DOCUMENT OBJECT MODEL

The **Document Object Model** (*DOM*) is the data representation of the objects that comprise the structure and content of a document on the web. This guide will introduce the DOM, look at how the DOM represents an HTML document in memory and how to use APIs[1] to create web content and applications.

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. (Introduction to DOM, 2021)

Objects Properties

The browser is an object, and the document it displays is an object too. The browser itself has a long list of objects including: the browser window, the document inside the window, the navigation buttons, the location or URL, and more. These objects are modelled by what's known as the Browser Object Model, or BOM for short.

Because all these things are just objects, we can interact with them using JavaScript the same way we interact with any other object. If you want to know the width of the current viewport, you can simply ask for the window object, and its inner width property**: window.innerWidth**. If you want to open a new tab you use the window **open** method. Window is the top-level object in the BOM, and it has a ton of properties and methods you can use to interact with the browser itself, and what it displays.

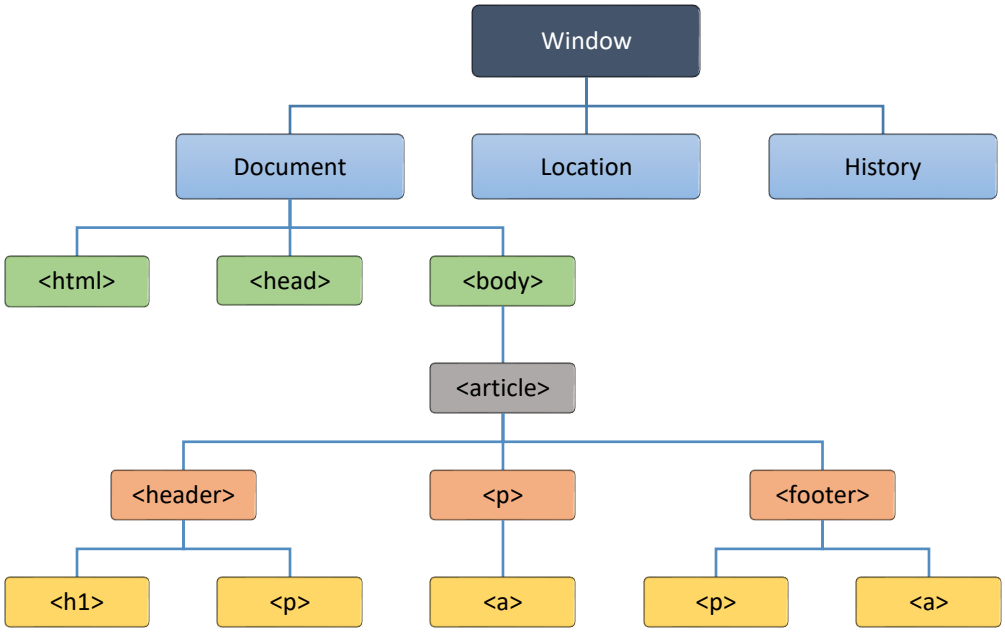
Document is one of the properties in the window object, which contains the current HTML document. Document has its own object model known as DOM. If you have worked with HTML and CSS before, you are already intimately familiar with the DOM, even if you did not know it existed. The Document Object Model is the model of the document that forms the current webpage.

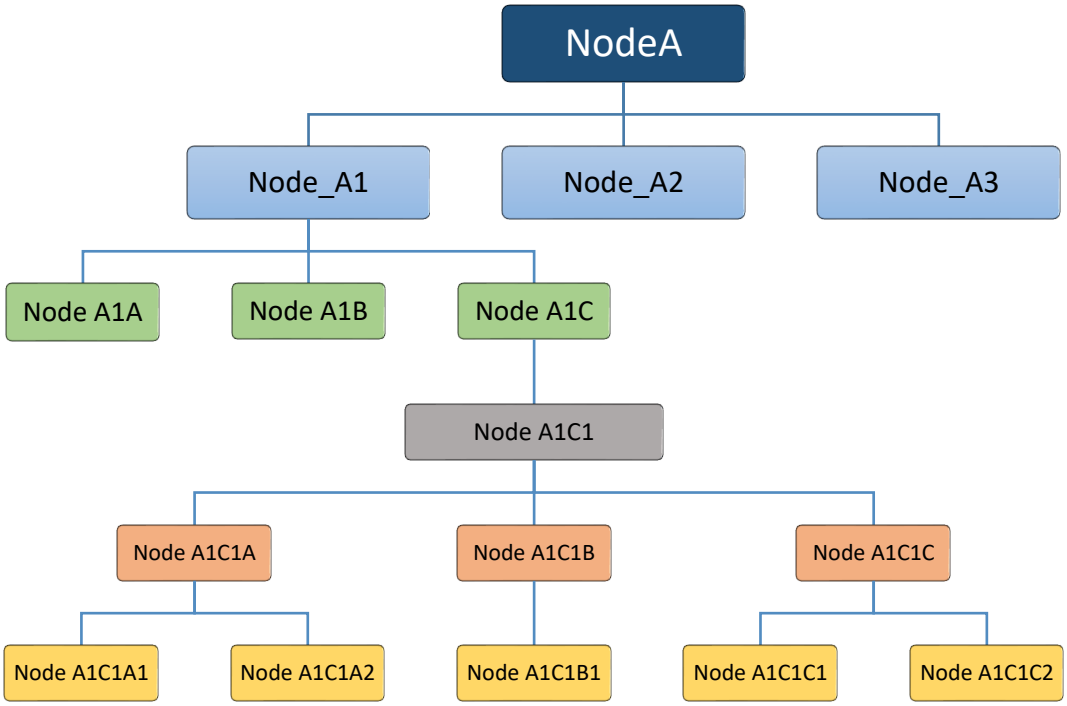
In HTML, every piece of content is wrapped in a beginning and end tag creating an HTML element. Each of these elements is a DOM node, and the browser handles each of them the same way it would handle an object. That's why when you target something with a CSS rule, say all

[1] An **application programming interface** (**API**) is a connection between computers or between computer programs. It is a type of software interface, offering a service to other pieces of software.[1]

anchor tags within the document, that rule is implemented to each of them individually. When you write a CSS rule targeting **a**, you're saying: "find me every **a** node, and apply the following style property settings to it". When a document is loaded in the browser, it is loaded into the document object in the BOM, and a Document Object Model is created for just this document instance. The browser now creates a node tree, modelling the relationships between the different nodes. In a standard HTML document you'll have an HTML object containing two nodes: head, and body. Head holds all the invisible objects like title, link, meta, script, etc., while body holds all the visible nodes in the viewport. *JavaScript sees any webpage as an object tree*

```
<!DOCTYPE html>
<html lang="en" dir="ltr">
  <head>
    <meta charset="utf-8">
    <title>DOM tree</title>
  </head>
  <body>
    <article>
      <header>
        <h1>Learning DOM</h1>
        <p>JavaScript sees any webpage as an object tree</p>
      </header>
      <p>Visit <a href="https://developer.mozilla.org"> Developer Mozilla</a></p>
      <footer>
        <p>Template by prof. Wu @ 2021</p>
        <a href="#">Top of the page</a>
      </footer>
    </article>
  </body>
</html>
```





NodeA.firstChild ➔ Node_A1

NodeA.lastChild ➔ Node_A3

NodeA1.firstChild ➔ Node A1A

NodeA.childNode[0] ➔ Node_A1

NodeA.childNode[1] ➔ Node_A2

NodeA.childNode[3] ➔ Null

Node A1.parentNode ➔Node A

Node A1C1B.parentNode ➔ Node A1C1

Node A2.nextSibling ➔ node A3

Node A2.prevSibling ➔ node A1

Node A3.prevSibling ➔ null

Node A1C1.lastChild.firstChild ➔ Node A1C1C1

How to target elements in the DOM

To get to a node or element, or a group of nodes inside the body, we use methods available for document. Traditionally the methods getElementByID, getElementsByClassName, getElementsByTagName, and getElementsByTagNameNS have been used to do this. getElementByID would return the element with the specified ID. The others would return HTML collections, or node lists of all elements with the same class name, tag, or namespace tag.

These methods work fine, but they are often too specific and a bit clunky to work with. Especially if you are looking for a node inside a node inside a node. More recently, two new catchall methods have come along to solve pretty much all our targeting needs. Query Selector, which returns the first instance that matches the specified selectors, and Query Selector All which returns a node list of all elements that match the specified selectors. These selectors are one or more comma separated CSS selectors.

So you target elements within the document with these methods the same way you would target them using a style rule. That makes Query Selector and its sibling both easy to work with, and incredibly powerful.

```
/* Get the first element matching
specified selector(s): */
document.querySelector(".images");

/* Get all element matching
specified selector(s): */
document.querySelectorAll(".ima
ges");
```

There are still cases where you might want to use one of the old methods. Specifically if you're working with forms, or if you want your code to be unambiguous. But for most uses, **querySelector**, and **querySelectorAll** is the way to go. For a full rundown of these methods, check out the documentation at Mozilla Developer Network. https://developer.mozilla.org/en-US/docs/Web/API/document

Accessing and changing elements

The purpose of targeting an Element within the DOM, using JavaScript, is to do something with it. Like change the text, change an image reference, change a class name, or ID, or maybe the HTML as a whole. Each of these Elements is its own DOM node, so effectively, an object, and each Element has a long list of properties and methods we can use to interact with it. The Mozilla Developer Network page for Element, gives us a full breakdown of all the available properties and methods for Elements. These properties include tag name, so the Elementtag, attributes, ID, class name, inner HTML, and more.

In 2009, JavaScript becomes the programming language for the web. It is one of the three languages for web developing. HTML is used to define the content of web pages, CSS is to specify the layout of web pages, and JS is to program the behavior of web pages.

DOCUMENT METHODS

JavaScript uses dots(.) to separate objects from their properties or methods. Let us see some of the JS methods and dot uses.

`getElementBtId()`

The document method, **`getElementBtId(),`** will go into the document and look for a specific element

```
<p id="one" >Here is some text</p>
    <script>
      document.getElementById('one').style.color ='red';
    </script>
```

`getElementsByTabName()`

The **`getElementsByTagName()`** method returns a collection of all elements in the document with the specified tag name, that I can use the elements for something else later.

```
<p>Here is some text 1</p>
    <p>Here is some text 2</p>
    <p>Here is some text 3</p>
    <script>
      var myText = document.getElementsByTagName('p');
      console.log(myText);
    </script>
```

In order to access the elements in a collection, we can use a for loop.

```
   <p>First Name </p>
  <p>Last Name </p>
  <p>Full Name </p>
  <script type="text/javascript">
var parag = document.getElementsByTagName('p');
for (var i = 0; i<parag.length; i++){
  parag[i].style.color='green';
  alert(`showing paragraph ${i+1}`);
}
  </script>
```

getElementsByClassName()

The **getElementsByClassName()** method returns a collection of all elements in the document with the specified class name

```
<p class="name">First Name </p>
<p class="name">Last Name </p>
<p>Full Name </p>
<script type="text/javascript">
      var n = document.getElementsByClassName('name');
      for (var i = 0; i < n.length; i++) {
        n[i].style.color = 'blue';
      }
</script>
```

You can also get one element:

```
<p class="name">First Name </p>
<p>Last Name </p>
<p>Full Name </p>
<script type="text/javascript">
      var n = document.getElementsByClassName('name');
      n[0].style.color = 'blue';
</script>
```

querySelector();

The **querySelector()** method returns the first element that matches a specified *CSS selector(s)* in the document.

```
<div id="special">
  <p class="n1">First Name</p>
  <p>Last Name</p>
  <p>Full Name</p>
</div>
<script type="text/javascript">
  var myText = document.querySelector('#special .n1');
  myText.style.color = 'red';
</script>
```

querySelectorAll();

The **querySelectorAll()** method returns all elements in the document that matches a specified CSS selector(s), as a static NodeList object.

The NodeList object represents a collection of nodes. The nodes can be accessed by index numbers. The index starts at 0.

```
<div id="special">
  <p class="n1">First Name</p>
  <p>Last Name</p>
  <p>Full Name</p>
</div>
<script type="text/javascript">
  var myText = document.querySelectorAll('.n1');
  for(var i=0; i<myText.length; i++){
    myText[i].style.fontWeight = "bold";
  }
</script>
```

Working with properties

JS has a number of useful properties. The ones that use the most are:

- .style.*property* = "value";
- .innerHTML = "value"
- .className = "value"

innerHTML

Using the innerHTML property can be very powerful. The innerHTML property sets or returns the HTML content (inner HTML) of an element. It can change a complete contend of an element.

```
<div id="special">
  <p class="n1">First Name</p>
  <p>Last Name</p>
  <p>Full Name</p>
</div>

<script type="text/javascript">
var myDiv = document.getElementById('special');
myDiv.innerHTML='New Paragraph using innerHTML';
</script>
```

className

The **className** property sets or returns the class name of an element (the value of an element's class attribute). For this, the class name should have set in `<style>` or CSS file

```
<!DOCTYPE html>
<html>
<head>
<style>
  .colorblue{color:blue;}
</style>
</head>
<body>
  <div class="special">
    <p>Here is some text</p>
```

```
    <p>Here is some more text</p>
  </div>

  <script type="text/javascript">
    var firstPara = document.querySelector('p');
    firstPara.className="colorblue";
</style>
```

.style.property

.style.*property* sets the style property of an element

```
 <div id="special">
    <p class="n1">First Name</p>
    <p>Last Name</p>
    <p>Full Name</p>
  </div>
  <script type="text/javascript">
    var myText = document.querySelectorAll('p');
      myText[0].style.color = "pink";
      myText[2].style.fontSize = "2em";
  </script>
```

setAttribute()

The `setAttribute()` method adds the specified attribute to an element, and gives it the specified value.

If the specified attribute already exists, only the value is set/changed.

```
<body>
<form>
  <label><input type="checkbox">YES</label>
</form>

<script type="text/javascript">
  var myCheckbox = document.querySelector('input');
  myCheckbox.setAttribute('checked', 'checked');
</script>
```

Creating Elements and Text Nodes

It can be easy to add HTML to a page using **`innerHTML`**, as you have already seen, but sometimes you need to actually create an element, put some text in it, and add it to the page. To add a new element to the HTML DOM, you must create the element (element node) first, and then append it to an existing element.


```
<div>
  <p>A paragraph</p>
</div>

<script type="text/javascript">
// step 1: create a new paragraph
   var myPara = document.createElement('p');
// step 2: give it content
   var mySentence = document.createTextNode('This is a text in a new paragraph');
// step 3: add it to the DOM
   myPara.appendChild(mySentence);
// step 3: find the position where the new element will be added
   document.querySelector('div').appendChild(myPara);
</script>
```

Removing Elements

```
<div>
  <p>PARAGRAPH 1</p>
  <p>PARAGRAPH 2</p>
</div>

<script type="text/javascript">
    var myDiv = document.querySelector('div');
    myDiv.removeChild(myDiv.children[0]);
</script>
```

READING

https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Introduction