Web developers are, of course, familiar with JavaScript and the Document Object Model (DOM). While DOM provides a neutral interface for abstracting XML/HTML documents, JavaScript provides an implementation of this interface that lets you interact with web pages. In this article, explore the JavaScript bindings of DOM and learn how to manipulate a web document for peak performance. An example application illustrates DOM methods and properties, and how to attach handlers to DOM events.

Introduction

The Document Object Model (DOM) has been defined in different groups of specifications (DOM Level 1, DOM Level 2, and DOM Level 3) by the World Wide Web Consortium (W3C). The DOM represents an HTML or XML document as a tree composed of a hierarchy of nodes with properties and methods. Using client-side languages such as JavaScript, you can add, modify, delete, and attach events to nodes inside the tree, making it possible to generate interactive, dynamic web pages.

Modifying the DOM with client-side scripting (JavaScript) is called DOM scripting. DOM scripting is used in lieu of the generic term Dynamic HTML (DHTML), which has been used in web development to indicate the construction of interactive web pages through HTML, CSS, and JavaScript.

In this article, explore the most commonly used methods and attributes in the DOM API. A detailed example shows how to traverse the DOM with JavaScript. A more complex model illustrates where events and listeners are taken into consideration. Learn how you can leverage JavaScript libraries to interact with the DOM.

You can download the source code used in this article. Related topics provides links for those who wish to dive deeper into the concepts discussed in the article.

DOM scripting

In DOM terminology, a document is represented as the root of the tree. In JavaScript it is window.document, or simply document (as it is attached to the Window object). This is the starting point for some of the JavaScript implementations. Listing 1 shows an example of an HTML fragment.
**Listing 1. HTML code**

```html
<body>
  <p id="paragraph1">
    <span>This is some text</span>
    <a href="/index.html" title="Click here">Click here</a>
  </p>
</body>
```

From a DOM perspective, in the example above the `p` tag is represented by the DOM `Element` interface. It is the parent of the `span` tag and of the `a` tag. The `span` and `a` tags are siblings.

Suppose you want to get the `href` attribute of the anchor in the code in **Listing 1**. An easy way to access an element in the DOM is to use the `getElementById` method. The following code string shows part of the definition of the document interface containing the `getElementById` signature written in the Interface Definition Language (IDL):

```
Element getElementById (in DOMString elementId)
```

JavaScript implements the `DOMString` interface with the `String` object, so the method accepts the element id as a parameter in the form of a string. In the example fragment, the only element equipped with an `id` attribute is the `p` tag, so it can be retrieved with `var paragraph = document.getElementById("paragraph1");`.

You can obtain the anchor nested into the `p` tag using the `childNodes` attribute. This attribute belongs to the `Node` interface, and returns an object of `NodeList` type. The object is an array-like object in JavaScript. Array-like objects don't have methods, such as `pop()` or `push()`, but they have the `length` property. The object returning from the `childNodes` attribute doesn't make any distinction between node elements (HTML tags), text nodes, or comments node. If you're only seeking node elements, you might consider the `children` attribute. Without considering text and comments nodes, it performs better than `childNodes` for our purposes. In the example, the anchor is the second child of the paragraph, which can be obtained with: `var aElement = paragraph.children[1];`.

Given an element, to obtain the value of the `href` attribute you can adopt the `getAttribute` method by passing the name of the attribute as a parameter (in this case, it's `href`). The part of the IDL definition containing the `getAttribute` method is: `DOMString getAttribute (in DOMString name)`.

In the example, you can implement the above interface like so: `var aHref = aElement.getAttribute("href");` // "index.html".

As in JavaScript, you can chain methods. To get the value of the `href` attribute of the `a` tag in just one line, use: `var aHref = document.getElementById("paragraph1").children[1].getAttribute("href");` // index.html */.

**Dig into DOM scripting: Example application**

This section explores some of the features in DOM scripting. The example Sticky Notes application is an interactive web page that lets the user add "sticky" notes without reloading the page. **Figure 1** shows the page.
Figure 1. Sticky Notes application front end

The HTML code for the page shown in Figure 1 is shown in Listing 2. Within the head tag are the references to CSS and JS files. In the body tag you can see the structure of the notes already in the page: the textarea tag and the anchor that trigger the creation of a new note.

Listing 2. HTML code

```html
<!DOCTYPE html>
<html>
<head>
  <meta charset=utf-8">
  <title>Dom Scripting</title>
  <link rel="stylesheet" href="css/master.css" />
  <script src="js/script.js"></script>
</head>
<body>
  <div class="wrapper">
    <h1>Sticky Notes</h1>
    <div class="links">
      <textarea id="contentArea" cols="10"> </textarea>
      <a href="/random.html" class="add">Click here</a>
      <span>to add a sticky note</span>
    </div>
    <div id="notes">
      <div class="note">
        <p>This is a note</p>
      </div>
    </div>
  </div>
</body>
</html>
```

Let's analyze the JavaScript code contained in the script.js file loaded by the page. You need to trigger the logic of the script once the page is loaded or once the document has been constructed. To do so, one alternative is to bind a function to the onload window attribute, as shown in Listing 3.

Listing 3. onload attribute

```javascript
window.onload = init;
function init() {}
```
The `onload` attribute is associated to the DOM event `load`, which is typical of how an event is bound to a listener function under DOM Level 0 (a "specification" supported by all browsers but not a standard). Conversely, a standard DOM Events Model is defined inside the DOM Level 2 Specifications. In this specification, the `addEventListener` method (from the `EventTarget` interface) is defined to register an event handler on a target element. The following code exposes the signature of this method:

```javascript
object.addEventListener(eventType, eventHandler, useCapture);
```

Where `eventType` is the event to register on the object, `eventHandler` is the function to bind to the specific event. `useCapture` is an optional boolean defining which phase of the event flow the function will be called on (bubbling or capture). The following code uses the `addEventListener` function to bind the `load` method to the `window`:

```javascript
window.addEventListener("load", init, false);
```

Unfortunately, Internet Explorer (IE), prior to version 9, doesn't support the above W3C method and has its own implementation: `object.attachListener(eventType, eventHandler);`. See Related topics for information about IE support for DOM Level 3 events.

`eventType` needs the prefix `on` applied to the event name. Events in IE bubble by default, so the `useCapture` parameter is not present.

Taken from script.js, Listing 4 shows the `addEvent` function, which handles the event binding in all browsers. It is a method of a global object called `SA`. This method works with all the approaches discussed previously.

### Listing 4. addEvent function

```javascript
window.SA = {
    addEvent : function(element, evType, fn, useCapture) {
        if (element.addEventListener) {
            element.addEventListener(evType, fn, useCapture);
            return true;
        } else if (element.attachEvent) {
            var r = element.attachEvent('on' + evType, fn);
            return r;
        } else {
            element['on' + evType] = fn;
        }
    }
}
```

If you use the `addEvent` function, you can bind a function (let’s call it `SA.load`) to the `load` event, as shown in Listing 5.

### Listing 5. Binding the function

```javascript
SA.addEvent(window, "load", SA.load, false);
SA = {
    ...
    load : function() {
        // init block
    }
}
```

The `SA.load` function above is triggered only when all the resources are downloaded, as it’s attached to the `load` event. In a generic scenario, the function attached to the `load` event can take a while before being
Traverse the Document Object Model with JavaScript

executed, especially if there are many images to be downloaded in a page. It's good practice to attach the function initializing the script to the DOMContentLoaded event, which is supported in modern browsers and triggered when the DOM is constructed. The function will be executed before external resources are downloaded, making the page more responsive. Prior to version 9, IE didn't include the DOMContentLoaded event out of the box so a workaround is needed to make it work like the other browsers. In the example, there aren't any images in the page so you can keep the load approach (the performance of the page won't be extensively affected).

You're now ready to associate a function handler to the click event on the target anchor. When the user clicks on the anchor, a specific behavior will be executed. In the example, a new note will be created. The first task is to traverse the DOM to retrieve the anchor we are targeting, as shown in Listing 6.

### Listing 6. Anchor with class name add retrieved

```javascript
load : function() {
  var anchorSelected;

  if (document.getElementsByClassName) {
    anchorSelected = document.getElementsByClassName("add")[0];
  } else {
    var anchors = document.getElementsByTagName("a"),
    alength = anchors.length;
    for (var i = 0; i < alength; i++) {
      var anchor = anchors[i];
      if (anchor.className === "add") {
        anchorSelected = anchor;
      }
    }
  }
}
```

In Listing 6, the document.getElementsByClassName method, as you can probably predict, lets you retrieve the elements with a given class name. This method returns a collection of HTML Elements but, unfortunately, is not fully supported in all browsers, such as IE6 and IE7. For those browsers, different logic needs to be written. You can first get a list of anchors through the document.getElementsByTagName method and loop through that list to get the anchor with a CSS class named add. The GetElementsByTagName method formally returns a NodeList object and, luckily, is fully supported in all the main browsers.

In Listing 6 you see how to store the size of the array of anchors in the alength variable so that in the for-loop you query the DOM just once. Modifying and working on the DOM is an expensive operation, so you should try to minimize the number of times you interact with it.

At this point, once you have retrieved the anchor you're able to bind the click event to the listener function in charge of adding a note, as shown in Listing 7.

### Listing 7. Binding the event

```javascript
load : function() {
  SA.addEvent(anchorSelected, "click", SA.addNote, false);
}
```
Listing 7 shows that the event listener attached to the click event is called `SA.addNote`. This function has several goals:

- Cloning the latest note created
- Injecting the text typed by the user into the note just cloned
- Appending the new note to the list of notes

Listing 8 shows the implementation to achieve the first goal.

### Listing 8. Cloning the latest note created

```javascript
addNote : function(event) {
    var notes = document.getElementById("notes");
    // Clone the node
    var newNode = notes.children[0].cloneNode(true);
}
```

After getting the `div` tag with note ID through the `getElementById` method, you retrieve the first child nested inside the `div` and clone it using the `cloneNode` method. Store the DOM node just cloned in a variable called `newNode`.

Select the paragraph node nested inside `newNode`, invoking the `getElementsByTagName` method on the cloned node. DOM offers an attribute called `textContent` to get the content of the node. Unfortunately, it's not fully supported in all browsers. You need to follow a different approach: from the paragraph, access the `firstChild` attribute and then retrieve the `nodeValue` property from it. The `nodeValue` just obtained is now set with the content of the `textarea` tag present in the page. The content of the `textarea` comes from the value property of the `textarea` DOM element, achieved through the `getElementById` method. Listing 9 shows how to inject the text typed by the user into the note just cloned (second goal).

### Listing 9. Injecting text from the text area into the note just cloned

```javascript
addNote : function(event) {
    ...
    // Set the content of the node
    newNode.getElementsByTagName("p")[0].firstChild.nodeValue =
    document.getElementById("contentArea").value;
    notes.appendChild(newNode);
}
```

For the last goal, append the new note created to the list of notes using the `appendChild` method, as shown in Listing 10.

### Listing 10. Appending the new note to the list

```javascript
addNote : function(event) {
    ...
    notes.appendChild(newNode);
}
```
Finally, you need to prevent the default behavior for the click event (which, for an anchor, is redirecting the user to the URL specified in the href attribute). The DOM specifies the `preventDefault()` method to accomplish this task, applied on the event, with the parameter passed to the handler function. Again, this method is not supported in IE prior to version 9. To achieve the same goal, in pre-version 9 IE you can set the event.returnValue attribute to false. **Listing 11** shows the code.

**Listing 11. Prevent default behaviour for the click event**

```javascript
addNote : function(event) {
    ...
    event.preventDefault ? event.preventDefault() : event.returnValue = false;
}
```

**Listing 12** shows all of the JavaScript code contained in the script.js file.

**Listing 12. Script.js**

```javascript
window.SA = {
    addEvent : function(element, evType, fn, useCapture) {
        if (element.addEventListener) {
            element.addEventListener (evType, fn, useCapture);
            return true;
        } else if (element.attachEvent) {
            var r = element.attachEvent('on' + evType, fn);
            return r;
        } else {
            element['on' + evType] = fn;
        }
    },
    load : function() {
        var anchorSelected;
        if (document.getElementsByClassName) {
            anchorSelected = document.getElementsByClassName("add")[0];
        } else {
            var anchors = document.getElementsByTagName("a");
            length = anchors.length;
            for (var i = 0; i < length; i++) {
                var anchor = anchors[i];
                if (anchor.className === "add") {
                    anchorSelected = anchor;
                }
            }
            SA.addEvent(anchorSelected, "click", SA.addNote, false);
        }
    },
    addNote : function(event) {
        var notes = document.getElementById("notes");
        // Clone the node
        ...
```
var newNode = notes.children[0].cloneNode(true);

    // Set the content of the node
    newNode.getElementsByTagName("p")[0].firstChild.nodeValue
    = document.getElementById("contentArea").value;

    notes.appendChild(newNode);

    event.preventDefault ? event.preventDefault() : event.returnValue = false;

})

SA.addEventListener(window, "load", SA.load, false);

### JavaScript libraries and the DOM

When developers write JavaScript code they often use JavaScript libraries, or frameworks, that handle the different implementations of the DOM in different browsers. Note how one could rewrite Listing 13 using the popular jQuery library.

**Listing 13. Script-jquery.js**

```javascript
$(function(){
    $('a.add').click(function(){
        var newNote = $('.note').eq(0).clone();
        newNote.find('p').text($('#contentArea').val());
        $('#notes').append(newNote);
        return false;
    });
})
```

Quite a few lines of code were saved, and the code is neat and clean.

Since you need to import the library in the HTML in order to use jQuery, as shown in Listing 14, one additive HTTP request will be made and will require more time to execute the library. This process could make an application slower, so it's up to you to decide how to balance using a library with writing less code.

**Listing 14. Importing the library into HTML**

```html
<html>
    <head>
        <script src="http://ajax.googleapis.com/ajax/libs/jquery/1.6.1/jquery.min.js"></script>
    </head>
</html>
```

JavaScript libraries are very powerful tools that can make your life easier. However, you need to know DOM scripting, because using libraries is not always the most efficient way to deal with the DOM. It is also suggested that you learn about what is happening behind the scenes of a library.

### Conclusion

DOM is important to web developers, since it's the way JavaScript accesses web pages. There are a few issues and limitations in the way browser vendors implement the DOM API. Some of the attributes and
methods are not fully supported across all the browsers (for example, `addEventListener()`, `textContent`), or in some cases they behave differently.

Performance is another important factor to consider with DOM scripting. As demonstrated in this article, you can leverage some of the JavaScript frameworks to manipulate and traverse the DOM, as long as you know how JavaScript and DOM interact.
## Downloadable resources

<table>
<thead>
<tr>
<th>Description</th>
<th>Name</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Article source code</td>
<td>DomScripting.zip</td>
<td>5KB</td>
</tr>
</tbody>
</table>
Related topics

- Check out JavaScript binding for the DOM Level 2 Core definitions.
- Read the W3C DOM Level 2 Core Specification.
- Explore the structure of a DOM document in the tutorial "Understanding DOM" (developerWorks, Mar 2007).
- Read "JavaScript and the Document Object Model" (developerWorks, Jul 2002) to learn about the JavaScript approach to DOM and the building of a Web page that allows the user to add notes and edit note content.
- Go through the JavaScript tutorial (w3schools) if you're new to JavaScript.
- Read "Get started with the JavaScript language" (developerWorks, Apr 2011) to explore basic JavaScript concepts for beginners and code examples that show how it's all done.
- Access a comprehensive guide to the JavaScript language at Dev Guru.
- Learn more about the JavaScript language by reading "Classical Inheritance in JavaScript" by Douglas Crockford.
- Learn more about DOM scripting at Dynamic Content with DOM-2.
- Check out the W3C DOM Level 3 Events Specification
- Read Chapter 3 of High Performance JavaScript (O'Reilly Media), by Nicholas C. Zakas, for more about DOM scripting.
- Get specifics about DOM Level 3 Events support in IE9.
- To listen to interesting interviews and discussions for software developers, check out developerWorks podcasts.
- Try out IBM software for free. Download a trial version, log into an online trial, work with a product in a sandbox environment, or access it through the cloud. Choose from over 100 IBM product trials.

© Copyright IBM Corporation 2011
Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)