# Gecko DOM Reference

# Preface

## About This Reference

This section describes the guide itself: who it's for, how the information is presented, and how you can use the examples in the reference in your own DOM development.

Note that this document is under development, and is not currently a comprehensive listing of the DOM methods and properties implemented for Gecko. Each individual section of the document (e.g., the **DOM Document Reference**) is complete for the object(s) it describes, however. As reference information for the various members of the huge APIs becomes available, it is integrated into this document here.

## Who Should Read This Guide

The reader of the *Gecko DOM Reference* is a web developer or savvy web user who knows something about how web pages are constructed. This reference avoids making presumptions about the reader's acquaintance with the DOM, with XML, with web servers or web standards, and even with JavaScript, the language in which the DOM is made accessible to the reader. But the document does presume familiarity with HTML, with markup, with the basic strucure of web pages, with web browsers, and with stylesheets.

In its introductory material, many examples, and high-level explanations, the document is a "beginners" web development guide. In general, however, the API reference should be valuable for inexperienced and experienced web developers alike.

## What is Gecko?

Netscape 6.1, Mozilla, and other Mozilla-based browsers have identical implementations of the DOM. This is so because they use the same technology.

Gecko, the software component in these browsers that handles the parsing of the HTML, the layout of the pages, the document object model, and even the rendering of the entire application interface, is a fast, standards-compliant rendering engine that implements the W3C DOM standards and the DOM-like (but not standardized) browser object model (i.e., `window` et al) in the context of web pages and the application interface, or *chrome*, of the browser.

Though the application interface and the content displayed by the browser are different in many practical ways, the DOM exposes them uniformly as a hierarchy of nodes. The tree structure of the DOM (which in its application to the user

## API Syntax

Each description in the API reference includes the syntax, the input and output parameters (where the return type of the return type is given), an example, any additional notes, and a link to the appropriate specification.

Typically, read-only properties have a single line of syntax because those properties can only be gotten and not set. For example, the read-only property **`availHeight`** of the `document` object includes the following syntax information:

### Syntax

```
iAvail = window.screen.availHeight
```

This means that you can only use the property on the right hand of the statement; whereas with read/write properties, you can assign to the property, as the following syntax example illustrates:

### Syntax

```
msg = window.status
window.status = msg
```

In general, the object whose member is being described is given in the syntax statement with a simple type, e.g, `element` for all elements, `document` for the top-level document object, table for the `TABLE` object, etc. (see **Important Data Types** for more information about data types).

## Using the Examples

Many of the examples in this reference are complete files that you can execute by cutting and pasting into a new file and then opening in your web browser. Others are snippets. You can run these latter by placing them within JavaScript callback functions. For example, the example for the **window.document** property can be tested or within a function like the following, which is called by the accompanying button:

```
<html>
<script>
function testWinDoc() {
  doc= window.document;
  alert(doc.title);
}
</script>
<button onclick=”testWinDoc();”>
  test document property</button>
</html>
```

Similar functions and pages can be devised for all the object members that are not already packaged up for use. See the **Testing the DOM API** section in the introduction for a “test harness” that you can use to test a number of APIs all at once.

# Introduction to the DOM

This section provides a brief conceptual introduction to the DOM: what it is, how it provides structure for HTML and XML documents, how you can access it, and how this API presents the reference information and examples.

## What is the DOM?

The Document Object Model is an API for HTML and XML documents. It does two things for web developers: it provides a structural representation of the document, and it defines the way that that structure is to be accessed from script, allowing you to get at the web page as a structured group of nodes, which we will discuss shortly. Essentially, it connects web pages to scripts or programming languages.

Note that the DOM is not a particular application, product, or proprietary ordering of web pages. Rather, it is an API, an interface that vendors must implement if they are to be conformant with the W3C DOM standard. Every browser vendor that supports the DOM, just to take one small example, must return all the `<P>` elements in an HTML web page as an array of nodes when the `getElementsByTagName` method is called against that web page in a script:

```
paragraphs = document.getElementsByTagName(“P”);
// paragraphs[0] is the first <p> element
// paragraphs[1] is the second <p> element, etc.
alert(paragraphs[0].nodeName);
```

All of the properties, methods, and events available to the web developer for manipulating and creating web pages are organized into objects (e.g., the `document` object that represents the document itself, the `table` object that implements that special `HTMLTableElement` DOM interface for accessing HTML tables, and so forth). This documentation provides an object-by-object reference to those APIs.

## DOM vs. JavaScript

The short example above, like all of the examples in this reference, is JavaScript. That is to say, it's *written* in JavaScript, but it *uses* the DOM to access the web page and its elements. The DOM is not a programming language, but without it, the JavaScript language wouldn't have any model or notion of the web pages, XML pages, and elements with which it is usually concerned. Every element in a document—the document as a whole, the head, tables within the document, table headers, text within the table cells—is part of the document object model for that document, so they can all be accessed and manipulated using the DOM and a scripting language like JavaScript.

The DOM was designed to be independent of any particular programming language, making the structural representation of the document available from a single, consistent API. Though we focus exclusively on JavaScript in this reference documentation, implementations of the DOM can be built for any language, as this Python example demonstrates:

```
# Python DOM example
import xml.dom.minidom as m
doc = m.parse("C:\\Projects\\Py\\chap1.xml");
doc.nodeName # DOM property of document object;
p_list = doc.getElementsByTagName("para");
```

## How Do I Access the DOM?

You don't have to do anything special to begin using the DOM. Different browsers have different implementations of the DOM, and these implementations exhibit varying degrees of conformance to the actual DOM standard (a subject we try to avoid in this documentation), but every web browser uses some document object model to make web pages accessible to script.

When you create a script—whether it's in-line in a `<SCRIPT>` element or included in the web page by means of a script loading instruction—you can immediately begin using the API for the `document` or `window` elements to manipulate the document itself or to get at the children of that document, which are the various elements in the web page. Your DOM programming may be something as simple as the following,

which displays an alert message by using the `alert()` function from the `window` object, or it may use more sophisticated DOM methods to actually create new content, as in the longer example below.

```
<body
   onload="window.alert('welcome to my home page!');">
```

Aside from the `<script>` element in which the JavaScript is defined, this JavaScript creates a new H1 element, adds text to that element, and then adds the `H1` to the tree for this document:

```
<html>
<script>
  // create a couple of elements
  // in an otherwise empty HTML page
  heading = document.createElement("H1");
  heading_text = document.createTextNode("Big Head!");
  heading.appendChild(heading_text);
  document.body.appendChild(heading);

</script>
</html>
```

## Important Data Types

This reference tries to describe the various objects and types in as simple a way as possible. But there are a number of different data types being passed around the API that you should be aware of. For the sake of simplicity, syntax examples in this API reference typically refer to `nodes` as elements, to arrays of nodes as `nodeLists` (or simply `elements`), and to `attribute` nodes simply as attributes.

The following table briefly describes these data types.

`document` — When a member returns an object of type `document` (e.g., the **ownerDocument** property of an element returns the `document` to which it belongs), this object is the root `document` object itself. The **DOM Document Reference** chapter describes the `document` object.

`element` — `element` refers to an element or a node of type `element` returned by a member of the DOM API. Rather than saying, for example, that the `document.createElement()` method returns an object reference to a `node`, we just say that this method returns the `element` that has just been created in the DOM.

`element` objects implement the DOM `Element` interface and also the more basic `Node` interface, both of which are included together in this reference.

`nodeList` — A `nodeList` is an array of elements, like the kind that is returned by the method `document.getElementsByTagName()`. Items in a `nodeList` are accessed by index in either of two ways:

- list.item(1)
- list[1]

These two are equivalent. In the first, **item()** is the single method on the `nodeList` object. The latter uses the typical array syntax to fetch the second item in the list.

`attribute` — When an `attribute` is returned by a member (e.g., by the **createAttribute()** method), it is an object reference that exposes a special (albeit small) interface for attributes. Attributes are nodes in the DOM just like elements are, though you may rarely use them as such.

`NamedNodeMap` — A `namedNodeMap` is like an array, but the items are accessed by name or index, though this latter case is merely a convenience for enumeration, as they are in no particular order in the list. A `NamedNodeMap` has an item() method for this purpose, and you can also add and remove items from a `NamedNodeMap`

# DOM Interfaces

A stated purpose of this guide is to minimize talk about abstract interfaces, inheritance, and other nerdy implementation details, and to talk instead about the objects in the DOM, about the actual *things* you can use to manipulate the DOM hierarchy. From the point of view of the web programmer, it's often a matter of indifference that the object representing the `HTML FORM` element gets its **name** property from the `HTMLFormElement` interface but its **className** property from the `HTMLElement` interface proper. In both cases, the property you want is simply *in* the `form` object.

But the relationship between objects and the interfaces that they implement in the DOM can be confusing, and so this section attempts to say a little something about the actual interfaces in the DOM specification and how they are made available.

## Interfaces Versus Objects

In some cases, an object exposes a single interface. But more often than not, an object like `table` represents several different interfaces. The `table` object, for example, implements a specialized `HTMLTableElement` interface, which includes such methods as XXX and YYY. But since it's also an HTML element, `table` implements the `Element` interface described in the **DOM Element Reference** chapter. And finally, since an HTML element is also, as far as the DOM is concerned, a node in the tree of nodes that make up the object model for a web page or an XML page, the table element also implements the more basic `Node` interface, from which `Element` derives.

When you get a reference to a `table` object, as in the following example, you routinely use all three of these interfaces interchangeably on the object, perhaps without knowing it.

```
table = document.getElementById("table");
tats = table.attributes; // Node/Element interface
for (var i = 0; i < tats.length; i++) ) {
  if tats[i] == "border"
    table.setAttribute("border", "2px solid blue");
    // HTMLTableElement interface: summary attribute
    table.summary = "note: increased border";
}
```

## Core Interfaces in the DOM

This section lists some of the mostly commonly-used interfaces in the DOM. The idea is not to describe what these APIs do here but to give you an idea of the sorts of methods and properties you will see very often as you use the DOM. These common APIs are used in the longer examples in the **DOM Examples** chapter at the end of this book.

`document` and `window` objects are the objects whose interfaces you generally use most often in DOM programming. In simple terms, the `window` object represents something like the browser, and the `document` object is the root of the document itself. `Element` inherits from the generic `Node` interface, and together these two interfaces provide many of the methods and properties you use on individual elements. These elements may also have specific interfaces for dealing with the kind of data those elements hold, as in the `table` object example in the previous section.

The following is a brief list of common APIs in web and XML page scripting using the DOM.

- `document.`**`getElementById`**`(id)`
- `document.`**`getElementsByTagName`**`(name)`
- `document.`**`createElement`**`(name)`
- `parentNode.`**`appendChild`**`(node)`
- `element.`**`innerHTML`**
- `element.`**`style`**`.left`
- `element.`**`setAttribute`**
- `element.`**`getAttribute`**
- `element.`**`addEventListener`**
- **`window._content`**
- **`window.onload`**
- **`window.dump()`**
- **`window.scrollTo()`**

# Testing the DOM API

This document provides samples for every interface that you can use in your own web development. In some cases, the samples are complete HTML pages, with the DOM access in a <script> element, the interface (e.g, buttons) necessary to fire up the script in a form, and the HTML elements upon which the DOM operates listed as well. When this is the case, you can cut and paste the example into a new HTML document, save it, and run the example from the browser.

There are some cases, however, when the examples are more concise. To run examples that only demonstrate the basic relatioship of the interface to the HTML elements, you may want to set up a test page in which interfaces can be easily accessed from scripts. The following very simple web page provides a <script> element in the header in which you can place functions that test the interface, a few HTML elements with attributes that you can retrieve, set, or otherwise manipulate, and the web user interface necessary to call those functions from the browser.

You can use this test page or create a similar one to test the DOM interfaces you are interested in and see how they work on the browser platform. You can update the contents of the `test()` function as needed, create more buttons, or add elements as necessary.

```
<html>
<head>
<title>DOM Tests</title>
<script type="application/x-javascript">
function setBodyAttr(attr,value){
  if(document.body) eval('document.body.'+attr+'="'+value+'"');
  else notSupported();
}

</script>
</head>
<body>
<div style="margin: .5in; height="400"">
<p><b><tt>text</tt> color</p>
<form>
<select onChange="setBodyAttr('text',
    this.options[this.selectedIndex].value);">
<option value="black">black
<option value="darkblue">darkblue
</select>

<p><b><tt>bgColor</tt></p>
<select onChange="setBodyAttr('bgColor',
    this.options[this.selectedIndex].value);">
<option value="white">white
<option value="lightgrey">gray
</select>

<p><b><tt>link</tt></p>
<select onChange="setBodyAttr('link',
     this.options[this.selectedIndex].value);">
<option value="blue">blue
<option value="green">green
</select>  <small>
     <a href="http://www.brownhen.com/dom_api_top.html"
```

```
id="sample">
(sample link)</a></small><br>

</form>
<form>
  <input type="button" value="version" onclick="ver()" />
</form>
</div>
</body>
</html>
```

To test a lot of interfaces in a single page—for example, a “suite” of properties that affect the colors of a web page—you can create a similar test page with a whole console of buttons, textfields, and other HTML elements. The following screenshot gives you some idea of how interfaces can be grouped together for testing.

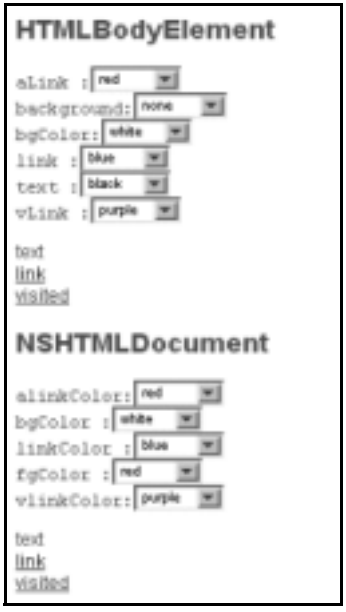



Figure 0.1 Sample DOM Test Page

In this example, the dropdown menus dynamically update such DOM-accessible aspects of the web page as its background color (`bgColor`), the color of the hyperlinks (`aLink`), and color of the text (`text`). However you design your test pages, testing the interfaces as you read about them is an important part of learning how to use the DOM effectively.

# DOM *Element* Reference

This chapter provides a brief reference for all of the methods, properties, and events available to all HTML and XML elements in the Netscape 6 DOM.

These DOM interfaces cross the various specification levels, but tend to concentrate on the published DOM Level 2 HTML recommendation. Each member includes a link to the appropriate place in the W3C DOM specifications.

## DOM Elements Interface

In this case, *Elements* refers to the interface that all HTML and XML elements have available to them from the DOM. There are more specialized interfaces for particular objects—the `BODY` element, for example, has extra functions and properties you can use, as do tables. This chapter refers to the interface that all elements share.

### Properties

| | |
|---|---|
| **attributes** | Returns an array of the attributes on the element. |
| **childNodes** | Returns an array of the child nodes on the element. |
| **className** | Gets/sets the class of the element. |
| **dir** | Gets/sets the directionality of the element. |
| **firstChild** | Returns the first direct child of the current node. |
| **id** | Gets/sets the id of the current element. |

| | |
|---|---|
| **innerHTML** | **innerHTML** returns all of the markup and content within a given element. |
| **lang** | Specifies the base language of an element's attribute values and text content. |
| **lastChild** | Returns the last child of the current node. |
| **length** | Returns the number of items in a list (e.g. childNodes). |
| **localName** | **localName** returns the local part of the qualified name of this node. |
| **namespaceURI** | The namespace URI of this node, or NULL if it is unspecified. |
| **nextSibling** | Returns the node immediately following the current one in the tree. |
| **nodeName** | Returns the name of the current node. |
| **nodeType** | Returns the type of the current node. |
| **nodeValue** | Returns the value of the current node. |
| **offsetHeight** | **offsetHeight** gets the number of pixels that the current element is offset within the offsetParent node |
| **offsetLeft** | **offsetLeft** gets/sets the number of pixels that the current element is offset to the left within the offsetParent node. |
| **offsetParent** | **offsetParent** returns a reference to the object in which the current element is offset (i.e., the parent element). |
| **offsetTop** | **offsetTop** returns the position of the current element relative to the top of the offsetParent node. |

| | |
|---|---|
| **offsetWidth** | **offsetWidth** gets the number of pixels that the current element is offset within the offsetParent node. |
| **ownerDocument** | Returns the `document` in which this node appears. |
| **parentNode** | Returns the parent node of the current node. |
| **prefix** | **prefix** returns the namespace prefix of the current node, or `NULL` if it is unspecified. |
| **previousSibling** | Returns the `node` immediately previous to the current one in the tree. |
| **style** | **style** returns the block of style rules on the current element. |
| **tabIndex** | Gets/sets the position of the element in the tabbing order. |
| **tagName** | **tagName** returns the name of the element. |
| **title** | **title** returns the title of the document. |

### Methods

| | |
|---|---|
| **addEventListener** | **addEventListener** allows the registration of event listeners on the event target. |
| **appendChild** | The **appendChild** method inserts the specified node into the list of nodes on the current document. |
| **blur** | The **blur** method removes keyboard focus from the current element. |
| **click** | The **click** method excecutes a click on the current element. |
| **cloneNode** | The **cloneNode** method returns a duplicate of the current node. |
| **dispatchEvent** | The **dispatchEvent** method allows the dispatch of events into the implementation's event model. |
| **focus** | **focus** sets focus on the current element. |
| **getAttribute** | **getAttribute** returns the value of the named attribute on the current node. |
| **getAttributeNS** | getAttributeNS returns the value of the attribute with the given name and namespace. |
| **getAttributeNode** | Returns the attribute of the current element as a separate node. |
| **getElementsByTagName** | Returns the elements of a particular name that are children of the current element. |

| | |
|---|---|
| **hasAttribute** | **hasAttribute** returns a boolean value indicating whether the current element has the specified attribute or not. |
| **hasAttributeNS** | **hasAttribute** is a boolean value indicating whether the current element has an attribute with the specified name and namespace. |
| **hasChildNodes** | **hasChildNodes** is a boolean value indicating whether the current element has children or not. |
| **insertBefore** | The **insertBefore** method allows you to insert a node before the current element in the DOM. |
| **item** | The **item** method retrieves a node from the tree by index. |
| **normalize** | The **normalize** method puts the current node and all of its subtree into a “normalized” form (see below). |
| **removeAttribute** | The **removeAttribute()** method removes an attribute from the current element. |
| **removeAttributeNode** | **removeAttributeNode** removes the specified attribute from the current element. |
| **removeChild** | The **removeChild()** method removes a child node from the current element. |
| **removeEventListener** | **removeEventListener()** allows the removal of event listeners from the event target. |

| | |
|---|---|
| **replaceChild** | The **replaceChild()** method replaces one child node on the current element with another. |
| **setAttribute** | **setAttribute** adds a new attribute or changes the value of an existing attribute on the current element. |
| **setAttributeNS** | **setAttributeNS** adds a new attribute or changes the value of an attribute with the given namespace and name. |
| **setAttributeNode** | **setAttributeNode** adds a new attribute node to the current element. |
| **supports** | The **supports** method tests if this DOM implementation supports a particular feature. |

### Event Handlers

These `element` properties cannot be assigned to in the way that the event handlers on the `document` and `window` objects can. All of the folllowing event handler properties are read-only, and are made to return the event handling code, if any, that has already been added to the element in the HTML or XML itself.

| | |
|---|---|
| **onblur** | Returns the event handling code for the blur event. |
| **onclick** | Returns the event handling code for the click event. |
| **ondblclick** | Returns the event handling code for the dblclick event. |
| **onfocus** | Returns the event handling code for the focus event. |

| | |
|---|---|
| **onkeydown** | Returns the event handling code for the keydown event. |
| **onkeypress** | Returns the event handling code for the keypress event. |
| **onkeyup** | Returns the event handling code for the keyup event. |
| **onmousedown** | Returns the event handling code for the mousedown event. |
| **onmousemove** | Returns the event handling code for the mousemove event. |
| **onmouseout** | Returns the event handling code for the mouseout event. |
| **onmouseover** | Returns the event handling code for the mouseover event. |
| **onmouseup** | Returns the event handling code for the mouseup event. |
| **onresize** | Returns the event handling code for the resize event. |

# attributes

Returns an array of attributes on the given element

## Syntax

```
attributes = elementNode.attributes
```

## Parameters

The `attributes` parameter returned by this property is a `NamedNodeMap` of attribute nodes.

## Example

```
// get the first <p> element in the document
para = document.getElementsByTag(“p”)[0];
atts = para.attributes;
```

## Notes

The array returned by this property is a `NamedNodeMap`, a list of objects rather than strings. The name and value of the attribute objects are accessible as separate properties, as in the following complete example, which retrieves the name/value pair of the first attribute of the “p1” paragraph in the document:

```
<html>
<head>
<script>
function showA() {
   p = document.getElementById("p1");
   t = document.getElementById("t");
   t.setAttribute("value",
      p.attributes[0].name + "->" +
p.attributes[0].value);
}
</script>
</head>

<p id="p1" style="color: blue;">Sample Paragraph</p>
<form>
<input type="button" value="show" onclick="showA()" />
<input id="t" type="text" value="" />
</form>
</html>
```

## Specification

`attributes`

# childNodes

Returns an array of child nodes on the given element node.

## Syntax

```
nodeList = elementNode.childNodes
```

## Parameters

`nodeList` is a list of elements that are children of the current element.

## Example

```
// table is an object reference to a table element
kids = table.childNodes;
for (var i = 0; i < kids.length; i++) {
  // do something with each kid as kids[i]
}
```

## Notes

The `document` object itself has only a single child, and that is the `HTML` element. Note again that the items in the array are objects and not strings. To get data from those objects you must use their properties (e.g. `childNode[2].nodeName` to get the name, etc.)

## Specification

`childNodes`

# className

This property gets/sets the class of the current element

## Syntax

```
name = element.className
element.className = name
```

## Parameters

`name` is a string representing the class of the current element.

## Example

```
el = document.getElementById(“div1”);
if (el.className == “fixed”) {
  // skip a particular class of element
  goNextElement();
}
```

## Notes

The name **className** is used for this property instead of “class” because of conflicts with the “class” keyword in many languages which use the DOM.

## Specification

`className`

# dir

The **dir** property specifies the directionality of the text of the current element.

### Syntax

*directionality* = element.dir
element.dir = *directionality*

### Parameters

directionality is a string representing the direction of the current element text. See Notes below for a list of the valid directions.

### Example

```
p = document.getElementById(“para1”);
p.dir = “rtl”; // change text direction on this <p>
```

### Notes

The directionality of element text is which direction that text goes (for support of different language systems). Possible values for **dir** are `ltr`, for Left-to-right, and `rtl`, for Right-to-left.

### Specification

dir

# firstChild

**firstChild** returns the first child element of the current element

### Syntax

*element* = element.firstChild

### Parameters

The element parameter returned is a node of type element.

### Example

```
trow = document.getElementById(“row1”);
left_cell = trow.firstChild;
```

### Notes

Returns `NULL` if the current node is childless.

### Specification

`firstChild`

# id

The **id** property uniquely identifies the current element.

### Syntax

```
id_str = element.id
element.id = id_str
```

### Parameters

`id_str` is a string representing the id of the current element.

### Example

```
if element.id != “main_loop”
    goBack();
```

### Notes

There is no more central property to the domain of web development than **id**. The ID of an element is what is most often used to retrieve it (i.e., with **getElementById**), and it allows the various nodes in a document to be manipulated independently of one another. In HTML and in XUL, the **id** is defined as an attribute on the element like so:

```
<td id="table-cell2" />
```

If you plan to use the DOM with your web pages, it's a good idea to give as many of your elements **id** attributes as is necessary. Note that the id element is also frequently used to associated style rules with individual markup elements.

### Specification

`id`

# innerHTML

**innerHTML** returns all of the markup and content within a given element.

### Syntax

```
HTML = element.innerHTML
```

### Parameters

`HTML` is a string that contains the current element and its content (including child elements) as raw HTML

## Example

```
// HTML:
// <div id=”d”><p>Content</p>
// <p>Further Elaborated</p>
// </div>
d = document.getElementById(“d”);
dump(d.innerHTML);

// the string “<p>Content</p><p>Further Elaborated</p>”
// is dumped to the console window
```

## Notes

Though not actually a part of the DOM spec, this property gives the web developer enormous flexibility over the contents of a web page. Consider the following example, where a script sets the blah blah..

// nuther example

You can get the HTML and text contained within any element—including BODY or HTML—and parse it yourself or blah blah. You can also set this property, which means that you can control the contents of the document by adding to or subtracting from the innerHTML. This third example gives you an idea about how evil this can be when it falls into the wrong hands.

// third example

Note that when you append to the innerHTML of a document, you have essentially created a new document. The session history for the browser is incremented, and when you go Back, the document is there in its original, unappended state.

## Specification

DOM Level 0. *Not part of specification*.

# lang

This property specifies the base language of an element's attribute values and text content.

## Syntax

```
language = element.lang
element.lang = language
```

## Parameters

`language` is a string that represents the language in which the text of the current element is written.

## Example

```
// this snippet checks that the base language blah blah
if ( document.lang != “en” ) {
   document.location = “other_lang_top.html”;
}
```

## Notes

The language code returned by this property is defined in RFC 1766. Common examples include “en” for English, “ja” for Japanese, “sp” for Spanish, and so on. The default value of this attribute is `unknown`. Note that this property, though valid at the individual element level described here, is most often used for the `BODY` or for the document itself.

## Specification

`lang`

# lastChild

**lastChild** returns the last child of the current element.

## Syntax

```
last_child = element.lastChild
```

## Parameters

`last_child` is the final element node in the nodeList of children on the current element.

## Example

```
tr = document.getElementById(“row1”);
corner_td = tr.lastChild;
```

## Notes

Returns `NULL` if there are no child elements.

## Specification

`lastChild`

# length

**length** returns the number of items in a list.

## Syntax

```
no_of_items = nodeList.length
```

## Parameters

`no_of_items` is an integer value representing the number of items in a list.

### Example

```
// all the paragraphs in the document
items = document.getElementsByTagName(“p”);
// are there any at all?
if ( items.length ) {
    // for each item in the list,
    // append the entire element as a string of HTML
    for (var i = 0; i < items.length; i++) {
        gross += items[0].innerHTML;
        // gross is now all the HTML for the paragraphs
    }
}
```

### Notes

**length** is a very common property in DOM programming. It’s very common to test the length of a list (to see if it exists at all) and to use it as the iterator in a for loop, as in the example above.

### Specification

`length`

# localName

**localName** returns the local part of the qualified name of this node.

### Syntax

*name* `= element.localName`

### Parameters

`name` is the local name as a string.

## Example

```
// qualifiedName = “XXXYYY”
d = document.getElementById(“div1”);
text_field = document.getElementById(“t”);
text_field.setAttribute(“value”, d.localName);
// text_field reads “YYY”
```

## Notes

For nodes of any type other than `ELEMENT_NODE` and `ATTRIBUTE_NODE` and nodes created with a DOM Level 1 method, such as `createElement` on the `document` object, this is always `NULL`.

The **localname** of a node is that part of the node’s qualified name that comes after the colon. Qualified names are typically used in XML as part of the namespace(s) of the particular XML documents. For example, in the qualified name “ecomm:partners,” “partners” is the localname and ecomm is the prefix:

```
<ecomm:business id=”soda_shop” type=”brick_n_mortar”>
  <ecomm:partners>
    <ecomm:partner id=”1001”>Tony’s Syrup Warehouse
    </ecomm:partner>
  </ecomm:partner>
</ecomm:business>
```

The prefix—in this case, “ecomm”—defines the namespace in which the localname can be used.

## See Also

**namespaceURI**

## Specification

`localName`

# namespaceURI

The namespace URI of this node, or `NULL` if it is unspecified.

## Syntax

```
namespace = element.namespaceURI
```

## Parameters

`namespace` is a string that represents the namespace URI of the current node.

## Example

In this snippet, a node is being examined for its **localName** and its **namespaceURI**. If the namespaceURI matches a variable in which the namespace for the XUL namespace is defined, then the node is understood to be a `<browser />` widget from XUL.

```
if (node.localName == "browser"
    && node.namespaceURI == kXULNSURI) {
// xul browser
this.viewee = node.webNavigation.document;
...
}
```

## Notes

This is not a computed value that is the result of a namespace lookup based on an examination of the namespace declarations in scope. It is merely the namespace URI given at creation time.

For nodes of any type other than `ELEMENT_NODE` and `ATTRIBUTE_NODE` and nodes created with a DOM Level 1 method, such as `createElement` from the Document interface, this is always `NULL`.

Per the Namespaces in XML Specification, an attribute does not inherit its namespace from the element it is attached to. If an attribute is not explicitly given a namespace, it simply has no namespace.

## Specification

`nameSpaceURI`

# nextSibling

Returns the `node` immediately following the current one in the tree.

### Syntax

```
next_element = element.nextSibling
```

### Parameters

`next_element` is the element node directly after the current element in the list of siblings (i.e., the list of children for the parentNode).

### Example

```
// in a table, the cells are siblings
cell1 = document.getElementById(“td1”);
cell2 = cell1.nextSibling;
```

### Notes

Returns `NULL` if there are no more nodes.

### Specification

`nextSibling`

# nodeName

Returns the name of the current node as a string.

### Syntax

*name* = element.nodeName

### Parameters

name is a string representing the name of the currrent element.

### Example

```
div1 = document.getElementById(“d1”);
text_field = document.getElementById(“t”);
text_field.setAttribute(“value”, div1.nodeName);
// textfield reads “div” now
```

### Notes

None.

### Specification

nodeName

# nodeType

Returns a code representing the type of the underlying node

### Syntax

*code* = document.nodeType

### Parameters

`code` is an unsigned short with one of the following values:

```
ELEMENT_NODE                    = 1;
ATTRIBUTE_NODE                  = 2;
TEXT_NODE                       = 3;
CDATA_SECTION_NODE              = 4;
ENTITY_REFERENCE_NODE           = 5;
ENTITY_NODE                     = 6;
PROCESSING_INSTRUCTION_NODE     = 7;
COMMENT_NODE                    = 8;
DOCUMENT_NODE                   = 9;
DOCUMENT_TYPE_NODE              = 10;
DOCUMENT_FRAGMENT_NODE          = 11;
NOTATION_NODE                   = 12;
```

### Example

```
if document.nodeType != 9
   document.close()
else
   document.write(“<p>I’m a doc!</p>”);
```

### Notes

None.

### Specification

`nodeType`

# nodeValue

Returns the value of the current node.

## Syntax

```
value = document.nodeValue
```

## Parameters

`value` is a string containing the value of the current node, if any.

## Example

`None.`

## Notes

For the document itself, **nodeValue** returns `NULL`. For text, comment, and `CDATA` nodes, nodeValue returns the content of the node. For attribute nodes, the value of the attribute is returned.

The following table shows the return values for different elements:

| | |
|---|---|
| Attr | value of attribute |
| CDATASection | content of the CDATA Section |
| Comment | content of the comment |
| Document | null |
| DocumentFragment | null |
| DocumentType | null |
| Element | null |
| NamedNodeMap | null |
| EntityReference | null |
| Notation | null |
| ProcessingInstruction | entire content excluding the target |
| Text | content of the text node |

When nodeValue is defined to be `NULL`, setting it has no effect.

## Specification

`nodeValue`

# offsetHeight

**offsetHeight** gets the number of pixels that the current element is offset within the `offsetParent` node.

## Syntax

```
height = element.offsetHeight
```

## Parameters

`height` is an integer representing the offset in pixels.

## Example

```
color_table = document.getElementById(“t1”);
tOffset = color_table.offsetHeight;
if ( tOffset > 5 ) {
    // large offset: do something here
}
```

## Notes

None.

## Specification

DOM Level 0. *Not part of specification.*

# offsetLeft

Gets/sets the number of pixels that the current element is offset to the left within the `offsetParent` node.

### Syntax

```
left = element.offsetLeft
```

### Parameters

`left` is an integer representing the offset to the left in pixels.

### Example

```
color_table = document.getElementById(“t1”);
tOLeft = color_table.offsetLeft;
if ( tOLeft > 5 ) {
    // large left offset: do something here
}
```

### Notes

None.

### Specification

DOM Level 0. *Not part of specification.*

# offsetParent

**offsetParent** returns a reference to the object which is the closest (nearest in the containment hierarchy) positioned containing element. If the element is non-positioned, the root element (html in standards compliant mode; body in quirks rendering mode) is the **offsetParent**.

### Syntax

```
parentObj = element.offsetParent
```

### Parameters

`parentObj` is an object reference to the element in which the current element is offset.

### Example

```
example here
```

### Notes

extra information

### Specification

DOM Level 0. *Not part of specification.*

# offsetTop

**offsetTop** returns the position of the current element relative to the top of the `offsetParent` node.

### Syntax

```
topPos = element.offsetTop
```

### Parameters

`topPos` is the number of pixels from the top of the parent element.

### Example

```
d = document.getElementById(“div1”);
topPos = d.offsetTop;
if (topPos > 10 ) {
    // object it offset less
    // than 10 pixels in its parent
}
```

### Notes

None.

### Specification

DOM Level 0. *Not part of specification.*

# offsetWidth

**offsetWidth** gets the number of pixels that the current element is offset within the `offsetParent` node.

### Syntax

*`width`* `= element.offsetWidth`

### Parameters

`width` is an integer representing the offset in pixels.

### Example

```
color_table = document.getElementById(“t1”);
tOffset = color_table.offsetWidth;
if ( tOffset > 5 ) {
    // large offset: do something here
}
```

### Notes

None.

### Specification

DOM Level 0. *Not part of specification.*

# ownerDocument

The **ownerDocument** property returns the top-level `document` object for this node.

### Syntax

*`document`* `= element.ownerDocument`

### Parameters

`document` is the `document` object parent of the current element.

### Example

```
// given a node “p”, get the top-level HTML child
// of the document object
d = p.ownerDocument;
htm = p.documentElement;
```

### Notes

The `document` object returned by this property is the main object with which all the child nodes in the actual HTML document are created.

If this property is used on a node that is itself a document, the result is `NULL`.

### Specification

`ownerDocument`

# parentNode

The **parentNode** property returns the parent of the current element.

### Syntax

```
pElement = element.parentNode
```

### Parameters

`pElement` is the `element` parent of the current node.

### Example

```
text_field = document.getElementById(“t”);
if ( div1.parentNode == document ){
    text_field.setAttribute(“value”, “top-level”);
    // textfield displays text “top-level”
}
```

### Notes

extra information

### Specification

`parentNode`

# prefix

**prefix** returns the namespace prefix of the current node, or `NULL` if it is unspecified.

## Syntax

```
pre = element.prefix
element.prefix = pre
```

## Parameters

`pre` is the namespace prefix as a string.

## Example

```
example here
```

## Notes

extra information

## Specification

`prefix`

# previousSibling

Returns the `node` immediately previous to the current one in the tree.

## Syntax

```
pNode = elementNode.previousSibling
```

### Parameters

`pNode` is the node prior to this one in the ordered list.

### Example

```
n1 = n2.previousSibling;
```

### Notes

Returns `NULL` if there are no more nodes.

### Specification

`previousSibling`

# style

**style** returns the block of style rules on the current element.

### Syntax

```
styleBlock = element.style
( element.style.styleAttr = value )
```

### Parameters

`styleBlock` is a string containing the

### Example

```
div = document.getElementById(“div1);
div.style.marginTop = “.25in”;
```

### Notes

**style** is a very commonly used property in DOM programming. You can use it to get the style rules associated with a particular element, and though you cannot set style on an element by assigning to the style property directly, you can use the style property to get to the writable style attributes on the node, as in the short example above.

See the **DOM CSS Properties List** for a list of the CSS properties that are accessible from the Gecko DOM. There are some additional notes there about the use of the style property to style elements in the DOM.

### Specification

`style`

# tabIndex

Gets/sets the tab order of the current element.

### Syntax

```
element.tabIndex = iIndex
```

### Parameters

`iIndex` is a number

### Example

```
b1 = document.getElementById(“button1”);
b1.tabIndex = 1;
```

### Notes

None.

### Specification

`tabIndex`

# tagName

**tagName** returns the name of the element.

## Syntax

```
elementName = element.tagName
```

## Parameters

`elementName` is a string containing the name of the current element.

## Example

```
<paragraph id=”p001”>When I was born...</paragraph>

p = document.getElementById(“p001”);
// p.tagName returns “paragraph”
```

## Notes

In XML, **tagName** preserves case. In HTML, **tagName** returns the element name in the canonical uppercase form. The value of **tagName** is the same as that of **nodeName**.

## Specification

`tagName`

# title

**title** returns the title of the document.

### Syntax

```
sTitle = document.title
```

### Parameters

`sTitle` is a string that contains the title of the current document.

### Example

```
<html>
  <title>Hello World!</title>
  <body>...

// document.title returns “Hello World!”
```

### Notes

None.

### Specification

`title`

# addEventListener

**addEventListener** allows the registration of event listeners on the event target.

### Syntax

```
element.addEventListener( type, listener, useCapture )
```

### Parameters

The `addEventListener()` method takes the following parameters:

| | |
|---|---|
| `type` | A string representing the event type being registered. |
| `listener` | The listener parameter takes an interface implemented by the user which contains the methods to be called when the event occurs. |
| `useCapture` | If true, useCapture indicates that the user wishes to initiate capture. After initiating capture, all events of the specified type will be dispatched to the registered EventListener before being dispatched to any EventTargets beneath them in the tree. Events which are bubbling upward through the tree will not trigger an EventListener designated to use capture. |

## Example

```
<html>
<head>
  <title>DOM Event Examples</title>
  <style>
    #t { border: 1px solid red }
    #t1 { background-color: pink; }
  </style>
  <script>
  // Event Registration Example
  function l_func() {
    t2 = document.getElementById("t2");
    t2.innerHTML = "three";
  }

  function load() {
    el = document.getElementById("t");
    el.addEventListener("click", l_func, false);
  }
  </script>
</head>
<body onload="load();">
<table id="t">
  <tr><td id="t1">one</td></tr>
  <tr><td id="t2">two</td></tr>
</table>
</body>
</html>
```

## Notes

If an `EventListener` is added to an `EventTarget` while it is processing an event, it will not be triggered by the current actions but may be triggered during a later stage of event flow, such as the bubbling phase.

If multiple identical `EventListeners` are registered on the same `EventTarget` with the same parameters the duplicate instances are discarded. They do not cause the `EventListener` to be called twice and since they are discarded they do not need to be removed with the **removeEventListener** method.

### Specification

`addEventListener`

# appendChild

The **appendChild** method inserts the specified node into the list of nodes on the current document.

### Syntax

`element.appendChild(`*`newChild`*`)`

### Parameters

`newChild` is a node.

### Example

```
// puts the new, empty paragraph at the end
// of the document
p = document.createElement(“p”);
element.appendChild(p);
```

### Notes

**appendChild** is one of the fundamental methods of web programming using the DOM. The **appendChild** method inserts a new node into the DOM structure of the HTML document, and is the second part of the one-two, create-and-append process so central to building web pages programmatically. The example above illustrates this basic process.

### Specification

`appendChild`

# blur

The **blur** method removes keyboard focus from the current element.

### Syntax

```
element.blur()
```

### Parameters

None.

### Example

```
None.
```

### Notes

None.

### Specification

`blur`

# click

The **click** method excecutes a click on the current element.

### Syntax

```
element.click()
```

### Parameters

None.

### Example

```
example here
```

### Notes

The **click** method simulates a click event on the current element. This is frequently used to execute event handlers that have been placed on the current element or on elements above it in the “event chain.”

### Specification

`click`

# cloneNode

The **cloneNode** method returns a duplicate of the current node.

### Syntax

`dupNode = element.cloneNode(`*deep*`)`

### Parameters

`deep` is a boolean value indicating whether the clone is a deep clone or not (see notes below).

### Example

```
p = document.getElementById(“para1”);
p_prime = p.cloneNode(true);
```

### Notes

The duplicate node returned by **cloneNode()** has no parent. Cloning a node copies all of its attributes and their values but does not copy any of the text that the node contains, since that text is contained in a child `Text` node.

A deep clone is a clone in which the given node and the whole subtree beneath it (including the text that makes up any child Text nodes) is copied and returned.

### Specification

`cloneNode`

# dispatchEvent

The **dispatchEvent** method allows the dispatch of events into the implementation's event model.

### Syntax

```
boolean = element.dispatchEvent(event)
```

### Parameters

`event` is an event object that contains information about the type, behavior, and contextual information of the event to be dispatched.

### Example

```
b = document.getElementById(“button1”);
res = b.dispatchEvent(“click”);
if ( res ) {
   // event dispatch was successful
   b.disabled = true;
}
```

### Notes

When you create and dispatch an event using this method, the event has the same effect as events dispatched by user interaction. They are “real” events, in other words, and they bubble up the UI in the same way. See the **event** object interface for more information about the information that is passed in with this method.

### Specification

`dispatchEvent`

# focus

**focus** sets focus on the current element.

### Syntax

```
element.focus()
```

### Parameters

None.

### Example

```
None.
```

### Notes

Calling the **focus** method on an element is equivalent to selecting that element in the user interface.

### Specification

`focus`

# getAttribute

**getAttribute** returns the value of the named attribute on the current node.

## Syntax

```
attribute = element.getAttribute(name)
```

## Parameters

`name` is the name of the attribute whose value you want to get.

## Example

```
div1 = document.getElementById(“div1”);
a = divvy.getAttribute(“align”);
alert(a); // shows the value of align for that DIV
```

## Notes

**getAttribute** is another very common and useful method for web developers. The corollary to this method is **setAttributeNS**, which allows you to change the value of a named attribute.

## Specification

`getAttribute`

# getAttributeNS

**getAttributeNS** returns the value of the attribute with the given name and namespace.

## Syntax

```
attribute = element.getAttributeNS(namespace, name)
```

### Parameters

*attribute* is an attribute node

*namespace* is the namespace of the requested attribute.

name is the name of the attribute whose value you want to get.

### Example

```
div1 = document.getElementById(“div1”);
a = divvy.getAttributeNS(
  “www.mozilla.org/ns/specialspace/”,
  “special-align”);
alert(a); // shows the value of align for that DIV
```

### Notes

**getAttributeNS** is another very common and useful method for web developers. It differs from **getAttribute** in that it allows you to further specify the requested attribute as being part of a particular namespace, as in the example above, where the attribute is part of the fictional “specialspace” namespace on mozilla.

The corollary to this method is **setAttributeNS**, which allows you to change the value of a named attribute.

### Specification

getAttributeNS

# getAttributeNode

Returns the attribute of the current element as a separate node.

### Syntax

*attributeNode* = element.getAttributeNode(*nodeName*)

### Parameters

`nodeName` is a string containing the name of the node.

`attributeNode` is a separate Attribute node.

### Example

```
// html: <div id=”top” />
t = document.getElementById(“top”);
iNode = t.getAttributeNode(“id”);
// iNode.value = “top”
```

### Notes

The `Attribute` node inherits from node, but is not considered a part of the document tree. Common node attributes like `parentNode`, `previousSibling`, and `nextSibling` are `NULL` for an Attribute node. You can, however, get the element to which the attribute belongs with the `ownerElement` property.

### Specification

`getAttributeNode`

# getAttributeNodeNS

Returns the attribute with the given namespace and name as a separate node.

### Syntax

*`attributeNode`* `= element.getAttributeNode(`*`namespace`*`, `*`nodeName`*`)`

### Parameters

*`namespace`* is a string containing the namespace of the attribute.

`nodeName` is a string containing the name of the node.

`attributeNode` is a separate Attribute node.

### Example

```
// html: <div id=”top” />
t = document.getElementById(“top”);
specialNode = t.getAttributeNodeNS(
  “http://www.mozilla.org/ns/specialspace”,
  “id”);
// iNode.value = “full-top”
```

### Notes

The `Attribute` node inherits from node, but is not considered a part of the document tree. Common node attributes like `parentNode`, `previousSibling`, and `nextSibling` are `NULL` for an Attribute node.

You can, however, get the element to which the attribute belongs with the `ownerElement` property.

**getAttributeNodeNS** is more specific than **getAttributeNode** in that it allows you to specify attributes that are part of a particular namespace. The corresponding setter method is **setAttributeNodeNS**.

### Specification

`getAttributeNodeNS`

# getElementsByTagName

Returns a list of the child elements of a given name on the current element.

### Syntax

*`elements`* `= element.getElementsByTagName(`*`Name`*`)`

### Parameters

`elements` is a `nodeList` of elements.

`tagName` is a string representing the name of the elements.

### Example

```
// check the alignment on a number of cells in a table.
table = document.getElementById(“forecast-table”);
cells = table.getElementsByTagName(“td”);
for (var i = 0; i < cells.length; i++) {
    status = cells[i].getAttribute(“status”);
    if ( status == “open”) {
        // grab the data
    }
}
```

### Notes

**getElementsByTagName** on the element is the same as **getElementsByTagName** on the document, except that its search is restricted to those elements which are children of the current element.

### Specification

`getElementsByTagName`

# hasAttribute

**hasAttribute** is a boolean value indicating whether the current element has the specified attribute or not.

### Syntax

`[ true | false ] = element.hasAttribute(`*`attName`*`)`

### Parameters

boolean `true | false`

*`attName`* is a string representing the name of the attribute

### Example

```
// check that the attirbute exists
// before you set a value
d = document.getElementById(“div1”);
if d.hasAttribute(“align”) {
  d.setAttribute(“align”, “center”);
}
```

### Notes

None.

### Specification

`hasAttribute`

# hasAttributeNS

**hasAttribute** is a boolean value indicating whether the current element has an attribute with the specified name and namespace.

### Syntax

`[ true | false ] = element.hasAttribute(`*`namespace, localName`*`)`

### Parameters

boolean `true | false`

*`namespace`* is a string representing the namespace you are looking for

*`localName`* is a string representing the name of the attribute

### Example

```
// check that the attirbute exists
// before you set a value
d = document.getElementById(“div1”);
if d.hasAttributeNS(
  “http://www.mozilla.org/ns/specialspace/”,
  “special-align”) { d.setAttribute(“align”, “center”);
}
```

### Notes

None.

### Specification

`hasAttributeNS`

# hasAttributes

**hasAttributes** is a boolean value indicating whether the current element has any attributes.

### Syntax

```
[ true | false ] = element.hasAttributes
```

### Parameters

boolean `true | false`

### Example

```
t1 = document.getElementById(“table-data”);
if ( t1.hasAttributes ) {
    // do something with
    // t1.attributes
}
```

### Notes

None.

### Specification

`hasAttributes`

# hasChildNodes

**hasChildNodes** is a method that returns a boolean value indicating whether the current element has children or not.

### Syntax

```
[ true | false ] = element.hasChildNodes()
```

### Parameters

boolean `true | false`

### Example

```
t1 = document.getElementById(“table-data”);
if ( t1.hasChildNodes() ) {
    // table has kids
}
```

### Notes

Note that `element.hasChildNodes`, without the parentheses, is the incorrect usage of this method, and always returns a `true` value indicating that the method is available on the object. Do not be fooled.

### Specification

`hasChildNodes`

# insertBefore

The `insertBefore` method allows you to insert a node before a reference element as a child of the current node.

## Syntax

```
insertedElement = element.insertBefore(
   newElement, targetElement)
```

## Parameters

| | |
|---|---|
| *`insertedElement`* | The node being inserted. |
| *`newElement`* | The node to insert. |
| *`targetElement`* | The node before which `newElement` is inserted. |

## Example

```
parentDiv = document.getElementById("parentDiv");
sp2 = document.getElementById("childSpan");
sp1 = document.createElement("span");
parentDiv.insertBefore(sp1, sp2);
```

## Notes

None.

## Specification

`insertBefore`

# item

The `item` method retrieves a node from the tree by index.

### Syntax

*nodeItem* = element.item(*index*)

### Parameters

nodeItem is a node.

index is the index of the node to be fetched. Index is zero-based.

### Example

```
tbls = document.getElementsByTagName(“table”);
first_table = tbls.item(1);
```

### Notes

A value of NULL is returned if the index is out of range.

### Specification

item

# nextSibling

Returns the node immediately following the current one in the tree.

### Syntax

*node* = elementNode.nextSibling

### Parameters

*node* is a node object.

### Example

```
// in a table, the cells are siblings
cell1 = document.getElementById(“td1”);
cell2 = cell1.nextSibling;
```

### Notes

Returns `NULL` if there are no more nodes.

### Specification

`nextSibling`

# normalize

The normalize method puts the current node and all of its subtree into a “normalized” form (see below).

### Syntax

`element.normalize()`

### Parameters

None.

### Example

```
example here
```

### Notes

The normalized form of a subtree is that subtree’s nodelist cleansed of extranous and adjacent Text nodes.

### Specification

`normalize`

# removeAttribute

The **removeAttribute()** method removes an attribute from the current element.

## Syntax

`element.removeAttribute(`*`attName`*`)`

## Parameters

`attName` is a string that names the attribute to be removed from the current node.

## Example

```
// <div align=”left” width=”200px” />
d = document.getElementById(“div1”);
d.removeAttribute(“align”);
// now: <div width=”200px” />
```

## Notes

**removeAttribute** allows you to change the attribute list dynamically on the current node.

## Specification

`removeAttribute`

# removeAttributeNS

The **removeAttributeNS()** method removes an attribute with the specified namespacde and name.

### Syntax

`element.removeAttribute(`*`namespace, attName`*`)`

### Parameters

`namespace` is a string that contains the namespace of the specified attribute.

`attName` is a string that names the attribute to be removed from the current node.

### Example

```
// <div special-align=”utterleft” width=”200px” />
d = document.getElementById(“div1”);
d.removeAttributeNS(
  “http://www.mozilla.org/ns/specialspace”,
  “special-align”);
// now: <div width=”200px” />
```

### Notes

**removeAttributeNS** allows you to change the attribute list dynamically on the current node.

### Specification

`removeAttributeNS`

# removeAttributeNode

**removeAttributeNode** removes the specified attribute from the current element.

### Syntax

*`remove_attr`* `= element.removeAttributeNode(`*`attribute`*`)`

### Parameters

`attribute` is the Attribute node that needs to be removed.

`remove_attr` is an Attribute node.

### Example

```
// <div id=”top” align=”center” />
d = document.getElementById(“top”);
d_align = d.getAttributeNode(“align”);
d.removeAttributeNode(d_align);
// align has a default value, center,
// so the removed attribute is immediately
// replaced: <div id=”top” align=”center” />
```

### Notes

If the removed Attribute has a default value it is immediately replaced. The replacing attribute has the same namespace URI and local name, as well as the original prefix, when applicable.

### Specification

`removeAttributeNode`

# removeChild

The **removeChild()** method removes a child node from the current element.

### Syntax

*`oldChild`* `= element.removeChild(`*`child`*`)`

### Parameters

`oldChild` is the node that needs to be removed.

`child` is a node.

### Example

```
// <div id=”top” align=”center”><div id=”nested”/></
div>
d = document.getElementById(“top”);
d_nested = document.getElementById(“nested”);
throwaway_node = d.removeChild(d_nested);
```

### Notes

None.

### Specification

`removeChild`

# removeEventListener

**removeEventListener()** allows the removal of event listeners from the event target.

### Syntax

`element.removeEventListener(`*`type`*`, `*`listener`*`, `*`useCapture`*`)`

### Parameters

The `removeEventListener()` method takes the following parameters:

| | |
|---|---|
| `type` | A string representing the event type being registered. |
| `listener` | The listener parameter takes an interface implemented by the user which contains the methods to be called when the event occurs. |
| `useCapture` | If true, useCapture indicates that the user wishes to initiate capture. After initiating capture, all events of the specified type will be dispatched to the registered EventListener before being dispatched to any EventTargets beneath them in the tree. Events which are bubbling upward through the tree will not trigger an EventListener designated to use capture. |

## Example

```
example here
```

## Notes

If an EventListener is removed from an EventTarget while it is processing an event, it will not be triggered by the current actions. EventListeners can never be invoked after being removed.

Calling **removeEventListener** with arguments which do not identify any currently registered EventListener on the EventTarget has no effect.

See also **addEventListener**.

## Specification

`removeEventListener`

# replaceChild

The **replaceChild()** method replaces one child node on the current element with another.

## Syntax

`element.replaceChild(`*`newChild`*`, `*`oldChild`*`)`

## Parameters

`newChild` is a node.

`oldChild` is the existing child node to be replaced.

## Example

```
<html>
<head>
<script language="javascript">
  function init()
  {
d1 = document.getElementById("top");
d2 = document.getElementById("in");
d_new = document.createElement("p");
d1.replaceChild(d_new, d2);
alert(d1.childNodes[1].nodeName)
}
</script>
</head>

<body onload="init()">
<div id="top" align="left">
  <div id="in">in
</div>

</body>
</html>
```

### Notes

extra information

### Specification

`replaceChild`

# setAttribute

**setAttribute** adds a new attribute or changes the value of an existing attribute on the current element.

### Syntax

`element.setAttribute(`*`name`*`, `*`value`*`)`

### Parameters

`name` is the name of the new attribute as a string.

`value` is the desired value of the new attribute.

### Example

s

```
d = document.getElementById(“d1”);
d.setAttribute(“align”, “center”);
```

### Notes

If the attribute named already exists, then the value of that attribute is changed to the value passed in as part of this function. If it does not exist, then a new attribute node is created.

### Specification

`setAttribute`

# setAttributeNS

**setAttributeNS** adds a new attribute or changes the value of an attribute with the given namespace and name.

## Syntax

`element.setAttribute(`*`namespace, name, value`*`)`

## Parameters

`namespace` is the namespace of the new attribute as a string.

`name` is the name of the new attribute as a string.

`value` is the desired value of the new attribute.

## Example

This example splits the text in a paragraph element. The new, second child of p2 is the sibling node hat contains the text after the split. `data` gets the text from the `text` object.

```
d = document.getElementById(“d1”);
d.setAttributeNS(
  “http://www.mozilla.org/ns/specialspace”,
  “align”,
  “center”);
```

## Notes

If the attribute named already exists, then the value of that attribute is changed to the value passed in as part of this function. If it does not exist, then a new attribute node is created.

## Specification

`setAttributeNS`

# setAttributeNode

**setAttributeNode** adds a new attribute node to the current element.

## Syntax

*replaced_attr* = element.setAttributeNode(*attribute*)

## Parameters

`attribute` is a node of type `Attribute`

`replaced_attr` is the replaced attribute node, if any, returned by this function

## Example

```
// <div id=”one” align=”left”>one</div>
// <div id=”two”>two</div>
d1 = document.getElementById("one");
d2 = document.getElementById("two");
a = d1.getAttributeNode(“align”);
d2.setAttributeNode(a);
alert(d2.attributes[1].value)
// returns: ‘left’
```

## Notes

If the attribute named already exists on the element, then that attribute is replaced with the new one and the replaced one is returned.

Note that this function does not the set the value of the new attribute, only creates it on the element. Use **setAttribute** to set or change the value on an existing node.

## Specification

`setAttributeNode`

# setAttributeNodeNS

**setAttributeNodeNS** adds a new attribute node with the specified namespace and name.

## Syntax

```
replaced_attr =
  element.setAttributeNodeNS(namespace, attribute)
```

## Parameters

*namespace* is the namespace of the attribute as a string.

*attribute* is a node of type `Attribute`.

*replaced_attr* is the replaced attribute node, if any, returned by this function.

## Example

```
// <div id=”one” special-align=”utterleft”>one</div>
// <div id=”two”>two</div>
myns = “http://www.mozilla.org/ns/specialspace”;

d1 = document.getElementById("one");
d2 = document.getElementById("two");
a = d1.getAttributeNodeNS(
  myns,
  “special-align”);
d2.setAttributeNodeNS(
  myns,
  a);
alert(d2.attributes[1].value)
// returns: ‘utterleft’
```

## Notes

If the attribute named already exists on the element, then that attribute is replaced with the new one and the replaced one is returned. The corresponding getter method for namespaced attribute nodes is **getAttributeNodeNS**.

Note that this function does not the set the value of the new attribute, only creates it on the element. Use **setAttributeNS** to set or change the value on an existing node within a particular namespace.

### Specification

`setAttributeNodeNS`

# supports

The **supports** method tests if this DOM implementation supports a particular feature.

### Syntax

`boolean = element.supports(`*`feature`*`[, `*`version`*`])`

### Parameters

`feature` is a string that contains the name of the feature (e.g., “”)

`version` is a string containing the version number of the feature.

### Example

```
if ( document.supports(“package”, “4.0”) ) {
    // do something that only package 4.0 allows
}
```

### Notes

If version is not supplied, the method returns true is any version of the specified feature is supported.

### Specification

*Not part of the specification.*

# onblur

The **onblur** property returns the onBlur event handler code, if any, that exists on the current element.

## Syntax

```
event handling code = element.onblur
```

## Example

```
warnFunc = window.onblur;
```

## Notes

The blur event is raised when an element loses focus.

## Specification

Not part of specification.

# onclick

The **onclick** property returns the onClick event handler code on the current element.

## Syntax

```
event handling code = element.onclick
```

### Example

Perhaps the simplest example of using the onclick DOM property is to retrieve the existing onclick event handler code. The following function sets the event handler code, then gets it and displays it.

```
function pawnClick()  {
  p = document.getElementById("mutable");
  p.onclick = "alert('moot!');";
  text = p.onclick;
  alert(text);
}
```

### Notes

The click event is raised when the user clicks on an element.

### Specification

Not part of specification.

# ondblclick

The **ondblclick** property returns the onDblClick event handler code on the current element.

### Syntax

```
event handling code = element.ondblclick
```

### Example

```
// <img src=”pawn.gif” onDblClick=”movePawn(this);” />
function pawnClick()  {
  i = document.getElementById("img1");
  alert(i.ondblclick);
}
// alerts: function anonymous(event) { movePawn(this) }
```

### Notes

The dblclick event is raised when a user double-clicks on an element.

### Specification

Not part of specification.

# onfocus

The **onfocus** property returns the onFocus event handler code on the current element.

### Syntax

```
event handling code = element.onfocus
```

### Example

```
None.
```

### Notes

The focus event is raised when the user sets focus on the given element.

### Specification

Not part of specification.

# onkeydown

The **onkeydown** property returns the onKeyDown event handler code on the current element.

### Syntax

```
event handling code = element.onkeydown
```

### Example

```
None.
```

### Notes

The keydown event is raised when the user presses a keyboard key.

### Specification

Not part of specification.

# onkeypress

The **onkeypress** property returns the onKeyPress event handler code for the current element.

### Syntax

```
event handling code = element.onkeypress
```

### Example

```
None.
```

### Notes

The keypress event is raised when the user presses a key on the keyboard.

### Specification

Not part of specification.

# onkeyup

The **onkeyup** property returns the onKeyUp event handler code for the current element.

## Syntax

```
event handling code = element.onclick
```

## Example

```
None.
```

## Notes

The keyup event is raised when the user releases a key that's been pressed.

## Specification

Not part of specification.

# onmousedown

The **onmousedown** property returns the onMouseDown event handler code on the current element.

## Syntax

```
event handling code = element.onmousedown
```

## Example

```
None.
```

## Notes

The mousedown event is raised when the user presses the left button button.

### Specification

Not part of specification.

# onmousemove

The **onmousemove** propety returns the onMouseMove event handler code on the current element.

### Syntax

*event handling code* = element.onmousemove

### Example

```
movement = element.onmousemove
```

### Notes

The mousemove event is raised when the user moves the mouse.

### Specification

Not part of specification.

# onmouseout

The **onmouseout** property returns the onMouseOut event handler code on the current element.

### Syntax

*event handling code* = element.onmouseout

### Example

```
None.
```

### Notes

The mouseout event is raised when the mouse leaves an element (e.g, when the mouse moves off of an image in the web page, the mouseout event is raised for that image element).

### Specification

Not part of specification.

# onmouseover

The **onmouseover** property returns the onMouseOver event handler code on the current element.

### Syntax

```
event handling code = element.onmouseover
```

### Example

```
None.
```

### Notes

The mouseover event is raised when the user moves the mouse over a particular element.

### Specification

Not part of specification.

# onmouseup

The **onmouseup** property returns the onMouseUp event handler code on the current element.

## Syntax

```
event handling code = element.onmouseup
```

## Example

```
None.
```

## Notes

The mouseup event is raised when the user releases the left mouse button.

## Specification

Not part of specification.

# onresize

The **onresize** property returns the onResize event handler code on the current element.

## Syntax

```
event handling code = element.onresize
```

## Example

```
// <img src=”pawn.gif” onResize=”growBoard();” />
function pawnClick()  {
  i = document.getElementById("img1");
  alert(i.onresize);
}
// alerts: function anonymous(event) { growBoard() }
```

## Notes

The resize event is raised when the user resizes a resizable element (such as a window).

## Specification

Not part of specification.

# DOM `window` Reference

This chapter provides a brief reference for all of the methods, properties, and events available through the DOM `window` object.

The `window` object represents the window itself. Typically, `window` contains the `document` as a child (see **DOM Document Reference**), provides access to the **window.navigator** and **window.screen** objects for manipulating the browsing environment itself, and provides a number of special properties for accessing the object model below it.

## DOM `window` Interface

The properties, methods, and event handlers of the `window` object are given here.

## Properties

| | |
|---|---|
| **window._content** | Returns a reference to the content element in the current window. |
| **window.closed** | This property indicates whether the current window is closed or not. |
| **window.Components** | Returns an array of the components installed in the browser. |
| **window.controllers** | Returns the XUL controller objects for the current chrome window. |
| **window.crypto** | Returns the browser `crypto` object |
| **window.defaultStatus** | Gets/sets the status bar text for the given window. |

| | |
|---|---|
| **window.directories** | Returns a reference to the directories toolbar in the current chrome. |
| **window.document** | Returns a reference to the document that the window contains. |
| **window.frames** | Returns an array of the subframes in the current window. |
| **window.history** | Returns a reference to the `history` object. |
| **window.innerHeight** | Gets/sets the height of the content area of the browser window. |
| **window.innerWidth** | Gets/sets the height of the content area of the browser window. |
| **window.length** | Returns the number of frames in the window. |
| **window.location** | Gets/sets the location, or current URL, of the window object. |
| **window.locationbar** | Returns the `locationbar` object, whose visibility can be toggled in the window. |
| **window.menubar** | Returns the `menubar` object, whose visibility can be toggled in the window. |
| **window.name** | Gets/sets the name of the window. |
| **window.navigator** | Returns a reference to the `navigator` object. |
| **window.navigator.appCodeName** | Returns the internal “code” name of the current browser. |
| **window.navigator.appName** | Returns the official name of the browser. |
| **window.navigator.appVersion** | Returns the version of the browser as a string. |

| | |
|---|---|
| **window.navigator.cookieEnabled** | Returns a boolean indicating whether cookies are enabled in the browser or not. |
| **window.navigator.language** | Returns a string representing the language version of the browser. |
| **window.navigator.mimeTypes** | Returns a list of the MIME types supported by the browser. |
| **window.navigator.oscpu** | Returns a string that represents the current operating system. |
| **window.navigator.platform** | Returns a string representing the platform of the browser. |
| **window.navigator.plugins** | Returns an array of the plugins installed in the browser. |
| **window.navigator.product** | Returns the product name of the browser (e.g., “Gecko”) |
| **window.navigator.productSub** | Returns the product version number (e.g., “5.0”) |
| **window.navigator.userAgent** | Returns the user agent string for the current browser. |
| **window.navigator.vendor** | Returns the vendor name of the current browser (e.g., “Netscape6”) |
| **window.navigator.vendorSub** | Returns the vendor version number (e.g., “6.1”) |
| **window.opener** | Returns a reference to the window that opened this current window. |
| **window.outerHeight** | Gets/sets the height of the outside of the browser window. |
| **window.outerWidth** | Gets/sets the width of the outside of the browser window. |
| **window.pageXOffset** | Gets the amount of content that has been hidden by scrolling to the right |
| **window.pageYOffset** | Gets the amount of content that has been hidden by scrolling down. |
| **window.parent** | Returns a reference to the parent of the current window or subframe. |

| | |
|---|---|
| **window.personalbar** | Returns the `personalbar` object, whose visibility can be toggled in the window. |
| **window.pkcs11** | Returns the `pkcs11` object , which can be used to install drivers other software associated with the pkcs11 protocol. |
| **window.prompter** | Returns a reference to the prompt window, if any, currently displayed. |
| **window.screen** | Returns a reference to the `screen` object associated with the window. |
| **window.screen.availHeight** | Specifies the y-coordinate of the first pixel that is not allocated to permanent or semipermanent user interface features. |
| **window.screen.availLeft** | Returns the first available pixel available from the left side of the screen. |
| **window.screen.availTop** | 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. |
| **window.screen.availWidth** | Returns the amount of horizontal space in pixels available to the window. |
| **window.screen.colorDepth** | Returns the color depth of the screen. |
| **window.screen.height** | Returns the height of the screen in pixels. |
| **window.screen.left** | Gets/sets the current distance in pixels from the left side of the screen. |
| **window.screen.pixelDepth** | Getst the bit depth of the screen. |
| **window.screen.top** | Gets/sets the distance from the top of the screen. |
| **window.screen.width** | Returns the width of the screen. |
| **window.screenX** | Returns the horizontal distance of the left border of the user's browser from the left side of the screen. |
| **window.screenY** | Returns the vertical distance of the top border of the user's browser from the top side of the screen. |

| | |
|---|---|
| **window.scrollbars** | Returns the `scrollbars` object, whose visibility can be toggled in the window. |
| **window.scrollX** | Returns the number of pixels that the document has already been scrolled horizontally. |
| **window.scrollY** | Returns the number of pixels that the document has already been scrolled vertically. |
| **window.self** | Returns an object reference to the `window` object itself. |
| **window.sidebar** | Returns a reference to the `window` object of the sidebar. |
| **window.status** | Gets/sets the text in the statusbar at the bottom of the browser. |
| **window.statusbar** | Returns the `statusbar` object, whose visibility can be toggled in the window. |
| **window.toolbar** | Returns the `toolbar` object, whose visibility can be toggled in the window. |
| **window.top** | Returns a reference to the topmost window in the window hierarchy. |
| **window.window** | Returns a reference to the current window. |

## Methods

| | |
|---|---|
| **window.alert()** | Displays an alert dialog. |
| **window.back()** | Moves back one in the window history. |
| **window.blur()** | Sets focus away from the window. |
| **window.captureEvents()** | Registers the window to capture all events of the specified type. |
| **window.clearInterval()** | Clears a delay that's been set for a specific function. |
| **window.clearTimeout()** | Clears the delay set by **window.setTimeout()**. |
| **window.close()** | Closes the current window. |
| **window.confirm()** | Displays a dialog with a message that the user needs to respond to. |
| **window.dump()** | Writes a message to the console. |
| **window.escape()** | Encodes a string. |
| **window.focus()** | Sets focus on the current window. |
| **window.forward()** | Moves the window one document forward in the history. |
| **window.GetAttention()** | Flashes the application icon. |
| **window.getSelection()** | Returns the selection object representing the selected item(s). |
| **window.home()** | Returns the browser to the home page. |
| **window.moveBy()** | Moves the current window by a specified amount. |

| | |
|---|---|
| **window.moveTo()** | Moves the window to the specified coordinates. |
| **window.open()** | Opens a new window. |
| **window.print()** | Prints the current document. |
| **window.prompt()** | Returns the text entered by the user in a prompt dialog. |
| **window.releaseEvents()** | Releases the window from trapping events of a specific type. |
| **window.resizeBy()** | Resizes the current window by a certain amount. |
| **window.resizeTo()** | Dynamically resizes window. |
| **window.scroll()** | Scrolls the window to a particular place in the document. |
| **window.scrollBy()** | Scrolls the document in the window by the given amount. |
| **window.scrollByLines()** | Scrolls the document by the given number of lines. |
| **window.scrollByPages()** | Scrolls the current document by the specified number of pages. |
| **window.scrollTo()** | Scrolls to a particular set of coordinates in the document. |
| **window.setCursor()** | Changes the cursor. |
| **window.setInterval()** | Set a delay for a specific function. |
| **window.setTimeout()** | Sets a delay for executing a function. |

| | |
|---|---|
| **window.sizeToContent()** | Sizes the window according to its content. |
| **window.stop()** | This method stops window loading. |
| **window.unescape()** | Unencodes a value that has been encoded in hexadecimal (e.g., a cookie). |

**window.updateCommands()**

## Event Handlers

| | |
|---|---|
| **window.onabort** | An event handler property for abort events on the window. |
| **window.onblur** | An event handler property for blur events on the window. |
| **window.onchange** | An event handler property for change events on the window. |
| **window.onclick** | An event handler property for click events on the window. |
| **window.onclose** | An event handler property for handling the window close event. |
| **window.ondragdrop** | An event handler property for drag and drop events on the window. |
| **window.onerror** | An event handler property for errors raised on the window. |
| **window.onfocus** | An event handler property for focus events on the window. |
| **window.onkeydown** | An event handler property for keydown events on the window. |
| **window.onkeypress** | An event handler property for keypress events on the window. |

| | |
|---|---|
| **window.onkeyup** | An event handler property for keyup events on the window. |
| **window.onload** | An event handler property for window loading. |
| **window.onmousedown** | An event handler property for mouse-down events on the window. |
| **window.onmousemove** | An event handler property for mouse-move events on the window. |
| **window.onmouseout** | An event handler property for mouse-out events on the window. |
| **window.onmouseover** | An event handler property for mou-seover events on the window. |
| **window.onmouseup** | An event handler property for mouseup events on the window. |
| **window.onpaint** | An event handler property for paint events on the window. |
| **window.onreset** | An event handler property for reset events on the window. |
| **window.onresize** | An event handler property for window resizing |
| **window.onscroll** | An event handler property for window scrolling. |
| **window.onselect** | An event handler property for window selection. |
| **window.onsubmit** | An event handler property for submits on window forms |
| **window.onunload** | An event handler property for unload events on the window. |

# window.alert()

Display an alert dialog with the specified text.

## Syntax

```
window.alert(text)
```

## Parameters

`text` is a string of the text you want displayed in the alert dialog.

## Example

```
window.alert("I'm a Scorpio!");
```


## Notes

The alert dialog should be used for messages which do not require any response of the part of the user. See also **window.confirm()**, **window.prompt()**.

## Specification

DOM Level 0. Not part of specification.

# window._content

Returns a reference to the content element in the current window.

## Syntax

```
cHolder = window._content
```

### Parameters

`cHolder` is an object reference.

### Example

```
// for HTML <iframe
//    type="content-primary"
//    src="blur.html" />
function cont() {
  loc = window._content.location.href;
  alert(loc);
}
```

### Notes

If there is no particular element or subframe given as the content element, this method returns a reference to the window itself. Also note that this does not give you a very good way to refer to a number of different content elements. In this case, you may want to use the `getElementById` method to get references to the subframes you want.

This property is effectively the same as **window.content**. It's generally used to get the content from a browser window.

### Specification

`DOM Level 0. Not part of specification.`

# window.back()

Returns the window to the previous item in the history.

### Syntax

```
window.back()
```

### Parameters

None.

### Example

```
function goBack() {
   if ( canGoBack )
      window.back();
}
```

### Notes

None.

### Specification

DOM Level 0. Not part of specification.

# window.blur()

Shifts focus away from the window.

### Syntax

```
window.blur()
```

### Parameters

None.

### Example

```
window.blur();
```

### Notes

The **window.blur()** method is the programmatic equivalent of the user shifting focus away from the current window.

### Specification

DOM Level 0. Not part of specification.

# window.captureEvents()

Registers the window to capture all events of the specified type.

### Syntax

window.captureEvents(Event.*eventType*)

### Parameters

eventType is a string with one of the following values:

| | |
|---|---|
| Abort | Load |
| Blur | MouseDown |
| Click | MouseMove |
| Change | MouseOut |
| DblClick | MouseOver |
| DragDrop | MouseUp |
| Error | Move |
| Focus | Reset |
| KeyDown | Resize |
| KeyPress | Select |
| KeyUp | Submit |
| | Unload |

## Example

```
<html>
<script>
function reg() {
  window.captureEvents(Event.CLICK);
  window.onclick = hit;
}

function hit() {
  alert('hit');
}
</script>

<body onload="reg();">
<button>test</button>
<div id="d"> </div>
</body>
</html>
```

## Notes

Events raised in the DOM by user activity (such as clicking buttons or shifting focus away from the current document) generally pass through the high-level `window` and `document` objects first before arriving at the object that initiated the event.

When you call the **captureEvents()** method on the `window`, events of the type you specify (for example, `Event.CLICK`) no longer pass through to "lower" objects in the hierarchy. In order for events to "bubble up" in the way that they normally do, you must call **releaseEvents()** on the window to keep it from trapping events.

Also note that the **eventType** parameter is case-insensitive, so you can also say, for example, `window.releaseEvents(Event.KeyPress)`.

See also **window.releaseEvents()**.

## Specification

`DOM Level 0. Not part of specification.`

# window.clearInterval()

Clears a delay that's been set for a specific function.

### Syntax

`window.clearInterval(`*`intervalID`*`)`

### Parameters

`intervalID` is the ID of the specific interval you want to clear.

### Example

```
window.clearInterval(animID);
```

### Notes

See also **window.setInterval()**.

### Specification

`DOM Level 0. Not part of specification.`

# window.clearTimeout()

Clears the delay set by **window.setTimeout()**.

### Syntax

`window.clearTimeout(`*`timeoutID`*`)`

### Parameters

`timeoutID` is the ID of the timeout you wish you clear.

### Example

```
window.clearTimeout(inactive_ID);
```

### Notes

The ID for the timeout is returned by the **window.setTimeout()** function.

### Specification

DOM Level 0. Not part of specification.

# window.close()

Closes this window.

### Syntax

```
window.close()
```

### Parameters

None.

### Example

```
window.close();
```

### Notes

None.

### Specification

DOM Level 0. Not part of specification.

# window.closed

This property indicates whether the current window is closed or not.

### Syntax

```
bRes = window.closed
```

### Parameters

`bRes` is a boolean value.

### Example

```
if ( window.opener.closed ) {
    // the window that opened me has been closed!
}
```

### Notes

Addtional notes.

### Specification

DOM Level 0. Not part of specification.

# window.Components

Returns a reference to the XPCOM components that are installed in Mozilla/ Netscape.

### Syntax

```
componentList = window.Components
```

### Parameters

*`componentList`* is a read-only array of XPCOM components accessible via XPConnect.

### Example

```
// use the Components property to get a particular class
// and a related interface
var cp = Components.classes['@mozilla.org/preferences;1']
var icp = Components.interfaces.nsIPref;
```

### Notes

The `Components` property does not work from HTML and XML pages loaded as content in the browser (i.e., it returns "`[object nsXPCComponents]`", which can't be further interrogated), since those pages are not typically considered part of the trusted application chrome and content that can use XPConnect. From trusted scripts, however, `Components` can be used to get and use the XPCOM objects which the browser itself uses for its internal functionality, as in the example above.

See the following document for more information about the `Components` object:

- http://www.mozilla.org/scriptable/components_object.html

### Specification

`DOM Level 0. Not part of specification.`

# window.confirm()

Displays a dialog with a message that the user needs to respond to.

### Syntax

*`result`* `= window.confirm(`*`text`*`)`

### Parameters

*`text`* is a string.

*`result`* is a boolean value indicating whether OK or Cancel was selected.

### Example

```
if ( window.confirm(“Want to see my mood ring?”)) {
    window.open(“mood.html”, “mood ring”, “”);
}
```


### Notes

Unlike the alert dialog, a confirm dialog has OK and Cancel buttons, and returns true only when the user confirms the choice being presented by clicking OK. The return value of **window.confirm()** is often tested as part of a conditional block, as in the example above. See also **window.alert()**, **window.prompt()**

### Specification

DOM Level 0. Not part of specification.

# window.controllers

Returns the XUL controllers of the chrome window.

### Syntax

*controllers* = window.controllers

### Parameters

*controllers* is an array of objects of the type XUL Controllers.

### Example

```
<script>
  function con() {
    alert(window.controllers);
  }
</script>
// displays: [object XULControllers]
```

### Notes

None.

### Specification

DOM Level 0. Not part of specification.

# window.crypto

Returns the browser `crypto` object, which can then be used to manipulate various browser security features.

### Syntax

```
syntax code
```

### Parameters

blah is a blah.

### Example

```
// example code here
```

### Notes

Addtional notes.

### Specification

DOM Level 0. Not part of specification.

# window.defaultStatus

Gets/sets the status bar text for the given window.

### Syntax

```
sMsg = window.defaultStatus
window.defaultStatus = sMsg
```

### Parameters

`sMsg` is a string containing the text to be displayed by default in the statusbar.

### Example

```
<html>
<body onload="window.defaultStatus='hello!';"/>
<button onclick="window.confirm('Are you sure you want
to quit?');">confirm</button>
</body>
</htm>
```

### Notes

To set the status once the window has been opened, use **window.status**.

### Specification

DOM Level 0. Not part of specification.

# window.directories

Returns the `window` directories toolbar object.

## Syntax

```
dirBar = window.directories
```

## Parameters

*`dirBar`* is an object of the type `barProp`.

## Example

```
<script>
   function dirs() {
      alert(window.directories);
   }
 </script>
```

## Notes

None.

## Specification

`DOM Level 0. Not part of specification.`

# window.document

Returns a reference to the document that the window contains.

## Syntax

```
doc = window.document
```

## Parameters

`doc` is an object reference.

### Example

```
doc= window.document;
window.dump(doc.title);
// prints the current document’s title to the console.
```

### Notes

None.

### Specification

DOM Level 0. Not part of specification.

# window.dump()

Prints messages to the console.

### Syntax

`window.dump(`*`text`*`)`

### Parameters

`text` is a string.

### Example

```
doc= window.document;
window.dump(doc.title);
// prints the current document’s title to the console.
```

### Notes

If you have no console available, this method prints nothing but doesn’t raise an error. window.dump is commonly used to print statements to the console can be used to debug JavaScript used to access the DOM.

### Specification

DOM Level 0. Not part of specification.

# window.escape()

Encodes a string.

### Syntax

*sEscaped* = window.escape(*sRegular*)

### Parameters

sEscaped is the endoded string.

sRegular is a regular string

### Example

```
alert(escape("http://www.cnn.com"));
// displays: http%3Awww.cnn.com
```

### Notes

The **escape()** method converts special characters (any characters that are not regular text or numbers) into hexadecimal characters, which is especially necessary for setting the values of cookies.

### Specification

DOM Level 0. Not part of specification.

# window.focus()

Sets focus on the window.

### Syntax

```
window.focus()
```

### Parameters

None.

### Example

```
if (clicked) { window.focus(); }
```

### Notes

None.

### Specification

```
DOM Level 0. Not part of specification.
```

# window.forward()

Moves the window one document forward in the history.

### Syntax

```
window.forward()
```

### Parameters

None.

### Example

```
function goForward() {
   if ( canGoForward)
      window.forward();
}
```

### Notes

None.

### Specification

DOM Level 0. Not part of specification.

# window.frames

Returns an array of the subframes in the current window.

### Syntax

*frameList* = window.frames

### Parameters

`frameList` is an array of `frame` objects.

### Example

```
frames = window.frames;
for (var i = 0; i < frames.length; i++) {
  // do something with each subframe as frames[i]
}
```

### Notes

None.

### Specification

DOM Level 0. Not part of specification.

## window.GetAttention()

Flashes the application icon to get the user's attention.

### Syntax

```
window.GetAttention()
```

### Parameters

None.

### Example

```
// from Chatzilla
function notifyAttention (source)
{
  if (typeof source != "object")
    source = client.viewsArray[source].source;

  if (client.currentObject != source)
  {
    var tb = getTabForObject (source, true);
    var vk = Number(tb.getAttribute("viewKey"));

    tb.setAttribute ("state", "attention");
    client.activityList[vk] = "!";
    updateTitle();
  }

  if (client.FLASH_WINDOW)
    window.GetAttention();
}
```

### Notes

On windows and linux, the icon flashes in the system tray. On macintosh, the icon in the upper right corner of the desktop flashes.

### Specification

DOM Level 0. Not part of specification.

## window.getSelection()

Returns a `selection` object representing the selected item(s).

### Syntax

*`selection`* = window.getSelection()

### Parameters

*`selection`* is a `selection` object.

### Example

```
function cutThis() {
  selObj = window.getSelection();
  selText = selObj.toSting();
  if len(selText) ...
```

### Notes

The `selection` object returned by this method is automatically converted to a string as needed. If you asked for the `len` of `selObj` in the listing above, for example, the method would return the length of the object converted into a string (using the selection object's own `toString()` method. But you can also get the ranges in the selection and you can navigate through the selected nodes using these range objects.

See the `nsISelection` interface in mozilla for more information about selection objects and the services they provide for manipulating selections:

http://lxr.mozilla.org/seamonkey/source/content/base/public/nsISelection.idl

### Specification

DOM Level 0. Not part of specification.

# window.history

Returns a reference to the `history` object, which provides an interface for manipulating the browser history.

### Syntax

```
historyObj = window.history
```

### Parameters

historyObject is an object reference.

### Example

```
h = window.history;
if ( h.length ) { // if there is a history
    h.back();     // equivalent to clicking back button
}
```

### Notes

The `history` object provides the following interface:

| | |
|---|---|
| `current` | `back()` |
| `length` | `forward()` |
| `next` | `go()` |
| `previous` | |

You can call access this interface from the `window` object by calling, for example, `window.history.back()`.

### Specification

DOM Level 0. Not part of specification.

# window.home()

Returns the window to the home page.

### Syntax

```
window.home()
```

### Parameters

None.

### Example

```
function goHome() {
   window.home();
}
```

### Notes

None.

### Specification

DOM Level 0. Not part of specification.

# window.innerHeight

Gets/sets the height of the content area of the browser window.

### Syntax

```
window.innerHeight = iPx
iPx = window.innerHeight
```

### Parameters

`iPx` is the number of pixels as an integer.

### Example

```
window.innerHeight = 400;
window.innerWidth = 400;
```

### Notes

See also **window.innerWidth**, **window.outerHeight**.

### Specification

`DOM Level 0. Not part of specification.`

# window.innerWidth

Gets/sets the height of the content area of the browser window.

### Syntax

```
window.innerWidth = iPx
iPx = window.innerWidth
```

### Parameters

`iPx` is the number of pixels as an integer.

### Example

```
window.innerHeight = 400;
window.innerWidth = 400;
```

### Notes

See also **window.innerHeight**, **window.outerHeight**.

### Specification

DOM Level 0. Not part of specification.

# window.length

Returns the number of frames in the window.

### Syntax

```
ifrms = window.length
```

### Parameters

`ifrms` is the number of frames as an integer.

### Example

```
if ( window.length )
    // this is a document with subframes
```

### Notes

None.

### Specification

DOM Level 0. Not part of specification.

# window.location

Gets/sets the location, or current URL, of the window object.

### Syntax

```
url = window.location
window.location = url
```

### Parameters

`url` is a string containing the URL for the specified location.

### Example

```
function getNews() {
   window.location= “http://www.cnn.com”;
}
// in html: <button onclick=”getNews();”>News</button>
```

### Notes

None.

### Specification

DOM Level 0. Not part of specification.

# window.locationbar

Returns the `locationbar` object, whose visibility can be toggled in the window.

### Syntax

```
lBarObj = window.locationbar
```

### Parameters

`lBarObj` is an object reference.

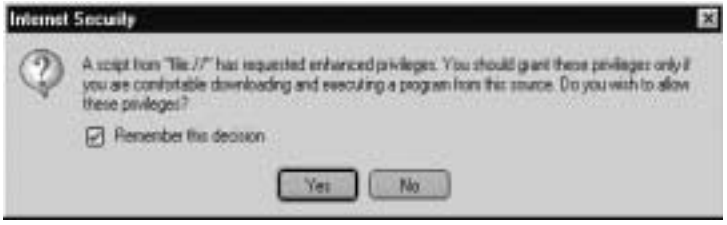
### Example

The following complete HTML example shows way that the visible property of the various “bar” objects is used, and also the change to the privileges necessary to write to the visible property of any of the bars on an existing window.

```
<html>
<head>
  <title>Various DOM Tests</title>
  <script>
    // changing bar states on the existing window
    netscape.security.PrivilegeManager.
        enablePrivilege("UniversalBrowserWrite");
    window.locationbar.visible=
        !window.locationbar.visible;
  </script>
</head>
<body>
  <p>Various DOM Tests</p>
</body>
</html>
```

## Notes

When you load the example page above, the browser displays the following dialog:




To toggle the visibility of these bars, you must either sign your scripts or enable the appropriate privileges, as in the example above. Also be aware that dynamically updating the visibilty of the various toolbars can change the size of the window rather dramatically, and may affect the layout of your page.

See also: **window.locationbar**, **window.menubar**, **window.personalbar**, **window.scrollbars**, **window.statusbar**, **window.toolbar**

### Specification

DOM Level 0. Not part of specification.

# window.menubar

Returns the `menubar` object, whose visibility can be toggled in the window.

### Syntax

*`mBarObj`* = window.menubar

### Parameters

`mBarObj` is an object reference.

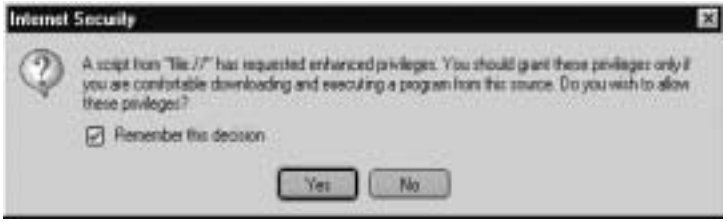
### Example

The following complete HTML example shows way that the visible property of the various “bar” objects is used, and also the change to the privileges necessary to write to the visible property of any of the bars on an existing window.

```
<html>
<head>
  <title>Various DOM Tests</title>
  <script>
    // changing bar states on the existing window
    netscape.security.PrivilegeManager.
        enablePrivilege("UniversalBrowserWrite");
    window.menubar.visible=!window.menubar.visible;
  </script>
</head>
<body>
  <p>Various DOM Tests</p>
</body>
</html>
```

### Notes

When you load the example page above, the browser displays the following dialog:




To toggle the visibility of these bars, you must either sign your scripts or enable the appropriate privileges, as in the example above. Also be aware that dynamically updating the visibilty of the various toolbars can change the size of the window rather dramatically, and may affect the layout of your page.

See also: **window.locationbar**, **window.menubar**, **window.personalbar**, **window.scrollbars**, **window.statusbar**, **window.toolbar**

### Specification

DOM Level 0. Not part of specification.

# window.moveBy()

Moves the current window by a specified amount.

### Syntax

```
window.moveBy(deltaX, deltaY)
```

### Parameters

`deltaX` is the amount of pixels to move the window horizontally.

`deltaY` is the amount of pixels to move the window vertically.

### Example

```
function budge() {
   moveBy(10, -10);
}
```

### Notes

You can use negative numbers as parameters for this function. This function makes a relative move while **window.moveTo()** makes an absolute move.

### Specification

DOM Level 0. Not part of specification.

# window.moveTo()

Moves the window to the specified coordinates.

### Syntax

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

### Parameters

`x` is the horizontal coordinate to be moved to.

`y` is the vertical coordinate to be moved to.

### Example

```
function origin() {
   // moves to top left corner of screen
   window.moveTo(0, 0)
}
```

### Notes

This function moves the window absolutely while **window.moveBy()** moves the window relative to its current location.

### Specification

DOM Level 0. Not part of specification.

# window.name

Gets/sets the name of the window.

### Syntax

```
sName = window.name
window.name = sName
```

### Parameters

`name` is a string.

### Example

```
window.name = “lab_view”;
```

### Notes

The **name** of the window is used primarily for setting targets for hyperlinks and forms. Windows do not need to have names.

See also **window.name**.

### Specification

DOM Level 0. Not part of specification.

# window.navigator

Returns a reference to the `navigator` object.

## Syntax

*nav* = window.navigator

## Parameters

*nav* is a `navigator` object reference.

## Example

```
nav = window.navigator;
if ( nav.language != en ) {
   res = window.confirm(lang_warn);
}
```

## Notes

The `navigator` object is used to examine the actual browser being used. It includes properties like `appName`, `appCore`, `plugins` (described below) for getting information about the browser itself.

All of the properties and methods available from `window.navigator` can also be referenced simple with `navigator`.

## Specification

DOM Level 0. Not part of specification.

# window.navigator.appCodeName

Returns the internal “code” name of the current browser.

## Syntax

codeName = window.navigator.appCodeName

### Parameters

`codeName` is the internal name of the browser as a string.

### Example

```
dump(window.navigator.appCodeName);
```

### Notes

Mozilla, Netscape 6, and IE5 all use the internal name “Mozilla.”

### Specification

DOM Level 0. Not part of specification.

# window.navigator.appName

Returns the official name of the browser.

### Syntax

```
appName = window.navigator.appName
```

### Parameters

`appName` is the name of the browser as a string.

### Example

```
dump(window.navigator.appName);
// prints “Navigator” to the console for NS6
```

### Notes

None.

### Specification

DOM Level 0. Not part of specification.

# window.navigator.appVersion

Returns the version of the browser as a string.

### Syntax

*ver* = window.navigator.appVersion

### Parameters

ver is the version number of the browser as a string.

### Example

```
if ( navigator.appVersion.charAt(0) == “5” ) {
    // browser is putatively a v5 browser
}
```

### Notes

The **window.navigator.userAgent** property also contains the version number (example: “Mozilla/5.0 (Windows; U; Win98; en-US; rv:0.9.2) Gecko/20010725 Netscape 6/6.1”), but you should be aware of how easy it is to change the user agent string and “spoof” other browsers, platforms, or user agents, and also how cavalier the browser vendor themselves are with these properties.

The **window.navigator.appVersion** and **window.navigator.userAgent** properties are quite often used in “browser sniffing” code: scripts that attempt to find out what kind of browser you are using and adjust pages accordingly.

### Specification

DOM Level 0. Not part of specification.

# window.navigator.cookieEnabled

Returns a boolean value indicating whether cookies are enabled or not.

### Syntax

```
res = window.navigator.cookieEnabled
```

### Parameters

*res* is a boolean True or False.

### Example

```
if (window.navigator.cookieEnabled) {
  // set a cookie
}
```

### Notes

None.

### Specification

DOM Level 0. Not part of specification.

# window.navigator.javaEnabled()

This method indicates whether the current browser is Java-enabled or not.

### Syntax

```
bool = window.navigator.javaEnabled
```

### Parameters

`bool` is a boolean value.

### Example

```
if ( window.navigator.javaEnabled() ) {
   // browser has java
}
```

### Notes

The return value for this method indicates whether the preference that controls Java is on or off—not whether the browser offers Java support in general.

### Specification

DOM Level 0. Not part of specification.

# window.navigator.language

Returns a string representing the language version of the browser.

### Syntax

*lang* = window.navigator.language

### Parameters

`lang` is a two character string (e.g., “en” or “ja”) representing the language version.

### Example

```
if ( window.navigator.language != “en” ) {
    doLangSelect(window.navigator.language);
}
```

### Notes

This property also shows up as part of the **window.navigator.userAgent** string.

### Specification

DOM Level 0. Not part of specification.

# window.navigator.mimeTypes

Returns a list of the MIME types supported by the browser.

### Syntax

```
syntax code
```

### Parameters

blah is a blah.

### Example

```
// example code here
```

### Notes

Addtional notes.

### Specification

DOM Level 0. Not part of specification.

# window.navigator.oscpu

Returns a string that identifies the current operating system.

### Syntax

```
oscpuInfo = window.navigator.oscpu
```

### Parameters

oscpu is a string that takes the following form.

### Example

```
function osInfo() {
  alert(window.navigator.oscpu);
}
// returns: Win98
```

### Notes

None.

### Specification

DOM Level 0. Not part of specification.

# window.navigator.platform

Returns a string representing the platform of the browser.

### Syntax

```
plat = window.navigator.platform
```

### Parameters

`plat` is a string with one of the following values:

| | |
|---|---|
| Win95 | Windows 95 |
| WinNT | Windows NT |
| MacPPC | Macintosh PowerPC |
| SunOS | Solaris |
| .... | |

### Example

```
function osInfo() {
  alert(window.navigator.platform);
}
// returns: win32
```

### Notes

None.

### Specification

DOM Level 0. Not part of specification.

# window.navigator.plugins

Returns an array of the plugins installed in the browser.

### Syntax

*plugins* = window.navigator.plugins

### Parameters

*plugins* is an array of plugin objects.

### Example

```
function pluginInfo() {
  alert(window.navigator.plugins.item(0).name);
}
// returns “Shockwave for Director”
```

### Notes

The plugin objecct exposes a small interface for getting information about the various plugins installed in your browser.

### Specification

DOM Level 0. Not part of specification.

# window.navigator.product

This property returns the product name of the current browser.

### Syntax

*productName* = window.navigator.product

### Parameters

*productName* is a string.

### Example

```
<script>
function prod() {
  dt = document.getElementById("d").childNodes[0];
  dt.data = window.navigator.userAgent;
}
</script>
<button onclick="prod();">product</button>
<div id="d"> </div>
// returns “Gecko”
```

### Notes

**product** is that portion of the full user agent string that comes directly after the platform. In the user agent for Netscape 6.1, for example, the product is “Gecko” and the full agent string is the following:

```
Mozilla/5.0 (Windows; U; Win98; en-US; rv:0.9.2) Gecko/
20010725 Netscape6/6.1
```

### Specification

DOM Level 0. Not part of specification.

# window.navigator.productSub

**productSub** returns the build number of the current browser.

### Syntax

```
prodSub = window.navigator.productSub
```

### Parameters

*prodSub* is a string.

### Example

```
<script>
function prodsub() {
  dt = document.getElementById("d").childNodes[0];
  dt.data = window.navigator.productSub;
}

</script>

<button onclick="prodsub();">productSub</button>
// returns: 20010725
```

### Notes

On IE, this property returns `undefined`.

### Specification

`DOM Level 0. Not part of specification.`

# window.navigator.userAgent

Returns the user agent string for the current browser.

### Syntax

```
uaString = window.navigator.userAgent
```

### Parameters

*`uaString`* is a string.

### Example

```
window.navigator.userAgent
// returns Mozilla/5.0 (Windows; U; Win98; en-US;
   rv:0.9.2) Gecko/20010725 Netscape6/6.1
```

### Notes

None.

### Specification

DOM Level 0. Not part of specification.

# window.navigator.vendor

Returns the name of the browser vendor for the current browser.

### Syntax

*venString* = window.navigator.vendor

### Parameters

*venString* is a string.

### Example

```
window.navigator.vendor
// returns “Netscape6”
```

### Notes

vendor is another portion of the **userAgent** string. The product and the vendor can be different--as when Netscape 6.1 uses the Gecko product to do its rendering.

See also **window.navigator.product**, **window.navigator.userAgent**

### Specification

DOM Level 0. Not part of specification.

# window.navigator.vendorSub

vendorSub is the substring of the vendor having to do with the vendor version number.

### Syntax

*venSub* = window.navigator.vendorSub

### Parameters

*venSub* is a string.

### Example

```
window.navigator.vendorSub
// returns “6.1” where the vendor part of userAgent is
// Netscape6/6.1
```

### Notes

vendorSub is yet another component of the full user agent string. It refers to the version number that the vendor themselves have given the current browser (as opposed to the version of the product, which may be different). In Netscape 6.1, the **productSub** is given as “5.0” and the **vendorSub** is “6.1.”

See also **window.navigator.productSub**, **window.navigator.userAgent**, **window.navigator.vendor**

### Specification

DOM Level 0. Not part of specification.

# window.onabort

An event handler for abort events sent to the window.

## Syntax

```
window.onabort = funcRef
```

## Parameters

`funcRef` is a reference to a function.

## Example

```
window.onabort = resetThatServerThing
```

## Notes

None.

## Specification

DOM Level 0. Not part of specification.

# window.onblur

The **onblur** property returns the onBlur event handler code, if any, that exists on the window.

## Syntax

```
window.onblur = funcRef
```

## Parameters

`funcRef` is a reference to the function to be executed.

### Example

```
warnFunc = window.onblur;
```

### Notes

The blur event is raised when a window loses focus.

### Specification

Not part of specification.

# window.onchange

An event handler for change events sent to the window.

### Syntax

`window.onchange = *funcRef*`

### Parameters

`funcRef` is a reference to a function.

### Example

```
window.onchange = resetThatServerThing
```

### Notes

None.

### Specification

`DOM Level 0. Not part of specification.`

# window.onclick

An event handler for click events sent to the window.

### Syntax

```
window.onclick = funcRef
```

### Parameters

`funcRef` is a reference to a function.

### Example

```
window.onclick = doPopup;
```

### Notes

The click event is raised when the user clicks on the window.

### Specification

DOM Level 0. Not part of specification.

# window.onclose

An event handler for close events sent to the window.

### Syntax

```
window.onclose = funcRef
```

### Parameters

`funcRef` is a reference to a function.

### Example

```
window.onclose = resetThatServerThing
```

### Notes

None.

### Specification

DOM Level 0. Not part of specification.

# window.ondragdrop

An event handler for drag & drop events sent to the window.

### Syntax

```
window.ondragdrop = funcRef
```

### Parameters

`funcRef` is a reference to a function.

### Example

```
window.ondragdrop = examineItem;
```

### Notes

None.

### Specification

DOM Level 0. Not part of specification.

# window.onerror

An event handler for error events sent to the window.

### Syntax

```
window.onerror = funcRef
```

### Parameters

`funcRef` is a reference to a function.

### Example

```
window.onerror = null;
```

### Notes

The error event is raised when an error occurs in the script. The example above prevents error dialogs from displaying—which is the window's normal behavior—by overriding the default event handler for error events that go to the window.

### Specification

DOM Level 0. Not part of specification.

# window.onfocus

An event handler for focus events sent to the window.

### Syntax

```
window.onfocus = funcRef
```

### Parameters

`funcRef` is a reference to a function.

### Example

```
window.onfocus = startTimer;
```

### Notes

The focus event is raised when the user sets focus on the current window.

### Specification

DOM Level 0. Not part of specification.

# window.onkeydown

An event handler for the keydown event on the window.

### Syntax

```
window.onkeydown = funcRef
```

### Parameters

`funcRef` is a reference to a function.

### Example

```
window.onkeydown = doFunc;
```

### Notes

The keydown event is raised when the user presses any key.

### Specification

DOM Level 0. Not part of specification.

# window.onkeypress

An event handler for the keypress event on the window.

### Syntax

`window.onkeypress = funcRef`

### Parameters

`funcRef` is a reference to a function.

### Example

```
window.onkeypress = doFunc;
```

### Notes

The keypress event is raised when the user presses and releases any key on the keyboard.

### Specification

`DOM Level 0. Not part of specification.`

# window.onkeyup

An event handler for the keyup event on the window.

### Syntax

`window.onkeyup = funcRef`

### Parameters

`funcRef` is a reference to a function.

### Example

```
window.onkeyup = doFunc;
```

### Notes

The keyup event is raised when a key that has been pressed is released.

### Specification

DOM Level 0. Not part of specification.

# window.onload

An event handler for the load event on the window.

### Syntax

```
window.onload = funcRef
```

### Parameters

`funcRef` is a reference to a function.

### Example

```
window.onload = init;
```

### Notes

The load event is fired at the end of the document loading process. At this point, all of the objects in the document are in the DOM.

### Specification

DOM Level 0. Not part of specification.

# window.onmousedown

An event handler for the mousedown event on the window.

### Syntax

```
window.onmousedown = funcRef
```

### Parameters

`funcRef` is a reference to a function.

### Example

```
window.onmousedown = doFunc;
```

### Notes

The mousedown event is raised when the user clicks the left mouse button anywhere in the document.

### Specification

DOM Level 0. Not part of specification.

# window.onmousemove

An event handler for the mousemove event on the window.

### Syntax

```
window.onmousemove = funcRef
```

### Parameters

`funcRef` is a reference to a function.

### Example

```
window.onmousemove = doFunc;
```

### Notes

The mousemove event is raised when the user moves the mouse at all.

### Specification

DOM Level 0. Not part of specification.

# window.onmouseout

An event handler for the mousedown event on the window.

## Syntax

```
window.onmouseout = funcRef
```

## Parameters

`funcRef` is a reference to a function.

## Example

```
window.onmouseout = doFunc;
```

## Notes

The mouseout event is raised when the mouse leaves the area of the specified element (in this case the window itself).

## Specification

DOM Level 0. Not part of specification.

# window.onmouseover

An event handler for the mouseover event on the window.

## Syntax

```
window.onmouseover = funcRef
```

## Parameters

`funcRef` is a reference to a function.

### Example

```
window.onmouseover = doFunc;
```

### Notes

The mouseover event is raised when the moves over the current element (in this case the window itself).

### Specification

DOM Level 0. Not part of specification.

# window.onmouseup

An event handler for the mousedown event on the window.

### Syntax

`window.onmousedown = *funcRef*`

### Parameters

`funcRef` is a reference to a function.

### Example

```
window.onmousedown = doFunc;
```

### Notes

The mousedown event is raised when the user clicks the left mouse button anywhere in the document.

### Specification

DOM Level 0. Not part of specification.

# window.onpaint

An event handler for the paint event on the window.

## Syntax

```
window.onpaint = funcRef
```

## Parameters

`funcRef` is a reference to a function.

## Example

```
window.onpaint = doFunc;
```

## Notes

The paint event is raised when the window is rendered. This event occurs after the load event for a window, and reoccurs each time the window needs to be rerendered, as when another window obscures it and is then cleared away.

## Specification

DOM Level 0. Not part of specification.

# window.onreset

An event handler for the reset event on the window.

## Syntax

```
window.onreset = funcRef
```

## Parameters

`funcRef` is a reference to a function.

### Example

```
<html>
<script>
function reg() {
  window.captureEvents(Event.RESET);
  window.onreset = hit;
}

function hit() {
  alert('hit');
}
</script>

<body onload="reg();">
<form>
  <input type="reset" value="reset" />
</form>
<div id="d"> </div>
</body>
</html>
```

### Notes

The reset event is raised when the user clicks a reset button in a form (`<input type=”reset”/>`).

### Specification

`DOM Level 0. Not part of specification.`

# window.onresize

An event handler for the resize event on the window.

### Syntax

```
window.onresize = funcRef
```

### Parameters

`funcRef` is a reference to a function.

### Example

```
window.onresize = doFunc;
```

### Notes

The resize event is fired when the window is resized.

### Specification

```
DOM Level 0. Not part of specification.
```

# window.onscroll

An event handler for the scroll event on the window.

### Syntax

```
window.onscroll = funcRef
```

### Parameters

`funcRef` is a reference to a function.

### Example

```
window.onscroll = doFunc;
```

### Notes

The scroll event is raised when the window is scrolled.

### Specification

DOM Level 0. Not part of specification.

# window.onselect

An event handler for the select event on the window.

### Syntax

```
window.onselect = funcRef
```

### Parameters

`funcRef` is a reference to a function.

### Example

```
window.onselect = textToCell;
```

### Notes

The select event is raised when text in the window is selected.

### Specification

DOM Level 0. Not part of specification.

# window.onsubmit

An event handler for the submit event on the window.

### Syntax

```
window.onsubmit = funcRef
```

## Parameters

`funcRef` is a reference to a function.

## Example

```
<html>
<script>
function reg() {
  window.captureEvents(Event.SUBMIT);
  window.onsubmit = hit;
}

function hit() {
  alert('hit');
}
</script>

<body onload="reg();">
<form>
  <input type="submit" value="submit" />
</form>
<div id="d"> </div>
</body>
</html>
```

## Notes

The submit event is raised when the user clicks a submit button in a form (`<input type=”submit”/>`).

## Specification

`DOM Level 0. Not part of specification.`

# window.onunload

An event handler for the unload event on the window.

## Syntax

```
window.onunload = funcRef
```

## Parameters

`funcRef` is a reference to a function.

## Example

```
window.onunload = saveStuff;
```

## Notes

The unload event is raised as one document is unloaded and another is about to be loaded into the browser.

## Specification

DOM Level 0. Not part of specification.

# window.open()

Opens a new window.

## Syntax

```
window.open("URL", "name" [, "windowfeatures"])
```

### Parameters

URL is a string that points to the window you want to open.

name is a string that names the new window.

windowfeatures is one or more of the following in a comma-separated list:

| | |
|---|---|
| `toolbar` | `Toolbar is present` |
| `location` | `Locationbar is present` |
| `directories` | |
| `status` | |
| `menubar` | `XXX have to update this whole list XXX` |
| `scrollbars` | |
| `resizable` | |
| `copyhistory` | |
| `width` | |
| `height` | |

### Example

```
window.open(“btest2.html”, “bwin”, “toolbar,status”);
```

### Notes

The **name** attribute is not a reference or the title of the window. It is used as a target to links and forms.

### Specification

`DOM Level 0. Not part of specification.`

# window.opener

Returns a reference to the window that opened this current window.

### Syntax

```
wObj = window.opener
```

### Parameters

wObj is an object reference.

### Example

```
if window.opener != indexWin {
    referToTop(window.opener);
}
```

### Notes

When a window is opened from another window, it maintains a reference to that first window as `window.opener`. If the current window has no opener, this method returns `NULL`.

### Specification

DOM Level 0. Not part of specification.

# window.outerHeight

Gets/sets the height of the outside of the browser window.

### Syntax

```
window.outerHeight = iPx
iPx = window.outerHeight
```

### Parameters

`iPx` is an integer representing the number of pixels.

### Example

```
window.outerHeight = ( window.screen.availHeight );
```

### Notes

As the snippet above demonstrates, the **outerHeight** property is very often used to size the browser to the available screen area. Contrast this with the **innerHeight** property, which controls the size of the content area of the browser.

See also **window.screen**, **window.innerHeight**, **window.outerWidth**

### Specification

DOM Level 0. Not part of specification.

# window.outerWidth

Gets/sets the width of the outside of the browser window.

### Syntax

```
window.outerWidth = iPx
iPx = window.outerWidth
```

### Parameters

`iPx` is an integer representing the number of pixels.

### Example

```
window.outerWidth = ( window.screen.availWidth );
```

### Notes

As the snippet above demonstrates, the **outerWidth** property is very often used to size the browser to the available screen area. Contrast this with the **innerWidth** property, which controls the size of the content area of the browser.

See also **window.screen**, **window.innerHeight**, **window.outerHeight**

### Specification

DOM Level 0. Not part of specification.

# window.pageXOffset

Gets the amount of content that has been hidden by scrolling to the right.

### Syntax

```
hScroll = window.pageXOffset
```

### Parameters

*hScroll* is the number of pixels scrolled as an integer.

### Example

```
var hScroll = pageXOffset;
var vScroll = pageYOffset;
```

### Notes

If the user has scrolled to the right and 200 pixels of the content is hidden by this, then the **pageXOffset** property returns 200.

**window.pageYOffset**

### Specification

DOM Level 0. Not part of specification.

# window.pageYOffset

Gets the amount of content that has been hidden by scrolling down.

## Syntax

```
vScroll = window.pageYOffset
```

## Parameters

*vScroll* is the number of pixels as an integer.

## Example

```
var hScroll = pageXOffset;
var vScroll = pageYOffset;
```

## Notes

If the user has scrolled down and 400 pixels of the content is hidden by this, then the **pageYOffset** property returns 400.

See also **window.pageXOffset**

## Specification

DOM Level 0. Not part of specification.

# window.parent

Returns a reference to the parent of the current window or subframe.

## Syntax

```
pWin = window.parent
```

## Parameters

pWin is an object reference to the parent window.

### Example

```
if window.parent != window.top
    // we're deeper than one down
```

### Notes

When a window is loaded in a frameset, its **parent** is .

### Specification

DOM Level 0. Not part of specification.

# window.personalbar

Returns the `personalbar` object, whose visibility can be toggled in the window.

### Syntax

```
pBarObj = window.menubar
```

### Parameters

`pBarObj` is an object reference.

### Example

The following complete HTML example shows way that the visible property of the various “bar” objects is used, and also the change to the privileges necessary to write to the visible property of any of the bars on an existing window.

```
<html>
<head>
  <title>Various DOM Tests</title>
  <script>
    // changing bar states on the existing window
    netscape.security.PrivilegeManager.
        enablePrivilege("UniversalBrowserWrite");
    window.personalbar.visible=
        !window.personalbar.visible;
  </script>
</head>
<body>
  <p>Various DOM Tests</p>
</body>
</html>
```

### Notes

To toggle the visibility of these bars, you must either sign your scripts or enable the appropriate privileges, as in the example above. Also be aware that dynamically updating the visibilty of the various toolbars can change the size of the window rather dramatically, and may affect the layout of your page.

See also: **window.locationbar**, **window.menubar**, **window.personalbar**, **window.scrollbars**, **window.statusbar**, **window.toolbar**

### Specification

`DOM Level 0. Not part of specification.`

# window.pkcs11

Returns the `pkcs11` object , which can be used to install drivers other software associated with the pkcs11 protocol.

### Syntax

*pkcsObj* = window.pkcs11

### Parameters

*pkcsObj* is an object reference.

### Example

```
window.pkcs11.addModule(sMod, secPath, 0, 0);
```

### Notes

See the *nsIDOMPkcs11.idl* in the mozilla source for more information about how to manipulate pkcs11 objects.

### Specification

DOM Level 0. Not part of specification.

# window.print()

Prints the current document.

### Syntax

window.print()

### Parameters

None.

### Example

```
window.print();
```

### Notes

None.

### Specification

DOM Level 0. Not part of specification.

# window.prompt()

Returns the text entered by the user in a prompt dialog.

### Syntax

syntax code

### Parameters

blah is a blah.

### Example

```
function pr() {
   sign = prompt("What's your sign?");
}
```


### Notes

See also **window.alert()**, **window.confirm()**.

### Specification

DOM Level 0. Not part of specification.

# window.prompter

Returns a reference to the prompt window, if any, currently displayed.

### Syntax

```
prompt = window.prompter
```

### Parameters

*prompt* is an object reference to the prompt window.

### Example

```
prompt_window = window.prompter
```

### Notes

None.

### Specification

DOM Level 0. Not part of specification.

# window.releaseEvents()

Releases the window from trapping events of a specific type.

### Syntax

```
window.releaseEvents(Event.eventType)
```

## Parameters

`eventType` is a string with one of the following values:

| | |
|---|---|
| `Abort` | `Load` |
| `Blur` | `MouseDown` |
| `Click` | `MouseMove` |
| `Change` | `MouseOut` |
| `DblClick` | `MouseOver` |
| `DragDrop` | `MouseUp` |
| `Error` | `Move` |
| `Focus` | `Reset` |
| `KeyDown` | `Resize` |
| `KeyPress` | `Select` |
| `KeyUp` | `Submit` |
| | `Unload` |

## Example

```
window.releaseEvents(Event.KEYPRESS)
```

## Notes

Note that you can pass a list of events to this method using the following syntax: `window.releaseEvents(Event.KEYPRESS | Event.KEYDOWN | Event.KEYUP)`. Also note that the **eventType** parameter is case-insensitive, so you can also say, for example, `window.releaseEvents(Event.KeyPress)`.

See also **window.captureEvents()**.

## Specification

`DOM Level 0. Not part of specification.`

# window.resizeBy()

Resizes the current window by a certain amount.

## Syntax

`window.resizeBy(xDelta, yDelta)`

## Parameters

*`xDelta`* is the number of pixels to grow the window horizontally.

*`yDelta`* is the number of pixels to grow the window vertically.

## Example

```
// shrink the window
window.resizeBy(-200, -200);
```

## Notes

This method resizes the window relative to its current size. To resize the window in absolute terms, use **window.resizeTo()**.

## Specification

DOM Level 0. Not part of specification.

# window.resizeTo()

Dynamically resizes window.

## Syntax

`window.resizeTo(iWidth, iHeight)`

### Parameters

`iWidth` is an integer representing the new width in pixels.

`iHeight` is an integer value representing the new height in pixels.

### Example

```
// function resizes the window to take up half
// of the available screen.
function halve() {
  window.resizeTo(window.screen.availWidth/2,
     window.screen.availHeight/2);
}
```

### Notes

See also **window.resizeBy()**.

### Specification

`DOM Level 0. Not part of specification.`

# window.screen

Returns a reference to the `screen` object associated with the window.

### Syntax

*`screenObj`* `= window.screen`

### Parameters

`screenObj` is an object reference.

### Example

```
s = window.screen;
if ( s.colorDepth < 8) {
    // use low-color version of page
} else {
    // use regular, colorful page
}
```

### Notes

The `screen` object is a special JavaScript object for controlling aspects of the screen on which the current window is being rendered. `screen` object properties such as `colorDepth`, `height`, and `availHeight` can be accessed from the `window` object by using properties like `window.screen.colorDepth` and others described below.

### Specification

DOM Level 0. Not part of specification.

# window.screen.availHeight

Returns the amount of vertical space available to the window on the screen.

### Syntax

*iAvail* = window.screen.availHeight

### Parameters

`iAvail` is an integer number representing the amount of space in pixels.

### Example

```
if window.screen.availHeight != window.screen.height {
    // something's in the way!
    // use available to size window
}
```

### Notes

Addtional notes.

### Specification

DOM Level 0. Not part of specification.

# window.screen.availLeft

Returns the first available pixel available from the left side of the screen.

### Syntax

*iAvail* = window.screen.availLeft

### Parameters

*iAvail* is an integer representing the amount of space in pixels.

### Example

```
setY = window.screen.height - window.screen.availTop;
setX = window.screen.width - window.screen.availLeft;
window.moveTo(setX, setY);
```

### Notes

In most cases, this property returns 0.

### Specification

DOM Level 0. Not part of specification.

# window.screen.availTop

Specifies the y-coordinate of the first pixel that is not allocated to permanent or semipermanent user interface features.

## Syntax

```
iAvail = window.screen.availTop
```

## Parameters

`iAvail` is an integer representing the amount of space in pixels.

## Example

```
setY = window.screen.height - window.screen.availTop;
setX = window.screen.width - window.screen.availLeft;
window.moveTo(setX, setY);
```

## Notes

In most cases, this property returns 0.

## Specification

DOM Level 0. Not part of specification.

# window.screen.availWidth

Returns the amount of horizontal space in pixels available to the window.

## Syntax

```
iAvail = window.screen.availWidth
```

## Parameters

`iAvail` is an integer representing the amount of space in pixels.

### Example

```
// example code here
```

### Notes

Addtional notes.

### Specification

DOM Level 0. Not part of specification.

# window.screen.colorDepth

Returns the color depth of the screen.

### Syntax

```
bitDepth = window.screen.colorDepth
```

### Parameters

bitDepth is an integer representing the color depth in bits.

### Example

```
// check the color depth of the screen
if ( window.screen.colorDepth < 8) {
    // use low-color version of page
} else {
    // use regular, colorful page
}
```

### Notes

See also **window.screen.pixelDepth**.

### Specification

DOM Level 0. Not part of specification.

# window.screen.height

Returns the height of the screen in pixels.

### Syntax

*iHeight* = window.screen.height

### Parameters

iHeight is an integer representing the height in pixels.

### Example

```
if (window.screen.availHeight != window.screen.height)
{
   // something is occupying some screen real estate!
}
```

### Notes

Note that not all of the height given by this property may be available to the window itself. Widgets such as taskbars or other special application windows that integrate with the OS (e.g., the Spinner player minimized to act like an additional toolbar on windows) may reduce the amount of space available to browser windows and other applications.

### Specification

DOM Level 0. Not part of specification.

# window.screen.left

Gets/sets the current distance in pixels from the left side of the screen.

### Syntax

```
lLeft = window.screen.left
window.screen.left = lLeft
```

### Parameters

lLeft is the number of pixels from the left side of the screen.

### Example

```
// move and resize the current window
window.resizeTo(window.screen.availWidth/2);
window.screen.left = 1;
```

### Notes

See also `window.screen.top`.

### Specification

`DOM Level 0. Not part of specification.`

# window.screen.pixelDepth

Returns the bit depth of the screen.

### Syntax

*`depth`* `= window.screen.pixelDepth`

### Parameters

*`depth`* is the number of bits per pixel as an integer.

### Example

```
// if there is not adequate bit depth
// choose a simpler color
if ( window.screen.pixelDepth > 8 ) {
   document.style.color = “#FAEBD7”;
} else {
   document.style.color = “#FFFFFF”;
}
```

### Notes

See also **window.screen.colorDepth**.

### Specification

DOM Level 0. Not part of specification.

# window.screen.top

Gets/sets the distance from the top of the screen.

### Syntax

```
lTop = window.screen.top
window.screen.top = lTop
```

### Parameters

lTop is the number of pixels from the top of the screen.

### Example

```
// move and resize the current window,
// making it like a bar across the top
window.resizeTo( window.screen.availHeight/4 );
window.screen.top = 0;
```

### Notes

See also `window.screen.left`.

### Specification

`DOM Level 0. Not part of specification.`

# window.screen.width

Returns the width of the screen.

### Syntax

```
lWidth = window.screen.width
```

### Parameters

lWidth is the width of the screen in pixels.

### Example

```
// crude way to check that the screen is at 1024x768
if (window.screen.width > 1000) {
    // resolution is below 10 x 7
}
```

### Notes

Note that not all of the width given by this property may be available to the window itself. When other widgets occupy space that cannot be used by the window object, there is a difference in `window.screen.width` and `window.screen.availWidth`. See also `window.screen.height`.

### Specification

`DOM Level 0. Not part of specification.`

# window.screenX

Returns the horizontal distance of the left border of the user's browser from the left side of the screen.

## Syntax

```
lLoc = window.screenX
```

## Parameters

`lLoc` is the number of pixels from the left side the screen.

## Example

```
None.
```

## Notes

See also **window.screenY**.

## Specification

DOM Level 0. Not part of specification.

# window.screenY

Returns the vertical distance of the top border of the user's browser from the top side of the screen.

## Syntax

```
lLoc = window.screenY
```

## Parameters

`lLoc` is the number of pixels from the top of the screen.

### Example

```
None.
```

### Notes

See also **window.screenX**.

### Specification

`DOM Level 0. Not part of specification.`

# window.scrollbars

Returns the `scrollbars` object, whose visibility can be toggled in the window.

### Syntax

*`sBarObj`* `= window.scrollbars`

### Parameters

`sBarObj` is an object reference.

### Example

The following complete HTML example shows way that the visible property of the various “bar” objects is used, and also the change to the privileges necessary to write to the visible property of any of the bars on an existing window.

```
<html>
<head>
  <title>Various DOM Tests</title>
  <script>
    // changing bar states on the existing window
    netscape.security.PrivilegeManager.
        enablePrivilege("UniversalBrowserWrite");
    window.menubar.visible=!window.menubar.visible;
  </script>
</head>
<body>
  <p>Various DOM Tests</p>
</body>
</html>
```

## Notes

Note that `scrollbars` is not an array of the scrollbars. The visibility of these objects can only be controlled as a group.

To toggle the visibility of these bars, you must either sign your scripts or enable the appropriate privileges, as in the example above. Also be aware that dynamically updating the visibilty of the various toolbars can change the size of the window rather dramatically, and may affect the layout of your page.

See also: **window.locationbar**, **window.menubar**, **window.personalbar**, **window.scrollbars**, **window.statusbar**, **window.toolbar**

## Specification

`DOM Level 0. Not part of specification.`

# window.scroll()

Scrolls the window to a particular place in the document.

## Syntax

`window.scroll(`*`x-coord`*`, `*`y-coord`*`)`

## Parameters

*`x-coord`* is the pixel along the horizontal axis of the document that you want displayed in the upper left.

*`y-coord`* is the pixel along the vertical axis of the document that you want displayed in the upper left.

## Example

```
// put the 1000th vertical pixel at
// the top of the window
<INPUT TYPE=”button” VALUE=”1000”
  onClick=”scroll(0, 1000);”/>
```

## Notes

**window.scrollTo()** is effectively the same as this method.

For scrolling a particular distance repeatedly, use the **window.scrollBy()**. Also see **window.scrollByLines()**, **window.scrollByPages()**.

## Specification

`DOM Level 0. Not part of specification.`

# window.scrollBy()

Scrolls the document in the window by the given amount.

### Syntax

`window.scrollBy(xDelta, yDelta)`

### Parameters

*xDelta* is the amount of pixels to scroll horizontally.

*yDelta* is the amount of pixels to scroll vertically.

### Example

```
// scroll one page
window.scrollBy(0, window.innerHeight);
```

### Notes

**window.scrollBy()** scrolls by a particular amount where **window.scroll()** scrolls to an absolute position in the document.

See also **window.scrollByLines()**, **window.scrollByPages()**

### Specification

DOM Level 0. Not part of specification.

# window.scrollByLines()

Scrolls the document by the given number of lines.

### Syntax

`window.scrollByLines(lines)`

### Parameters

*lines* is the number of lines.

### Example

```
<button onclick="scrollByLines(10);">jump</button>
```

### Notes

See also **window.scrollBy()**, **window.scrollByPages()**.

### Specification

DOM Level 0. Not part of specification.

# window.scrollByPages()

Scrolls the current document by the specified number of pages.

### Syntax

`window.scrollByPages(`*`pages`*`)`

### Parameters

*pages* is the number of pages to scroll.

### Example

```
// scroll one page
window.scrollByPages(1);
```

### Notes

See also **window.scrollBy()**, **window.scrollByLines()**, **window.scroll()**, **window.scrollTo()**.

### Specification

DOM Level 0. Not part of specification.

# window.scrollTo()

Scrolls to a particular set of coordinates in the document.

## Syntax

```
window.scrollTo(x-coord, y-coord)
```

## Parameters

*x-coord* is the pixel along the horizontal axis of the document that you want displayed in the upper left.

*y-coord* is the pixel along the vertical axis of the document that you want displayed in the upper left.

## Example

```
window.scrollTo(0, 1000);
```

## Notes

This function is effectively the same as **window.scroll()**. For relative scrolling, **window.scrollBy()**, **window.scrollByLines()**, and **window.scrollByPages()**.

## Specification

DOM Level 0. Not part of specification.

# window.scrollX

Returns the number of pixels that the document has already been scrolled horizontally.

## Syntax

```
xpix = window.scrollX
```

### Parameters

*xpix* is the number of pixels.

### Example

```
// make sure and go over to the second horizontal page
if (window.scrollX) {
   scroll(0,0);
}
scrollBy(400, 0);
```

### Notes

Use this property to check that the document hasn't already been scrolled some if you are using relative scroll functions such as **window.scrollBy()**, **window.scrollByLines()**, or **window.scrollByPages()**.

### Specification

DOM Level 0. Not part of specification.

# window.scrollY

Returns the number of pixels that the document has already been scrolled vertically.

### Syntax

*ypix* = window.scrollY

### Parameters

*ypix* is the number of pixels.

### Example

```
// make sure and go down to the second page
if (window.scrollY) {
   scroll(0,0);
}
scrollByPages(1);
```

### Notes

Use this property to check that the document hasn't already been scrolled some if you are using relative scroll functions such as **window.scrollBy()**, **window.scrollByLines()**, or **window.scrollByPages()**

### Specification

DOM Level 0. Not part of specification.

# window.self

Returns an object reference to the `window` object.

### Syntax

*`selfObj`* `= window.self`

### Parameters

`selfObj` is an object reference.

### Example

```
if (window.parent.frames[0] != window.self) {
   // this window is not the first frame in the list
}
```

### Notes

`window.self` is almost always used in comparisons like in the example above, which finds out if the current window is the first subframe in the parent frameset.

### Specification

`DOM Level 0. Not part of specification.`

# window.setCursor()

Changes the cursor for the current window.

### Syntax

`syntax code`

### Parameters

blah is a blah.

### Example

```
function SetBusyCursor(window, enable)
{
  if(enable)
    window.setCursor("wait");
  else
    window.setCursor("auto");

  var numFrames = window.frames.length;
  for(var i = 0; i < numFrames; i++)
  SetBusyCursor(window.frames[i], enable);
}
```

### Notes

The cursor is locked until it's set back to auto.

### Specification

DOM Level 0. Not part of specification.

# window.setInterval()

Set a delay for a specific function.

### Syntax

```
ID = window.setInterval("funcName", delay)
```

### Parameters

**funcName** is the name of the function for which you want to set a delay.

**delay** is the number of milliseconds (thousandths of a second) that the function should be delayed.

**ID** is the interval ID.

### Example

```
intervalID = window.setInterval("animalate()", 500);
```

### Notes

The interval ID is used to refer to the specific interval when it needs to be cleared. The **setInterval()** function is commonly used to set a delay for functions that are executed again and again, such as animations.

See also **window.clearInterval()**.

### Specification

DOM Level 0. Not part of specification.

# window.setTimeout()

Sets a delay for executing a function.

## Syntax

```
ID = window.setTimeout("funcName", delay)
```

## Parameters

**funcName** is the name of the function for which you want to set a delay.

**delay** is the number of milliseconds (thousandths of a second) that the function should be delayed.

**ID** is the interval ID.

## Example

```
setTimeout('parent.generateOutput("Cancel")',0);
```

## Notes

The setTimeout() method is often used to establish a time limit on certain applications, as when a user is logged out or certain information is reset if there has not been any interaction within the given time.

See also **window.clearTimeout()**

## Specification

DOM Level 0. Not part of specification.

# window.sidebar

Returns a reference to the window object of the sidebar.

### Syntax

*sidebar* = window.sidebar

### Parameters

*sidebar* is a window object.

### Example

```
sbar = window.sidebar;
if (sbar) {
  sbar_content = sbar._content;
}
```

### Notes

The sidebar is a subframe in the DOM of the application window. Its content can be accessed with `sidebar._content`, as in the foregoing example, and it is a sibling of the window's main content frame.

### Specification

DOM Level 0. Not part of specification.

# window.sizeToContent()

Sizes the window according to its content.

### Syntax

window.sizeToContent()

### Parameters

None.

### Example

```
window.sizeToContent();
```

### Notes

None.

### Specification

DOM Level 0. Not part of specification.

# window.status

Gets/sets the text in the statusbar at the bottom of the browser.

### Syntax

```
msg = window.status
window.status = msg
```

### Parameters

msg is a string containing the text to appear in the statusbar.

### Example

```
while ( bigLoad ) {
   window.status = “Loading...”;
}
```

### Notes

None.

### Specification

DOM Level 0. Not part of specification.

# window.statusbar

Returns the `statusbar` object, whose visibility can be toggled in the window.

## Syntax

```
sBarObj = window.menubar
```

## Parameters

sBarObj is an object reference.

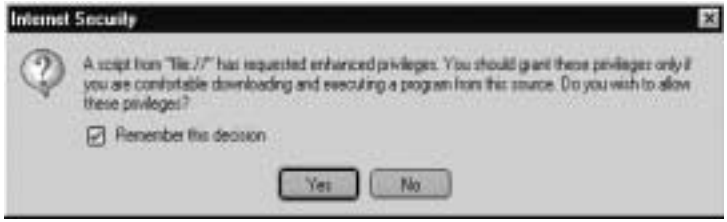
## Example

The following complete HTML example shows way that the visible property of the various “bar” objects is used, and also the change to the privileges necessary to write to the visible property of any of the bars on an existing window.

```
<html>
<head>
  <title>Various DOM Tests</title>
  <script>
    // changing bar states on the existing window
    netscape.security.PrivilegeManager.
        enablePrivilege("UniversalBrowserWrite");
    window.statusbar.visible=!window.statusbar.visible;
  </script>
</head>
<body>
  <p>Various DOM Tests</p>
</body>
</html>
```

## Notes

When you load the example page above, the browser displays the following dialog:




To toggle the visibility of these bars, you must either sign your scripts or enable the appropriate privileges, as in the example above. Also be aware that dynamically updating the visibilty of the various toolbars can change the size of the window rather dramatically, and may affect the layout of your page.

See also: **window.locationbar**, **window.menubar**, **window.personalbar**, **window.scrollbars**, **window.statusbar**, **window.toolbar**

### Specification

`DOM Level 0. Not part of specification.`

# window.stop()

This method stops window loading.

### Syntax

```
window.stop()
```

### Parameters

None.

### Example

```
window.stop();
```

### Notes

The stop() method is exactly equivalent to clicking the stop button in the browser. Because of the order in which scripts are loaded, the stop() method cannot stop the document in which it is contained from loading, but it will stop the loading of large images, new windows, and other objects whose loading is deferred.

### Specification

`DOM Level 0. Not part of specification.`

# window.toolbar

Returns the `toolbar` object, whose visibility can be toggled in the window.

### Syntax

```
tBarObj = window.menubar
```

### Parameters

tBarObj is an object reference.

### Example

The following complete HTML example shows way that the visible property of the various “bar” objects is used, and also the change to the privileges necessary to write to the visible property of any of the bars on an existing window.

```
<html>
<head>
  <title>Various DOM Tests</title>
  <script>
    // changing bar states on the existing window
    netscape.security.PrivilegeManager.
        enablePrivilege("UniversalBrowserWrite");
    window.toolbar.visible=!window.toolbar.visible;
  </script>
</head>
<body>
  <p>Various DOM Tests</p>
</body>
</html>
```

### Notes

To toggle the visibility of these bars, you must either sign your scripts or enable the appropriate privileges, as in the example above. Also be aware that dynamically updating the visibilty of the various toolbars can change the size of the window rather dramatically, and may affect the layout of your page.

See also: **window.locationbar**, **window.menubar**, **window.personalbar**, **window.scrollbars**, **window.statusbar**, **window.toolbar**

### Specification

DOM Level 0. Not part of specification.

# window.top

Returns a reference to the topmost window in the window hierarchy.

### Syntax

*windowObj* = window.top

### Parameters

*windowObj* is an object reference.

### Example

```
None.
```

### Notes

Where the window.parent property returns the immediate parent of the current window, window.top returns the topmost window in the hierarchy of window objects. This property is especially useful when you are dealing with a window that is in a subframe of a parent or parents, and you want to get to the top-level frameset.

### Specification

DOM Level 0. Not part of specification.

# window.unescape()

Unencodes a value that has been encoded in hexadecimal (e.g., a cookie).

### Syntax

window.escape(sValue)

### Parameters

sValue is an encoded string.

### Example

```
cookieValuePlain = unescape( cookieValue );
```

### Notes

See also **window.escape()**.

### Specification

DOM Level 0. Not part of specification.

# window.updateCommands()

Brief description.

### Syntax

```
syntax code
```

### Parameters

blah is a blah.

### Example

```
// example code here
```

### Notes

Addtional notes.

### Specification

DOM Level 0. Not part of specification.

# window.window

Returns a reference to this window.

## Syntax

```
windowObj = window.window
```

## Parameters

*windowObj* is an object reference to the current window.

## Example

```
if ( window.top ) != ( window.window ) {
  ...
}
```

## Notes

This property is redundant. window is itself an object reference that can be used in all cases where window.window can. If for no other reason, it may exist so that comparisons like the example above are more readable.

## Specification

DOM Level 0. Not part of specification.

# DOM *Document* Reference

## The `document` Interface

In the DOM, the `document` object represents the entire HTML or XML document[1]. It is contained by the `window` object (see ***DOM window Reference***) and may contain any number of elements (see ***DOM Element Reference***.)

As you can see from the lists below, the interfaces on `document` deal with such things as the *document type*, features of the document such as its color and formatting, the *plugins* and *applets* that are exposed to the user in the document, as well as methods for creating all of the document's child nodes, or elements that typically live in the structural representation of the whole document, such as the `<BODY>` element, a `<TABLE>` and so forth.

### Properties

| | |
|---|---|
| **attributes** | Returns an array of attributes on the given element. |
| **alinkColor** | Returns or sets the color of active links in the document body. |
| **anchors** | **anchors** returns a list of all of the anchors in the document. |
| **applets** | **applets** returns an ordered list of the applets within a document. |

1. Strictly speaking, the `document` object represents only HTML and XHTML documents. The XML document is represented by a different document object, `XMLDocument`, and the XUL document by `XULDocument`. Though these objects provide very similar interfaces, they are not quite the same as the `document` object described here.

| | |
|---|---|
| **bgColor** | **bgColor** gets/sets the background color of the current document. |
| **body** | **body** returns the BODY node of the current document. |
| **characterSet** | Returns the character set being used by the document. |
| **childNodes** | Returns an array of child nodes on the given element node. |
| **compatMode** | Indicates whether the document is rendered in Quirks or Strict mode. |
| **cookie** | Returns a semicolon-separated list of the cookies for that document or sets a single cookie. |
| **contentWindow** | Returns the window object for the containing window. |
| **doctype** | Returns the Document Type Definition (DTD) of the current document. |
| **documentElement** | Returns the Element that is a direct child of document, which in most cases is the HTML element. |
| **domain** | **domain** returns the domain of the current document. |
| **embeds** | **embeds** returns a list of the embedded OBJECTS within the current document. |
| **fgColor** | **fgColor** gets/sets the foreground color, or text color, of the current document. |
| **firstChild** | **firstChild** returns the first node in the list of direct children of the document. |
| **forms** | **forms** returns a list of the `FORM` elements within the current document. |
| **height** | **height** gets/sets the height of the current document. |

| | |
|---|---|
| **images** | **images** returns a list of the images in the current document. |
| **implementation** | Returns the DOM implementation associated with the current document. |
| **lastModified** | Returns the date on which the document was last modified. |
| **linkColor** | Gets/sets the color of hyperlinks in the document. |
| **links** | Returns an array of all the hyperlinks in the document. |
| **location** | Returns the URI of the current document. |
| **namespaceURI** | Returns the XML namespace of the current document. |
| **nextSibling** | Returns the node immediately following the current one in the tree. |
| **nodeName** | Returns the name of the current node as a string. |
| **nodeType** | Returns the node type of the current document. |
| **nodeValue** | Returns the value of particular types of nodes. |
| **ownerDocument** | Returns an object reference to the document that owns the current element. |
| **parentNode** | Returns an object reference to the parent node. |
| **plugins** | Returns an array of the available plugins. |
| **previousSibling** | Returns the node immediately previous to the current one in the tree. |
| **referrer** | Returns the URI of the page that linked to this page. |

| | |
|---|---|
| **styleSheets** | The **stylesheets** property returns a list of the `stylesheet` objects on the current document. |
| **title** | Returns the title of the current document. |
| **URL** | Returns a string containing the URL of the current document. |
| **vlinkColor** | Gets/sets the color of visited hyperlinks. |
| **width** | Returns the width of the current document. |

### Methods

| | |
|---|---|
| **clear** | Clears a document. |
| **close** | Closes a document stream for writing. |
| **createAttribute** | Create a new attribute on the current element. |
| **createDocumentFragment** | Creates a new document fragment. |
| **createElement** | Creates a new element. |
| **createTextNode** | Creates a text node. |
| **getElementById** | Returns an object reference to the identified element. |
| **getElementsByName** | Returns a list of elements with the given name. |
| **getElementsByTagName** | Returns a list of elements with the given tag name. |
| **open** | Opens a document stream for writing. |
| **write** | Writes text to a document. |
| **writeln** | Write a line of text to a document. |

### Event Handlers

| | |
|---|---|
| **onblur** | Returns the onBlur event handler code, if any, that exists on the current element. |
| **onclick** | Returns the onClick event handler code on the current element. |
| **ondblclick** | Returns the onDblClick event handler code on the current element. |
| **onfocus** | Returns the onFocus event handler code on the current element. |
| **onkeydown** | Returns the onKeyDown event handler code on the current element. |
| **onkeypress** | Returns the onKeyPress event handler code for the current element. |
| **onkeyup** | Returns the onKeyUp event handler code for the current element. |
| **onmousedown** | Returns the onMouseDown event handler code for the current element. |
| **onmousemove** | Returns the onMouseMove event handler code for the current element. |
| **onmouseout** | Returns the onMouseOut event handler code for the current element. |
| **onmouseover** | Returns the onMouseOver event handler code for the current element. |
| **onmouseup** | Returns the onMouseUp event handler code for the current element. |
| **onresize** | Returns the onResize event handler code for the current element. |

# attributes

Returns an array of attributes on the given element.

## Syntax

```
attributes = elementNode.attributes
```

## Parameters

`attributes` is a `namedNodeMap` of attributes on the current element.

## Example

```
// get the first <p> element in the document
para = document.getElementsByTag(“p”)[0];
atts = para.attributes;
```

## Notes

The array returned by this property is a `namedNodeMap`, a list of objects rather than strings. The name and value of the attribute objects are accessible as separate properties, as in the following complete example, which retrieves the name/value pair of the first attribute of the “p1” paragraph in the document:

```
<html>
<head>
<script>
function showA() {
   p = document.getElementById("p1");
   t = document.getElementById("t");
   t.setAttribute("value",
      p.attributes[0].name + "->" +
p.attributes[0].value);
}
</script>
</head>

<p id="p1" style="color: blue;">Sample Paragraph</p>
<form>
<input type="button" value="show" onclick="showA()" />
<input id="t" type="text" value="" />
</form>
</html>
```

### Specification

`attribute`

# alinkColor

Returns or sets the color of active links in the document body.

### Syntax

```
color = HTMLBodyElement.aLinkColor
HTMLBodyElement.aLinkColor = color
```

### Parameters

`color` is a string containing the name of the color (e.g., “blue”, “darkblue”, etc., or the octal value of the color (e.g., `FFFFF`)

### Example

```
example here
```

### Notes

extra information

### Specification

DOM Level 0. *Not part of specification.*

# anchors

**anchors** returns a list of all of the anchors in the document.

### Syntax

```
a_list = document.anchors
```

### Parameters

`a_list` is a `nodeList` of all of the anchor elements within the document

### Example

```
if ( document.anchors.length >= 5 ) {
    dump(“dump found too many anchors”);
    window.location = “http://www.getoutahere.com”;
}
```

### Notes

For reasons of backwards compatibility, the returned set of anchors only contains those anchors created with the `name` attribute, not those created with the `id` attribute.

### Specification

`anchor`

# applets

applets returns an ordered list of the applets within a document.

### Syntax

*`app_list`* `= document.applets`

### Parameters

`app_list` is a `nodeList` of the applets within the document.

### Example

```
// ( When you know the second applet is the one
// you want )
my_java_app = document.applets[1];
```

### Notes

None.

### Specification

`applets`

# bgColor

**bgColor** gets/sets the background color of the current document.

### Syntax

*`color`* `= document.bgColor`
`document.bgColor =` *`color`*

### Parameters

`color` is a string representing the color as a word (e.g., “red’) or as an octal value, as in HTML (e.g., “#eee”)

### Example

```
document.bgColor = “darkblue”;
```

### Notes

None.

### Specification

DOM Level 0. *Not part of specification.*

# body

**body** returns the `BODY` node of the current document.

## Syntax

```
bodyObj = document.body
document.body = aNewBodyElement
```

## Parameters

*`bodyObj`* is a node representing the `BODY` or `FRAMESET` element in the current document.

*`aNewBodyElement`* is a `BODY` or `FRAMESET` element that will replace the current `BODY` or `FRAMESET` element

## Example

```
// in HTML: <body id="oldBodyElement"></body>
alert(document.body.id); // "oldBodyElement"
var aNewBodyElement = document.createElement("body");
aNewBodyElement.id = "newBodyElement";
document.body = aNewBodyElement;
alert(document.body.id); // "newBodyElement"
```

## Notes

`body` is the element that contains the content for the document. In documents with `BODY` contents, returns the `BODY` element, and in frameset documents, this returns the outermost `FRAMESET` element.

Though `body` is settable, setting a new body on a document will effectively remove all the current children of the existing `BODY` element.

## Specification

`body`

# characterSet

Returns the character set of the current document.

## Syntax

```
charSet = document.characterSet
```

## Parameters

`charSet` is a string.

## Example

```
<input
   value="char"
   type="button"
   onclick="alert(document.characterSet);" />
// returns “ISO-8859-1”
```

## Notes

For a complete list of character sets, see:
http://www.iana.org/assignments/character-sets.

## Specification

DOM Level “0”. *Not part of specification.*

# childNodes

Returns an array of child nodes on the given element node.

### Syntax

```
children = elementNode.childNodes
```

### Parameters

`children` is a nodeList of children of the document.

### Example

```
// table is an object reference to a table element
kids = table.childNodes;
for (var i = 0; i < kids.length; i++) {
    // do something with each kid as kids[i]

}
```

### Notes

The `document` object itself has only a single child, and that is the `HTML` element. Note again that the items in the array are objects and not strings. To get data from those objects you must use their properties (e.g. `childNode[2].nodeName` to get the name, etc.)

### Specification

`childNodes`

# cookie

Gets/sets a list of the cookies associated with the current document.

### Syntax

```
cookie_list = document.cookie
document.cookie = cookie_list
```

### Parameters

`cookie_list` is a string containing a semicolon-separated list of cookies

### Example

```
// this function sets two cookies and then
// displays them in an alert
function sgCookie() {
  document.cookie = “name=oeschger”;
  document.cookie = “favorite_food=tripe”;
  alert(document.cookie);
}
// returns: name=oeschger;favorite_food=tripe
```

### Notes

If there are no cookies associated with a document, this function returns an empty string. Note also that you cannot use this property to set more than one cookie at a time.

### Specification

`cookie`

# compatMode

Indicates whether the document is rendered in Quirks mode or Strict mode.

### Syntax

*`mode`* `= document.compatMode`

### Parameters

*`mode`* is a string containing “BackCompat” for Quirks mode or “CSS1Compat” for Strict mode.

### Example

```
if ( document.compatMode == “BackCompat” ){
  // use some quirky stuff
}
```

### Notes

None.

# contentWindow

Returns the containing window of the current document.

### Syntax

*window* = document.contentWindow

### Parameters

*window* is a window object for the window that contains the current document.

### Example

```
// get to the browser
// then load new content
browserWindow = document.contentWindow;
browserWindow.home();
```

### Notes

None.

# doctype

Returns the Document Type Definition (DTD) of the current document.

## Syntax

```
docType = document.DocumentType
```

## Parameters

`docType` is a string representing the DTD, if any, of the current document.

## Example

```
None.
```

## Notes

`DocumentType` attribute is a read-only property. It returns `NULL` if there is no DTD for the current document.

## Specification

`docType`

# documentElement

Returns the `Element` that is a direct child of `document`, which in most cases is the `HTML` element.

## Syntax

```
doc = document.documentElement
```

## Parameters

`doc` is a node representing the direct child of document.

### Example

```
actual_doc = document.documentElement;
first_tier = actual_doc.childNodes;
// first_tier are the direct children of HTML
for (var i = 0; i < first_tier.length; i++) {
   // do something with each kid of HTML
   // as first_tier[i]
}
```

### Notes

This property is a read-only convenience for getting the `HTML` element associated with all valid HTML documents. The example above is quite typical: you actually want the `HTML` element so you can access all of *its* children, and so you use this `document` property to get a hold of it.

Note that `document` itself typically contains a single child node, `HTML`, which itself contains all of the elements in the actual HTML document as a `nodeList` of children.

### Specification

`documentElement`

# domain

**domain** gets/sets the domain of the current document.

### Syntax

```
domain_name = document.domain
document.domain = domain_name
```

### Parameters

`domain_name` is a string referring to the domain of the current document.

### Example

```
bad_domain = “www.love.com”;
if ( document.domain == bad_domain ) {
   window.close();
}
// for document www.love.com/good.html,
// this script closes the window
```

### Notes

This property returns `NULL` if the server of the document cannot be identified. In the DOM spec, this property is listed as being read-only, but Mozilla lets you set it.

### Specification

`domain`

# embeds

embeds returns a list of the embedded OBJECTS within the current document.

### Syntax

*`embedded_objects`* `= document.embeds`

### Parameters

`embedded_objects` is a `nodeList` of embedded objects.

### Example

`None.`

### Notes

None.

### Specification

DOM Level 0. *Not part of specification.*

# fgColor

**fgColor** gets/sets the foreground color, or text color, of the current document.

## Syntax

```
color = document.fgColor
document.fgColor = color
```

## Parameters

`color` is a string representing the color as a word (e.g., “red”) or as an octal value, as in HTML (e.g., “#eee”).

## Example

```
document.fgColor = “white”;
document.bgColor = “darkblue”;
```

## Notes

None.

## Specification

DOM Level 0. *Not part of specification.*

# firstChild

**firstChild** returns the first node in the list of direct children of the document.

### Syntax

*child* = document.firstChild

### Parameters

child is a node of the type element.

### Example

```
function fChild() {
  f = document.firstChild;
  alert(f.tagName);
} // returns: HTML
```

### Notes

Note that you may have to recurse into the DOM tree with this property to get the the child nodes you expect, since HTML is almost always given as the first child of the document itself.

### Specification

firstChild

# forms

forms returns a list of the FORM elements within the current document.

### Syntax

*form_list* = document.forms

### Parameters

form_list is a nodeList of FORM elements.

### Example

```
<form id=”marjoree”>
<input
   type=”button”
   onclick="alert(document.forms[0].id);"/>
</form>
```

### Notes

None.

### Specification

`forms`

# height

**height** gets/sets the height of the current document.

### Syntax

```
height_value = document.height
document.height = height_value
```

### Parameters

`height_value` is a string representing the height of the document in pixels, inches, or ems. If no type is specified (e.g., “px” in 200px), the value is assumed to be the number of pixels.

### Example

```
// make the window small on load
function onLoad() {
    document.height = “200”;
    document.width = “200”;
}
```

### Notes

None.

### Specification

`height`

# images

**images** returns a list of the images in the current document.

### Syntax

*`image_list`* `= document.images`

### Parameters

`image_list` is a `nodeList` of all the `IMG` elements in the document.

### Example

```
ilist = document.images;
for ( var i = 0; i < ilist.length; i++ ) {
    if ( ilist[i] == “banner.gif” ) {
         // found the banner
    }
}
```

### Notes

None.

### Specification

`images`

# implementation

Returns the DOM implementation associated with the current document.

### Syntax

```
implemetation = document.DOMImplentation
```

### Parameters

`implementation` is a `DOMImplementation` object

### Example

`None.`

### Notes

If available, the `DOMImplementation` is a special object that provides services for controlling things outside of a single document. For example, the `DOMImplementation` interface includes a `createDocumentType` method with which DTDs can be created for one or more documents managed by the implementation.

### Specification

`implementation`

# lastModified

Returns the date on which the current document was last modified.

### Syntax

```
date = document.lastModified
```

### Parameters

`date` is a string containing the date and time of last modification.

### Example

```
dump(document.lastModified);
// returns: Tuesday, July 10, 2001 10:19:42
```

### Notes

Note that as a string, **lastModified** cannot easily be used for comparisions between the modified dates of documents.

### Specification

link to DOM spec here

# linkColor

**linkcolor** gets/sets the color of links within the document.

### Syntax

```
color = document.linkColor
document.linkColor = color
```

### Parameters

`color` is a string representing the color as a word (e.g., “red’) or as an octal value, as in HTML (e.g., “#eee”)

### Example

```
document.linkColor = “blue”;
```

### Notes

None.

### Specification

DOM Level 0. *Not part of specification.*

# links

The **links** property returns a collection of all `AREA` elements and anchor elements in a document with a value for the `href` attribute.

### Syntax

```
linkArray = document.links
```

### Parameters

`linkArray` is an array of the links in the document

### Example

```
var links = document.links;
for(var i = 0; i < links.length; i++) {
    var linkHref =
       document.createTextNode(links[i].href);
    var lineBreak = document.createElement("br");
    document.body.appendChild(linkHref);
    document.body.appendChild(lineBreak);
}
```

### Notes

None.

### Specification

`links`

# location

Gets the URL of the current document.

## Syntax

```
loc = document.location
```

## Parameters

`loc` is the URL as a string.

## Example

```
dump(document.location);
// returns a string like
// http://www.peoplemagazine.com/juicybits.html
```

## Notes

`document.location` works the same as `document.URL`. Both are read-only properties unlike window.location, which can be set. Since the `document` object represents a single document or URL, its location cannot be changed.

## Specification

DOM Level 0. *Not part of specification.*

# namespaceURI

**namespaceURI** returns the XML namespace for the current document.

## Syntax

```
NSURI = document.namespaceURI
```

### Parameters

`NSURI` is a string containing the namespace.

### Example

`None.`

### Notes

The DOM does not handle or enforce namespace validation per se. It is up to the DOM application to do any validation necessary. Note too that the namespace prefix, once it is associated with a particular node, cannot be changed.

### Specification

`namespaceURI`

# nextSibling

Returns the `node` immediately following the current one in the tree.

### Syntax

```
node = elementNode.nextSibling
```

### Parameters

`node` is a node object.

### Example

```
// in a table, the cells are siblings
cell1 = document.getElementById(“td1”);
cell2 = cell1.nextSibling;
```

### Notes

Returns `NULL` if there are no more nodes.

### Specification

`nextSibling`

# nodeName

Returns the name of the current node as a string.

### Syntax

*name* = `nodeElement.nodeName`

### Parameters

`name` is a string that contains the name of the node.

### Example

```
div1 = document.getElementById(“d1”);
text_field = document.getElementById(“t”);
text_field.setAttribute(“value”, div1.nodeName);
// textfield reads “div” now
```

### Notes

None.

### Specification

`nodeName`

# nodeType

Returns a code representing the type of the underlying node

## Syntax

```
code = document.nodeType
```

## Parameters

`code` is an unsigned short with one of the following values:

```
ELEMENT_NODE                    = 1;
ATTRIBUTE_NODE                  = 2;
TEXT_NODE                       = 3;
CDATA_SECTION_NODE              = 4;
ENTITY_REFERENCE_NODE           = 5;
ENTITY_NODE                     = 6;
PROCESSING_INSTRUCTION_NODE     = 7;
COMMENT_NODE                    = 8;
DOCUMENT_NODE                   = 9;
DOCUMENT_TYPE_NODE              = 10;
DOCUMENT_FRAGMENT_NODE          = 11;
NOTATION_NODE                   = 12;
```

## Example

```
if document.nodeType != 9
   document.close()
else
   document.write(“<p>I’m a doc!</p>”);
```

## Notes

None.

## Specification

`nodeType`

# nodeValue

Returns the value of the current node.

### Syntax

```
value = document.nodeValue
```

### Parameters

`value` is a string containing the value of the current node.

### Example

```
None.
```

### Notes

For the document itself, the **nodeValue** is null. For text, comment, and CDATA nodes, nodeValue returns the content of the node. For attribute nodes, the value of the attribute is returned.

### Specification

`nodeValue`

# ownerDocument

ownerDocument returns the `document` object associated with this node.

### Syntax

```
ownerDoc = document.ownerDocument
```

### Parameters

`ownerdoc` is a `document` object.

### Example

```
None.
```

### Notes

This property returns `NULL` for a document object.

### Specification

`ownerDocument`

# parentNode

Returns the parent of the current node.

### Syntax

*node* = `element.parentNode`

### Parameters

`node` is a `node` object.

### Example

As you can see from the following not very interesting example, even simple HTML documents can contain a complex hierarchy of parents and children. In this case, the document object is a parent of the `HTML` object, which is a parent of the `BODY` object, which in turn is a parent of the `H1` object being examined.

```
// alerts: 9 for Document object
<html>
<head>
<script>
function init() {
  h1 = document.createElement('H1');
  t = document.createTextNode("heading 1");
  h1.appendChild(t);
  bod = document.getElementById("b");
  bod.appendChild(h1);
}

function findParent() {
  h1 = document.getElementsByTagName("H1");
  alert(h1[0].parentNode.parentNode.parentNode.nodeName);
}
</script>
</head>
<body id="b" onload="init();">
<form><input type="button" value="find parent"
onclick="findParent();" /></form>
</body>
</html>
```

## Notes

Note that this property returns `NULL` for the `document` itself, but can be used on the children of the document to refer back to the document or other intermediate parents (see example above).

## Specification

`parentNode`

# plugins

Returns a list of the plugins currently installed.

## Syntax

*pluginCollection* = document.plugins

## Parameters

pluginCollection is an object of the type PluginArray

## Example

The following example prints out information about the installed plug-ins for the high-level document. Note that properties available on the plugin object: **length** (on the array of plug-ins), **name**, **filename**, and **description**.

```
<script TYPE="text/javascript">
<!--
var L = navigator.plugins.length
document.write( L );
document.write("Plugins".bold());
document.write("<BR>");
document.write("Name | Filename | description".bold());
document.write("<BR>");
for(i=0; i<L; i++){
document.write(navigator.plugins[i].name);
document.write(" | ".bold());
document.write(navigator.plugins[i].filename);
document.write(" | ".bold());
document.write(navigator.plugins[i].description);
document.write("<BR>");
}
//-->
</script>
```

## Notes

None.

### Specification

DOM Level 0. *Not part of specification.*

# previousSibling

Returns the `node` immediately previous to the current one in the tree.

### Syntax

```
node = elementNode.previousSibling
```

### Parameters

`node` is a node object.

### Example

```
n1 = n2.previousSibling;
```

### Notes

Returns `NULL` if there are no more nodes.

### Specification

`previousSibling`

# referrer

Returns the URI of the page that linked to this page.

### Syntax

```
referring_page = document.referrer
```

### Parameters

`referring_page` is a string containing the URI of the referring page.

### Example

`None.`

### Notes

The value is an empty string if the user navigated to the page directly (not through a link, but, for example, via a bookmark).

Note that since this property returns only a string it does not give you DOM access to the referring page.

### Specification

`referrer`

# styleSheets

The **stylesheets** property returns a list of the `stylesheet` objects on the current document.

### Syntax

```
style_sheets = document.stylesheets
```

### Parameters

`style_sheets` is a `nodeList` of stylesheet objects.

### Example

`None.`

### Notes

None.

### Specification

DOM Level 0. *Not part of specification.*

# title

Gets/sets the title of the current document.

### Syntax

```
document.title = t
t = document.title
```

### Parameters

`t` is a string containing the document title.

### Example

This example checks to see if the document has a title that can be set and, if so, changes it to the value of the input parameter:

```
function setTitle(str){
  if(document.title){
    document.title=str;
}
```

You can call a function like this in the HTML with the following elements:

```
<input type="text"
   name="t"
   value="HTMLTitleElement"
   size=30>
<input type="button"
   value="set the HTMLDocument:title"
   onClick="setTitle(form.t.value);">
```

Where the button takes the value in the text element “t” and passes it to the function.

### Notes

None.

### Specification

DOM Level 0. *Not part of specification.*

# URL

Returns the URL of the current document.

### Syntax

```
url = document.URL
```

### Parameters

`url` is a string representing the URL of the current document.

### Example

```
var currentURL = document.URL;
alert(currentURL);
```

### Notes

URL is a replacement for the DOM Level 0 `document.location.href` property. However `document.URL` is not settable, unlike `document.location.href`.

### Specification

URL

# vlinkColor

Returns the color of links that the user has visited in the document.

## Syntax

```
color = document.vlinkColor
document.vlinkColor = color
```

## Parameters

`color` is a string representing the color as a word (e.g., “red’) or as an octal value, as in HTML (e.g., “#eee”)

## Example

`None.`

## Notes

The default for this property is purple.

## Specification

DOM Level 0. *Not part of specification.*

# width

Returns the width of the current document in pixels.

## Syntax

```
wdth = document.width
```

## Parameters

`wdth` is the width in pixels.

### Example

```
function init() {
  alert(document.width - 100);
}
```

### Notes

extra information

### Specification

`width`

# clear

The **clear** method clears the current document of all its content.

### Syntax

`document.clear()`

### Parameters

None.

### Example

```
<button label="empty" onclick="document.clear();" />
```

### Notes

None.

### Specification

DOM Level 0. Not part of specification.

# close

The document.close() method finishes writing to the open document.

## Syntax

```
document.close()
```

## Parameters

None.

## Example

```
// open a document to write to it.
// finish by closing the document.
document.open();
document.write("<P>The only content</P>.");
document.close();
```

## Notes

None.

## Specification

DOM Level 0. Not part of specification.

# createAttribute

**createAttribute** creates a new attribute on the current element.

## Syntax

```
attribute = element.createAttribute(name)
```

### Parameters

`attribute` is an attribute node.

`name` is a string containing the name of the attribute.

### Example

```
d = document.getElementById(“div1);
p = d.createAttribute(“proportion”);
p.value = “100”;
```

### Notes

The return is a node of type `attribute`. Once you have this node you can, as in the foregoing example, set its value with the **value** property. The DOM does not enforce what sort of attributes can be added to a particular element in this manner.

### Specification

`createAttribute`

# createDocumentFragment

**createDocumentFragment** creates an empty document fragment

### Syntax

*`documentFragment`* `= element.createDocumentFragment`

### Parameters

`documentFragment` is an object.

### Example

```
var frag = document.createDocumentFragment();
frag.appendChild(
   document.createTextNode('<div>Moveable Div</div>')
);
```

### Notes

A `documentFragment` is a "lightweight" or "minimal" `document` object. It is very common to want to be able to extract a portion of a document's tree or to create a new fragment of a document. Imagine implementing a user command like cut or rearranging a document by moving fragments around. It is desirable to have an object which can hold such fragments and it is quite natural to use a Node for this purpose.

While it is true that a `document` object could fulfill this role, a `document` object can potentially be a heavyweight object, depending on the underlying implementation. What is really needed for this is a very lightweight object. `documentFragment` is such an object.

### Specification

`createDocumentFragment`

# createElement

Creates an element of the type specified. Note that the instance returned implements the Element interface, so attributes can be specified directly on the returned object.

### Syntax

*`element`* `= element.createElement(type)`

### Parameters

`element` is an object.

`type` is a string that represents the type of element to be created.

### Example

```
div = document.createElement(“div”);
preface = document.getElementById(“preface”);
document.insertBefore(div, preface);
```

### Notes

In addition, if there are known attributes with default values, `attribute` nodes representing them are automatically created and attached to the element.

To create an element with a qualified name and namespace URI, use the `createElementNS` method.

### Specification

`createElement`

# createTextNode

Creates a new Text node.

### Syntax

*`text`* `= document.createTextNode(data)`

### Parameters

`text` is a `text` node.

`data` is a string containing the data to be put in the `text` node.

### Example

```
t = document.createTextNode(“Plain Old Text.”);
p = document.getElementById(“firstP”);
p.appendChild(t);
```

### Notes

None.

### Specification

`createTextNode`

# getElementById

Returns the element whose ID is specified.

### Syntax

```
element = document.getElementById(id);
```

### Parameters

`element` is an object.

`id` is a string representing the unique id of the element being sought.

### Example

```
// for <p id="p1" class="reggy" align="right">text</p>
first_p = document.getElementById("p1");
a_list = first_p.attributes;
```

### Notes

`getElementById` is an absolute mainstay of the DOM. One of the most important notions in DOM programming is that elements be identified uniquely so that they can be grabbed and manipulated.

If there is no element with the given ID, this function returns `NULL`. Note also that the DOM implementation must have information that says which attributes are of type ID. Attributes with the name "ID" are not of type ID unless so defined. Implementations that do not know whether attributes are of type ID or not are expected to return null.

`getElementById` was introduced in DOM Level 2.

### Specification

`getElementById`

# getElementsByName

Returns a list of elements of a given name in the document.

### Syntax

```
elements = document.getElementsByName(name)
```

### Parameters

`elements` is a `nodeList` of elements.

`name` is a string representing the value of the `name` attribute on the element.

### Example

```
// return some of the divs
<div name=”up”>200</div>
<div name=”up”>145</div>
<div name=”down”>146</div>
<div name=”other”>178</div>

up_divs = document.getElementsByName(“up”);
dump(up_divs.item(0).tagName); // returns “div”
```

### Notes

getElementsByName() returns a nodeList of all the elements with a given value for the name attribute. Unlike **`getElementsByTagName`**, which takes the name of the element itself, this method only works for elements for which name attributes have been explicitly given.

### Specification

`getElementsByName`

# getElementsByTagName

Returns a list of elements of a given name in the document.

### Syntax

*`elements`* `= document.getElementsByTagName(`*`Name`*`)`

### Parameters

`elements` is a `nodeList` of elements.

`tagName` is a string representing the name of the elements.

### Example

```
// check the alignment on a number of tables
tables = document.getElementsByTagName(“table”);
dump(“no. of tables: “ + tables.length);
for (var i = 0; i < tables.length; i++) {
   dump(tables[i].alignment);
}
```

### Notes

Another very useful feature of DOM programming is the **getElementsByTagName()** method, which returns a nodeList of all the elements with the given name.

### Specification

`getElementsByTagName`

# open

The **open()** methods opens a document stream for writing.

## Syntax

`document.open()`

## Parameters

None.

## Example

```
// In this example, the document contents are
// overwritten as the document
// is reinitialized on open().
document.write(“<html><p>remove me</p></html>”);
document.open(); // document is empty.
```

## Notes

If a document exists in the target, this method clears it (see the example above). Also, an automatic `document.open()` is executed if `document.write()` is called after the page has loaded.

## Specification

`open`

# write

Writes a string of text to a document stream opened by `document.`**`open`**`()`

## Syntax

`document.write(`*`text`*`)`

### Parameters

`text` is a string containing the text to be written to the current document.

### Example

```
document.open();
document.write(“<h1>hello!</h1>”);
document.close();
```

### Notes

Writing to a document that has already loaded without calling `document.open()` will automatically perform a `document.open()` call. Once you have finished writing, it is recommended to call `document.close()`, to tell the browser to finish loading the page. The text you write is parsed into the document's structure model. In the example above, the H1 element becomes a node in the document.

If the `document.write()` call is embedded directly in the HTML code, then it will not call `document.open()`. For example:

```
<div>
  <script type="text/javascript">
    document.write("<h1>Main title</h1>")
  </script>
</div>
```

### Specification

`write`

# writeln

Writes a string of text followed by a newline character to a document.

### Syntax

`document.writeln(`*`line`*`)`

### Parameters

`line` is string containing a line of text

### Example

```
document.writeln(“<p>enter password:</p>”);
```

### Notes

writeln is the same as write but adds a newline.

### Specification

`writeln`

# onblur

The **onblur** property returns the onBlur event handler code, if any, that exists on the current element.

### Syntax

```
event handling code = element.onblur
```

### Example

```
warnFunc = window.onblur;
```

### Notes

The blur event is raised when an element loses focus.

### Specification

DOM Level 0. Not part of specification.

# onclick

The **onclick** property returns the onClick event handler code on the current element.

## Syntax

```
event handling code = element.onclick
```

## Example

Perhaps the simplest example of using the onclick DOM property is to retrieve the existing onclick event handler code. The following function sets the event handler code, then gets it and displays it.

```
function pawnClick()  {
  p = document.getElementById("mutable");
  p.onclick = "alert('moot!');";
  text = p.onclick;
  alert(text);
}
```

## Notes

The click event is raised when the user clicks on an element.

## Specification

DOM Level 0. Not part of specification.

# ondblclick

The **ondblclick** property returns the onDblClick event handler code on the current element.

## Syntax

```
event handling code = element.ondblclick
```

### Example

```
// <img src=”pawn.gif” onDblClick=”movePawn(this);” />
function pawnClick()  {
  i = document.getElementById("img1");
  alert(i.ondblclick);
}
// alerts: function anonymous(event) { movePawn(this) }
```

### Notes

The dblclick event is raised when the user double clicks an element.

### Specification

DOM Level 0. Not part of specification.

# onfocus

The **onfocus** property returns the onFocus event handler code on the current element.

### Syntax

*event handling code* = element.onFocus

### Example

None.

### Notes

The focus event is raised when the user sets focus on the given element.

### Specification

DOM Level 0. Not part of specification.

# onkeydown

The **onkeydown** property returns the onKeyDown event handler code on the current element.

## Syntax

```
event handling code = element.onkeydown
```

## Example

```
None.
```

## Notes

The keydown event is raised when the user presses a keyboard key.

## Specification

DOM Level 0. Not part of specification.

# onkeypress

The **onkeypress** property returns the onKeyPress event handler code for the current element.

## Syntax

```
event handling code = element.onkeypress
```

## Example

```
None.
```

## Notes

The keypress event is raised when the user presses a key on the keyboard.

## Specification

DOM Level 0. Not part of specification.

# onkeyup

The **onkeyup** property returns the onKeyUp event handler code for the current element.

### Syntax

```
event handling code = element.onClick
```

### Example

```
None.
```

### Notes

The keyup event is raised when the user releases a key that's been pressed.

### Specification

DOM Level 0. Not part of specification.

# onmousedown

The **onmousedown** property returns the onMouseDown event handler code on the current element.

### Syntax

```
event handling code = element.onMouseDown
```

### Example

```
None.
```

### Notes

The mousedown event is raised when the user presses the left button button.

### Specification

DOM Level 0. Not part of specification.

# onmousemove

The **onmousemove** propety returns the onMouseDown event handler code on the current element.

### Syntax

```
event handling code = element.onMouseMove
```

### Example

```
None.
```

### Notes

The mousemove event is raised when the user moves the mouse.

### Specification

DOM Level 0. Not part of specification.

# onmouseout

The **onmouseout** property returns the onMouseOut event handler code on the current element.

### Syntax

```
event handling code = element.onMouseOut
```

### Example

```
None.
```

### Notes

The mouseout event is raised when the mouse leaves an element (e.g, when the mouse moves off of an image in the web page, the mouseout event is raised for that image element).

### Specification

DOM Level 0. Not part of specification.

# onmouseover

The **onmouseover** property returns the onMouseOver event handler code on the current element.

### Syntax

```
event handling code = element.onMouseOver
```

### Example

```
None.
```

### Notes

The mouseover event is raised when the user moves the mouse over a particular element.

### Specification

DOM Level 0. Not part of specification.

# onmouseup

The **onmouseup** property returns the onMouseUp event handler code on the current element.

## Syntax

*event handling code* = element.onMouseUp

## Example

None.

## Notes

The mouseup event is raised when the user releases the left mouse button.

## Specification

DOM Level 0. Not part of specification.

# onresize

The **onResize** property returns the onResize event handler code on the current element.

## Syntax

*event handling code* = element.onresize

## Example

```
// <img src=”pawn.gif” onResize=”growBoard();” />
function pawnClick()  {
   i = document.getElementById("img1");
   alert(i.onresize);
}
// alerts: function anonymous(event) { growBoard() }
```

## Notes

The resize event is raised when the user resizes a resizable element (such as a window).

## Specification

DOM Level 0. Not part of specification.

# DOM *Event* Reference

This chapter describes the DOM Level 2 Event Model as implemented by Gecko. The `event` object itself is described, as well as the interfaces for event registration on other nodes in the DOM, event handers and event listeners, and several longer examples that show how the various event interfaces relate to one another.

- **DOM Event Interface**
- **DOM Event Handler List**

## DOM Event Interface

The DOM `Event` interface is exposed in the `event` objects that are passed to the event handlers on various elements in the DOM. The following very simple example shows how an `event` object can be referenced and manipulated from within one such event handler.

```
function foo(e) {
  // event handling functions like this one
  // get a reference to the event they handle
  // (in this case as “e”).
  alert(e);
}
table_el.onclick = foo;
```

This example is woefully simplistic, but it shows an important feature of events in the Gecko DOM, which is that `event` objects in the DOM are typically accessed in the event handler functions. Once you have a reference to the `event` object, you can access all of the properties and methods described in this chapter.

Also see **Example 5: Event Propagation** in the Examples chapter for a more detailed example of how events move through the DOM.

In addition to the `event` object described here, the Gecko DOM also provides methods for registering event listeners on nodes in the DOM, removing those event listeners, and dispatching events from the DOM. These and the various **Event Handlers** on HTML or XML elements are the main entry points for events in the DOM. These three methods are described in the **DOM Element Reference** chapter of this book.

## Properties

| | |
|---|---|
| ***altKey*** | Returns a boolean indicating whether the `<alt>` key was pressed during the event. |
| ***bubbles*** | Returns a boolean indicating whether the event bubbles up through the DOM or not. |
| ***cancelBubble*** | Returns a boolean indicating whether the bubbling up of the event has been canceled or not. |
| ***cancelable*** | Returns a boolean indicating whether the event is cancelable. |
| ***charCode*** | Returns a number representing the character that was pressed as part of the key event. |
| ***clientX*** | Returns the horizontal position of the event. |
| ***clientY*** | Returns the vertical position of the event. |
| ***ctrlKey*** | Returns a boolean indicating whether the `<ctrl>` key was pressed during the event. |
| ***currentTarget*** | Returns a reference to the currently registered target for the event. |
| ***detail*** | Returns detail about the event, depending on the type of event. |
| ***eventPhase*** | Used to indicate which phase of the event flow is currently being evaluated. |
| ***isChar*** | Returns a boolean indicating whether the event produced a key character or not. |
| ***keyCode*** | Returns a number representing the character that was pressed as part of the key event. |
| ***layerX*** | Returns the horizontal coordinate of the event relative to the current layer. |
| ***layerY*** | Returns the vertical coordinate of the event relative to the current layer. |
| ***metaKey*** | Returns a boolean indicating whether the `meta` key was pressed during the event. |

| | |
|---|---|
| ***pageX*** | Returns the horizontal coordinate of the event relative to the page |
| ***pageY*** | Returns the vertical coorindate of the page relative to the page. |
| ***relatedTarget*** | Identifies a secondary target for the event. |
| ***screenX*** | Returns the horizontal position of the event on the screen. |
| ***screenY*** | Returns the vertical position of the event on the screen. |
| ***shiftKey*** | Returns a boolean indicating whether the `<shift>` key was pressed when the event was fired. |
| ***target*** | Returns a reference to the target to which the event was originally dispatched. |
| ***timeStamp*** | Returns the time that the event was created. |
| ***type*** | Returns the name of the event (case-insensitive). |
| ***view*** | The view attribute identifies the `AbstractView` from which the event was generated. |

## Methods

| | |
|---|---|
| ***initEvent*** | The initEvent method is used to initialize the value of an Event created through the `DocumentEvent` interface. |
| ***initMouseEvent*** | This method intializes the value of a mouse event once it's been created |
| ***initUIEvent*** | Initializes a UI event once it's been created. |
| ***preventDefault*** | Cancels the event (if it is cancelable). |
| ***stopPropagation*** | Stops the propagation of events further along in the DOM. |

# altKey

Indicates whether the <alt> key was pressed when the event was fired.

### Syntax

```
bool = event.altKey
```

### Parameters

`bool` is a boolean true | false.

### Example

```
function goInput(e) { // checks altkey and
  if e.altKey         // passes event along
    superSizeOutput(e);
  else
    doOutput(e)
```

### Notes

None.

### Specification

link to spec

# bubbles

Indicates whether the given event bubbles up through the DOM or not.

### Syntax

```
bool = event.bubbles
```

### Parameters

`bool` is a boolean true | false.

### Example

```
function goInput(e) {  // checks bubbles and
  if not e.bubbles     // passes event along if it's not
    passItOn(e);       // already bubbling
  doOutput(e)
```

### Notes

None..

### Specification

link to spec

# cancelBubble

Indicates whether the event bubbling was canceled for this event.

### Syntax

*bool* = event.cancelBubble

### Parameters

bool is a boolean true | false.

### Example

```
// example here
```

### Notes

Additional notes.

### Specification

link to spec

# cancelable

Indicates whether the event is cancelable.

### Syntax

```
bool = event.cancelable
```

### Parameters

`bool` is a boolean true | false.

### Example

```
// example here
```

### Notes

Whether an event can be canceled or not is something that's determined when that event is created. To cancel an event, you must call the preventDefault() method on the event, which keeps it from executing the default action that is its usual result.

### Specification

`cancelable`

# charCode

Returns a number representing the character that was pressed as part of the key event.

### Syntax

```
character = event.charCode
```

### Parameters

*character* is a number representing the key that was pressed for the event.

### Example

```
if e.charCode == 0
   // mouseEvent!
```

### Notes

Mouse events generate a **charCode** of `0`. For a list of the charCode values associated with particular keys, run the example in **Example 7: Displaying Event Object Constants** and view the resulting HTML table.

### Specification

```
Not part of specification.
See nsIDOMKeyEvent.IDL
```

# clientX

Returns the horizontal coordinate of the event within the DOM client area.

### Syntax

```
returnType = event.property
```

### Parameters

`param` is a something.

### Example

```
function checkClientClickMap(e) {
  if e.clientX < 50
    doRedButton();
  if 50 <= e.clientX < 100
    doYellowButton();
  if e.clientX >= 100
    doRedButton();
}
```

### Notes

See also **clientY**.

### Specification

`clientX`

# clientY

Returns the vertical coordinate of the event within the DOM client area.

### Syntax

*`returnType`* `= event.property`

### Parameters

`param` is a something.

### Example

```
function checkClientClickMap(e) {
  if e.clientY < 50
    doRedButton();
  if 50 <= e.clientY < 100
    doYellowButton();
  if e.clientY >= 100
    doRedButton();
}
```

### Notes

See also **clientX**.

### Specification

`clientY`

# ctrlKey

Indicates whether the `<ctrl>` key was pressed when the event was fired.

### Syntax

*`bool`* `= event.ctrlKey`

### Parameters

`bool` is a boolean true | false.

### Example

```
function goInput(e) { // checks ctrlKey and
  if e.ctrlKey        // passes event along
    superSizeOutput(e);
  else
    doOutput(e)
```

### Notes

None.

### Specification

`ctrlKey`

# currentTarget

Identifies the currently registered target for the event.

### Syntax

*`targetObj`* `= event.currentTarget`

### Parameters

*`targetObj`* is an object reference to a node in the DOM.

### Example

```
if e.currentTarget != t_el
   resetEventEngine();
```

### Notes

None.

### Specification

`currentTarget`

# detail

Returns detail about the event, depending on the type of event.

### Syntax

*`detailedInfo`* `= event.detail`

### Parameters

*`detailedInfo`* is a number.

## Example

```
<html>
<head>
  <title>event.detail</title>
  <script>
  function giveDetails(e) {
      // details = e.detail;
      text = document.getElementById("t");
      text.setAttribute("value", e.detail);
  }
  function init() {
      b1 = document.getElementById("b");
      b1.onclick=giveDetails;
  }
  </script>
</head>

<body onload="init();">

<form>
 <input id="b" type="button" value="details" />
 <input id="t" type="text" value=""  /><br/>
 <input type="reset" />
</form>

</body>
</html>
```

## Notes

**detail** is a number representing how many times the mouse has been clicked in the same location for this event. The value of **detail** is usually 1.

## Specification

`detail`

# eventPhase

Indicates which phase of the event flow is currently being evaluated.

## Syntax

*phase* = event.eventPhase

## Parameters

phase is a number with one of the following possible values:

| | |
|---|---|
| *0* | CAPTURING_PHASE |
| *1* | AT_TARGET |
| *2* | BUBBLING_PHASE |

## Example

```
// example here
```

## Notes

Additional notes.

## Specification

eventPhase

# isChar

Returns a boolean indicating whether the event produced a key character or not.

## Syntax

*bool* = event.isChar

## Parameters

boolean true | false

### Example

```
if e.isChar
  echoInput(e.type);
}
```

### Notes

Some key combos may raise events but not produce any character (example: `ctrl + alt` ?). When this is the case, **isChar** returns false.

**isChar** is used when event handlers need to do something like echo the input on the screen.

### Specification

`Not part of specification.`

# keyCode

Returns a number representing the character that was pressed as part of the key event.

### Syntax

*`character`* `= event.keyCode`

### Parameters

*`character`* is a number representing the key that was pressed for the event.

### Example

```
if e.keyCode == 0
   // mouseEvent!
```

### Notes

Mouse events generate a **keyCode** of `0`. For a list of the keyCode values associated with particular keys, run the example in **Example 7: Displaying Event Object Constants** and view the resulting HTML table.

### Specification

`Not part of specification.`
`See nsIDOMKeyEvent.IDL`

# layerX

Returns the horizontal coordinate of the event relative to the current layer.

### Syntax

*`coordinate`* `= event.layerX`

### Parameters

`coordinate` is a number.

### Example

```
// example here
```

### Notes

Additional notes.

### Specification

`link to spec`

# layerY

Returns the vertical coordinate of the event relative to the current layer.

### Syntax

```
coordinate = event.pageY
```

### Parameters

*coordinate* is a number.

### Example

```
// example here
```

### Notes

Additional notes.

### Specification

Not part of specification.

# metaKey

Returns a boolean indicating whether the <meta> key was pressed when the event was fired.

### Syntax

```
returnType = event.property
```

### Parameters

param is a something.

### Example

```
function goInput(e) { // checks metaKey and
  if e.metaKey        // passes event along
    superSizeOutput(e);
  else
    doOutput(e)
```

### Notes

None.

### Specification

`metaKey`

# pageX

Returns the horizontal coordinate of the event relative to the visible page.

### Syntax

*`coordinate`* `= event.pageX`

### Parameters

*`coordinate`* is a number.

### Example

```
// example here
```

### Notes

Additional notes.

### Specification

`link to spec`

# pageY

Returns the vertical coordinate of the event relative to the visible page.

### Syntax

*`coordindate`* `= event.pageY`

### Parameters

`coordinate` is a number.

### Example

```
// example here
```

### Notes

Additional notes.

### Specification

`link to spec`

# relatedTarget

Identifies a secondary target for the event.

### Syntax

*`sTargetObj`* `= event.relatedTarget`

## Parameters

*sTargetObj* is a reference to an additional event target.

## Example

```
var rel = event.relatedTarget;
// dump("LEAVING " + (rel ? rel.localName : "null") + "\n");

// relatedTarget is null when the titletip is first shown:
// a mouseout event fires because the mouse is exiting
// the main window and entering the titletip "window".

// relatedTarget is also null when the mouse exits the main
// window completely, so count how many times relatedTarget
// was null after titletip is first shown and hide popup
// the 2nd time
if (!rel) {
  ++this._mouseOutCount;
  if (this._mouseOutCount > 1)
    this.hidePopup();
  return;
}

// find out if the node we are entering is one of our
// anonymous children
while (rel) {
  if (rel == this)
    break;
  rel.parentNode;
}

// if the entered node is not a descendant of ours, hide
// the tooltip
if (rel != this && this._isMouseOver) {
  this.hidePopup();
}
```

### Notes

From the W3 spec: “Currently this attribute is used with the mouseover event to indicate the `EventTarget` which the pointing device exited and with the mouseout event to indicate the `EventTarget` which the pointing device entered.”

The example above is typical: the **`relatedTarget`** property is used to find the *other* element, if any, involved in an event. Events like mouseovers are oriented around a certain target, but may also involve a secondary target, such as the target that is exited as the mouseover fires for the primary target.

### Specification

`relatedTarget`

# screenX

Returns the horizontal coordinate of the event within the screen as a whole..

### Syntax

```
xCoord = event.screenX
```

### Parameters

*`xCoord`* is the offset from the left side of the screen in pixels.

### Example

```
function checkClickMap(e) {
  if e.screenX < 50
    doRedButton();
  if 50 <= e.screenX < 100
    doYellowButton();
  if e.screenX >= 100
    doRedButton();
}
```

### Notes

When you trap events on the window, document, or other roomy elements, you can get the coordinates of that event (e.g., a click) and route it properly, as the “clickMap” example demonstrates.

### Specification

`screenX`

# screenY

Returns the vertical coordinate of the event within the screen as a whole..

### Syntax

```
yCoord = event.screenY
```

### Parameters

*`yCoord`* is the offset from the top of the screen in pixels.

### Example

```
function checkClickMap(e) {
  if e.screenY < 50
    doRedButton();
  if 50 <= e.screenY < 100
    doYellowButton();
  if e.screenY >= 100
    doRedButton();
}
```

### Notes

When you trap events on the window, document, or other roomy elements, you can get the coordinates of that event (e.g., a click) and route it properly, as the “clickMap” example demonstrates.

### Specification

screenY

# shiftKey

Returns a boolean indicating whether the <shift> key was pressed when the event was fired.

### Syntax

*bool* = event.shiftKey

### Parameters

bool is a boolean true | false.

### Example

```
function goInput(e) { // checks shiftKey and
  if e.shiftKey       // passes event along
    superSizeOutput(e);
  else
    doOutput(e)
```

### Notes

None.

### Specification

shiftKey

# target

Returns a reference to the **`target`** to which the event was originally dispatched.

## Syntax

```
targ = event.target
```

## Parameters

*`targ`* is a reference to an `EventTarget`.

## Example

```
d = document.getElementById(“d1”);
if e.target != d
   resetGame(); // not my event!
```

## Notes

Additional notes.

## Specification

`target`

# timeStamp

Returns the time (in milliseconds since the epoch) that the event was created.

## Syntax

```
time = event.timeStamp
```

## Parameters

*`time`* is a number.

### Example

```
// example here
```

### Notes

This property only works if the event system supports it for the particular event.

### Specification

`timestamp`

# type

Returns the name of the event (case-insensitive).

### Syntax

*`type`* `= event.type`

### Parameters

*`type`* is a string.

### Example

```
// example here
```

### Notes

The **type** must be an XML name..

### Specification

`type`

# view

The view attribute identifies the `AbstractView` from which the event was generated.

## Syntax

```
aView = event.view
```

## Parameters

*`aView`* is a reference to an `AbstractView` object.

## Example

```
// example here
```

## Notes

None.

## Specification

`view`

# initEvent

The `initEvent` method is used to initialize the value of an Event created through the `DocumentEvent` interface.

## Syntax

```
event.initKeyEvent(type, bubbles, cancelable)
```

### Parameters

| | |
|---|---|
| *type* | The type of event |
| *bubbles* | A boolean indicating whether the event should bubble up through the event chain or not (see **bubbles**). |
| *cancelable* | A boolean indicating whether the event can be canceled (**cancelable**). |

### Example

```
// create a click event that bubbles up and
// cannot be canceled
event.initEvent("click", 1, 0)
```

### Notes

Events initialized in this way must have been created with the `DocumentEvent` interface method `createEvent()`. This method must be called to set the event before it is dispatched.

### Specification

`initEvent`

# initMouseEvent

This method intializes the value of a mouse event once it's been created (by the `createEvent()` method on the `DocumentEvent` interface).

## Syntax

```
event.initMouseEvent(String typeArg,
  boolean canBubbleArg,
  boolean cancelableArg,
  AbstractView viewArg,
  int detailArg,
  int screenXArg,
  int screenYArg,
  int clientXArg,
  int clientYArg,
  boolean ctrlKeyArg,
  boolean altKeyArg,
  boolean shiftKeyArg,
  boolean metaKeyArg,
  short buttonArg,
  EventTarget relatedTargetArg)
```

## Parameters

| | |
|---|---|
| *typeArg* | Specifies - the event type. |
| *canBubbleArg* | Specifies - whether or not the event can bubble. |
| *cancelableArg* | Specifies - whether or not the event's default action can be prevented. |
| *viewArg* | Specifies - the Event's AbstractView. |
| *detailArg* | Specifies - the Event's mouse click count. |
| *screenX* | ArgSpecifies - the Event's screen x coordinate |
| *screenYArg* | Specifies - the Event's screen y coordinate |
| *clientXArg* | Specifies - the Event's client x coordinate |
| *clientYArg* | Specifies - the Event's client y coordinate |
| *ctrlKeyArg* | Specifies - whether or not control key was depressed during the Event. |
| *altKeyArg* | Specifies - whether or not alt key was depressed during the Event. |
| *shiftKeyArg* | Specifies - whether or not shift key was depressed during the Event. |
| *metaKeyArg* | Specifies - whether or not meta key was depressed during the Event. |
| *buttonArg* | Specifies - the Event's mouse button. |
| *relatedTargetArg* | Specifies - the Event's related EventTarget. |

## Example

```
e.initMouseEvent(“click”, 1, 1,
  window, 1,
  10, 50, 10, 50,
  0, 0, 0, 0,
  1, div1)
```

## Notes

None.

### Specification

`initMouseEvent`

# initUIEvent

Initializes a UI event once it's been created.

### Syntax

`event.initUIEvent(`*`type`*`, `*`canBubble`*`, `*`view`*`, `*`detail`*`)`

### Parameters

| | |
|---|---|
| *type* | The type of event |
| *canBubble* | A boolean indicating whether the event should bubble up through the event chain or not (see **bubbles**). |
| *view* | The AbstractView associated with the event. |
| *detail* | Number indicating how many times the mouse has been clicked on a given screen location (usually 1). |

### Example

```
e = document.createEvent( // fill // );
e.initUIEvent(
  “click”
  0,
  window
  1
)
```

### Notes

None.

### Specification

`initUIEvent`

# preventDefault

Cancels the event (if it is cancelable).

### Syntax

```
event.preventDefault()
```

### Parameters

None.

### Example

```
if e.cancelable // may as well check.
  e.preventDefault();
```

### Notes

In this case, default is the default action performed when the event is handled. Calling preventDefault cancels this action.

### Specification

`preventDefault`

# stopPropagation

Prevents further propagation of the current event.

### Syntax

```
event.stopPropagation()
```

### Parameters

None.

### Example

```
e.stopPropagation();
```

### Notes

See **Example 5: Event Propagation** in the Examples chapter for a more detailed example of this method and event propagation in the DOM.

### Specification

`stopPropagation`

## DOM Event Handler List

The following is a complete list of the event handlers supported in the Gecko DOM. Note that not all elements support the full list. See the **DOM Element Reference** for the event handlers that are common to all elements.

| Event Handler | Event |
|---|---|
| `onmousedown` | mouse button is pressed down. |
| `onmouseup` | mouse button is released. |
| `onclick` | event raised when mouse is clicked. |
| `ondblclick` | mouse is double-clicked. |
| `onmouseover` | mouse cursor moves over the target. |
| `onmouseout` | mouse cursor leaves target. |
| `onmousemove` | mouse cursor moves. |

| | |
|---|---|
| `oncontextmenu` | context menu is created. |
| `onkeydown` | a key has been pressed. |
| `onkeyup` | a key has been released. |
| `onkeypress` | a key has been pressed. |
| `onfocus` | focus has been set on the target. |
| `onblur` | focus has moved away from the target. |
| `onload` | the element/window has loaded. |
| `onunload` | the element/window has been unloaded. |
| `onabort` | the action has been aborted. |
| `onerror` | there has been an error. |
| `onsubmit` | a form has been submitted. |
| `onreset` | a form has been reset. |
| `onchange` | a value in a form has been changed. |
| `onselect` | an element has been selected. |
| `oninput` | |
| `onpaint` | |
| `ontext` | |
| `onpopupShowing` | |
| `onpopupShown` | |
| `onpopupHiding` | |
| `onpopupHidden` | |
| `onclose` | the window/frame has been closed. |
| `oncommand` | the target element has been activated (e.g., clicked, selected, etc.) |
| `onbroadcast` | |
| `oncommandupdate` | |
| `ondragenter` | |
| `ondragover` | an item has been dragged over the event target. |
| `ondragexit` | |
| `ondragdrop` | an item has been dropped onto the event target. |
| `ondraggesture` | |
| `onresize` | Element/window has been resized. |

| | |
|---|---|
| `onscroll` | Window/frame has been scrolled |
| `overflow` | text in window/frame overflows available space. |
| `onunderflow` | |
| `onoverflowchanged` | |
| `onsubtreemodified` | a subtree of the current document has been modified in some way. |
| `onnodeinserted` | a node has been inserted into the document. |
| `onnoderemoved` | a node has been removed from the document. |
| `onnoderemovedfromdocument` | a node has been removed from the document. |
| `onnodeinsertedintodocument` | a new node has been inserted into the document. |
| `onattrmodified` | a DOM attribute has been modified. |
| `oncharacterdatamodified` | Character data has been modified. |

# DOM Style Reference

To ADD: using element.setAttribute("style", "background-color: blue;") will remove any existing style properties on the element, so it's considered dangerous.

This chapter describes the `style` objects and the various interfaces they make available for manipulating style rules for HTML and XML documents. The final section, **DOM CSS Properties List**, is a list of the style properties that can be set or returned with the `element.style` property.

- **DOM Style Object**
- **DOM styleSheet Object**
- **DOM cssRule Object**
- **DOM CSS Properties List**

The basic `style` object exposes the `StyleSheet` and the `CSSStyleSheet` interfaces from the *DOM Level 2 Events* specification. Those interfaces contain members like `insertRule`, `selectorText`, and `parentStyleSheet` for accessing and manipulating the individual style rules that make up a CSS stylesheet.

To get to the `style` objects from the `document`, you can use the `document.styleSheets` property and access the individual objects by index (e.g., `document.styleSheets[0]` is the first stylesheet defined for the document, etc.).

Though there are various syntaxes for expressing stylesheets for a document, Netscape supports CSS exclusively, so the `style` object retrieved from this array implements both the StyleSheet and CSSStyleSheet interfaces.

```
ss = document.styleSheets[1];
ss.cssRules[0].style.backgroundColor="blue";
```

The list of properties available in the DOM from the style property is given in the **DOM CSS Properties List** section below.

The element **style** property can also be used to get the styles on an element. However, this property only returns style attributes that have been set *in-line* (e.g, `<td style="background-color: lightblue">` returns the string "background-color:lightblue," though there may be other styles on the element from a stylesheet). Also, when you set this property on an element, you override and erase any styles that have set elsewhere.

Instead, the `getComputedStyle()` method on the `document.defaultView` object returns all styles that have actually been computerd for an element. See **Example 6: getComputedStyle** in the examples chapter for more information on how to use this method.

## DOM Style Object

The `style` object represents an individual style statement. Unlike the individual rules available from the `document.styleSheets` array, the style object is accessed from the `document` or from the elements to which that style is applied. It represents the *in-line* styles on a particular element.

More important than the two properties surfaced here is the use of the `style` object in the following sorts of manipulations, where the style object can be used to set individual style properties on an element:

```
<html>
<head>
  <link rel="StyleSheet" href="example.css"/>
  <script>
     function stilo() {
       document.getElementById("d").
            style.color = "orange";
     }
  </script>
</head>

<body>
<div id="d" class="thunder">Thunder</div>
<button onclick="stilo()">ss</button>
</body>
</html>
```

The **media** and **type** of the style may or may not be given. Note that you can also change styles on an element by getting a reference to that element and then using the setAttribute() DOM method to specify both the CSS property and its value.

```
el = document.getElementById(“some-element”);
el.setAttribute(“style”, “background-color:darkblue;”);
```

Be aware, however, that when you set the style attribute in this way, you are overwriting whatever style properties may already have been defined in the style attribute. If the document referenced in the snippet above by the id “some-element” has a `style` attribute in which the font size is set to `18px`, for example, that information is erased when the style attribute is manipulated in this crude way.

## Properties

| | |
|---|---|
| **media** | Specifies the intended destination medium for style information. |
| **type** | Returns the type of style being applied by this statement. |

# media

**media** specifies the intended destination medium for style information.

### Syntax

```
medium = style.media
style.media = medium
```

### Parameters

*`medium`* is a string describing a single medium or a comma-separated list.

### Example

```
example here
```

### Notes

The default value for **media** is “screen.”

### Specification

*DOM Level 2 Styles* - STYLE

# type

Returns the type of the current style.

## Syntax

```
type = style.type
```

## Parameters

*type* is a sting containing the type.

## Example

```
if ( newStyle.type != “text/css” ){
  // not supported!
  warnCSS();
}
```

## Notes

For Gecko, the type is most often given as “text/css.”

From the W3C spec on CSS: “The expectation is that binding-specific casting methods can be used to cast down from an instance of the CSSRule interface to the specific derived interface implied by the type.”

## Specification

link to DOM spec here

## DOM styleSheet Object

The stylesheet object represents the stylesheet itself. A stylesheet contains any number of separate rules, which can be manipulated with the `cssRule` (see **`DOM cssRule Object`** below).

Using the `stylesheet` object, you can add and delete style rules. You can also travel the hierarchy of stylesheets that can be associated with a particular document using the **parentStyleSheet** property.

## Properties

| | |
|---|---|
| **cssRules** | Returns all of the CSS rules in the stylesheet as an array. |
| **disabled** | This property indicates whether the current stylesheet has been applied or not. |
| **href** | Returns the location of the stylesheet. |
| **media** | Specifies the intended destination medium for style information. |
| **ownerNode** | Returns the node that associates this style sheet with the document. |
| **ownerRule** | If this style sheet comes from an `@import` rule, the **ownerRule** property will contain the CSSImportRule. |
| **parentStyleSheet** | Returns the stylesheet that is including this one, if any. |
| **title** | Returns the advisory title of the current style sheet. |
| **type** | Specifies the style sheet language for this style sheet. |

## Methods

| | |
|---|---|
| **deleteRule** | Deletes a rule from the stylesheet. |
| **insertRule** | Inserts a new style rule into the current style sheet. |

# cssRules

Returns all of the CSS rules in the stylesheet as an array.

## Syntax

```
rules = stylesheet.cssRules
```

## Parameters

*rules* is an array of individual `cssRule` objects.

## Example

```
// get to the first rule
first_rule = document.styleSheets[0].cssRules[0];
```

## Notes

See **DOM cssRule Object**.

## Specification

`cssRule`

# disabled

This property indicates whether the current stylesheet is applied or not.

## Syntax

```
res = stylesheet.disabled
```

### Parameters

*res* is a boolean True or False

### Example

```
// if the stylesheet is applied...
if (stylesheet.disabled) {
  // apply style in-line
}
```

### Notes

None.

### Specification

disabled

# href

Returns the location of the stylesheet.

### Syntax

*href* = stylesheet.href

### Parameters

*href* is a string containing the URI of the stylesheet.

### Example

```
// on a local machine:
<html>
<head>
  <link rel="StyleSheet" href="example.css" type="text/
css" />
  <script>
     function sref() {
       alert(document.styleSheets[0].href);
     }
  </script>
</head>

<body>
<div class="thunder">Thunder</div>
<button onclick="sref()">ss</button>
</body>
</html>

// returns “file:////C:/Windows/Desktop/example.css
```

### Notes

If the style sheet is a linked style sheet, the value of its attribute is its location. For inline style sheets, the value of this attribute is `NULL`.

### Specification

`href`

# media

**media** specifies the intended destination medium for style information.

### Syntax

```
medium = stylesheet.media
stylesheet.media = medium
```

### Parameters

*medium* is a string describing a single medium or a comma-separated list.

### Example

```
<link rel="StyleSheet"
  href="document.css"
  type="text/css"
  media="screen" />
```

### Notes

The default value for **media** is “screen.”

### Specification

*DOM Level 2 Styles* - STYLESHEET

# ownerNode

**ownerNode** returns the node that associates this style sheet with the document.

### Syntax

```
node = stylesheet.ownerNode
```

### Parameters

*node* is an object reference to the element that brings in the stylesheet.

### Example

```
<html>
<head>
  <link rel="StyleSheet" href="example.css" type="text/
css" />
  <script>
     function stilo() {
       alert(document.styleSheets[0].ownerNode);
     }
  </script>
</head>

<body>
<div class="thunder">Thunder</div>
<button onclick="stilo()">ss</button>
</body>
</html>
// displays “object HTMLLinkElement”
```

### Notes

For HTML, **ownerNode** may be the corresponding `LINK` or `STYLE` element. For XML, it may be the linking processing instruction. For style sheets that are included by other style sheets, the value of this attribute is `null`.

### Specification

*DOM Level 2 Styles* - `STYLESHEET`

# ownerRule

If this style sheet comes from an `@import` rule, the **ownerRule** property will contain the CSSImportRule.

### Syntax

*rule* = stylesheet.ownerRule

### Parameters

*rule* is a string containing the importing rule in the HTML or XML document

### Example

```
// example here
```

### Notes

Note that if the value of the **ownerNode** property on the current STYLE element is NULL, then then **ownerRule** returns the rule that blah. And vice versa.

### Specification

ownerRule

# parentStyleSheet

Returns the stylesheet that is including this one, if any.

### Syntax

*parent* = stylesheet.parentStyleSheet

### Parameters

*parent* is an object reference

### Example

```
// find the top level stylesheet
if (stylesheet.parentStyleSheet) {
  sheet = stylesheet.parentStyleSheet;
} else { sheet = stylesheet; }
```

### Notes

This property returns `NULL` is the current stylesheet is a top-level stylesheet or if stylesheeet inclusion is not supported.

### Specification

`parentStyleSheet`

# title

**title** returns the advisory title of the current style sheet.

### Syntax

`line of code`

### Example

```
// example here
```

### Notes

The **title** is often specified in the **ownerNode**.

### Specification

`title`

# type

**type** specifies the style sheet language for this style sheet.

### Syntax

*`type`* `= stylesheet.type`

### Parameters

*type* is a string.

### Example

```
ss.type = “text/css”;
```

### Notes

None.

### Specification

type

# deleteRule

The **deleteRule** method removes a style rule from the current style sheet object.

### Syntax

```
stylesheet.deleteRule(index)
```

### Parameters

*index* is a long number representing the position of the rule

### Example

```
myStyles.deleteRule(0);
```

### Notes

None.

### Specification

deleteRule

# insertRule

The **insertRule** method inserts a new style rule into the current style sheet.

## Syntax

`stylesheet.insertRule(`*`rule`*`, `*`index`*`)`

## Parameters

*`rule`* is a string containing the rule to be inserted (selector and declaration)

*`index`* is a number representing the position to be inserted

## Example

```
// push a new rule onto the top of my stylesheet
myStyle.insertRule(“#blanc { color: white }”, 0);
```

## Notes

For rule sets this contains both the selector and the style declaration. For at-rules, this specifies both the at-identifier and the rule content.

## Specification

`insertRule`

## DOM `cssRule` Object

The `cssRule` object represents a single CSS style rule. These rules may be a part of a stylesheet or they may be placed in-line with the individual nodes in the HTML or XML document. Each stylesheet object exposes an array of the cssRules that make it up, and you can also get to the rules on individual elements by using the `element.style` property.

## Properties

| | |
|---|---|
| **cssText** | **cssText** returns the actual text of the style rule. |
| **parentStyleSheet** | **parentStyleSheet** returns the stylesheet object in which the current rule is defined. |
| **selectorText** | **selectorText** gets/sets the textual representation of the selector for the rule set. |
| **style** | **style** returns the declaration block for the current style. |

# cssText

**cssText** returns the actual text of the style rule.

### Syntax

```
text = cssRule.cssText
```

### Parameters

*`text`* is a string containing the style rule text.

### Example

```
if ( myRule.cssText.indexOf(“background-color”) != -1 )
{
   bgRule = myRule;
}
...
```

### Notes

None.

### Specification

*DOM Level 2 Style* - `cssRule`

# parentStyleSheet

**parentStyleSheet** returns the stylesheet object in which the current rule is defined.

### Syntax

```
stylesheet = cssRule.parentStyleSheet
```

### Parameters

**stylesheet** is a `stylesheet` object

### Example

```
if ( bgRule.parentStyleSheet != mySheet ) {
   // alien style rule!
}
```

### Notes

See **DOM styleSheet Object** for more information about the `stylesheet` object interface.

### Specification

*DOM Level 2 Style* - `cssRule`

# selectorText

**selectorText** gets/sets the textual representation of the selector for the rule set.

### Syntax

```
text = cssRule.selectorText
cssRule.selectorText = text
```

### Parameters

**text** is a string containing the text of the selector.

### Example

```
// for cssrule: body { background-color: darkblue; }
cssrule = document.styleSheets[1] XXX.
selector = cssrule.selectorText;
// selector is now “body”
```

### Notes

The implementation may have stripped out insignificant whitespace while parsing the selector.

### Specification

*DOM Level 2 Style* - `cssRule`

# style

**style** returns the declaration block for the current style.

## Syntax

*styleObj* = cssRule.style

## Parameters

styleObj is an object reference to the style declaration

## Example

```
function stilo() {
   alert(document.styleSheets[0].
       cssRules[0].style.cssText);
}
// displays “background-color: gray;”
```

## Notes

The *declaration block* is that part of the style rule that appears within the braces and that actually provides the style definitions (for the *selector*, the part that comes before the braces).

## Specification

*DOM Level 2 Style* - cssRule

# DOM CSS Properties List

The following is a list of the CSS properties that are supported in the Netscape 6 DOM and accessible by means of the style property on elements.

The following complete document examples shows the typical use of the `element.style` property to get and set the CSS properties listed here:

```
<html>
<head>
<script>
  function changeStyle()  {
    c = document.getElementById("tid");
    c.style = "padding right: 20px";
  }
</script>
<table border="1"><tr>
  <td id="tid">Example Cell</td></tr>
</table>
<form>
  <input value="addpad"
    type="button"
    onclick="changeStyle();" />
</form>
</html>
```

Styles can be returned or set with the `style` property and these attributes but that you cannot set values *directly* using constructions such as `style="background-color: blue"` from the DOM, where the value is a string that contains both the attribute and the value you wish to set. By itself, the `style` property should only be used as a "getter" and not a "setter." In other words, the first of the following two constructions is bad, and the latter is better practice in the DOM:

```
bad: element.style = "background-color: blue";

good: element.style.backgroundColor = "blue";
```

Note that the bad example above may actually set the background color of the given element, but this assignment overwrites any style information that already existed on that element, and then cannot be added to or updated without other overwrites. The special style attributes available off of the element's style property allow you to "manage" the style of your elements in a safer and more organized way.

See also: the Element **style** property.

You can check the syntax for the values of these attributes by consulting the DOM CSS specification.

| | | |
|---|---|---|
| accelerator | font | pause |
| azimuth | fontFamily | pauseAfter |
| background | fontSize | pauseBefore |
| backgroundAttachment | fontSizeAdjust | pitch |
| backgroundColor | fontStretch | pitchRange |
| backgroundImage | fontStyle | playDuring |
| backgroundPosition | fontVariant | position |
| backgroundRepeat | fontWeight | quotes |
| border | height | richness |
| borderBottom | left | right |
| borderBottomColor | length | size |
| borderBottomStyle | letterSpacing | speak |
| borderBottomWidth | lineHeight | speakHeader |
| borderCollapse | listStyle | speakNumeral |
| borderColor | listStyleImage | speakPunctuation |
| borderLeft | listStylePosition | speechRate |
| borderLeftColor | listStyleType | stress |
| borderLeftStyle | margin | tableLayout |
| borderLeftWidth | marginBottom | textAlign |
| borderRight | marginLeft | textDecoration |
| borderRightColor | marginRight | textIndent |
| borderRightStyle | marginTop | textShadow |
| borderRightWidth | markerOffset | textTransform |
| borderSpacing | marks | top |
| borderStyle | maxHeight | unicodeBidi |
| borderTop | maxWidth | verticalAlign |
| borderTopColor | minHeight | visibility |
| borderTopStyle | minWidth | voiceFamily |
| borderTopWidth | MozBinding | volume |
| borderWidth | MozOpacity | whiteSpace |
| bottom | orphans | widows |
| captionSide | outline | width |

clear
clip
color
content
counterIncrement
counterReset
cssFloat
cssText
cue
cueAfter
cueBefore
cursor
direction
display
elevation
emptyCells
outlineColor
outlineStyle
outlineWidth
overflow
padding
paddingBottom
paddingLeft
paddingRight
paddingTop
page
pageBreakAfter
pageBreakBefore
pageBreakInside
parentRule
wordSpacing
zIndex

### DOM *Frame* Reference

This document contains reference information for the following DOM objects:

- **FRAMESET**
- **FRAME**
- **IFRAME**

# FRAMESET

The `FRAMESET` element is a structure for containing subframes in HTML. It manages `FRAME` elements but not `IFRAMES`, which are inserted "in-line" into the document. In HTML, a frameset takes the following basic structure:

```
<frameset>
  <frame src="some_doc.html" id="frame1" />
  <frame src="other_doc.html" />
</frameset>
```

The DOM `frameset` object provides minimal programmatic access to the FRAMESET HTML element. Its interface—the two optional properties **cols** and **rows**—are a way to indicate the dimensions of the frame set, how many subframes it manages. Note that the `frameset` object does not provide interfaces for getting to the subframes it manages.

To get the frames in a document, you must ask the document itself, using the `document.getElementById()` method and the id of the frame(s) you want, or the `document.getElementsByTagName("FRAME")`, which returns a `NamedNodeList` array:

```
f1 = document.getElementById(“frame1”);
alert(f1.src); // print the src of the first frame

frames = document.getElementsByTagName(“FRAME”);
for (var i = 0; i < frames.length; i++) {
  // do something with each frame as frames[i]

}
```

## Properties

| | |
|---|---|
| ***cols*** | This property sets or returns the size of the columns of frames in the frameset.. |
| ***rows*** | This property sets or returns the number of rows of frames in the frameset. |

# cols

This property sets or returns the size of the columns of frames in the frameset.

### Syntax

```
cols = frameSetElement.cols
frameSetElement.cols = cols
```

### Parameters

*cols* is the size of columns in the frameset as a comma-separated list.

### Example

```
// element in HTML: <frameset id=”fset” >
fs = document.getElementById(“fset”);
fs.cols = "200, *"; // two columns of frames, first 200px
```

### Notes

The **cols** and **rows** properties are often used together to lay out the dimensions of a frameset in HTML. If these are not specified, then the frameset simply counts the number of frames it manages. The number of values in the cols attribute determines how many frames there are; the “*” specifies that the column take up all of the remaining space.

### Specification

*DOM Level 2* -- `HTMLFrameSetElement`

# rows

This property sets or returns the number of rows of frames in the frameset.

### Syntax

```
rows = frameSetElement.rows
frameSetElement.rows = rows
```

### Parameters

*rows* is the number of rows in the frameset.

### Example

```
// element in HTML: <frameset id=”fset” >
fs = document.getElementById(“fset”);
fs.rows = 8; // eight rows of frames
```

### Notes

The **cols** and **rows** properties are often used together to lay out the dimensions of a frameset in HTML. If these are not specified, then the frameset simply counts the number of frames it manages. For example, if you have a frameset like the one below, then blah blah...

### Specification

*DOM Level 2* -- `HTMLFrameSetElement`

# FRAME

The `frame` object provides methods and properties for manipulating the HTML `FRAME` element. While many of the properties (e.g., **frameBorder**, **marginWidth**) handle the `FRAME` itself, the **contentDocument** property allows you to get the actual document contained in the subframe, which can then be futher manipulated.

## Properties

| | |
|---|---|
| ***contentDocument*** | The **contentDocument** property returns the document the frame contains, if any. |
| ***contentWindow*** | The `contentWindow` property returns the window object for the frame. |
| ***frameBorder*** | `frameBorder` gets/sets the border around the current frame. |
| ***longDesc*** | The **longDesc** property specifies a url for a longer description of the contents of the current frame. |
| ***marginHeight*** | Gets/sets the height of the frame's margin in pixels. |
| ***marginWidth*** | Gets/sets the width of the frame's margin in pixels. |

| | |
|---|---|
| ***name*** | The name of the frame element. |
| ***noResize*** | Gets/sets whether the user can resize the current frame. |
| ***scrolling*** | The **scrolling** property specifies whether the current frame should provide scrollbars or not. |
| ***src*** | The **src** property provides a url to load as content into the current frame. |

# contentDocument

The `contentDocument` property returns the document the frame contains, if any.

## Syntax

```
document = frameElement.contentDocument
```

## Parameters

*`document`* is an object reference to the document contained in the given frame.

## Example

```
f = document.getElementById(“frame”);
if ( f.contentDocument ) {
    src_doc = f.contentDocument;
    working_title = src_doc.title;
}
```

## Notes

This property returns `NULL` if there is no current document

## Specification

*DOM Level 2* -- `HTMLFrameElement`

# contentWindow

The `contentWindow` property returns the window object for the frame.

## Syntax

```
frameWindow = frameElement.contentWindow
```

## Parameters

*`frameWindow`* is an object reference to the `window` object for this frame.

## Example

```
f = document.getElementById(“frame”);
f.contentWindow.location = “http://mozilla.org”;
f.contentWindow.history.back();
```

## Notes

You can also get to the `window` object through a named frame. For example, a frame with the `name=”myFrame”` can refer back to the `window` object as `window.frames[“myFrame”]`.

## Specification

*DOM Level 2* -- `HTMLFrameElement`

# frameBorder

**frameBorder** gets/sets the border around the current frame.

### Syntax

```
border = frameElement.frameBorder
frameElement.frameBorder = border
```

### Parameters

*border* gives the border width in number of pixels as string.

### Example

```
// create a border around the current frame
f = document.getElementById(“frame-1”);
f.frameBorder = 2;
```

### Notes

Note that the data type for the value of the `frameBorder` is a string. This is because the value may be specified as either a number of pixels (e.g., “2”) or a percentage, in which case the percent sign is included. A value of “0” means no border at all.

### Specification

*DOM Level 2* -- `HTMLFrameElement`

# longDesc

The **longDesc** property specifies a url for a longer description of the contents of the current frame.

### Syntax

```
url = frameElement.longDesc
frameElement.longDesc = url
```

### Parameters

The *url* string provides a url where a longer description of the current frame can be found.

### Example

```
f = document.getElementById(“main-frame”);
f.longDesc = “http://www.netcape.com/supplements/
more.html”;
```

### Notes

In the case of FRAME elements, the **longDesc** property is a way to point to a longer description than the **title** of the FRAME provides

### Specification

*DOM Level 2* -- HTMLFrameElement

# marginHeight

Gets/sets the height of the frame’s margin in pixels.

### Syntax

```
sHeight = frameElement.marginHeight
frameElement.marginHeight = sHeight
```

### Parameters

The *sHeight* string gives the height of the frame’s margin in pixels.

### Example

```
f = document.getElementById(“frame1”);
f.marginHeight = 3;
```

### Notes

None.

### Specification

*DOM Level 2* -- `HTMLFrameElement`

# marginWidth

Gets/sets the width of the frame's margin in pixels.

### Syntax

```
sWidth = frameElement.marginWidth
frameElement.marginWidth = sWidth
```

### Parameters

The *`sWidth`* string gives the width of the frame's margin in pixels.

### Example

```
f = document.getElementById("frame1");
f.marginWidth = 3;
```

### Notes

None.

### Specification

*DOM Level 2* -- `HTMLFrameElement`

# name

The name of the current frame.

## Syntax

```
framename = frameElement.name
```

## Parameters

*frameName* is a string.

## Example

```
f = document.getElementById(“main-frame”);
if (f.name != “main”) {
  //
}
```

## Notes

In the HTML, you can set the FRAME name attribute directly: `<frame name=”f1”/>` and then refer to that frame in the DOM with `window.frames[“f1”]`.

## Specification

*DOM Level 2* -- `HTMLFrameElement`

# noResize

Gets/sets whether the user can resize the current frame.

### Syntax

```
bool = frameElement.noResize
frameElement.noResize = bool
```

### Parameters

*bool* is a boolean value indicating whether the current frame is resizable or not.

### Example

```
f = document.getElementById(“main-frame”);
if ( user_level == “barney” ) {
    f.noResize = true;
}
```

### Notes

None.

### Specification

*DOM Level 2* -- HTMLFrameElement

# scrolling

The **scrolling** property specifies whether the current frame should provide scrollbars or not.

### Syntax

```
sBars = frameElement.scrolling
frameElement.scrolling = sBars
```

### Parameters

*sBars* is a string with the value “auto”, “no,” or “yes.”

### Example

```
f = document.getElementById(“main-frame”);
f.scrolling = yes;
```

### Notes

The default value for this property is “auto,” which specifies that scrollbars be added whenever necessary. “yes” means that they are always present, and “no” means that they are never present.

### Specification

*DOM Level 2* -- `HTMLFrameElement`

## src

The **src** property provides a url to load as content into the current frame.

### Syntax

```
frameElement.src = sURL
```

### Parameters

*`sURL`* is a string containing a URL.

### Example

```
f = document.getElementById(“content-frame”);
f.src = “www.netscape.com”;
```

### Notes

None.

### Specification

*DOM Level 2* -- `HTMLFrameElement`

# IFRAME

The `IFRAME` element is very much like a `FRAME`. The difference is that an `IFRAME` can be inserted *in-line*, hence the name `IFRAME`. Regular `FRAME` elements take up as much space as they can (given the dimensions of the `FRAMESET`, the presence of other frames, and the confines of the document itself). But iframes can be aligned in their containing elements, given a specific height and width, etc.

## Properties

| | |
|---|---|
| ***align*** | Specifies how the `IFRAME` is to be aligned in the containing element. |
| ***contentDocument*** | The `contentDocument` property returns the document the frame contains, if any. |
| ***contentWindow*** | The `contentWindow` property returns the window parent of the iframe. |
| ***frameBorder*** | **frameBorder** gets/sets the border around the current `IFRAME`. |
| ***longDesc*** | The `longDesc` property points to a long description of the current iframe. |
| ***marginHeight*** | Gets/sets the height of the frame's margin in pixels. |
| ***marginWidth*** | Gets/sets the width of the frame's margin in pixels. |
| ***name*** | The name of the `IFRAME` element. |
| ***scrolling*** | The **scrolling** property specifies whether the current frame should provide scrollbars or not. |
| ***src*** | The **src** property provides a url to load as content into the current frame. |

# align

Specifies how the `IFRAME` is to be aligned in the containing element.

## Syntax

```
sAlign = iFrameElement.align
iFrameElement.align = sAlign
```

## Parameters

*`sAlign`* is a string with one of the values given in the Notes section below.

## Example

```
i = document.getElementById("first-frame");
i.align = "top";
```

## Notes

Possible values for align are "top", "middle", "bottom", "left" and "right." The latter two cause the `IFRAME` to float to the current right or left margin. The default value for **align** is bottom. The align attribute is deprecated in HTML 4.0

## Specification

*DOM Level 2* -- `HTMLIFrameElement`

# contentDocument

The `contentDocument` property returns the document the frame contains, if any.

## Syntax

```
document = iFrameElement.contentDocument
```

### Parameters

*document* is an object reference to the document contained in the given frame.

### Example

```
i= document.getElementById(“my-iframe”);
if (i.contentDocument ) {
   src_doc = i.contentDocument;
   working_title = src_doc.title;
}
```

### Notes

This property returns NULL if there is no current document.

### Specification

*DOM Level 2* -- HTMLFrameElement

# contentWindow

The contentWindow property returns the window object for the iframe.

### Syntax

*iframeWindow* = iframeElement.contentWindow

### Parameters

*iframeWindow* is an object reference to the window object for this iframe.

### Example

```
i = document.getElementById(“iframe”);
i.contentWindow.location = “http://mozilla.org”;
i.contentWindow.history.back();
```

### Notes

You can also get to the `window` object through a named iframe. For example, an iframe with the `name="myFrame"` can refer back to the `window` object as `window.frames["myFrame"]`.

### Specification

*None.*

# frameBorder

**frameBorder** gets/sets the border around the current `IFRAME`.

### Syntax

```
sBorder = iFrameElement.frameBorder
iFrameElement.frameBorder = sBorder
```

### Parameters

*`sBorder`* gives the border width in number of pixels as string.

### Example

```
// create a border around the current iframe
i= document.getElementById("iframe-1");
i.frameBorder = 2;
```

### Notes

Note that the data type for the value of the `frameBorder` is a string. This is because the value may be specified as either a number of pixels (e.g., "2") or a percentage, in which case the percent sign is included. A value of "0" means no border at all.

### Specification

*DOM Level 2* -- `HTMLFrameElement`

# longDesc

The `longDesc` property points to a long description of the current iframe.

## Syntax

```
url = IFrameElement.longDesc
```

## Parameters

The *`url`* string provides a url where a longer description of the current frame can be found.

## Example

```
f = document.getElementById("main-frame");
f.longDesc = "http://www.netcape.com/suppl/more.html";
```

## Notes

The "long description" is meant to provide more information about the contents of a frame element. In the case of an `<img>` element, which also uses the **longDesc** property, it is meant to accompany `<img>` alt descriptions.

## Specification

*DOM Level 2* -- `HTMLIFrameElement`

# marginHeight

Gets/sets the height of the frame's margin in pixels.

### Syntax

```
sHeight = iFrameElement.marginHeight
iFrameElement.marginHeight = sHeight
```

### Parameters

The *sHeight* string gives the height of the frame's margin in pixels.

### Example

```
i= document.getElementById("iframe1");
i.marginHeight = 3;
```

### Notes

Note that the data type for the value of the `marginHeight` is a string. This is because the value may be specified as either a number of pixels (e.g., "2") or a percentage, in which case the percent sign is included. A value of "0" means no margin at all.

### Specification

*DOM Level 2* -- `HTMLIFrameElement`

# marginWidth

Gets/sets the width of the frame's margin in pixels.

### Syntax

```
sWidth = iFrameElement.marginWidth
iFrameElement.marginWidth = sWidth
```

### Parameters

The *sWidth* string gives the width of the frame's margin in pixels.

### Example

```
i = document.getElementById(“iframe1”);
i.marginWidth = 3;
```

### Notes

Note that the data type for the value of the `marginWidth` is a string. This is because the value may be specified as either a number of pixels (e.g., “2”) or a percentage, in which case the percent sign is included. A value of “0” means no margin at all.

### Specification

*DOM Level 2* -- `HTMLIFrameElement`

# name

The name of the current iframe.

### Syntax

```
iframename = iframeElement.name
```

### Parameters

*`iframeName`* is a string.

### Example

```
i = document.getElementById(“main-frame”);
if (i.name != “main”) {
  //
}
```

### Notes

In the HTML, you can set the `IFRAME` name attribute directly: `<frame name="f1"/>` and then refer to that frame in the DOM with `window.frames["f1"]`.

### Specification

*DOM Level 2* -- `HTMLIFrameElement`

# scrolling

The **scrolling** property specifies whether the current frame should provide scrollbars or not.

### Syntax

```
sBars = iFrameElement.scrolling
iFrameElement.scrolling = sBars
```

### Parameters

*`sBars`* is a string with the value "auto", "no," or "yes."

### Example

```
i= document.getElementById("iframe");
i.scrolling = yes;
```

### Notes

The default value for this property is "auto," which specifies that scrollbars be added whenever necessary. "yes" means that they are always present, and "no" means that htey are never present.

### Specification

*DOM Level 2* -- `HTMLFrameElement`

# src

The **src** property provides a url to load as content into the current frame.

## Syntax

```
iFrameElement.src = sURL
sURL= iFrameElement.src
```

## Parameters

*sURL* is a string containing a URL.

## Example

```
i = document.getElementById(“content-frame”);
i.src = “www.netscape.com”;
```

## Notes

None.

## Specification

*DOM Level 2* -- `HTMLIFrameElement`

# DOM *HTML Elements* Reference

This chapter provides reference information for several specific `HTMLElement` interfaces.

- **HTMLFormElement Interface**
- **HTMLTableElement Interface**
- [under construction...]

# HTMLFormElement Interface

As `HTML` elements, `FORM` elements expose all of the properties and methods described in the *DOM Elements Reference* chapter. They also expose the specialized interface described here.

The APIs for manipulating FORM elements described here allow you to create and fully configure `FORM` elements using the DOM. The following snippet gives you some idea of how you might create a new form element and equip it with some of the attributes that the form submission process requires.

```
f = document.createElement(“form”);
body.appendChild(f);
f.action = “\cgi-bin\some.cgi”;
f.method = “POST”
...
f.submit(); // submit the newly-created form!
```

In addition, the following complete HTML document shows how to extract info from an existing form element and how to set some of the read/write properties on that `FORM`.

```
<html>
<head>
  <title>form tests</title>
  <script> function getFormInfo() {
    var info = "";
    ta = document.getElementById("tex");
    f = document.forms["myform"];
    info += "f.elements: "+f.elements+"\n";
    info += "f.length: "+f.length+"\n";
    info += "f.name: "+f.elements+"\n";
    info += "f.acceptCharset: "+f.acceptCharset+"\n";
    info += "f.action: "+f.action+"\n";
    info += "f.enctype: "+f.enctype+"\n";
    info += "f.encoding: "+f.encoding+"\n";
    info += "f.method: "+f.method+"\n";
    info += "f.target: "+f.target+"\n";

    ta.setAttribute("value", info);
  }
// cont'd...
```

```
  function setFormInfo() {
    f = document.forms["myform"];
    f.method = "GET";
    f.action = "/cgi-bin/evil_executable.cgi";
    f.name = "totally_new";
    // click info again to get this new data
    // back from the form
  }</script>
</head>

<body>
<h1>form tests</h1>
<form name="myform" id="myform" action="/cgi-bin/test"
method="POST">
<input type="button" value="info"
onclick="getFormInfo();"/>
<input type="button" value="set"
onclick="setFormInfo();"/>
<input type="reset" value="reset"/>
<br>
<textarea id="tex"
   style="min-height:300;min-width:300" />
</form>
</body>
</html>
```

## Properties

| | |
|---|---|
| ***elements*** | **elements** returns an array of all the form controls contained in the `FORM` element. |
| ***length*** | **length** returns the number of controls in the `FORM` element. |
| ***name*** | **name** returns the name of the current `FORM` element as a string. |
| ***acceptCharset*** | **elements** returns a list of the supported character sets for the current `FORM` element. |
| ***action*** | **action** gets/sets the action of the `FORM` element. |
| ***enctype*** | **enctype** gets/sets the content type of the `FORM` element. |
| ***encoding*** | **encoding** gets/sets the content type of the `FORM` element. |
| ***method*** | **method** gets/sets the HTTP method used to submit the form. |
| ***target*** | **target** gets/sets the target of the action (i.e., the frame to render its output in). |

## Methods

| | |
|---|---|
| ***submit()*** | **submit()** submits the form. |
| ***reset()*** | **reset()** resets the form to its initial state. |

# elements

**elements** returns an array of all the form controls contained in the `FORM` element.

### Syntax

```
controls = form.elements
```

### Parameters

*controls* is a nodeList.

### Example

```
inputs = document.getElementById(“form1”).elements
```

### Notes

None.

### Specification

*elements*

# length

**length** returns the number of controls in the FORM element.

### Syntax

```
num = form.elements
```

### Parameters

*num* is an integer.

### Example

```
if (document.getElementById(“form1”).length > 1) {
  // more than one form control here
}
```

### Notes

None.

### Specification

*length*

# name

**name** returns the name of the current `FORM` element as a string.

### Syntax

```
name = form.name
form.name = name
```

### Parameters

*name* is a `string`.

### Example

```
form1 = document.getElementById(“form1”).name;
if (form1 != document.form.form1) {
   // browser doesn’t support this form of reference
}
```

### Notes

Note that this property is read/write, which means that you can change the name of a form or set it if it hasn’t been set alreaDY.

### Specification

*name*

# acceptCharset

**elements** returns a list of the supported character sets for the current `FORM` element.

## Syntax

```
charSets = form.acceptCharset;
```

## Parameters

*charSets* is a `string`.

## Example

```
inputs = document.forms[“myform”].acceptCharset
```

## Notes

None.

## Specification

*acceptCharset*

# action

**action** gets/sets the action of the `FORM` element.

## Syntax

```
action = form.action
form.action = action
```

### Parameters

*action* is a `string`.

### Example

```
form.action = “/cgi-bin/publish”;
```

### Notes

The action of a form is the program that is executed on the server when the form is submitted. This property can be retrieved or set.

### Specification

*action*

# enctype

**enctype** gets/sets the content type of the `FORM` element.

### Syntax

```
enctype = form.enctype
form.enctype = enctype
```

### Parameters

*enctype* is a `string`.

### Example

```
form.enctype = “application/x-www-form-urlencoded”;
```

### Notes

The encoding type is generally `“application/x-www-form-urlencoded”`.

### Specification

*enctype*

# encoding

**encoding** gets/sets the content type of the `FORM` element.

### Syntax

```
encoding = form.enctype
form.enctype = encoding
```

### Parameters

*`encoding`* is a `string`.

### Example

```
form.encoding = “application/x-www-form-urlencoded”;
```

### Notes

The encoding type is generally `“application/x-www-form-urlencoded”`.

### Specification

*encoding*

# method

**method** gets/sets the HTTP method used to submit the form.

### Syntax

```
meth = form.method
form.method = meth
```

### Parameters

*meth* is a `string`.

### Example

```
document.forms[“myform”].method = “POST”;
```

### Notes

None.

### Specification

*method*

# target

**target** gets/sets the target of the action (i.e., the frame to render its output in).

### Syntax

```
targ = form.target
form.target = targ
```

### Parameters

*targ* is a `string`.

### Example

```
myForm.target = document.frames[1].name;
```

### Notes

None.

### Specification

*target*

# submit()

**submit()** submits the form.

### Syntax

```
form.submit()
```

### Parameters

*None.*

### Example

```
document.forms[“myform”].submit()
```

### Notes

This method does the same as the form submit button.

* can “automatically” submit things this way without having to hit the submit button. ?But do you have to pass in the form parameters?

### Specification

*submit*

# reset()

**reset()** resets the form to its initial state.

## Syntax

```
form.reset()
```

## Parameters

*None*.

## Example

```
document.forms[“myform”].reset();
```

## Notes

This method is the same as the form reset button.

## Specification

*reset*

# HTMLTableElement Interface

`table` objects expose the `HTMLTableElement` interface, which provides special properties and methods (beyond the regular `element` object interface they also have available to them by inheritance) for manipulating the layout and presentation of tables in HTML.

## Properties

| | |
|---|---|
| ***caption*** | **caption** returns the table caption. |
| ***tHead*** | **tHead** returns the table head. |
| ***tFoot*** | **tFoot** returns the table footer. |
| ***rows*** | **rows** returns the rows in the table. |
| ***tBodies*** | **tBodies** returns the table bodies. |
| ***align*** | **align** gets/sets the alignment of the table. |
| ***bgColor*** | **bgColor** gets/sets the background color of the table. |
| ***border*** | **border** gets/sets the table border. |
| ***cellPadding*** | **cellPadding** gets/setst the cell padding. |
| ***cellSpacing*** | **cellSpacing** gets/set the spacing around the table. |
| ***frame*** | **frame** specifies which sides of the table have borders. |
| ***rules*** | **rules** specifies which interior borders are visible. |
| ***summary*** | **summary** gets/sets the table summary. |
| ***width*** | **width** gets/sets the width of the table. |

## Methods

| | |
|---|---|
| ***createTHead()*** | **createTHead()** creates a table header. |
| ***deleteTHead()*** | **deleteTHead()** removes the table header. |
| ***createTFoot()*** | **createTFoot()** creates a table footer. |
| ***deleteTFoot()*** | **deleteTFoot()** removes a table footer. |

| | |
|---|---|
| ***createCaption()*** | **createCaption()** creates a new caption for the table. |
| ***deleteCaption()*** | **deleteCaption()** removes the table caption. |
| ***insertRow()*** | **insertRow()** inserts a new row. |
| ***deleteRow()*** | **deleteRow()** removes a row. |

# caption

**caption** returns the table caption.

## Syntax

```
caption = table.caption
```

## Parameters

*caption* is a string.

## Example

```
if (table.caption) {
  // do something with the caption
}
```

## Notes

This property returns void if no caption exists on the table.

## Specification

*caption*

# tHead

**tHead** returns the table's `THEAD`.

## Syntax

```
th_el = table.tHead
```

## Parameters

*th_el* is a `HTMLTableSectionElement`.

## Example

```
if (table.tHead == my_head_el) {
   ...
}
```

## Notes

This property returns `VOID` if no `THEAD` element exists.

## Specification

*tHead*

# tFoot

**tFoot** returns the table's `TFOOT` element.

## Syntax

```
th_el = table.tFoot
```

## Parameters

*tf_el* is a `HTMLTableSectionElement`.

### Example

```
if (table.tFoot == my_foot) {
   ...
}
```

### Notes

This property returns `VOID` if no `TFOOT` element exists.

### Specification

*tFoot*

# rows

**rows** returns a collection of the rows in the table.

### Syntax

```
rows = table.rows
```

### Parameters

*rows* is an `HTMLCollection`.

### Example

```
myrows = mytable.rows;
```

### Notes

The collection returned by this property includes the `THEAD` and `TFOOT` and `TBODY` elements, if any, on the current table.

### Specification

*rows*

# tBodies

**tBodies** returns a collection of the table bodies.

## Syntax

```
bodies = table.tBodies
```

## Parameters

*bodies* is an `HTMLCollection`.

## Example

```
length(mytable.tBodies);
```

## Notes

None.

## Specification

*tBodies*

# align

**align** gets/sets the alignment of the table.

### Syntax

```
alignment = table.align
table.align = alignment
```

### Parameters

*alignment* is a string with one of the following values:

*left*

*center*

*right*

### Example

```
mytable.align = “center”;
```

### Notes

The **align** attribute is deprecated in HTML4.0.

### Specification

*align*

# bgColor

**bgcolor** gets/sets the background color of the table.

### Syntax

```
color = table.bgColor
table.bgColor = color
```

### Parameters

*color* is a string representing a color value.

### Example

```
mytable.bgColor = “lightblue”;
```

### Notes

The **bgColor** attribute is deprecated in HTML 4.0.

### Specification

*bgColor*

# border

**border** gets/sets the border width.

### Syntax

```
table.border = width
width = table.border
```

### Parameters

*width* is a string representing the width in pixels.

### Example

```
mytable.border=”2”;
```

### Notes

This attribute is deprecated in HTML 4.0.

### Specification

*border*

# cellPadding

**cellPadding** gets/sets the padding around the individual cells of the table.

## Syntax

```
table.cellPadding = padding
padding = table.cellPadding
```

## Parameters

*padding* is a string representing the padding in pixels.

## Example

```
mytable.cellPadding = “10”;
```

## Notes

None.

## Specification

*cellPadding*

# cellSpacing

**cellSpacing** gets/sets the spacing around the individual cells of the table.

## Syntax

```
table.cellSpacing = spacing
spacing = table.cellSpacing
```

### Parameters

*spacing* is a string representing the spacing around the table in pixels.

### Example

```
mytable.cellSpacing = “10”;
```

### Notes

None.

### Specification

*cellSpacing*

# frame

**frame** specifies which external table borders to render.

### Syntax

```
table.frame = side
side = table.frame
```

### Parameters

*side* is a string with one of the following values:

| | |
|---|---|
| *void* | no sides. this is the default. |
| *above* | top side |
| *below* | bottom side |
| *hsides* | top and bottom only |
| *vsides* | right and left sides only |

| | |
|---|---|
| *lhs* | left-hand side only |
| *rhs* | right-hand side only |
| *box* | all four sides |
| *border* | all four sides |

### Example

```
mytable.frame = “border”;
mytable.border = “2px”;
```

### Notes

None.

### Specification

*frame*

# rules

**rules** specifies which cell borders to render in the table.

### Syntax

```
table.rules = rules
rules = table.rules
```

### Parameters

`rules` is a string with one of the following values:

| | |
|---|---|
| *none* | no rules |

| | |
|---|---|
| *groups* | lines between groups only |
| *rows* | lines between rows |
| *cols* | lines between cols |
| *all* | lines between all cells |

### Example

```
t = document.getElementById(“mytable”);
t.rules = “all”; // turn on all the internal borders
```

### Notes

None.

### Specification

*rules*

# summary

**summary** gets/sets a table description.

### Syntax

```
table.summary = summary
summary = table.summary
```

### Parameters

*summary* is a string.

### Example

```
t.rules = “none”;
t.summary = “removed internal borders”;
```

### Notes

None.

### Specification

*summary*

# width

**width** specifies the desired width of the table.

### Syntax

```
table.width = width
width = table.width
```

### Parameters

*`width`* is a string representing the width in number of pixels or as a percentage value.

### Example

```
mytable.width=”75%”
```

### Notes

None.

### Specification

*width*

# createTHead()

**createTHead()** creates a new `THEAD` for the table.

## Syntax

```
th = table.createTHead()
```

## Parameters

*`th`* is an `HTMLElement`.

## Example

```
myhead = mytable.createTHead();
//checking:
myhead == mytable.tHead
```

## Notes

If the element already exists on the table, then this method returns that element

## Specification

*createTHead()*

# deleteTHead()

**deleteTHead()** removes a `THEAD` from the table.

## Syntax

```
table.deleteTHead()
```

## Parameters

*`None.`*

### Example

```
mytable.deleteTHead();
```

### Notes

None.

### Specification

*deleteTHead()*

# createTFoot()

**createTFoot()** creates a new `TFOOT` for the table.

### Syntax

```
tf = table.createTFoot()
```

### Parameters

*`tf`* is an `HTMLElement`.

### Example

```
myfoot = mytable.createTFoot();
//checking:
myfoot == mytable.tFoot
```

### Notes

If the element already exists on the table, then this method returns that element

### Specification

*createTFoot()*

# deleteTFoot()

**deleteTFoot()** removes a `TFOOT` from the table.

### Syntax

```
table.deleteTFoot()
```

### Parameters

*`None.`*

### Example

```
mytable.deleteTFoot();
```

### Notes

None.

### Specification

*deleteTFoot()*

# createCaption()

**createCaption()** creates a new caption for the table.

### Syntax

```
tcap = table.createCaption()
```

### Parameters

*tcap* is an `HTMLElement`.

### Example

```
mycap = mytable.createCaption();
```

### Notes

If the element already exists on the table, then this method returns that element.

### Specification

*createCaption()*

# deleteCaption()

**deleteCaption()** removes the caption from the table.

### Syntax

```
table.deleteCaption()
```

### Parameters

*None*.

### Example

```
mytable.deleteCaption();
```

### Notes

None.

### Specification

*deleteCaption()*

# insertRow()

**insertRow()** inserts a new row in the table.

### Syntax

```
row = table.insertRow(index)
```

### Parameters

*index* is a number representing where in the table to insert the new row.

*row* is an `HTMLElement`

### Example

```
newrow = mytable.insertRow(0); // insert a new first row
```

### Notes

None.

### Specification

*insertRow()*

# deleteRow()

**deleteRow()** removes a row from the table.

## Syntax

```
table.deleteRow(index)
```

## Parameters

*index* is a number representing the row that should be deleted.

## Example

```
mytable.deleteRow(1); // delete the second row
```

## Notes

None.

## Specification

*deleteRow()*

# DOM *Range* Reference

This chapter provides reference information for the DOM `Range` interface.



## DOM 2 Range Interface

### Properties

| | |
|---|---|
| ***collapsed*** | Returns a boolean indicating whether a range is collapsed. |
| ***commonAncestorContainer*** | Returns the deepest `Node` that contains the `startContainer` and `endContainer` Nodes. |
| ***endContainer*** | Returns the `Node` within which the Range ends. |
| ***endOffset*** | Returns a number representing where in the `endContainer` the Range ends. |
| ***startContainer*** | Returns the `Node` within which the Range starts. |
| ***startOffset*** | Returns a number representing where in the `startContainer` the Range starts. |

## Creation Methods

| | |
|---|---|
| ***createRange*** | Returns a new `Range` object. |
| ***setStart*** | Sets the start position of a `Range`. |
| ***setEnd*** | Sets the end position of a `Range`. |
| ***setStartBefore*** | Sets the start position of a `Range` relative to another `Node`. |
| ***setStartAfter*** | Sets the start position of a Range relative to another Node. |
| ***setEndBefore*** | Sets the end position of a `Range` relative to another `Node`. |
| ***setEndAfter*** | Sets the end position of a Range relative to another Node. |
| ***selectNode*** | Sets the `Range` to contain the node and its contents. |
| ***selectNodeContents*** | Sets the `Range` to contain the contents of a `Node`. |

## Editing Methods

| | |
|---|---|
| ***collapse*** | Collapses the `Range` to one of its boundary points. |
| ***cloneContents*** | Returns a document fragment copying the nodes of a Range. |
| ***deleteContents*** | Removes the contents of a Range from the document. |
| ***extractContents*** | Moves contents of a Range from the document tree into a document fragment |
| ***insertNode*** | Insert a node at the start of a Range. |
| ***surroundContents*** | Moves content of a Range into a new node. |

### Other Methods

| | |
|---|---|
| ***compareBoundaryPoints*** | Compares the boundary points of two Ranges. |
| ***cloneRange*** | Returns a Range object with boundary points identical to the cloned Range. |
| ***detach*** | Releases Range from use to improve performance. |
| ***toString*** | Returns the text of the Range. |

# collapsed

Returns a boolean indicating whether a range is collapsed.

## Syntax

```
isCollapsed = range.collapsed;
```

## Parameters

*`isCollapsed`* is boolean with values of true or false

## Example

```
range = document.createRange();
range.setStart(startNode,startOffset);
range.setEnd(endNode,endOffset);
isCollapsed = range.collapsed;
```

## Notes

Returns a boolean of true if the start and end boundary points of the Range are the same point in the DOM, false if not.

A collapsed Range is empty, containing no content, specifying a single-point in a DOM tree. The collapsed property is read-only. To collapse a range, see the `collapse` method

### Specification:

*collapsed*

# commonAncestorContainer

Returns the deepest `Node` that contains the `startContainer` and `endContainer` Nodes.

### Syntax

```
rangeAncestor = range.commonAncestorContainer;
```

### Parameters

*`rangeAncestor`* is a `Node` of type `Document`, `DocumentFragment`, or `Attr`

### Example

```
range = document.createRange();
range.setStart(startNode,startOffset);
range.setEnd(endNode,endOffset);
rangeAncestor = range.commonAncestorContainer;
```

### Notes

Returns the deepest, or further down the document tree, `Node` that contains both the `startContainer` and `endContainer` nodes. Since a Range need not be continuous, and may also partially select Nodes, this is a convenient way to find a Node which encloses a Range.

This property is read-only. To change the ancestor container of a Node, consider using the various methods to set the start and end positions of the Range.

### Specification:

*commonAncestorContainer*

# endContainer

Returns the Node within which the Range ends.

### Syntax

```
endRangeNode = range.endContainer;
```

### Parameters

*endRangeNode* is a reference to a `Node`.

### Example

```
range = document.createRange();
range.setStart(startNode,startOffset);
range.setEnd(endNode,endOffset);
endRangeNode = range.endContainer;
```

### Notes

Returns a reference to the Node in the document within which the Range ends  This property is read-only.  To change the end position of a node, use one of the `setEnd` methods.

### Specification:

*endParent*

# endOffset

Returns a number representing where in the endContainer the Range ends.

## Syntax

```
endRangeOffset = range.endOffset;
```

## Parameters

*endRangeOffset* is the number characters or child Node index where the Range ends in the endContainer

## Example

```
range = document.createRange();
range.setStart(startNode,startOffset);
range.setEnd(endNode,endOffset);
endRangeOffset = range.endOffset;
```

## Notes

**endOffset** has two meanings. If the endContainer is a Node of type Text, Comment, or CDATASection, then the offset is the number of characters from the start of the endContainer to the boundary point of the Range. For other Node types, the **endOffset** is the number of child nodes between the start of the endContainer and the boundary point of the Range. This property is read-only. To change the **endOffset** of a Range, use one of the setEnd methods.

## Specification:

*endOffset*

# startContainer

Returns the Node within which the Range starts.

### Syntax

```
startRangeNode = range.startContainer;
```

### Parameters

*startRangeNode* is a reference to a Node

### Example

```
range = document.createRange();
range.setStart(startNode,startOffset);
range.setEnd(endNode,endOffset);
startRangeNode = range.startContainer;
```

### Notes

Returns a reference to the Node in the document within which the Range starts. This property is read-only. To change the start position of a node, use one of the `setStart` methods.

### Specification:

*startParent*

# startOffset

Returns a number representing where in the `startContainer` the Range starts.

### Syntax

```
startRangeOffset = range.startOffset;
```

### Parameters

*startRangeOffset* is the number characters or a child Node index where the Range starts in the startContainer

### Example

```
range = document.createRange();
range.setStart(startNode,startOffset);
range.setEnd(endNode,endOffset);
startRangeOffset = range.startOffset;
```

### Notes

**startOffset** has two meanings. If the `startContainer` is a Node of type `Text`, `Comment`, or `CDATASection`, then the offset is the number of characters from the start of the `startContainer` to the boundary point of the Range. For other Node types, the **startOffset** is the number of child nodes between the start of the `startContainer` and the boundary point of the Range. This property is read-only. To change the **startOffset** of a Range, use one of the `setStart` methods.

### Specification

*startOffset*

# createRange

Returns a new `Range` object.

### Syntax

```
range = document.createRange();
```

### Parameters

*`range`* is a new instance of the `Range` object

### Example

```
range = document.createRange();
range.setStart(startNode,startOffset);
range.setEnd(endNode,endOffset);
```

### Notes

Once a `Range` is created, you need to set its boundary points before you can make use of most of its methods.

### Specification:

*createRange*

# setStart

Sets the start position of a `Range`.

### Syntax

```
range.setStart(startNode,startOffset);
```

### Parameters

The **setStart** method takes the following parameters:

| | |
|---|---|
| *`startNode`* | The `Node` to start the `Range` |
| *`startOffset`* | An integer greater than or equal to zero representing the offset for the start of the `Range` from the start of `startNode`. |

### Example

```
range = document.createRange();
startNode = document.getElementsByTagName("p").item(2);
startOffset = 0;
range.setStart(startNode,startOffset);
```

### Notes

If the `startNode` is a `Node` of type `Text`, `Comment`, or `CDATASection`, then `startOffset` is the number of characters from the start of `startNode`. For other `Node` types, `startOffset` is the number of child nodes between the start of the `startNode`.

### Specification:

*setStart*

# setEnd

Sets the end position of a `Range`.

### Syntax

```
range.setEnd(endNode,endOffset);
```

### Parameters

The **setEnd** method takes the following parameters:

| | |
|---|---|
| *endNode* | The `Node` to end the `Range` |
| *endOffset* | An integer greater than or equal to zero representing the offset for the end of the `Range` from the start of `endNode`. |

### Example

```
range = document.createRange();
endNode = document.getElementsByTagName("p").item(3);
endOffset = document.getElementsByTagName("p").item(3).childNodes.length;
range.setEnd(endNode,endOffset);
```

### Notes

If the `endNode` is a `Node` of type `Text`, `Comment`, or `CDATASection`, then `endOffset` is the number of characters from the start of `endNode`. For other `Node` types, `endOffset` is the number of child nodes between the start of the `endNode`.

### Specification:

*setEnd*

# setStartBefore

Sets the start position of a `Range` relative to another `Node`.

### Syntax

```
range.setStartBefore(referenceNode);
```

### Parameters

The **setStartBefore** method takes the following parameters:

*`referenceNode`* The `Node` to start the `Range` before

### Example

```
range = document.createRange();
referenceNode =
  document.getElementsByTagName("div").item(0);
range.setStartBefore(referenceNode);
```

### Notes

The parent Node of the start of the Range will be the same as that for the referenceNode.

### Specification:

*setStartBefore*

# setStartAfter

Sets the start position of a Range relative to another Node.

## Syntax

```
range.setStartAfter(referenceNode);
```

## Parameters

The setStartBefore() method takes the following parameters:

*`referenceNode`* The `Node` to start the `Range` after

## Example

```
range = document.createRange();
referenceNode =
document.getElementsByTagName("div").item(0);
range.setStartAfter(referenceNode);
```

## Notes

The parent `Node` of the start of the `Range` will be the same as that for the `referenceNode`.

## Specification:

*setStartAfter*

# setEndBefore

Sets the end position of a `Range` relative to another `Node`.

### Syntax

```
range.setEndBefore(referenceNode);
```

### Parameters

The setEndBefore() method takes the following parameters:

referenceNode  The Node to end the Range before

### Example

```
range = document.createRange();
referenceNode =
document.getElementsByTagName("div").item(0);
range.setEndBefore(referenceNode);
```

### Notes

The parent Node of end of the Range will be the same as that for the referenceNode.

### Specification

*setEndBefore*

# setEndAfter

Sets the end position of a Range relative to another Node.

### Syntax

```
range.setEndAfter(referenceNode);
```

### Parameters

The setEndAfter() method takes the following parameters:

*referenceNode* The Node to end the Range after

### Example

```
range = document.createRange();
referenceNode =
document.getElementsByTagName("div").item(0);
range.setEndAfter(referenceNode);
```

### Notes

The parent Node of end of the Range will be the same as that for the referenceNode.

### Specification

*setEndAfter*

# selectNode

Sets the Range to contain the node and its contents.

### Syntax

```
range.selectNode(referenceNode);
```

### Parameters

The selectNode() method takes the following parameters:

*referenceNode* The Node to select within a Range

### Example

```
range = document.createRange();
referenceNode =
document.getElementsByTagName("div").item(0);
range.selectNode(referenceNode);
```

### Notes

The parent Node of the start and end of the Range will be the same as the parent of the referenceNode.

### Specification

*selectNode*

# selectNodeContents

Sets the `Range` to contain the contents of a `Node`.

### Syntax

```
range.selectNodeContents(referenceNode);
```

### Parameters

The **selectNodeContents** method takes the following parameters:

| | |
|---|---|
| *`referenceNode`* | The `Node` whose contents will be selected within a `Range` |

### Example

```
range = document.createRange();
referenceNode =
document.getElementsByTagName("div").item(0);
range.selectNodeContents(referenceNode);
```

### Notes

The parent `Node` of the start and end of the `Range` will be the `referenceNode`. The `startOffset` is 0, and the `endOffset` is the number of child `Nodes` or number of characters contained in the reference node.

### Specification

*selectNodeContents*

# collapse

Collapses the `Range` to one of its boundary points.

### Syntax

```
range.collapse(toStart);
```

### Parameters

The **collapse** method takes the following parameters:

| | |
|---|---|
| *`toStart`* | A boolean, `true` collapses the `Range` to its start, `false` to its end. |

### Example

```
range = document.createRange();
referenceNode =
document.getElementsByTagName("div").item(0);
range.selectNode(referenceNode);
range.collapse(true);
```

### Notes

A collapsed `Range` is empty, containing no content, specifying a single-point in a DOM tree. To determine if a `Range` is already collapsed, see the **collapsed** property.

### Specification

*collapse*

# cloneContents

Returns a document fragment copying the nodes of a Range.

### Syntax

```
documentFragment = range.cloneContents();
```

### Parameters

`documentFragment` is a document fragment

### Example

```
range = document.createRange();
range.selectNode(document.getElementsByTagName("div").
item(0));
documentFragment = range.cloneContents();
document.body.appendChild(documentFragment);
```

### Notes

Event Listeners added using DOM Events are not copied during cloning. HTML attribute events are duplicated as they are for the DOM Core `cloneNode` method. HTML `id` attributes are also cloned, which can lead to an invalid document through cloning.

Partially selected nodes include the parent tags necessary to make the document fragment valid.

### Specification

*cloneContents*

# deleteContents

Removes the contents of a Range from the document.

## Syntax

```
range.deleteContents()
```

## Parameters

None.

## Example

```
range = document.createRange();
range.selectNode(
  document.getElementsByTagName("div").item(0));
range.deleteContents();
```

## Notes

Unlike `extractContents`, this method does not return a documentFragment containing the deleted content.

## Specification

*deleteContents*

# extractContents

Moves contents of a Range from the document tree into a document fragment.

### Syntax

```
documentFragment = range.extractContents();
```

Parameters

*documentFragment* is a document fragment

### Example

```
range = document.createRange();
range.selectNode(document.getElementsByTagName("div").i
tem(0));
documentFragment = range.extractContents();
document.body.appendChild(documentFragment);
```

### Notes

Event Listeners added using DOM Events are not retained during extraction. HTML attribute events are retained or duplicated as they are for the DOM Core cloneNode method. HTML id attributes are also cloned, which can lead to an invalid document if a partially-selected node is extracted and appened to the document.

Partially selected nodes are cloned to include the parent tags necessary to make the document fragment valid.

### Specification:

*extractContents*

# insertNode

Insert a node at the start of a Range.

### Syntax

```
range.insertNode(newNode);
```

### Parameters

*newNode* is a Node.

### Example

```
range = document.createRange();
newNode = document.createElement("p"):
newNode.appendChild(document.createTextNode("New Node
Inserted Here"));
range.selectNode(document.getElementsByTagName("div").i
tem(0));
range.insertNode(newNode);
```

### Notes

newNode is inserted at the start boundary point of the Range. If the newNodes is to be added to a text Node, that Node is split at the insertion point, and the insertion occurs between the two text Nodes (Blocked by http://bugzilla.mozilla.org/show_bug.cgi?id=135922 )

If newNode is a document fragment, the children of the document fragment are inserted instead.

### Specification

*insertNode*

# surroundContents

Moves content of a Range into a new node.

### Syntax

```
range.surroundContents(newNode);
```

### Parameters

*newNode* a Node

### Example

```
range = document.createRange();
newNode = document.createElement("p"):
range.selectNode(document.getElementsByTagName("div").item(0));
range.surroundContents(newNode);
```

### Notes

surroundContents is equivalent to newNode.appendChild(range.extractContents());range.insertNode(newNode). After surrounding, the boundary points of the Range include newNode. (Hindered by http://bugzilla.mozilla.org/show_bug.cgi?id=135928 )

### Specification

*surroundContents*

# compareBoundaryPoints

Compares the boundary points of two Ranges.

### Syntax

```
compare = range.compareBoundaryPoints(how,sourceRange);
```

### Parameters

The **compareBoundaryPoints** method takes the following parameters:

| | |
|---|---|
| *compare* | A number, 1, 0, -1. |
| *how* | A constant describing the comparison method |
| *sourceRange* | A Range to boundary points with range |

### Example

```
range = document.createRange();
range.selectNode(
  document.getElementsByTagName("div").item(0));
sourceRange = document.createRange();
sourceRange.selectNode(document.getElementsByTagName(
  "div").item(1));
compare = range.compareBoundaryPoints(
  START_TO_END,sourceRange);
```

### Notes

`END_TO_END` compares the end boundary-point of *`sourceRange`* to the end boundary-point of range.

`END_TO_START` compares the end boundary-point of *`sourceRange`* to the start boundary-point of range.

`START_TO_END` compares the start boundary-point of *`sourceRange`* to the end boundary-point of range.

`START_TO_START` compares the start boundary-point of *`sourceRange`* to the start boundary-point of range.

### Specification

*compareBoundaryPoints*

# cloneRange

Returns a Range object with boundary points identical to the cloned Range.

### Syntax

```
clone = range.cloneRange();
```

### Parameters

*clone* is a Range object.

### Example

```
range = document.createRange();
range.selectNode(
  document.getElementsByTagName("div").item(0));
clone = range.cloneRange();
```

### Notes

*clone* is copied by value, not reference, so a change in either Range does not effect the other.

### Specification

*cloneRange*

# detach

Releases Range from use to improve performance.

### Syntax

```
range.detach();
```

### Parameters

None.

### Example

```
range = document.createRange();
range.selectNode(
  document.getElementsByTagName("div").item(0));
range.detach();
```

### Notes

Allows mozilla to relinquish resources associated with this Range. Subsequent attempts to use the detached range will result in a `DOMException` being thrown with an error code of `INVALID_STATE_ERR`.

### Specification

*detach*

# toString

Returns the text of the Range.

### Syntax

```
text = range.toString();
```

### Parameters

`text` is the text contained in range.

### Example

```
range = document.createRange();
range.selectNode(
  document.getElementsByTagName("div").item(0));
text = range.toString();
```

### Notes

Alerting the contents of a Range makes an implicit **toString()** call, so comparing range and text through an alert dialog is ineffective

### Specification

*toString*

## Gecko Range Interface Extensions

This section describes Range methods that are particular to Mozilla and not part of the W3 DOM specifications.

### Methods

| | |
|---|---|
| ***compareNode*** | Returns a constant. |
| ***comparePoint*** | Returns -1, 0, or 1. |
| ***createContextualFragment*** | Returns a document fragment. |
| ***intersectsNode*** | Returns a boolean indicating whether the given point intersects the range. |
| ***isPointInRange*** | Returns a boolean indicating whether the given point is in the range. |

# compareNode

Returns a constant (see notes)

### Syntax

```
returnValue = range.compareNode( referenceNode );
```

### Paramaters

### Example

```
range = document.createRange();
range.selectNode(document.getElementsByTagName("div").item(0));
returnValue =
range.compareNode(document.getElementsByTagName("p").item(0));
```

### Notes

| | |
|---|---|
| `NODE_BEFORE = 0` | Node starts before the Range |
| `NODE_AFTER = 1` | Node starts after the Range |
| `NODE_BEFORE_AND_AFTER = 2` | Node starts before and ends after the Range |
| `NODE_INSIDE = 3` | Node starts after and ends before the Range, i.e. the Node is completely selected by the Range. |

### Specification

This method is not part of specification.

# comparePoint

comparePoint()

Returns -1, 0, or 1

### Syntax

```
returnValue = range.comparePoint( referenceNode, offset )
```

### Parameters

| | |
|---|---|
| `returnValue` | -1, 0, or 1 depending on whether the referenceNode is before, the same as, or after the range. |
| `referenceNode` | The Node to compare with the Range. |
| `offset` | An integer greater than or equal to zero representing the offset inside the referenceNode. |

### Example

```
range = document.createRange();
range.selectNode(document.getElementsByTagName("div").item(0));
returnValue =
range.comparePoint(document.getElementsByTagName("p").item(0),1);
```

### Notes

If the `referenceNode` is a Node of type `Text`, `Comment`, or `CDATASection`, then offset is the number of characters from the start of referenceNode. For other Node types, offset is the number of child nodes between the start of the referenceNode.

### Specification

This method is not part of a specification.

# createContextualFragment

Returns a document fragment

### Syntax

documentFragment = range.createContextualFragment( tagString )

### Parameters

tagString is text that contains text and tags to be converted to a document fragment

### Example

tagString = " I am a div node ";

range = document.createRange();

range.selectNode(document.getElementsByTagName("div").item(0));

documentFragment = range.createContextualFragment(tagString);

document.body.appendChild(documentFragment);

### Notes

This method takes a string and uses mozilla's parser to convert it to a DOM tree.

### Specification

This method is not part of a specification.

# intersectsNode

Returns a boolean indicating whether the given point intersects the range.

### Syntax

```
bool = range.intersectsNode( referenceNode )
```

### Parameters

*bool* is true if the `referenceNode` intersects the Range, false if not.

*referenceNode* is the Node to compare with the Range.

### Example

```
range = document.createRange();
range.selectNode(document.
  getElementsByTagName("div").item(0));
bool = range.intersectsNode(document.
  getElementsByTagName("p").item(0),1);
```

### Notes

None

### Specification

This method is not part of a specification.

# isPointInRange

Returns a boolean indicating whether the given point is in the range.

### Syntax

```
bool = range.intersectsNode( referenceNode )
```

### Parameters

*bool* is true if the `referenceNode` intersects the Range, false if not.

*referenceNode* is the Node to compare with the Range.

### Example

```
range = document.createRange();
range.selectNode(document.getElementsByTagName("div").item(0));
bool = range.intersectsNode(document.
   getElementsByTagName("p").item(0),1);
```

### Notes

None

### Specification

This method is not part of a specification.

# DOM Examples

This chapter provides some longer examples of web and XML development using the DOM. Wherever possible, the examples use common APIs, tricks, and patterns in JavaScript for manipulating the document object.

# Example 1: height and width

The following example shows the use of the **height** and **width** properties alongside images of varying dimensions:

```
<html>
<head>
<title>image example</title>
<script language="javascript">
function init()
{
  var image = new Array();
  image[0] = document.getElementById('image1');
  image[1] = document.getElementById('image2');
  image[2] = document.getElementById('image3');
  var output = document.getElementById('output');
  var html   = '';
  var i;
  html = '<ul>';
  for (i = 0; i < image.length; i++)
      html += '<li>image' + (i+1) + ': height=' +
image[i].height + ', width=' + image[i].width + ',
style.height=' + image[i].style.height + ', style.width=' +
image[i].style.width + '</li>';

  html += '</ul>';

  output.innerHTML = html;
}

</script>
</head>
<body onload="init()">

<p>Image 1: no height, width, or style <img id="image1"
src="http://www.mozilla.org/images/mozilla-banner.gif"></p>
<p>Image 2: height=50, width=500, but no style <img
id="image2" src="http://www.mozilla.org/images/mozilla-
banner.gif" height="50" width="500"></p>
<p>Image 3: no height, width, but style="height: 50px; width:
```

```
500px;" <img id="image3" src="http://www.mozilla.org/images/
mozilla-banner.gif" style="height: 50px; width:500px;"></p>

<div id="output">
</div>
```

**height** and **width** are also properties of the `EMBED`, `OBJECT`, and `APPLET` objects.

# Example 2: Image Attributes

```
<html>
<head>
<title>border</title>
<script TYPE="text/javascript">
<!--
function border1(){document.IMG.setAttribute('border',20)}
function border2(){document.IMG.setAttribute('border',5)}
//-->
</script>
<style TYPE="text/css">
<!--
BODY { background-color: #ffffff }
//-->
</style>
</head>
<body>
*border
<hr>
<p>
<img SRC="image1.gif" ID="IMG" border="5" WIDTH="100"
```

```
HEIGHT="100" ALT="border">
</p>
<form NAME="FORM">
<input TYPE="button" VALUE="Change" onClick="border1()">
<input TYPE="button" VALUE="Return" onClick="border2()">
</form>
<hr>
<p>
<font SIZE=-1>
Made by <a HREF="http://www.bekkoame.ne.jp/~hamba/webimage/
java/java.html">MasahitoHamba</a>(<a
HREF="mailto:hamba@bekkoame.ne.jp">hamba@bekkoame.ne.jp</
a>).<br>
It's free to copy and arrange this sample.But please keep to
regulation of <a HREF="http://cgi.din.or.jp/~hagi3/
JavaScript/JSTips/Mozilla/MDSProject.htm"
TARGET=msg>MDSProject</a>. <br>Last update 2001.01.14</font>
</body>
</html>
```

# Example 3: Manipulating Styles

In this simple example, some basic style properties of an HTML paragraph element are accessed using the `style` object on the element and that object's CSS style properties, which can be retrieved and set from the DOM. In this case, you are manipulating the individual styles directly. In the next example (see *Example 4*), you can use stylesheets and their rules to change styles for whole documents.

```
<html>
<head>
<script>
function changeText() {
p = document.getElementById("pid");
p.style.color = "blue"
p.style.fontSize = "18pt"
}

</script>
</head>
<body>
<p id="pid"
onclick="window.location='http://www.cnn.com';" >linker</p>
<form>
<input value="rec" type="button" onclick="changeText();" />
</form>
</body>
</html>
```

# Example 4: Using Stylesheets

The **styleSheets** property on the `document` object returns a list of the stylesheets that have been loaded on that document. You can access these stylesheets and their rules individually using the `stylesheet`, `style`, and `CSSRule` objects, as demonstrated in this example, which prints out all of the style rule selectors to the console.

```
ss = document.styleSheets;
for(ii=0;ii<ss.length;ii++) {
for(i=0;i<ss[0].cssRules.length;i++) {
   dump( ss[ii].cssRules[i].style.selectorText + "\n" );
}
```

For a document with a single stylesheet in which the following three rules are defined:

```
BODY { background-color: darkblue; }
P { font-face: Arial; font-size: 10pt; margin-left: .125in; }
#lumpy { display: none; }
```

This script outputs the following:

```
BODY
P
#LUMPY
```

# Example 5: Event Propagation

This example demonstrates how events fire and are handled in the DOM in a very simple way. When the `BODY` of this `HTML` document loads, an event listener is registered with the top row of the `TABLE`. The event listener handles the event by executing the function `l_func`, which changes the value in the bottom cell of the table.

However, `l_func` also calls an event object method, *stopPropagation*, which keeps the event from bubbling any further up into the DOM. Note that the table itself has an `onclick` event handler that ought to display a message when the table is clicked. But the `l_func` method has stopped propagation, and so after the data in the table is updated, the event phase is effectively ended.

```
<html>
<head>
  <style>
    #t-daddy { border: 1px solid red }
    #t1 { background-color: pink; }
  </style>
  <script>
  function l_func(e) {
    t2 = document.getElementById("t2");
    t2.innerHTML = "three";
    e.stopPropagation();
    // this ought to keep t-daddy from getting the click.
  }

  function load() {
    el = document.getElementById("t");
    el.addEventListener("click", l_func, false);
  }
  </script>
</head>
<body onload="load();">
<table id="t-daddy" onclick="alert('hi');">
  <tr id="t">
     <td id="t1">one</td>
  </tr>
  <tr><td id="t2">two</td></tr>
```

```
</table>
</body>
</html>
```

# Example 6: getComputedStyle

This example demonstrates how the DOM `document.defaultView.getComputedStyle()` method can be used to get the styles on an element that *aren't* set in-line or with JavaScript (e.g., `element.style.backgroundColor="lightblue"`). These latter types of styles can be retrieved with the more direct *`style`* `= element.style` property, a list of which properties is listed in the *DOM Style Reference* of this book (see *DOM CSS Properties List*). See also the *style* property in the *DOM Elements Reference*.

**getComputedStyle()** returns a `ComputedCSSStyleDeclaration` object, whose individual style properties can be referenced with this object's `getPropertyValue()` method, as the following example document shows.

```
<html>
<head>
  <title>getComputedStyle</title>
  <script>
   function cStyles() {
     div = document.getElementById("d1");

     t1 = document.getElementById("t1");
     h_style = document.defaultView.getComputedStyle(div, '').getPropertyValue("height");
     t1.setAttribute("value", h_style);

     t2 = document.getElementById("t2");
     w_style = document.defaultView.getComputedStyle(div, '').getPropertyValue("width");
     t2.setAttribute("value", w_style);

     t3 = document.getElementById("t3");
     b_style = document.defaultView.getComputedStyle(div,
'').getPropertyValue("background-color");
     t3.setAttribute("value", b_style);
  }
```

```
  </script>
  <style>
    .d { margin-left: 10px; background-color: lightblue; height: 20px; max-width: 20px; }
  </style>
</head>

<body>
<div id="d1" class="d"> </div>
<p> 

<blockquote>
  <button onclick="cStyles();">getComputedStyle</button>
  height<input id="t1" type="text" value="1"  />
  max-width<input id="t2" type="text" value="2"  />
  bg-color<input id="t3" type="text" value="3"  /></pre>
</blockquote>

</body>
</html>
```

# Example 7: Displaying Event Object Constants

This example shows how to use the DOM to create a table in which all of the constants in the `event` object and their values are displayed. It shows off several useful aspects of the DOM, including the `Event.prototype` property, which allows you to get to the properties of a particular object, a good pattern for iterating over the properties in that prototype, and the values of the constants themselves displayed in the table. Note that the middle range of these constants are the character codes that represent the actual keys pressed during the event (and fetchable with the *charCode* property).

Load the following code as a web page to see the `event` object constants.

```
<?xml version="1.0" ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "DTD/xhtml1-
transitional.dtd">
```

```
<!--
 * ***** BEGIN LICENSE BLOCK *****
 * Version: NPL 1.1/GPL 2.0/LGPL 2.1
 *
 * The contents of this file are subject to the Netscape Public License
 * Version 1.1 (the "License"); you may not use this file except in
 * compliance with the License. You may obtain a copy of the License at
 * http://www.mozilla.org/NPL/
 *
 * Software distributed under the License is distributed on an "AS IS" basis,
 * WITHOUT WARRANTY OF ANY KIND, either express or implied. See the License
 * for the specific language governing rights and limitations under the
 * License.
 *
 * Contributor(s):
 *   Alexander J. Vincent <jscript@pacbell.net>
 *
 * Alternatively, the contents of this file may be used under the terms of
 * either the GNU General Public License Version 2 or later (the "GPL"), or
 * the GNU Lesser General Public License Version 2.1 or later (the "LGPL"),
 * in which case the provisions of the GPL or the LGPL are applicable instead
 * of those above. If you wish to allow use of your version of this file only
 * under the terms of either the GPL or the LGPL, and not to allow others to
 * use your version of this file under the terms of the NPL, indicate your
 * decision by deleting the provisions above and replace them with the notice
 * and other provisions required by the GPL or the LGPL. If you do not delete
 * the provisions above, a recipient may use your version of this file under
 * the terms of any one of the NPL, the GPL or the LGPL.
 *
 * ***** END LICENSE BLOCK ***** * -->

<html xmlns="http://www.w3.org/1999/xhtml" >
<head><title></title>
<script language="JavaScript" type="text/javascript">
<!--
function respond() {
```

```
// creating a table
    var table = document.createElement("table")
    table.setAttribute("border", "1")
    var tbody = document.createElement("tbody")

    var y = 0
    var tr = document.createElement("tr")
    var td = document.createElement("th")
// reusing the same variable name in the loop

// begin table heading information
    td.appendChild(document.createTextNode("Index"))
    tr.appendChild(td)

    td = document.createElement("th")
    td.appendChild(document.createTextNode("Property Name"))
    tr.appendChild(td)

    td = document.createElement("th")
    td.appendChild(document.createTextNode("Property Value"))
    tr.appendChild(td)

    tbody.appendChild(tr)
// end table heading information

    for (property in Event.prototype) {
        if (property == property.toUpperCase()) {
// adding a new row for each property of the event object
            tr = document.createElement("tr")
            td = document.createElement("td")
            td.appendChild(document.createTextNode(y))
// which property number it happens to be
            tr.appendChild(td)
            y++

            td = document.createElement("td")
```

```
            var td_text = document.createTextNode(property)
// the property name
            td.appendChild(td_text)
            tr.appendChild(td)

            td = document.createElement("td")
            var td_text = document.createTextNode(Event.prototype[property])
// the property value
            td.appendChild(td_text)
            tr.appendChild(td)

            tbody.appendChild(tr)
            }
        }
    table.appendChild(tbody)
    document.body.appendChild(table)
    }
//-->
</script>
</head>

<body onload="respond()">
<!-Results after clicking on the button:
The this object is myInput.
Index    Property Name Property Value
0        type          click
1        target        [object HTMLInputElement]
...
-->
</body>
</html>
```

# Index

D