

The **Document Object Model (DOM)** is the data representation of the objects that comprise the structure and content of a document on the web. In this guide, we'll briefly introduce the DOM. We'll look at how the DOM represents an HTML or XML document in memory and how you use APIs to create web content and applications.

What is the DOM?

The Document Object Model (DOM) is a programming interface for HTML and XML documents. It represents the page so that programs can change the document structure, style, and content. The DOM represents the document as nodes and objects. That way, programming languages can connect to the page.

A Web page is a document. This document can be either displayed in the browser window or as the HTML source. But it is the same document in both cases. The Document Object Model (DOM) represents that same document so it can be manipulated. The DOM is an object-oriented representation of the web page, which can be modified with a scripting language such as JavaScript.

The W3C DOM and WHATWG DOM standards are implemented in most modern browsers. Many browsers extend the standard, so care must be exercised when using them on the web where documents may be accessed by various browsers with different DOMs.

For example, the standard DOM specifies that the `getElementsByTagName` method in the code below must return a list of all the `<p>` elements in the document:

```
const paragraphs = document.getElementsByTagName("p");
// paragraphs[0] is the first <p> element
// paragraphs[1] is the second <p> element, etc.
alert(paragraphs[0].nodeName);
```

All of the properties, methods, and events available for manipulating and creating web pages are organized into objects (for example, the `document` object that represents the document itself, the `table` object that implements the special `HTMLTableElement` DOM interface for accessing HTML tables, and so forth). This documentation provides an object-by-object reference to the DOM.

The modern DOM is built using multiple APIs that work together. The core DOM defines the objects that fundamentally describe a document and the objects within it. This is expanded upon as needed by other APIs that add new features and capabilities to the DOM. For example, the HTML DOM API adds support for representing HTML documents to the core DOM.

DOM and JavaScript

The short example above, like nearly all of the examples in this reference, is JavaScript. That is to say, it's *written* in JavaScript, but it *uses* the DOM to access the document and its elements. The DOM is not a programming language, but without it, the JavaScript language wouldn't have any model or notion of web pages, HTML documents, XML documents, and their component parts (e.g. elements). Every element in a document—the document as a whole, the head, tables within the document, table headers, text within the table cells—is part of the document object model for that document, so they can all be accessed and manipulated using the DOM and a scripting language like JavaScript.

In the beginning, JavaScript and the DOM were tightly intertwined, but eventually, they evolved into separate entities. The page content is stored in the DOM and may be accessed and manipulated via JavaScript, so that we may write this approximative equation:

API = DOM + JavaScript

The DOM was designed to be independent of any particular programming language, making the structural representation of the document available from a single, consistent API. Though we focus exclusively on JavaScript in this reference documentation, implementations of the DOM can be built for any language, as this Python example demonstrates:

```
# Python DOM example
import xml.dom.minidom as m
doc = m.parse(r"C:\Projects\Py\chap1.xml")
doc.nodeName # DOM property of document object
p_list = doc.getElementsByTagName("para")
```

For more information on what technologies are involved in writing JavaScript on the web, see [JavaScript technologies overview](#).

Accessing the DOM

You don't have to do anything special to begin using the DOM. Different browsers have different implementations of the DOM, and these implementations exhibit varying degrees of conformance to the actual DOM standard (a subject we try to avoid in this documentation), but every web browser uses some document object model to make web pages accessible via JavaScript.

When you create a script—whether it's inline in a `<script>` element or included in the web page by means of a script loading instruction—you can immediately begin using the API for the `document` or `window` elements to manipulate the document itself or to get at the children of that document, which are the various elements in the web page. Your DOM programming may be something as simple as the following, which displays an alert message by using the `alert()` function from the `window` object, or it may use more sophisticated DOM methods to actually create new content, as in the longer example below.

This following JavaScript will display an alert when the document is loaded (and when the whole DOM is available for use):

```
<body onload="window.alert('Welcome to my home page!');">
```

Another example. This function creates a new H1 element, adds text to that element, and then adds the H1 to the tree for this document:

```
<html>
  <head>
    <script>
      // run this function when the document is loaded
      window.onload = function() {

        // create a couple of elements in an otherwise empty HTML page
        const heading = document.createElement("h1");
        const heading_text = document.createTextNode("Big Head!");
        heading.appendChild(heading_text);

        document.body.appendChild(heading);
      }
    </script>
  </head>
  <body>
  </body>
</html>
```

Fundamental data types

This reference tries to describe the various objects and types in simple terms. But there are a number of different data types being passed around the API that you should be aware of.

Note: Because the vast majority of code that uses the DOM revolves around manipulating HTML documents, it's common to refer to the nodes in the DOM as **elements**, although strictly speaking not every node is an element.

The following table briefly describes these data types.

Data type (Interface)	Description
Document	When a member returns an object of type <code>document</code> (e.g., the <code>ownerDocument</code> property of an element returns the document to which it belongs), this object is the root document object itself. The DOM document Reference chapter describes the document object.
Node	Every object located within a document is a node of some kind. In an HTML document, an object can be an element node but also a text node or attribute node.
Element	The <code>element</code> type is based on <code>node</code> . It refers to an element or a node of type <code>element</code> returned by a member of the DOM API. Rather than saying, for example, that the <code>document.createElement()</code> method returns an object reference to a <code>node</code> , we just say that this method returns the <code>element</code> that has just been created in the DOM. <code>element</code> objects implement the DOM <code>Element</code> interface and also the more basic <code>Node</code> interface, both of which are included together in this reference. In an HTML document, elements are further enhanced by the HTML DOM API's <code>HTMLElement</code> interface as well as other interfaces describing capabilities of specific kinds of elements (for instance, <code>HTMLTableElement</code> for <code><table></code> elements).
NodeList	<p>A <code>nodeList</code> is an array of elements, like the kind that is returned by the method <code>document.querySelectorAll()</code>. Items in a <code>nodeList</code> are accessed by index in either of two ways:</p> <ul style="list-style-type: none">• <code>list.item(1)</code>• <code>list[1]</code> <p>These two are equivalent. In the first, <code>item()</code> is the single method on the <code>nodeList</code> object. The latter uses the typical array syntax to fetch the second item in the list.</p>
Attribute	When an attribute is returned by a member (e.g., by the <code>createAttribute()</code> method), it is an object reference that exposes a special (albeit small) interface for attributes. Attributes are nodes in the DOM just like elements are, though you may rarely use them as such.

Data type (Interface)	Description
NamedNodeMap	A <code>namedNodeMap</code> is like an array, but the items are accessed by name or index, though this latter case is merely a convenience for enumeration, as they are in no particular order in the list. A <code>namedNodeMap</code> has an <code>item()</code> method for this purpose, and you can also add and remove items from a <code>namedNodeMap</code> .

There are also some common terminology considerations to keep in mind. It's common to refer to any **Attribute** node as simply an `attribute`, for example, and to refer to an array of DOM nodes as a `nodeList`. You'll find these terms and others to be introduced and used throughout the documentation.

DOM interfaces

This guide is about the objects and the actual *things* you can use to manipulate the DOM hierarchy. There are many points where understanding how these work can be confusing. For example, the object representing the HTML form element gets its `name` property from the `HTMLFormElement` interface but its `className` property from the `HTMLElement` interface. In both cases, the property you want is simply in that form object.

But the relationship between objects and the interfaces that they implement in the DOM can be confusing, and so this section attempts to say a little something about the actual interfaces in the DOM specification and how they are made available.

Interfaces and Objects

Many objects borrow from several different interfaces. The `table` object, for example, implements a specialized `HTMLTableElement` interface, which includes such methods as `createCaption` and `insertRow`. But since it's also an HTML element, `table` implements the `Element` interface described in the DOM `Element` Reference chapter. And finally, since an HTML element is also, as far as the DOM is concerned, a node in the tree of nodes that make up the object model for an HTML or XML page, the `table` object also implements the more basic `Node` interface, from which `Element` derives.

When you get a reference to a `table` object, as in the following example, you routinely use all three of these interfaces interchangeably on the object, perhaps without knowing it.

```
const table = document.getElementById("table");
const tableAttrs = table.attributes; // Node/Element interface
for (let i = 0; i < tableAttrs.length; i++) {
```

```
// HTMLTableElement interface: border attribute
if(tableAttrs[i].nodeName.toLowerCase() == "border")
    table.border = "1";
}
// HTMLTableElement interface: summary attribute
table.summary = "note: increased border";
```

Core Interfaces in the DOM

This section lists some of the most commonly-used interfaces in the DOM. The idea is not to describe what these APIs do here but to give you an idea of the sorts of methods and properties you will see very often as you use the DOM. These common APIs are used in the longer examples in the DOM Examples chapter at the end of this book.

The document and window objects are the objects whose interfaces you generally use most often in DOM programming. In simple terms, the window object represents something like the browser, and the document object is the root of the document itself. Element inherits from the generic Node interface, and together these two interfaces provide many of the methods and properties you use on individual elements. These elements may also have specific interfaces for dealing with the kind of data those elements hold, as in the table object example in the previous section.

The following is a brief list of common APIs in web and XML page scripting using the DOM.

- `document.getElementById(id)`
- `document.getElementsByTagName(name)`
- `document.createElement(name)`
- `parentNode.appendChild(node)`
- `element.innerHTML`
- `element.style.left`
- `element.setAttribute()`
- `element.getAttribute()`
- `element.addEventListener()`
- `window.content`
- `window.onload`
- `window.scrollTo()`

Testing the DOM API

This document provides samples for every interface that you can use in your own web development. In some cases, the samples are complete HTML pages, with the DOM access in a `<script>` element, the interface (e.g, buttons) necessary to fire up the script in a form, and the HTML elements upon which the DOM operates listed as well. When this is the case, you can cut and paste the example into a new HTML document, save it, and run the example from the browser.

There are some cases, however, when the examples are more concise. To run examples that only demonstrate the basic relationship of the interface to the HTML elements, you may want to set up a test page in which interfaces can be easily accessed from scripts. The following very simple web page provides a `<script>` element in the header in which you can place functions that test the interface, a few HTML elements with attributes that you can retrieve, set, or otherwise manipulate, and the web user interface necessary to call those functions from the browser.

You can use this test page or create a similar one to test the DOM interfaces you are interested in and see how they work on the browser platform. You can update the contents of the `test()` function as needed, create more buttons, or add elements as necessary.

```
<html>
<head>
  <title>DOM Tests</title>
  <script>
    function setBodyAttr(attr, value) {
      if (document.body) document.body[attr] = value;
      else throw new Error("no support");
    }
  </script>
</head>
<body>
  <div style="margin: .5in; height: 400px;">
    <p><b><tt>text</tt></b></p>
    <form>
      <select onChange="setBodyAttr('text',
        this.options[this.selectedIndex].value);">
        <option value="black">black</option>
        <option value="red">red</option>
      </select>
      <p><b><tt>bgColor</tt></b></p>
      <select onChange="setBodyAttr('bgColor',
        this.options[this.selectedIndex].value);">
        <option value="white">white</option>
        <option value="lightgrey">gray</option>
      </select>
      <p><b><tt>link</tt></b></p>
```



```

<select onChange="setBodyAttr('link',
    this.options[this.selectedIndex].value);">
    <option value="blue">blue</option>
    <option value="green">green</option>
</select>
<small>
    <a href="http://some.website.tld/page.html" id="sample">
        (sample link)
    </a>
</small><br />
<input type="button" value="version" onclick="ver()" />
</form>
</div>
</body>
</html>

```

To test a lot of interfaces in a single page—for example, a "suite" of properties that affect the colors of a web page—you can create a similar test page with a whole console of buttons, textfields, and other HTML elements. The following screenshot gives you some idea of how interfaces can be grouped together for testing.

Figure 0.1 Sample DOM Test Page

HTMLBodyElement

alink :

background:

bgColor:

link :

text :

vlink :

text
[link](#)
[visited](#)

NSHTMLDocument

alinkColor:

bgColor :

linkColor :

fgColor :

vlinkColor:

text
[link](#)
[visited](#)

In this example, the drop-down menus dynamically update such DOM—accessible aspects of the web page as its background color (bgColor), the color of the hyperlinks (aLink), and color of the text (text). However, you design your test pages, testing the interfaces as you read about them is an important part of learning how to use the DOM effectively.

