Migrate apps from Internet Explorer to Mozilla

From MDC

Contents

[hide]

Introduction [edit]

When Netscape started the Mozilla browser, it made the conscious decision to support W3C standards. As a result, Mozilla is not fully backwards-compatible with Netscape Navigator 4.x and Microsoft Internet Explorer legacy code; for example, Mozilla does not support `<layer>` as I will discuss later. Browsers, like Internet Explorer 4, that were built before the conception of W3C standards inherited many quirks. In this article, I will describe Mozilla's quirks mode, which provides strong backwards HTML compatibility with Internet Explorer and other legacy browsers.

I'll also cover nonstandard technologies, such as XMLHttpRequest and rich text editing, that Mozilla does support because no W3C equivalent existed at the time. They include:

- HTML 4.01, XHTML 1.0 and XHTML 1.1
- Cascade Style Sheets (CSS): CSS Level 1, CSS Level 2.1 and parts of CSS Level 3
- Document Object Model (DOM): DOM Level 1, DOM Level 2 and parts of DOM Level 3
- Mathematical Markup Language: MathML Version 2.0
- Extensible Markup Language (XML): XML 1.0, Namespaces in XML, Associating Style Sheets with XML Documents 1.0, Fragment Identifier for XML
- XSL Transformations: XSLT 1.0
- XML Path Language: XPath 1.0
- Resource Description Framework: RDF
- Simple Object Access Protocol: SOAP 1.1
- ECMA-262, revision 3 (JavaScript 1.5): ECMA-262

General cross-browser coding tips [edit]

Even though Web standards do exist, different browsers behave differently (in fact, the same browser may behave differently depending on the platform). Many browsers, such as Internet Explorer, also support pre-W3C APIs and have never added extensive support for the W3C-compliant ones.

Before I go into the differences between Mozilla and Internet Explorer, I'll cover some basic ways you can make a Web application extensible in order to add new browser support later.

Since different browsers sometimes use different APIs for the same functionality, you can often find multiple `if() else()` blocks throughout the code to differentiate between the browsers. The following code shows blocks designated for Internet Explorer:

```
. . .

var elm;

if (ns4)
  elm = document.layers["myID"];
else if (ie4)
  elm = document.all["myID"]
```

The above code isn't extensible, so if you want it to support a new browser, you must update these blocks throughout the Web application.

The easiest way to eliminate the need to recode for a new browser is to abstract out functionality. Rather than multiple `if() else()` blocks, you increase efficiency by taking

common tasks and abstracting them out into their own functions. Not only does this make the code easier to read, it simplifies adding support for new clients:

```
var elm = getElmById("myID");

function getElmById(aID){
  var element = null;

  if (isMozilla || isIE5)
    element = document.getElementById(aID);
  else if (isNetscape4)
    element = document.layers[aID];
  else if (isIE4)
    element = document.all[aID];

  return element;
}
```

The above code still has the issue of *browser sniffing*, or detecting which browser the user is using. Browser sniffing is usually done through the useragent, such as:

```
Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.5) Gecko/20031016
```

While using the useragent to sniff the browser provides detailed information on the browser in use, code that handles useragents often can make mistakes when new browser versions arrive, thus requiring code changes.

If the type of browser doesn't matter (suppose that you have already blocked nonsupported browsers from accessing the Web application), **it is much better and more reliable to sniff by browser capability or object feature support**. You can usually do this by testing the required functionality in JavaScript. For example, rather than:

```
if (isMozilla || isIE5)
```

You would use:

```
if (document.getElementById)
```

This would allow other browsers that support that W3C standard method, such as Opera or Safari, to work without any changes.

Useragent sniffing, however, makes sense when accuracy is important, such as when you're verifying that a browser meets the Web application's version requirements or you are trying to work around a bug.

JavaScript also allows inline conditional statements, which can help with code readability:

```
var foo = (condition) ? conditionIsTrue : conditionIsFalse;
```

For example, to retrieve an element, you would use:

```
function getElement(aID){
  return (document.getElementById) ? document.getElementById(aID)
                                   : document.all[aID]);
}
```

Or another way is to use the || operator:

```
function getElement(aID){
  return (document.getElementById(aID)) || document.all[aID]);
}
```

Differences between Mozilla and Internet Explorer [edit]

First, I'll discuss the differences in the way HTML behaves between Mozilla and Internet Explorer.

Tooltips [edit]

Legacy browsers introduced tooltips into HTML by showing them on links and using the value of the `alt` attribute as a tooltip's content. The latest W3C HTML specification created the `title` attribute, which is meant to contain a detailed description of the link. Modern browsers will use the `title` attribute to display tooltips, and Mozilla only supports showing tooltips for that attribute and not the `alt` attribute.

Entities [edit]

HTML markup can contain several entities, which the W3C web standards body has defined. You can reference entities through their numerical or character reference. For example, you can reference the white space character #160 with ` ` or with its equivalent character reference ` `.

Some older browsers, such as Internet Explorer, had such quirks as allowing you to use entities by replacing the `;` (semi-colon) character at the end with regular text content:

```
  Foo
   Foo
```

Mozilla will render the above ` ` as white spaces, even though that is against the W3C specification. The browser will not parse a ` ` if it is directly followed by more characters, for example:

```
&nbsp12345
```

This code does not work in Mozilla, since it goes against the W3C web standards. Always use the correct form (` `) to avoid browser discrepancies.

DOM differences [edit]

The Document Object Model (DOM) is **the tree structure that contains the document elements**. You can manipulate it through JavaScript APIs, which the W3C has standardized. However, prior to W3C standardization, Netscape 4 and Internet Explorer 4 implemented the APIs similarly. Mozilla only implements legacy APIs if they are unachievable with W3C web standards.

Accessing elements [edit]

To retrieve an element reference using the cross-browser approach, you use `document.getElementById(aID)`, which works in Internet Explorer 5.0+, Mozilla-based browsers, other W3C-compliant browsers and is part of the DOM Level 1 specification.

Mozilla does not support accessing an element through `document.elementName` or even through the element's name, which Internet Explorer does (also called *global namespace polluting*). Mozilla also does not support the Netscape 4 `document.layers` method and Internet Explorer's `document.all`. While `document.getElementById` lets you retrieve one element, you can also use `document.layers` and `document.all` to obtain a list of all document elements with a certain tag name, such as all `<div>` elements.

The W3C DOM Level 1 method gets references to all elements with the same tag name through `getElementsByTagName()`. The method returns an array in JavaScript, and can be called on the `document` element or other nodes to search only their subtree. To get an array of all elements in the DOM tree, you can use `getElementsByTagName("*")`.

The DOM Level 1 methods, as shown in Table 1, are commonly used to move an element to a certain position and toggle its visibility (menus, animations). Netscape 4 used the `<layer>` tag, which Mozilla does not support, as an HTML element that can be positioned anywhere. In Mozilla, you can position any element using the `<div>` tag, which Internet Explorer uses as well and which you'll find in the HTML specification.

Table 1. Methods used to access elements

Method	Description
document.getElementById(aId)	Returns a reference to the element with the specified ID.
document.getElementsByTagName(aTagName)	Returns an array of elements with the specified name in the document.

Traverse the DOM [edit]

Mozilla supports the W3C DOM APIs for traversing the DOM tree through JavaScript (see Table 2). The APIs exist for each node in the document and allow walking the tree in any direction. Internet Explorer supports these APIs as well, but it also supports its legacy APIs for walking a DOM tree, such as the `children` property.

Table 2. Methods used to traverse the DOM

Property / Method	Description
childNodes	Returns an array of all child nodes of the element.
firstChild	Returns the first child node of the element.
getAttribute(aAttributeName)	Returns the value for the specified attribute.
hasAttribute(aAttributeName)	Returns a boolean stating if the current node has an attribute defined with the specified name.
hasChildNodes()	Returns a boolean stating whether the current node has any child nodes.
lastChild	Returns the last child node of the element.
nextSibling	Returns the node immediately following the current one.
nodeName	Returns the name of the current node as a string.
nodeType	Returns the type of the current node. **Value** / **Description** 1 Element Node 2 Attribute Node 3 Text Node 4 CDATA Section Node 5 Entity Reference Node 6 Entity Node 7 Processing Instruction Node 8 Comment Node 9 Document Node 10 Document Type Node 11 Document Fragment Node 12 Notation Node
nodeValue	Returns the value of the current node. For nodes that contain text, such as text and comment nodes, it will return their string value. For attribute nodes, the attribute value is returned. For all other nodes, `null` is returned.
ownerDocument	Returns the `document` object containing the current node.
parentNode	Returns the parent node of the current node.
previousSibling	Returns the node immediately preceding the current one.
removeAttribute(aName)	Removes the specified attribute from the current node.

setAttribute(aName, aValue)	Sets the value of the specified attribute with the specified value.

Internet Explorer has a nonstandard quirk, where many of these APIs will skip white space text nodes that are generated, for example, by new line characters. Mozilla will not skip these, so sometimes you need to distinguish these nodes. Every node has a `nodeType` property specifying the node type. For example, an element node has type 1, while a text node has type 3 and a comment node is type 8. The best way to only process element nodes is to iterate over all child nodes and only process those with a nodeType of 1:

```
HTML:
  <div id="foo">
    <span>Test</span>
  </div>

JavaScript:
  var myDiv = document.getElementById("foo");
  var myChildren = myXMLDoc.childNodes;
  for (var i = 0; i < myChildren.length; i++) {
    if (myChildren[i].nodeType == 1){
      // element node
    };
  };
```

Generate and manipulate content

[edit]

Mozilla supports the legacy methods for adding content into the DOM dynamically, such as `document.write`, `document.open` and `document.close`. Mozilla also supports Internet Explorer's `innerHTML` method, which it can call on almost any node. It does not, however, support `outerHTML` (which adds markup around an element, and has no standard equivalent) and `innerText` (which sets the text value of the node, and which you can achieve in Mozilla by using `textContent`).

Internet Explorer has several content manipulation methods that are nonstandard and unsupported in Mozilla, including retrieving the value, inserting text and inserting elements adjacent to a node, such as `getAdjacentElement` and `insertAdjacentHTML`. Table 3 shows how the W3C standard and Mozilla manipulate content, all of which are methods of any DOM node.

Table 3. Methods Mozilla uses to manipulate content

Method	Description
appendChild(aNode)	Creates a new child node. Returns a reference to the new child node.
cloneNode(aDeep)	Makes a copy of the node it is called on and returns the copy. If aDeep is true, it copies over the node's entire subtree.
createElement(aTagName)	Creates and returns a new and parentless DOM node of the type specified by aTagName.
createTextNode(aTextValue)	Creates and returns a new and parentless DOM textnode with the data value specified by aTextValue.

insertBefore(aNewNode, aChildNode)	Inserts aNewNode before aChildNode, which must be a child of the current node.
removeChild(aChildNode)	Removes aChildNode and returns a reference to it.
replaceChild(aNewNode, aChildNode)	Replaces aChildNode with aNewNode and returns a reference to the removed node.

[edit]

Document fragments

For performance reasons, you can create documents in memory, rather than working on the existing document's DOM. DOM Level 1 Core introduced document fragments, which are lightweight documents that contain a subset of a normal document's interfaces. For example, `getElementById` does not exist, but `appendChild` does. You can also easily add document fragments to existing documents.

Mozilla creates document fragments through `document.createDocumentFragment()`, which returns an empty document fragment.

Internet Explorer's implementation of document fragments, however, does not comply with the W3C web standards and simply returns a regular document.

JavaScript differences

[edit]

Most differences between Mozilla and Internet Explorer are usually blamed on JavaScript. However, the issues usually lie in the APIs that a browser exposes to JavaScript, such as the DOM hooks. The two browsers possess few core JavaScript differences; issues encountered are often timing related.

JavaScript date differences

[edit]

The only `Date` difference is the `getYear` method. As per the ECMAScript specification (which is the specification JavaScript follows), the method is not Y2k-compliant, and running `new Date().getYear()` in 2004 will return "104". Per the ECMAScript specification, `getYear` returns the year minus 1900, originally meant to return "98" for 1998. `getYear` was deprecated in ECMAScript Version 3 and replaced with `getFullYear()`. Internet Explorer changed `getYear()` to work like `getFullYear()` and make it Y2k-compliant, while Mozilla kept the standard behavior.

JavaScript execution differences

[edit]

Different browsers execute JavaScript differently. For example, the following code assumes that the `div` node already exists in the DOM by the time the `script` block executes:

```
...
<div id="foo">Loading...</div>

<script>
```

```
    document.getElementById("foo").innerHTML = "Done.";
  </script>
```

However, this is not guaranteed. To be sure that all elements exist, you should use the `onload` event handler on the `<body>` tag:

```
  <body onload="doFinish();">

  <div id="foo">Loading...</div>

  <script>
    function doFinish() {
      var element = document.getElementById("foo");
      element.innerHTML = "Done.";
    }
  </script>
  ...
```

Such timing-related issues are also hardware-related -- slower systems can reveal bugs that faster systems hide. One concrete example is `window.open`, which opens a new window:

```
  <script>
    function doOpenWindow(){
      var myWindow = window.open("about:blank");
      myWindow.location.href = "http://www.ibm.com";
    }
  </script>
```

The problem with the code is that `window.open` is asynchronous -- it does not block the JavaScript execution until the window has finished loading. Therefore, you may execute the line after the `window.open` line before the new window has finished. You can deal with this by having an `onload` handler in the new window and then call back into the opener window (using `window.opener`).

Differences in JavaScript-generating HTML

[edit]

JavaScript can, through `document.write`, generate HTML on the fly from a string. The main issue here is when JavaScript, embedded inside an HTML document (thus, inside an `<script>` tag), generates HTML that contains a `<script>` tag. If the document is in strict rendering mode, it will parse the `</script>` inside the string as the closing tag for the enclosing `<script>`. The following code illustrates this best:

```
  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
  ...
  <script>
    document.write("<script type='text\/javascript'>alert('Hello');<\/
  script>")
  </script>
```

Since the page is in strict mode, Mozilla's parser will see the first `<script>` and parse until it finds a closing tag for it, which would be the first `</script>`. This is because the parser

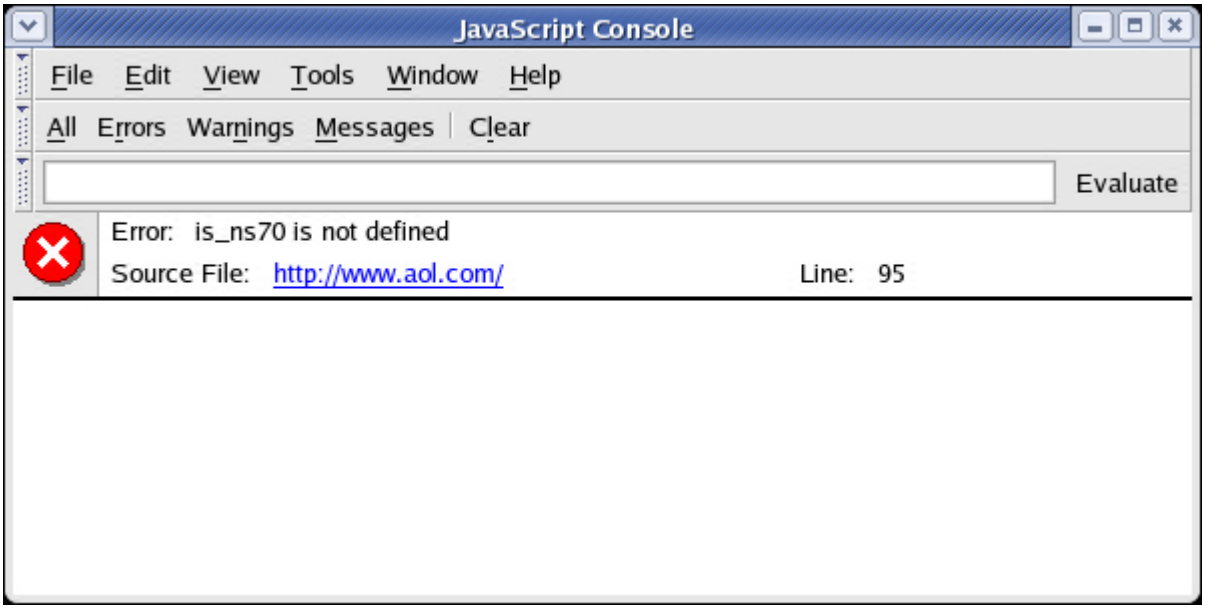
has no knowledge about JavaScript (or any other language) when in strict mode. In quirks mode, the parser is aware of JavaScript when parsing (which slows it down). Internet Explorer is always in quirks mode, as it does not support true XHTML. To make this work in strict mode in Mozilla, separate the string into two parts:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
...
<script>
  document.write("<script type='text\/javascript'>alert('Hello');</"
 + "script>")
</script>
```

Debug JavaScript [edit]

Mozilla provides several ways to debug JavaScript-related issues found in applications created for Internet Explorer. The first tool is the built-in JavaScript console, shown in Figure 1, where errors and warnings are logged. You can access it in Mozilla by going to **Tools -> Web Development -> JavaScript Console** or in Firefox (the standalone browser product from Mozilla) at **Tools -> JavaScript Console**.

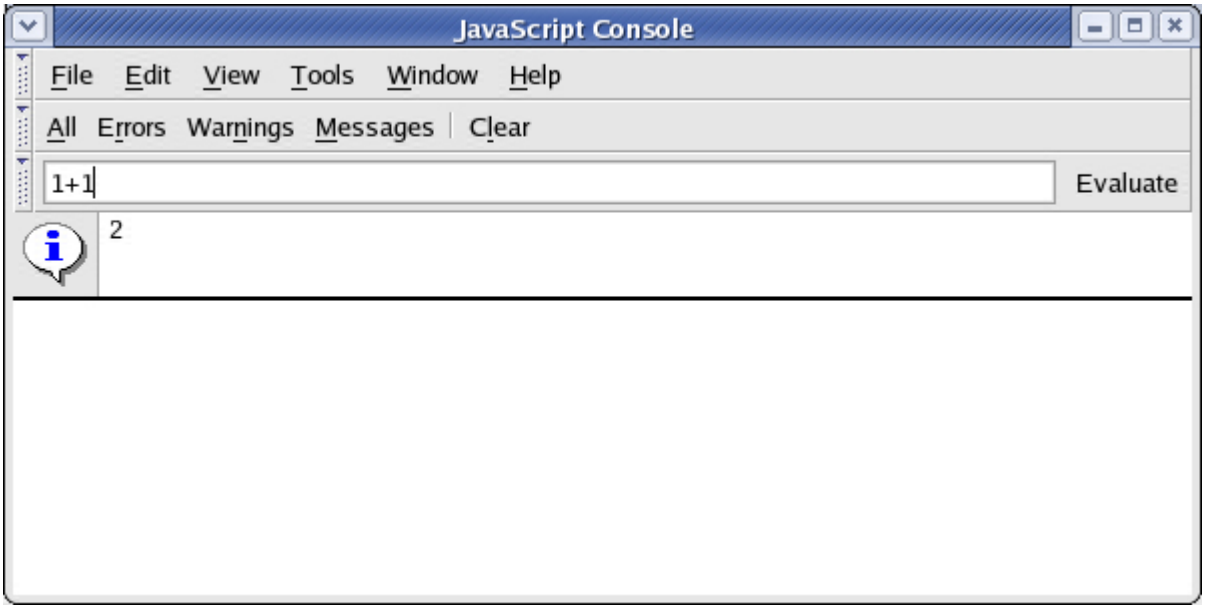
Figure 1. JavaScript console



The JavaScript console can show the full log list or just errors, warnings, and messages. The error message in Figure 1 says that at aol.com, line 95 tries to access an undefined variable called is_ns70. Clicking on the link will open Mozilla's internal view source window with the offending line highlighted.

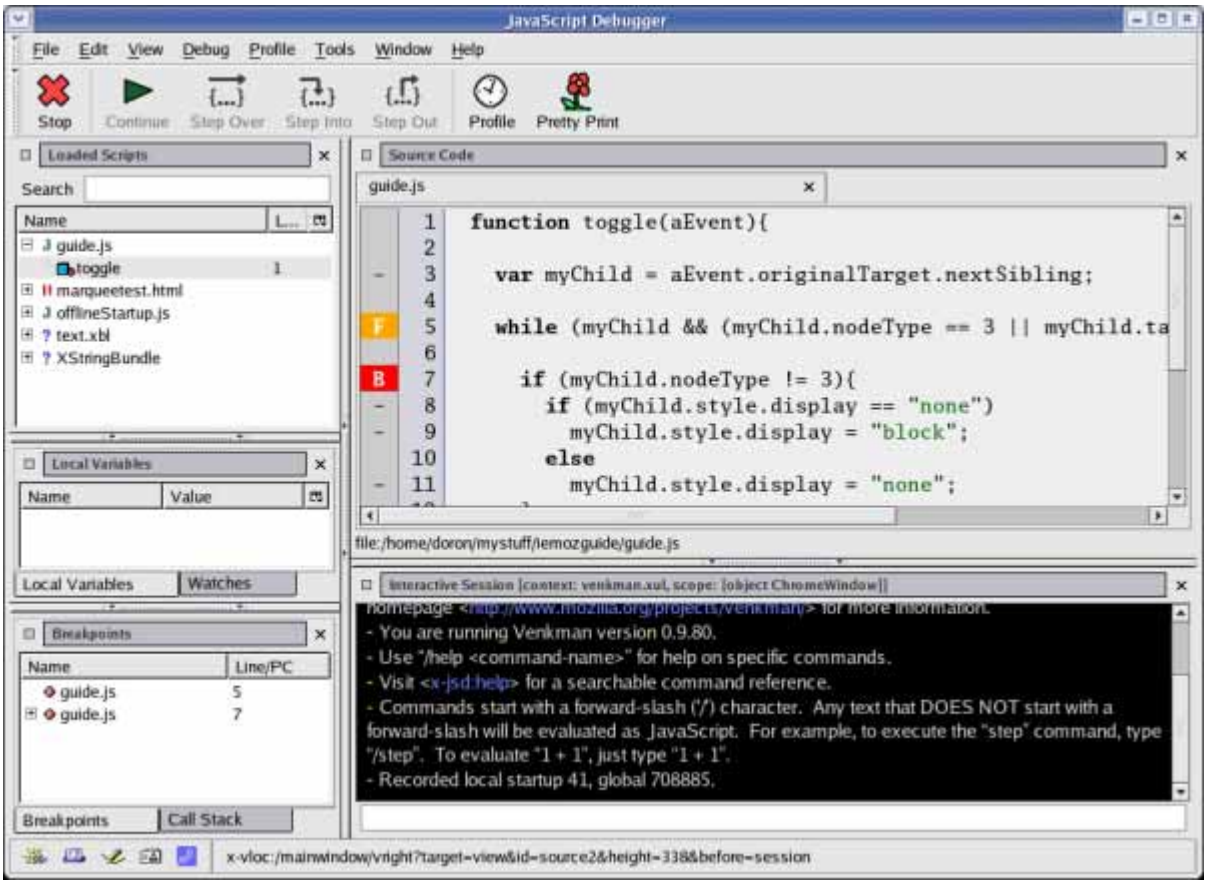
The console also allows you to evaluate JavaScript. To evaluate the entered JavaScript syntax, type in `1+1` into the input field and press **Evaluate**, as Figure 2 shows.

Figure 2. JavaScript console evaluating



Mozilla's JavaScript engine has built-in support for debugging, and thus can provide powerful tools for JavaScript developers. Venkman, shown in Figure 3, is a powerful, cross-platform JavaScript debugger that integrates with Mozilla. It is usually bundled with Mozilla releases; you can find it at **Tools -> Web Development -> JavaScript Debugger**. For Firefox, the debugger isn't bundled; instead, you can download and install it from the Venkman Project Page. You can also find tutorials at the development page, located at the Venkman Development Page.

Figure 3. Mozilla's JavaScript debugger



The JavaScript debugger can debug JavaScript running in the Mozilla browser window. It supports such standard debugging features as breakpoint management, call stack

inspection, and variable/object inspection. All features are accessible through the user interface or through the debugger's interactive console. With the console, you can execute arbitrary JavaScript in the same scope as the JavaScript currently being debugged.

CSS differences [edit]

Mozilla-based products have the strongest support for Cascading Style Sheets (CSS), including most of CSS1, CSS2.1 and parts of CSS3, compared to Internet Explorer as well as all other browsers.

For most issues mentioned below, Mozilla will add an error or warning entry into the JavaScript console. Check the JavaScript console if you encounter CSS-related issues.

Mimetypes (when CSS files are not applied) [edit]

The most common CSS-related issue is that CSS definitions inside referenced CSS files are not applied. This is usually due to the server sending the wrong mimetype for the CSS file. The CSS specification states that CSS files should be served with the `text/css` mimetype. Mozilla will respect this and only load CSS files with that mimetype if the Web page is in strict standards mode. Internet Explorer will always load the CSS file, no matter with which mimetype it is served. Web pages are considered in strict standards mode when they start with a strict doctype. To solve this problem, you can make the server send the right mimetype or remove the doctype. I'll discuss more about doctypes in the next section.

CSS and units [edit]

Many Web applications do not use units with their CSS, especially when you use JavaScript to set the CSS. Mozilla tolerates this, as long as the page is not rendered in strict mode. Since Internet Explorer does not support true XHTML, it does not care if no units are specified. If the page is in strict standards mode, and no units are used, then Mozilla ignores the style:

```
<DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
  "http://www.w3.org/TR/html4/strict.dtd">
<html>
  <head>
   <meta http-equiv="Content-Type" content="text/html; charset=iso-
8859-1">
   <title>CSS and units example</title>
  </head>
  <body>
    // works in strict mode
    <div style="width: 40px; border: 1px solid black;">
      Text
    </div>

    // will fail in strict mode
    <div style="width: 40; border: 1px solid black;">
      Text
    </div>
  </body>
</html>
```

Since the above example has a strict doctype, the page is rendered in strict standards mode. The first div will have a width of 40px, since it uses units, but the second div won't get a width, and thus will default to 100% width. The same would apply if the width were set through JavaScript.

JavaScript and CSS [edit]

Since Mozilla supports the CSS standards, it also supports the CSS DOM standard for setting CSS through JavaScript. You can access, remove, and change an element's CSS rules through the element's `style` member:

```
<div id="myDiv" style="border: 1px solid black;">
  Text
</div>

<script>
  var myElm = document.getElementById("myDiv");
  myElm.style.width = "40px";
</script>
```

You can reach every CSS attribute that way. Again, if the Web page is in strict mode, you must set a unit or else Mozilla will ignore the command. When you query a value, say through `.style.width`, in Mozilla and Internet Explorer, the returned value will contain the unit, meaning a string is returned. You can convert the string into a number through `parseFloat("40px")`.

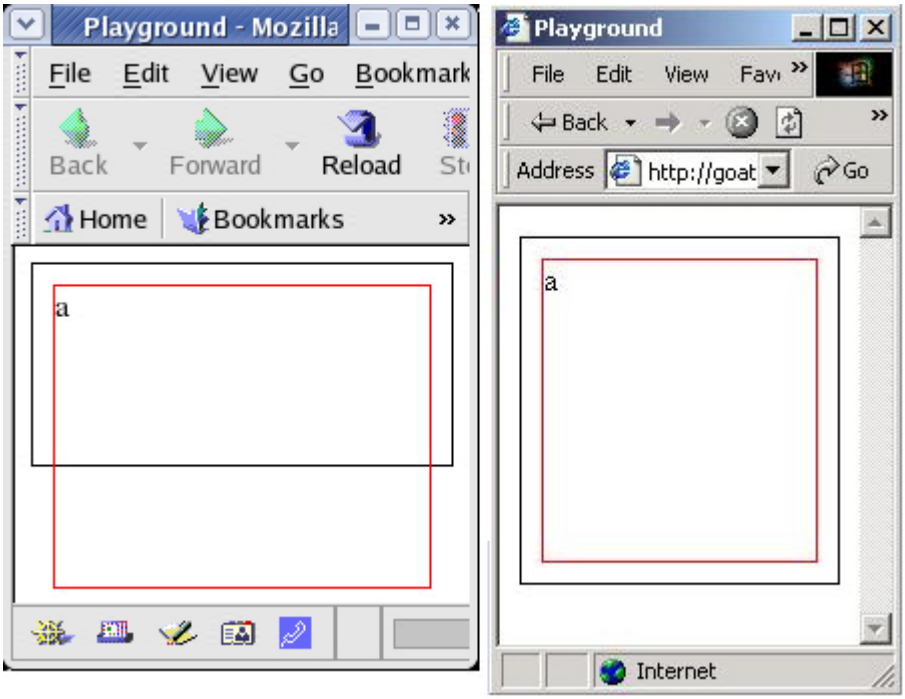
CSS overflow differences [edit]

CSS added the notion of overflow, which allows you to define how to handle overflow; for example, when the contents of a `div` with a specified height are taller than that height. The CSS standard defines that if no overflow behavior is set in this case, the `div` contents will overflow. However, Internet Explorer does not comply with this and will expand the `div` beyond its set height in order to hold the contents. Below is an example that shows this difference:

```
<div style="height: 100px; border: 1px solid black;">
  <div style="height: 150px; border: 1px solid red; margin: 10px;">
    a
  </div>
</div>
```

As you can see in Figure 4, Mozilla acts like the W3C standard specifies. The W3C standard says that, in this case, the inner `div` overflows to the bottom since the inner content is taller than its parent. If you prefer the Internet Explorer behavior, simply do not specify a height on the outer element.

Figure 4. DIV overflow



hover differences

[edit]

The nonstandard CSS hover behavior in Internet Explorer occurs on quite a few web sites. It usually manifests itself by changing text style when hovered over in Mozilla, but not in Internet Explorer. This is because the `a:hover` CSS selector in Internet Explorer matches `<a href="">...</a>` but not `<a name="">...</a>`, which sets anchors in HTML. The text changes occur because authors encapsulate the areas with the anchor-setting markup:

```
CSS:
  a:hover {color: green;}

HTML:
  <a href="foo.com">This text should turn green when you hover over
it.</a>

  <a name="anchor-name">
    This text should change color when hovered over, but doesn't
    in Internet Explorer.
  </a>
```

Mozilla follows the CSS specification correctly and will change the color to green in this example. You can use two ways to make Mozilla behave like Internet Explorer and not change the color of the text when hovered over:

- First, you can change the CSS rule to be `a:link:hover {color: green;}`, which will only change the color if the element is a link (has an `href` attribute).
- Alternatively, you can change the markup and close the opened `<a />` before the start of the text -- the anchor will continue to work this way.

[edit]

Quirks versus standards mode

Older legacy browsers, such as Internet Explorer 4, rendered with so-called quirks under certain conditions. While Mozilla aims to be a standards-compliant browser, it has three modes that support older Web pages created with these quirky behaviors. The page's content and delivery determine which mode Mozilla will use. Mozilla will indicate the rendering mode in **View -> Page Info** (or `Ctrl+I`) ; Firefox will list the rendering mode in **Tools -> Page Info**. The mode in which a page is located depends on its doctype.

Doctypes (short for document type declarations) look like this:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
```

The section in blue is called the public identifier, the green part is the system identifier, which is a URI.

Standards mode [edit]

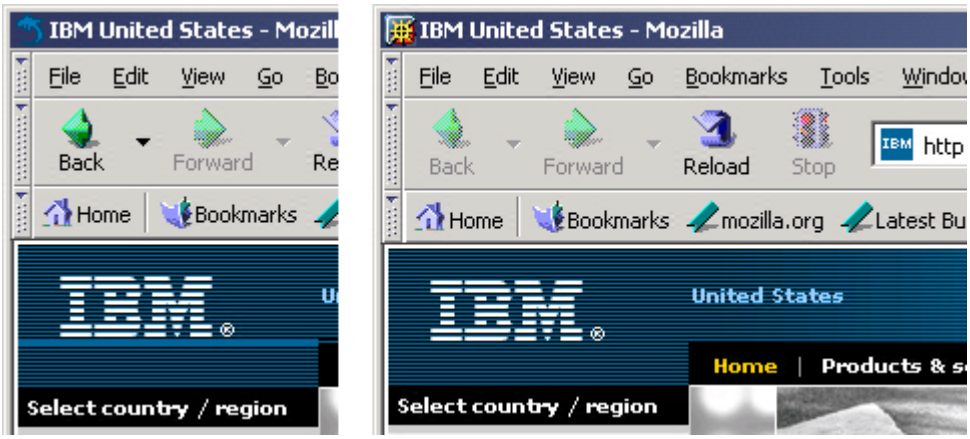
Standards mode is the strictest rendering mode -- it will render pages per the W3C HTML and CSS specifications and will not support any quirks. Mozilla uses it for the following conditions:

- If a page is sent with a `text/xml` mimetype or any other XML or XHTML mimetype
- For any "DOCTYPE HTML SYSTEM" doctype (for example, `<!DOCTYPE HTML SYSTEM "http://www.w3.org/TR/REC-html40/strict.dtd">`), except for the IBM doctype
- For unknown doctypes or doctypes without DTDs

Almost standards mode [edit]

Mozilla introduced almost standards mode for one reason: a section in the CSS 2 specification breaks designs based on a precise layout of small images in table cells. Instead of forming one image to the user, each small image ends up with a gap next to it. The old IBM homepage shown in Figure 5 offers an example.

Figure 5. Image gap



Almost standards mode behaves almost exactly as standards mode, except when it comes to an image gap issue. The issue occurs often on standards-compliant pages and causes them to display incorrectly.

Mozilla uses almost standards mode for the following conditions:

- For any "loose" doctype (for example, `<!DOCTYPE HTML PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN">`, `<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">`)
- For the IBM doctype (`<!DOCTYPE html SYSTEM "http://www.ibm.com/data/dtd/v11/ibmxhtml1-transitional.dtd">`)

You can read more about the image gap issue.

Quirks mode [edit]

Currently, the Web is full of invalid HTML markup, as well as markup that only functions due to bugs in browsers. The old Netscape browsers, when they were the market leaders, had bugs. When Internet Explorer arrived, it mimicked those bugs in order to work with the content at that time. As newer browsers came to market, most of these original bugs, usually called **quirks**, were kept for backwards compatibility. Mozilla supports many of these in its quirks rendering mode. Note that due to these quirks, pages will render slower than if they were fully standards-compliant. Most Web pages are rendered under this mode.

Mozilla uses quirks mode for the following conditions:

- When no doctype is specified
- For doctypes without a system identifier (for example, `<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">`)

For further reading, check out: Mozilla Quirks Mode Behavior and Mozilla's DOCTYPE sniffing.

Event differences [edit]

Mozilla and Internet Explorer are almost completely different in the area of events. The Mozilla event model follows the W3C and Netscape model. In Internet Explorer, if a function is called from an event, it can access the `event` object through `window.event`. Mozilla passes an `event` object to event handlers. They must specifically pass the object on to the function called through an argument.

A cross-browser event handling example follows (note that it means you can't define a global variable named `event` in your code):

```
<div onclick="handleEvent(event);">Click me!</div>

<script>
  function handleEvent(aEvent) {
    var myEvent = window.event ? window.event : aEvent;
  }
</script>
```

The properties and functions that the event object exposes are also often named differently in Mozilla and Internet Explorer, as Table 4 shows.

Table 4. Event properties differences between Mozilla and Internet Explorer

Internet Explorer Name	Mozilla Name	Description
altKey	altKey	Boolean property that returns whether the alt key was pressed during the event.
cancelBubble	stopPropagation ()	Used to stop the event from bubbling farther up the tree.
clientX	clientX	The X coordinate of the event, in relation to the element viewport.
clientY	clientY	The Y coordinate of the event, in relation to the element viewport.
ctrlKey	ctrlKey	Boolean property that returns whether the Ctrl key was pressed during the event.
fromElement	relatedTarget	For mouse events, this is the element from which the mouse moved away.
keyCode	keyCode	For keyboard events, this is a number representing the key that was pressed. It is 0 for mouse events. For keypress events (not keydown/keyup) of keys that produce output, the Mozilla equivalent is charCode, not keyCode.
returnValue	preventDefault ()	Used to prevent the event's default action from occurring.
screenX	screenX	The X coordinate of the event, in relation to the screen.
screenY	screenY	The Y coordinate of the event, in relation to the screen.
shiftKey	shiftKey	Boolean property that returns whether the Shift key was pressed during the event.
srcElement	target	The element to which the event was originally dispatched.
toElement	currentTarget	For mouse events, this is the element to which the mouse moved.
type	type	Returns the name of the event.

[edit]

Attach event handlers

Mozilla supports two ways to attach events through JavaScript. The first, supported by all browsers, sets event properties directly on objects. To set a `click` event handler, a function reference is passed to the object's `onclick` property:

```
<div id="myDiv">Click me!</div>

<script>
  function handleEvent(aEvent) {
    // if aEvent is null, means the Internet Explorer event model,
    // so get window.event.
    var myEvent = aEvent ? aEvent : window.event;
  }

  function onPageLoad(){
    document.getElementById("myDiv").onclick = handleEvent;
  }
</script>
```

Mozilla fully supports the W3C standard way of attaching listeners to DOM nodes. You use the `addEventListener()` and `removeEventListener()` methods, and have the benefit of being able to set multiple listeners for the same event type. Both methods require three parameters: the event type, a function reference, and a boolean denoting whether the listener should catch events in their capture phase. If the boolean is set to false, it will only catch bubbling events. W3C events have three phases: capturing, at target, and bubbling. Every event object has an `eventPhase` attribute indicating the phase numerically (0 indexed). Every time you trigger an event, the event starts at the DOM's outermost element, the element at the top of the DOM tree. It then walks the DOM using the most direct route toward the target, which is the capturing phase. When the event reaches the target, the event is in the target phase. After arriving at the target, it walks up the DOM tree back to the outermost node; this is **bubbling**. Internet Explorer's event model only has the bubbling phase; therefore, setting the third parameter to false results in Internet Explorer-like behavior:

```
<div id="myDiv">Click me!</div>

<script>

  function handleEvent(aEvent) {
    // if aEvent is null, it is the Internet Explorer event model,
    // so get window.event.
    var myEvent = aEvent ? aEvent : window.event;
  }

  function onPageLoad() {
    var element = document.getElementById("myDiv");
    element.addEventListener("click", handleEvent, false);
  }
</script>
```

One advantage of `addEventListener()` and `removeEventListener()` over setting properties is that you can have multiple event listeners for the same event, each calling another function. Thus, to remove an event listener requires all three parameters be the same as the ones you use when adding the listener.

Mozilla does not support Internet Explorer's method of converting <script> tags into event handlers, which extends <script> with `for` and `event` attributes (see Table 5). It also does not support the `attachEvent` and `detachEvent` methods. Instead, you should

use the `addEventListener` and `removeEventListener` methods. Internet Explorer does not support the W3C events specification.

Table 5. Event method differences between Mozilla and Internet Explorer

Internet Explorer Method	Mozilla Method	Description
attachEvent(aEventType, aFunctionReference)	addEventListener(aEventType, aFunctionReference, aUseCapture)	Adds an event listener to a DOM element.
detachEvent(aEventType, aFunctionReference)	removeEventListener(aEventType, aFunctionReference, aUseCapture)	Removes an event listener to a DOM element.

[edit]

Rich text editing

While Mozilla prides itself with being the most W3C web standards compliant browser, it does support nonstandard functionality, such as `innerHTML` and rich text editing, if no W3C equivalent exists.

Mozilla 1.3 introduced an implementation of Internet Explorer's designMode feature, which turns an HTML document into a rich text editor field. Once turned into the editor, commands can run on the document through the `execCommand` command. Mozilla does not support Internet Explorer's `contentEditable` attribute for making any widget editable. You can use an iframe to add a rich text editor.

[edit]

Rich text differences

Mozilla supports the W3C standard of accessing iframe's document object through `IFrameElmRef.contentDocument`, while Internet Explorer requires you to access it through `document.frames["IframeName"]` and then access the resulting `document`:

```
<script>
function getIFrameDocument(aID) {
  var rv = null;

  // if contentDocument exists, W3C compliant (Mozilla)
  if (document.getElementById(aID).contentDocument){
    rv = document.getElementById(aID).contentDocument;
  } else {
    // IE
    rv = document.frames[aID].document;
  }
  return rv;
}
</script>
```

Another difference between Mozilla and Internet Explorer is the HTML that the rich text editor creates. Mozilla defaults to using CSS for the generated markup. However, Mozilla allows you to toggle between HTML and CSS mode using the `useCSS` execCommand and toggling it between true and false. Internet Explorer always uses HTML markup.

```
  Mozilla (CSS):
    <span style="color: blue;">Big Blue</span>

  Mozilla (HTML):
    <font color="blue">Big Blue</font>

  Internet Explorer:
    <FONT color="blue">Big Blue</FONT>
```

Below is a list of commands that execCommand in Mozilla supports:

Table 6. Rich text editing commands

Command Name	Description	Argument
bold	Toggles the selection's bold attribute.	---
createlink	Generates an HTML link from the selected text.	The URL to use for the link
delete	Deletes the selection.	---
fontname	Changes the font used in the selected text.	The font name to use (Arial, for example)
fontsize	Changes the font size used in the selected text.	The font size to use
fontcolor	Changes the font color used in the selected text.	The color to use
indent	Indents the block where the caret is.	---
inserthorizontalrule	Inserts an <hr> element at the cursor's position.	---
insertimage	Inserts an image at the cursor's position.	URL of the image to use
insertorderedlist	Inserts an ordered list ( ) element at the cursor's position.	---
insertunorderedlist	Inserts an unordered list ( ) element at the cursor's position.	---
italic	Toggles the selection's italicize attribute.	---
justifycenter	Centers the content at the current line.	---
justifyleft	Justifies the content at the current line to the left.	---
justifyright	Justifies the content at the current line to the right.	---
outdent	Outdents the block where the caret is.	---
redo	Redoes the previous undo command.	---
removeformat	Removes all formatting from the selection.	---

selectall	Selects everything in the rich text editor.	---
strikethrough	Toggles the strikethrough of the selected text.	---
subscript	Converts the current selection into subscript.	---
superscript	Converts the current selection into superscript.	---
underline	Toggles the underline of the selected text.	---
undo	Undoes the last executed command.	---
unlink	Removes all link information from the selection.	---
useCSS	Toggles the usage of CSS in the generated markup.	Boolean value

For more information, visit Rich-Text Editing in Mozilla.

XML differences [edit]

Mozilla has strong support for XML and XML-related technologies, such as XSLT and Web services. It also supports some non-standard Internet Explorer extensions, such as `XMLHttpRequest`.

How to handle XML [edit]

As with standard HTML, Mozilla supports the W3C XML DOM specification, which allows you to manipulate almost any aspect of an XML document. Differences between Internet Explorer's XML DOM and Mozilla are usually caused by Internet Explorer's nonstandard behaviors. Probably the most common difference is how they handle white space text nodes. Often when XML generates, it contains white spaces between XML nodes. Internet Explorer, when using `Node.childNodes`, will not contain these white space nodes. In Mozilla, those nodes will be in the array.

```
XML:
  <?xml version="1.0"?>
  <myXMLdoc xmlns:myns="http://myfoo.com">
    <myns:foo>bar</myns:foo>
  </myXMLdoc>

JavaScript:
  var myXMLDoc = getXMLDocument().documentElement;
  alert(myXMLDoc.childNodes.length);
```

The first line of JavaScript loads the XML document and accesses the root element (`myXMLDoc`) by retrieving the `documentElement`. The second line simply alerts the

number of child nodes. Per the W3C specification, the white spaces and new lines merge into one text node if they follow each other. For Mozilla, the `myXMLdoc` node has three children: a text node containing a new line and two spaces; the `myns:foo` node; and another text node with a new line. Internet Explorer, however, does not abide by this and will return "1" for the above code, namely only the `myns:foo` node. Therefore, to walk the child nodes and disregard text nodes, you must distinguish such nodes.

As mentioned earlier, every node has a `nodeType` property representing the node type. For example, an element node has type 1, while a document node has type 9. To disregard text nodes, you must check for types 3 (text node) and 8 (comment node).

```
XML:
  <?xml version="1.0"?>
  <myXMLdoc xmlns:myns="http://myfoo.com">
    <myns:foo>bar</myns:foo>
  </myXMLdoc>

JavaScript:
  var myXMLDoc = getXMLDocument().documentElement;
  var myChildren = myXMLDoc.childNodes;

  for (var run = 0; run < myChildren.length; run++){
    if ( (myChildren[run].nodeType != 3) &&
         myChildren[run].nodeType != 8) ){
      // not a text or comment node
    };
  };
```

See Whitespace in the DOM for more detailed discussion and a possible solution.

XML data islands [edit]

Internet Explorer has a nonstandard feature called XML data islands, which allow you to embed XML inside an HTML document using the nonstandard HTML tag `<xml>`. Mozilla does not support XML data islands and handles them as unknown HTML tags. You can achieve the same functionality using XHTML; however, because Internet Explorer's support for XHTML is weak, this is usually not an option.

IE XML data island:

```
<xml id="xmldataisland">
  <foo>bar</foo>
</xml>
```

One cross-browser solution is to use DOM parsers, which parse a string that contains a serialized XML document and generates the document for the parsed XML. Mozilla uses the `DOMParser` object, which takes the serialized string and creates an XML document out of it. In Internet Explorer, you can achieve the same functionality using ActiveX. The object created using `new ActiveXObject("Microsoft.XMLDOM")` has a `loadXML` method that can take in a string and generate a document from it. The following code shows you how:

```
var xmlString = "<xml id=\"xmldataisland\"><foo>bar</foo></xml>";
var myDocument;

if (window.DOMParser) {
  // This browser appears to support DOMParser
  var parser = new DOMParser();
  myDocument = parser.parseFromString(xmlString, "text/xml");
} else if (window.ActiveXObject){
  // Internet Explorer, create a new XML document using ActiveX
  // and use loadXML as a DOM parser.
  myDocument = new ActiveXObject("Microsoft.XMLDOM");
  myDocument.async = false;

  myDocument.loadXML(xmlString);
} else {
  // Not supported.
}
```

See Using XML Data Islands in Mozilla for an alternative approach.

XMLHttpRequest [edit]

Internet Explorer allows you to send and retrieve XML files using MSXML's `XMLHTTP` class, which is instantiated through ActiveX using `new ActiveXObject("Msxml2.XMLHTTP")` or `new ActiveXObject("Microsoft.XMLHTTP")`. Since there is no standard method of doing this, Mozilla provides the same functionality in the global JavaScript `XMLHttpRequest` object. Since version 7 IE also supports the "native" `XMLHttpRequest` object.

After instantiating the object using `new XMLHttpRequest()`, you can use the `open` method to specify what type of request (GET or POST) you use, which file you load, and if it is asynchronous or not. If the call is asynchronous, then give the `onload` member a function reference, which is called once the request has completed.

Synchronous request:

```
var myXMLHTTPRequest = new XMLHttpRequest();
myXMLHTTPRequest.open("GET", "data.xml", false);

myXMLHTTPRequest.send(null);

var myXMLDocument = myXMLHTTPRequest.responseXML;
```

Asynchronous request:

```
var myXMLHTTPRequest;

function xmlLoaded() {
  var myXMLDocument = myXMLHTTPRequest.responseXML;
}

function loadXML(){
  myXMLHTTPRequest = new XMLHttpRequest();
  myXMLHTTPRequest.open("GET", "data.xml", true);
  myXMLHTTPRequest.onload = xmlLoaded;
```

```
    myXMLHTTPRequest.send(null);
  }
```

Table 7 features a list of available methods and properties for Mozilla's `XMLHttpRequest`.

Table 7. XMLHttpRequest methods and properties

Name	Description
void abort()	Stops the request if it is still running.
string getAllResponseHeaders ()	Returns all response headers as one string.
string getResponseHeader (string headerName)	Returns the value of the specified header.
functionRef onerror	If set, the references function will be called whenever an error occurs during the request.
functionRef onload	If set, the references function will be called when the request completes successfully and the response has been received. Use when an asynchronous request is used.
void open (string HTTP_ Method, string URL) void open (string HTTP_ Method, string URL, boolean async, string userName, string password)	Initializes the request for the specified URL, using either GET or POST as the HTTP method. To send the request, call the `send()` method after initialization. If `async` is false, the request is synchronous, else it defaults to asynchronous. Optionally, you can specify a username and password for the given URL needed.
int readyState	State of the request. Possible values: **Value** / **Description** 0 — UNINITIALIZED - open() has not been called yet. 1 — LOADING - send() has not been called yet. 2 — LOADED - send() has been called, headers and status are available. 3 — INTERACTIVE - Downloading, responseText holds the partial data. 4 — COMPLETED - Finished with all operations.
string responseText	String containing the response.
DOMDocument responseXML	DOM Document containing the response.
void send(variant body)	Initiates the request. If `body` is defined, it is sent as the body of the POST request. `body` can be an XML document or a string serialized XML document.
void setRequestHeader (string headerName, string headerValue)	Sets an HTTP request header for use in the HTTP request. Has to be called after `open()` is called.

string status	The status code of the HTTP response.

[edit]

XSLT differences

Mozilla supports XSL Transformations (XSLT) 1.0. It also allows JavaScript to perform XSLT transformations and allows running XPath on a document.

Mozilla requires that you send the XML and XSLT files with an XML mimetype (`text/xml` or `application/xml`). This is the most common reason why XSLT won't run in Mozilla but will in Internet Explorer. Mozilla is strict in that way.

Internet Explorer 5.0 and 5.5 supported XSLT's working draft, which is substantially different than the final 1.0 recommendation. The easiest way to distinguish what version an XSLT file was written against is to look at the namespace. The namespace for the 1.0 recommendation is `http://www.w3.org/1999/XSL/Transform`, while the working draft's namespace is `http://www.w3.org/TR/WD-xsl`. Internet Explorer 6 supports the working draft for backwards compatibility, but Mozilla does not support the working draft, only the final recommendation.

If XSLT requires you to distinguish the browser, you can query the "xsl:vendor" system property. Mozilla's XSLT engine will report itself as "Transformiix" and Internet Explorer will return "Microsoft".

```
<xsl:if test="system-property('xsl:vendor') = 'Transformiix'">
  <!-- Mozilla specific markup -->
</xsl:if>
<xsl:if test="system-property('xsl:vendor') = 'Microsoft'">
  <!-- Internet Explorer specific markup -->
</xsl:if>
```

Mozilla also provides JavaScript interfaces for XSLT, allowing a Web site to complete XSLT transformations in memory. You can do this using the global `XSLTProcessor` JavaScript object. `XSLTProcessor` requires you to load the XML and XSLT files, because it needs their DOM documents. The XSLT document, imported by the `XSLTProcessor`, allows you to manipulate XSLT parameters.

`XSLTProcessor` can generate a standalone document using `transformToDocument()`, or it can create a document fragment using `transformToFragment()`, which you can easily append into another DOM document. Below is an example:

```
var xslStylesheet;
var xsltProcessor = new XSLTProcessor();

// load the xslt file, example1.xsl
var myXMLHTTPRequest = new XMLHttpRequest();
myXMLHTTPRequest.open("GET", "example1.xsl", false);
myXMLHTTPRequest.send(null);

// get the XML document and import it
xslStylesheet = myXMLHTTPRequest.responseXML;

xsltProcessor.importStylesheet(xslStylesheet);
```

```
// load the xml file, example1.xml
myXMLHTTPRequest = new XMLHttpRequest();
myXMLHTTPRequest.open("GET", "example1.xml", false);
myXMLHTTPRequest.send(null);

var xmlSource = myXMLHTTPRequest.responseXML;

var resultDocument = xsltProcessor.transformToDocument(xmlSource);
```

After creating an `XSLTProcessor`, you load the XSLT file using `XMLHttpRequest`. The XMLHttpRequest's `responseXML` member contains the XML document of the XSLT file, which is passed to `importStylesheet`. You then use the `XMLHttpRequest` again to load the source XML document that must be transformed; that document is then passed to the `transformToDocument` method of `XSLTProcessor`. Table 8 features a list of `XSLTProcessor` methods.

Table 8. XSLTProcessor methods

Method	Description
void importStylesheet(Node styleSheet)	Imports the XSLT stylesheet. The `styleSheet` argument is the root node of an XSLT stylesheet's DOM document.
DocumentFragment transformToFragment(Node source, Document owner)	Transforms the Node `source` by applying the stylesheet imported using the `importStylesheet` method and generates a DocumentFragment. `owner` specifies what DOM document the DocumentFragment should belong to, making it appendable to that DOM document.
Document transformToDocument(Node source)	Transforms the Node `source` by applying the stylesheet imported using the `importStylesheet` method and returns a standalone DOM document.
void setParameter(String namespaceURI, String localName, Variant value)	Sets a parameter in the imported XSLT stylesheet.
Variant getParameter(String namespaceURI, String localName)	Gets the value of a parameter in the imported XSLT stylesheet.
void removeParameter(String namespaceURI, String localName)	Removes all set parameters from the imported XSLT stylesheet and makes them default to the XSLT-defined defaults.
void clearParameters()	Removes all set parameters and sets them to defaults specified in the XSLT stylesheet.
void reset()	Removes all parameters and stylesheets.

[edit]

Original Document Information

- Author(s): Doron Rosenberg, IBM Corporation
- Published: 26 Jul 2005

- Link: http://www-128.ibm.com/developerworks/web/library/wa-ie2mozgd/

Migration d'applications d'Internet Explorer vers Mozilla

Retrieved from "http://developer.mozilla.org/en/docs/Migrate_apps_from_Internet_Explorer_to_Mozilla"