DOM - Document Object Model

Everything you need for DOM – Link: <https://www.javascripttutorial.net/javascript-dom/>

**What is the DOM?**

The Document Object Model (DOM) is a programming interface for web 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 interact with the page.

A web page is a document that can be either displayed in the browser window or as the HTML source. In both cases, it is the same document but the Document Object Model (DOM) representation allows it to be manipulated. As an object-oriented representation of the web page, it can be modified with a scripting language such as JavaScript.

For example, the DOM specifies that the **querySelectorAll** method in this code snippet must return a list of all the <p> elements in the document:

JS

const paragraphs = document.querySelectorAll("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, any table objects that implement the HTMLTableElement DOM interface for accessing HTML tables, and so forth, are all objects.

The DOM is built using multiple APIs that work together. The core DOM defines the entities describing any 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, and the SVG API adds support for representing SVG documents.

**What is the DOM?**

The Document Object Model (DOM) is a programming interface for web 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 interact with the page.

A web page is a document that can be either displayed in the browser window or as the HTML source. In both cases, it is the same document but the Document Object Model (DOM) representation allows it to be manipulated. As an object-oriented representation of the web page, it can be modified with a scripting language such as JavaScript.

For example, the DOM specifies that the querySelectorAll method in this code snippet must return a list of all the <p> elements in the document:

JS

Copy to Clipboard

const paragraphs = document.querySelectorAll("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, any table objects that implement the HTMLTableElement DOM interface for accessing HTML tables, and so forth, are all objects.

The DOM is built using multiple APIs that work together. The core DOM defines the entities describing any 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, and the SVG API adds support for representing SVG documents.

**Accessing the DOM**

You don't have to do anything special to begin using the DOM. You use the API directly in JavaScript from within what is called a script, a program run by a browser.

When you create a script, whether inline in a <script> element or included in the web page, you can immediately begin using the API for the document or window objects to manipulate the document itself, or any of the various elements in the web page (the descendant elements of the document). Your DOM programming may be something as simple as the following example, which displays a message on the console by using the console.log() function:

HTML

<body onload="console.log('Welcome to my home page!');">

…

</body>

As it is generally not recommended to mix the structure of the page (written in HTML) and manipulation of the DOM (written in JavaScript), the JavaScript parts will be grouped together here, and separated from the HTML.

For example, the following function creates a new h1 element, adds text to that element, and then adds it to the tree for the document:

HTML

<html lang="en">

<head>

<script>

// run this function when the document is loaded

window.onload = () => {

// create a couple of elements in an otherwise empty HTML page

const heading = document.createElement("h1");

const headingText = document.createTextNode("Big Head!");

heading.appendChild(headingText);

document.body.appendChild(heading);

};

</script>

</head>

<body></body>

</html>

**Fundamental data types**

This page 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](https://developer.mozilla.org/en-US/docs/Web/API/Document) | When a member returns an object of type document (e.g., the ownerDocument property of an element returns the document to which it belongs), this object is the root document object itself. The [DOM document Reference](https://developer.mozilla.org/en-US/docs/Web/API/Document) chapter describes the document object. |
| [Node](https://developer.mozilla.org/en-US/docs/Web/API/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](https://developer.mozilla.org/en-US/docs/Web/API/Element) | The element type is based on node. It refers to an element or a node of type element returned by a member of the DOM API. Rather than saying, for example, that the [document.createElement()](https://developer.mozilla.org/en-US/docs/Web/API/Document/createElement) method returns an object reference to a node, we just say that this method returns the element that has just been created in the DOM. element objects implement the DOM Element interface and also the more basic Node interface, both of which are included together in this reference. In an HTML document, elements are further enhanced by the HTML DOM API's [HTMLElement](https://developer.mozilla.org/en-US/docs/Web/API/HTMLElement) interface as well as other interfaces describing capabilities of specific kinds of elements (for instance, [HTMLTableElement](https://developer.mozilla.org/en-US/docs/Web/API/HTMLTableElement) for [<table>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/table) elements). |
| [NodeList](https://developer.mozilla.org/en-US/docs/Web/API/NodeList) | A nodeList is an array of elements, like the kind that is returned by the method [document.querySelectorAll()](https://developer.mozilla.org/en-US/docs/Web/API/Document/querySelectorAll). Items in a nodeList are accessed by index in either of two ways:   * list.item(1) * list[1]   These two are equivalent. In the first, item() is the single method on the nodeList object. The latter uses the typical array syntax to fetch the second item in the list. |
| [Attr](https://developer.mozilla.org/en-US/docs/Web/API/Attr) | When an attribute is returned by a member (e.g., by the createAttribute() 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. |
| [NamedNodeMap](https://developer.mozilla.org/en-US/docs/Web/API/NamedNodeMap) | A namedNodeMap 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 namedNodeMap has an item() method for this purpose, and you can also add and remove items from a namedNodeMap. |

There are also some common terminology considerations to keep in mind. It's common to refer to any [Attr](https://developer.mozilla.org/en-US/docs/Web/API/Attr) node as 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 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 implement 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.

JS

Copy to Clipboard

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.querySelector()](https://developer.mozilla.org/en-US/docs/Web/API/Document/querySelector)
* [document.querySelectorAll()](https://developer.mozilla.org/en-US/docs/Web/API/Document/querySelectorAll)
* [document.createElement()](https://developer.mozilla.org/en-US/docs/Web/API/Document/createElement)
* [Element.innerHTML](https://developer.mozilla.org/en-US/docs/Web/API/Element/innerHTML)
* [Element.setAttribute()](https://developer.mozilla.org/en-US/docs/Web/API/Element/setAttribute)
* [Element.getAttribute()](https://developer.mozilla.org/en-US/docs/Web/API/Element/getAttribute)
* [EventTarget.addEventListener()](https://developer.mozilla.org/en-US/docs/Web/API/EventTarget/addEventListener)
* [HTMLElement.style](https://developer.mozilla.org/en-US/docs/Web/API/HTMLElement/style)
* [Node.appendChild()](https://developer.mozilla.org/en-US/docs/Web/API/Node/appendChild)
* [window.onload](https://developer.mozilla.org/en-US/docs/Web/API/Window/load_event)
* [window.scrollTo()](https://developer.mozilla.org/en-US/docs/Web/API/Window/scrollTo)

Examples

Setting text content

This example uses a <div> element containing a <textarea> and two <button> elements. When the user clicks the first button we set some text in the <textarea>. When the user clicks the second button we clear the text. We use:

1. [Document.querySelector()](https://developer.mozilla.org/en-US/docs/Web/API/Document/querySelector) to access the <textarea> and the button
2. [EventTarget.addEventListener()](https://developer.mozilla.org/en-US/docs/Web/API/EventTarget/addEventListener) to listen for button clicks
3. [Node.textContent](https://developer.mozilla.org/en-US/docs/Web/API/Node/textContent) to set and clear the text.

HTML

<div class="container">

<textarea class="story"></textarea>

<button id="set-text" type="button">Set text content</button>

<button id="clear-text" type="button">Clear text content</button>

</div>

CSS

.container {

display: flex;

gap: 0.5rem;

flex-direction: column;

}

button {

width: 200px;

}

JS

const story = document.body.querySelector(".story");

const setText = document.body.querySelector("#set-text");

setText.addEventListener("click", () => {

story.textContent = "It was a dark and stormy night...";

});

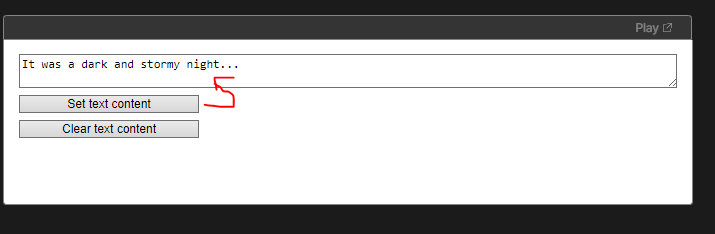
const clearText = document.body.querySelector("#clear-text");

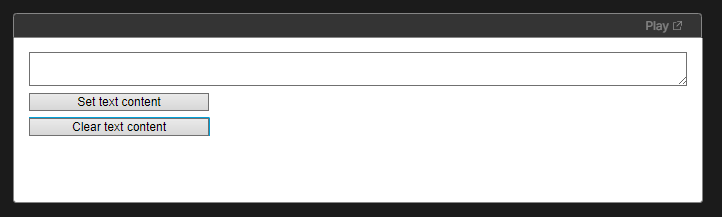
clearText.addEventListener("click", () => {

story.textContent = "";

});

Result





Adding a child element

This example uses a <div> element containing a <div> and two <button> elements. When the user clicks the first button we create a new element and add it as a child of the <div>. When the user clicks the second button we remove the child element. We use:

1. [Document.querySelector()](https://developer.mozilla.org/en-US/docs/Web/API/Document/querySelector) to access the <div> and the buttons
2. [EventTarget.addEventListener()](https://developer.mozilla.org/en-US/docs/Web/API/EventTarget/addEventListener) to listen for button clicks
3. [Document.createElement](https://developer.mozilla.org/en-US/docs/Web/API/Document/createElement) to create the element
4. [Node.appendChild()](https://developer.mozilla.org/en-US/docs/Web/API/Node/appendChild) to add the child
5. [Node.removeChild()](https://developer.mozilla.org/en-US/docs/Web/API/Node/removeChild) to remove the child.

HTML

<div class="container">

<div class="parent">parent</div>

<button id="add-child" type="button">Add a child</button>

<button id="remove-child" type="button">Remove child</button>

</div>

CSS

.container {

display: flex;

gap: 0.5rem;

flex-direction: column;

}

button {

width: 100px;

}

div.parent {

border: 1px solid black;

padding: 5px;

width: 100px;

height: 100px;

}

div.child {

border: 1px solid red;

margin: 10px;

padding: 5px;

width: 80px;

height: 60px;

box-sizing: border-box;

}

JavaScript

const parent = document.body.querySelector(".parent");

const addChild = document.body.querySelector("#add-child");

addChild.addEventListener("click", () => {

// Only add a child if we don't already have one

// in addition to the text node "parent"

if (parent.childNodes.length > 1) {

return;

}

const child = document.createElement("div");

child.classList.add("child");

child.textContent = "child";

parent.appendChild(child);

});

const removeChild = document.body.querySelector("#remove-child");

removeChild.addEventListener("click", () => {

const child = document.body.querySelector(".child");

parent.removeChild(child);

});

Result

