Introducing the JavaScript DOM

Tutorial by Matt Doyle | Level: Beginner | Published on 3 October 2008
Categories:

- *Web Development > JavaScript > The Document Object Model*

What is the Document Object Model, and why is it useful? This article gives you a gentle introduction to this powerful JavaScript feature.

The Document Object Model lets you access and manipulate the contents of Web pages using JavaScript. By using the DOM, you can do wonderful things like:

- Create tabbed Web pages
- Create expandable/collapsible ("accordion"-style) Web page elements
- Generate Web page content dynamically (on the fly)

In this introductory article, you learn about the concept of the DOM, and how it's used to access Web page elements from within JavaScript.

The DOM concept

From a JavaScript perspective, the Document Object Model uses JavaScript objects to represent the various parts that make up a Web page. For example, a whole Web page is represented by a `Document` object, while elements within the page — `body`, `h1`, `p`, and so on — are represented by `Element` objects. The page elements are represented as a tree, with the `Document` object at the top.

Elements, attributes and text nodes

Every part of a Web page is represented in the DOM by a ***node***. The most common nodes you'll encounter are ***element nodes***, ***attribute nodes***, and ***text nodes***. In addition, the whole document is represented by a ***document node***. Consider the following HTML markup:

```
<h1 class="intro">Introducing the Widget Company</h1>
```

In the above example, the `h1` element is a DOM Element node, the `class` attribute is a DOM Attribute node, and the `Introducing the Widget Company` text is a DOM Text node.

The DOM tree

A Web page is represented in the DOM as a tree of nodes. For example, here's the markup for a simple XHTML page:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
        "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en"
lang="en">
  <head>
    <title>My Web Page</title>
  </head>
  <body>
    <h1 class="intro">The Widget Company</h1>
  </body>
</html>
```

The above Web page is represented in the DOM by the following tree:

```
                Document
                    |
                  HTML
      ______________|______________
     |                             |
   HEAD                          BODY
     |                             |
   TITLE                          H1
     |                             |
"My Web Page"             "The Widget Company"
```

In this tree, `Document` is a DOM Document node, while `HTML`, `HEAD`, `BODY`, `TITLE` and`H1` are all DOM Element nodes, and `"My Web Page"` and `"The Widget Company"` are DOM Text nodes.

The `H1` element node has a DOM Attribute node associated with it, which has a name of `"class"` and a value of `"intro"`.

The `HTML` element node also has three DOM Attribute nodes: `"xmlns"`, with value of`"http://www.w3.org/1999/xhtml"`; `"xml:lang"`, with a value of `"en"`; and `"lang"`, also with a value of `"en"`.

The DOM and JavaScript

You can access the various element, attribute and text nodes in a DOM document as JavaScript objects, allowing you to read and alter the contents of nodes, as well as add new nodes to the tree or remove nodes from the tree.

Over the coming series of tutorials, you'll learn how to manipulate the DOM using JavaScript, and use your new-found knowledge to create powerful, dynamic Web pages.

To kick things off, in the next tutorial you learn how to use JavaScript to retrieve DOM elements in a Web page. Enjoy!

Retrieving Page Elements via the DOM

Tutorial by Matt Doyle | Level: Intermediate | Published on 31 October 2008

Categories:

- *Web Development > JavaScript > The Document Object Model*

How can you use the Document Object Model to access the various parts of your Web page? This tutorial shows you how to retrieve HTML elements using three handy DOM methods.

In the last DOM article, you learned that the DOM represents the contents of a Web page as a "tree" of JavaScript objects. By accessing the parts of the tree, called the ***nodes***, you can read existing page content, alter content, and even add new content from scratch.

In this article you'll learn how to locate and retrieve the elements of a Web page using JavaScript and the DOM.

Here's a simple XHTML Web page:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
        "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en"
lang="en">
  <head>
    <title>My Web Page</title>
  </head>
  <body>
    <h1 class="intro">The Widget Company</h1>
    <p>Welcome to the Widget Company!</p>
    <p>Widgets for sale:</p>
    <ul>
      <li id="widget1"><a
href="superwidget.html">SuperWidget</a></li>
      <li id="widget2"><a
href="megawidget.html">MegaWidget</a></li>
      <li id="widget3"><a
href="wonderwidget.html">WonderWidget</a></li>
    </ul>
    <form method="post" action="">
      <div>
        <label for="widgetName">Enter the name of a widget to
buy it:</label>
        <input type="text" name="widgetName" id="widgetName" />
        <input type="submit" name="buy" value="Buy" />
      </div>
    </form>
  </body>
</html>
```

All the elements in this page can be accessed via the single `Document` object that represents the page.

Finding elements by ID

The easiest way to access an element is via its `id` attribute, because an element's `id` is unique in the page. Here's how it's done:

```
var element = document.getElementById( elementId );
```

So you could access the second list item in the above Web page ("MegaWidget") as follows:

```
var widget2 = document.getElementById( "widget2" );
alert( widget2 );
```

The above code displays an alert box containing:

```
[object HTMLLIElement]
```

So, `document.getElementById()` returns an object representing the element (in this case an `HTMLLIElement`, or "HTML list item element", object).

While most modern browsers display the object type (such as `[object HTMLLIElement]`), Internet Explorer (at the time of writing) rather unhelpfully just displays `[object]`.

Finding elements by tag name

Using `document.getElementById()` is all very well if the element you're after has an ID (or you can easily add one). How do you access elements that don't have an ID?

One way is to find all the elements of certain type, such as all `h1` elements or all `p`elements. You can do this using `document.getElementsByTagName()`:

```
var elements = document.getElementsByTagName( tagName );
```

`document.getElementsByTagName()` returns a collection of all the elements with that particular tag name. For example, this code retrieves the two paragraph elements in the Web page:

```
var paragraphs = document.getElementsByTagName( "p" );
alert( paragraphs );
```

The above code displays:

```
[object HTMLCollection]
```

A ***collection*** is simply a special type of object that itself contains a list of objects — in this case, `HTMLParagraphElement` objects. You can loop through the collection with a`for` loop, using the collection's `length` property to determine how many objects are in the collection:

```
var paragraphs = document.getElementsByTagName( "p" );
for ( var i = 0; i < paragraphs.length; i++ ) {
  alert( paragraphs[i] );
}
```

This code displays two alert boxes, each containing:

```
[object HTMLParagraphElement]
```

Finding elements via their `name` attributes

If you have elements within your page that have `name` attributes — such as form `input`fields and `select` menus — then you can retrieve those elements via their names with`document.getElementsByName`:

```
var elements = document.getElementsByName( name );
```

As with `getElementsByTagName()`, `getElementsByName()` returns a collection of elements with the given name.

For example:

```
var widgetNameField = document.getElementsByName( "widgetName"
)[0];
alert( widgetNameField );
```

This code pops up an alert displaying the object that represents the `"widgetName"``input` text field in the form:

```
[object HTMLInputElement]
```

Finding elements by their relationship

Because each node in the DOM tree is related to every other node, once you have retrieved one node you can theoretically access any other node in the tree by hopping between the nodes. To do this, you use properties of each node such as `parentNode`(to retrieve the node's parent) and `childNodes` (to retrieve the children of the node). You'll learn more about this technique in the next tutorial.

Digging deep into elements

In this tutorial you learned how to retrieve any HTML element within a Web page. You used `getElementById()` to retrieve an element with a specific ID,`getElementsByTagName()` to get all the elements of a certain type, and`getElementsByName()` to find all elements with a particular `name` attribute.

However, simply getting these element objects doesn't really help much. How do you access the contents of an element? Find out in our next tutorial, Looking inside DOM page elements.

Home : Articles : Looking Inside DOM Page Elements

Looking Inside DOM Page Elements

Tutorial by Matt Doyle | Level: Intermediate | Published on 7 November 2008
Categories:

- *Web Development > JavaScript > The Document Object Model*

This article shows you how to access the contents and attributes of any DOM element in a Web page. You also find out about node properties and relationships, and learn how to move around the DOM tree.

In our last DOM tutorial, you learned how to access the elements inside your Web page as JavaScript objects. Once you've done that, how do you find out more about each element? This tutorial shows you how to delve deep into any DOM element object.

Everything is a node

As you've seen in our previous tutorials, the Document Object Model breaks an entire Web page down into a tree of node objects. Elements are nodes; attributes are nodes; chunks of text are nodes. Even the document itself is a node.

All nodes are related to each other, with the `document` node at the top of the tree. For example, if you have a `p` element inside a `div` element at the top of your Web page, the `p` node is a child of the `div` node. In turn, the `div` node is a child of the `body`node.

Once you understand this concept, it's easy to manipulate DOM elements. For example, to access the text inside a paragraph, you retrieve the paragraph node's child text node, then read that node's value. To access an attribute of a paragraph, you retrieve its attribute node, and so on.

Finding out about a node

Each node in the DOM tree contains three properties that tell you about the node:

`nodeType`

An integer value representing the type of the node (element, attribute, text, and so on). See below for details.

`nodeName`

The name of the node, as a string. For example, the`nodeName` of an `h1` element is `"H1"`.

`nodeValue`

The value of the node. For element nodes, the value will be `null`. For text nodes, the value is the text itself. For attribute nodes, the value is the attribute's value, and so on.

The `nodeType` property is an integer that tells you the type of the node. The value corresponds to one of twelve constants. Here's a list of the ones you're most likely to use:

Value	Constant	Description
1	`Node.ELEMENT_NODE`	The node is an (X)HTML element
2	`Node.ATTRIBUTE_NODE`	The node is an attribute of an element
3	`Node.TEXT_NODE`	The node is a chunk of plain text
8	`Node.COMMENT_NODE`	The node is an (X)HTML comment
9	`Node.DOCUMENT_NODE`	The node is the `Document` node

Say you have the following `p` element in your Web page:

```
<p id="welcome">Welcome to the Widget Company!</p>
```

You could find out some info about this node as follows:

```
var element = document.getElementById( "welcome" );
alert ( element.nodeType );  // Displays "1"
alert ( element.nodeName );  // Displays "P"
alert ( element.nodeValue ); // Displays "null"
```

Node relationships

Once you have one node, you can access any other node that is related to it. All nodes have the following properties:

`childNodes`

A collection of all the children of the node

`firstChild`

The first node in the collection of child nodes

`lastChild`	
	The last node in the collection of child nodes
`nextSibling`	
	The next node that has the same parent as the node
`previousSibling`	
	The previous node that has the same parent as the node
`parentNode`	
	The node's parent

Let's say you have the following markup in your page:

```
<ul>
  <li id="widget1"><a
href="superwidget.html">SuperWidget</a></li>
  <li id="widget2"><a
href="megawidget.html">MegaWidget</a></li>
  <li id="widget3"><a
href="wonderwidget.html">WonderWidget</a></li>
</ul>
```

The following JavaScript displays the text inside the second link ("MegaWidget"):

```
var widget2 = document.getElementById( "widget2" );
alert( widget2.firstChild.firstChild.nodeValue );
```

The first line uses a `widget2` variable to store the `li` element node with the `id` of`"widget2"`.

The second line displays the text inside the link. The first child of the `"widget2" li`element is the `a` element, and the first child of that `a` element is the text node, whose`nodeValue` is "MegaWidget".

Example: Getting all paragraph text in a page

Here's a simple page containing, amongst other things, three paragraphs of text. There's also a JavaScript function, `displayParas()`, triggered by clicking the "Display paragraph text" link, that displays the text inside each paragraph in the page:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
        "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en"
lang="en">
  <head>
    <title>My Web Page</title>
    <script type="text/javascript">
      // <![CDATA[
        function displayParas() {
          var output = "";
          var paras = document.getElementsByTagName( "p" );

          for ( i=0; i < paras.length; i++ ) {
            for ( j=0; j < paras[i].childNodes.length; j++ ) {
              if ( paras[i].childNodes[j].nodeType ==
Node.TEXT_NODE ) {
                output += paras[i].childNodes[j].nodeValue +
"\n";
              }
            }
          }

          alert( output );
        }
      // ]]>
    </script>
  </head>
  <body>
    <h1>The Widget Company</h1>
    <p>Welcome to the Widget Company!</p>
    <p>We have lots of fantastic widgets for sale.</p>
    <p>Feel free to browse!</p>
    <p><a href="javascript:displayParas()">Display paragraph 
text</a></p>
  </body>
</html>
```

Try it out! Click the "Display paragraph text" link, then use your Back button to return here.

Clicking the link displays an alert box with the following contents:

```

Welcome to the Widget Company!
We have lots of fantastic widgets for sale.
Feel free to browse!
```

First, the code stores a list of the page's `p` elements in a `paras` variable. It then loops through the elements in `paras`. For each element, it loops through all the child nodes of the element. When it finds a text node, it adds its value (the text) to the `output`string, which is then displayed using `alert()` at the end of the script.

You may be wondering why the fourth paragraph's text, "Display paragraph text", isn't displayed. This is because the child node of the fourth `p` node is in fact the `a`element, so the code skips it. The "Display paragraph text" node is the *grandchild* of the `p` node, so if you want to access this text, you have to dig deeper than the `p`node's child.

Be wary of whitespace!

One thing to watch out for when retrieving child nodes is any whitespace in the HTML markup. Consider the following markup:

```
    <div id="welcome">
      <p>Welcome to the Widget Company!</p>
    </div>
```

You might expect the following code to display the value "1" (`Node.ELEMENT_NODE`), because the child node of the `"welcome" div` appears to be the `p` element:

```
var ul = document.getElementById( "welcome" );
alert( ul.firstChild.nodeType );
```

In fact it displays "3" (`Node.TEXT_NODE`). This is because the whitespace — the carriage return and space/tab characters — between the opening `div` tag and the opening `p`tag is in fact a text node in its own right. This text node is the first child of the `div`node.

In order to accurately locate the paragraph element node, you need to loop through the child nodes until you find the correct node:

```
var ul = document.getElementById( "welcome" );
var para = null;

for ( i=0; i < ul.childNodes.length; i++ ) {
  if ( ul.childNodes[i].nodeType == Node.ELEMENT_NODE &&
ul.childNodes[i].nodeName == "P" ) {
    para = ul.childNodes[i];
    break;
  }
}

// Displays "Welcome to the Widget Company!"
if ( para ) alert( para.firstChild.nodeValue );
```

This is obviously tedious, so it's a good idea to wrap code such as this inside a function so that you can reuse it.

Retrieving attributes

Attribute nodes are stored a bit differently to other nodes. An element's attribute nodes aren't children of the element; instead, you access the nodes through the element's`attributes` collection:

```
var attributes = element.attributes;
```

Say you have the following form field in your Web page:

```
<input type="text" name="widgetName" id="widgetName" />
```

The following code displays each of the attributes of the field:

```
var output = "";
var widgetName = document.getElementById( "widgetName" );
var attrs = widgetName.attributes;
for ( i=0; i < attrs.length; i++ ) {
  output += ( attrs[i].name + "=" + attrs[i].value ) + "\n";
}
alert( output );

type=text
id=widgetName
name=widgetName
```

Note that the attribute nodes aren't in any particular order.

You can also retrieve an attribute node directly if you know its name:

```
var widgetName = document.getElementById( "widgetName" );
alert( widgetName.attributes["type"].value ); // Displays
"text"
```

The following methods also let you retrieve an attribute by name:

***element*.getAttributeNode (*name*)**
: Returns the attribute node called *name*

***element*.getAttribute (*name*)**
: Returns the value of the attribute node called *name*

For example:

```
var widgetName = document.getElementById( "widgetName" );
alert( widgetName.getAttribute( "type" )); // Displays "text"
```

By the way, you can test if an element contains a particular attribute with the`element.hasAttribute()` method:

```
result = element.hasAttribute( attributeName )
```

`hasAttribute()` returns `true` if the element contains the attribute named*`attributeName`*; `false` otherwise.

Now you've read this tutorial, you can hop around from one node in the DOM tree to another, and you can dig deep inside any element node to view its contents – that is, its child nodes and its attributes. However, so far you've merely retrieved element content. In the next tutorial you'll learn how to alter the contents of an element, as well as add and remove elements in a Web page.

Changing Page Elements with the DOM

Tutorial by Matt Doyle | Level: Intermediate | Published on 28 November 2008

Categories:

- *Web Development* > *JavaScript* > *The Document Object Model*

Learn how to use the JavaScript DOM to alter the content of your Web pages. This article shows you how to change element content, how to add and remove elements in the page, how to move elements around, and how to work with element attributes.

So far, you've looked at how to get hold of page elements using JavaScript and the DOM, and how to look inside those elements to retrieve their contents.

In this tutorial, you'll take your skills a step further, and learn how to *change* elements. You'll see how to:

- Change the content inside an element
- Add and remove elements in the page
- Move an element to a different position in the page, and
- Manipulate element attributes.

Once you know how to manipulate DOM elements you can start creating flexible, JavaScript-generated Web content.

Changing the contents of an element

In the last tutorial you learned how to read the contents of an element using properties such as `childNodes`, `firstChild`, and `nodeValue`. As well as reading the value of a node with the `nodeValue` property, you can also change its value simply by assigning a new value to `nodeValue`.

For example, say your page contains a paragraph element inside a `div` element:

```
<div id="welcome"><p>Welcome to the Widget Company!</p></div>
```

You already know that you can read the text inside the paragraph using:

```
var welcome = document.getElementById( "welcome" );
alert( welcome.firstChild.firstChild.nodeValue );
```

Changing the paragraph's text is as simple as assigning a new value to the child text node's `nodeValue` property:

```
var welcome = document.getElementById( "welcome" );
welcome.firstChild.firstChild.nodeValue = "Welcome to Widgets R
Us!";
```

What if you want to make more radical changes to your page? Let's say you want to add a horizontal rule (`hr`) element after the paragraph. To do this, you can't just change a node's `nodeValue`. Instead, you need to add a new child element to the `"welcome" div` element, as discussed in the following section.

Adding, removing and replacing elements

You know that a Web page is made up of a tree of DOM nodes. For example, a paragraph containing a string of text and an image is represented by a `p` element node with two child nodes: a text node, and an `img` element node.

This means that, in order to alter the contents of an element that itself contains child elements, you need to be able to add and remove those child elements. Here's how to do it.

Adding elements

To add an element to the DOM tree, you first need to create the element. You do this using the `document.createElement()` method, passing in the tag name of the element you want to create:

```
elementVar = document.createElement( tagName );
```

Once you've created your element node, you can then add it as a child to an existing node in the document using `element.appendChild()`:

```
element.appendChild( elementVar );
```

The following example adds a horizontal rule (`hr`) element after the paragraph inside our `"welcome"` `div`:

```
var welcome = document.getElementById( "welcome" );
var horizRule = document.createElement( "hr" );
welcome.appendChild( horizRule );
```

What if you wanted to insert the horizontal rule *before* the paragraph? You can use`element.insertBefore()` to insert an element before a specified child node, rather than at the end of the list of child nodes. `insertBefore()` takes two parameters: the element to insert, and the child node to insert the element before.

So to insert the horizontal rule before the paragraph, first create the `hr` element, then retrieve the paragraph node, then call
the `div` element's `insertBefore()` method, as follows:

```
var welcome = document.getElementById( "welcome" );
var horizRule = document.createElement( "hr" );
var paragraph = welcome.firstChild;
welcome.insertBefore( horizRule, paragraph );
```

Removing elements

To remove an element from the page, first retrieve the element using, for example,`document.getElementById()` or `element.firstChild`. Once you have the element, you can then remove it from its parent element by calling `parent.removeChild()`, passing in the element to remove:

```
removedElement = element.removeChild( elementToRemove );
```

`removeChild()` removes the element from the DOM tree and returns it. You can just ignore the returned object, or store it for later use if required.

For example, here's how to remove the `hr` element that you just added after your paragraph:

```
var welcome = document.getElementById( "welcome" );
var horizRule = welcome.lastChild;
welcome.removeChild( horizRule );
```

You can use `appendChild()` and `removeChild()` to add or remove any type of DOM node, not just elements — provided, of course, that the node in question can be added to or removed from the parent element.

Replacing elements

You can also replace one element in the DOM tree with another. For example, you could replace the `hr` element in the previous example with a new `p` element:

```
var welcome = document.getElementById( "welcome" );
var horizRule = welcome.lastChild;
welcome.removeChild( horizRule );
var newPara = document.createElement( "p" );
newPara.appendChild( document.createTextNode( "Feel free to
have a browse." ) );
```

```
welcome.appendChild( newPara );
```

Note the use of a new method, `document.createTextNode()`. This works much like`document.createElement()`, but it returns a text node containing the supplied text.

Also notice that you can add child nodes to a node with `appendChild()`, even though the parent node isn't yet part of the main document's DOM tree. For example, in the above code the text node is added to the `newPara` element before`newPara` is added to the `"welcome" div` element.

Although the above approach works, there's a cleaner way to replace an element, and that is to use `element.replaceChild()`. This method takes the new child to use, followed by the child to replace, and returns the replaced child (which you can ignore if you don't need it). So the above code can be rewritten as:

```
var welcome = document.getElementById( "welcome" );
var horizRule = welcome.lastChild;
var newPara = document.createElement( "p" );
newPara.appendChild( document.createTextNode( "Feel free to
have a browse." ) );
welcome.replaceChild( newPara, horizRule );
```

Moving elements around

Now that you know how to add, remove, and replace elements, you can easily move elements from one part of the DOM tree to another. Here's a simple example. Imagine that your page contains the following list:

```
<ul>
  <li id="widget1"><a
href="superwidget.html">SuperWidget</a></li>
  <li id="widget2"><a
href="megawidget.html">MegaWidget</a></li>
```

```
  <li id="widget3"><a
href="wonderwidget.html">WonderWidget</a></li>
</ul>
```

To move the SuperWidget list item to the end of the list, you could use:

```
var superWidget = document.getElementById( "widget1" );
var ul = superWidget.parentNode;
ul.appendChild( superWidget );
```

Notice that you don't need to remove the SuperWidget list item from the list before you then append it. When you use `appendChild()`, `insertBefore()`, or `replaceChild()`to add an element that already exists in the page, the element is automatically removed from its old position in the page. In other words, the element is moved rather than copied.

Changing attributes

So far you've looked at changing the contents of elements: you've altered the value of a text node inside an element, and manipulated the child nodes of an element. How about changing the *attributes* of an element?

Changing an attribute's value

In the last tutorial you learned how to retrieve an element's attribute node using`getAttributeNode()`. Once you have the attribute, you can change its `value` property. This immediately updates the attribute's value in the page. Say you have a photo of your cat in your Web page:

```
<div>
  <img id="photo" src="cat.jpg" alt="My cat" />
</div>
```

To replace the cat photo with a dog photo, updating both
the `src` and `alt` attributes, you could write:

```
var photo = document.getElementById( "photo" );
var photoSrc = photo.getAttributeNode( "src" );
var photoAlt = photo.getAttributeNode( "alt" );
photoSrc.value = "dog.jpg";
photoAlt.value = "My dog";
```

Adding and removing attribute nodes

To add a new attribute node to an element, first create the attribute node by calling`document.createAttribute()`, passing in the name of the attribute to create. The method returns the attribute node:

```
attributeNode = document.createAttribute( attributeName );
```

Then you can assign a value to your new attribute node's `value` property:

```
attributeNode.value = value;
```

Finally, add the attribute to the element by
calling `element.setAttributeNode()`:

```
element.setAttributeNode( attributeNode );
```

If an attribute with the same name already exists for the element, it is replaced by the new attribute node. The old attribute node is then returned by`element.setAttributeNode()`.

Here's an example that adds the attribute `width="50"` to your

"photo" `img` element:

```
var photo = document.getElementById( "photo" );
var widthAttr = document.createAttribute( "width" );
widthAttr.value = 50;
photo.setAttributeNode( widthAttr );
```

To remove an attribute node, call `element.removeAttributeNode()`, passing in the node to remove:

```
var photo = document.getElementById( "photo" );
var widthAttr = photo.getAttributeNode( "width" );
photo.removeAttributeNode( widthAttr );
```

`removeAttributeNode()` returns the removed attribute node, which you can store for later, or ignore.

Attribute shortcuts

If you're thinking that all of this retrieving, creating, adding, and removing of attribute nodes is somewhat tedious, you'd be right. Fortunately the DOM provides a couple of shortcut methods to make life easier. These methods work directly with `element`objects, rather than dealing with attribute nodes:

`element.setAttribute( name, value )`

Adds an attribute with a name of `name` and a value of `value` to the element. If an attribute with the same name already exists in the element, its value is changed to `value`.

`element.removeAttribute( name )`

Removes the attribute called `name` from the element.

For example:

```
var photo = document.getElementById( "photo" );
photo.setAttribute( "width", 50 );
photo.removeAttribute( "width" );
```

Most of the time you'll probably want to use `setAttribute()` and `removeAttribute()`rather than the node-based approach. They're quick and easy, and you rarely need to work at the node level anyway.

Now that you've read this tutorial, you have pretty much all you need to create an entire Web page from scratch using nothing but DOM objects. You also have the knowledge to manipulate an existing Web page using the DOM, including changing element content, adding and removing elements, moving elements around, and altering element attributes.