Using Web Standards in Your Web Pages

> "Browser makers are no longer the problem. The problem lies with designers and developers chained to the browser-quirk-oriented markup of the 1990s-often because they don't realize it is possible to support current standards while accommodating old browsers."
>
> *Web Standards Project*

A number of elements and practices for adding *DHTML* to web pages were excluded from the *W3C* HTML 4.01 and Document Object Model (DOM) specifications. Elements like <LAYER> and collection of objects like `document.layers[]` (Netscape 4) or `document.all` (Internet Explorer 5+), for example, are actually not a part of any web standard. Browsers that comply with the W3C web standards, such as Firefox, Mozilla and Netscape 6/7, do not support these non-compliant elements and these proprietary *DOM* collections.

This article provides an overview of the process for upgrading the content of your web pages to conform to the W3C web standards. The various sections identify some practices which are at odds with the standards and suggest replacements. **Every proposed web standards replacement in this article has been tested, verified and is working without a problem in modern browsers like MSIE 6, Netscape 7.x, Firefox 1.x, Opera 7+, Safari 1.2+, Konqueror 3.x, etc.** The final section, Summary of Changes, outlines all the changes described in this article.

In this document:

Upgrading Layer Elements (Netscape 4)

This section explains how to replace Netscape 4 `<layer>` and `<ilayer>` elements with standards-compliant

HTML 4.01. Because <layer> and <ilayer> elements are not part of any W3C web standards, **Netscape 6/7, Firefox and Mozilla and other browsers that comply with the W3C web standards do not support <layer> and <ilayer> elements**.

The document.layers[] collection of objects and other specific features of the Netscape 4 Layer DOM are not supported either and are discussed the DOM section below.

In Netscape 4, <layer> elements are used primarly for 2 purposes:

- to embed external HTML content inside a webpage and
- to position a defined block of HTML content; such block of HTML content is usually named, referred as layer or DHTML layer by web authors, books and references.

Replacing <layer> and <ilayer> as embedded external HTML content

If you have:

```
<LAYER SRC="foo.html" height="300" width="400"> </LAYER>
```

... then you can for HTML 4.01 Transitional documents replace it with:

```
<iframe src="foo.html" height="300" width="400">
  <a href="foo.html">Foo content</a>
</iframe>
```

User agents and very old visual browsers which do not support IFRAME (like Netscape 4) will render its content: here, it is a link. In this manner, accessibility to content (content degradation) for older browsers is assured and is as graceful as it can be.

... or, for HTML 4.01 Strict documents, you can replace it with:

```
<object data="foo.html" type="text/html" height="300" width="400">
  <a href="foo.html">Foo content</a>
</object>
```

Again, the link will be rendered in user agents and browsers which do not support the object element, therefore assuring access to content.

The general accessibility strategy when using <iframe> or <object> is to embed the most common and most supported element inside <iframe> or <object>: that way, an user agent which is not able to render the <iframe> or <object> will render its content serving it as an alternative. The general rule applied by most browsers when meeting an unknown element is to render its content as best as it can. Note 1

More on embedding HTML content:
Notes on embedded documents from W3C HTML 4.01
Note 1 "If a user agent encounters an element it does not recognize, it should try to render the element's content.": *Notes on invalid documents*

Replacing <layer> as positioned block of HTML content

To upgrade positioned <layer> elements, the best W3C web standards compliant replacement is to use . A element can not transclude, can not import HTML content external to the webpage; so, defining a src attribute in a element will be ignored by W3C compliant browsers.

If you have

```
<LAYER style="position: absolute;" top="50" left="100"
width="150" height="200">
   ... content here ...
</LAYER>
```

then you can replace it with:

```
<div style="position: absolute; top: 50px; left: 100px;
width: 150px; height: 200px;">
   ... content here ...
</div>
```

Deprecated elements

Elements deprecated in HTML 4.01 are typically in wide use, but have been supplanted by other techniques.

The function of several of the deprecated tags (and of some excluded tags, as well) has been assumed by the W3C *Cascading Style Sheets* recommendation. Style sheets provide powerful presentation and organization capabilities. A full discussion of CSS is beyond the scope of this document.

APPLET

The APPLET element has been deprecated in HTML 4.01 in favor of OBJECT.

```
<p>
<applet code="HelloWorldApplet.class" height="200" width="350"></applet>
</p>
```

can be converted to:

```
<p>
<object classid="clsid:8AD9C840-044E-11D1-B3E9-00805F499D93"
codebase=
"http://java.sun.com/products/plugin/autodl/jinstall-1_4_2-windows-i586.
cab#Version=1,4,2,0"
codetype="application/java" standby="Loading of applet in progress..."
height="200" width="350">
<param name="code" value="HelloWorldApplet.class">
<!--[if !IE]>
Mozilla 1.x, Firefox 1.x, Netscape 7.x and others will use the inner
```

```
object, the nested object
-->
     <object classid="java:HelloWorldApplet.class"
     standby="Loading of applet in progress..."
     height="200" width="350">
     <p>Your browser does not seem to have java support enabled
     or it does not have a Java Plug-in.<br>
     <a href="http://www.java.com/en/download/manual.jsp">You can download
     the latest Java Plug-in here. (free download; 15MB)</a></p>
     </object>
<!--<![endif]-->
</object>
</p>
```

The above code will work for MSIE 6, Mozilla-based browsers and other standards-based browsers; also, it will validate in either HTML 4.01 transitional or HTML 4.01 strict.

Explanations on the code:

According to HTML 4.01 recommendation, when an <object> is not rendered (because its content type is unsupported e.g. the browser does not support the ActiveX control called, requested by the <object>), then the browser should render its contents instead: here, it is another <object>, an alternate <object>. Here, the inner <object> will be rendered by browsers not supporting java plug-in triggered by an ActiveX.

classid="clsid:8AD9C840-044E-11D1-B3E9-00805F499D93":
this clsid value will make the MSIE 6 browser use the highest possible version (installed on the user's machine) of JRE. Sometimes, MSIE 6 users have several JRE plug-in versions installed.

codebase=
"http://java.sun.com/products/plugin/autodl/jinstall-1_4_2-windows-i586.cab#Version=1,4,2,0":
the codebase defines the minimum version for the JRE; here it is 1.4.2. In case the browser does not have a java plug-in or if its version is earlier than 1.4.2, then an automatic download of the latest 1.4.2 version will start. This may be a debatable choice: on one hand, the latest available JRE plug-in has **several security patches** and bug fixes, on the other hand, forcing a 15MB download without a prior explicit consent of the user can not be best.

More on applet to object conversion:

- *Java applet using <object> tag* by Shayne Steele
- *HTML 4.01 on Including an applet*
- *HTML 4.01 on Including an object*

FONT

The deprecated FONT element is widely used to specify typeface, color and size of the enclosed text. This functionality has been offloaded from HTML to CSS. The FONT element can be directly replaced with a SPAN element that includes the same style information:

```
<P><FONT color="blue" face="Helvetica">
A really <FONT size="+1">big</FONT> shoe.
</FONT></P>
```

... becomes:

```
<P><SPAN style="color:blue; font-family: Helvetica, sans-serif;">
A really <SPAN style="font-size: larger;">big</SPAN> shoe.
</SPAN></P>
```

... or even more concisely:

```
<P style="color: blue; font-family: Helvetica, sans-serif;">
A really <SPAN style="font-size: larger;">big</SPAN> shoe.
</P>
```

This is appropriate usage for a local change to the font. However, this is not the best use of styles; the strength of CSS lies in the ability to gather text and other styling into logical groupings that can be applied across a document, without repeating the specific styling on every element that requires it.

More on conversion of <FONT>:
W3C Quality Assurance tip for webmaster:
Care With Font Size, Recommended Practices: Forget <font>, use CSS

Other deprecated elements

Deprecated Element and Attribute	W3C Replacement
CENTER or align="center"	CSS1: *text-align*: center The text-align property specifies how text or inline elements are aligned within an element.
CENTER or align="center"	CSS1: margin-left: auto; margin-right: auto; for block-level elements When both margin-left and margin-right are auto they are set to equal values, thus centering a block-level element within its parent. *CSS1 horizontal formating* Worth mentioning is the excellent tutorial: *Centring using CSS* by D. Dorward
bgcolor attribute	CSS1: background-color: ; *CSS1 background-color*
S STRIKE	CSS1: text-decoration: line-through;
U	CSS1: text-decoration: underline;
DIR MENU	HTML 4.01: <UL>

Other Excluded Elements

There are a number of proprietary elements used for animation and other tricks that are not a part of any web standard. This section highlights those elements and suggests practices for achieving the same effect with W3C HTML 4.01:

Excluded Element	W3C Replacement
BLINK *Nav2+*	CSS1 `text-decoration: blink;` User agents are required by the CSS1 specification to recognize the blink keyword, but not to support the blink effect, so CSS1-compliant browsers may or may not make the text blink on the screen. The best approach is not to make content blink at all.
MARQUEE *IE2+*	HTML 4.01 `DIV` or `SPAN`, with content string rotated over time by JavaScript via the DOM level 1. As with blinking text, this sort of effect is discouraged. Studies have shown that constantly moving objects or moving text disturb reading and weakens peripherical vision. DHTML marquee also greatly consumes user system resources (cpu, RAM) on tested browsers and will put modest systems under considerable stress. If after webpage assessment and consideration, you still want to include a marquee effect in your page, then you can use the following tutorials: *Cross-browser and web standard compliant Stock Ticker example* by D. Rosenberg *Comprehensive web standard compliant alternative to <marquee>* by D. Rosenberg Mozilla 1.4+, NS 7.2 and Firefox 1.x support the non-standard <marquee> element. On the other hand, users can disable such support using this tip
BGSOUND *IE2+*	HTML 4.01 `OBJECT`, e.g.: `<OBJECT data="audiofile.wav" type="audio/wav" ...></OBJECT>` See *this DevEdge article* for information on rendering a sound OBJECT invisible within the page. Web page background sound often slows down considerably web page loading; like the text effects above, music or sound accompanying a page is seldom appreciated. According to the survey page *What we really hate on the web*, 41.9% of survey respondents will avoid sites that automatically play music; 71.1% strongly dislike sites that automatically play music. *Why Playing Music on your Web Site is a Bad Idea* by A. Gulez
EMBED *Nav2+,IE3+*	HTML 4.01 `OBJECT`. See *this DevEdge article* for information on translating `EMBED` tags into `OBJECT` tags. `EMBED` has *never* been part of a W3C HTML recommendation, yet it is still supported by Gecko and other modern browsers. Quality of support varies; Internet Explorer's support is incompatible with most Netscape plug-ins. Support for `OBJECT` is not universal, either, particularly for older browsers.

Using the W3C DOM

The `document` objects for some browsers have properties for accessing arrays of elements and types of elements. `document.all[]`, for example, is used by Internet Explorer to access particular elements within the document. Many of these arrays were not made a part of the W3C specification for the Document Object Model and will cause JavaScript errors in standards-compliant browsers like Mozilla, Firefox and Netscape 6/7.

The *W3C Document Object Model* exposes almost all of the elements in an HTML page as scriptable objects. In general the attributes and methods of the W3C DOM are more powerful than the proprietary object models used in DHTML programming. **The attributes and methods of the W3C DOM are overall well supported by modern browsers like MSIE 6, Opera 7+, Safari 1.x, Konqueror 3.x and Mozilla-based browsers (Firefox, Mozilla, Netscape 6/7): so there is no gain from relying on proprietary object models**.

Unsupported DOM-related Properties

The following `document` object properties are not supported in the W3C Document Object Model:

- document.layers[]
- document.*elementName*
  (i.e., getting a reference to the element <p **name**="yooneek"> with document.yooneek)
- id_attribute_value
- document.all.id_attribute_value
- document.all[id_attribute_value]

The following element properties (originally from Internet Explorer) are not supported in the W3C Document Object Model:

- FormName.InputName.value
- document.forms(0)
- *element*.innerText

Scripts that use these properties will not execute in Firefox, Mozilla and Netscape 6/7 or other standards-compliant browsers. Instead, use the W3C DOM access attributes and access methods described in the next section; since these are supported by Internet Explorer too, then there is no need to use MSIE-specific attributes and methods.

Accessing Elements with the W3C DOM

The best and most supported practice for getting scriptable access to an element in an HTML page is to use document.getElementById(*id*). All modern browsers (NS 6+, Mozilla, MSIE 5+, Firefox, Opera 6+, Safari 1.x, Konqueror 3.x, etc.) support document.getElementById(*id*). This method returns an object reference to the uniquely identified element, which can then be used to script that element. For example, the following short sample dynamically sets the left margin of a div element with an id of "inset" to half an inch:

```
// in the HTML: <div id="inset">Sample Text</div>
document.getElementById("inset").style.marginLeft = ".5in";
```

IE-specific ways to access elements	W3C web standards replacements
id_attribute_value	document.getElementById(id_attribute_value)
document.all.id_attribute_value	document.getElementById(id_attribute_value)
document.all[id_attribute_value]	document.getElementById(id_attribute_value)
FormName.InputName.value	document.forms["FormName"].InputName.value or document.forms["FormName"].elements["InputName"].value
document.forms(0)	document.forms[0]

More on accessing forms and form elements:
Referencing Forms and Form Controls by comp.lang.javascript newsgroup FAQ notes
DOM 1 specification on accessing forms and form elements

For accessing a group of elements, the DOM specification also includes getElementsByTagName, which returns a list of all the elements with the given tag name in the order they appear in the document:

```
var arrCollection_Of_Anchors = document.getElementsByTagName("a");
var objFirst_Anchor = arrCollection_Of_Anchors[0];
alert("The url of the first link is " + objFirst_Anchor.href);
```

In addition to these access methods, the W3C DOM2 specifications provide methods for creating new elements and inserting them in a document, for creating attributes, new content, for traversing the content tree and for handling events raised as the user interacts with the document itself.

Manipulating Document Style and Content

Changing an Element's Style Using the DOM

The following table describes standards-based methods for accessing and updating style rules defined for various HTML elements in a web page. See the W3C DOM2 Recommendation, *CSS2 Extended Interface*.

DOM level 2 provides for the assignment of new values to the CSS properties of an element using the *element*.style object reference. You can get the element to which that style corresponds by using the DOM's getElementById or one of the other methods described in the DOM access section above.

Deprecated coding practices	Appropriate DOM 2 replacements
Nav4: *element*.visibility = *value*;	DOM level 2: *element*.style.visibility = *value*;
Nav4: *element*.left IE4/5: *element*.style.pixelLeft	DOM level 2: parseInt(*element*.style.left, 10)
Nav4: *element*.top IE4/5: *element*.style.pixelTop	DOM level 2: parseInt(*element*.style.top, 10)
Nav4: *element*.moveTo(*x*,*y*); IE4/5: *element*.style.pixelLeft = *x*; *element*.style.pixelTop = *y*;	DOM level 2: *element*.style.left = *x* + "px"; *element*.style.top = *y* + "px";

W3C DOM2 Reflection of an Element's CSS Properties

Keep in mind that according to the W3C Recommendation, the values returned by the style property of an element reflect static settings in the element's STYLE attribute only, not the total "computed style" that includes any inherited style settings from parent elements. Therefore, if you wish to read and write these properties from JavaScript through the DOM2, use one of these two approaches:

- Place all of the element's static CSS declarations (if it has any) in the element's STYLE attribute.
- Use no static CSS declarations for the element and initialize its CSS properties from JavaScript through the DOM.

W3C DOM2 Reflection of an Element's CSS Positioning Properties

The values returned by the W3C DOM2 style.left and style.top properties are strings that include the CSS

unit suffix (such as "px"), whereas Netscape 4 *element*.left and IE4/5 *element*.style.pixelLeft (and the corresponding properties for top) return an integer. So, if you want to get the element's inline STYLE settings for left and top as integers, parse the integer from the string by using parseInt(). Conversely, if you want to set the element's inline STYLE settings for left and top, make sure to construct a string that includes the unit (such as "140px") by appending the unit string to the integer value.

CSS1 and CSS 2.x specifications require that non-zero values must be specified with a length unit; otherwise, the css declaration will be ignored. Mozilla-based browsers, MSIE 6, Opera 7+ and other W3C standards-compliant browsers enforce such handling of parsing error.
CSS1 Forward-compatible parsing
CSS2.1 Rules for handling parsing errors

Changing an Element's Text Using the DOM

Changing the actual text content of an element has changed substantially compared to the normal means of operation. Each element's content is broken up into a set of child nodes, consisting of plain text and sub-elements. In order to change the text of the element, the script operates on the node.

The node structure and supporting methods are defined in the W3C *DOM level 1* recommendation.

If the element has no sub-elements, just text, then it (normally) has one child node, accessed as element.childNodes[0]. In such precise case, the W3C web standards equivalent of element.innerText is element.childNodes[0].nodeValue.

The following examples show how to modify the text of a SPAN element that already exists in the HTML file.

```
<body>
  <P>Papa's got <SPAN id="dynatext">a lot of nerve</SPAN>!</P>

  <script type="text/javascript">
    // get reference to the SPAN element
    var span_el = document.getElementById("dynatext");

    // implement span_el.innerText = "a brand new bag"
    var new_txt = document.createTextNode("a brand new bag");
    span_el.replaceChild(new_txt, span_el.childNodes[0]);

    // alternate, slightly more dangerous implementation
    //   (will not work if childNodes[0] is not a text node)
    span_el.childNodes[0].nodeValue = "a brand new bag";

    // implement span_el.innerHTML = "a brand <b>new</b> bag"
    var new_el = document.createElement(span_el.nodeName);
    new_el.appendChild(document.createTextNode("a brand "));
    var bold_el = document.createElement("B");
    bold_el.appendChild(document.createTextNode("new"));
    new_el.appendChild(bold_el);
    new_el.appendChild(document.createTextNode(" bag"));
    span_el.parentNode.replaceChild(new_el, span_el);
  </script>
</body>
```

The first example shows the relatively simple method of replacing the text in the SPAN by substituting a new

text node for the original. The assumption here is that the SPAN has a single, text child; the code would work even if that were not true, but the results might be unexpected.

The second example shows a more concise but brute-force technique of accomplishing the same thing. As noted, if the first child node is not a text node, this action will not work: although sub-element nodes have a nodeValue, the contents of that field are not displayed in the document.

The final example shows a technique equivalent to setting innerHTML. First, it constructs a new element, created as the same type (SPAN) as the original. Next it adds three nodes: an initial text node, a B element with its own text node, and a final text node. The script then accesses the SPAN element's parent (the P element), and substitutes the new element for the original SPAN in the parent's list of child nodes.

It should be clear that translating scripts to modify document content is not a trivial undertaking. The benefit of such a conversion is that the script will work in modern, W3C DOM-compliant browsers such as Netscape 6/7 and other Gecko-based programs. Requirements of backward compatibility, however, will not only prolong but worsen the difficulties of dealing with multiple platforms.

Developing Cross Browser/Cross Platform Pages

An important practice when doing cross-browser, cross-platform pages and DHTML development involves the ability to determine the capabilities of the browser which loads your web page. As a web author, you understandably want to avoid script errors and page layout problems and you may want to ensure your scripts reach as wide an audience as possible. There are 2 known approaches for such goals: the browser identification (also known as userAgent string detection and often referred as "browser sniffing") and the Object/Feature support detection. The browser identification approach is now known to be complicated, unreliable and difficult to maintain.

Browser identification (aka "browser sniffing"): not best, not reliable approach

This approach, still commonly used nowadays, attempts to identify the browser and makes the web author at design time decide what that implies in terms of capabilities of the visiting browser. Such approach is fraught with problems and difficulties. It requires from the web author to have knowledge of the capabilities of all current browsers that may visit the page and then to code appropriately for these. It requires from the web author to make assumptions about what will happen with future browsers or to decide to provide future browsers a safe fallback service. It assumes that web authors are able to correctly identify browsers and browser versions in the first place... which is far from being a reliable and easy task to achieve.

The browser identification approach relies on functions that check the browser type string value and browser version string value and that search for certains characters or sub-strings in the navigator.userAgent property string. Once "detected", the web author then uses different functions (aka code branching) or points the user to different pages (aka site branching) or web content. Site branching can be particularly dangerous as people may enter a page through a link, bookmark, search engine or cache with a "wrong" browser.

Let's see a basic example of this approach.

```
if (navigator.appVersion.charAt(0) == "7")
{
   if (navigator.appName == "Netscape")
   {
     isNS7 = true;
```

```
     alert("NS7");
   };
}
else if (navigator.appVersion.indexOf("MSIE") != -1)
   {
    isIE = true;
    alert("IE");
   };
```

While this kind of checking in the above code can work in a crude sense, sharp readers may wonder what happens when IE 7 is released or when an Opera 7.x user visits the page or even when an user with any non-Netscape browser starting with a "7" character in the appVersion string visits that page. As new browsers are released, it becomes necessary to make updates to such code which attempts to narrow down the browser and browser version and to make the appropriate switches.

Another major problem with this approach is that the browser identity can be "spoofed" because, in many modern browsers, the **navigator.appVersion and navigator.userAgent string properties are user configurable strings**. For example,

- Mozilla 1.x uses the preference "general.useragent.override"
- Opera 6+ allows users to set the browser identification string via a menu
- MSIE uses the Windows registry
- Safari and ICab browsers mask their browser identity under Internet Explorer or Netscape labels
- etc..

A user or browser distributor can put what they want in the navigator.userAgent string and this may trick your code into executing a "wrong" block of code. Moreover, there are many cases where even the accurately-identified browser does not perform as it is reputed/expected to.

So if "browser sniffing" is unreliable and difficult, how do you code safely for different browsers and different browser versions? ...

Using Object/Feature detection: best and overall most reliable

When you use object/feature detection, you only implement those features whose support you have first tested and verified on the visiting browser. This method has the advantage of not requiring you to test for anything except whether the particular features you code are supported in the visiting browser.

Let's see a basic, simple object detection example.

```
function hideElement(id_attribute_value)
{
   if (document.getElementById &&
         document.getElementById(id_attribute_value) &&
            document.getElementById(id_attribute_value).style)
   {
   document.getElementById(id_attribute_value).style.visibility="hidden";
   };
}
```

```
// example:
// <button type="button" onclick="hideElement('d1');">hide div</button>
// <div id="d1">Some text</div>
```

These repeated calls to document.getElementById are not the most efficient way to check for the existence of particular objects or features in the browser's DOM implementation, but they serve to illustrate clearly how object detection works.

The top-level if clause looks to see if there's an object called getElementById on the document object, which is the one of the most basic levels of support for the DOM in a browser. If there is, the code sees if getElementById(id_attribute_value) returns an element, which it then checks for a style object. If the style object exists on the element, then it sets that object's visibility property. The browser will not error if you set this unimplemented property, so you do not need to check that the visiblity property itself exists.

So, instead of needing to know which browsers and browser versions support a particular DOM method (or DOM attribute or DOM feature), you can verify the support for that particular method in the visiting browser. With this approach, you ensure that all browsers -- including future releases and browsers whose userAgent strings you do not know about -- will continue working with your code.

More on object/feature detection:
A Strategy That Works: Object/Feature Detecting by comp.lang.javascript newsgroup FAQ notes
Browser detection - No; Object detection - Yes by Peter-Paul Koch

Summary of Changes

This section outlines all of the element and practice updates described in this article. For a complete discussion of these items, see the sections in which they are described.

Proprietary or Deprecated Feature	W3C Feature or Recommended Replacement
NS 4 LAYER as positioned block of HTML content	HTML 4.01 DIV
NS 4 ILAYER	iframe in HTML 4.01 transitional or object in HTML 4.01 strict
NS 4 LAYER SRC=, DIV SRC=	iframe src= in HTML 4.01 transitional or object data= in HTML 4.01 strict
IE2+ MARQUEE	HTML 4.01 DIV plus scripting
Nav2+ BLINK	CSS1 text-decoration: blink;
IE2+ BGSOUND	HTML 4.01 OBJECT
Nav 2+, IE3+ EMBED	HTML 4.01 OBJECT
deprecated APPLET	HTML 4.01 OBJECT
deprecated FONT	HTML 4.01 SPAN plus CSS1 color: ; font-family: ; font-size: ;
deprecated CENTER or align="center"	CSS1 text-align: center; for inline elements

deprecated CENTER or align="center"	CSS1 margin-left: auto; margin-right: auto; for block-level elements
deprecated bgcolor	CSS1 background-color: ;
deprecated U, S, STRIKE	CSS1 text-decoration: underline, line-through;
deprecated DIR, MENU	HTML 4.01 UL
Proprietary or Deprecated Feature	**W3C Feature or Recommended Replacement**
Nav4 document.layers[]	DOM level 2: document.getElementById(*id*)
IE5/6 *id_attribute_value* document.all.*id_attribute_value* document.all[*id_attribute_value*]	DOM level 2: document.getElementById(*id_attribute_value*)
IE5/6 *FormName*.*InputName*.value	DOM level 1: document.forms["*FormName*"].*InputName*.value
IE5/6 document.forms(0)	DOM level 1: document.forms[0]
Nav4 document.layers[*id*].document.write() IE *element*.innerText	DOM Level 1 (Core) interface
Nav4 *element*.visibility = *value*;	DOM level 2: *element*.style.visibility = *value*;
Nav4 *element*.left IE4/5 *element*.style.pixelLeft	DOM level 2: parseInt(*element*.style.left, 10)
Nav4 *element*.top IE4/5 *element*.style.pixelTop	DOM level 2: parseInt(*element*.style.top, 10)
Nav4 *element*.moveTo(*x*, *y*); IE4/5 *element*.style.pixelLeft = *x*; *element*.style.pixelTop = *y*;	DOM level 2: *element*.style.left = *x* + "px"; *element*.style.top = *y* + "px";
Nav4/IE4/5 document.*elementName*	DOM access methods
Proprietary or Deprecated Feature	**W3C Feature or Recommended Replacement**

You can learn more on using web standards from these sites:
What are web standards and why should I use them?
What are the advantages of using web standards? from *Web Standards Project*
Web Quality Assurance tips for webmasters:
My Web site is standard! And yours? from *W3C Quality Assurance*
Making A Commercial Case for Adopting Web Standards
Case Study in a Successful Standards-Based Migration
Web Standards Group
Web Page Development: Best Practices
Mozilla Web Author FAQ

Mike Cowperthwaite, Marcio Galli, Jim Ley, Ian Oeschger, Simon Paquet, Gérard Talbot

---