Table of contents

Below you find the contents of the W3C DOM section of my site.

General

Compatibility tables

See also the key to my compatibility tables.

Module	Explorer 5 Windows	Explorer 6 Windows	Explorer 5.2 Mac	Mozilla 1.6	Safari 1.2	Opera 7.50
Core Node manipulation	Yes	Yes	Yes	Yes	Yes	Yes

The W3C DOM Core module defines how to access, read and manipulate an XML document. Well-formed HTML documents are XML documents, so these methods and properties can be used to completely rewrite any HTML page, if you so wish.

Here you find details on how to find elements, how to create new ones, how to read out node information and how to change the structure of the document.

HTML HTML tag manipulation	Yes	Yes	Yes	Yes	Yes	Yes

Tables

Though HTML documents are XML documents, they have a number of special features that the average XML document doesn't have. The W3C DOM HTML module defines these special cases and how to deal with them.

Here you find details on getting and setting properties of HTML elements, such as `className` or `id`. The `innerHTML` property is of prime importance to any DOM script.

CSS Stylesheet manipulation	Yes	Yes	Read only	Yes	Read only	No

Tables

Style sheets are part of the document, too (sort of). The W3C DOM CSS module gives access to style sheets and some browsers even allow you to change a style sheet. Where traditional DHTML changes only the styles of specially selected elements, these new methods allow you to change the styles of an entire page. Two lines of code suffice to make all paragraphs on your page red instead of black, for instance.

This module contains some browser incompatibilities, but they are of the cute kind. W3C and Microsoft define some different methods and arrays, but some simple object detection allows you to evade these problems.

Events Event manipulation	Microsoft	Microsoft	Mainly Microsoft	W3C	Mainly W3C; incomplete	Both

Tables

The W3C DOM Events module specifies how events are being handled. The MS DOM events module does the same, but in a different way. These tables detail both Event modules and also Netscape 4's old system.

See the Introduction to events and subsequent pages for more information.

Module	Explorer 5 Windows	Explorer 6 Windows	Explorer 5.2 Mac	Mozilla 1.6	Safari 1.2	Opera 7.50

Scripts

Script	Explorer 5 Windows	Explorer 6 Windows	Explorer 5.2 Mac	Mozilla 1.6	Safari 1.2	Opera 7.50
Import XML	Yes	Yes	No	Yes	No	No
	Example script how to load an XML document into your HTML page and create a table to display the data. In my opinion this will become important in the future, as soon as we can *save* XML documents back to the server.					
Edit text	Yes	Yes	Almost	Yes	Yes	Incomplete
	How to let the user click on a paragraph and let him edit its text in an input box. When the user is ready, the text becomes paragraph again. This is a very useful script for content management systems. The only flaw is that there is no simple way to send the revised texts to the server.					

Script	Explorer 5 Windows	Explorer 6 Windows	Explorer 5.2 Mac	Mozilla 1.6	Safari 1.2	Opera 7.50
Get styles	Yes	Yes	Yes	Yes	No	Yes
	How to get the default styles of HTML elements. For instance, a paragraph gets a percentual width from a general style sheet. How wide is the paragraph? Doesn't work perfectly yet, but you can read out some interesting information.					
Change style sheet	Yes	Yes	Yes	Yes	Yes	No
	This script changes an entire style sheet, so that you can, for instance, change the text colour of all your paragraphs with just a few lines of code. Unfortunately there are very grave browser incompatibilities that make this technique badly usable for the moment.					
Table of Contents	Yes	Yes	Crash	Yes	Yes	Yes

How to generate a Table of Contents based on the headers in the page. I use this script on every page of my site.

Image replacement	Yes	Yes	Yes	Yes	Yes	Yes

The Fahrner Image Replacement is very fashionable these days. It tries to replace a header text by an image by means of CSS. Unfortunately the countless variations on this theme make invalid assumptions about screen readers. Besides, I feel CSS is the wrong tool for image replacement. Instead we should use JavaScript. This script.

Three column stretching	5.5	Yes	Yes	Yes	Yes	Yes

This script gives an example of CSS enhancement through minimal JavaScript interference. It's only an example, it's not yet good enough to use in real pages.

Script	Explorer 5 Windows	Explorer 6 Windows	Explorer 5.2 Mac	Mozilla 1.6	Safari 1.2	Opera 7.50

Forms

Script	Explorer 5 Windows	Explorer 6 Windows	Explorer 5.2 Mac	Mozilla 1.6	Safari 1.2	Opera 7.50
Usable forms	Yes	Buggy	Crash	Yes	Yes	Incomplete

How to show and hide form fields based on user actions.
For once the explanation of the script is not on my site. I published the companion Forms, usability, and the W3C DOM article on Digital Web Magazine.

The script works in Explorer 6 Windows, but this browser may hide other page elements.

Extending forms	Incomplete	Incomplete	Buggy	Yes	Yes	Yes

Example script for the way the W3C DOM is going to change the interaction of web sites. In this example the user can choose how many form fields he'd like to see. We web developers don't have to decide on a maximum number any more, the user is completely free to do as he likes.

Form error	Yes	Yes	Danger	Yes	Yes	Yes

messages	How to write error messages next to the form field they apply to. This is clearly better than using alerts to show errors. The example script works in Explorer Mac, but this browser may fail spectacularly when you insert the script in a real web page.					
Styling an input type="file"	5.5	Yes	No	Yes	Yes	No
	Can almost be done in pure CSS, but I prefer JavaScript.					
Script	**Explorer 5 Windows**	**Explorer 6 Windows**	**Explorer 5.2 Mac**	**Mozilla 1.6**	**Safari 1.2**	**Opera 7.50**
Web Forms 2.0	Notes and comments on Ian Hickson's Web Forms 2.0 specification.					

Home Sitemap

W3C DOM Compatibility - Core

These compatibility tables detail support for the W3C DOM Core Level 1 and 2 modules in all modern browsers.

On this page I grouped the various W3C DOM methods and properties in eight tables. Basically you must know the first four tables by heart and you'll rarely need the second four tables.

1. The Four Methods with which the W3C DOM starts. Creating and finding HTML elements.
2. Node information. Once you found a node you need more information about it.
3. Walking the DOM tree. How to go from one node to another.
4. Node manipulation. How to move nodes through the document.

---

5. Microsoft extensions to Level 1 DOM Core. Generally not interesting.
6. Manipulating data. Data is always text, and there are some specialized methods for dealing with it.
7. Manipulating attributes. Terrible browser incompatibilities.
8. Miscellaneous methods and properties. You'll rarely need one of them.

See also the key to my compatibility tables.

All Safari 1.2 tests by Mark 'Tarquin' Wilton-Jones.

The Four Methods

First of all the Four Methods. The average W3C DOM script uses them all. The first two methods allow you to create element nodes and text nodes. Later you insert these newly created nodes into the document.

The second two methods are for finding elements in the page. You can either find a single one, identified by an `id`, or all tags of one type.

You must know these methods by heart.

Method or property	Explorer 5 Windows	Explorer 6 Windows	Explorer 5.2 Mac	Mozilla 1.6	Safari 1.2	Opera 7.50
createElement() Create a new element Test page	Yes	Yes	Yes	Yes	Yes	Yes
	x = document.createElement('P') Create a new HTML element node `<P>` and temporarily place it in `x`, which is later inserted into the document. • Explorer Windows also supports `('<P>')`.					
createTextNode() Create a new text node Test page	Yes	Yes	Yes	Yes	Yes	Yes
	x = document.createTextNode('text') Create a text node with content `text` and temporarily place it in `x`, which is later inserted into the document.					
getElementById() Get the element with this ID Test page **Lower case 'd'!!**	Almost	Almost	Yes	Yes	Yes	Yes

x = document.getElementById('test')
Take the element with `ID="test"` (wherever it is in the document) and put it in `x`. Now you can use all methods and properties on `x` and the element with `ID="test"` will change accordingly.
This is the equivalent of `document.layers` and `document.all` in the Version 4 browsers and it is necessary to make your DHTML work.

- Explorer Windows also returns the element with `name="test"`.

getElementsByTagName() Get all tags of this type Test page	Incomplete	Yes	Yes	Yes	Incomplete	Incomplete

x = document.getElementsByTagName('P')
Make `x` into an array of all P's in the document, so `x[1]` is the second P etc.
x = y.getElementsByTagName('P'): all P's that are descendants of node `y`.

- document.getElementsByTagName('*')
  Returns a list of all tags in a document. Doesn't work in Explorer 5 Windows.
- document.getElementsByTagName('PPK')
  Custom tags cannot be accessed in Safari and Opera.

Node information

These four properties give basic information about all nodes. What they return depends on the node type. They are read-only, except for `nodeValue`.

There are three basic node types: element nodes (HTML tags), attribute nodes and text nodes. I test these properties for all these three types and added a fourth node type: the document node (the root of all other nodes).

You must know these properties by heart.

Method or property	Explorer 5 Windows	Explorer 6 Windows	Explorer 5.2 Mac	Mozilla 1.6	Safari 1.2	Opera 7.50
nodeName The name of the node Test page	Yes	Yes	Yes	Yes	Yes	Yes

x.nodeName
The name of node `x`. Please note that `nodeName` returns an *UPPER case* tag name, so you must do

```
if (x.nodeName == 'SPAN')
```

Element nodes	Tag name
Attribute nodes	Attribute name
Text nodes	`#text`
Document node	`#document`

nodeType The type of the node Test page	Incomplete in 5.0 x.nodeType Element nodes 1 Attribute nodes 2 Text nodes 3 Document node 9 • Explorer 5.0 Windows assigns no nodeType to attributes. Solved in Explorer 5.5	Yes	Yes	Yes	Yes	Yes
nodeValue The value of the node, if any Test page	Yes x.nodeValue Element nodes Not available Attribute nodes Attribute value Text nodes Text Document node Not available You can also set the nodeValue: `x.nodeValue = 'new value'`	Yes	Yes	Yes	Yes	Yes
tagName The tag name of an element node Test page	Yes x.tagName Element nodes Tag name Attribute nodes Not available Text nodes Not available Document node Not available	Yes	Yes	Yes	Yes	Yes

The DOM tree

Five properties and two arrays for walking through the DOM tree. Using these properties, you can reach nodes that are close to the current node in the document structure.

In general you shouldn't use too many of these properties. As soon as you're doing something like

```
x.parentNode.firstChild.nextSibling.children[2]
```

your code is too complicated. Complex relationships between nodes can suddenly and unexpectedly change when you alter the document structure, and altering the document structure is the point of the W3C DOM. In general you should use only one or two of these properties per action.

You must know these properties by heart.

Method or property	Explorer 5 Windows	Explorer 6 Windows	Explorer 5.2 Mac	Mozilla 1.6	Safari 1.2	Opera 7.50
childNodes[] An array with all child nodes of the node Test page	Yes	Yes	Yes	Yes	Yes	Yes
children[] An array with all child element nodes of the node Test page	Yes	Yes	Yes	No	Yes	Yes
firstChild The first child node of the node Test page	Yes	Yes	Yes	Yes	Yes	Yes
lastChild The last child node of the node Test page	Yes	Yes	Yes	Yes	Yes	Yes
nextSibling The next sibling node of the node Test page	Yes	Yes	Yes	Yes	Yes	Yes
parentNode The parent node of the node Test page	Yes	Yes	Yes	Yes	Yes	Yes
previousSibling The previous sibling node of the node Test page	Yes	Yes	Yes	Yes	Yes	Yes
sourceIndex The index number of the node in the page source Test page	Yes	Yes	Yes	No	No	No

x.childNodes[1]
The second child node of node `x`.

x.children[1]
The second element child node of node `x`.
Where `childNodes` holds all child nodes, `children` only holds those that are element nodes (HTML tags).

x.firstChild
The first child node of node `x`.

x.lastChild
The last child of node `x`.

x.nextSibling
The next child of the parent of `x`.

x.parentNode
The parent node of `x`.

x.previousSibling
The previous child of the parent of `x`.

x.sourceIndex
The `sourceIndex` is also the index number for the element in the `document.getElementsByTagName('*')` array.

Node manipulation

These five methods allow you to restructure the document. The average DOM script uses at least two of these methods.

The changes in the document structure are applied immediately, the whole DOM tree is altered. The browser, too, will immediately show the changes.

You must know these methods by heart.

Method or property	Explorer 5 Windows	Explorer 6 Windows	Explorer 5.2 Mac	Mozilla 1.6	Safari 1.2	Opera 7.50
appendChild() Append a child node as the last node to an element Test page	Almost	Yes	Yes	Yes	Yes	Yes
	x.appendChild(y) Make node `y` the last child of node `x`. If you append a node that's somewhere else in the document, it moves to the new position. • I have heard, but not tested, that Explorer 5 Windows crashes when you append an `option` to a `select`.					
cloneNode() Clone a node Test page	Yes	Yes	Incomplete	Yes	Yes	Yes
	x = y.cloneNode(true | false) Make node `x` a copy of node `y`. If the argument is `true`, the entire tree below `y` is copied, if it's `false` only the root node `y` is copied. Later you insert the clone into the document. • Explorer 5 Mac doesn't clone the event handlers of the element.					
insertBefore() Insert a node into the child nodes of an element Test page	Yes	Yes	Yes	Yes	Yes	Yes
	x.insertBefore(y,z) Insert node `y` as a child of node `x` just before node `z`.					
removeChild() Remove a child node from an element Test page	Yes	Yes	Yes	Yes	Yes	Yes
	x.removeChild(y) Remove child `y` of node `x`.					
replaceChild() Replace a child node of an element by another child node Test page	Yes	Yes	Yes	Yes	Yes	Yes
	x.replaceChild(y,z) Replace node `z`, a child of node `x`, by node `y`.					

Microsoft extensions

As usual Microsoft has extended the standard somewhat. Though sometimes its extensions are brilliant

(`innerHTML` springs to mind), in the case of the DOM Core they aren't.

Note the difference between W3C and Microsoft methods. The W3C methods are owned by the parent element of the node you want to adjust, the Microsoft methods by the node itself.

Method or property	Explorer 5 Windows	Explorer 6 Windows	Explorer 5.2 Mac	Mozilla 1.6	Safari 1.2	Opera 7.50
applyElement() Give a node another parent node Test page	Yes	Yes	No	No	No	No
	x.applyElement(y) Make node `x` a child of node `y` This only applies the actual node, not its content (text, for instance). Not recommended.					
clearAttributes() Remove all attributes from a node Test page	Incomplete	Incomplete	No	No	No	No
	x.clearAttributes() Remove all attributes from node `x` • Explorer doesn't clear event handlers.					
mergeAttributes() Copy all attributes of one node to another node Test page	Yes	Yes	No	No	No	No
	x.mergeAttributes(y) Copy all of node `y`'s attributes to node `x`.					
removeNode() Remove a node Test page	Yes	Yes	No	No	No	Yes
	x.removeNode(true | false) Remove node `x` from the document. If you add `true` its children are also removed.					
replaceNode() Replace a node by another node Test page	Yes	Yes	No	No	No	No
	x.replaceNode(y) Replace node `x` by node `y`.					
swapNode() Swap two nodes Test page	Yes	Yes	No	No	No	No
	x.swapNode(y) Put node `x` in node `y`'s place and vice versa.					

Data

These methods are for manipulating text data. Note the difference between a text node and text data: the text node is a node and contains the data in its `nodeValue`. The methods below only work with this contained data.

Method or property	Explorer 5 Windows	Explorer 6 Windows	Explorer 5.2 Mac	Mozilla 1.6	Safari 1.2	Opera 7.50
appendData()	No	Yes	Yes	Yes	Yes	Yes

Append data to a text node Test page	x.appendData(' some extra text') Appends the string `some extra text` to x, which must be a text node.					
data The content of a text node Test page	Yes	Yes	Yes	Yes	Yes	Yes
	x.data The content of `x`, which must be a text node. The same as `x.nodeValue` . Can also be set: `x.data = 'The new text'`.					
deleteData() Delete text from a text node Test page	No	Yes	Yes	Yes	Yes	Yes
	x.deleteData(4,3) Delete some data from `x`, which must be a text node, starting at the fifth character and deleting three characters. Second argument is required.					
insertData() Insert text into a text node Test page	No	Yes	Yes	Yes	Yes	Yes
	x.insertData(4,' and now for some extra text ') Insert the string `and now for some extra text` after the fourth character into `x`, which must be a text node.					
replaceData() Replace text in a text node Test page	No	Yes	Buggy	Yes	Yes	Yes
	x.replaceData(4,3,' and for some new text ') Replace three characters, beginning at the fifth one, of node `x`, which must be a text node, by the string `and for some new text`. • Explorer 5 Mac replaces the three characters by the first three characters of the new string, while the other characters of the new string are ignored.					
substringData() Take a substring of the text in the text node Test page	No	Yes	Yes	Yes	Yes	Yes
	x.substringData(4,3) Takes a substring of `x`, which must be a text node, starting at the fifth character and with a length of three characters. Thus it's the same as the old substr() method of strings.					

Attributes

A bloody mess. Try influencing attributes in this order:

1. Try getting or setting a specific property, like `x.id` or `y.onclick`.
2. If there is no specific property, use `getAttribute()` or `setAttribute()`.
3. If even that doesn't work, try any other method or property in the table below. Most have horrible browser incompatibility patterns, though.
4. Avoid `attributes[]`. It's worse than anything else.

Note

- In my view any method or property concerning attribute nodes should also work on the `style` attribute, event handlers and custom attributes. If not I judge the method or property incomplete.

Method or property	Explorer 5 Windows	Explorer 6 Windows	Explorer 5.2 Mac	Mozilla 1.6	Safari 1.2	Opera 7.50
attributes[index] An array with the attributes of a node, accessed by index number Test page **Do not use** Use `getAttribute()` instead	Incorrect	Incorrect	More incorrect	Yes	Yes	Yes
attributes[key] An array with the attributes of a node, accessed by attribute name Test page **Do not use** Use `getAttribute()` instead	Almost	Yes	Incomplete	Yes	No	Yes
createAttribute() and setAttributeNode() Create a new attribute node and append it to an element node. Test page	No	Yes	No	Yes	No	Yes
getAttribute() Get the value of an attribute Test page	Incomplete	Incomplete	Yes	Yes	Yes	Almost
getAttributeNode() Get an attribute node Test page	No	Yes	Almost	Yes	Yes	Yes
hasAttribute() Check if a node has a	No	No	No	Yes	Yes	Incorrect

x.attributes[1]
According to Explorer this means the second `possible attribute` of node `x`. So Explorer puts its entire list of possible attributes in the attributes[] array (84 in IE 6.0 Win!)
The list of attributes is alphabetical in Explorer 5, non-alphabetical in 6. Explorer Mac doesn't put custom attributes in the list.

According to all other browsers this means the second `actual attribute` of node `x`, decidedly the saner interpretation.

- Explorer 5 Win and Mac initially give the `value` of the attribute, and not an attribute object. You can access the properties of the object, though.

x.attributes['align']
For accessing an attribute by its name. Because of the total confusion with index numbers this would be the best method for using the `attributes[]` array, except that Safari doesn't support it.

- Explorer 5 Win and Mac initially give the `value` of the attribute, and not an attribute object. You can access the properties of the object, though.
- Explorer 5 Mac cannot access custom attributes.

```
z = document.createAttribute('title');
z.value = 'Test title';
x.setAttributeNode(z)
```

where `x` is an element node.

- Explorer 5 (Win and Mac) can't create an attribute.
- Safari creates the attribute node but refuses to set its value.

x.getAttribute('align')
The value of the attribute `align` of node `x`.

- Gives weird results on `style` in Opera (its own internal styles, I presume) and Explorer gives 'object'. All other browsers sanely return the actual value of the `style` attribute of tag `x`.

x.getAttributeNode('align')
Get the attribute `align` of node `x`. This is an object, not a value.

- Explorer 5 Mac initially returns a value, not an object.

certain attribute

Test page

x.hasAttribute('align')
Is true when node `x` has an attribute 'align'. Else it is false.

- Opera is wildly off the mark. It gives `true` for an undefined `class` attribute and `false` for defined `ppk` and `onclick` attributes.

hasAttributes() Check if a node has attributes Test page	No	No	No	Yes	Yes	Yes

x.hasAttributes()
Is `true` when node `x` has attributes, `false` when it hasn't.

name The name of an attribute Test page	Minimal	Yes	Yes	Yes	Yes	Yes

x.name
The `name` of attribute `x`.

- Explorer 5 Windows supports this property only on XML attributes

removeAttribute() Remove an attribute node Test page	Incomplete	Incomplete	No	Yes	Incomplete	Incomplete

x.removeAttribute('align')
Remove the `align` attribute of node `x`.

- Safari doesn't remove style.
- Explorer Windows and Opera don't remove events.

removeAttributeNode() Remove an attribute node Test page	No	Minimal	Minimal	Yes	Yes	Yes

x.removeAttributeNode(x.attributes['align']) or
x.removeAttributeNode(x.attributes[1])
Removes the attribute node. There is no difference with `removeAttribute()`

- Explorer 6 Win and 5 Mac don't remove the attribute, but don't give an error message either.

Method or property	**Explorer 5 Windows**	**Explorer 6 Windows**	**Explorer 5.2 Mac**	**Mozilla 1.6**	**Safari 1.2**	**Opera 7.50**
setAttribute() Set the value of an attribute Test page	Incomplete	Incomplete	Incomplete	Yes	Almost	Incomplete

x.setAttribute('align','left')
Set the attribute `align` of node `x` to `left`. The name and value are both strings.

- Explorer doesn't set styles. It removes events when you try to set them.
- Opera doesn't set events
- Safari sets the new styles but doesn't remove the old ones

setAttributeNode()	See `createAttribute()`

specified Whether an attribute is specified Test page	Yes	Yes	Yes	Minimal	Minimal	Minimal

x.attributes['align'].specified or
x.attributes[1].specified
Is `true` when node `x` has an `align` attribute, `false` when it hasn't.
So it's meant to find out whether a node has a certain attribute.

This property must work on *unspecified* attributes if it is to be of any use. This is possible in Explorer, where the `attributes[]` array contains all possible attributes.
In the other browsers the array only contains the specified attributes. Hence `specified` is useless in them.

value The value of an attribute Test page	Minimal	Incomplete	Yes	Yes	Yes	Incomplete

x.value
The `value` of attribute `x`.

- Explorer 5 Windows supports this property only on XML attributes
- Explorer 6 cannot access the value of styles.
- Opera cannot access the values of events or styles.

Miscellaneous

A lot of miscellaneous methods and properties that you'll rarely need. I use only two of them in an actual script.

Method or property	Explorer 5 Windows	Explorer 6 Windows	Explorer 5.2 Mac	Mozilla 1.6	Safari 1.2	Opera 7.50
contains() Check whether an element contains another element Test page	Yes	Yes	Yes	No	Incorrect	Yes
createDocument() Create a new document Test page; main test is the Import XML script	No	No	No	Yes	Minimal	No
createDocumentFragment() Create a document fragment Test page	No	No	Minimal	Yes	Yes	Almost
documentElement The HTML tag Test page	Yes	Yes	Yes	Yes	Yes	Yes

x.contains(document.getElementById('test'))
If node `x` has a descendant with ID=test, it is `true`, else `false`.

- In Safari the method always returns `true`, even when the node is not a descendant.

x = document.implementation.createDocument('','',null)

- Safari supports the bare method. v73 could import XML, but not read out the XML and write the data into the HTML page. v80 has regressed on this point

x = document.createDocumentFragment();
x.[fill with nodes];
document.appendChild(x);

Create a fragment, add a lot of nodes to it, and then insert it into the document. Note that the fragment itself is not inserted, only its child nodes.
You may not move a node from the existing document to the document fragment.

- Opera creates the fragment but does not apply styles to the created elements.
- Explorer Mac does not allow text nodes to be appended to the fragment.

document.documentElement
Represents the HTML tag and thus the entire HTML document.

documentElement can hold interesting information about the dimensions of the document and/or the window. However, it's not exactly easy to use. See the document.body and doctypes page for more information.

Method or property	Explorer 5 Windows	Explorer 6 Windows	Explorer 5.2 Mac	Mozilla 1.6	Safari 1.2	Opera 7.50
getElementsByName() Get elements by their name attribute Test page	Incorrect and incomplete	Incorrect and incomplete	Incomplete	Yes	Incomplete	Incorrect and incomplete
hasChildNodes() Check if the node has child nodes Test page	Yes	Yes	Yes	Yes	Yes	Yes
hasFeature() Check if an element has a certain feature. Test page	No	Yes	Yes	Yes	Yes	Yes
item() Access an item in an array Test page **Not necessary in JavaScript**	Yes	Yes	Yes	Yes	Yes	Yes

x = document.getElementsByName('MapNode')
Make `x` into an array of all elements with `name="MapNode"` in the document

On my test page the `<p>`, `<input>`, `<img>` and `<ppk>` tags have this name.

- Explorer Windows and Opera only look at input and img, elements that traditionally have a name attribute, and also add an element with `id="MapNode"`.
- Explorer Mac does the same, except for the element with `id="MapNode"`
- Safari refuses the custom `<ppk>` tag

x.hasChildNodes()
Is true when node `x` has child nodes.

document.implementation.hasFeature('XML','1.0')
Is true if the browser supports XML 1.0 in. See the test page for a fuller list of possible features.

Unfortunately most browsers answer `false` to most features, even Core 1, so this method is not very useful.

document.getElementsByTagName('P').item(0)
The same as `document.getElementsByTagName('P')[0]`.

This method is meant for other programming languages where NodeLists like those returned by getElementsByTagName are not conveniently made into arrays.
You don't need item() at all.

Method or property	Explorer 5 Windows	Explorer 6 Windows	Explorer 5.2 Mac	Mozilla 1.6	Safari 1.2	Opera 7.50
normalize() Merge adjacent text nodes into one node Test page	No	Danger	Yes	Yes	Yes	Yes

x.normalize()
All child nodes of node `x` that are text nodes and have as siblings other text nodes, are merged with those. This is in fact the reverse of `splitText`: text nodes that were split, come together again.
Safari removes the incorrect whitespace splitText() caused.

This method used to crash Explorer 6. If you do the test once now, it holds stable, though. Two or more rapid clicks will still bring it down.

ownerDocument The document that 'owns' the element Test page	No	Yes	Yes	Yes	Yes	Yes
splitText() Split a text node into two text nodes Test page	Yes	Yes	Yes	Yes	Buggy	Yes

x.ownerDocument
Refers to the document object that 'owns' node `x`. This is the document node.

x.splitText(5)
Split the text node `x` at the 6th character. `x` now contains the first part (char. 0-5), while a new node is created (and becomes `x.nextSibling`) which contains the second part (char. 6-end) of the orginial text.

- Safari adds whitespace between the two text nodes.

Home Sitemap

Import XML Document

At the moment it only works in Explorer 5+ on Windows and Mozilla.

Mark Wilton-Jones wrote a script that makes XML importing work in the other browsers, by using a hidden iframe to receive the XML and then reading out the data from the hidden iframe.

Many thanks to Dylan Schiemann for putting me on the right track with the importing of XML documents.

On this page I import an XML document and then read out the data and put them in a table on the page.

I think importing XML documents will become more and more important in the future. You can manage the XML document as a kind of database, while the HTML document contains all information about the displaying of the data.

Anyway, try it first by clicking the link and loading some crucial information about the Roman emperors of the Julian-Claudian dynasty. You can also view the XML document separately.

First I import the document *emperors.xml*, then I enter the XML document through the W3C DOM and extract the data I need, while building a table to display the data.

The script

```
function importXML()
{
	if (document.implementation && document.implementation.createDocument)
	{
		xmlDoc = document.implementation.createDocument("", "", null);
		xmlDoc.onload = createTable;
	}
	else if (window.ActiveXObject)
	{
		xmlDoc = new ActiveXObject("Microsoft.XMLDOM");
		xmlDoc.onreadystatechange = function () {
			if (xmlDoc.readyState == 4) createTable()
		};
	}
	else
	{
		alert('Your browser can\'t handle this script');
		return;
	}
	xmlDoc.load("emperors.xml");
}

function createTable()
{
	var x = xmlDoc.getElementsByTagName('emperor');
	var newEl = document.createElement('TABLE');
	newEl.setAttribute('cellPadding',5);
	var tmp = document.createElement('TBODY');
	newEl.appendChild(tmp);
	var row = document.createElement('TR');
	for (j=0;j<x[0].childNodes.length;j++)
	{
		if (x[0].childNodes[j].nodeType != 1) continue;
		var container = document.createElement('TH');
		var theData = document.createTextNode(x[0].childNodes[j].nodeName);
```

```
                container.appendChild(theData);
                row.appendChild(container);
        }
        tmp.appendChild(row);
        for (i=0;i<x.length;i++)
        {
                var row = document.createElement('TR');
                for (j=0;j<x[i].childNodes.length;j++)
                {
                        if (x[i].childNodes[j].nodeType != 1) continue;
                        var container = document.createElement('TD');
                        var theData =
document.createTextNode(x[i].childNodes[j].firstChild.nodeValue);
                        container.appendChild(theData);
                        row.appendChild(container);
                }
                tmp.appendChild(row);
        }
        document.getElementById('writeroot').appendChild(newEl);
}
```

Importing the XML

First of all I import the XML document and make it accessible through the object `xmlDoc`. When the document has finished loading, I want the script `createTable()` that construes the table to be executed immediately. Of course, the coding for all this is browser specific.

Clicking the link activates the function `importXML`.

```
function importXML()
{
```

Mozilla

Netscape imports an XML document through the method `document.implementation.createDocument()`. First check if `document.implementation` is supported, then check if `document.implementation.createDocument()` is supported. Explorer 5 on Mac also supports document.implementation, but not the createDocument method, so it shouldn't execute this script.

```
        if (document.implementation && document.implementation.createDocument)
        {
```

Then create the document and give it an onLoad event handler: as soon as the document has been loaded the script `createTable()` is executed, creating the table:

```
                xmlDoc = document.implementation.createDocument("", "", null);
                xmlDoc.onload = init;
        }
```

Explorer

Explorer on Windows doesn't support document.implementation . Instead, you must create an Active X Object that will contain the XML document. So we see if the browser can create ActiveXObjects:

```
        else if (window.ActiveXObject)
        {
```

If it does, we can proceed by creating the object

```
                xmlDoc = new ActiveXObject("Microsoft.XMLDOM");
```

Unfortunately there's no onLoad event for this object. To see if it's ready we should use the MS proprietary ReadyStateChange event. I don't quite understand all this myself, but it works. When the onReadyStateChange event handler fires, the `readyState` has a value between 1 and 4. 4 means that all data has been received (= onLoad). So if it's 4, start creating the table.

```
                xmlDoc.onreadystatechange = function () {
                        if (xmlDoc.readyState == 4) createTable()
                };
        }
```

Other browsers

If the browser supports neither way, give an alert and end everything:

```
        else
        {
                alert('Your browser can\'t handle this script');
                return;
        }
```

Load document

Finally, load the actual document. Surprisingly, the command for this is the same in both browsers:

```
        xmlDoc.load("emperors.xml");
}
```

Now we wait for the XML document to be loaded completely, then the function `createTable()` is started up.

Strangely, Mozilla sometimes only accepts a URL to the XML file that is *relative* to the page with the script, which in practice means that you can only access local XML files. Only the Linux version does accept absolute paths in all cases.

You might be able to solve the problem by putting the page containing the script on a real web server instead of a local host. However, this does not help in all cases.

Creating output

This function is entirely specific for the XML document I created. Each XML document is different, each way of displaying the content is different, so the function I wrote is only an example.

The XML document

The XML document consists of nodes named `<emperor>`, which all contain the same children. As an example, this is the structure of the first node:

```
                        <emperor>
                            |
        ------------------------------------------
        |                   |                    |
    <name>              <rule>               <death>
        |                   |                    |
    Augustus            27BC-14AD            Peaceful
```

In the script below I assume that every emperor has this structure. If one hasn't, it could lead to huge problems, but this way I keep the script simple.

Creating the table

Function `createTable()` starts by creating an array of all the tags `<emperor>` in the XML document. For each of these tags I want to create a TR containing several TD's with the data.

```
function createTable()
{
        var x = xmlDoc.getElementsByTagName('emperor');
```

Then we create a new TABLE with CELLPADDING=5. Note the special spelling `cellPadding`, Explorer requires this and it doesn't hurt Netscape.

```
        var newEl = document.createElement('TABLE');
        newEl.setAttribute('cellPadding',5);
```

Explorer requires that we also create a TBODY and append it to the table. Don't ask me why, I think TBODY is a completely useless tag, but without it the example doesn't work in Explorer.

```
        var tmp = document.createElement('TBODY');
        newEl.appendChild(tmp);
```

First of all, a row with TH's for the headers. Create a TR

```
        var row = document.createElement('TR');
```

then go through the childNodes of the first emperor.

```
        for (j=0;j<x[0].childNodes.length;j++)
        {
```

A problem here: the XML document looks like this:

```
```

```
		<emperor>
			<name>
```

which means that Netscape considers the empty text node between `emperor` and `name` as the first child of `emperor`. Since these nodes are only a nuisance, we have to check if the nodeType of the childNode is 1 (= it's a tag). If it isn't, continue with the next child:

```
				if (x[0].childNodes[j].nodeType != 1) continue;
```

Create a container element (TH):

```
				var container = document.createElement('TH');
```

then read out the name of the childNode (ie. `name`, `rule` and `death`). I want these names to be printed inside the TH. So first I append the name to the TH, then I append the container to the TR

```
				var theData = document.createTextNode(x[0].childNodes[j].nodeName);
				container.appendChild(theData);
				row.appendChild(container);
		}
```

Finally, append the row to the TBODY tag

```
		tmp.appendChild(row);
```

Then we go through all elements `emperor` and create a TR for each one

```
		for (i=0;i<x.length;i++)
		{
			var row = document.createElement('TR');
```

Go through the childNodes of each emperor, check if it's a node (tag) and create a TD to hold the data

```
			for (j=0;j<x[i].childNodes.length;j++)
			{
				if (x[i].childNodes[j].nodeType != 1) continue;
				var container = document.createElement('TD');
```

Then extract the actual data. Remember that the text inside a tag is the nodeValue of the first childNode of that tag

```
				var theData =
document.createTextNode(x[i].childNodes[j].firstChild.nodeValue);
```

Append the text to the TD and the TD to the TR

```
```

```
                        container.appendChild(theData);
                        row.appendChild(container);
                }
```

and when you've finished going through the emperor, append the TR to the TBODY

```
                tmp.appendChild(row);
        }
```

Finally, when you've gone through each emperor, append the TABLE to the special P with ID="writeroot" I created:

```
        document.getElementById('writeroot').appendChild(newEl);
}
```

and a table has been magically created.

Home Sitemap

Edit text

The TEXTAREA behaves strangely in Explorer 5 on Mac: it falls over the other elements, instead of stretching up the page, and initially not all text is shown. Move your mouse up to see all the text. It doesn't register hard returns, either.

Opera 7.50 gives one error message.

Explorer and Opera refuse to accept block level elements inside the test paragraphs.

On this page I give a script for updating a page which can be very useful in a content management system. Click on any paragraph and you can edit the text. When you're done, press the button and the new text shows up normally.

Warning: This script **crashes** Mozilla 1.4 and lower.

Example

This entire page is the example. Click on any paragraph, edit it, then press 'Ready'. Your changes will show up in the page.

Unfortunately your browser *can't handle* the script so nothing will happen.

Problems

First of all some problems I encountered. I want to use a textarea as the edit field. First I didn't get the content into the TEXTAREA. An alert reader discovered that Mozilla needs the `value` to be set *after* inserting the textarea into the document.

Besides, the content doesn't wrap nicely in Mozilla, it retains the hard returns from the source code. I experimented with several values of the `WRAP` attribute, but doing nothing gave the best result in the end.

The worst problem is sending the altered text back to the server, something any content management system will want to do.
Readers have sent me various ingenious suggestions to do this. However, since it cannot be done through JavaScript (yet) I don't treat the solution to this problem on my site. So **please don't mail me** that you found this or that way to do it: it's certainly possible, but I'm only interested in a pure JavaScript solution without any server side scripts.

The script

```
var editing  = false;

if (document.getElementById && document.createElement)
{
        var butt = document.createElement('BUTTON');
        var buttext = document.createTextNode('Ready!');
        butt.appendChild(buttext);
        butt.onclick = saveEdit;
}

function catchIt(e)
{
        if (editing) return;
        if (!document.getElementById || !document.createElement) return;
        if (!e) var obj = window.event.srcElement;
        else var obj = e.target;
        while (obj.nodeType != 1)
        {
                obj = obj.parentNode;
        }
        if (obj.tagName == 'TEXTAREA' || obj.tagName == 'A') return;
        while (obj.nodeName != 'P' && obj.nodeName != 'HTML')
        {
                obj = obj.parentNode;
        }
        if (obj.nodeName == 'HTML') return;
        var x = obj.innerHTML;
        var y = document.createElement('TEXTAREA');
        var z = obj.parentNode;
        z.insertBefore(y,obj);
        z.insertBefore(butt,obj);
        z.removeChild(obj);
        y.value = x;
        y.focus();
        editing = true;
}

function saveEdit()
{
        var area = document.getElementsByTagName('TEXTAREA')[0];
        var y = document.createElement('P');
        var z = area.parentNode;
        y.innerHTML = area.value;
        z.insertBefore(y,area);
        z.removeChild(area);
        z.removeChild(document.getElementsByTagName('button')[0]);
        editing = false;
}

document.onclick = catchIt;
```

Explanation

We set a flag `editing` to false. This shows whether the user is currently editing a paragraph. Initially he isn't, of course.

```
var editing  = false;
```

Create the button

Then we create the 'Ready' button that we'll need several times. For this we need the most advanced scripting, so first some object detection:

```
if (document.getElementById && document.createElement)
{
```

If this is a modern browser, create a BUTTON node

```
        var butt = document.createElement('BUTTON');
```

and its text

```
        var buttext = document.createTextNode('Ready!');
```

Append the text to the button so that it is shown

```
        butt.appendChild(buttext);
```

Finally add an onClick event that calls the function `saveEdit()`.

```
        butt.onclick = saveEdit;
}
```

The button now resides in the variable `butt`. We can call it if we need it.

P to TEXTAREA

Later on we'll define a general onClick event for the entire pages. All these events are sent to the function `catchIt()`.

```
function catchIt(e)
{
```

First of all, check if the user is currently editing a paragraph. If he is (`editing` is true) end the

function.

```
if (editing) return;
```

Support detection.

```
if (!document.getElementById || !document.createElement) return;
```

Then we find the target of the event,

```
if (!e) var obj = window.event.srcElement;
else var obj = e.target;
```

Now we have the target, but a problem is that Mozilla considers the text node (and not the containing P) to be the target. Therefore we go up through the DOM tree as long as the current node is not a tag (as long as its nodeType is not 1).

```
while (obj.nodeType != 1)
{
	obj = obj.parentNode;
}
```

Now we have ended up with a tag. If this is the textarea tag the user has clicked in the edit box to edit the text. If it's a link, it should be clickable as usual. In these cases nothing should happen so we end the function.

```
if (obj.tagName == 'TEXTAREA' || obj.tagName == 'A') return;
```

Once again we go up through the DOM tree to find either a P or the HTML tag as the ultimate ancestor of the target of the click.

```
while (obj.nodeName != 'P' && obj.nodeName != 'HTML')
{
	obj = obj.parentNode;
}
```

If we find the HTML tag the user has clicked outside a paragraph and nothing should happen. We end the function.

```
if (obj.nodeName == 'HTML') return;
```

After this check we finally are certain that the user has clicked on a paragraph and that he wants to edit it. Take the innerHTML of the paragraph and store it.

```
var x = obj.innerHTML;
```

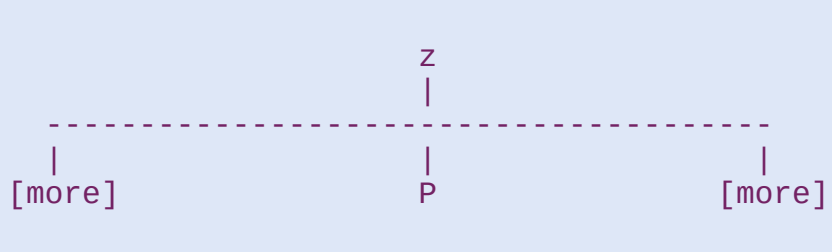
Create a new TEXTAREA element and store it.

```
var y = document.createElement('TEXTAREA');
```

Then take the parent node of the paragraph.

```
var z = obj.parentNode;
```

The situation is now:



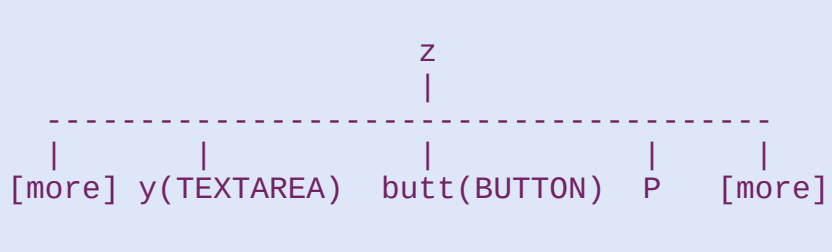
Insert the new TEXTAREA just before the paragraph in its parent node.

```
z.insertBefore(y,obj);
```

Do the same for the 'Ready' button.

```
z.insertBefore(butt,obj);
```

Now the tree looks like this:



Remove the paragraph itself. Now it seems as if the textarea and button have taken the place of the paragraph.

```
        z.removeChild(obj);
```

Only now, after inserting it, we can put the inner HTML of the P in the TEXTAREA. Doing this before insertion is impossible in Mozilla.

```
        y.value = x;
```

Put the focus on the textarea for user convenience

```
        y.focus();
```

and set `editing` to true: user is busy editing.

```
        editing = true;
}
```

TEXTAREA to P

When the user clicks on the 'Ready' button we should do the reverse. The function `saveEdit()` takes care of everything.

```
function saveEdit()
{
```

Get the TEXTAREA (this assumes there is only one such field in the entire page).

```
        var area = document.getElementsByTagName('TEXTAREA')[0]
```

Create a new paragraph and store it.

```
        var y = document.createElement('P');
```

Find the parent node of the textarea: the paragraph should be appended to it.

```
        var z = area.parentNode;
```

Set the innerHTML of the new paragraph to the value of the textarea.

```
        y.innerHTML = area.value;
```

Then insert the new paragraph just before the textarea.

```
        z.insertBefore(y,area);
```

Remove the textarea.

```
        z.removeChild(area);
```

Remove the 'Ready' button (again, this assumes there is only one button on the page).

```
        z.removeChild(document.getElementsByTagName('button')[0]);
```

Finally, set `editing` to false: the user has stopped editing.

```
        editing = false;
}
```

Event

Outside and after the functions, set a general onclick event for the entire page calling the function `catchIt()`:

```
document.onclick = catchIt;
```

Home Sitemap

Get Styles

The example script doesn't work in Safari 1.0

Sometimes you'll want to see what styles the default document view has. For instance, you gave a paragraph an width of 50%, but how do you see how many pixels that is in your users' browser?

In addition, sometimes you want to read out the styles applied to an element by embedded or linked style sheets. The `style` property only reflects the *inline* styles of an element, so if you want to read out other styles you have to resort to other means.

Return the paragraph to this place.

This is our test paragraph with ID="test", on which we're going to try our scripts.
For good measure I've also added a
.

This is the style sheet of the test paragraph:

```
#test {font-size: 16px;
        padding: 10px;
        width: 50%;
        border-width: 1px;
        border-style: solid;
        border-color: #cc0000;
}
```

offset

Before going to the tricky bits, first a nice shortcut that has been inserted into both Mozilla and Explorer: `offsetSomething`. Using these few properties you can read out the most important information about the styles the paragraph currently has.

As an example, get the offsetWidth of the test paragraph. You'll see how many pixels its width is at the moment. To test it some more, resize the window (the paragraph, having a width of 50%, will also resize) and try again.

The script is quite simple:

```
function getOff()
{
        x = document.getElementById('test');
        return x.offsetWidth;
}
```

and we alert the returned value: `alert('offsetWidth = ' + getOff())`.

Now you have the width of the paragraph in the user's browser and you can start calculating things. There are several more offsets that you can read out:

Property	Meaning
offsetHeight	height in pixels
offsetLeft	distance of paragraph from the left, in pixels (left of what? see below)
offsetTop	distance of paragraph from the top, in pixels (top of what? see below)
offsetWidth	width in pixels

Please note that these properties are *read-only*, you cannot change the offsetWidth of an element directly.

To show you what you can do, I've prepared a little example script. First of all, please move the test paragraph to this area, so you can see what's happening.

Then we're going to add 100 pixels to its width. If we look at the offsetWidth again, we'll see that it's changed. You can also remove 100 pixels from the width.

If you view this example in both browsers, you'll note that in Explorer the new width is the old width + 100px, but in Mozilla it isn't. That's because Mozilla is more standards-conforming here: as per the spec it counts only the width of the actual content as offsetWidth, while Explorer also adds the padding and the border. Even though Mozilla is correct here, I tend to favour the Explorer approach because it's more intuitive.

Anyway, this script is also simple if you don't mind the Mozilla/Explorer incompatibility:

```
function changeOff(amount)
{
        var y = getOff();
        x.style.width = y + amount;
}
```

We hand it the `amount` of pixels it should expand (100 or -100 in our example), then use `getOff()` to load the paragraph into `x` and to get the current offsetWidth. Finally we change the width to offsetWidth plus amount.

offsetTop and -Left

The `offsetTop` and `offsetLeft` properties have their own possibilities and quirks. See the Find position page for more information.

getStyle

As we've seen the offset properties are limited. The browsers give us a more general way of accessing default style information, but unfortunately it is less reliable and less generally usable than the offset properties.

Mozilla

Mozilla expects CSS syntax, not JavaScript syntax. For instance, to get the font size you must use `font-size`, as in CSS, and not `fontSize`, as in JavaScript.

Explorer

In Explorer we can get most of the styles, but unfortunately we sometimes need to be very exact. In this example `border` doesn't work, you'll need to get `borderStyle`, `borderWidth` and `borderColor`.

> Please remember that to access the style property `border-width` we need to spell it `borderWidth` in JavaScript, because the dash can be mistaken for a minus sign. This goes for all style properties with a dash in them.

In addition, Explorer often gives `auto` (for the `top` property, for instance). Although this is very true (the natural flow of the page determines where the top of the paragraph will be), it's not very useful information. Another reason to stick with the `offset` properties whenever possible.

Try it

If you wish, you can again move the test paragraph so you can see it better. Then fill in a style property in the form field below and get the style.

> In Explorer you can try the border properties (see above), `fontSize`, `fontFamily`, `margin`, `padding`, `backgroundColor`, `backgroundImage` and some more properties.
>
> Remember that Mozilla expects `font-size`, `font-family` and `background-color`.

The script

The script is once again fairly simple:

```
function getStyle(el,styleProp)
{
        var x = document.getElementById(el);
        if (window.getComputedStyle)
                var y = window.getComputedStyle(x,null).getPropertyValue(styleProp);
        else if (x.currentStyle)
                var y = eval('x.currentStyle.' + styleProp);
        return y;
}
```

First we pass the function the ID of the HTML element and the style property we wish to access

```
function getStyle(el,styleProp)
{
```

Then we store the HTML element in `x`:

```
        var x = document.getElementById(el);
```

First the Mozilla way: the `getComputedStyle()` method:

```
        if (window.getComputedStyle)
                var y = window.getComputedStyle(x,null).getPropertyValue(styleProp);
```

Then the Explorer way: the `currentStyle` of the HTML element:

```
        else if (x.currentStyle)
                var y = eval('x.currentStyle.' + styleProp);
```

Finally return `y` to whichever function asked for it (in this page it's the function `prepare()` that is called when you submit the form or click the link '*get the style*').

```
        return y;
}
```

Although this function doesn't yet work well, it's the best you can do.

Home Sitemap

Change style sheet

Opera 7 doesn't support the `styleSheets[]` array.

On this page I try to change the background colour of a PRE, not by accessing the element directly but by changing the entire style sheet of the page. Unfortunately browser incompatibilities are so severe that this script isn't really usable in practice yet.

Please note the difference between traditional DHTML and this example script. While in DHTML you change the styles of one specific HTML element on the page (usually identified by an ID), this example changes the style sheet of the entire document.

See also the W3C DOM - CSS compatibility table.

Definitions

A page contains one or more *style sheets* which in turn contain one or more *rules* which contain the actual style declarations. These are the styles in this page:

```
<link rel="stylesheet" href="../quirksmode.css">
<style>
<!--
@import url("test.css");

p,h2,h3 {
        padding-right: 10px;
}

pre.test + * {
        margin-right: 20%;
}

pre.test {
        background-color: #ffffff;
}
-->
</style>
```

The purpose of our test script will be to change the white background colour of the `pre.test` to its normal value `transparent`.

Style sheet

All linked and embedded style sheets are available through the `document.styleSheets` array. `quirksmode.css`, the general style sheet for the entire site, is `document.styleSheets[0]`. The special style block I added to this page is `document.styleSheets[1]`. We're going to do our tests on this special block of styles.

cssRules[] and rules[]

A rule is one set of style declarations for one or more elements. There are two ways to access these rules. W3C insists on the `cssRules` array while Microsoft has decided on the `rules` array. Both arrays work with index numbers, the first rule is `(css)Rules[0]`, the second one `[1]` etc.

> Please note that Explorer 5 on Mac supports both arrays.

Workaround:

```
var theRules = new Array();
if (document.styleSheets[1].cssRules)
        theRules = document.styleSheets[1].cssRules
else if (document.styleSheets[1].rules)
        theRules = document.styleSheets[1].rules
```

Now `theRules` contains all style rules.

Number of rules

This is the style sheet:

```
@import url("test.css");

p,h2,h3 {
        padding-right: 10px;
}

pre.test + * {
        margin-right: 20%;
}

pre.test {
        background-color: #ffffff;
}
```

Now at first you'd say that the special test style sheet has four rules: `@import`, then `p,h2,h3`, then `pre.test + *` and finally `pre.test`. Unfortunately that is not the case. Here are the selectors of the rules according to your browser:

```
Your browser doesn't support document.styleSheets
```

- **Safari** sees 4 rules:
  0. `undefined`
  1. `P`
  2. `PRE.test[CLASS~="test"] + *`
  3. `PRE.test[CLASS~="test"]`

  Note: UPPER case.

- **Explorer Windows** sees 5 rules:

0. `P`
1. `H2`
2. `H3`
3. `UNKNOWN`
4. `PRE.test`

Note: UPPER case.

- **Explorer Mac** also sees 5 rules:
  0. `P`
  1. `H2`
  2. `H3`
  3. `PRE.test *` (note the absence of the `+`)
  4. `PRE.test`

  Note: UPPER case.

- **Mozilla** sees 6 rules:
  0. `undefined`
  1. `p`
  2. `h2`
  3. `h3`
  4. `pre.test + *`
  5. `pre.test`

  Note: lower case.

A fine mess.

1. `undefined` refers to the imported style sheet, which indeed does not have a selector. Explorer doesn't see the imported style sheet, though.

2. Then, according to Explorer and Mozilla, the `p,h2,h3` rule is not one rule but three separate ones, while Safari sees it as a rule for `P` only. As far as I understand the spec both variations are incorrect behaviour.
   "The selector consists of everything up to (but not including) the first left curly brace ({)."
   No browser adheres to this part of the spec.

3. The `pre.test + *` is not recognized by Explorer Windows since it doesn't support this selector. Fair enough. Explorer Mac's behaviour, however, is weird. It changes the selector to `pre.test *`, which has a completely different meaning. Serious, very serious.

4. Finally the `pre.test` is reasonably well supported, except that Safari adds unnecessary selection syntax.

So to access the `pre.test` rule Safari needs `cssRules[3]`, Explorer `rules[4]` and Mozilla `cssRules[5]`. Lovely, isn't it?

No keys

So we encounter serious problems when trying to access the rule by index number. What I'd rather do is access them by keys, where the selector is the key. Something like:

```
document.styleSheets[1].cssRules['PRE.test']
```

so that I can access the rule of the PRE regardless of where in the style sheet it is. It does not seem to have occurred to either W3C or the browser vendors that web developers have a need for such a way of accessing rules. All documentation is completely silent on this point.

This failure means that it's nearly impossible to access the rule you want.

Style declarations

Let's pretend we have accessed the desired rule. Now we change one of its style declarations. The general syntax for this is

```
rule.style.color = '#0000cc';
```

The W3C approved way is

```
rule.style.setProperty('color','#00cc00',null);
```

Since it is vastly simpler I prefer the `style.color` syntax above `setProperty()`.

Example

Try changing the background colour of the PRE's below.

```
Test PRE
```

The script

To avoid the rules problems noted above I use the fact that the `pre.test` rule is the last rule in the stylesheet. It's an ugly kludge, but it's the only way to get a proper cross-browser test case.

```
function changeIt()
{
        if (!document.styleSheets) return;
        var theRules = new Array();
        if (document.styleSheets[1].cssRules)
                theRules = document.styleSheets[1].cssRules
        else if (document.styleSheets[1].rules)
                theRules = document.styleSheets[1].rules
        else return;
        theRules[theRules.length-1].style.backgroundColor = 'transparent';
}
```

Home Sitemap

Table of Contents script

This script crashes Explorer 5.2 on Mac.

In other Explorers on Mac clicking on the 'Contents' bar is only possible with the page scrolled completely up.

The script works in Explorer 5.0 Windows, but on my site this browser subsequently hides all floated elements (like this one) and most long code examples. Therefore I disabled the script in this browser. It might work perfectly in other situations, though.

On this page I explain the Table of Contents script that runs in all pages on this site. It generates a list of all `h3`'s and `h4`'s on the page and offers links to them.

Principles

The main problem was getting both sorts of headers *in the order they appear in the source code*. I couldn't use `getElementsByTagName()`, since that would have destroyed this order. Therefore the script searches for all `h3`'s and `h4`'s that are children of the `body`.

On this site I use `h3`'s for main headers and `h4`'s for sub-headers. In the generated table of contents I want to show this hierarchy. Initially I wanted to use nested `ul`s, but that would have made the script much more complicated. Therefore I decided to fake them.

The script

```
function createTOC()
{
	if (top.bugRiddenCrashPronePieceOfJunk) return;
	var x = document.body.childNodes;
	var y = document.createElement('div');
	y.id = 'toc';
	var a = y.appendChild(document.createElement('span'));
	a.onclick = showhideTOC;
	a.innerHTML = 'Contents';
	var z = y.appendChild(document.createElement('div'));
	z.onclick = showhideTOC;
	var toBeTOCced = new Array();
	for (var i=0;i<x.length;i++)
	{
		if (x[i].nodeName.indexOf('H') != -1)
			toBeTOCced.push(x[i])
	}

	if (toBeTOCced.length < 2) return;

	for (var i=0;i<toBeTOCced.length;i++)
	{
		var tmp = document.createElement('a');
```

```
                tmp.innerHTML = toBeTOCced[i].innerHTML;
                tmp.href = '#link' + i;
                tmp.className = 'page';
                z.appendChild(tmp);
                if (toBeTOCced[i].nodeName == 'H4')
                        tmp.className += ' indent';
                var tmp2 = document.createElement('a');
                tmp2.id = 'link' + i;
                if (toBeTOCced[i].nodeName == 'H2')
                {
                        tmp.innerHTML = 'Top';
                        tmp.href = '#top';
                        tmp2.id = 'top';
                }
                toBeTOCced[i].parentNode.insertBefore(tmp2,toBeTOCced[i]);
        }
        document.body.insertBefore(y,document.body.childNodes[2]);
}

var TOCstate = 'none';

function showhideTOC()
{
        TOCstate = (TOCstate == 'none') ? 'block' : 'none';
        document.getElementById('toc').lastChild.style.display = TOCstate;

}
```

Explanation

First of all I detect Explorer 5.2 Mac by a special variable `bugRiddenCrashPronePieceOfJunk`. See the making of QuirksMode.org page for more sordid details.

```
function createTOC()
{
        if (top.bugRiddenCrashPronePieceOfJunk) return;
```

Preparation

Then I take all children of the `body` and create a `div id="toc"` to contain the table of contents.

```
        var x = document.body.childNodes;
        var y = document.createElement('div');
        y.id = 'toc';
```

I append a `span` to it with the text 'Contents'. Clicking on this element runs the `showhideTOC()` script I explain below.

```
        var a = y.appendChild(document.createElement('span'));
        a.onclick = showhideTOC;
```

```
        a.innerHTML = 'Contents';
```

The I create yet another `div` to contain the actual links. This `div` is initially hidden but can be made visible. Clicking on this `div` (which really means: clicking on any link) also calls `showhideTOC()`.

```
        var z = y.appendChild(document.createElement('div'));
        z.onclick = showhideTOC;
```

Finding the headers

Then I create a new array `toBeTOCced` and go through all children of the `body` tag.

```
        var toBeTOCced = new Array();
        for (var i=0;i<x.length;i++)
        {
```

I want to insert an internal anchor (`<a name>`) before every header as a target for the link in the table of contents. It is very important to *first* create an array with all headers, before doing anything else. If I'd immediately start adding internal anchors my function would create an infinite loop:

1. The script finds a header. Say it is the 12th child of the `body`.
2. Create an internal anchor just before this header. Now *this anchor becomes the 12th child of the `body`*.
3. Go on to the 13th child. Hey, this is a header! In fact, it's the same header as in step 1. The creation of the anchor has moved it up one place.
4. Create an internal anchor just before this header.
5. Go on to the 14th child. Again the same header.
6. etc. etc. The browser never wakes up.

So I `push` all headers into the array `toBeTOCced` first. I create a list of all headers in the page before changing the page.

```
                if (x[i].nodeName.indexOf('H') != -1)
                        toBeTOCced.push(x[i])
        }
```

Now `toBeTOCced` contains all headers *in the order they appear in the source code*.

The very first element of the array is the `h2` that contains the page title. This is deliberate: I want a 'Top' link as the very first link in my table of contents, and linking it to the page title is the easiest way of doing this.

On the other hand, if the page doesn't contain any `h3` or `h4` I don't want to create a table of contents at all. So if the array contains 0 or only 1 element, I end the function. (That one element is of course the `h2`, which doesn't count).

```
```

```
        if (toBeTOCced.length < 2) return;
```

Creating the table of contents

I can finally start creating my table of contents. Go through `toBeTOCced`.

```
        for (var i=0;i<toBeTOCced.length;i++)
        {
```

Create a new `a class="page"` element. Its innerHTML becomes the content of the current header. Its `href` becomes '#link' plus the current value of `i`. This ensures a unique link name for each TOC link.

> Why innerHTML? A few of my headers contain HTML, and I want this HTML to appear in the TOC link, too.

```
                var tmp = document.createElement('a');
                tmp.innerHTML = toBeTOCced[i].innerHTML;
                tmp.href = '#link' + i;
                tmp.className = 'page';
```

Append the new TOC link to the TOC.

```
                z.appendChild(tmp);
```

If the current element is an `h4` I add a class 'indent'. It is meant for faking nested `li`s.

```
                if (toBeTOCced[i].nodeName == 'H4')
                        tmp.className += ' indent';
```

Create another `a` element and give it the same `id` as the TOC link, but of course without the '#'. This is the internal anchor the link in the TOC jumps to.

```
                var tmp2 = document.createElement('a');
                tmp2.id = 'link' + i;
```

If the current element is the `h2` page header I use the special value 'Top' for its innerHTML and internal anchor. I don't want to see the page title repeated in the TOC.

```
                if (toBeTOCced[i].nodeName == 'H2')
                {
                        tmp.innerHTML = 'Top';
                        tmp.href = '#top';
                        tmp2.id = 'top';
```

```
                }
```

Finally insert the internal anchor into the document, just before the header it belongs to.

```
                toBeTOCced[i].parentNode.insertBefore(tmp2,toBeTOCced[i]);
        }
```

When we've gone through all headers the TOC is ready. Append the entire TOC block to the document *as its third child*.

```
        document.body.insertBefore(y,document.body.childNodes[2]);
}
```

Why the third child? Because of Explorer on Windows. I give the entire TOC a `position: fixed` and place it in the upper right corner. Unfortunately Explorer on Windows doesn't support this, so it shows the TOC at the place I insert it into the document.

I want Explorer on Windows to show it directly after the page title. Although initially the page title is the first element in the page, I *insert an internal anchor before it*, causing it to become the second element. Therefore the TOC should become the third one.

The other browsers don't care where I insert it: they'll show it fixed in the upper right corner anyway.

showhideTOC

Finally a very simple function to show and hide the TOC content. A variable to remember its current state, which initially is 'none'.

```
var TOCstate = 'none';
```

Then a function to switch its state from 'none' to 'block' or vice versa.

```
function showhideTOC()
{
        TOCstate = (TOCstate == 'none') ? 'block' : 'none';
        document.getElementById('toc').lastChild.style.display = TOCstate;

}
```

This function is called whenever the user clicks on the 'Contents' span, so he can toggle the display of the TOC. In addition, it's called whenever the user clicks on a link. This immediately hides the TOC.

Home Sitemap

Image replacement

There don't seem to be any browser incompatibilities.
See Shaun Inman's IFR for another technique, this one using Flash.

Recently there has been much discussion about the "Fahrner Image Replacement". Although I liked the concept, I disliked the countless CSS variants that have popped up, because they are inherently unsafe and hacky. I feel we should use JavaScript instead of CSS. This page explains my script.

The idea of FIR is simple: Initially a page is served with text as the content of its headers. However, some clever (or not so clever) hacks are applied that hide this text and show an image instead. Theoretically browsers that can't handle advanced CSS, as well as screen readers, should not execute the replacement. Therefore the headers would remain accessible under all circumstances.

Assumptions

Unfortunately all CSS variants are making assumptions about screen reader behaviour. Joe Clark has admirably summarized the problem and his research. Screen readers turn out to support the CSS bits that everyone assumed they wouldn't support, so they hide the text. They can't show the image either, so accessibility is severely compromised. Clark's conclusion is that any CSS variant is inherently unsafe.

When I read a question about using JavaScript to enhance FIR, I realized that everyone was approaching this problem from the wrong way. We shouldn't use CSS at all to replace text by images. Instead, such a replacement job is a typical JavaScript task. In fact, it's ridiculously easy.

Nonetheless I'm not any better than the myriad CSS hackers who have preceded me: I make assumptions about screen readers. Although to me my assumption seems safer than all others, it may still be totally wrong. That's the chance you take when writing code for user agents you can't test in.

My assumption: Screen readers do not download images.

Example

All `h3` texts on this page are replaced by images, if your browser allows it. The images are not examples of rarified design beauty. That's because I'm not a designer.

If you switch off images your browser will show normal `h3`s.

The script

I add an `id` attribute to any `h3` that should be affected. It contains the source of the image to be displayed instead of the text. This script assumes that every `h3` with an `id` should be replaced by an image.

```
<h3 id="fir_script">The script</h3>
```

This `h3` expects the image `pix/fir_script.gif` for a replacement.

Then I run this script `onload`.

```
function init()
{
        var W3CDOM = (document.createElement && document.getElementsByTagName);
        if (!W3CDOM) return;
        var test = new Image();
        var tmp = new Date();
        var suffix = tmp.getTime();
        test.src = 'pix/fir_assumptions.gif?'+suffix;
        test.onload = imageReplacement;
}

function imageReplacement()
{
        replaceThem(document.getElementsByTagName('h3'));
}

function replaceThem(x)
{
        var replace = document.createElement('img');
        for (var i=0;i<x.length;i++)
        {
                if (x[i].id)
                {
                        var y = replace.cloneNode(true);
                        y.src = 'pix/' + x[i].id + '.gif';
                        y.alt = x[i].firstChild.nodeValue;
                        x[i].replaceChild(y,x[i].firstChild);
                }
        }
}
```

Browser compatibility

	Explorer 6 Windows	Explorer 5.2 Mac	Mozilla 1.6	Safari 1.0	Opera 7.50
Images on	Shows images	Shows images	Shows images	Shows images	Shows images
Images off	Shows text	Shows text	Shows text	How do I turn them off?	Shows text

Explanation

onload you should run the function `init()`.

```
function init()
{
```

The first thing we do is checking for W3C DOM support. If it's absent we stop the script.

```
        var W3CDOM = (document.createElement && document.getElementsByTagName);
        if (!W3CDOM) return;
```

Detecting image support

Detecting W3C DOM support is not enough, though. We also have to see if the browser supports images. If it doesn't our script shouldn't run, since it would create all kinds of odd effects.

Therefore we generate an image and set its `src`. This is a classic preloading trick: the browser now fetches the image. This is the only support detection we need. **If this test image loads succesfully, the browser supports images**. I *assume* that a browser that doesn't support images doesn't download them, either.

So we activate the main image replacement routine only when this image has been loaded, ie. after its `load` event has taken place.

A browser problem here. If you return to this page by using the Back button, Explorer on Windows does *not* fire the `onload` event of cached images. Therefore we have to make sure that it fetches a new image every time by adding a suffix that contains the current time in milliseconds.

I don't like this feature of the script, it causes unnecessary HTTP requests, but at the moment I don't see a way around it.

```
        var test = new Image();
        var tmp = new Date();
        var suffix = tmp.getTime();
        test.src = 'pix/fir_assumptions.gif?'+suffix;
        test.onload = imageReplacement;
}
```

Defining the target elements

Once the test image has been loaded, it's safe to do wholesale image replacement. The function `imageReplacement` is called. It's an intermediate function to allow you to define several areas of the document where images should be replaced. This example script just replaces `h3`s.

```
function imageReplacement()
{
        replaceThem(document.getElementsByTagName('h3'));
}
```

However, if you'd also like to replace, say, all links in `div id="nav"`, do

```
function imageReplacement()
{
        replaceThem(document.getElementsByTagName('h3'));
        replaceThem(document.getElementById('nav').getElementsByTagName('a'));
```

```
}
```

Image replacement

The function `replaceThem()` handles the actual image replacement. It is handed an array which it searches for elements with an `id`. If it finds one, the script replaces the element's `firstChild` (the text node containing the text) for an image with a name that's similar to the `id`.

To be on the safe side I also set the `alt` attribute. Theoretically it's unnecessary, since images will shown only when the browser supports images, but better safe than sorry.

```
function replaceThem(x)
{
        var replace = document.createElement('img');
        for (var i=0;i<x.length;i++)
        {
                if (x[i].id)
                {
                        var y = replace.cloneNode(true);
                        y.src = 'pix/' + x[i].id + '.gif';
                        y.alt = x[i].firstChild.nodeValue;
                        x[i].replaceChild(y,x[i].firstChild);
                }
        }
}
```

Home Sitemap

Three column stretching

This script is not yet ready for prime time. It works fine on this page, except in Explorer 5.0 Windows and Safari 1.0, but on more complicated pages it may misfire in a number of subtle ways which I haven't investigated yet.

The tiny script presented on this page begins to solve a common CSS problem: stretching three floated columns to the height of the highest column. The script isn't yet ready for real use, but it nicely illustrates the principle of **minimal CSS enhancement**.

CSS has a few very annoying blanks that may frustrate your attempts to create a nicely designed pure XHTML/CSS page. I'm thinking of the lack of vertical alignment, or the stretching up of several floated columns to the height of the highest one.

In my opinion JavaScript can lend a helping hand. However, we should avoid the interminable and useless DHTML libraries that scourged web development in earlier, less enlightened eras. Instead, we should establish the principle of **minimal CSS enhancement**.

The purpose of such a script is to help smooth out rough edges on CSS by applying the *minimal* necessary change to achieve an effect. Ideally, this means tweaking only one single style value, as this example script does, and leaving the rest to normal, simple CSS.

Once we've written such a simple script, accessibility issues become manageable. If the script doesn't work the CSS problem will not be solved, but since these solutions usually fall in the nice-to-have category and are in no way necessary for the basic functioning of the page, I don't see any fundamental reason not to use JavaScript to help CSS along.

For a more fundamental and extensive discussion, see Bobby van der Sluis's Presentational JavaScript article.

Example

This example doesn't work under all circumstances, and therefore it is **not ready** for prime time. Please read **all browser notes** below on this page before mailing me about it.

This is the left column

This is a classic example of a three column layout. The left and right columns have only a little content (in a real website they'd have more than just one line of text, but still less than the main column).

Ideally, all these three columns should have the same height. Of themselves, the left and right columns have "as much height as they need" (ie. `height: auto`) for reasons I'll explain later on.

Nonetheless graphic designers are less than thrilled by this lack of stretching, and rightly so. Pure CSS just doesn't give us the possibility to solve this problem. Do we embrace the purity argument and leave the CSS as it is, or do we use JavaScript to solve the problem?

This is the right column

The styles

The three column layout above is very simple:

```
<div id="testmain">
        <div class="left">
                This is the left column
        </div>
        <div class="mid">

                [text]

        </div>
        <div class="right">
                This is the right column
        </div>
        <div class="clearer"> </div>
</div>
```

The HTML is styled as follows:

```
.left {
        float: left;
        background-color: #732264;
        width: 15%;
        color: #ffffff;
        height: 100%;
}
```

```
.mid {
        float: left;
        width: 50%;
        height: 100%;
        margin: 0 15px;
}

.right {
        float: right;
        background-color: #B0BDEC;
        width: 15%;
        height: 100%;
}

.clearer {
        clear: both;
        font-size: 1px;
}
```

As you see I floated the three columns and gave each of them a width. I also gave each of them a `height: 100%`, and that's the key to the effect.

`height: 100%` means: make the height 100% of that of the parent element. In this case the parent element is `<div id="testmain">`, and that's where the problems start. Since I do not know in advance how much text the middle column will contain, nor how wide the user's browser window will be, I cannot give this parent element a `height` (well, I could, but it'd be ugly in 99.9% of the cases).

Therefore the parent element has `height: auto`, ie. "as much as you need". The problem is that the CSS specification says that any block with a percentual height that is contained by a block that has `height: auto` should also get `height: auto`. In other words, the `height: 100%` does not apply to naturally stretched blocks.

JavaScript to the rescue

Superficially this is a very serious problem. To enable the `height: 100%` we'd have to give the container block a specific `height`, but how do we know what value it should have, when we don't know how much text the middle column contains and how wide the user's window is?

The answer, of course, is reading out the **real**, rendered height by JavaScript and setting the `style.height` to this value:

```
function threeColumn() {
```

```
        var elm = document.getElementById('testmain');
        elm.style.height = 'auto';
        var x = elm.offsetHeight;
        elm.style.height = x + "px";
}
```

We read out the `offsetHeight` of the containing block, which contains the actual height the block has in the rendered page. We then add `"px"` to this value and paste it in `style.height`.

Now, all of a sudden the container block has a specified `height`, and therefore the `height: 100%` of the three columns kicks in. The three columns stretch themselves to completely fill their container and the effect has been established.

As to the `elm.style.height = 'auto';`, this line is necessary if the function is called more than once. If we have to recalculate the height of the container, we should *first* reset it to `auto`, to allow it to stretch naturally, and read out the new `offsetHeight` only afterwards.

This script solves an annoying problem by *tweaking one single style value* and leaving the rest to regular CSS definitions. Obviously, if JavaScript is disabled the CSS value is not tweaked, but in that case we're not worse off than we were before. The page is not inaccessible if the three columns don't have equal height, it's just less beautiful.

Browser incompatibilities

Unfortunately the tiny script above is only a first approximation of a solution. Cross-browser reality is more obnoxious, to say the least.

Most of these problems can be solved by fundamental research into `offsetHeight` and its changing, but at the moment I don't have time for it.

Clearer element

First of all we need a clearer element. As you see the last element inside the container is

```
<div class="clearer"> </div>
.clearer {
        clear: both;
        font-size: 1px;
}
```

All browsers except for Explorer require it. The reason is that without this element the container div contains *only* floated elements, which means that its normal height defaults back to 0. We don't want that, we want the container to stretch up as much as necessary so we can read out its actual height.

The clearer solves this problem for us. It is the last element in the container, it is not floated, hence it stretches up the container so that it actually contains the floater, and thus the three columns above it.

As to the single ` `: Mozilla requires it (don't ask me why). If it's not there the trick doesn't work. I set the clearer's font size to 1px to hide the ` ` as much as possible.

Resizing

If you resize this page you'll see that the column height doesn't change. The obvious solution to this problem is

```
window.onresize = threeColumn;
```

However, this solution turns out to work only in Mozilla and Explorer Mac. Explorer Windows and Opera make the container larger and larger and larger with each resize, and I'm not yet sure why.

Resizing and Explorer Windows

You'll notice that when the page becomes narrower, and the container block higher, Explorer Windows nicely changes the height of the columns, too. This does not happen when you make the page wider and the container block less high.

The cause is Explorer's peculiar interpretation of `overflow: visible`. If the content is too large for a block to contain, Explorer stretches up the block to contain all content, even if the block has a set `height`. Therefore it correctly handles any *increase* (but not decrease) in container height without further JavaScript aid.

Box model

Browser differences in box model will certainly cause serious problems. This example entirely avoids the issue by not applying any paddings or borders, but for real-world applications these problems must be solved.

Opera: not fast enough

If you call the script more than once, you'll notice that Opera doesn't behave right. As

far as I can see this is because it doesn't take the time to calculate the effect of resetting the height to `auto` before calculating the new `offsetHeight`. Therefore the offsetHeight remains what it was, and the columns do not react properly.

The same issue may bug Explorer on Windows if we call the script `onresize`, but I haven't yet investigated this.

Opera: extra offset

You may note that in Opera the container element becomes slightly larger each time you run the script, even when you don't resize the browser window. I'm not yet completely sure of the cause, but in tightly designed pages this may become a problem.

Explorer Windows: large image problem?

When I tried this script in a page where one column contained a large image, Explorer added the height of this image to the height of the container element, and the columns became much too high. I don't yet know why, but it means that this technique is not yet usable in all pages.

Explorer 5.0 Windows: float problem?

The default styles of my example are very weird in Explorer 5.0 Windows. I assume this is because of a problem in the handling of `float`, but I haven't yet investigated it. In any case the example doesn't work in this browser.

Home Sitemap

Usable forms

It crashes Explorer 5 Mac when you use the script too much (a few changes is OK).

Opera sends all form fields to the server, also those in the waiting room.
I have heard, but not tested, that Safari 1.1.1 does the same.

In 'Quirks mode' Opera cannot handle nested select boxes, though in 'Strict mode' it can.

Explorer 6 Windows hides this floating div as soon as you add or remove form fields.

On this page I give the download link and some extra information on the Usable Forms script. This script is meant to show and hide form fields based on user actions. I explain this script in great detail in my Digital Web Magazine article Forms, usability and the W3C DOM.

Download

Download the script. Current version: 1.0 . For further instructions see my article.

Safari update: The 1.0 release supports this script also in Strict mode.

Example

The script adds and removes form fields based on user actions. Try the form below.

Name

Address

Country ☐ Other than The Netherlands

If not the Netherlands: Which other country?

Marital status

- ○ Single
- ○ Married
- ○ Divorced

If Married: Marriage contract? ☐ Yes

If Divorced: Date of divorce

If Divorced: Type of divorce [Messy]

If Very Messy: Gory details

Why do you fill in this form? [Select]

If Just a Feeling: Please describe this feeling in excessive detail.

Submit form

Overview

The script works as follows:

1. When initializing the form, the script searches for all TR's with a `relation` attribute.
2. Every one of these TR's is placed in the "waiting room" outside the form and thus becomes invisible to the user. The script inserts a hidden marker TR with the same `id` value as the `relation` value of the removed TR. This marker is for returning the TR's to their correct places in the form later on.
3. Then the script goes through all form fields. When it encounters a form field that is checked or selected and has a `show` attribute, the appropriate TR's are shown.
4. Finally, the script registers an overall `onclick` event handler to the entire document.

Now the script waits for user input.

1. If the user clicks anywhere on the page, the script checks if the target of this click event is a form field with a `show` attribute. If it isn't, nothing happens.
2. If the user clicks on a form field with a `show` attribute, the script looks for TR's having the same value for their `relation` attribute. It gathers them and puts them back in the form, using the marker to determine the exact spot.
3. If the form field is part of a group (radios or options) the script goes through all other fields in that group and hides the TR's related to them.
4. If the script finds a value `'none'` for the `show` attribute, it doesn't show any new TR's. It does hide the TR's related to the other form fields in the group, though.

Critique

A few WDF-DOM members gave some good feedback. There were two points of critique:

1. It's better to move hidden TR's to a JavaScript array instead of putting them somewhere else in the HTML document. Having thought about it, I agree. The next version of the script will use a JS array for a waiting room, not an HTML construct. I'll need to do a major rewrite of the script to change this, though. Don't know when I'll have the time.
2. Server side handling of the form becomes more difficult because the server side script cannot be certain which data it'll receive.
   Is this because of my script or because of assumptions server side programmers make? I'm not sure. There is no fundamental reason why a form handling script would not be able to handle my script. On the other hand, there certainly are some practical difficulties.
   One remark set me thinking: how about letting the server side script parse the actual HTML page to see what it can expect when the form is submitted? That might be a solution.

Home Sitemap

Extending forms

Explorer 6 Windows hase serious trouble with radio buttons. Despite different names they see all generated radio buttons as one array, so the user can only select one radio button in all the copies.

The script doesn't work in Explorer 5 on Mac because this browser cannot properly generate form fields: it makes them all text fields. Besides it doesn't send the generated form fields to the server.

On this page I treat a very simple W3C DOM script. It serves to explain why I think the W3C DOM will allow us to see interaction design in a radically new way.

The idea

Suppose you have an online CD ranking tool. You want your users to review as many CD's as they like. However, how do you know how many CD's an average user wants to review? How many form fields should you add to the page?

Before the W3C DOM this was quite a problem. Suppose you add form fields for 7 CD's. Some users will review only one CD and don't need the rest of the form (it might even frighten them). Other users might want to add their entire CD collection of hundreds of titles to your database and have to submit the form dozens of times. This is quite annoying.

Only using the W3C DOM you can allow your users to generate as many form fields as they need. This effect is impossible to mimic with any previous JavaScript technique.

This is a very simple example of the new way of interaction that the W3C DOM allows. For further information about the general idea, see the What shall we do with the W3C DOM? article on Evolt.

Example

Which CD's did you listen to recently?

Give me more fields! Send form

When you hit 'Send form' the form is sent to a script that lists the parameters it has received. This is to check whether the generated fields are really sent to the server. Unfortunately it turns out that Explorer Mac and Safari don't send any fields to the server.

Problems in Explorer

Unfortunately there are two serious problems in Explorer Windows:

First of all it sees all generated radio buttons as belonging to one single array, even if they have different `name`s. Thus the user can select only one radio button in all generated fields. Basically this means that you *cannot use radio buttons at all* in generated forms.

A reader said that generating radio buttons through `innerHTML` works fine. If you must use radio buttons, you might try this approach.

Secondly the generated form fields are unreachable by a traditional `document.forms` call: Explorer simply doesn't enter them in the arrays. This can be worked around by giving the form field an ID and then using `getElementById()`.

Explanation

The HTML of the form is:

```
<div id="readroot" style="display: none">
        <p class="hr"> </p>

        <input type="button" value="Remove review" style="font-size: 10px"
                onClick="
                        this.parentNode.parentNode.removeChild(this.parentNode);
                "><br><br>

        <input name="cd_1" value="title">

        <select name="rankingsel_1">
                <option>Rating</option>
                <option value="excellent">Excellent</option>
                <option value="good">Good</option>
                <option value="ok">OK</option>
                <option value="poor">Poor</option>
                <option value="bad">Bad</option>
        </select><br><br>

        <textarea name="review_1">Short review</textarea>

        <br>Radio buttons included to test them in Explorer:<br>
        <input type="radio" name="something" value="test1">Test 1<br>
        <input type="radio" name="something" value="test2">Test 2

</div>

<form>
<span id="writeroot"></span>

<input type="button" value="Give me more fields!" onClick="moreFields()">
<input type="button" value="Send form" onClick="alert('Fake submit')">

</form>
```

The actual form fields are in a DIV with id readroot and `display: none`. This DIV is a template that should not be changed by the user. When the user wants more fields we clone the DIV and append this clone to the form. We do this once onLoad, so that the user sees one set of form fields when entering the page.

The DIV is outside the actual FORM so that the template fields aren't sent to the server when the form is submitted.

The span with id writeroot serves as a marker. The new sets of form fields should be inserted just before it.

Adding form fields

This script adds sets of form fields when necessary:

```
var counter = 0;

function moreFields()
{
        counter++;
        var newFields = document.getElementById('readroot').cloneNode(true);
        newFields.id = '';
        newFields.style.display = 'block';
        var newField = newFields.childNodes;
        for (var i=0;i<newField.length;i++)
        {
                var theName = newField[i].name
                if (theName)
                        newField[i].name = theName + counter;
        }
        var insertHere = document.getElementById('writeroot');
        insertHere.parentNode.insertBefore(newFields,insertHere);
}

window.onload = moreFields;
```

First of all we need a `counter`, because all sets of form fields should get unique names. We do this by appending `counter` to the names in the template. Initialize `counter`:

```
var counter = 0;
```

Then for the actual function. Start by increasing `counter` by 1.

```
function moreFields()
{
        counter++;
```

Then we clone our template, remove its `id` and set its `display` to `block`. The id 'readroot' should remain unique in the document, and the clone of the template should be visible to the user.

```
        var newFields = document.getElementById('readroot').cloneNode(true);
        newFields.id = '';
        newFields.style.display = 'block';
```

We go through the child nodes of the clone

```
        var newField = newFields.childNodes;
        for (var i=0;i<newField.length;i++)
        {
```

Whenever a child node has a `name` we append `counter` to it. Thus the names of all form fields remain unique.

```
                var theName = newField[i].name
                if (theName)
```

```
                        newField[i].name = theName + counter;
        }
```

Now the clone is ready to be inserted into the document. We insert it just before the span with id="writeroot".

```
        var insertHere = document.getElementById('writeroot');
        insertHere.parentNode.insertBefore(newFields,insertHere);
}
```

Finally we execute this function once onLoad so that the user will initially see one set of form fields.

```
window.onload = moreFields;
```

Removing form fields

Each clone of the template contains a 'Remove review' button:

```
<input type="button" value="Remove review" style="font-size: 10px"
        onClick="
                this.parentNode.parentNode.removeChild(this.parentNode);
        ">
```

Clicking on it causes the button to remove its parent node (the DIV) from its own parent node (the FORM). Thus one set of form fields disappears entirely, never to return.

Home Sitemap

Form error messages

This page details a way of showing form validation error messages that is far superior to the silly alerts most forms use.

In my opinion alerts should only be used if the browser doesn't support a better way of displaying form error messages. Instead, the W3C DOM allows us to write error messages next to the form field they apply to. This is clearly the superior method, so we only use alerts if the browser doesn't support advanced scripting.

Example

Try this example form. Every field is required. In addition, I check if the "email" field contains a "@". If it doesn't, the value is not a valid email address and an error message is shown.

name

address

city

e-mail

The script

```
var W3CDOM = (document.getElementsByTagName && document.createElement);

window.onload = function () {
	document.forms[0].onsubmit = function () {
		return validate()
	}
}

function validate()
{
	validForm = true;
	errorstring = '';
	var x = document.forms[0].elements;
	for (var i=0;i<x.length;i++)
	{
		if (!x[i].value)
			writeError(x[i],'This field is required');
	}
	if (x['email'].value.indexOf('@') == -1)
		writeError(x['email'],'This is not a valid email address');
	if (!W3CDOM)
		alert(errorstring);
	if (firstError)
		firstError.focus();
	return validForm;
}

function writeError(obj,message)
{
	if (obj.hasError) return;
	if (W3CDOM)
	{
		obj.className += ' error';
		obj.onchange = removeError;
		var sp = document.createElement('span');
		sp.className = 'error';
```

```
                sp.appendChild(document.createTextNode(message));
                obj.parentNode.appendChild(sp);
                obj.hasError = sp;
        }
        else
        {
                errorstring += obj.name + ': ' + message + '\n';
                obj.hasError = true;
        }
        if (validForm)
                firstError = obj;
        validForm = false;
}

function removeError()
{
        this.className = this.className.substring(0,this.className.lastIndexOf(' '));
        this.parentNode.removeChild(this.hasError);
        this.hasError = null;
        this.onchange = null;
}
```

Explanation

First we check W3C DOM support. The example script on this page works (somewhat) in Explorer on Mac, but don't be surprised if this browser craps out in a real web page. Its W3C DOM engine is simply not good enough to support this script in all circumstances.

Then we create an `onsubmit` event handler for the form that calls the function `validate()` (see also the Introduction to forms).

```
var W3CDOM = (document.getElementsByTagName && document.createElement);

window.onload = function () {
        document.forms[0].onsubmit = function ()        {
                return validate()
        }
}
```

validate()

We assume that the form is valid (`validForm = true`) and we create a string `errorstring`, which will eventually contain all error messages. This string is only for non-W3C DOM browsers.

```
function validate()
{
        validForm = true;
        errorstring = '';
```

The core of the `validate()` function works pretty much as always (see the example form for an example). Check the form fields for any sort of error you wish. When you find an error, call the `writeError()` function and hand it the faulty form field and an error message.

```
        var x = document.forms[0].elements;
        for (var i=0;i<x.length;i++)
        {
                if (!x[i].value)
                        writeError(x[i],'This field is required');
```

```
        }
        if (x['email'].value.indexOf('@') == -1)
                writeError(x['email'],'This is not a valid email address');
```

If the browser does *not* support the W3C DOM, generate an alert with `errorstring`. You might want to rewrite this bit to show an alert for each error message.

```
        if (!W3CDOM)
                alert(errorstring);
```

As a service to the user, put the focus on the first faulty field you found.

```
        if (firstError)
                firstError.focus();
```

Finally, return `validForm`. If still `true` (no errors found) the form is submitted, if `false` the form submission is halted.

```
        return validForm;
}
```

writeError()

The `writeError()` function tries to write the error message next to the form field. If it fails, because the browser doesn't support the W3C DOM enough, it appends the error message to `errorstring`.

The function is handed a form field object and an error message.

```
function writeError(obj,message)
{
```

First we check if this form field already has an error message. If it does, return to the main `validate()` function. I don't want to show two or more messages for the same field.

```
        if (obj.hasError) return;
```

Check if the browser supports the W3C DOM.

```
        if (W3CDOM)
        {
```

If it does, we can start the magic. First of all add `error` to the `className` of the form field. Specify the styles of faulty form fields in CSS.

```
                obj.className += ' error';
```

Next, set an `onchange` event handler for the `removeError()` function. See below.

```
                obj.onchange = removeError;
```

Create a `<span>` tag to hold the error message and give it a `class="error"`. Again, specify the presentation of the error message in CSS.

```
                var sp = document.createElement('span');
                sp.className = 'error';
```

Append a text node with the error message to the `<span>`.

```
                sp.appendChild(document.createTextNode(message));
```

Append the entire `<span>` to the parent node of the form field (in my case, every label/field pair is contained in a `<p>`).

```
                obj.parentNode.appendChild(sp);
```

Finally, set an `hasError` property for the form field. This property serves both to indicate that this form field has an associated error message, and to point to this error message for easy future removal.

```
                obj.hasError = sp;
        }
```

For JavaScriptually challenged browsers we append the name of the form field and the error message to `errorstring`. This string is alerted at the end of the validation (see above). Also set `hasError`. It still indicates that the form field already has an associated error message.

```
        else
        {
                errorstring += obj.name + ': ' + message + '\n';
                obj.hasError = true;
        }
```

If `validForm` is still `true` (if this is the first error message we found), set the `firstError` variable to point to this form field. `validate()` uses this information to put the focus on the first faulty form field.

```
        if (validForm)
                firstError = obj;
```

Set `validForm` to `false`: form is not valid.

```
        validForm = false;
}
```

removeError()

Every faulty form field has an `onchange` event handler that points to this function. If the user changes a faulty form field, I politely assume he has corrected the error. Therefore the error message should disappear.

First remove the `error` from the form field's class name. This removes the special error styles.

```
function removeError()
{
        this.className = this.className.substring(0,this.className.lastIndexOf(' '));
```

Secondly, remove the error message. The `hasError` property points to the `<span>` containing the message, so we remove it from the form field's `parentNode`.

```
        this.parentNode.removeChild(this.hasError);
```

Finally a bit of house cleaning. Set the `hasError` property to `null` (form field has no more associated error messages) and remove the `onchange` event handler.

```
        this.hasError = null;
        this.onchange = null;
}
```

Home Sitemap

Styling an input type="file"

Credits wholly go to Michael McGrady, who invented this technique.

A browser must support opacity to support this technique. Therefore it doesn't work in Explorer 5.0 on Windows, Explorer 5 on Mac and Opera.

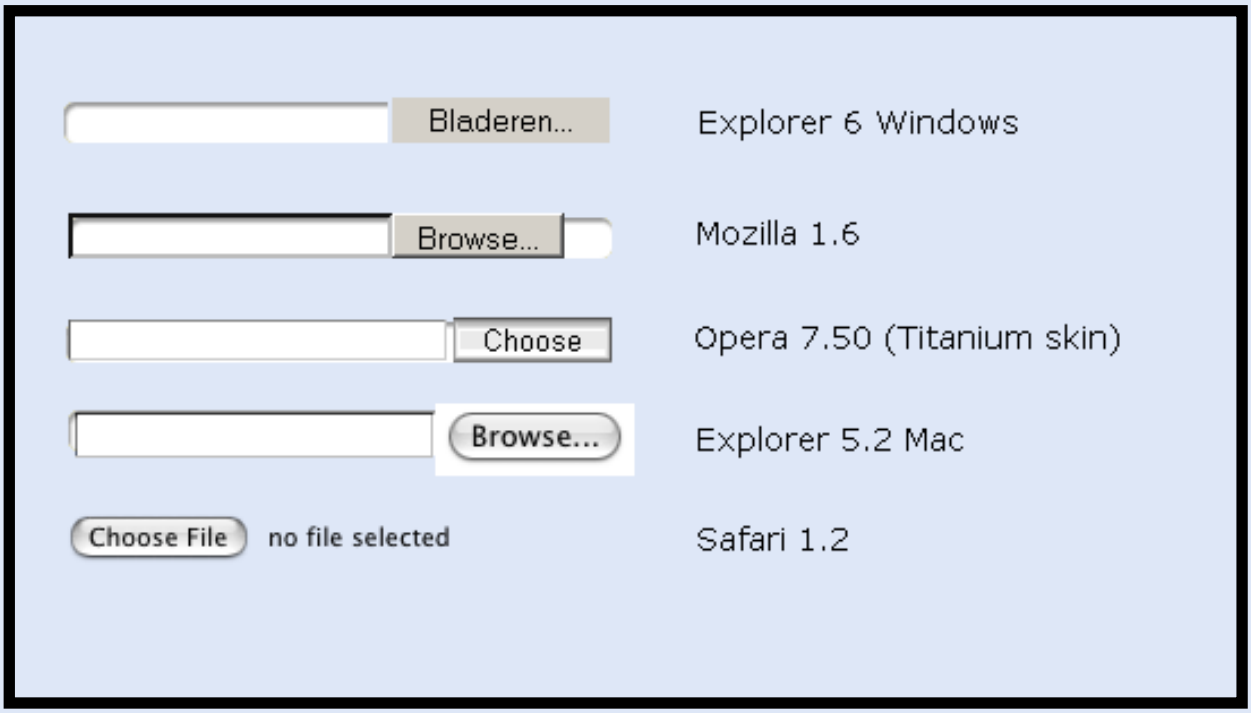
Of all form fields, the file upload field is by far the worst when it comes to styling. Explorer Windows offers some (but not many) style possibilities, Mozilla slightly less, and the other browsers none at all. The "Browse" button, especially, is completely inaccessible to CSS manipulation.

The problem

For a site I created I needed input fields like this one:

The designer wanted the same styles, plus a "Select" image, to apply to all file upload fields. When I applied the rules of normal input fields to file upload fields, though, it didn't really work. There were wild differences between the browsers, and styling the default button is totally impossible.

Ponder the differences.



This is hardly what anyone would call a nicely designed form field. Until recently, though, it was the best we could do.

> Also note Safari's fundamentally different approach. The Safari team has probably decided on this approach to disallow the manual entering of a file name and this avoid exploits like this one. The drawback is that the user can't decide not to upload a file after having selected one.

The solution

Fortunately, reader Michael McGrady invented a very neat trick that allows us to (more or less) style file upload fields. The credits for the solution presented on this page are wholly his, I only added the `position: relative`, a few notes and tests, and ported it entirely to JavaScript.

Without the technique your browser reacts like this:

Using McGrady's technique, I was able to produce this file upload field:

Now that looks much better, doesn't it? (Provided your browser supports `opacity`)

McGrady's technique is elegant in its simplicity:

1. Take a normal `input type="file"` and put it in an element with `position: relative`.
2. To this same parent element, add a normal `input` and an image, which have the correct styles. Position these elements absolutely, so that they occupy the same place as the `input type="file"`.
3. Set the `z-index` of the `input type="file"` to `2` so that it lies *on top of* the styled input/image.
4. Finally, set the `opacity` of the `input type="file"` to `0`. The `input type="file"` now becomes effectively invisible, and the styles input/image shines through, *but you can still click on the "Browse" button*. If the button is positioned on top of the image, the user appears to click on the image and gets the normal file selection window.
   (Note that you can't use `visibility: hidden`, because a truly invisible element is unclickable, too, and we need the `input type="file"` to remain clickable)

Until here the effect can be achieved through pure CSS. However, one feature is still lacking.

5. When the user has selected a file, the visible, fake input field should show the correct path to this file, as a normal `input type="file"` would. It's simply a matter of copying the new value of the `input type="file"` to the fake input field, but we need JavaScript to do this.

Therefore this technique will not wholly work without JavaScript. For reasons I'll explain later, I decided to port the entire idea to JavaScript. If you're willing to do without the visible file name you can use the pure CSS solution. I'm not sure if this would be a good idea, though.

The HTML/CSS structure

I've decided on the following HTML/CSS approach:

```
div.fileinputs {
        position: relative;
}
```

```
div.fakefile {
        position: absolute;
        top: 0px;
        left: 0px;
        z-index: 1;
}

input.file {
        position: relative;
        text-align: right;
        -moz-opacity:0 ;
        filter:alpha(opacity: 0);
        opacity: 0;
        z-index: 2;
}

<div class="fileinputs">
        <input type="file" class="file">
        <div class="fakefile">
                <input>
                <img src="search.gif">
        </div>
</div>
```

The surrounding `<div class="fileinputs">` is positioned relatively so that we can create an absolutely positioned layer inside it: the fake file input.

The `<div class="fakefile">`, containing the fake input and the button, is positioned absolutely and has a `z-index` of 1, so that it appears underneath the real file input.

The real file input field also gets `position: relative` to be able to assign it a `z-index`. After all, the real field should be on top of the fake field. Then we set the `opacity` of the real file input field to 0, making it effectively invisible.

Also note the `text-align: right`: since Mozilla refuses a `width` declaration for the real file field, we should make sure that the "Browse" button is at the right edge of the `div`. The fake "Search" button is also positioned at the right edge, and it should be underneath the real button.

You'll need some subtle CSS to set all widths, heights, borders and so on, but I didn't include it in this code example. View the source of this page to study my approach in this particular case; your pages will be different, though, so you'll have to change these values.

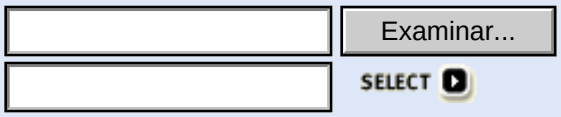
Why JavaScript?

Nonetheless, I decided to go for a strict JavaScript solution. My first reason is that we need JavaScript anyway to copy the file path to the fake field.

Secondly, a JavaScript solution would avoid meaningless extra HTML: the `<div class="fakefile">`. It'd keep my code cleaner.

Finally, older browsers can't really handle the CSS, down to the point that the file input becomes inaccessible in Netscape and Explorer 4. As to users of no-CSS browsers, they'd see two input fields, and wouldn't understand what the second one was for.

Below is a pure CSS solution:

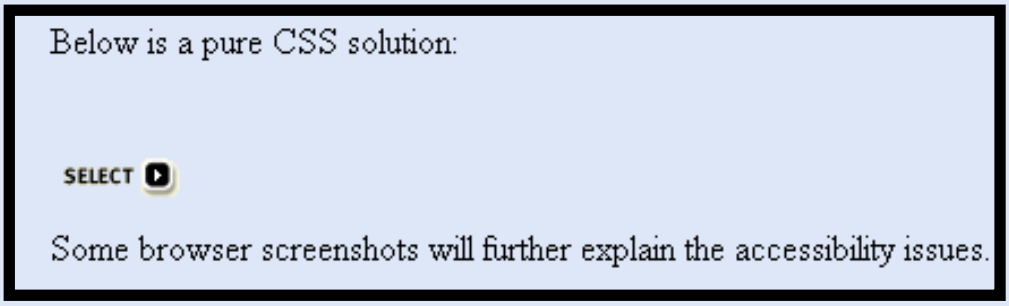


Some browser screenshots will further explain the accessibility issues.

Problems - Netscape 4

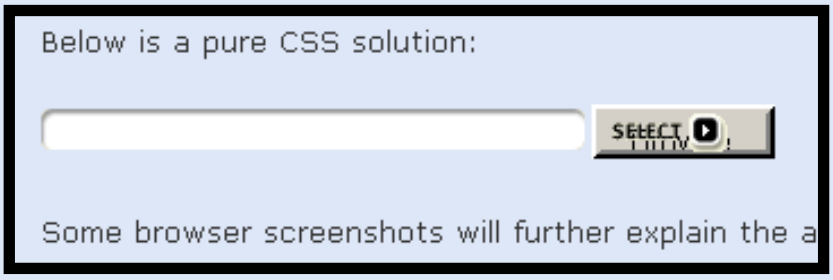
First Netscape 4. As you see, the user sees only the button. This may be because the form is spread across two layers by my use of `position: absolute`, and Netscape 4 can't handle that.

Worse: the user can't click on the button. Maybe an extensive week long study would reveal a partial solution to this problem, but frankly I can't be bothered. Nonetheless, the field must be accessible to Netscape 4 users.



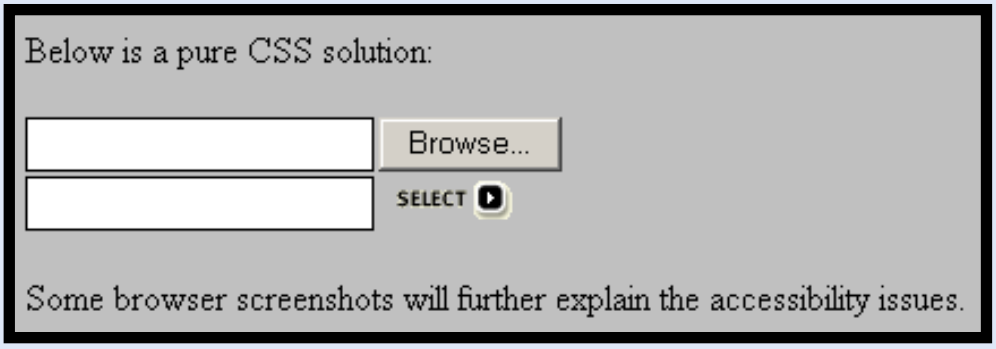
Problems - Explorer 4

Explorer 4: an odd shadow of the original "Browse" button, and it's not clickable, either. The solution is inaccessible in Explorer 4.



Problems - Netscape 3

Finally, users of Netscape 3, or any other non-CSS browser. The field is still accessible, but users will quite likely be confused by the extra input field.



Solution - JavaScript

The solution to all this nastiness is simple: generate the fake input and button through JavaScript. Now, the worst that can happen is that the script doesn't work, in which case the user only sees the real `input type="file"`. Less beautiful, certainly, but still accessible.

So the hard coded HTML is reduced to:

```
<div class="fileinputs">
        <input type="file" class="file">
</div>
```

We'll add the rest of the elements through JavaScript.

The script

So I wrote the following script:

```
var W3CDOM = (document.createElement && document.getElementsByTagName);

function initFileUploads()
{
        if (!W3CDOM) return;
        var fakeFileUpload = document.createElement('div');
        fakeFileUpload.className = 'fakefile';
        fakeFileUpload.appendChild(document.createElement('input'));
        var image = document.createElement('img');
        image.src='pix/button_select.gif';
        fakeFileUpload.appendChild(image);
        var x = document.getElementsByTagName('input');
        for (var i=0;i<x.length;i++)
        {
                if (x[i].type != 'file') continue;
                if (x[i].parentNode.className != 'fileinputs') continue;
                x[i].className = 'file hidden';
                var clone = fakeFileUpload.cloneNode(true);
                x[i].parentNode.appendChild(clone);
                x[i].relatedElement = clone.getElementsByTagName('input')[0];
                x[i].onchange = x[i].onmouseout = function () {
                        this.relatedElement.value = this.value;
                }
        }
}
```

Explanation

If the browser doesn't support the W3C DOM, don't do anything.

```
var W3CDOM = (document.createElement && document.getElementsByTagName);

function initFileUploads()
{
```

```
	if (!W3CDOM) return;
```

Create the and its content. We'll clone it as often as necessary.

```
	var fakeFileUpload = document.createElement('div');
	fakeFileUpload.className = 'fakefile';
	fakeFileUpload.appendChild(document.createElement('input'));
	var image = document.createElement('img');
	image.src='pix/button_select.gif';
	fakeFileUpload.appendChild(image);
```

Then go through all inputs on the page and ignore the ones that aren't `input type="file"`.

```
	var x = document.getElementsByTagName('input');
	for (var i=0;i<x.length;i++)
	{
		if (x[i].type != 'file') continue;
```

Yet another check: if the `input type="file"` does not have a parent element with class `fileinputs`, ignore it.

```
		if (x[i].parentNode.className != 'fileinputs') continue;
```

Now we've found an `input type="file"` that needs tweaking. First we add "hidden" to its class name. I add the advanced styles (opacity and positioning) with this new class, since they might conceivably cause problems in old browsers.

```
		x[i].className = 'file hidden';
```

Clone the fake field and append it to the `input type="file"`'s parent node.

```
		var clone = fakeFileUpload.cloneNode(true);
		x[i].parentNode.appendChild(clone);
```

Now we've succesfully styled the `input type="file"`. We're not yet ready, though, we have to make sure the user sees the path to the file he wants to upload.

First we create a new property for the `input type="file"` that points to the fake input field:

```
		x[i].relatedElement = clone.getElementsByTagName('input')[0];
```

We use this to easily access the fake field as soon as the user changes the real `input type="file"` (ie. selects a file), so that we can copy its value to the fake input field.

A problem here, though: which event do we use? Most natural would seem the `change` event of the file field: if its value changes, the fake input field's value should also change.

Unfortunately Mozilla 1.6 doesn't support the `change` event on file fields (Firefox 0.9.3 does, by the way). For Mozilla's sake I also use the `mouseout` event, which conveniently fires only after the user has selected a file. (This also works in Explorer, but not in Safari)

```
x[i].onchange = x[i].onmouseout = function () {
        this.relatedElement.value = this.value;
}
```

Problems and extensions

Now there's only one problem left: the user can't choose not to upload a file after all.

Suppose the user selects a file, then on second thought decides not to upload it. In a normal `input type="file"` he can simply remove the path, and the file won't be uploaded. In our example, though, this is very difficult. Try it, it can be done, but it's totally counter-intuitive.

So what we'd like to do is allow the user to select and modify the content of the fake file upload and copy all changes to the real file upload.

Allowing selection is somewhat possible. When the user selects any part of the real file upload, we select the entire value of the fake file upload.

```
x[i].onselect = function () {
        this.relatedElement.select();
}
```

Unfortunately, JavaScript security does not allow us to change the value of an `input type="file"`, so we can't let the user manually change the fake input. Therefore I decided to entirely leave out the `onselect` event handler.

A possible solution would be to add a "Clear" button to the fake input, which triggers a script that entirely trashes the `input type="file"` and creates a new one. It's a bit cumbersome, but we might be able to remove the chosen file entirely. I didn't write that part of the script, though, so I'm not sure if it would actually work.

Home Sitemap

Web Forms 2.0 - notes and comments

Written on 10 June 2004.

In his Web Forms 2.0 specification, Ian Hickson extends the HTML 4.01 Forms specification to allow Web authors to easily add validation requirements to form fields. This page contains my notes and comments on the 9 June version of this specification.

Hickson's specification bridges the gap between current practice in form validation and W3C's XForms specification that, though interesting in theory, has no connection at all to everyday reality. When, months ago, I saw an earlier version of Web Forms, I decided that this is the workable, useful specification I've been looking for to help me make the forms I create for my clients more user-friendly, especially in input validation.

Web Forms JavaScript implementation

Most of the Web Forms specification can be implemented in JavaScript right now, and that's exactly what I'm going to do.

More than a year ago I started working on my own form validation script (I don't want to use the word *library* because I dislike JavaScript libraries, but I'm afraid it might be considered as such). This old script is rather crappy, and I've decided to completely rework it. More importantly, today I decided that the script is eventually going to follow the Web Forms specification.

Practical considerations (money and time, mainly) require me to postpone the Web Forms compatible version of my script, though. I'm going to rework the core functions of my old script first, but it has to be backwardly compatible with the forms I already deployed on a few websites. These forms use my own, proprietary set of HTML attributes, so that the next version will not yet be Web Forms compatible. Neither will I publicly release it.

Only after this 2.0 version of my script is finished will I start working on the Web Forms compatible 3.0 version. Don't expect too much too quickly. The 3.0 version will certainly not be released before winter 2004/5.

An early example

This ING test mortgage form created in May 2003, certain aspects of which I discuss in my article Forms, usability, and the W3C DOM, represents my earliest attempt at a modern form validation script. It uses the almost-Web Forms-compatible `required="true"` attribute to indicate required fields, connected to a user-friendly way of presenting error messages.

The form is in Dutch, but you'll get the picture. Press the orange "Ga verder" button at the bottom of the screen and see the error messages pop up. Incidentally, I feel that this way of presenting error messages is superior to the method proposed in Web Forms. I'll discuss this later.

The form contains my Extending Forms script to generate sets of form fields, which could be considered an early implementation of the Web Forms `repeat` functionality (though I don't actually use the tag, which is impossible anyway due to browser considerations).
The repeat is on page 2 of the form. Switch off the error message by checking the checkbox at the very bottom of the page, press "Ga verder" to go to the next page and press the "Meer velden" button to see the repeat in action.

The form also contains a DHTML script that breaks it up into easily manageable sections. Although this sort of user interaction is currently not a part of Web Forms, it's nonetheless useful for web authors to keep the possibilities of DHTML in mind when creating large and complicated forms.

My areas of interest

The rest of this page contains suggestions, notes and comments on the Web Form specification. They reflect my main areas of interest: JavaScript and usability.

I shamelessly interpret Web Forms as a specification for a script (library), and will entirely ignore the possibility of Web Forms being natively supported by any browser (ie. without any scripts being necessary to perform the checks).

Although in the long term native support will be welcome, conforming browsers will not appear in the near future, and when they do appear other browsers will not support the specification. Besides, any conforming browser will likely support JavaScript, too, so that my script will remain functional.

Such a script **requires W3C DOM Level 1 support**, so older browsers (Version 4 and below) will never support it. Besides, even users of modern browsers may switch off JavaScript. Server side form validation will therefore remain mandatory (probably forever).

Such a script **must work in the current generation browsers**, Explorer Windows 5+, Mozilla 1.6 (or thereabout), Opera 7 and Safari 1.2 . I hope to get it working in Explorer 5 on Mac, too, but I'm afraid its W3C DOM implementation is too buggy to allow this.

My second area of interest is usability. Any form extension *must* serve to make the form more usable, so that users are more likely to submit the form, giving the owner of the site more feedback, and probably more transactions. Therefore my take on Web Forms is commercial in scope.

At the moment I ignore the parts of the specification that do not directly relate to a JavaScript implementation or usability, most notably section 5, Form submission.

Custom `type` values

Web Forms heavily depends on as yet unsupported values of the `type` attribute of a form field. This simple test tries to read out such custom values.

Reading out `element.type` from a form element with a custom value works only in Mozilla. Reading out `element.getAttribute('type')` works in Mozilla, Explorer Windows (but not Mac) and Safari.

Therefore custom values of the `type` attribute cannot be used at the moment.

I suggest adding a (temporary?) attribute `WFtype` as an alternative. This attribute doesn't exist at all, and can therefore be safely read by using `element.getAttribute('WFtype')` (sounds odd, but it's true).

> (I'd like to use the more descriptive `valueType`, but Opera refuses to get its value it for obscure reasons. Opera is by far the most annoying browser when it comes to reading out custom attributes.)

Custom tags

Web Forms uses custom tags, most notably the `<repeat>` tag, which should be replaced by a cloned set of form fields.

Try this simple test.

- Explorer on Mac and Safari are unable to recognize the `<repeat>` tag.
- Explorer Windows and Opera recognize the `<repeat>` tag but seem to insert the test paragraph before it. Any text contained by the `<repeat>` tag remains visible. (This is no problem in Web Forms, though, since it uses empty `<repeat>` tags).
- Only Mozilla succeeds perfectly.

Therefore custom tags cannot yet be used.

I suggest replacing `<repeat>` tags by something like `<div WFaction="repeat">` (or whatever, as long as it uses an existing tag with a non-existing attribute).

`output` tags will also have to replaced by existing tags (probably `span`s).

Templates

Section 3: Repeating form controls.

I fully support the idea of templates (in fact, I wrote the first template script back in November 2002, so maybe I even invented them). Although we cannot use the `<repeat>` tags specified by Web Forms 2.0 as yet, we can probably devise another way of adding this functionality.

What I'm missing, though, is flexibility. Right now it seems that the specification only allows for repeating templates as children of the same parent node, and every node may contain only one template/repetition among its children. (Correct?)

I feel that this system is too restrictive. For instance, a template might be a bunch of Address fields, including number, street, postal code, city, country etc. I'd like to be able to use this template throughout the form.

For instance, forms for an an online gift service might use this Address template both in the "Sender" and in the "Recipient" sections of the form. These sections might have different parent nodes (say, divs with DHTML behaviour to make sure the user sees only one of these sections at a time).

Besides, between the Sender and Recipient sections we might put the bit where the gifts are actually selected, using, of course, a Gift template. If the Sender and Recipient sections would have the same parent node, the Gift section would be a child of this node, too, which results in two templates as children of one node.

I feel we need an ID of some kind that specifies which template a `<repeat>` tag copies. Maybe something like

```
<div WFaction="repeat" WFtemplate="Gift">
```

I feel the Add, Move and Delete buttons should use a `<button>` tag instead of `<input>` for backward compatiblity with really old browsers. They'd show a normal text field instead of a button, which would confuse the users.

Date formats

Section 2.1: Extensions to the `input` element, date-related values.

Although in principle it's a good idea to define better ways of handling date and time (anything is better than the current chaos), I recently encountered a practical problem that might make the Web Forms proposal less ideal for everyday applications.

For a while, I used a set of `select`s to generate dates, which could be seen as a primitive precursor of Web Forms's `date` type.

day month year

My main reason for this solution was that it severely restricts the data the user can enter. In fact, the only possible user error in this implementation is the entering of a non-existing day (30 February, for instance). A simple script suffices to catch these errors.

The proposed Web Forms date formats would use different widgets, but the validation script would work roughly in the same way.

The problem

A client asked me to implement a date of birth check in a Web form. The old version of the site already used select boxes to receive this date. When I suggested continuing the triple `select` solution, though, the client immediately and expressly vetoed my proposal. He had excellent reasons for doing so.

The problem, my client explained, was that many users didn't bother to enter their correct date of birth. Their birth year was already known from a previous form, but when asked to provide month and day, too, many users simply ignored the fields, causing the form to be submitted with the default date of birth, 1 January.

This presented severe problems to my client, whose back office needed an exact date of birth. Therefore we decided to use open `input` fields (which caused severe problems in the validation script, but that's another story).

Would Web Forms's proposed date types suffer from the same problem? I'm afraid so, since any widget that offers a default date and expects the user to enter a real date could be ignored by time-pressed users.

We cannot disallow the default date, since in a very few cases the default might be the actual date the user wishes to enter. Using an empty default ("Select a date") might not work, either, since users might quickly select the first option after this default, especially in select boxes or similar widgets. The calendar example might not suffer from this problem, although it might be harder to use than a select box.

At the moment I have no idea how to solve this problem.

Form validation

Section 4.4: Form validation. I strongly disagree with this section, since I feel it violates a number of usability rules.

New events

First of all I'm not sure if we need new events. The old ones will serve fine, and inserting new ones will require a lot of extra coding for no purpose I can see.

I feel that an `invalid` event is only necessary if we allow implementations to validate the form at any point in time, instead of only when the user submits the form. From a usability point of view this is a very bad idea, since validating form fields while the user is still busy filling out the form could become a fatal distraction.

For a while, `blur`-based form validation scripts enjoyed a moderate popularity. As soon as the user leaves a form field, it is checked for errors and the user is immediately notified. Unfortunately this very notification is extremely annoying to the user (especially in combination with alerts). He doesn't want to be disturbed while filling out a complicated form.

I think it's dangerous to change the traditional `submit`-based form validation. When the user submits the form (or moves to a next page, for a form divided into sections), he implicitly states that he thinks he has entered all data correctly and that he's ready to receive feedback. A form should **never** be validated at any other point in time.

Since a form should only be validated at submission time, I feel that the extra events are unnecessary. Worse, they could lure inexperienced web authors into reviving `blur`-based form validation, with all usability disasters this entails.

Presenting error messages

If possible, I disagree even stronger with this paragraph:

> If it was fired during form submission, then the default action is UA-specific, but is expected to consist of focusing the element (possibly firing focus events if appropriate), alerting the user that the entered value is unacceptable in the user's native language along with explanatory text saying why the value is currently invalid, and aborting the form submission. UAs would typically only do this for the first form control found to be invalid; while the event is dispatched to all successful but invalid controls, it is simpler for the user to deal with one error at a time.

I feel that this is exactly the wrong way to go about presenting error messages. Consider the following scenario:

1. User is filling out a form, but isn't paying much attention and skips 10 required fields.
2. `onsubmit` (and not before!) the script finds one required field without a value. It flashes an alert and places the focus on the form field.
3. User sighs, fills out the field, and submits again. Note that he **doesn't know** that there are other required fields,

since he isn't told and he doesn't pay attention to subtle hints like "(required)".
4. `onsubmit` the script finds one required field without a value. It flashes an alert and places the focus on the form field.
5. User gets annoyed, but in a spirit of "what-the-heck" he fills out the required form field. He still doesn't notice that there are 8 more required fields.
6. `onsubmit` the script finds one required field without a value. It flashes an alert and places the focus on the form field.
7. User gets really angry and leaves website, never to return. "Those bloody impossible forms!".

Alerts should not be used (except maybe for one generic warning "Errors have been found; you cannot proceed"). Instead, **all** error messages should **right away** be written next to the form field they apply to. (Optionally, we could place the focus on the first invalid form field.) This way, the user immediately gets **complete** feedback and his level of irritation would remain manageable.

In a heavy duty W3C DOM scripting environment, generating such error messages is a trivial task. See my ING test mortgage form for an example. Press the orange "Ga verder" button at the bottom of the page to see the system in action.

Error message texts

The Web Forms specification is silent on the topic of error message texts. In my opinion it should include a system for defining both general and specific error messages.

Obviously, most error messages would be standard. "This field is required", "This field requires a date", etc. At the moment these error messages seem to be defined within a script, but I feel they should be integrated in the XHTML, especially if Web Forms is ever to become a non-scripted (ie. "native") browser functionality.

Furthermore, a few form fields might need more specific error messages, so there should be a way of overruling general error messages in specific circumstances.

I'm envisioning something like:

```
<form>
        <span class="errorMessage"
                WFfor="required">This form field is required</span>
        <span class="errorMessage"
                WFfor="number">This field requires a number</span>

        <input WFtype="number" precision="float"
                name="height" required="required"
                WFerrorMessage="We need to know your height (in meters) because you might
not fit in all our cabins">
</form>
```

We'd hide the spans by CSS, and read out the error messages as necessary. The "height" field has a special error message, since it's unusual to make this a required field, so the user needs some extra information.

(Eventually the spans would have to be replaced by other tags, but for now we need existing tags).

Groups of form fields - `group` attribute

I'm missing one feature in Web Forms, a feature that I've decided to implement in my script. It should be possible to add a check on a **group** of elements to see if **at least one** of them has a value.

There is a practical need for such controls. I'm currently working on three sets of forms, two of which need a group check:

1. One form serves to order brochures. The possible brochures are named in the form, each preceded by a checkbox. What I need is a simple check to make sure that at least one of these checkboxes is checked. If none is checked (ie. user does not want to receive any brochures), submitting the form is useless.

2. The other form is a contact form. It contains a lot of contact-related text fields (voice phone, fax, mobile phone, email etc. etc.), and the rule is that the user should enter at least one way of contacting him. Again, without the user giving any contact information submitting the form is useless.

I propose something like the following:

```
<input type="text" group="contact" name="voicephone"> Voice phone
<input type="text" group="contact" name="fax"> Fax
<input type="text" group="contact" name="mobile"> Mobile phone
<input type="text" group="contact" name="email"> Email
```

If the user fills out none of the form fields in the "contact" group, an error message is shown and the form is not submitted.

Extending Web Forms towards usability?

It might be useful to extend Web Forms so that it also takes more general usability questions into account. Right now the focus of Web Forms is very much on validation, but the general presentation of forms (especially long and complicated ones) also deserves attention.

I was thinking of the following points:

- **Sections**. Long and complicated forms might be broken up into several sections, only one of which is visible at any time (standard DHTML behaviour). Optionally, we might fire up the validation script when the user moves from one section to another.
- **Hiding form fields**. Forms become much more usable if individual form fields are hidden until needed. Of course this only goes for form fields that are necessary only in a certain context, like "Date of divorce", which only makes sense if the user indicates he's in fact divorced.

Minor stuff

A few minor details.

read-only fields

I feel read-only fields are unnecessary. If they're read-only, they don't need to be a form field. If the form somehow needs the data, make them hidden fields. If the user needs to see the data, too, print out the data as normal text.

Older browsers don't support `read-only`, so the form fields are editable in those browsers, and edits may cause serious back-end problems. Simply leaving out the (visible) form fields solves this problem.

`required` on select-ones

I feel the `required` attribute on select-ones is unnecessary. If a select-one (or a group of radios) is required, I always solve the problem by checking/selecting one of the options. As a result there's always a value.

The code example in section 2.10 is incorrect: one of the options is automatically selected, so it'd always pass a `required` check.

`autocomplete`

I feel the default value of `autocomplete` should be `off`. In general only a section of the form would have to be remembered, and besides there's the question of backward compatibility with existing systems like online banking sites.

In JavaScript, we'd create an `autocomplete` function by cookies, most likely, and I'd like to make sure that the

cookie wouldn't contain too many fields (especially free textareas with wraps and such would be a problem).

Typo

Typo in 2.16 under "For parsing errors in HTML": "divine" should (probably) read "define".

Key to compatibility tables

Yes	Supported completely and correctly
Almost	Supported completely and correctly except for a minor issue
Incomplete	Supported correctly but not completely
Alternative	Supported in an alternative way.
Untestable	Test doesn't work at all, usually because it depends on another method or property that is not supported
Minimal	Just barely supported but unusable in practice.
Incorrect	Returns incorrect object or value and becomes badly usable
Buggy	Does something it should not do
No	Not supported
Crash	Crashes browser

Home Sitemap