



# Modelling the Document Objects

Release: 2010-04-03

Table Of Contents

- [DOM](#)
- [More DOM](#)
- [Scripted images with canvas](#)
- [Selectors API](#)

### **DOM**

In order to access parts of a webpage in a scripting or programming language to be able to manipulate the structure, the scripting or programming language has a model of objects. These objects represent the elements, their attributes, the text values for attribute values and element content. Plus properties for the information of the relationship of the element within the document such as its parent element or next sibling. This model is the Document Object Model or DOM for short.

There are a few main levels of the DOM, each are made up of modules. We are going to look at the basics of DOM 1 Core and HTML, DOM 2 Core, Events and HTML, DOM 3 Core and DOM 5 HTML.

As the most typical scripting environment that uses the DOM is ECMAScript, the open standard that most web browsers use and is the base of Netscape/Mozilla's JavaScript and Microsoft's JScript (used in Internet Explorer); then this is the perspective I will have when providing examples. This article is not exhaustive.

DOM is also used within many programming languages such as C++, Java and Perl.

The core of the model is the `node` object which provides core properties like `nodeName`, `nodeValue`, `parentNode`, `namespaceURI`, `attributes` which contains an array of attribute objects, etc. Also includes these methods `hasChildNodes()`, `appendChild()`, `removeChild()`, `hasAttributes()`. This object's properties and methods are inherited by most other objects in the model. The starting point of the webpage's model is the `document` object. From here you can create elements (`createElement()` and `createElementNS()`), text nodes (`createTextNode()`), set attributes (`setAttribute()` and `setAttributeNS()`) and append the text nodes to the element and the element to other elements (`appendChild()`) to build up new structure. You can identify certain element objects by using the `getElementById()`, `getElementsByTagName()` and `getElementsByTagNameNS()` methods.

For instance we have a `<div id="me"> </div>` element directly within the `body` element and we want to dynamically add a `<p class="highBlue" title="DOM Level 1">Modelling the Document Objects</p>` element within it:

```
// Create the paragraph element
pEL = document.createElement("p");
```

```
// Create the text node
```

## Modelling the Document Objects - Legend Scrolls

```
pText = document.createTextNode("Modelling the Document Objects");

// Appending the text node to the paragraph object
pEL.appendChild(pText);

// Set a couple of attributes
pEL.setAttribute("class", "highBlue");
pEL.setAttribute("title", "DOM Level 1");

// Append the paragraph object to the div object with id 'me'
mediv = document.getElementById("me");
mediv.appendChild(pEL);
```

**Figure 1:** Basic DOM 1 example

But you can't guarantee that an element will always have an `id` attribute. So you can walk the document structure starting from the `body` element and obtain an array (or `nodeList`) of elements with that name in the current level of child elements with the `getElementsByTagName()` method on the element object. Or you can use the `getElementsByTagName()` method directly on the document object which gets a `nodeList` of elements with that name from the entire document.

An `item()` method allows you to pick a single element from that `nodeList` with a number as the parameter. To reference the first element you use the number `0` (zero), second element would be the number `1`, etc. To get the text value of the element you need to get the text node by using the `firstChild` property and then get the text by using the `nodeValue` property.

A note about getting to the `body` element: the `body` object property is available on the `document` object in HTML in `text/html` and XHTML in `application/xhtml+xml` documents in web browsers. But usually is not in XHTML in `text/xml` and `application/xml` documents. So you may need to check if it exists, if not walk the structure from the top.

```
// Get the body of the document
if (document.body) {
    docbody = document.body;
} else {
    docbody = document.getElementsByTagName('body').item(0);
}

// Get the third division block (div element) directly within the body element
divEL = docbody.getElementsByTagName('div').item(2);

// Get the first paragraph from that div element
para = divEL.getElementsByTagName('p').item(0);

// Get the paragraph's text
paraText = para.firstChild.nodeValue;
```

## Modelling the Document Objects - Legend Scrolls

### **Figure 2:** Walking the document structure

Once you have the particular element object such as `divEL` you can get its name by using `divEL.nodeName`; and checking what type of node it is with `divEL.nodeType`. Node types are numbers for instance `1` is an element, `2` is an attribute and `3` is a text node.

DOM 2 adds support for XML Namespaces and so can create namespaced elements and set namespaced attributes (mostly for native XHTML and XML in general):

```
// Save the XHTML namespace for improved readability
xmlns = "http://www.w3.org/1999/xhtml";

// Create the XHTML paragraph element
pELNS = document.createElementNS(xmlns, "p");

// Create the text node
pText = document.createTextNode("Modelling the Document Objects");

// Appending the text node to the XHTML paragraph object
pELNS.appendChild(pText);

// Set a couple of XHTML attributes
pELNS.setAttributeNS(xmlns, "class", "highBlue");
pELNS.setAttributeNS(xmlns, "title", "DOM Level 2");

// Append the XHTML paragraph object to the XHTML div object with id 'me'
mediv = document.getElementById("me");
mediv.appendChild(pELNS);
```

### **Figure 3:** Basic DOM 2 example

Or if prefixed with `xhtml:` then include the prefix when naming the element or attribute when creating. Such as `.createElementNS(xmlns, "xhtml:p")` and `.setAttributeNS(xmlns, "xhtml:class", "highBlue")`. You don't put the prefix in when using `getElementsByNameNS()` or `getAttributeNS()` even if they are prefixed. But usually just the elements would be prefixed unless the attributes are from a different XML language to the current element.

Also DOM 3 Core adds the `textContent` property to node objects so that they can retrieve text from and set text to elements.

```
// Get the body of the document
if (document.body) {
    docbody = document.body;
} else {
    docbody = document.getElementsByTagNameNS(xmlns, 'body').item(0);
}
```

## Modelling the Document Objects - Legend Scrolls

```
// Get the third division block (div element) directly within the body element
divELNS = docbody.getElementsByTagNameNS(xhtmlns, 'div').item(2);

// Get the first paragraph's text from that div element
paraText = divELNS.getElementsByTagNameNS(xhtml,
'p').item(0).firstChild.nodeValue;
// or
paraText = divELNS.getElementsByTagNameNS(xhtml, 'p').item(0).textContent;
```

**Figure 4:** Walking the namespaced document structure

So far Mozilla based, WebKit based, Chromium based, Presto based and KHTML based web browsers such as Firefox, Safari, Chrome, Opera and Konqueror respectfully, all support DOM 3 Core's `textContent` property. Microsoft's Trident based like Internet Explorer does not.

When using the DOM in ECMAScript / JavaScript it is best to test if the objects are supported as not all web browsers or other environments that use scripting may have DOM support:

```
if ((document.getElementById) && (document.createElementNS)) {
    // Do DOM 2 stuff
} else if ((document.getElementById) && (document.createElement)) {
    // Do DOM 1 stuff
} else {
    // Do non DOM stuff
} // end if DOM 2 stuff
```

**Figure 5:** Using the latest possible DOM code

## Events

Instead of attributes such as `onclick`, `onload`, `onkeyup`, an unobtrusive way to get elements to listen for events is by using the `addEventListener()` method from DOM 2 Events.

The first parameter is a string with the name of the event such as `'click'`, `'load'`, `'keyup'`. Also the second parameter is either an anonymous function or the name of the separate function (no parenthesis or brackets) that will be run when that event is received. Referring to a separate function is better especially if more than one event will run the function.

Plus the third and final parameter is whether or not the function will be run when the event is received during the capture phase ( `true` ) or during the bubbling phase ( `false` ).

For instance when you activate a button or link, or an element within a button or link, the event (keydown, keypress, keyup or mousedown, mouseup, etc) travels from the top of the document, down the structure, to the target element that you activated and any of its descendant elements if any. This is

## Modelling the Document Objects - Legend Scrolls

the *capture* phase. Any ancestor elements listening with the third parameter as `true` will trigger their functions before reaching the target element.

Then the event *bubbles* back up the document structure until it hits the top. Any of the target element's descendant elements and ancestor elements that are listening for any of the appropriate events with the third parameter as `false` will run their functions.

You can have an element listening for several events or even some same events but different functions by applying more than one `addEventListener()` method.

To make the element stop listening for the event you use the `removeEventListener()` method with exactly the same parameter values as you did for adding it. This ensures you stop listening for the right event, during bubbling or capture phase and which function is associated.

```
divEL.addEventListener("keyup", doIt, false);
divEL.addEventListener("click", doIt, false);
```

**Figure 6:** Adding some events

Microsoft's Trident 4 and under Web Platform (in Internet Explorer (IE) 8 and under) uses a Microsoft variation. You use `attachEvent()` and `dettachEvent()` methods to add and remove events. The first parameter is similar but with the 'on' prefix in the event name and the second parameter is the same as the main DOM. There is no third parameter as events are only handled within the bubbling phase.

```
if (divEL.addEventListener) {
    divEL.addEventListener("keyup", doIt, false);
    divEL.addEventListener("click", doIt, false);
} else if (divEL.attachEvent) {
    divEL.attachEvent("onkeyup", doIt);
    divEL.attachEvent("onclick", doIt);
}
```

**Figure 7:** Adding events for old IE and other browsers

The function should have at least a parameter to receive the event. This parameter is usually called `evt` or `e` for simplicity but you can call it whatever fits the nature of the script or program.

Then you can access information about the event within the function. Such as the target element node that receives the event is obtained by the `evt.target` property ( or IE 8 and under's version is `evt.srcElement` ) and the type of event (click, keyup, etc) is obtained by the `evt.type` property (even in IE).

For getting which keyboard or keypad key was used, the `evt.keyCode` property provides the numeric representation of that key such as `9` is the tab key, `13` is the enter or return key and `32` is the spacebar key.

If the event was a pointer event such as a mouse event then using `evt.clientX` and `evt.clientY` will give you the x and y coordinates of the mouse click within

## Modelling the Document Objects - Legend Scrolls

the webpage. Also the `evt.button` property will provide the number corresponding to the mouse button used: `0` (zero) for the primary button (left in right-handed mode, right in left-handed mode), `1` is the middle or scroll button and `2` is the secondary button (right in right-handed mode, left in left-handed mode).

```
function eInfo (evt) {
    pReport = document.getElementsByTagName('p').item(4);
    if (evt.target) {
        evtEL = evt.target;
    } else if (evt.srcElement) {
        evtEL = evt.srcElement;
    }

    pReport.firstChild.nodeValue = 'The point of activation was at ' + evt.clientX
    + ' by ' + evt.clientY + '. Key pressed was ' + evt.keyCode + '. Event type was ' +
    evt.type + '. The target element was ' + evtEL + '!';
}
```

**Figure 8:** A function providing event information

## More DOM

Most web browsers have added extra objects, methods and properties to cover features that the existing DOM does not address or to be more efficient than the current way. Some of these are now being made official in DOM 5 HTML.

One of the most popular is the `innerHTML` property. This was originally invented by Microsoft and other browser vendors added support for it as compatibility for IE only websites. But they found it quite useful in its own right and is an unofficial standard.

`innerHTML` is a property that you can retrieve the contents of the element or assign new content (replacing the old).

```
// Capture the div element
divEL = document.getElementById("me");

// Assign content
divEL.innerHTML = "<p class=\"highBlue\" title=\"DOM Level 5
HTML\">Modelling the document objects</p>";

// Get content
currentContents = divEL.innerHTML;

// Forgot to add extra sentence so need to add to current contents
divEL.innerHTML = currentContents + " <p class=\"fancyStyling\">Adding an
```

## Modelling the Document Objects - Legend Scrolls

```
extra sentence</p>”;
```

**Figure 9:** Using innerHTML

Plus more web browsers have support for the `outerHTML` property which will replace not only the contents but the current element object itself such as:

```
// Obtain 3rd paragraph (<p id="info">Accessible versions of a video.</p>)  
collection = document.getElementsByTagName('p');  
theInfo = collection.item(2);
```

```
// Replace with an unordered list  
theInfo.outerHTML = '<ul id="info"><li>Captions,</li><li>Audio  
Description</li><li>Extended Audio Description</li><li>Sign Language  
Video</li></ul>';
```

**Figure 10:** Replacing with outerHTML

## Scripted Images with Canvas

Canvas provides a way to melodramatically create images using the canvas element and its canvas scripting API (application programming interface).

```
<canvas id="interimage1" width="300" height="300">  
  <span>A red box</span>  
</canvas>  
<script type="text/javascript" src="scripts/drawit.js"></script>
```

**Figure 11:** Canvas element

In drawit.js:

```
canv = document.getElementById("interimage1");  
if (canv.getContext) {  
  // Get 2d context for the image  
  image2d = canv.getContext("2d");  
  // Set color to red  
  image2d.fillStyle = "#ff0000";  
  // Draw rectangle  
  image2d.fillRect(100, 100, 50, 50);  
} else {  
  // Non Canvas handling code.  
} // End if Canvas is supported
```

**Figure 12:** Basic Canvas script

To colour the lines rather than the fill you can use the `strokeStyle` property. Both `fillStyle` and `strokeStyle` can take the hex colour codes, `rgb()` colour function from CSS and if the environment like a web browser supports them you can use the `hsl()` colour function from CSS 3 and the `rgba()`, `hsla()` colour functions for opacity colours.

Rectangles can be drawn providing the x and y values for the top left corner

## Modelling the Document Objects - Legend Scrolls

and the width and height values as the parameters of the `fillRect()`, `strokeRect()` and `clearRect()` 2d methods. `fillRect()` draws a fully coloured rectangle, `strokeRect()` draws a rectangle line box and `clearRect()` draws a clearing rectangle.

For more complex images than rectangles you use paths. To start a path you open it using the `beginPath()` 2d method with no parameters. Moving the virtual pen or pencil you use the `moveTo()` 2d method with the x and y values of the location to move to. Lines are drawn using the `lineTo()` 2d method also with x and y values as the parameters.

A quadratic bezier curve can be drawn with `quadraticCurveTo()` method with the x and y values of the curve point and x and y values of the end of the curve. For a cubic bezier curve you provide the x and y values for the first curve point, x and y values for the second curve point and the x and y values of the end of the curve as parameters of the `bezierCurveTo()` method.

If it is not an open path then you need to use `closePath()` method without parameters. Whether or not the path is open or closed you need to specify the `stroke()` and / or `fill()` methods if you want the lines and / or filled colour to actually be visible.

For a more in depth tutorial on canvas: [Mozilla Canvas Tutorial](#) seems to be the prominent one.

## **Selectors API**

This new application programming interface (API) allows you to use the power of Selectors from Cascade StyleSheets (CSS) to obtain a single element or an array of elements. Only two methods are added to the `document` object and the `element` object of the Document Object Model.

The first is `querySelector()` which obtains the first element that the selector matches. `querySelectorAll()` provides an array of all elements that match the selector. So for an equivalent comparison:

```
// Instead of  
meEL = document.getElementById("me");
```

```
// Could use  
meEL = document.querySelector("#me");
```

**Figure 13:** ID Selector

Now to retrieve an array of all paragraphs with a `title` attribute that starts with `'itemDesc'`:

```
pARR = divEL.querySelectorAll('p[title^="itemDesc"]');
```

**Figure 14:** `querySelectorAll()` example

## Modelling the Document Objects - Legend Scrolls

You can then apply your functions or other DOM manipulation on the whole collection of obtained elements.

Mozilla 1.9.1 like Firefox 3.5, Presto 2.2 like Opera 10, Trident 4 Standards Mode like Windows Internet Explorer 8 Standards Mode, WebKit based browsers like Safari and Chromium based like Google Chrome support the Selectors API level 1. But all selectors with this API is limited to the support of all the CSS 2.1 and 3 Selectors that the web browser supports generally for StyleSheets. So Internet Explorer 8 and under has a limited CSS 3 collection of Selectors but at least has a full CSS 2.1 selectors collection.

For more information about Document Object Model (DOM) you can visit <http://www.w3.org/standards/techs/dom>.

## Modelling the Document Objects - Legend Scrolls

This article and others are available online and in other document formats at:

<http://www.legendscrolls.co.uk/webstandards/>

Author: Peter Davison from [Legend Scrolls](#).

Copyright ©2005-2010 Legend Scrolls and Peter Davison.  
All rights reserved.