syntax `setResizable(`*`trueOrFalse`*`)`

Parameters

`trueOrFalse`	A Boolean value specifying whether a user can resize a window: • true lets a user resize the window • false prevents a user from resizing the window

Description By default, a new Navigator window is resizable. With the `setResizable` method, you can explicitly enable or disable the ability of a user to resize a window. Not all operating systems support this method.

You can also specify whether a window is resizable at the time you create it when you use the `window.open` method.

See also `window.open`

setTimeout

Evaluates an expression or calls a function once after a specified number of milliseconds elapses.

Method of	`window`
Implemented in	JavaScript 1.0: evaluating an expression JavaScript 1.2: calling a function

Syntax

```
setTimeout(expression, msec)
setTimeout(function, msec[, arg1[, ..., argN]])
```

Parameters

`expression`	A string containing a JavaScript expression. The expression must be quoted; otherwise, `setTimeout` calls it immediately. For example, `setTimeout("calcnum(3, 2)", 25)`.
`msec`	A numeric value or numeric string, in millisecond units.
`function`	Any function.
`arg1, ..., argN`	The arguments, if any, passed to `function`.

Description The `setTimeout` method evaluates an expression or calls a function after a specified amount of time. It does not act repeatedly. For example, if a `setTimeout` method specifies five seconds, the expression is evaluated or the function is called after five seconds, not every five seconds. For repetitive timeouts, use the `setInterval` method.

`setTimeout` does not stall the script. The script continues immediately (not waiting for the timeout to expire). The call simply schedules a future event.

Examples **Example 1.** The following example displays an alert message five seconds (5,000 milliseconds) after the user clicks a button. If the user clicks the second button before the alert message is displayed, the timeout is canceled and the alert does not display.

```
<SCRIPT LANGUAGE="JavaScript">
function displayAlert() {
   alert("5 seconds have elapsed since the button was clicked.")
}
</SCRIPT>
<BODY>
<FORM>
Click the button on the left for a reminder in 5 seconds;
click the button on the right to cancel the reminder before
it is displayed.
<P>
<INPUT TYPE="button" VALUE="5-second reminder"
   NAME="remind_button"
   onClick="timerID=setTimeout('displayAlert()',5000)">
<INPUT TYPE="button" VALUE="Clear the 5-second reminder"
   NAME="remind_disable_button"
   onClick="clearTimeout(timerID)">
</FORM>
</BODY>
```

Example 2. The following example displays the current time in a `Text` object. The `showtime` function, which is called recursively, uses the `setTimeout` method to update the time every second.

```
<HEAD>
<SCRIPT LANGUAGE="JavaScript">
<!--
var timerID = null
var timerRunning = false
function stopclock(){
   if(timerRunning)
      clearTimeout(timerID)
   timerRunning = false
}
function startclock(){
   // Make sure the clock is stopped
   stopclock()
   showtime()
}
```

```
function showtime(){
   var now = new Date()
   var hours = now.getHours()
   var minutes = now.getMinutes()
   var seconds = now.getSeconds()
   var timeValue = "" + ((hours > 12) ? hours - 12 : hours)
   timeValue += ((minutes < 10) ? ":0" : ":") + minutes
   timeValue += ((seconds < 10) ? ":0" : ":") + seconds
   timeValue += (hours >= 12) ? " P.M." : " A.M."
   document.clock.face.value = timeValue
   timerID = setTimeout("showtime()",1000)
   timerRunning = true
}
//-->
</SCRIPT>
</HEAD>

<BODY onLoad="startclock()">
<FORM NAME="clock" onSubmit="0">
   <INPUT TYPE="text" NAME="face" SIZE=12 VALUE ="">
</FORM>
</BODY>
```

See also `window.clearTimeout`, `window.setInterval`

setZOptions

Specifies the z-order stacking behavior of a window.

Method of	`window`
Implemented in	JavaScript 1.2

Syntax `setZOptions(`*`windowPosition`*`)`

Parameters

`windowPosition` A string evaluating to any of the following values:

- alwaysRaised creates a new window that floats on top of other windows, whether it is active or not.
- alwaysLowered creates a new window that floats below other windows, whether it is active or not.
- z-lock creates a new window that does not rise above other windows when activated.

Security To set this property, you need the `UniversalBrowserWrite` privilege. For information on security, see the *Client-Side JavaScript Guide*.

Description By default, a Navigator window rises to the top of the z-order when it is activated and moves down in the z-order as other windows are activated. With the `setZOptions` method, you can explicitly specify a window's position in the z-order.

If you do not specify an argument for `setZOptions`, this method restores the default z-order stacking behavior of a Navigator window.

You can also specify the order stacking behavior of a window at the time you create it when you use the `window.open` method.

See also `window.open`

status

Specifies a priority or transient message in the status bar at the bottom of the window, such as the message that appears when a `mouseOver` event occurs over an anchor.

Property of	`window`
Implemented in	JavaScript 1.0

Security **JavaScript 1.1.** This property is tainted by default. For information on data tainting, see the *Client-Side JavaScript Guide*.

Description Do not confuse the `status` property with the `defaultStatus` property. The `defaultStatus` property reflects the default message displayed in the status bar.

You can set the `status` property at any time. You must return true if you want to set the `status` property in the `onMouseOver` event handler.

Examples Suppose you have created a JavaScript function called `pickRandomURL` that lets you select a URL at random. You can use the `onClick` event handler of an anchor to specify a value for the `HREF` attribute of the anchor dynamically, and the `onMouseOver` event handler to specify a custom message for the window in the `status` property:

```
<A HREF=""
   onClick="this.href=pickRandomURL()"
   onMouseOver="self.status='Pick a random URL'; return true">
Go!</A>
```

In the preceding example, the `status` property of the window is assigned to the window's `self` property, as `self.status`.

See also `window.defaultStatus`

statusbar

Represents the browser window's status bar. This is the region containing the security indicator, browser status, and so on.

Property of	`window`
Implemented in	JavaScript 1.2

Description The value of the `statusbar` property itself one property, `visible`. If true, the status bar is visible; if false, it is hidden.

Security Setting the value of the status bar's `visible` property requires the `UniversalBrowserWrite` privilege. For information on security, see the *Client-Side JavaScript Guide*.

Examples The following example would make the referenced window "chromeless" (chromeless windows lack toolbars, scrollbars, status areas, and so on, much like a dialog box) by hiding most of the user interface toolbars:

```
self.menubar.visible=false;
self.toolbar.visible=false;
self.locationbar.visible=false;
self.personalbar.visible=false;
self.scrollbars.visible=false;
self.statusbar.visible=false;
```

stop

Stops the current download.

Method of	`window`
Implemented in	JavaScript 1.2

Syntax `stop()`

Parameters None

Definition This method performs the same action as a user choosing the Stop button in the browser.

toolbar

Represents the browser window's toolbar, containing the navigation buttons, such as Back, Forward, Reload, Home, and so on.

Property of	`window`
Implemented in	JavaScript 1.2

Description The value of the `toolbar` property itself has one property, `visible`. If true, the toolbar is visible; if false, it is hidden.

Security Setting the value of the toolbar's `visible` property requires the `UniversalBrowserWrite` privilege. For information on security, see the *Client-Side JavaScript Guide*.

Examples The following example would make the referenced window "chromeless" (chromeless windows lack toolbars, scrollbars, status areas, and so on, much like a dialog box) by hiding most of the user interface toolbars:

```
self.menubar.visible=false;
self.toolbar.visible=false;
self.locationbar.visible=false;
self.personalbar.visible=false;
self.scrollbars.visible=false;
self.statusbar.visible=false;
```

top

The `top` property is a synonym for the topmost browser window, which is a document window or web browser window.

Property of	`window`
Read-only	
Implemented in	JavaScript 1.0

Description The `top` property refers to the topmost window that contains frames or nested framesets. Use the `top` property to refer to this ancestor window.

The value of the `top` property is

```
<object objectReference>
```

where `objectReference` is an internal reference.

Examples The statement `top.close()` closes the topmost ancestor window.

The statement `top.length` specifies the number of frames contained within the topmost ancestor window. When the topmost ancestor is defined as follows, `top.length` returns three:

```
<FRAMESET COLS="30%,40%,30%">
<FRAME SRC=child1.htm NAME="childFrame1">
<FRAME SRC=child2.htm NAME="childFrame2">
<FRAME SRC=child3.htm NAME="childFrame3">
</FRAMESET>
```

The following example sets the background color of a frame called `myFrame` to red. `myFrame` is a child of the topmost ancestor window.

```
top.myFrame.document.bgColor="red"
```

window

The `window` property is a synonym for the current window or frame.

Property of	`window`
Read-only	
Implemented in	JavaScript 1.0

Description The `window` property refers to the current window or frame. That is, the value of this property is a synonym for the object itself.

Although you can use the `window` property as a synonym for the current frame, your code may be more readable if you use the `self` property. For example, `window.name` and `self.name` both specify the name of the current frame, but `self.name` may be easier to understand (because a frame is not displayed as a separate window).

Use the `window` property to disambiguate a property of the `window` object from a form or form element of the same name. You can also use the `window` property to make your code more readable.

The value of the `window` property is

```
<object nameAttribute>
```

where `nameAttribute` is the `NAME` attribute if `window` refers to a frame, or an internal reference if `window` refers to a window.

Examples In the following example, `window.status` is used to set the `status` property of the current window. This usage disambiguates the `status` property of the current window from a form called "status" within the current window.

```
<A HREF=""
   onClick="this.href=pickRandomURL()"
   onMouseOver="window.status='Pick a random URL' ; return true">
Go!</A>
```

See also `window.self`

Chapter

2

Top-Level Properties and Functions

This chapter contains all JavaScript properties and functions not associated with any object. In the ECMA specification, these properties and functions are referred to as properties and methods of the global object.

The following table summarizes the top-level properties.

Table 2.1 Top-level properties

Property	Description
`Infinity`	A numeric value representing infinity.
`NaN`	A value representing Not-A-Number.
`undefined`	The value undefined.

The following table summarizes the top-level functions.

Table 2.2 Top-level functions

Function	Description
`escape`	Returns the hexadecimal encoding of an argument in the ISO Latin-1 character set; used to create strings to add to a URL.
`eval`	Evaluates a string of JavaScript code without reference to a particular object.

Table 2.2 Top-level functions

Function	Description
`isFinite`	Evaluates an argument to determine whether it is a finite number.
`isNaN`	Evaluates an argument to determine if it is not a number.
`Number`	Converts an object to a number.
`parseFloat`	Parses a string argument and returns a floating-point number.
`parseInt`	Parses a string argument and returns an integer.
`String`	Converts an object to a string.
`taint`	Adds tainting to a data element or script.
`unescape`	Returns the ASCII string for the specified hexadecimal encoding value.
`untaint`	Removes tainting from a data element or script.

escape

Returns the hexadecimal encoding of an argument in the ISO-Latin-1 character set.

Core function

Implemented in	JavaScript 1.0, NES 2.0
ECMA version	ECMA-262 compatible, except for Unicode characters.

Syntax `escape("`*`string`*`")`

Parameters

`string`	A string in the ISO-Latin-1 character set.

Description `escape` is a top-level function and is not associated with any object.

Use the `escape` and `unescape` functions to encode and decode (add property values manually) a Uniform Resource Locator (URL), a Uniform Resource Identifier (URI), or a URI-type string.

The `escape` function encodes special characters in the specified string and returns the new string. It encodes spaces, punctuation, and any other character that is not an ASCII alphanumeric character, with the exception of these characters:

```
* @ - _ + . /
```

Unicode. The `escape` and `unescape` functions do not use Unicode as specified by the ECMA specification. Instead, they use the Internet Engineering Task Force (IETF) guidelines for escaping characters. Within a URI, characters use US-ASCII characters (ISO-Latin-1 character set). A URI is a sequence of characters from the basic Latin alphabet, digits, and a few special characters (for example, / and @). The escape sequences do not support \uXXXX as in Unicode or %uXXXX as specified by ECMA, but %XX, where XX is a 2-digit hexadecimal number (for example, %7E). In URI, characters are represented in octets, as 8-bit bytes.

To allow the `escape` and `unescape` functions to work with Web server-supported URLs and URIs, JavaScript does not use Unicode for these functions.

- `escape` returns the hexadecimal encoding of the specified string in the ISO-Latin-1 character set.
- `unescape` returns the ASCII string, an ISO-Latin-1 character set sequence.

Unicode-specific escape sequences, %uXXXX, are not supported.

Examples **Example 1.** The following example returns `"%26"`:

```
escape("&") // returns "%26"
```

Example 2. The following statement returns a string with encoded characters for spaces, commas, and apostrophes.

```
// returns "The_rain.%20In%20Spain%2C%20Ma%92am"
escape("The_rain. In Spain, Ma'am")
```

See also `unescape`

eval

Evaluates a string of JavaScript code without reference to a particular object.

Core function

Implemented in	JavaScript 1.0
ECMA version	ECMA-262

Syntax `eval(`*`string`*`)`

Parameters

`string`	A string representing a JavaScript expression, statement, or sequence of statements. The expression can include variables and properties of existing objects.

Description `eval` is a top-level function and is not associated with any object.

The argument of the `eval` function is a string. If the string represents an expression, `eval` evaluates the expression. If the argument represents one or more JavaScript statements, `eval` performs the statements. Do not call `eval` to evaluate an arithmetic expression; JavaScript evaluates arithmetic expressions automatically.

If you construct an arithmetic expression as a string, you can use `eval` to evaluate it at a later time. For example, suppose you have a variable `x`. You can postpone evaluation of an expression involving `x` by assigning the string value of the expression, say `"3 * x + 2"`, to a variable, and then calling `eval` at a later point in your script.

If the argument of `eval` is not a string, `eval` returns the argument unchanged. In the following example, the `String` constructor is specified, and `eval` returns a `String` object rather than evaluating the string.

```
eval(new String("2+2”)) // returns a String object containing "2+2"
eval("2+2”)             // returns 4
```

You should not indirectly use the `eval` function by invoking it via a name other than `eval`. For example, you should not use the following code:

```
var x = 2
var y = 4
var myEval = eval
myEval("x + y")
```

Backward Compatibility **JavaScript 1.1.** `eval` is also a method of all objects. This method is described for the `Object` class.

Examples The following examples display output using `document.write`. In server-side JavaScript, you can display the same output by calling the `write` function instead of using `document.write`.

Example 1. In the following code, both of the statements containing `eval` return 42. The first evaluates the string `"x + y + 1"`; the second evaluates the string `"42"`.

```
var x = 2
var y = 39
var z = "42"
eval("x + y + 1") // returns 42
eval(z)           // returns 42
```

Example 2. In the following example, the `getFieldName(n)` function returns the name of the specified form element as a string. The first statement assigns the string value of the third form element to the variable `field`. The second statement uses `eval` to display the value of the form element.

```
var field = getFieldName(3)
document.write("The field named ", field, " has value of ",
   eval(field + ".value"))
```

Example 3. The following example uses `eval` to evaluate the string `str`. This string consists of JavaScript statements that open an Alert dialog box and assign `z` a value of 42 if `x` is five, and assigns 0 to `z` otherwise. When the second statement is executed, `eval` will cause these statements to be performed, and it will also evaluate the set of statements and return the value that is assigned to `z`.

```
var str = "if (x == 5) {alert('z is 42'); z = 42;} else z = 0; "
document.write("<P>z is ", eval(str))
```

Example 4. In the following example, the `setValue` function uses `eval` to assign the value of the variable `newValue` to the text field `textObject`:

```
function setValue (textObject, newValue) {
   eval ("document.forms[0]." + textObject + ".value") = newValue
}
```

Example 5. The following example creates `breed` as a property of the object `myDog`, and also as a variable. The first write statement uses `eval('breed')` without specifying an object; the string `"breed"` is evaluated without regard to any object, and the `write` method displays `"Shepherd"`, which is the value of the `breed` variable. The second write statement uses `myDog.eval('breed')` which specifies the object `myDog`; the string `"breed"` is evaluated with regard to the `myDog` object, and the `write` method displays `"Lab"`, which is the value of the `breed` property of the `myDog` object.

```
function Dog(name,breed,color) {
   this.name=name
   this.breed=breed
   this.color=color
}
myDog = new Dog("Gabby")
myDog.breed="Lab"
var breed='Shepherd'
document.write("<P>" + eval('breed'))
document.write("<BR>" + myDog.eval('breed'))
```

See also `Object.eval` method

Infinity

A numeric value representing infinity.

Core property

Implemented in	JavaScript 1.3 (In previous versions, `Infinity` was defined only as a property of the `Number` object)
ECMA version	ECMA-262

Syntax `Infinity`

Description `Infinity` is a top-level property and is not associated with any object.

The initial value of `Infinity` is `Number.POSITIVE_INFINITY`. The value `Infinity` (positive infinity) is greater than any other number including itself. This value behaves mathematically like infinity; for example, anything multiplied by `Infinity` is `Infinity`, and anything divided by `Infinity` is 0.

See also `Number.NEGATIVE_INFINITY`, `Number.POSITIVE_INFINITY`

isFinite

Evaluates an argument to determine whether it is a finite number.
Core function

Implemented in	JavaScript 1.3
ECMA version	ECMA-262

Syntax `isFinite(`*`number`*`)`

Parameters

`number`	The number to evaluate.

Description `isFinite` is a top-level function and is not associated with any object.

You can use this method to determine whether a number is a finite number. The `isFinite` method examines the number in its argument. If the argument is `NaN`, positive infinity or negative infinity, this method returns `false`, otherwise it returns `true`.

Examples You can check a client input to determine whether it is a finite number.

```
if(isFinite(ClientInput) == true)
{
   /* take specific steps */
}
```

See also `Number.NEGATIVE_INFINITY`, `Number.POSITIVE_INFINITY`

isNaN

Evaluates an argument to determine if it is not a number.
Core function

Implemented in	JavaScript 1.0: Unix only
	JavaScript 1.1, NES 2.0: all platforms
ECMA version	ECMA-262

Syntax `isNaN(`*`testValue`*`)`

Parameters

`testValue`	The value you want to evaluate.

Description `isNaN` is a top-level function and is not associated with any object.

On platforms that support NaN, the `parseFloat` and `parseInt` functions return `NaN` when they evaluate a value that is not a number. `isNaN` returns true if passed `NaN`, and false otherwise.

Examples The following example evaluates `floatValue` to determine if it is a number and then calls a procedure accordingly:

```
floatValue=parseFloat(toFloat)

if (isNaN(floatValue)) {
   notFloat()
} else {
   isFloat()
}
```

See also `Number.NaN`, `parseFloat`, `parseInt`

NaN

A value representing Not-A-Number.

Core property

Implemented in	JavaScript 1.3 (In previous versions, `NaN` was defined only as a property of the `Number` object)
ECMA version	ECMA-262

Syntax `NaN`

Description `NaN` is a top-level property and is not associated with any object.

The initial value of `NaN` is NaN.

`NaN` is always unequal to any other number, including NaN itself; you cannot check for the not-a-number value by comparing to `Number.NaN`. Use the `isNaN` function instead.

Several JavaScript methods (such as the `Number` constructor, `parseFloat`, and `parseInt`) return `NaN` if the value specified in the parameter is not a number.

You might use the `NaN` property to indicate an error condition for a function that should return a valid number.

See also `isNaN`, `Number.NaN`

Number

Converts the specified object to a number.

Core function

Implemented in	JavaScript 1.2, NES 3.0
ECMA version	ECMA-262

Syntax `Number(`*`obj`*`)`

Parameter

`obj`	An object

Description `Number` is a top-level function and is not associated with any object.

When the object is a `Date` object, `Number` returns a value in milliseconds measured from 01 January, 1970 UTC (GMT), positive after this date, negative before.

If `obj` is a string that does not contain a well-formed numeric literal, `Number` returns NaN.

Example The following example converts the `Date` object to a numerical value:

```
d = new Date ("December 17, 1995 03:24:00")
alert (Number(d))
```

This displays a dialog box containing "819199440000."

See also `Number`

parseFloat

Parses a string argument and returns a floating point number.

Core function

Implemented in	JavaScript 1.0: If the first character of the string specified in parseFloat(`string`) cannot be converted to a number, returns `NaN` on Solaris and Irix and 0 on all other platforms. JavaScript 1.1, NES 2.0: Returns `NaN` on all platforms if the first character of the string specified in parseFloat(`string`) cannot be converted to a number.
ECMA version	ECMA-262

Syntax `parseFloat(*string*)`

Parameters

`string`	A string that represents the value you want to parse.

Description `parseFloat` is a top-level function and is not associated with any object.

`parseFloat` parses its argument, a string, and returns a floating point number. If it encounters a character other than a sign (+ or -), numeral (0-9), a decimal point, or an exponent, it returns the value up to that point and ignores that character and all succeeding characters. Leading and trailing spaces are allowed.

If the first character cannot be converted to a number, `parseFloat` returns `NaN`.

For arithmetic purposes, the `NaN` value is not a number in any radix. You can call the `isNaN` function to determine if the result of `parseFloat` is `NaN`. If `NaN` is passed on to arithmetic operations, the operation results will also be `NaN`.

Examples The following examples all return 3.14:

```
parseFloat("3.14")
parseFloat("314e-2")
parseFloat("0.0314E+2")
var x = "3.14"
parseFloat(x)
```

The following example returns `NaN`:

```
parseFloat("FF2")
```

See also `isNaN`, `parseInt`

parseInt

Parses a string argument and returns an integer of the specified radix or base.

Core function

Implemented in	JavaScript 1.0: If the first character of the string specified in parseInt(`string`) cannot be converted to a number, returns `NaN` on Solaris and Irix and 0 on all other platforms. JavaScript 1.1, LiveWire 2.0: Returns `NaN` on all platforms if the first character of the string specified in parseInt(`string`) cannot be converted to a number.
ECMA version	ECMA-262

Syntax `parseInt(`*`string`*`[, `*`radix`*`])`

Parameters

`string`	A string that represents the value you want to parse.
`radix`	An integer that represents the radix of the return value.

Description `parseInt` is a top-level function and is not associated with any object.

The `parseInt` function parses its first argument, a string, and attempts to return an integer of the specified radix (base). For example, a radix of 10 indicates to convert to a decimal number, 8 octal, 16 hexadecimal, and so on. For radixes above 10, the letters of the alphabet indicate numerals greater than 9. For example, for hexadecimal numbers (base 16), A through F are used.

If `parseInt` encounters a character that is not a numeral in the specified radix, it ignores it and all succeeding characters and returns the integer value parsed up to that point. `parseInt` truncates numbers to integer values. Leading and trailing spaces are allowed.

If the radix is not specified or is specified as 0, JavaScript assumes the following:

- If the input `string` begins with `"0x"`, the radix is 16 (hexadecimal).
- If the input `string` begins with `"0"`, the radix is eight (octal).
- If the input `string` begins with any other value, the radix is 10 (decimal).

If the first character cannot be converted to a number, `parseInt` returns `NaN`.

For arithmetic purposes, the `NaN` value is not a number in any radix. You can call the `isNaN` function to determine if the result of `parseInt` is `NaN`. If `NaN` is passed on to arithmetic operations, the operation results will also be `NaN`.

Examples The following examples all return 15:

```
parseInt("F", 16)
parseInt("17", 8)
parseInt("15", 10)
parseInt(15.99, 10)
parseInt("FXX123", 16)
parseInt("1111", 2)
parseInt("15*3", 10)
```

The following examples all return `NaN`:

```
parseInt("Hello", 8)
parseInt("0x7", 10)
parseInt("FFF", 10)
```

Even though the radix is specified differently, the following examples all return 17 because the input `string` begins with `"0x"`.

```
parseInt("0x11", 16)
parseInt("0x11", 0)
parseInt("0x11")
```

See also `isNaN`, `parseFloat`, `Object.valueOf`

String

Converts the specified object to a string.
Core function

Implemented in	JavaScript 1.2, NES 3.0
ECMA version	ECMA-262

Syntax `String(`*`obj`*`)`

Parameter

`obj`	An object.

Description `String` is a top-level function and is not associated with any object.

The `String` method converts the value of any object into a string; it returns the same value as the `toString` method of an individual object.

When the object is a `Date` object, `String` returns a more readable string representation of the date. Its format is: Thu Aug 18 04:37:43 Pacific Daylight Time 1983.

Example The following example converts the `Date` object to a readable string.

```
D = new Date (430054663215)
alert (String(D))
```

This displays a dialog box containing "Thu Aug 18 04:37:43 GMT-0700 (Pacific Daylight Time) 1983."

See also `String`

taint

Adds tainting to a data element or script.

Client-side function

Implemented in	JavaScript 1.1 JavaScript 1.2: removed

Syntax `taint([dataElementName])`

Parameters

`dataElementName`	The property, variable, function, or object to taint. If omitted, taint is added to the script itself.

Description `taint` is a top-level function and is not associated with any object.

Tainting prevents other scripts from passing information that should be secure and private, such as directory structures or user session history. JavaScript cannot pass tainted values on to any server without the end user's permission.

Use `taint` to mark data that otherwise is not tainted.

In some cases, control flow rather than data flow carries tainted information. In these cases, taint is added to the script's window. You can add taint to the script's window by calling `taint` with no arguments.

`taint` does not modify its argument; instead, it returns a marked copy of the value, or, for objects, an unmarked reference to the value.

Examples The following statement adds taint to a property so that a script cannot send it to another server without the end user's permission:

```
taintedStatus=taint(window.defaultStatus)
// taintedStatus now cannot be sent in a URL or form post without
// the end user's permission
```

See also `navigator.taintEnabled`, `untaint`

undefined

The value undefined.
Core property

Implemented in	JavaScript 1.3
ECMA version	ECMA-262

Syntax `undefined`

Description `undefined` is a top-level property and is not associated with any object.

A variable that has not been assigned a value is of type undefined. A method or statement also returns `undefined` if the variable that is being evaluated does not have an assigned value.

You can use `undefined` to determine whether a variable has a value. In the following code, the variable `x` is not defined, and the `if` statement evaluates to true.

```
var x
if(x == undefined) {
   // these statements execute
}
```

`undefined` is also a primitive value.

unescape

Returns the ASCII string for the specified hexadecimal encoding value.
Core function

Implemented in	JavaScript 1.0, NES 1.0
ECMA version	ECMA-262 compatible, except for Unicode characters.

Syntax `unescape(`*`string`*`)`

Parameters

`string`	A string containing characters in the form `"%xx"`, where `xx` is a 2-digit hexadecimal number.

Description `unescape` is a top-level function and is not associated with any object.

The string returned by the `unescape` function is a series of characters in the ISO-Latin-1 character set.

The `escape` and `unescape` methods do not use Unicode as specified by the ECMA specification. For information, see the description of “Unicode” on page 557.

Examples The following example returns `"&"`:

```
unescape("%26")
```

The following example returns `"!#"`:

```
unescape("%21%23")
```

See also `escape`

untaint

Removes tainting from a data element or script.
Client-side function

Implemented in	JavaScript 1.1
	JavaScript 1.2: removed

Syntax `untaint([`*`dataElementName`*`])`

Parameters

`dataElementName`	The property, variable, function, or object to remove tainting from. If omitted, taint is removed from the script itself.

Description `untaint` is a top-level function and is not associated with any object.

Tainting prevents other scripts from passing information that should be secure and private, such as directory structures or user session history. JavaScript cannot pass tainted values on to any server without the end user's permission.

Use `untaint` to clear tainting that marks data that should not to be sent by other scripts to different servers.

A script can untaint only data that originated in that script (that is, only data that has the script's taint code or has the identity (null) taint code). If you use `untaint` with a data element from another server's script (or any data that you cannot untaint), `untaint` returns the data without change or error.

In some cases, control flow rather than data flow carries tainted information. In these cases, taint is added to the script's window. You can remove taint from the script's window by calling `untaint` with no arguments, if the window contains taint only from the current window.

`untaint` does not modify its argument; instead, it returns an unmarked copy of the value, or, for objects, an unmarked reference to the value.

Examples The following statement removes taint from a property so that a script can send it to another server:

```
untaintedStatus=untaint(window.defaultStatus)
// untaintedStatus can now be sent in a URL or form post by other
// scripts
```

See also `navigator.taintEnabled`, `taint`

Chapter

3

Event Handlers

This chapter contains the event handlers that are used with client-side objects in JavaScript to evoke particular actions.

For general information on event handlers, see the *Client-Side JavaScript Guide*.

The following table summarizes the event handlers. The name of an event handler is the name of the event, preceded by "on." For example, the event handler for the `focus` event is `onFocus`.

Table 3.1 Event handlers

Event	Event handler	Description
`Abort`	`onAbort`	Executes JavaScript code when the user aborts the loading of an image.
`Blur`	`onBlur`	Executes JavaScript code when a form element loses focus or when a window or frame loses focus.
`Change`	`onChange`	Executes JavaScript code when a `Select`, `Text`, or `Textarea` field loses focus and its value has been modified
`Click`	`onClick`	Executes JavaScript code when an object on a form is clicked.
`DblClick`	`onDblClick`	Executes JavaScript code when the user double-clicks a form element or a link.

Table 3.1 Event handlers

Event	Event handler	Description
`DragDrop`	`onDragDrop`	Executes JavaScript code when the user drops an object onto the browser window, such as dropping a file.
`Error`	`onError`	Executes JavaScript code when the loading of a document or image causes an error.
`Focus`	`onFocus`	Executes JavaScript code when a window, frame, or frameset receives focus or when a form element receives input focus.
`KeyDown`	`onKeyDown`	Executes JavaScript code when the user depresses a key.
`KeyPress`	`onKeyPress`	Executes JavaScript code when the user presses or holds down a key.
`KeyUp`	`onKeyUp`	Executes JavaScript code when the user releases a key.
`Load`	`onLoad`	Executes JavaScript code when the browser finishes loading a window or all frames within a `FRAMESET` tag.
`MouseDown`	`onMouseDown`	Executes JavaScript code when the user depresses a mouse button.
`MouseMove`	`onMouseMove`	Executes JavaScript code when the user moves the cursor.
`MouseOut`	`onMouseOut`	Executes JavaScript code each time the mouse pointer leaves an area (client-side image map) or link from inside that area or link.
`MouseOver`	`onMouseOver`	Executes JavaScript code once each time the mouse pointer moves over an object or area from outside that object or area.
`MouseUp`	`onMouseUp`	Executes JavaScript code when the user releases a mouse button.
`Move`	`onMove`	Executes JavaScript code when the user or script moves a window or frame.
`Reset`	`onReset`	Executes JavaScript code when a user resets a form (clicks a Reset button).
`Resize`	`onResize`	Executes JavaScript code when a user or script resizes a window or frame.
`Select`	`onSelect`	Executes JavaScript code when a user selects some of the text within a text or textarea field.
`Submit`	`onSubmit`	Executes JavaScript code when a user submits a form.
`Unload`	`onUnload`	Executes JavaScript code when the user exits a document.

onAbort

Executes JavaScript code when an abort event occurs; that is, when the user aborts the loading of an image (for example by clicking a link or clicking the Stop button).

Event handler for	`Image`
Implemented in	JavaScript 1.1

Syntax `onAbort="`*`handlerText`*`"`

Parameters

`handlerText`	JavaScript code or a call to a JavaScript function.

Event properties used

Property	Description
`type`	Indicates the type of event.
`target`	Indicates the object to which the event was originally sent.

Examples In the following example, an `onAbort` handler in an `Image` object displays a message when the user aborts the image load:

```
<IMG NAME="aircraft" SRC="f15e.gif"
   onAbort="alert('You didn\'t get to see the image!')">
```

See also `event`, `onError`, `onLoad`

onBlur

Executes JavaScript code when a blur event occurs; that is, when a form element loses focus or when a window or frame loses focus.

Event handler for	`Button`, `Checkbox`, `FileUpload`, `Layer`, `Password`, `Radio`, `Reset`, `Select`, `Submit`, `Text`, `Textarea`, `window`
Implemented in	JavaScript 1.0 JavaScript 1.1: event handler of `Button`, `Checkbox`, `FileUpload`, `Frame`, `Password`, `Radio`, `Reset`, `Submit`, and `window`

Syntax `onBlur="`*`handlerText`*`"`

Parameters

`handlerText`	JavaScript code or a call to a JavaScript function.

Description The `blur` event can result from a call to the `window.blur` method or from the user clicking the mouse on another object or window or tabbing with the keyboard.

For windows, frames, and framesets, `onBlur` specifies JavaScript code to execute when a window loses focus.

A frame's `onBlur` event handler overrides an `onBlur` event handler in the `BODY` tag of the document loaded into frame.

Note In JavaScript 1.1, on some platforms placing an `onBlur` event handler in a `FRAMESET` tag has no effect.

Event properties used

Property	Description
`type`	Indicates the type of event.
`target`	Indicates the object to which the event was originally sent.

Examples **Example 1: Validate form input.** In the following example, `userName` is a required text field. When a user attempts to leave the field, the `onBlur` event handler calls the `required` function to confirm that `userName` has a legal value.

```
<INPUT TYPE="text" VALUE="" NAME="userName"
   onBlur="required(this.value)">
```

Example 2: Change the background color of a window. In the following example, a window's `onBlur` and `onFocus` event handlers change the window's background color depending on whether the window has focus.

```
<BODY BGCOLOR="lightgrey"
   onBlur="document.bgColor='lightgrey'"
   onFocus="document.bgColor='antiquewhite'">
```

Example 3: Change the background color of a frame. The following example creates four frames. The source for each frame, `onblur2.html` has the `BODY` tag with the onBlur and `onFocus` event handlers shown in Example 1. When the document loads, all frames are light grey. When the user clicks a frame, the `onFocus` event handler changes the frame's background color to antique white. The frame that loses focus is changed to light grey. Note that the `onBlur` and `onFocus` event handlers are within the `BODY` tag, not the `FRAME` tag.

```
<FRAMESET ROWS="50%,50%" COLS="40%,60%">
<FRAME SRC=onblur2.html NAME="frame1">
<FRAME SRC=onblur2.html NAME="frame2">
<FRAME SRC=onblur2.html NAME="frame3">
<FRAME SRC=onblur2.html NAME="frame4">
</FRAMESET>
```

The following code has the same effect as the previous code, but is implemented differently. The `onFocus` and `onBlur` event handlers are associated with the frame, not the document. The `onBlur` and `onFocus` event handlers for the frame are specified by setting the `onblur` and `onfocus` properties.

```
<SCRIPT>
function setUpHandlers() {
   for (var i = 0; i < frames.length; i++) {
      frames[i].onfocus=new Function("document.bgColor='antiquewhite'")
      frames[i].onblur=new Function("document.bgColor='lightgrey'")
   }
}
</SCRIPT>
```

```
<FRAMESET ROWS="50%,50%" COLS="40%,60%" onLoad=setUpHandlers()>
<FRAME SRC=onblur2.html NAME="frame1">
<FRAME SRC=onblur2.html NAME="frame2">
<FRAME SRC=onblur2.html NAME="frame3">
<FRAME SRC=onblur2.html NAME="frame4">
</FRAMESET>
```

Example 4: Close a window. In the following example, a window's `onBlur` event handler closes the window when the window loses focus.

```
<BODY onBlur="window.close()">
This is some text
</BODY>
```

See also `event`, `onChange`, `onFocus`

onChange

Executes JavaScript code when a change event occurs; that is, when a `Select`, `Text`, or `Textarea` field loses focus and its value has been modified.

Event handler for	`FileUpload`, `Select`, `Text`, `Textarea`
Implemented in	JavaScript 1.0 event handler for `Select`, `Text`, and `Textarea`
	JavaScript 1.1: added as event handler of `FileUpload`

Syntax `onChange="`*`handlerText`*`"`

Parameters

`handlerText`	JavaScript code or a call to a JavaScript function.

Description Use `onChange` to validate data after it is modified by a user.

Event properties used

Property	Description
`type`	Indicates the type of event.
`target`	Indicates the object to which the event was originally sent.

Examples In the following example, `userName` is a text field. When a user changes the text and leaves the field, the onChange event handler calls the `checkValue` function to confirm that `userName` has a legal value.

```
<INPUT TYPE="text" VALUE="" NAME="userName"
   onChange="checkValue(this.value)">
```

See also `event`, `onBlur`, `onFocus`

onClick

Executes JavaScript code when a click event occurs; that is, when an object on a form is clicked. (A click event is a combination of the MouseDown and MouseUp events).

Event handler for	`Button`, `document`, `Checkbox`, `Link`, `Radio`, `Reset`, `Submit`
Implemented in	JavaScript 1.0 JavaScript 1.1: added the ability to return false to cancel the action associated with a click event

Syntax `onClick="`*`handlerText`*`"`

Parameters

`handlerText`	JavaScript code or a call to a JavaScript function.

Event properties used

Property	Description
`type`	Indicates the type of event.
`target`	Indicates the object to which the event was originally sent.
When a link is clicked, `layerX`, `layerY`, `pageX`, `pageY`, `screenX`, `screenY`	Represent the cursor location at the time the event occurred.
`which`	Represents 1 for a left-mouse click and 3 for a right-mouse click.
`modifiers`	Contains the list of modifier keys held down when the event occurred.

Description For checkboxes, links, radio buttons, reset buttons, and submit buttons, `onClick` can return false to cancel the action normally associated with a `click` event.

For example, the following code creates a link that, when clicked, displays a confirm dialog box. If the user clicks the link and then chooses cancel, the page specified by the link is not loaded.

```
<A HREF = "http://home.netscape.com/"
   onClick="return confirm('Load Netscape home page?')">
Netscape</A>
```

If the event handler returns false, the default action of the object is canceled as follows:

- Buttons—no default action; nothing is canceled
- Radio buttons and checkboxes—nothing is set
- Submit buttons—form is not submitted
- Reset buttons—form is not reset

Note In JavaScript 1.1, on some platforms, returning false in an `onClick` event handler for a reset button has no effect.

Examples **Example 1: Call a function when a user clicks a button.** Suppose you have created a JavaScript function called `compute`. You can execute the `compute` function when the user clicks a button by calling the function in the onClick event handler, as follows:

```
<INPUT TYPE="button" VALUE="Calculate" onClick="compute(this.form)">
```

In the preceding example, the keyword `this` refers to the current object; in this case, the Calculate button. The construct `this.form` refers to the form containing the button.

For another example, suppose you have created a JavaScript function called `pickRandomURL` that lets you select a URL at random. You can use `onClick` to specify a value for the `HREF` attribute of the `A` tag dynamically, as shown in the following example:

```
<A HREF=""
   onClick="this.href=pickRandomURL()"
   onMouseOver="window.status='Pick a random URL'; return true">
Go!</A>
```

In the above example, `onMouseOver` specifies a custom message for the browser's status bar when the user places the mouse pointer over the Go! anchor. As this example shows, you must return true to set the `window.status` property in the `onMouseOver` event handler.

Example 2: Cancel the checking of a checkbox. The following example creates a checkbox with `onClick`. The event handler displays a confirm that warns the user that checking the checkbox purges all files. If the user chooses Cancel, `onClick` returns false and the checkbox is not checked.

```
<INPUT TYPE="checkbox" NAME="check1" VALUE="check1"
   onClick="return confirm('This purges all your files. Are you sure?')"> Remove files
```

See also `event`

onDblClick

Executes JavaScript code when a DblClick event occurs; that is, when the user double-clicks a form element or a link.

Event handler for `document`, `Link`

Implemented in JavaScript 1.2

Syntax `onDblClick="`*`handlerText`*`"`

Parameters

`handlerText`	JavaScript code or a call to a JavaScript function.

Note `DblClick` is not implemented on the Macintosh.

Event properties used

Property	Description
`type`	Indicates the type of event.
`target`	Indicates the object to which the event was originally sent.
`layerX, layerY, pageX, pageY, screenX, screenY`	Represent the cursor location at the time the event occurred.
`which`	Represents 1 for a left-mouse double-click and 3 for a right-mouse double-click.
`modifiers`	Contains the list of modifier keys held down when the event occurred.

Examples The following example opens an alert dialog box when a user double-clicks a button:

```
<form>
<INPUT Type="button" Value="Double Click Me!"
          onDblClick="alert('You just double clicked me!')">
</form>
```

See also `event`

onDragDrop

Executes JavaScript code when a DragDrop event occurs; that is, when the user drops an object onto the browser window, such as dropping a file.

Event handler for	`window`
Implemented in	JavaScript 1.2

Syntax `onDragDrop="`*`handlerText`*`"`

Parameters

`handlerText`	JavaScript code or a call to a JavaScript function.

Event properties used

Property	Description
`type`	Indicates the type of event.
`target`	Indicates the object to which the event was originally sent.
`data`	Returns an Array of Strings containing the URLs of the dropped objects.
`modifiers`	Contains the list of modifier keys held down when the event occurred.
`screenX`, `screenY`	Represent the cursor location at the time the event occurred.

Security Getting the `data` property of the DragDrop event requires the `UniversalBrowserRead` privilege. For information on security, see the *Client-Side JavaScript Guide*.

Description The `DragDrop` event is fired whenever a system item (file, shortcut, and so on) is dropped onto the browser window using the native system's drag and drop mechanism. The normal response for the browser is to attempt to load the item into the browser window. If the event handler for the `DragDrop` event returns true, the browser loads the item normally. If the event handler returns false, the drag and drop is canceled.

See also `event`

onError

Executes JavaScript code when an error event occurs; that is, when the loading of a document or image causes an error.

Event handler for	`Image`, `window`
Implemented in	JavaScript 1.1

Syntax `onError="`*`handlerText`*`"`

Parameters

`handlerText`	JavaScript code or a call to a JavaScript function.

Description An error event occurs only when a JavaScript syntax or runtime error occurs, not when a browser error occurs. For example, if you try set `window.location.href='notThere.html'` and `notThere.html` does not exist, the resulting error message is a browser error message; therefore, `onError` would not intercept that message. However, an error event *is* triggered by a bad URL within an `IMG` tag or by corrupted image data.

`window.onerror` applies only to errors that occur in the window containing `window.onerror`, not in other windows.

`onError` can be any of the following:

- null to suppress all JavaScript error dialogs. Setting `window.onerror` to null means your users won't see JavaScript errors caused by your own code.
- The name of a function that handles errors (arguments are message text, URL, and line number of the offending line). To suppress the standard JavaScript error dialog, the function must return true. See Example 3 below.
- A variable or property that contains null or a valid function reference.

If you write an error-handling function, you have three options for reporting errors:

- Trace errors but let the standard JavaScript dialog report them (use an error handling function that returns false or does not return a value)
- Report errors yourself and disable the standard error dialog (use an error handling function that returns true)
- Turn off all error reporting (set the onError event handler to null)

Event properties used

Property	Description
`type`	Indicates the type of event.
`target`	Indicates the object to which the event was originally sent.

Examples

Example 1: Null event handler. In the following `IMG` tag, the code `onError="null"` suppresses error messages if errors occur when the image loads.

```
<IMG NAME="imageBad1" SRC="corrupt.gif" ALIGN="left" BORDER="2"
   onError="null">
```

Example 2: Null event handler for a window. The `onError` event handler for windows cannot be expressed in HTML. Therefore, you must spell it all lowercase and set it in a `SCRIPT` tag. The following code assigns null to the `onError` handler for the entire window, not just the `Image` object. This suppresses all JavaScript error messages, including those for the `Image` object.

```
<SCRIPT>
window.onerror=null
</SCRIPT>
<IMG NAME="imageBad1" SRC="corrupt.gif" ALIGN="left" BORDER="2">
```

However, if the `Image` object has a custom `onError` event handler, the handler would execute if the image had an error. This is because `window.onerror=null` suppresses JavaScript error messages, not `onError` event handlers.

```
<SCRIPT>
window.onerror=null
function myErrorFunc() {
   alert("The image had a nasty error.")
}
</SCRIPT>
<IMG NAME="imageBad1" SRC="corrupt.gif" ALIGN="left" BORDER="2"
   onError="myErrorFunc()">
```

In the following example, `window.onerror=null` suppresses all error reporting. Without `onerror=null`, the code would cause a stack overflow error because of infinite recursion.

```
<SCRIPT>
window.onerror = null;
function testErrorFunction() {
   testErrorFunction();
}
```

```
</SCRIPT>
<BODY onload="testErrorFunction()">
test message
</BODY>
```

Example 3: Error handling function. The following example defines a function, `myOnError`, that intercepts JavaScript errors. The function uses three arrays to store the message, URL, and line number for each error. When the user clicks the Display Error Report button, the `displayErrors` function opens a window and creates an error report in that window. Note that the function returns true to suppress the standard JavaScript error dialog.

```
<SCRIPT>
window.onerror = myOnError

msgArray = new Array()
urlArray = new Array()
lnoArray = new Array()

function myOnError(msg, url, lno) {
   msgArray[msgArray.length] = msg
   urlArray[urlArray.length] = url
   lnoArray[lnoArray.length] = lno
   return true
}

function displayErrors() {
   win2=window.open('','window2','scrollbars=yes')
   win2.document.writeln('<B>Error Report</B><P>')

   for (var i=0; i < msgArray.length; i++) {
      win2.document.writeln('<B>Error in file:</B> ' + urlArray[i] + '<BR>')
      win2.document.writeln('<B>Line number:</B> ' + lnoArray[i] + '<BR>')
      win2.document.writeln('<B>Message:</B> ' + msgArray[i] + '<P>')
   }
   win2.document.close()
}
</SCRIPT>

<BODY onload="noSuchFunction()">
<FORM>
<BR><INPUT TYPE="button" VALUE="This button has a syntax error"
   onClick="alert('unterminated string)">

<P><INPUT TYPE="button" VALUE="Display Error Report"
   onClick="displayErrors()">
</FORM>
```

This example produces the following output:

```
Error Report

Error in file: file:///c%7C/temp/onerror.html
Line number: 34
Message: unterminated string literal

Error in file: file:///c%7C/temp/onerror.html
Line number: 34
Message: missing ) after argument list

Error in file: file:///c%7C/temp/onerror.html
Line number: 30
Message: noSuchFunction is not defined
```

Example 4: Event handler calls a function. In the following `IMG` tag, `onError` calls the function `badImage` if errors occur when the image loads.

```
<SCRIPT>
function badImage(theImage) {
   alert('Error: ' + theImage.name + ' did not load properly.')
}
</SCRIPT>
<FORM>
<IMG NAME="imageBad2" SRC="orca.gif" ALIGN="left" BORDER="2"
   onError="badImage(this)">
</FORM>
```

See also `event`, `onAbort`, `onLoad`

onFocus

Executes JavaScript code when a focus event occurs; that is, when a window, frame, or frameset receives focus or when a form element receives input focus.

Event handler for	`Button`, `Checkbox`, `FileUpload`, `Layer`, `Password`, `Radio`, `Reset`, `Select`, `Submit`, `Text`, `Textarea`, `window`
Implemented in	JavaScript 1.0 JavaScript 1.1: event handler of `Button, Checkbox, FileUpload, Frame, Password, Radio, Reset, Submit,` and `window` JavaScript 1.2: event handler of `Layer`

Syntax `onFocus="`*`handlerText`*`"`

Parameters

`handlerText`	JavaScript code or a call to a JavaScript function.

Description The focus event can result from a `focus` method or from the user clicking the mouse on an object or window or tabbing with the keyboard. Selecting within a field results in a select event, not a focus event. `onFocus` executes JavaScript code when a focus event occurs.

A frame's `onFocus` event handler overrides an `onFocus` event handler in the `BODY` tag of the document loaded into frame.

Note that placing an alert in an `onFocus` event handler results in recurrent alerts: when you press OK to dismiss the alert, the underlying window gains focus again and produces another focus event.

Note In JavaScript 1.1, on some platforms, placing an `onFocus` event handler in a `FRAMESET` tag has no effect.

Event properties used

Property	Description
`type`	Indicates the type of event.
`target`	Indicates the object to which the event was originally sent.

Examples The following example uses an `onFocus` handler in the `valueField` `Textarea` object to call the `valueCheck` function.

```
<INPUT TYPE="textarea" VALUE="" NAME="valueField"
   onFocus="valueCheck()">
```

See also the examples for `onBlur`.

See also `event`, `onBlur`, `onChange`

onKeyDown

Executes JavaScript code when a KeyDown event occurs; that is, when the user depresses a key.

Event handler for	`document`, `Image`, `Link`, `Textarea`
Implemented in	JavaScript 1.2

Syntax `onKeyDown="`*`handlerText`*`"`

Parameters

`handlerText`	JavaScript code or a call to a JavaScript function.

Event properties used

Property	Description
`type`	Indicates the type of event.
`target`	Indicates the object to which the event was originally sent.
`layerX`, `layerY`, `pageX`, `pageY`, `screenX`, `screenY`	For an event over a window, these represent the cursor location at the time the event occurred. For an event over a form, they represent the position of the form element.
`which`	Represents the ASCII value of the key pressed. To get the actual letter, number, or symbol of the pressed key, use the `String.fromCharCode` method. To set this property when the ASCII value is unknown, use the `String.charCodeAt` method.
`modifiers`	Contains the list of modifier keys held down when the event occurred.

Description A `KeyDown` event always occurs before a `KeyPress` event. If `onKeyDown` returns false, no `KeyPress` events occur. This prevents `KeyPress` events occurring due to the user holding down a key.

Examples The following example uses the `blockA` function to evaluate characters entered from the keyboard in the `textentry` text box. If a user enters either “a” or “A”, the function returns false and the text box does not display the value.

```
<form name="main">
   <input name="textentry" type=text size=10 maxlength=10>
</form>

<script>
function blockA(e) {
        var keyChar = String.fromCharCode(e.which);
        if (keyChar == 'A' || keyChar == 'a')
                return false;
}

document.main.textentry.onkeydown = blockA;
</script>
```

In the function, the `which` property of the event assigns the ASCII value of the key the user presses to the `keyChar` variable. The `if` statement evaluates `keyChar` and returns false for the specified characters.

See also `event`, `onKeyPress`, `onKeyUp`

onKeyPress

Executes JavaScript code when a KeyPress event occurs; that is, when the user presses or holds down a key.

Event handler for	`document`, `Image`, `Link`, `Textarea`
Implemented in	JavaScript 1.2

Syntax `onKeyPress="`*`handlerText`*`"`

Parameters

`handlerText`	JavaScript code or a call to a JavaScript function.

Event properties used

Property	Description
`type`	Indicates the type of event.
`target`	Indicates the object to which the event was originally sent.
`layerX`, `layerY`, `pageX`, `pageY`, `screenX`, `screenY`	For an event over a window, these represent the cursor location at the time the event occurred. For an event over a form, they represent the position of the form element.
`which`	Represents the ASCII value of the key pressed. To get the actual letter, number, or symbol of the pressed key, use the `String.fromCharCode` method. To set this property when the ASCII value is unknown, use the `String.charCodeAt` method.
`modifiers`	Contains the list of modifier keys held down when the event occurred.

Description A `KeyPress` event occurs immediately after a `KeyDown` event only if `onKeyDown` returns something other than false. A `KeyPress` event repeatedly occurs until the user releases the key. You can cancel individual `KeyPress` events.

Examples In this example, the `captureEvents` method catches keyboard input and the `onKeyPress` handler calls the `blockA` function to examine the keystrokes. If the keystrokes are “a” or “z”, the function scrolls the Navigator window.

```
function blockA(e) {
        var keyChar = String.fromCharCode(e.which);
        if (keyChar == 'A' || keyChar == 'a')
                self.scrollBy(10,10);

        else if(keyChar == 'Z' || keyChar == 'z')
                self.scrollBy(-10,-10);

        else    return false;
}
document.captureEvents(Event.KEYPRESS);
document.onkeypress = blockA;
```

See also `event`, `onKeyDown`, `onKeyUp`

onKeyUp

Executes JavaScript code when a KeyUp event occurs; that is, when the user releases a key.

Event handler for	`document`, `Image`, `Link`, `Textarea`
Implemented in	JavaScript 1.2

Syntax `onKeyUp="`*`handlerText`*`"`

Parameters

`handlerText`	JavaScript code or a call to a JavaScript function.

Event properties used

Property	Description
`type`	Indicates the type of event.
`target`	Indicates the object to which the event was originally sent.
`layerX`, `layerY`, `pageX`, `pageY`, `screenX`, `screenY`	For an event over a window, these represent the cursor location at the time the event occurred. For an event over a form, they represent the position of the form element.
`which`	Represents the ASCII value of the key pressed. To get the actual letter, number, or symbol of the pressed key, use the `String.fromCharCode` method. To set this property when the ASCII value is unknown, use the `String.charCodeAt` method.
`modifiers`	Contains the list of modifier keys held down when the event occurred.

Examples In this example, the `captureEvents` method catches keyboard input and the `onKeyUp` handler calls the `Key_Up` function. An alert method within the function opens a dialog box to display the value of the keystroke.

```
function Key_Up(e) {
        var keyChar = String.fromCharCode(e.which);
        alert("Hold '" + keyChar +"' again for me, okay?");
}
document.onkeyup=Key_Up;
document.captureEvents(Event.KEYUP);
```

See also `event`

onLoad

Executes JavaScript code when a load event occurs; that is, when the browser finishes loading a window or all frames within a `FRAMESET` tag.

Event handler for	`Image`, `Layer`, `window`
Implemented in	JavaScript 1.0 JavaScript 1.1: event handler of `Image`

Syntax `onLoad="`*`handlerText`*`"`

Parameters

`handlerText`	JavaScript code or a call to a JavaScript function.

Description Use the `onLoad` event handler within either the `BODY` or the `FRAMESET` tag, for example, `<BODY onLoad="...">`.

In a `FRAMESET` and `FRAME` relationship, an `onLoad` event within a frame (placed in the `BODY` tag) occurs before an `onLoad` event within the `FRAMESET` (placed in the `FRAMESET` tag).

For images, the `onLoad` event handler indicates the script to execute when an image is displayed. Do not confuse displaying an image with loading an image. You can load several images, then display them one by one in the same `Image` object by setting the object's `src` property. If you change the image displayed in this way, `onLoad` executes every time an image is displayed, not just when the image is loaded into memory.

If you specify an `onLoad` event handler for an `Image` object that displays a looping GIF animation (multi-image GIF), each loop of the animation triggers the `onLoad` event, and the event handler executes once for each loop.

You can use the `onLoad` event handler to create a JavaScript animation by repeatedly setting the `src` property of an `Image` object. See `Image` for information.

Event properties used

Property	Description
`type`	Indicates the type of event.
`target`	Indicates the object to which the event was originally sent.
`width, height`	For an event over a window, but not over a layer, these represent the width and height of the window.

Examples **Example 1: Display message when page loads.** In the following example, the onLoad event handler displays a greeting message after a Web page is loaded.

```
<BODY onLoad="window.alert("Welcome to the Brave New World home page!")>
```

Example 2: Display alert when image loads. The following example creates two `Image` objects, one with the `Image` constructor and one with the `IMG` tag. Each `Image` object has an `onLoad` event handler that calls the `displayAlert` function, which displays an alert. For the image created with the `IMG` tag, the alert displays the image name. For the image created with the `Image` constructor, the alert displays a message without the image name. This is because the onLoad handler for an object created with the `Image` constructor must be the name of a function, and it cannot specify parameters for the `displayAlert` function.

```
<SCRIPT>
imageA = new Image(50,50)
imageA.onload=displayAlert
imageA.src="cyanball.gif"

function displayAlert(theImage) {
   if (theImage==null) {
      alert('An image loaded')
   }
   else alert(theImage.name + ' has been loaded.')
}
</SCRIPT>
```

```
<IMG NAME="imageB" SRC="greenball.gif" ALIGN="top"
   onLoad=displayAlert(this)><BR>
```

Example 3: Looping GIF animation. The following example displays an image, `birdie.gif`, that is a looping GIF animation. The onLoad event handler for the image increments the variable `cycles`, which keeps track of the number of times the animation has looped. To see the value of `cycles`, the user clicks the button labeled Count Loops.

```
<SCRIPT>
var cycles=0
</SCRIPT>
<IMG ALIGN="top" SRC="birdie.gif" BORDER=0
   onLoad="++cycles">
<INPUT TYPE="button" VALUE="Count Loops"
   onClick="alert('The animation has looped ' + cycles + ' times.')">
```

Example 4: Change GIF animation displayed. The following example uses an onLoad event handler to rotate the display of six GIF animations. Each animation is displayed in sequence in one `Image` object. When the document loads, `!anim0.html` is displayed. When that animation completes, the onLoad event handler causes the next file, `!anim1.html`, to load in place of the first file. After the last animation, `!anim5.html`, completes, the first file is again displayed. Notice that the `changeAnimation` function does not call itself after changing the `src` property of the `Image` object. This is because when the `src` property changes, the image's `onLoad` event handler is triggered and the `changeAnimation` function is called.

```
<SCRIPT>
var whichImage=0
var maxImages=5

function changeAnimation(theImage) {
   ++whichImage
   if (whichImage <= maxImages) {
      var imageName="!anim" + whichImage + ".gif"
      theImage.src=imageName
   } else {
      whichImage=-1
      return
   }
}
</SCRIPT>

<IMG NAME="changingAnimation" SRC="!anim0.gif" BORDER=0 ALIGN="top"
   onLoad="changeAnimation(this)">
```

See also the examples for `Image`.

See also `event`, `onAbort`, `onError`, `onUnload`

onMouseDown

Executes JavaScript code when a MouseDown event occurs; that is, when the user depresses a mouse button.

Event handler for	`Button`, `document`, `Link`
Implemented in	JavaScript 1.2

Syntax

```
onMouseDown="handlerText"
```

Parameters

`handlerText`	JavaScript code or a call to a JavaScript function.

Event properties used

Property	Description
`type`	Indicates the type of event.
`target`	Indicates the object to which the event was originally sent.
`layerX`, `layerY`, `pageX`, `pageY`, `screenX`, `screenY`	Represent the cursor location at the time the `MouseDown` event occurred.
`which`	Represents 1 for a left-mouse-button down and 3 for a right-mouse-button down.
`modifiers`	Contains the list of modifier keys held down when the `MouseDown` event occurred.

Description If `onMouseDown` returns false, the default action (entering drag mode, entering selection mode, or arming a link) is canceled.

Arming is caused by a `MouseDown` over a link. When a link is armed it changes color to represent its new state.

Examples This example lets users move an image on an HTML page by dragging it with the mouse. Your HTML code defines the image and positions it in a layer called `container1`. In your JavaScript code, event handlers set the position properties of `container1` as users drag the image, creating the animation.

Using style sheets, the image is initially defined and positioned as follows:

```
<HEAD>
<STYLE type="text/css">
   #container1 { position:absolute; left:200; top:200}
</STYLE>
</HEAD>

<BODY>
<P ID="container1">
<img src="backgrnd.gif" name="myImage" width=96 height=96>
</P>
</BODY>
```

In the previous HTML code, the `ID` attribute for the `P` element which contains the image is set to `container1`, making `container1` a unique identifier for the paragraph and the image. The `STYLE` tag creates a layer for `container1` and positions it.

The following JavaScript code defines `onMouseDown`, `onMouseUp`, and `onMouseMove` event handlers:

```
<SCRIPT>
container1.captureEvents(Event.MOUSEUP|Event.MOUSEDOWN);
container1.onmousedown=DRAG_begindrag;
container1.onmouseup=DRAG_enddrag;
var DRAG_lastX, DRAG_lastY, DRAG_dragging;
function DRAG_begindrag(e) {
        if (e.which == 1) {
                window.captureEvents(Event.MOUSEMOVE);
          window.onmousemove=DRAG_drag;
                DRAG_lastX=e.pageX;
                DRAG_lastY=e.pageY;
                DRAG_dragging=true;
                return false;
        }
        else {
                /*Do any right mouse button processing here*/
                return true;
        }
}
function DRAG_enddrag(e) {
        if (e.which == 1) {
                window.releaseEvents(Event.MOUSEMOVE);
          window.onmousemove=null
                DRAG_dragging=false;
                return false;
        }
        else {
                /*Do any right mouse button processing here*/
                return true;
        }
}
function DRAG_drag(e) {
        if (DRAG_dragging) {
                /*This function called only if MOUSEMOVEs are captured*/
                moveBy(e.pageX-DRAG_lastX, e.pageY-DRAG_lastY);
                DRAG_lastX = e.pageX;
                DRAG_lastY = e.pageY;
                return false;
        }
        else {
                return true;
        }
}
</SCRIPT>
```

In the previous code, the `captureEvents` method captures `MouseUp` and `MouseDown` events. The `DRAG_begindrag` and `DRAG_enddrag` functions are respectively called to handle these events.

When a user presses the left mouse button, the `DRAG_begindrag` function starts capturing `MouseMove` events and tells the `DRAG_drag` function to handle them. It then assigns the value of the `MouseDown` event's `pageX` property to `DRAG_lastX`, the value of the `pageY` property to `DRAG_lastY`, and `true` to `DRAG_dragging`.

The `DRAG_drag` function evaluates `DRAG_dragging` to make sure the `MouseMove` event was captured by `DRAG_begindrag`, then it uses the `moveBy` method to position the object, and reassigns values to `DRAG_lastX` and `DRAG_lastY`.

When the user releases the left mouse button, the `DRAG_enddrag` function stops capturing `MouseMove` events. `DRAG_enddrag` then makes sure no other functions are called by setting `onmousemove` to `Null` and `DRAG_dragging` to `false`.

See also `event`

onMouseMove

Executes JavaScript code when a MouseMove event occurs; that is, when the user moves the cursor.

Event handler for	None
Implemented in	JavaScript 1.2

Syntax `onMouseMove="`*`handlerText`*`"`

Parameters

`handlerText`	JavaScript code or a call to a JavaScript function.

Event of Because mouse movement happens so frequently, by default, `onMouseMove` is not an event of any object. You must explicitly set it to be associated with a particular object.

Event properties used

Property	Description
`type`	Indicates the type of event.
`target`	Indicates the object to which the event was originally sent.
`layerX`, `layerY`, `pageX`, `pageY`, `screenX`, `screenY`	Represent the cursor location at the time the `MouseMove` event occurred.

Description The `MouseMove` event is sent only when a capture of the event is requested by an object. For information on events, see the *Client-Side JavaScript Guide*.

Examples See the examples for `onMouseDown`.

See also `event`, `document.captureEvents`

onMouseOut

Executes JavaScript code when a MouseOut event occurs; that is, each time the mouse pointer leaves an area (client-side image map) or link from inside that area or link.

Event handler for	`Layer`, `Link`
Implemented in	JavaScript 1.1

Syntax `onMouseOut="`*`handlerText`*`"`

Parameters

`handlerText`	JavaScript code or a call to a JavaScript function.

Description If the mouse moves from one area into another in a client-side image map, you'll get `onMouseOut` for the first area, then `onMouseOver` for the second.

`Area` tags that use `onMouseOut` must include the `HREF` attribute within the `AREA` tag.

You must return true within the event handler if you want to set the `status` or `defaultStatus` properties with `onMouseOver`.

Event properties used

Property	Description
`type`	Indicates the type of event.
`target`	Indicates the object to which the event was originally sent.
`layerX`, `layerY`, `pageX`, `pageY`, `screenX`, `screenY`	Represent the cursor location at the time the `MouseOut` event occurred.

Examples See the examples for `Link`.

See also `event`, `onMouseOver`

onMouseOver

Executes JavaScript code when a MouseOver event occurs; that is, once each time the mouse pointer moves over an object or area from outside that object or area.

Event handler for	`Layer`, `Link`
Implemented in	JavaScript 1.0 JavaScript 1.1: event handler of `Area`

Syntax `onMouseOver="`*`handlerText`*`"`

Parameters

`handlerText`	JavaScript code or a call to a JavaScript function.

Description If the mouse moves from one area into another in a client-side image map, you'll get `onMouseOut` for the first area, then `onMouseOver` for the second.

`Area` tags that use `onMouseOver` must include the `HREF` attribute within the `AREA` tag.

You must return true within the event handler if you want to set the `status` or `defaultStatus` properties with `onMouseOver`.

Event properties used

Property	Description
`type`	Indicates the type of event.
`target`	Indicates the object to which the event was originally sent.
`layerX`, `layerY`, `pageX`, `pageY`, `screenX`, `screenY`	Represent the cursor location at the time the `MouseOver` event occurred.

Examples By default, the `HREF` value of an anchor displays in the status bar at the bottom of the browser when a user places the mouse pointer over the anchor. In the following example, `onMouseOver` provides the custom message “Click this if you dare.”

```
<A HREF="http://home.netscape.com/"
   onMouseOver="window.status='Click this if you dare!'; return true">
Click me</A>
```

See `onClick` for an example of using `onMouseOver` when the `A` tag’s `HREF` attribute is set dynamically.

See also the examples for `Link`.

See also `event`, `onMouseOut`

onMouseUp

Executes JavaScript code when a MouseUp event occurs; that is, when the user releases a mouse button.

Event handler for	`Button`, `document`, `Link`
Implemented in	JavaScript 1.2

Syntax `onMouseUp="`*`handlerText`*`"`

Parameters

`handlerText`	JavaScript code or a call to a JavaScript function.

Event properties used

Property	Description
`type`	Indicates the type of event.
`target`	Indicates the object to which the event was originally sent.
`layerX`, `layerY`, `pageX`, `pageY`, `screenX`, `screenY`	Represent the cursor location at the time the `MouseUp` event occurred.
`which`	Represents 1 for a left-mouse-button up and 3 for a right-mouse-button up.
`modifiers`	Contains the list of modifier keys held down when the `MouseUp` event occurred.

Description If `onMouseUp` returns false, the default action is canceled. For example, if `onMouseUp` returns false over an armed link, the link is not triggered. Also, if `MouseUp` occurs over an unarmed link (possibly due to `onMouseDown` returning false), the link is not triggered.

Note Arming is caused by a `MouseDown` over a link. When a link is armed it changes color to represent its new state.

Examples See the examples for `onMouseDown`.

See also `event`

onMove

Executes JavaScript code when a move event occurs; that is, when the user or script moves a window or frame.

Event handler for	`window`
Implemented in	JavaScript 1.2

Syntax `onMove="`*`handlerText`*`"`

Parameters

`handlerText`	JavaScript code or a call to a JavaScript function.

Event properties used

Property	Description
`type`	Indicates the type of event.
`target`	Indicates the object to which the event was originally sent.
`screenX`, `screenY`	Represent the position of the top-left corner of the window or frame.

Examples In this example, the `open_now` function creates the `myWin` window and captures `Move` events. The `onMove` handler calls another function which displays a message when a user moves `myWin`.

```
function open_now(){
    var myWin;

myWin=window.open("","displayWindow","width=400,height=400,menubar=no,
                        location=no,alwaysRaised=yes");
    var text="<html><head><title>Test</title></head>"
        +"<body bgcolor=white><h1>Please move this window</h1></body>"
        +"</html>";
    myWin.document.write(text);
    myWin.captureEvents(Event.MOVE);
    myWin.onmove=fun2;
}


function fun2(){
    alert("Hey you moved me!");
    this.focus(); //'this' points to the current object
}
```

See also `event`

onReset

Executes JavaScript code when a reset event occurs; that is, when a user resets a form (clicks a Reset button).

Event handler for	`Form`
Implemented in	JavaScript 1.1

Syntax `onReset="`*`handlerText`*`"`

Parameters

`handlerText`	JavaScript code or a call to a JavaScript function.

Examples The following example displays a `Text` object with the default value "CA" and a reset button. If the user types a state abbreviation in the `Text` object and then clicks the reset button, the original value of "CA" is restored. The form's onReset event handler displays a message indicating that defaults have been restored.

```
<FORM NAME="form1" onReset="alert('Defaults have been restored.')">
State:
<INPUT TYPE="text" NAME="state" VALUE="CA" SIZE="2"><P>
<INPUT TYPE="reset" VALUE="Clear Form" NAME="reset1">
</FORM>
```

Event properties used

Property	Description
`type`	Indicates the type of event.
`target`	Indicates the object to which the event was originally sent.

See also `event`, `Form.reset`, `Reset`

onResize

Executes JavaScript code when a resize event occurs; that is, when a user or script resizes a window or frame.

Event handler for	`window`
Implemented in	JavaScript 1.2

Syntax `onResize="`*`handlerText`*`"`

Parameters

`handlerText`	JavaScript code or a call to a JavaScript function.

Event properties used

Property	Description
`type`	Indicates the type of event.
`target`	Indicates the object to which the event was originally sent.
`width, height`	Represent the width and height of the window or frame.

Description This event is sent after HTML layout completes within the new window inner dimensions. This allows positioned elements and named anchors to have their final sizes and locations queried, image `SRC` properties can be restored dynamically, and so on.

Examples In this example, the `open_now` function creates the `myWin` window and captures `Resize` events. The `onResize` handler calls the `alert_me` function which displays a message when a user resizes `myWin`.

```
function open_now(){
    var myWin;

myWin=window.open("","displayWin","width=400,height=300,resizable=yes,

menubar=no,location=no,alwaysRaised=yes");
    var text="<html><head><title>Test</title></head>"
           +"<body bgcolor=white><h1>Please resize me</h1></body>"
           +"</html>";
    myWin.document.write(text);
    myWin.captureEvents(Event.RESIZE);
    myWin.onresize=alert_me;
}
```

```
function alert_me(){
    alert("You resized me! \nNow my outer width: " + this.outerWidth +
        "\n and my outer height: " +this.outerHeight);
    this.focus();
}
```

See also `event`

onSelect

Executes JavaScript code when a select event occurs; that is, when a user selects some of the text within a text or textarea field.

Event handler for	`Text`, `Textarea`
Implemented in	JavaScript 1.0

Syntax `onSelect="`*`handlerText`*`"`

Parameters

`handlerText`	JavaScript code or a call to a JavaScript function.

Event properties used

Property	Description
`type`	Indicates the type of event.
`target`	Indicates the object to which the event was originally sent.

Examples The following example uses `onSelect` in the `valueField` `Text` object to call the `selectState` function.

```
<INPUT TYPE="text" VALUE="" NAME="valueField" onSelect="selectState()">
```

See also `event`

onSubmit

Executes JavaScript code when a submit event occurs; that is, when a user submits a form.

Event handler for	`Form`
Implemented in	JavaScript 1.0

Syntax `onSubmit="`*`handlerText`*`"`

Parameters

`handlerText`	JavaScript code or a call to a JavaScript function.

Security Submitting a form to a `mailto:` or `news:` URL requires the `UniversalSendMail` privilege. For information on security, see the *Client-Side JavaScript Guide*.

Description You can use `onSubmit` to prevent a form from being submitted; to do so, put a `return` statement that returns false in the event handler. Any other returned value lets the form submit. If you omit the `return` statement, the form is submitted.

Event properties used

Property	Description
`type`	Indicates the type of event.
`target`	Indicates the object to which the event was originally sent.

Examples In the following example, `onSubmit` calls the `validate` function to evaluate the data being submitted. If the data is valid, the form is submitted; otherwise, the form is not submitted.

```
<FORM onSubmit="return validate(this)">
...
</FORM>
```

See also the examples for `Form`.

See also `event`, `Submit`, `Form.submit`

onUnload

Executes JavaScript code when an unload event occurs; that is, when the user exits a document.

Event handler for	`window`
Implemented in	JavaScript 1.0

Syntax `onUnload="`*`handlerText`*`"`

Parameters

`handlerText`	JavaScript code or a call to a JavaScript function.

Description Use `onUnload` within either the `BODY` or the `FRAMESET` tag, for example, `<BODY onUnload="...">`.

In a frameset and frame relationship, an `onUnload` event within a frame (placed in the `BODY` tag) occurs before an `onUnload` event within the frameset (placed in the `FRAMESET` tag).

Event properties used

Property	Description
`type`	Indicates the type of event.
`target`	Indicates the object to which the event was originally sent.

Examples In the following example, `onUnload` calls the `cleanUp` function to perform some shutdown processing when the user exits a Web page:

```
<BODY onUnload="cleanUp()">
```

See also `onLoad`

For general information on event handlers, see the *Client-Side JavaScript Guide*.

For information about the `event` object, see `event`.

Language Elements

2

- **Statements**
- **Operators**

Chapter

4

Statements

This chapter describes all JavaScript statements. JavaScript statements consist of keywords used with the appropriate syntax. A single statement may span multiple lines. Multiple statements may occur on a single line if each statement is separated by a semicolon.

Syntax conventions: All keywords in syntax statements are in bold. Words in italics represent user-defined names or statements. Any portions enclosed in square brackets, [ ], are optional. {statements} indicates a block of statements, which can consist of a single statement or multiple statements delimited by a curly braces { }.

The following table lists statements available in JavaScript.

Table 4.1 JavaScript statements.

`break`	Terminates the current while or for loop and transfers program control to the statement following the terminated loop.
`comment`	Notations by the author to explain what a script does. Comments are ignored by the interpreter.
`continue`	Terminates execution of the block of statements in a while or for loop, and continues execution of the loop with the next iteration.
`do...while`	Executes the specified statements until the test condition evaluates to false. Statements execute at least once.
`export`	Allows a signed script to provide properties, functions, and objects to other signed or unsigned scripts.
`for`	Creates a loop that consists of three optional expressions, enclosed in parentheses and separated by semicolons, followed by a block of statements executed in the loop.
`for...in`	Iterates a specified variable over all the properties of an object. For each distinct property, JavaScript executes the specified statements.
`function`	Declares a function with the specified parameters. Acceptable parameters include strings, numbers, and objects.
`if...else`	Executes a set of statements if a specified condition is true. If the condition is false, another set of statements can be executed.
`import`	Allows a script to import properties, functions, and objects from a signed script that has exported the information.
`label`	Provides an identifier that can be used with break or continue to indicate where the program should continue execution.
`return`	Specifies the value to be returned by a function.
`switch`	Allows a program to evaluate an expression and attempt to match the expression's value to a case label.
`var`	Declares a variable, optionally initializing it to a value.
`while`	Creates a loop that evaluates an expression, and if it is true, executes a block of statements. The loop then repeats, as long as the specified condition is true.
`with`	Establishes the default object for a set of statements.

break

Use the break statement to terminate a loop, `switch`, or label statement.

Terminates the current loop, `switch`, or label statement and transfers program control to the statement following the terminated loop.

Implemented in	JavaScript 1.0, NES 2.0
ECMA version	ECMA-262

Syntax `break [`*`label`*`]`

Parameter

`label`	Identifier associated with the label of the statement.

Description The `break` statement includes an optional label that allows the program to break out of a labeled statement. The statements in a labeled statement can be of any type.

Examples **Example 1.** The following function has a `break` statement that terminates the `while` loop when `e` is 3, and then returns the value 3 * `x`.

```
function testBreak(x) {
   var i = 0
   while (i < 6) {
      if (i == 3)
         break
      i++
   }
   return i*x
}
```

Example 2. In the following example, a statement labeled `checkiandj` contains a statement labeled `checkj`. If `break` is encountered, the program breaks out of the `checkj` statement and continues with the remainder of the `checkiandj` statement. If `break` had a label of `checkiandj`, the program would break out of the `checkiandj` statement and continue at the statement following `checkiandj`.

```
checkiandj :
   if (4==i) {
      document.write("You've entered " + i + ".<BR>");
      checkj :
         if (2==j) {
            document.write("You've entered " + j + ".<BR>");
            break checkj;
            document.write("The sum is " + (i+j) + ".<BR>");
         }
      document.write(i + "-" + j + "=" + (i-j) + ".<BR>");
   }
```

See also `continue`, `label`, `switch`

comment

Notations by the author to explain what a script does. Comments are ignored by the interpreter.

Implemented in	JavaScript 1.0, NES 2.0
ECMA version	ECMA-262

Syntax

```
// comment text
/* multiple line comment text */
```

Description JavaScript supports Java-style comments:

- Comments on a single line are preceded by a double-slash (//).
- Comments that span multiple lines are preceded by a /* and followed by a */.

Examples

```
// This is a single-line comment.
/* This is a multiple-line comment. It can be of any length, and
you can put whatever you want here. */
```

continue

Restarts a `while`, `do-while`, `for`, or `label` statement.

Implemented in	JavaScript 1.0, NES 2.0
ECMA version	ECMA-262

Syntax `continue [`*`label`*`]`

Parameter

`label`	Identifier associated with the label of the statement.

Description In contrast to the `break` statement, `continue` does not terminate the execution of the loop entirely: instead,

- In a `while` loop, it jumps back to the `condition`.
- In a `for` loop, it jumps to the `update` expression.

The `continue` statement can now include an optional label that allows the program to terminate execution of a labeled statement and continue to the specified labeled statement. This type of continue must be in a looping statement identified by the label used by `continue`.

Examples **Example 1.** The following example shows a `while` loop that has a `continue` statement that executes when the value of `i` is 3. Thus, `n` takes on the values 1, 3, 7, and 12.

```
i = 0
n = 0
while (i < 5) {
   i++
   if (i == 3)
      continue
   n += i
}
```

Example 2. In the following example, a statement labeled `checkiandj` contains a statement labeled `checkj`. If `continue` is encountered, the program continues at the top of the `checkj` statement. Each time `continue` is encountered, `checkj` reiterates until its condition returns false. When false is returned, the remainder of the `checkiandj` statement is completed. `checkiandj` reiterates until its condition returns false. When false is returned, the program continues at the statement following checkiandj.

If `continue` had a label of `checkiandj`, the program would continue at the top of the checkiandj statement.

```
checkiandj :
while (i<4) {
   document.write(i + "<BR>");
   i+=1;

   checkj :
   while (j>4) {
      document.write(j + "<BR>");
      j-=1;
      if ((j%2)==0)
         continue checkj;
      document.write(j + " is odd.<BR>");
   }
   document.write("i = " + i + "<br>");
   document.write("j = " + j + "<br>");
}
```

See also `break`, `label`

do...while

Executes the specified statements until the test condition evaluates to false. Statements execute at least once.

Implemented in	JavaScript 1.2, NES 3.0

Syntax

```
do
   statements
while (condition);
```

Parameters

`statements`	Block of statements that is executed at least once and is re-executed each time the condition evaluates to true.
`condition`	Evaluated after each pass through the loop. If `condition` evaluates to true, the statements in the preceding block are re-executed. When `condition` evaluates to false, control passes to the statement following `do while`.

Examples In the following example, the `do` loop iterates at least once and reiterates until i is no longer less than 5.

```
do {
   i+=1
   document.write(i);
while (i<5);
```

export

Allows a signed script to provide properties, functions, and objects to other signed or unsigned scripts.

Implemented in	JavaScript 1.2, NES 3.0

Syntax

```
export name1, name2, ..., nameN
export *
```

Parameters

`nameN`	List of properties, functions, and objects to be exported.
`*`	Exports all properties, functions, and objects from the script.

Description Typically, information in a signed script is available only to scripts signed by the same principals. By exporting properties, functions, or objects, a signed script makes this information available to any script (signed or unsigned). The receiving script uses the companion import statement to access the information.

See also `import`

for

Creates a loop that consists of three optional expressions, enclosed in parentheses and separated by semicolons, followed by a block of statements executed in the loop.

Implemented in	JavaScript 1.0, NES 2.0
ECMA version	ECMA-262

Syntax

```
for ([initial-expression]; [condition]; [increment-expression])
{
   statements
}
```

Parameters

`initial-expression`	Statement or variable declaration. Typically used to initialize a counter variable. This expression may optionally declare new variables with the `var` keyword. These variables are local to the function, not to the loop.
`condition`	Evaluated on each pass through the loop. If this condition evaluates to true, the statements in `statements` are performed. This conditional test is optional. If omitted, the condition always evaluates to true.
`increment-expression`	Generally used to update or increment the counter variable.
`statements`	Block of statements that are executed as long as condition evaluates to true. This can be a single statement or multiple statements. Although not required, it is good practice to indent these statements from the beginning of the `for` statement.

Examples The following `for` statement starts by declaring the variable `i` and initializing it to 0. It checks that `i` is less than nine, performs the two succeeding statements, and increments `i` by 1 after each pass through the loop.

```
for (var i = 0; i < 9; i++) {
   n += i
   myfunc(n)
}
```

for...in

Iterates a specified variable over all the properties of an object. For each distinct property, JavaScript executes the specified statements.

Implemented in	JavaScript 1.0, NES 2.0
ECMA version	ECMA-262

Syntax

```
for (variable in object) {
   statements
}
```

Parameters

`variable`	Variable to iterate over every property, declared with the `var` keyword. This variable is local to the function, not to the loop.
`object`	Object for which the properties are iterated.
`statements`	Specifies the statements to execute for each property.

Examples The following function takes as its argument an object and the object's name. It then iterates over all the object's properties and returns a string that lists the property names and their values.

```
function show_props(obj, objName) {
   var result = ""
   for (var i in obj) {
      result += objName + "." + i + " = " + obj[i] + "\n"
   }
   return result
}
```

function

Declares a function with the specified parameters. Acceptable parameters include strings, numbers, and objects.

Implemented in	JavaScript 1.0, NES 2.0
ECMA version	ECMA-262

Syntax

```
function name([param] [, param] [..., param]) {
   statements
}
```

You can also define functions using the `Function` constructor; see "Function" on page 169.

Parameters

`name`	The function name.
`param`	The name of an argument to be passed to the function. A function can have up to 255 arguments.
`statements`	The statements which comprise the body of the function.

Description To return a value, the function must have a `return` statement that specifies the value to return.

A function created with the `function` statement is a `Function` object and has all the properties, methods, and behavior of `Function` objects. See "Function" on page 169 for detailed information on functions.

Examples The following code declares a function that returns the total dollar amount of sales, when given the number of units sold of products a, b, and c.

```
function calc_sales(units_a, units_b, units_c) {
   return units_a*79 + units_b*129 + units_c*699
}
```

See also "Function" on page 169

if...else

Executes a set of statements if a specified condition is true. If the condition is false, another set of statements can be executed.

Implemented in	JavaScript 1.0, NES 2.0
ECMA version	ECMA-262

Syntax

```
if (condition) {
   statements1
}
[else {
   statements2
}]
```

Parameters

`condition`	Can be any JavaScript expression that evaluates to true or false. Parentheses are required around the condition. If condition evaluates to true, the statements in `statements1` are executed.
`statements1`, `statements2`	Can be any JavaScript statements, including further nested `if` statements. Multiple statements must be enclosed in braces.

Description

You should not use simple assignments in a conditional statement. For example, do not use the following code:

```
if(x = y)
{
   /* do the right thing */
}
```

If you need to use an assignment in a conditional statement, put additional parentheses around the assignment. For example, use `if( (x = y) )`.

Backward Compatibility

JavaScript 1.2 and earlier versions. You can use simple assignments in a conditional statement. An assignment operator in a conditional statement is converted to an equality operator. For example, `if(x = y)` is converted to `if(x == y)`. In Navigator, this expression also displays a dialog box with the message "Test for equality (==) mistyped as assignment (=)? Assuming equality test."

Examples

```
if (cipher_char == from_char) {
   result = result + to_char
   x++}
else
   result = result + clear_char
```

import

Allows a script to import properties, functions, and objects from a signed script that has exported the information.

Implemented in	JavaScript 1.2, NES 3.0

Syntax

```
import objectName.name1, objectName.name2, ..., objectName.nameN
import objectName.*
```

Parameters

`objectName`	Name of the object that will receive the imported names.
`name1,` `name2,` `nameN`	List of properties, functions, and objects to import from the export file.
`*`	Imports all properties, functions, and objects from the export script.

Description The `objectName` parameter is the name of the object that will receive the imported names. For example, if `f` and `p` have been exported, and if `obj` is an object from the importing script, the following code makes `f` and `p` accessible in the importing script as properties of `obj`.

```
import obj.f, obj.p
```

Typically, information in a signed script is available only to scripts signed by the same principals. By exporting (using the `export` statement) properties, functions, or objects, a signed script makes this information available to any script (signed or unsigned). The receiving script uses the `import` statement to access the information.

The script must load the export script into a window, frame, or layer before it can import and use any exported properties, functions, and objects.

See also `export`

label

Provides a statement with an identifier that lets you refer to it elsewhere in your program.

Implemented in	JavaScript 1.2, NES 3.0

For example, you can use a label to identify a loop, and then use the `break` or `continue` statements to indicate whether a program should interrupt the loop or continue its execution.

Syntax

```
label :
   statements
```

Parameter

`label`	Any JavaScript identifier that is not a reserved word.
`statements`	Block of statements. `break` can be used with any labeled statement, and continue can be used with looping labeled statements.

Examples For an example of a label statement using `break`, see `break`. For an example of a label statement using `continue`, see `continue`.

See also `break`, `continue`

return

Specifies the value to be returned by a function.

Implemented in	JavaScript 1.0, NES 2.0
ECMA version	ECMA-262

Syntax

```
return expression
```

Parameters

`expression`	The expression to return.

Examples The following function returns the square of its argument, `x`, where `x` is a number.

```
function square(x) {
   return x * x
}
```

switch

Allows a program to evaluate an expression and attempt to match the expression's value to a case label.

Implemented in JavaScript 1.2, NES 3.0

Syntax

```
switch (expression){
   case label :
    statements;
    break;
   case label :
    statements;
    break;
   ...
   default : statements;
}
```

Parameters

`expression`	Value matched against label.
`label`	Identifier used to match against expression.
`statements`	Block of statements that is executed once if `expression` matches `label`.

Description If a match is found, the program executes the associated statement. If multiple cases match the provided value, the first case that matches is selected, even if the cases are not equal to each other.

The program first looks for a label matching the value of expression and then executes the associated statement. If no matching label is found, the program looks for the optional default statement, and if found, executes the associated statement. If no default statement is found, the program continues execution at the statement following the end of `switch`.

The optional `break` statement associated with each case label ensures that the program breaks out of switch once the matched statement is executed and continues execution at the statement following switch. If `break` is omitted, the program continues execution at the next statement in the `switch` statement.

Examples In the following example, if `expression` evaluates to "Bananas", the program matches the value with case "Bananas" and executes the associated statement. When `break` is encountered, the program breaks out of `switch` and executes the statement following `switch`. If `break` were omitted, the statement for case "Cherries" would also be executed.

```
switch (i) {
   case "Oranges" :
      document.write("Oranges are $0.59 a pound.<BR>");
      break;
   case "Apples" :
      document.write("Apples are $0.32 a pound.<BR>");
      break;
   case "Bananas" :
      document.write("Bananas are $0.48 a pound.<BR>");
      break;
   case "Cherries" :
      document.write("Cherries are $3.00 a pound.<BR>");
      break;
   default :
      document.write("Sorry, we are out of " + i + ".<BR>");
}
document.write("Is there anything else you'd like?<BR>");
```

var

Declares a variable, optionally initializing it to a value.

Implemented in	JavaScript 1.0, NES 2.0
ECMA version	ECMA-262

Syntax `var` *`varname`* `[=` *`value`*`] [...,` *`varname`* `[=` *`value`*`] ]`

Parameters

`varname`	Variable name. It can be any legal identifier.
`value`	Initial value of the variable and can be any legal expression.

Description The scope of a variable is the current function or, for variables declared outside a function, the current application.

Using `var` outside a function is optional; you can declare a variable by simply assigning it a value. However, it is good style to use `var`, and it is necessary in functions in the following situations:

- If a global variable of the same name exists.
- If recursive or multiple functions use variables with the same name.

Examples

```
var num_hits = 0, cust_no = 0
```

while

Creates a loop that evaluates an expression, and if it is true, executes a block of statements. The loop then repeats, as long as the specified condition is true.

Implemented in	JavaScript 1.0, NES 2.0
ECMA version	ECMA-262

Syntax

```
while (condition) {
   statements
}
```

Parameters

`condition`	Evaluated before each pass through the loop. If this condition evaluates to true, the statements in the succeeding block are performed. When `condition` evaluates to false, execution continues with the statement following `statements`.
`statements`	Block of statements that are executed as long as the condition evaluates to true. Although not required, it is good practice to indent these statements from the beginning of the statement.

Examples The following `while` loop iterates as long as `n` is less than three.

```
n = 0
x = 0
while(n < 3) {
   n ++
   x += n
}
```

Each iteration, the loop increments `n` and adds it to `x`. Therefore, `x` and `n` take on the following values:

- After the first pass: n = 1 and x = 1
- After the second pass: n = 2 and x = 3
- After the third pass: n = 3 and x = 6

After completing the third pass, the condition `n < 3` is no longer true, so the loop terminates.

with

Establishes the default object for a set of statements.

Implemented in	JavaScript 1.0, NES 2.0
ECMA version	ECMA-262

Syntax

```
with (object){
   statements
}
```

Parameters

`object`	Specifies the default object to use for the statements. The parentheses around object are required.
`statements`	Any block of statements.

Description JavaScript looks up any unqualified names within the set of statements to determine if the names are properties of the default object. If an unqualified name matches a property, then the property is used in the statement; otherwise, a local or global variable is used.

Examples The following `with` statement specifies that the `Math` object is the default object. The statements following the `with` statement refer to the `PI` property and the `cos` and `sin` methods, without specifying an object. JavaScript assumes the `Math` object for these references.

```
var a, x, y
var r=10
with (Math) {
   a = PI * r * r
   x = r * cos(PI)
   y = r * sin(PI/2)
}
```

Chapter

5

Operators

JavaScript has assignment, comparison, arithmetic, bitwise, logical, string, and special operators. This chapter describes the operators and contains information about operator precedence.

The following table summarizes the JavaScript operators.

Table 5.1 JavaScript operators.

Operator category	Operator	Description
Arithmetic Operators	`+`	(Addition) Adds 2 numbers.
	`++`	(Increment) Adds one to a variable representing a number (returning either the new or old value of the variable)
	`-`	(Unary negation, subtraction) As a unary operator, negates the value of its argument. As a binary operator, subtracts 2 numbers.
	`--`	(Decrement) Subtracts one from a variable representing a number (returning either the new or old value of the variable)
	`*`	(Multiplication) Multiplies 2 numbers.
	`/`	(Division) Divides 2 numbers.
	`%`	(Modulus) Computes the integer remainder of dividing 2 numbers.
String Operators	`+`	(String addition) Concatenates 2 strings.
	`+=`	Concatenates 2 strings and assigns the result to the first operand.

Table 5.1 JavaScript operators. (Continued)

Operator category	Operator	Description
Logical Operators	&&	(Logical AND) Returns the first operand if it can be converted to false; otherwise, returns the second operand. Thus, when used with Boolean values, && returns true if both operands are true; otherwise, returns false.
	||	(Logical OR) Returns the first operand if it can be converted to true; otherwise, returns the second operand. Thus, when used with Boolean values, || returns true if either operand is true; if both are false, returns false.
	!	(Logical NOT) Returns false if its single operand can be converted to true; otherwise, returns true.
Bitwise Operators	&	(Bitwise AND) Returns a one in each bit position if bits of both operands are ones.
	^	(Bitwise XOR) Returns a one in a bit position if bits of one but not both operands are one.
	|	(Bitwise OR) Returns a one in a bit if bits of either operand is one.
	~	(Bitwise NOT) Flips the bits of its operand.
	<<	(Left shift) Shifts its first operand in binary representation the number of bits to the left specified in the second operand, shifting in zeros from the right.
	>>	(Sign-propagating right shift) Shifts the first operand in binary representation the number of bits to the right specified in the second operand, discarding bits shifted off.
	>>>	(Zero-fill right shift) Shifts the first operand in binary representation the number of bits to the right specified in the second operand, discarding bits shifted off, and shifting in zeros from the left.

Table 5.1 JavaScript operators. (Continued)

Operator category	Operator	Description
Assignment Operators	=	Assigns the value of the second operand to the first operand.
	+=	Adds 2 numbers and assigns the result to the first.
	-=	Subtracts 2 numbers and assigns the result to the first.
	*=	Multiplies 2 numbers and assigns the result to the first.
	/=	Divides 2 numbers and assigns the result to the first.
	%=	Computes the modulus of 2 numbers and assigns the result to the first.
	&=	Performs a bitwise AND and assigns the result to the first operand.
	^=	Performs a bitwise XOR and assigns the result to the first operand.
	|=	Performs a bitwise OR and assigns the result to the first operand.
	<<=	Performs a left shift and assigns the result to the first operand.
	>>=	Performs a sign-propagating right shift and assigns the result to the first operand.
	>>>=	Performs a zero-fill right shift and assigns the result to the first operand.
Comparison Operators	==	Returns true if the operands are equal.
	!=	Returns true if the operands are not equal.
	===	Returns true if the operands are equal and of the same type.
	!==	Returns true if the operands are not equal and/or not of the same type.
	>	Returns true if the left operand is greater than the right operand.
	>=	Returns true if the left operand is greater than or equal to the right operand.
	<	Returns true if the left operand is less than the right operand.
	<=	Returns true if the left operand is less than or equal to the right operand.

Table 5.1 JavaScript operators. (Continued)

Operator category	Operator	Description
Special Operators	`?:`	Performs a simple `"if...then...else"`
	`,`	Evaluates two expressions and returns the result of the second expression.
	`delete`	Deletes an object, an object's property, or an element at a specified index in an array.
	`new`	Creates an instance of a user-defined object type or of one of the built-in object types.
	`this`	Keyword that you can use to refer to the current object.
	`typeof`	Returns a string indicating the type of the unevaluated operand.
	`void`	Specifies an expression to be evaluated without returning a value.

Assignment Operators

An assignment operator assigns a value to its left operand based on the value of its right operand.

Implemented in	JavaScript 1.0
ECMA version	ECMA-262

The basic assignment operator is equal (=), which assigns the value of its right operand to its left operand. That is, x = y assigns the value of y to x. The other assignment operators are usually shorthand for standard operations, as shown in the following table.

Table 5.2 Assignment operators

Shorthand operator	Meaning
`x += y`	`x = x + y`
`x -= y`	`x = x - y`
`x *= y`	`x = x * y`
`x /= y`	`x = x / y`
`x %= y`	`x = x % y`

Table 5.2 Assignment operators

Shorthand operator	Meaning
`x <<= y`	`x = x << y`
`x >>= y`	`x = x >> y`
`x >>>= y`	`x = x >>> y`
`x &= y`	`x = x & y`
`x ^= y`	`x = x ^ y`
`x |= y`	`x = x | y`

In unusual situations, the assignment operator is not identical to the Meaning expression in Table 5.2. When the left operand of an assignment operator itself contains an assignment operator, the left operand is evaluated only once. For example:

```
a[i++] += 5 //i is evaluated only once
a[i++] = a[i++] + 5 //i is evaluated twice
```

Comparison Operators

A comparison operator compares its operands and returns a logical value based on whether the comparison is true.

Implemented in	JavaScript 1.0 JavaScript 1.3: Added the `===` and `!==` operators.
ECMA version	ECMA-262 includes all comparison operators except === and !==.

The operands can be numerical or string values. Strings are compared based on standard lexicographical ordering, using Unicode values.

A Boolean value is returned as the result of the comparison.

- Two strings are equal when they have the same sequence of characters, same length, and same characters in corresponding positions.
- Two numbers are equal when they are numerically equal (have the same number value). NaN is not equal to anything, including NaN. Positive and negative zeros are equal.

- Two objects are equal if they refer to the same Object.
- Two Boolean operands are equal if they are both `true` or `false`.
- Null and Undefined types are equal.

The following table describes the comparison operators.

Table 5.3 Comparison operators

Operator	Description	Examples returning true[a]

Equal (`==`)	Returns true if the operands are equal. If the two operands are not of the same type, JavaScript attempts to convert the operands to an appropriate type for the comparison.	`3 == var1`
		`"3" == var1`
		`3 == '3'`

Not equal (`!=`)	Returns true if the operands are not equal. If the two operands are not of the same type, JavaScript attempts to convert the operands to an appropriate type for the comparison.	`var1 != 4`
		`var1 != "3"`

Strict equal (`===`)	Returns true if the operands are equal and of the same type.	`3 === var1`

Strict not equal (`!==`)	Returns true if the operands are not equal and/or not of the same type.	`var1 !== "3"`
		`3 !== '3'`

Greater than (`>`)	Returns true if the left operand is greater than the right operand.	`var2 > var1`

Greater than or equal (`>=`)	Returns true if the left operand is greater than or equal to the right operand.	`var2 >= var1`
		`var1 >= 3`

Less than (`<`)	Returns true if the left operand is less than the right operand.	`var1 < var2`

Less than or equal (`<=`)	Returns true if the left operand is less than or equal to the right operand.	`var1 <= var2`
		`var2 <= 5`

a. These examples assume that `var1` has been assigned the value 3 and `var2` has been assigned the value 4.

Using the Equality Operators

The standard equality operators (== and !=) compare two operands without regard to their type. The strict equality operators (=== and !==) perform equality comparisons on operands of the same type. Use strict equality operators if the operands must be of a specific type as well as value or if the exact type of the operands is important. Otherwise, use the standard equality operators, which allow you to compare the identity of two operands even if they are not of the same type.

When type conversion is needed, JavaScript converts `String`, `Number`, `Boolean`, or `Object` operands as follows.

- When comparing a number and a string, the string is converted to a number value. JavaScript attempts to convert the string numeric literal to a `Number` type value. First, a mathematical value is derived from the string numeric literal. Next, this value is rounded to nearest `Number` type value.
- If one of the operands is `Boolean`, the Boolean operand is converted to 1 if it is `true` and +0 if it is `false`.
- If an object is compared with a number or string, JavaScript attempts to return the default value for the object. Operators attempt to convert the object to a primitive value, a `String` or `Number` value, using the `valueOf` and `toString` methods of the objects. If this attempt to convert the object fails, a runtime error is generated.

Backward Compatibility

The behavior of the standard equality operators (== and !=) depends on the JavaScript version.

JavaScript 1.2. The standard equality operators (== and !=) do not perform a type conversion before the comparison is made. The strict equality operators (=== and !==) are unavailable.

JavaScript 1.1 and earlier versions. The standard equality operators (== and !=) perform a type conversion before the comparison is made. The strict equality operators (=== and !==) are unavailable.

Arithmetic Operators

Arithmetic operators take numerical values (either literals or variables) as their operands and return a single numerical value. The standard arithmetic operators are addition (+), subtraction (-), multiplication (*), and division (/).

Implemented in	JavaScript 1.0
ECMA version	ECMA-262

These operators work as they do in most other programming languages, except the / operator returns a floating-point division in JavaScript, not a truncated division as it does in languages such as C or Java. For example:

```
1/2 //returns 0.5 in JavaScript
1/2 //returns 0 in Java
```

% (Modulus)

The modulus operator is used as follows:

```
var1 % var2
```

The modulus operator returns the first operand modulo the second operand, that is, `var1` modulo `var2`, in the preceding statement, where `var1` and `var2` are variables. The modulo function is the integer remainder of dividing `var1` by `var2`. For example, 12 % 5 returns 2.

++ (Increment)

The increment operator is used as follows:

`var++` or `++var`

This operator increments (adds one to) its operand and returns a value. If used postfix, with operator after operand (for example, x++), then it returns the value before incrementing. If used prefix with operator before operand (for example, ++x), then it returns the value after incrementing.

For example, if x is three, then the statement `y = x++` sets `y` to 3 and increments `x` to 4. If `x` is 3, then the statement `y = ++x` increments `x` to 4 and sets `y` to 4.

-- (Decrement)

The decrement operator is used as follows:

`var--` or `--var`

This operator decrements (subtracts one from) its operand and returns a value. If used postfix (for example, x--), then it returns the value before decrementing. If used prefix (for example, --x), then it returns the value after decrementing.

For example, if x is three, then the statement `y = x--` sets `y` to 3 and decrements `x` to 2. If `x` is 3, then the statement `y = --x` decrements `x` to 2 and sets `y` to 2.

- (Unary Negation)

The unary negation operator precedes its operand and negates it. For example, `y = -x` negates the value of `x` and assigns that to `y`; that is, if `x` were 3, `y` would get the value -3 and `x` would retain the value 3.

Bitwise Operators

Bitwise operators treat their operands as a set of 32 bits (zeros and ones), rather than as decimal, hexadecimal, or octal numbers. For example, the decimal number nine has a binary representation of 1001. Bitwise operators perform their operations on such binary representations, but they return standard JavaScript numerical values.

The following table summarizes JavaScript's bitwise operators:

Table 5.4 Bitwise operators

Operator	Usage	Description
Bitwise AND	`a & b`	Returns a one in each bit position for which the corresponding bits of both operands are ones.
Bitwise OR	`a | b`	Returns a one in each bit position for which the corresponding bits of either or both operands are ones.
Bitwise XOR	`a ^ b`	Returns a one in each bit position for which the corresponding bits of either but not both operands are ones.
Bitwise NOT	`~ a`	Inverts the bits of its operand.
Left shift	`a << b`	Shifts `a` in binary representation `b` bits to left, shifting in zeros from the right.
Sign-propagating right shift	`a >> b`	Shifts `a` in binary representation `b` bits to right, discarding bits shifted off.
Zero-fill right shift	`a >>> b`	Shifts `a` in binary representation `b` bits to the right, discarding bits shifted off, and shifting in zeros from the left.

Bitwise Logical Operators

Implemented in	JavaScript 1.0
ECMA version	ECMA-262

Conceptually, the bitwise logical operators work as follows:

- The operands are converted to thirty-two-bit integers and expressed by a series of bits (zeros and ones).
- Each bit in the first operand is paired with the corresponding bit in the second operand: first bit to first bit, second bit to second bit, and so on.
- The operator is applied to each pair of bits, and the result is constructed bitwise.

For example, the binary representation of nine is 1001, and the binary representation of fifteen is 1111. So, when the bitwise operators are applied to these values, the results are as follows:

- 15 & 9 yields 9 (1111 & 1001 = 1001)
- 15 | 9 yields 15 (1111 | 1001 = 1111)
- 15 ^ 9 yields 6 (1111 ^ 1001 = 0110)

Bitwise Shift Operators

Implemented in	JavaScript 1.0
ECMA version	ECMA-262

The bitwise shift operators take two operands: the first is a quantity to be shifted, and the second specifies the number of bit positions by which the first operand is to be shifted. The direction of the shift operation is controlled by the operator used.

Shift operators convert their operands to thirty-two-bit integers and return a result of the same type as the left operator.

<< (Left Shift)

This operator shifts the first operand the specified number of bits to the left. Excess bits shifted off to the left are discarded. Zero bits are shifted in from the right.

For example, `9<<2` yields thirty-six, because 1001 shifted two bits to the left becomes 100100, which is thirty-six.

>> (Sign-Propagating Right Shift)

This operator shifts the first operand the specified number of bits to the right. Excess bits shifted off to the right are discarded. Copies of the leftmost bit are shifted in from the left.

For example, 9>>2 yields two, because 1001 shifted two bits to the right becomes 10, which is two. Likewise, -9>>2 yields -3, because the sign is preserved.

>>> (Zero-Fill Right Shift)

This operator shifts the first operand the specified number of bits to the right. Excess bits shifted off to the right are discarded. Zero bits are shifted in from the left.

For example, 19>>>2 yields four, because 10011 shifted two bits to the right becomes 100, which is four. For non-negative numbers, zero-fill right shift and sign-propagating right shift yield the same result.

Logical Operators

Logical operators are typically used with Boolean (logical) values; when they are, they return a Boolean value. However, the && and || operators actually return the value of one of the specified operands, so if these operators are used with non-Boolean values, they may return a non-Boolean value.

Implemented in	JavaScript 1.0
ECMA version	ECMA-262

The logical operators are described in the following table.

Table 5.5 Logical operators

Operator	Usage	Description
&&	*`expr1 && expr2`*	(Logical AND) Returns `expr1` if it can be converted to false; otherwise, returns `expr2`. Thus, when used with Boolean values, && returns true if both operands are true; otherwise, returns false.
||	*`expr1 || expr2`*	(Logical OR) Returns `expr1` if it can be converted to true; otherwise, returns `expr2`. Thus, when used with Boolean values, || returns true if either operand is true; if both are false, returns false.
!	`!`*`expr`*	(Logical NOT) Returns false if its single operand can be converted to true; otherwise, returns true.

Examples of expressions that can be converted to false are those that evaluate to null, 0, the empty string (“”), or undefined.

Even though the && and || operators can be used with operands that are not Boolean values, they can still be considered Boolean operators since their return values can always be converted to Boolean values.

Short-Circuit Evaluation. As logical expressions are evaluated left to right, they are tested for possible "short-circuit" evaluation using the following rules:

- `false` && *anything* is short-circuit evaluated to false.
- `true ||` *anything* is short-circuit evaluated to true.

The rules of logic guarantee that these evaluations are always correct. Note that the *anything* part of the above expressions is not evaluated, so any side effects of doing so do not take effect.

Backward Compatibility

JavaScript 1.0 and 1.1. The && and || operators behave as follows:

Operator	Behavior
&&	If the first operand (`expr1`) can be converted to false, the && operator returns false rather than the value of `expr1`.
||	If the first operand (`expr1`) can be converted to true, the || operator returns true rather than the value of `expr1`.

Examples

The following code shows examples of the && (logical AND) operator.

```
a1=true && true       // t && t returns true
a2=true && false      // t && f returns false
a3=false && true      // f && t returns false
a4=false && (3 == 4)  // f && f returns false
a5="Cat" && "Dog"     // t && t returns Dog
a6=false && "Cat"     // f && t returns false
a7="Cat" && false     // t && f returns false
```

The following code shows examples of the || (logical OR) operator.

```
o1=true || true       // t || t returns true
o2=false || true      // f || t returns true
o3=true || false      // t || f returns true
o4=false || (3 == 4)  // f || f returns false
o5="Cat" || "Dog"     // t || t returns Cat
o6=false || "Cat"     // f || t returns Cat
o7="Cat" || false     // t || f returns Cat
```

The following code shows examples of the ! (logical NOT) operator.

```
n1=!true              // !t returns false
n2=!false             // !f returns true
n3=!"Cat"             // !t returns false
```

String Operators

In addition to the comparison operators, which can be used on string values, the concatenation operator (+) concatenates two string values together, returning another string that is the union of the two operand strings. For example, `"my " + "string"` returns the string `"my string"`.

Implemented in	JavaScript 1.0
ECMA version	ECMA-262

The shorthand assignment operator `+=` can also be used to concatenate strings. For example, if the variable `mystring` has the value "alpha," then the expression `mystring += "bet"` evaluates to "alphabet" and assigns this value to `mystring`.

Special Operators

?: (Conditional operator)

The conditional operator is the only JavaScript operator that takes three operands. This operator is frequently used as a shortcut for the `if` statement.

Implemented in	JavaScript 1.0
ECMA version	ECMA-262

Syntax *`condition ? expr1 : expr2`*

Parameters

`condition`	An expression that evaluates to `true` or `false`
`expr1, expr2`	Expressions with values of any type.

Description If `condition` is `true`, the operator returns the value of `expr1`; otherwise, it returns the value of `expr2`. For example, to display a different message based on the value of the `isMember` variable, you could use this statement:

```
document.write ("The fee is " + (isMember ? "$2.00" : "$10.00"))
```

, (Comma operator)

The comma operator evaluates both of its operands and returns the value of the second operand.

Implemented in	JavaScript 1.0
ECMA version	ECMA-262

Syntax *expr1*, *expr2*

Parameters

expr1, expr2	Any expressions

Description You can use the comma operator when you want to include multiple expressions in a location that requires a single expression. The most common usage of this operator is to supply multiple parameters in a `for` loop.

For example, if `a` is a 2-dimensional array with 10 elements on a side, the following code uses the comma operator to increment two variables at once. The code prints the values of the diagonal elements in the array:

```
for (var i=0, j=9; i <= 9; i++, j--)
   document.writeln("a["+i+","+j+"]= " + a[i,j])
```

delete

The delete operator deletes an object, an object's property, or an element at a specified index in an array.

Implemented in	JavaScript 1.2, NES 3.0
ECMA version	ECMA-262

Syntax

```
delete objectName
delete objectName.property
delete objectName[index]
delete property // legal only within a with statement
```

Parameters

objectName	The name of an object.
property	The property to delete.
index	An integer representing the array index to delete.

Description The fourth form is legal only within a `with` statement, to delete a property from an object.

You can use the `delete` operator to delete variables declared implicitly but not those declared with the `var` statement.

If the `delete` operator succeeds, it sets the property or element to `undefined. The delete` operator returns true if the operation is possible; it returns false if the operation is not possible.

```
x=42
var y= 43
myobj=new Number()
myobj.h=4      // create property h
delete x       // returns true (can delete if declared implicitly)
delete y       // returns false (cannot delete if declared with var)
delete Math.PI // returns false (cannot delete predefined properties)
delete myobj.h // returns true (can delete user-defined properties)
delete myobj   // returns true (can delete objects)
```

Deleting array elements. When you delete an array element, the array length is not affected. For example, if you delete a[3], a[4] is still a[4] and a[3] is undefined.

When the `delete` operator removes an array element, that element is no longer in the array. In the following example, trees[3] is removed with `delete`.

```
trees=new Array("redwood","bay","cedar","oak","maple")
delete trees[3]
if (3 in trees) {
   // this does not get executed
}
```

If you want an array element to exist but have an undefined value, use the `undefined` keyword instead of the `delete` operator. In the following example, trees[3] is assigned the value undefined, but the array element still exists:

```
trees=new Array("redwood","bay","cedar","oak","maple")
trees[3]=undefined
if (3 in trees) {
   // this gets executed
}
```

new

The new operator creates an instance of a user-defined object type or of one of the built-in object types that has a constructor function.

Implemented in	JavaScript 1.0
ECMA version	ECMA-262

Syntax `objectName = new objectType (param1 [,param2] ...[,paramN])`

Parameters

`objectName`	Name of the new object instance.
`objectType`	Object type. It must be a function that defines an object type.
`param1...paramN`	Property values for the object. These properties are parameters defined for the `objectType` function.

Description Creating a user-defined object type requires two steps:

1. Define the object type by writing a function.
2. Create an instance of the object with `new`.

To define an object type, create a function for the object type that specifies its name, properties, and methods. An object can have a property that is itself another object. See the examples below.

You can always add a property to a previously defined object. For example, the statement `car1.color = "black"` adds a property `color` to `car1`, and assigns it a value of `"black"`. However, this does not affect any other objects. To add the new property to all objects of the same type, you must add the property to the definition of the `car` object type.

You can add a property to a previously defined object type by using the `Function.prototype` property. This defines a property that is shared by all objects created with that function, rather than by just one instance of the object type. The following code adds a `color` property to all objects of type `car`, and then assigns a value to the `color` property of the object `car1`. For more information, see `prototype`

```
Car.prototype.color=null
car1.color="black"
birthday.description="The day you were born"
```

Examples **Example 1: Object type and object instance.** Suppose you want to create an object type for cars. You want this type of object to be called `car`, and you want it to have properties for make, model, and year. To do this, you would write the following function:

```
function car(make, model, year) {
   this.make = make
   this.model = model
   this.year = year
}
```

Now you can create an object called `mycar` as follows:

```
mycar = new car("Eagle", "Talon TSi", 1993)
```

This statement creates `mycar` and assigns it the specified values for its properties. Then the value of `mycar.make` is the string `"Eagle"`, `mycar.year` is the integer `1993`, and so on.

You can create any number of `car` objects by calls to `new`. For example,

```
kenscar = new car("Nissan", "300ZX", 1992)
```

Example 2: Object property that is itself another object. Suppose you define an object called `person` as follows:

```
function person(name, age, sex) {
   this.name = name
   this.age = age
   this.sex = sex
}
```

And then instantiate two new `person` objects as follows:

```
rand = new person("Rand McNally", 33, "M")
ken = new person("Ken Jones", 39, "M")
```

Then you can rewrite the definition of `car` to include an owner property that takes a `person` object, as follows:

```
function car(make, model, year, owner) {
   this.make = make;
   this.model = model;
   this.year = year;
   this.owner = owner;
}
```

To instantiate the new objects, you then use the following:

```
car1 = new car("Eagle", "Talon TSi", 1993, rand);
car2 = new car("Nissan", "300ZX", 1992, ken)
```

Instead of passing a literal string or integer value when creating the new objects, the above statements pass the objects `rand` and `ken` as the parameters for the owners. To find out the name of the owner of `car2`, you can access the following property:

```
car2.owner.name
```

this

The this keyword refers to the current object. In general, in a method `this` refers to the calling object.

Implemented in	JavaScript 1.0
ECMA version	ECMA-262

Syntax `this[.`*`propertyName`*`]`

Examples Suppose a function called `validate` validates an object's value property, given the object and the high and low values:

```
function validate(obj, lowval, hival) {
   if ((obj.value < lowval) || (obj.value > hival))
      alert("Invalid Value!")
}
```

You could call `validate` in each form element's `onChange` event handler, using `this` to pass it the form element, as in the following example:

```
<B>Enter a number between 18 and 99:</B>
<INPUT TYPE = "text" NAME = "age" SIZE = 3
   onChange="validate(this, 18, 99)">
```

typeof

The `typeof` operator is used in either of the following ways:

```
1. typeof operand
2. typeof (operand)
```

The `typeof` operator returns a string indicating the type of the unevaluated operand. `operand` is the string, variable, keyword, or object for which the type is to be returned. The parentheses are optional.

Implemented in	JavaScript 1.1
ECMA version	ECMA-262

Suppose you define the following variables:

```
var myFun = new Function("5+2")
var shape="round"
var size=1
var today=new Date()
```

The `typeof` operator returns the following results for these variables:

```
typeof myFun is object
typeof shape is string
typeof size is number
typeof today is object
typeof dontExist is undefined
```

For the keywords `true` and `null`, the `typeof` operator returns the following results:

```
typeof true is boolean
typeof null is object
```

For a number or string, the `typeof` operator returns the following results:

```
typeof 62 is number
typeof 'Hello world' is string
```

For property values, the `typeof` operator returns the type of value the property contains:

```
typeof document.lastModified is string
typeof window.length is number
typeof Math.LN2 is number
```

For methods and functions, the `typeof` operator returns results as follows:

```
typeof blur is function
typeof eval is function
typeof parseInt is function
typeof shape.split is function
```

For predefined objects, the `typeof` operator returns results as follows:

```
typeof Date is function
typeof Function is function
typeof Math is function
typeof Option is function
typeof String is function
```

void

The void operator is used in either of the following ways:

```
1. void (expression)
2. void expression
```

The void operator specifies an expression to be evaluated without returning a value. `expression` is a JavaScript expression to evaluate. The parentheses surrounding the expression are optional, but it is good style to use them.

Implemented in	JavaScript 1.1
ECMA version	ECMA-262

You can use the `void` operator to specify an expression as a hypertext link. The expression is evaluated but is not loaded in place of the current document.

The following code creates a hypertext link that does nothing when the user clicks it. When the user clicks the link, `void(0)` evaluates to 0, but that has no effect in JavaScript.

```
<A HREF="javascript:void(0)">Click here to do nothing</A>
```

The following code creates a hypertext link that submits a form when the user clicks it.

```
<A HREF="javascript:void(document.form.submit())">
Click here to submit</A>
```

3 *LiveConnect Class Reference*

Chapter 6

Java Classes, Constructors, and Methods

This chapter documents the Java classes used for LiveConnect, along with their constructors and methods. It is an alphabetical reference for the classes that allow a Java object to access JavaScript code.

This reference is organized as follows:

- Full entries for each class appear in alphabetical order.

  Tables included in the description of each class summarize the constructors and methods of the class.

- Full entries for the constructors and methods of a class appear in alphabetical order after the entry for the class.

JSException

The public class `JSException` extends `Exception`.

```
java.lang.Object
   |
   +----java.lang.Throwable
           |
           +----java.lang.Exception
                   |
                   +----netscape.javascript.JSException
```

Description `JSException` is an exception which is thrown when JavaScript code returns an error.

Constructor Summary The `netscape.javascript.JSException` class has the following constructors:

Constructor	Description
`JSException`	Constructs a `JSException.` You specify whether the `JSException` has a detail message and other information.

The following sections show the declaration and usage of the constructors.

JSException

Constructor. Constructs a `JSException`. You specify whether the `JSException` has a detail message and other information.

Declaration

```
1. public JSException()

2. public JSException(String s)

3. public JSException(String s,
   String filename,
   int lineno,
   String source,
   int tokenIndex)
```

Arguments

`s`	The detail message.
`filename`	The URL of the file where the error occurred, if possible.
`lineno`	The line number if the file, if possible.
`source`	The string containing the JavaScript code being evaluated.
`tokenIndex`	The index into the source string where the error occurred.

Description A detail message is a string that describes this particular exception.

Each form constructs a `JSException` with different information:

- Form 1 of the declaration constructs a `JSException` without a detail message.
- Form 2 of the declaration constructs a `JSException` with a detail message.
- Form 3 of the declaration constructs a `JSException` with a detail message and all the other information that usually comes with a JavaScript error.

JSObject

The public final class `netscape.javascript.JSObject` extends `Object`.

```
java.lang.Object
   |
   +----netscape.javascript.JSObject
```

Description JavaScript objects are wrapped in an instance of the class `netscape.javascript.JSObject` and passed to Java. `JSObject` allows Java to manipulate JavaScript objects.

When a JavaScript object is sent to Java, the runtime engine creates a Java wrapper of type `JSObject`; when a `JSObject` is sent from Java to JavaScript, the runtime engine unwraps it to its original JavaScript object type. The `JSObject` class provides a way to invoke JavaScript methods and examine JavaScript properties.

Any JavaScript data brought into Java is converted to Java data types. When the JSObject is passed back to JavaScript, the object is unwrapped and can be used by JavaScript code. See the *Client-Side JavaScript Guide* for more information about data type conversions.

Method Summary The `netscape.javascript.JSObject` class has the following methods:

Method	Description
`call`	Calls a JavaScript method.
`equals`	Determines if two `JSObject` objects refer to the same instance.
`eval`	Evaluates a JavaScript expression.
`getMember`	Retrieves the value of a property of a JavaScript object.
`getSlot`	Retrieves the value of an array element of a JavaScript object.
`removeMember`	Removes a property of a JavaScript object.
`setMember`	Sets the value of a property of a JavaScript object.
`setSlot`	Sets the value of an array element of a JavaScript object.
`toString`	Converts a `JSObject` to a string.

The `netscape.javascript.JSObject` class has the following static methods:

Method	Description
`getWindow`	Gets a `JSObject` for the window containing the given applet.

The following sections show the declaration and usage of these methods.

call

Method. Calls a JavaScript method. Equivalent to "`this.methodName(args[0], args[1], ...)`" in JavaScript.

Declaration

```
public Object call(String methodName,
   Object args[])
```

equals

Method. Determines if two `JSObject` objects refer to the same instance.

Overrides: `equals` in class `java.lang.Object`

Declaration

```
public boolean equals(Object obj)
```

eval

Method. Evaluates a JavaScript expression. The expression is a string of JavaScript source code which will be evaluated in the context given by "this".

Declaration

```
public Object eval(String s)
```

getMember

Method. Retrieves the value of a property of a JavaScript object. Equivalent to "`this.name`" in JavaScript.

Declaration

```
public Object getMember(String name)
```

getSlot

Method. Retrieves the value of an array element of a JavaScript object. Equivalent to "`this[index]`" in JavaScript.

Declaration

```
public Object getSlot(int index)
```

getWindow

Static method. Returns a `JSObject` for the window containing the given applet. This method is useful in client-side JavaScript only.

Declaration

```
public static JSObject getWindow(Applet applet)
```

removeMember

Method. Removes a property of a JavaScript object.

Declaration

```
public void removeMember(String name)
```

setMember

Method. Sets the value of a property of a JavaScript object. Equivalent to "`this.name = value`" in JavaScript.

Declaration

```
public void setMember(String name,
   Object value)
```

setSlot

Method. Sets the value of an array element of a JavaScript object. Equivalent to "`this[index] = value`" in JavaScript.

Declaration

```
public void setSlot(int index,
   Object value)
```

toString

Method. Converts a `JSObject` to a `String`.

Overrides: `toString` in class `java.lang.Object`

Declaration `public String toString()`

Plugin

The public class `Plugin` extends `Object`.

```
java.lang.Object
   |
   +----netscape.plugin.Plugin
```

Description

This class represents the Java reflection of a plug-in. Plug-ins that need to have Java methods associated with them should subclass this class and add new (possibly native) methods to it. This allows other Java entities (such as applets and JavaScript code) to manipulate the plug-in.

Constructor and Method Summary

The `netscape.plugin.Plugin` class has the following constructors:

Constructor	Description
`Plugin`	Constructs a Plugin.

The `netscape.plugin.Plugin` class has the following methods:

Method	Description
`destroy`	Called when the plug-in is destroyed
`getPeer`	Returns the native NPP object—the plug-in instance that is the native part of a Java `Plugin` object
`getWindow`	Returns the JavaScript window on which the plug-in is embedded
`init`	Called when the plug-in is initialized
`isActive`	Determines whether the Java reflection of a plug-in still refers to an active plug-in

The following sections show the declaration and usage of these constructors and methods.

destroy

Method. Called when the plug-in is destroyed. You never need to call this method directly, it is called when the plug-in is destroyed. At the point this method is called, the plug-in will still be active.

Declaration `public void destroy()`

See also `init`

getPeer

Method. Returns the native NPP object—the plug-in instance that is the native part of a Java `Plugin` object. This field is set by the system, but can be read from plug-in native methods by calling:

```
NPP npp = (NPP)netscape_plugin_Plugin_getPeer(env, thisPlugin);
```

Declaration `public int getPeer()`

getWindow

Method. Returns the JavaScript window on which the plug-in is embedded.

Declaration `public JSObject getWindow()`

init

Method. Called when the plug-in is initialized. You never need to call this method directly, it is called when the plug-in is created.

Declaration `public void init()`

See also `destroy`

isActive

Method. Determines whether the Java reflection of a plug-in still refers to an active plug-in. Plug-in instances are destroyed whenever the page containing the plug-in is left, thereby causing the plug-in to no longer be active.

Declaration `public boolean isActive()`

Plugin

Constructor. Constructs a `Plugin`.

Declaration `public Plugin()`

4

Appendixes

- **Reserved Words**
- **Color Values**
- **Netscape Cookies**

Appendix

A

Reserved Words

This appendix lists the reserved words in JavaScript.

The reserved words in this list cannot be used as JavaScript variables, functions, methods, or object names. Some of these words are keywords used in JavaScript; others are reserved for future use.

abstract	else	instanceof	switch
boolean	enum	int	synchronized
break	export	interface	this
byte	extends	long	throw
case	false	native	throws
catch	final	new	transient
char	finally	null	true
class	float	package	try
const	for	private	typeof
continue	function	protected	var
debugger	goto	public	void
default	if	return	volatile
delete	implements	short	while
do	import	static	with
double	in	super	

Appendix

B

Color Values

The string literals in this appendix can be used to specify colors in the JavaScript alinkColor, bgColor, fgColor, linkColor, and vLinkColor properties and the fontcolor method.

You can also use these string literals to set the colors in HTML tags, for example

```
<BODY BGCOLOR="bisque">
```

or

```
<FONT COLOR="blue">color me blue</FONT>
```

Instead of using the string to specify a color, you can use the red, green, and blue hexadecimal values shown in the following table.

Color	Red	Green	Blue
aliceblue	F0	F8	FF
antiquewhite	FA	EB	D7
aqua	00	FF	FF
aquamarine	7F	FF	D4
azure	F0	FF	FF
beige	F5	F5	DC
bisque	FF	E4	C4

Color	Red	Green	Blue
black	00	00	00
blanchedalmond	FF	EB	CD
blue	00	00	FF
blueviolet	8A	2B	E2
brown	A5	2A	2A
burlywood	DE	B8	87
cadetblue	5F	9E	A0
chartreuse	7F	FF	00
chocolate	D2	69	1E
coral	FF	7F	50
cornflowerblue	64	95	ED
cornsilk	FF	F8	DC
crimson	DC	14	3C
cyan	00	FF	FF
darkblue	00	00	8B
darkcyan	00	8B	8B
darkgoldenrod	B8	86	0B
darkgray	A9	A9	A9
darkgreen	00	64	00
darkkhaki	BD	B7	6B
darkmagenta	8B	00	8B
darkolivegreen	55	6B	2F
darkorange	FF	8C	00
darkorchid	99	32	CC
darkred	8B	00	00
darksalmon	E9	96	7A
darkseagreen	8F	BC	8F
darkslateblue	48	3D	8B

Color	Red	Green	Blue
darkslategray	2F	4F	4F
darkturquoise	00	CE	D1
darkviolet	94	00	D3
deeppink	FF	14	93
deepskyblue	00	BF	FF
dimgray	69	69	69
dodgerblue	1E	90	FF
firebrick	B2	22	22
floralwhite	FF	FA	F0
forestgreen	22	8B	22
fuchsia	FF	00	FF
gainsboro	DC	DC	DC
ghostwhite	F8	F8	FF
gold	FF	D7	00
goldenrod	DA	A5	20
gray	80	80	80
green	00	80	00
greenyellow	AD	FF	2F
honeydew	F0	FF	F0
hotpink	FF	69	B4
indianred	CD	5C	5C
indigo	4B	00	82
ivory	FF	FF	F0
khaki	F0	E6	8C
lavender	E6	E6	FA
lavenderblush	FF	F0	F5
lawngreen	7C	FC	00
lemonchiffon	FF	FA	CD

Color	Red	Green	Blue
lightblue	AD	D8	E6
lightcoral	F0	80	80
lightcyan	E0	FF	FF
lightgoldenrodyellow	FA	FA	D2
lightgreen	90	EE	90
lightgrey	D3	D3	D3
lightpink	FF	B6	C1
lightsalmon	FF	A0	7A
lightseagreen	20	B2	AA
lightskyblue	87	CE	FA
lightslategray	77	88	99
lightsteelblue	B0	C4	DE
lightyellow	FF	FF	E0
lime	00	FF	00
limegreen	32	CD	32
linen	FA	F0	E6
magenta	FF	00	FF
maroon	80	00	00
mediumaquamarine	66	CD	AA
mediumblue	00	00	CD
mediumorchid	BA	55	D3
mediumpurple	93	70	DB
mediumseagreen	3C	B3	71
mediumslateblue	7B	68	EE
mediumspringgreen	00	FA	9A
mediumturquoise	48	D1	CC
mediumvioletred	C7	15	85
midnightblue	19	19	70

Color	Red	Green	Blue
mintcream	F5	FF	FA
mistyrose	FF	E4	E1
moccasin	FF	E4	B5
navajowhite	FF	DE	AD
navy	00	00	80
oldlace	FD	F5	E6
olive	80	80	00
olivedrab	6B	8E	23
orange	FF	A5	00
orangered	FF	45	00
orchid	DA	70	D6
palegoldenrod	EE	E8	AA
palegreen	98	FB	98
paleturquoise	AF	EE	EE
palevioletred	DB	70	93
papayawhip	FF	EF	D5
peachpuff	FF	DA	B9
peru	CD	85	3F
pink	FF	C0	CB
plum	DD	A0	DD
powderblue	B0	E0	E6
purple	80	00	80
red	FF	00	00
rosybrown	BC	8F	8F
royalblue	41	69	E1
saddlebrown	8B	45	13
salmon	FA	80	72
sandybrown	F4	A4	60

Color	Red	Green	Blue
seagreen	2E	8B	57
seashell	FF	F5	EE
sienna	A0	52	2D
silver	C0	C0	C0
skyblue	87	CE	EB
slateblue	6A	5A	CD
slategray	70	80	90
snow	FF	FA	FA
springgreen	00	FF	7F
steelblue	46	82	B4
tan	D2	B4	8C
teal	00	80	80
thistle	D8	BF	D8
tomato	FF	63	47
turquoise	40	E0	D0
violet	EE	82	EE
wheat	F5	DE	B3
white	FF	FF	FF
whitesmoke	F5	F5	F5
yellow	FF	FF	00
yellowgreen	9A	CD	32

Appendix

C

Netscape Cookies

A cookie is a small piece of information stored on the client machine in the cookies.txt file. This appendix discusses the implementation of cookies in the Navigator client; it is not a formal specification or standard.

You can manipulate cookies

- Explicitly, with a CGI program.
- Programmatically, with client-side JavaScript using the `cookie` property of the `document` object.
- Transparently, with the server-side JavaScript using the `client` object, when using client-cookie maintenance.

For information about using cookies in server-side JavaScript, see the *Server-Side JavaScript Guide*.

This appendix describes the format of cookie information in the HTTP header, and discusses using CGI programs and JavaScript to manipulate cookies.

Syntax A CGI program uses the following syntax to add cookie information to the HTTP header:

```
Set-Cookie:
   name=value
   [;EXPIRES=dateValue]
   [;DOMAIN=domainName]
   [;PATH=pathName]
   [;SECURE]
```

Parameters `name=value` is a sequence of characters excluding semicolon, comma and white space. To place restricted characters in the `name` or `value`, use an encoding method such as URL-style %XX encoding.

`EXPIRES=dateValue` specifies a date string that defines the valid life time of that cookie. Once the expiration date has been reached, the cookie will no longer be stored or given out. If you do not specify `dateValue`, the cookie expires when the user's session ends.

The date string is formatted as:

```
Wdy, DD-Mon-YY HH:MM:SS GMT
```

where `Wdy` is the day of the week (for example, Mon or Tues); `DD` is a two-digit representation of the day of the month; `Mon` is a three-letter abbreviation for the month (for example, Jan or Feb); `YY` is the last two digits of the year; `HH:MM:SS` are hours, minutes, and seconds, respectively.

`DOMAIN=domainName` specifies the domain attributes for a valid cookie. See "Determining a Valid Cookie" on page 677. If you do not specify a value for `domainName`, Navigator uses the host name of the server which generated the cookie response.

`PATH=pathName` specifies the path attributes for a valid cookie. See "Determining a Valid Cookie" on page 677. If you do not specify a value for `pathName`, Navigator uses the path of the document that created the cookie property (or the path of the document described by the HTTP header, for CGI programming).

`SECURE` specifies that the cookie is transmitted only if the communications channel with the host is a secure. Only HTTPS (HTTP over SSL) servers are currently secure. If `SECURE` is not specified, the cookie is considered sent over any channel.

Description A server sends cookie information to the client in the HTTP header when the server responds to a request. Included in that information is a description of the range of URLs for which it is valid. Any future HTTP requests made by the client which fall in that range will include a transmittal of the current value of the state object from the client back to the server.

Many different application types can take advantage of cookies. For example, a shopping application can store information about the currently selected items for use in the current session or a future session, and other applications can store individual user preferences on the client machine.

Determining a Valid Cookie. When searching the cookie list for valid cookies, a comparison of the domain attributes of the cookie is made with the domain name of the host from which the URL is retrieved.

If the domain attribute matches the end of the fully qualified domain name of the host, then path matching is performed to determine if the cookie should be sent. For example, a domain attribute of `royalairways.com` matches hostnames `anvil.royalairways.com` and `ship.crate.royalairways.com`.

Only hosts within the specified domain can set a cookie for a domain. In addition, domain names must use at least two or three periods. Any domain in the `COM`, `EDU`, `NET`, `ORG`, `GOV`, `MIL`, and `INT` categories requires only two periods; all other domains require at least three periods.

`PATH=pathName` specifies the URLs in a domain for which the cookie is valid. If a cookie has already passed domain matching, then the pathname component of the URL is compared with the path attribute, and if there is a match, the cookie is considered valid and is sent along with the URL request. For example, `PATH=/foo matches /foobar` and `/foo/bar.html`. The path `"/"` is the most general path.

Syntax of the Cookie HTTP Request Header. When requesting a URL from an HTTP server, the browser matches the URL against all existing cookies. When a cookie matches the URL request, a line containing the name/value pairs of all matching cookies is included in the HTTP request in the following format:

```
Cookie: NAME1=OPAQUE_STRING1; NAME2=OPAQUE_STRING2 ...
```

Saving Cookies. A single server response can issue multiple Set-Cookie headers. Saving a cookie with the same `PATH` and `NAME` values as an existing cookie overwrites the existing cookie. Saving a cookie with the same `PATH` value but a different `NAME` value adds an additional cookie.

The `EXPIRES` value indicates when to purge the mapping. Navigator will also delete a cookie before its expiration date arrives if the number of cookies exceeds its internal limits.

A cookie with a higher-level `PATH` value does not override a more specific `PATH` value. If there are multiple matches with separate paths, all the matching cookies are sent, as shown in the examples below.

A CGI script can delete a cookie by returning a cookie with the same `PATH` and `NAME` values, and an `EXPIRES` value which is in the past. Because the `PATH` and `NAME` must match exactly, it is difficult for scripts other than the originator of a cookie to delete a cookie.

Specifications for the Client. When sending cookies to a server, all cookies with a more specific path mapping are sent before cookies with less specific path mappings. For example, a cookie "name1=foo" with a path mapping of "/ " should be sent after a cookie "name1=foo2" with a path mapping of "/bar" if they are both to be sent.

The Navigator can receive and store the following:

- 300 total cookies
- 4 kilobytes per cookie, where the name and the `OPAQUE_STRING` combine to form the 4 kilobyte limit.
- 20 cookies per server or domain. Completely specified hosts and domains are considered separate entities, and each has a 20 cookie limitation.

When the 300 cookie limit or the 20 cookie per server limit is exceeded, Navigator deletes the least recently used cookie. When a cookie larger than 4 kilobytes is encountered the cookie should be trimmed to fit, but the name should remain intact as long as it is less than 4 kilobytes.

Examples The following examples illustrate the transaction sequence in typical CGI programs.

Example 1. Client requests a document, and receives in the response:

```
Set-Cookie: CUSTOMER=WILE_E_COYOTE; path=/; expires=Wednesday,
   09-Nov-99 23:12:40 GMT
```

When client requests a URL in path "/" on this server, it sends:

```
Cookie: CUSTOMER=WILE_E_COYOTE
```

Client requests a document, and receives in the response:

```
Set-Cookie: PART_NUMBER=ROCKET_LAUNCHER_0001; path=/
```

When client requests a URL in path "/" on this server, it sends:

```
Cookie: CUSTOMER=WILE_E_COYOTE; PART_NUMBER=ROCKET_LAUNCHER_0001
```

Client receives:

```
Set-Cookie: SHIPPING=FEDEX; path=/foo
```

When client requests a URL in path "/" on this server, it sends:

```
Cookie: CUSTOMER=WILE_E_COYOTE; PART_NUMBER=ROCKET_LAUNCHER_0001
```

When client requests a URL in path "/foo" on this server, it sends:

```
Cookie: CUSTOMER=WILE_E_COYOTE; PART_NUMBER=ROCKET_LAUNCHER_0001;
   SHIPPING=FEDEX
```

Example 2. This example assumes all mappings from Example 1 have been cleared.

Client receives:

```
Set-Cookie: PART_NUMBER=ROCKET_LAUNCHER_0001; path=/
```

When client requests a URL in path "/" on this server, it sends:

```
Cookie: PART_NUMBER=ROCKET_LAUNCHER_0001
```

Client receives:

```
Set-Cookie: PART_NUMBER=RIDING_ROCKET_0023; path=/ammo
```

When client requests a URL in path "/ammo" on this server, it sends:

```
Cookie: PART_NUMBER=RIDING_ROCKET_0023;
   PART_NUMBER=ROCKET_LAUNCHER_0001
```

There are two name/value pairs named "PART_NUMBER" due to the inheritance of the "/" mapping in addition to the "/ammo" mapping.

Index