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.

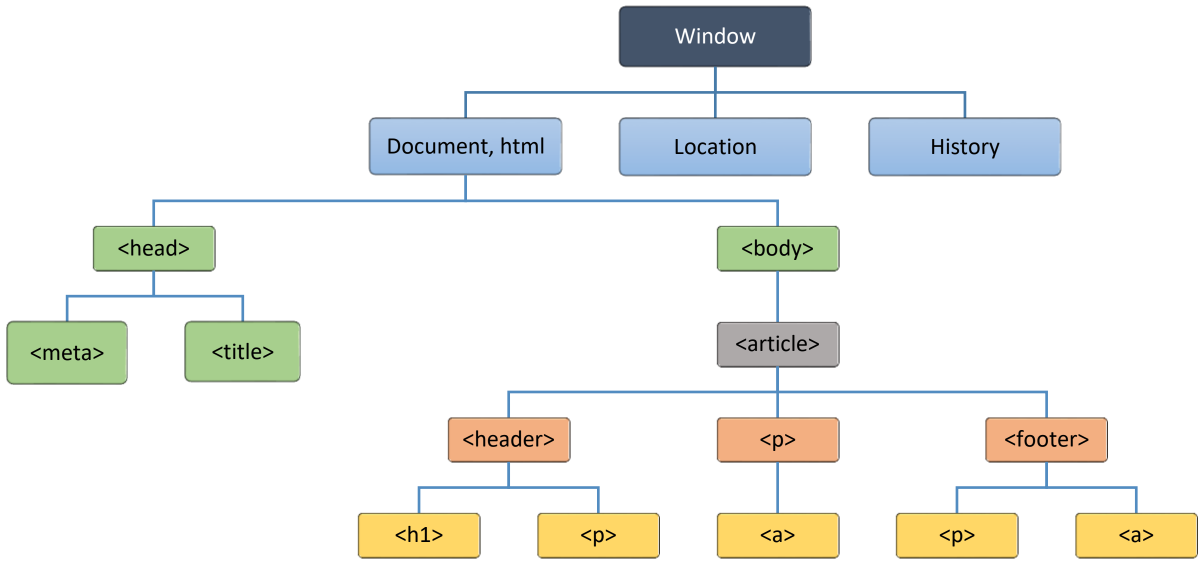
Document is one of the properties in the window object, which contains the current HTML document. Document has its own object model known as DOM. If you have worked with HTML and CSS before, you are already intimately familiar with the DOM, even if you did not know it existed. The Document Object Model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

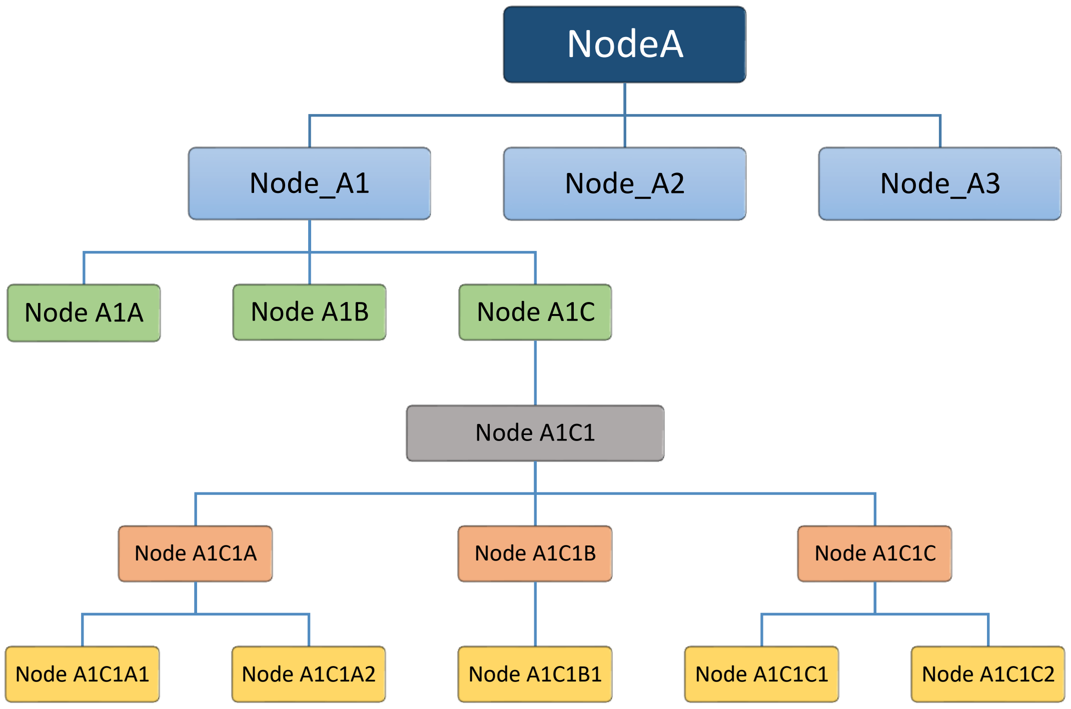
In HTML, every piece of content is wrapped in a beginning and end tag creating an HTML element. Each of these elements is a DOM node, and the browser handles each of them the same way it would handle an object. That's why when you target something with a CSS rule, say all

[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. **`querySelector(),`** which returns the first instance that matches the specified selector, and **`querySelectorAll()`** 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 in the DOM

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.

getElementById()

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

Example 1)

```
<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.

Example 2)

```
<p>Here is some text 1</p>
<p>Here is some text 2</p>
<p>Here is some text 3</p>
<script>
      let 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">
let parag = document.getElementsByTagName('p');
for (let 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

Example 3)

```
  <p class="name">First Name </p>
  <p class="name">Last Name </p>
  <p>Full Name </p>

  <script type="text/javascript">
       let n = document.getElementsByClassName('name');
       for (let 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">
       let 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. It can selects by using any method, by class name, by id name, or by element name. `querySelector()` only return the first matching of the selection.

Example 4)

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

<script type="text/javascript">
   let myText = document.querySelector('#special .n1');
   myText.style.color = 'red';
   let firstP = document.querySelector('p');
   firstP.style.backgroundColor = 'orange';
 </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.

Example 5)

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

<u>Manipulating DOM elements</u>

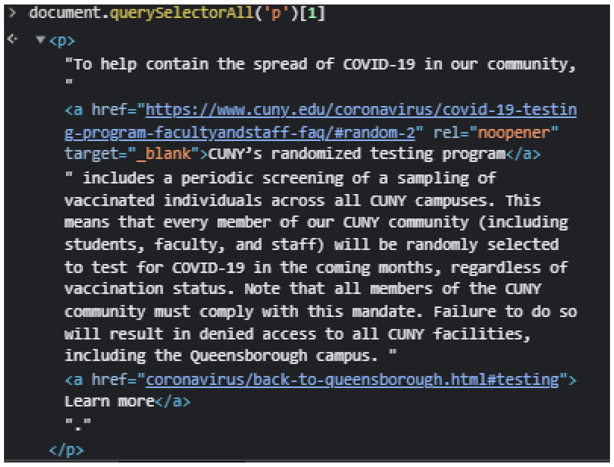
Properties and methods (the important ones)

- `classList`
- `getAttribute()`
- `setAtrribute()`
- `appendChild()`
- `append()`
- `prepend()`
- `removeChild()`
- `remove()`
- `createElement()`
- `innerText()`
- `textContent()`
- `innerHTML`
- `value`
- `parentElement`
- `children`
- `nextSibling`
- `previousSibling`
- `style`

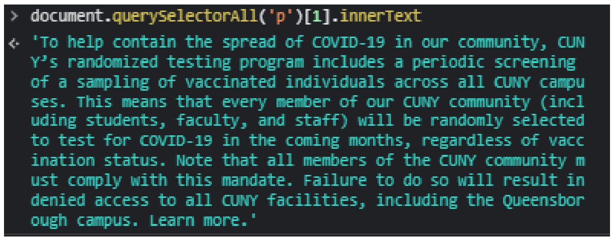
Modifying text content

We can modify the text content of an element by using the properties **`.innerText`, `textContent` ,** and **`innerHTML`.** `textContents` is all text contained by an element and all its children that are for formatting purposes only. `innerText` returns all text contained by an element and all its child elements. `innerHTML` returns all text, including html tags, that is contained by an element.

`.innerText`



```
> document.querySelectorAll('p')[1]
<· ▼<p>
      "To help contain the spread of COVID-19 in our community,
      "
      <a href="https://www.cuny.edu/coronavirus/covid-19-testin
      g-program-facultyandstaff-faq/#random-2" rel="noopener"
      target="_blank">CUNY's randomized testing program</a>
      " includes a periodic screening of a sampling of
      vaccinated individuals across all CUNY campuses. This
      means that every member of our CUNY community (including
      students, faculty, and staff) will be randomly selected
      to test for COVID-19 in the coming months, regardless of
      vaccination status. Note that all members of the CUNY
      community must comply with this mandate. Failure to do so
      will result in denied access to all CUNY facilities,
      including the Queensborough campus. "
      <a href="coronavirus/back-to-queensborough.html#testing">
      Learn more</a>
      "."
    </p>
```



```
> document.querySelectorAll('p')[1].innerText
<· 'To help contain the spread of COVID-19 in our community, CUN
  Y's randomized testing program includes a periodic screening
  of a sampling of vaccinated individuals across all CUNY campu
  ses. This means that every member of our CUNY community (incl
  uding students, faculty, and staff) will be randomly selected
  to test for COVID-19 in the coming months, regardless of vacc
  ination status. Note that all members of the CUNY community m
  ust comply with this mandate. Failure to do so will result in
  denied access to all CUNY facilities, including the Queensbor
  ough campus. Learn more.'
```

`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.

Example 6)

```
  <div id="special">
    <p class="n1">First Name</p>
    <p>Last Name</p>
    <p>Full Name</p>
  </div>
  <script type="text/javascript">
  let myDiv = document.querySelectorAll('p')[1];
      myDiv.innerHTML='New Paragraph using innerHTML <em><b> Hey there</b></em>';
</script>
```

.style.property

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

Example 7)

```
 <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>
```

Changing styles

Styles properties in JS is all camel case. For example, for background-color, in JS will be backgroundColor.

To change the style of an object in JS:

`Object.style.property = 'value'`

Example 8)

```
<div id="container">
   <h1>I &hearts; Trees</h1>
   <img src="https://images.unsplash.com/photo-1596328546171-
77e37b5e8b3d?ixlib=rb-
1.2.1&ixid=eyJhcHBfaWQiOjEyMDd9&auto=format&fit=crop&w=1400&q=80" >
</div>


 <script type="text/javascript">
    document.querySelector('#container').style.textAlign ='center'
    let image = document.querySelector('img');
    image.style.width = '150px'
    image.style.borderRadius ='50%'
  </script>
```

Example 9)

```
   <h1>
         <span>R</span>
         <span>A</span>
         <span>I</span>
         <span>N</span>
         <span>B</span>
         <span>O</span>
         <span>W</span>
     </h1>

 const colors = ['red', 'orange', 'yellow', 'green', 'blue', 'indigo', 'violet'];
 //PLEASE DON'T CHANGE THIS LINE!
 const letters = document.querySelectorAll('span');
 let counter = 0
 for (let eachLetter of letters) {
      eachLetter.style.color = colors[counter]
      counter++  }
```

className

Example 10)

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">
    let firstPara = document.querySelector('p');
    firstPara.className="colorblue";
  </style>
```

`setAttribute()`

The `setAttribute(attribute’s name, attribute’s value)` 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.

Example 11)

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

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

Example 12) Manipulating Attributes: using the DOM elements attributes. Select the image and change its `alt` text to "`image of egg`"

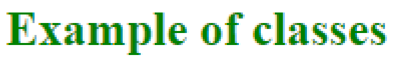
```
<img src="https://devsprouthosting.com/images/egg.jpg" width="200px">
<script type="text/javascript">
   let image = document.querySelector('img');
   image.setAttribute('alt', 'chicken')
</script>
```

Adding classes

Adding a class using `setAttribute` will override the current class

Example 13)

```
<!DOCTYPE html>
<head>
    <title>Forest</title>
      <style media="screen">
      .subtitle{ text-align: center; color: green; }
      .colorSubtitle{color: red; text-align: right; font-family: algerian }
    </style>
</head>
<body>
  <!-- example adding classes -->
  <h2 class=" colorSubtitle ">Example of classes</h2>
<script type="text/javascript">
    // adding classes
    document.querySelector('h2').setAttribute('class','subtitle')
</script>
```

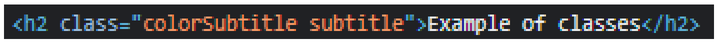


Instead of overwrite the current class, we can use the `classList.add()` to add a class to an element:

```
document.querySelector('h2').classList.add('subtitle')
```

EXAMPLE OF CLASSES

If we check on the console, we can see that `<h2>` now has two classes



```
<h2 class="colorSubtitle subtitle">Example of classes</h2>
```

We can also remove a class by using the **`.classList.remove().`** For example, if we want to remove class colorSubtitle from the `<h2>` element.

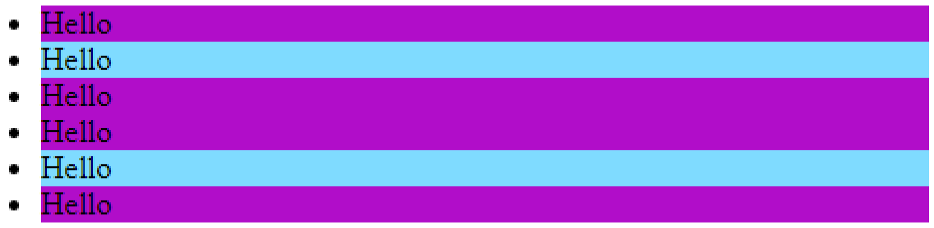
```
document.querySelector('h2').classList.remove('colorSubtitle')
```

We can use `.classList.toggle('className')` to add or remove a class

Example 14) use JS to invert the class in each element `<li>`

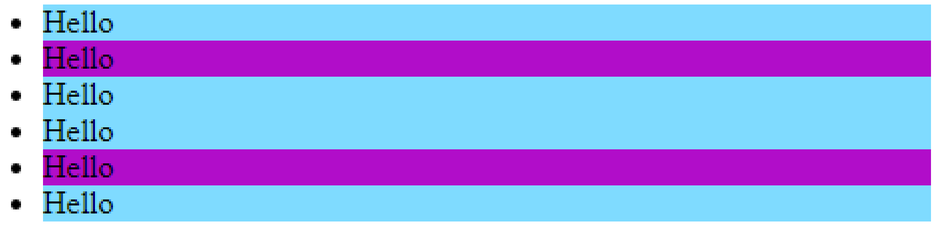
```
<style>
  li { background-color: #B10DC9; }
  .highlight {  background-color: #7FDBFF;}
</style>

<ul>
      <li>Hello</li>
      <li class="highlight">Hello</li>
      <li>Hello</li>
      <li>Hello</li>
      <li class="highlight">Hello</li>
      <li>Hello</li>
  </ul>
```



- Hello
- Hello
- Hello
- Hello
- Hello
- Hello

```
<script>
  const listItem = document.querySelectorAll('li');
      for(let eachLi of listItem){
      eachLi.classList.toggle('highlight')
    }
</script>
```


- Hello
- Hello
- Hello
- Hello
- Hello
- Hello

Parent/sibling/child

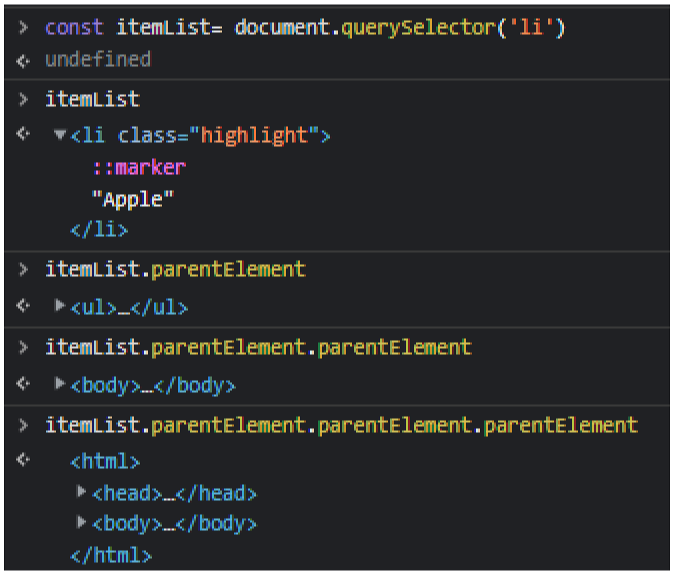
Parent elements can traverse upwards of an element. We can find the parent element by using the property `.parentElement.` Every element can only have one direct parent but can have multiple children

Example 15)

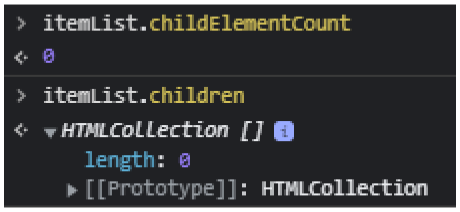
```
<!DOCTYPE html>

<head>
    <title>DOM example</title>
 </head>

<body>
<p>Fruit Shopping List</p>
  <ul>
      <li>Apple</li>
      <li class="highlight">Orange</li>
      <li>Pearl</li>
      <li>Grapes</li>
      <li class="highlight">Strawberries</li>
      <li>Watermelon</li>
  </ul>
```



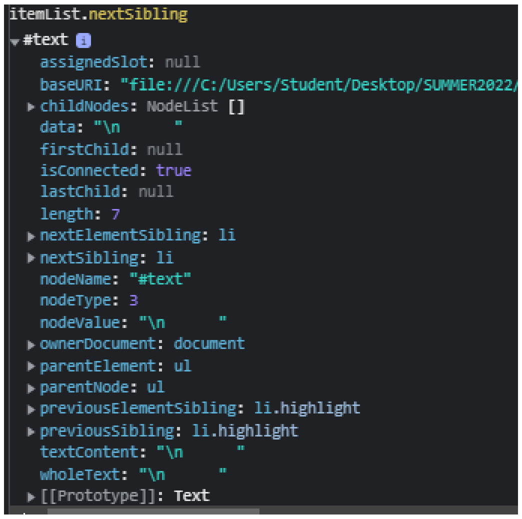
```
> const itemList= document.querySelector('li')
<· undefined
> itemList
<·  ▼<li class="highlight">
       ::marker
       "Apple"
     </li>
> itemList.parentElement
<·  ▶<ul>…</ul>
> itemList.parentElement.parentElement
<·  ▶<body>…</body>
> itemList.parentElement.parentElement.parentElement
<·   <html>
      ▶<head>…</head>
      ▶<body>…</body>
     </html>
```



```
> itemList.childElementCount
<· 0
> itemList.children
<· ▼HTMLCollection []
      length: 0
    ▶[[Prototype]]: HTMLCollection
```

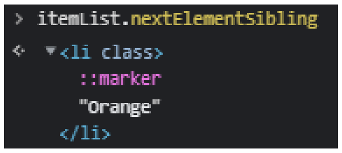
Sibling property returns information of the same type adjacent to the element.

Property `.nextSibling` returns the node text of the next adjacent sibling element:



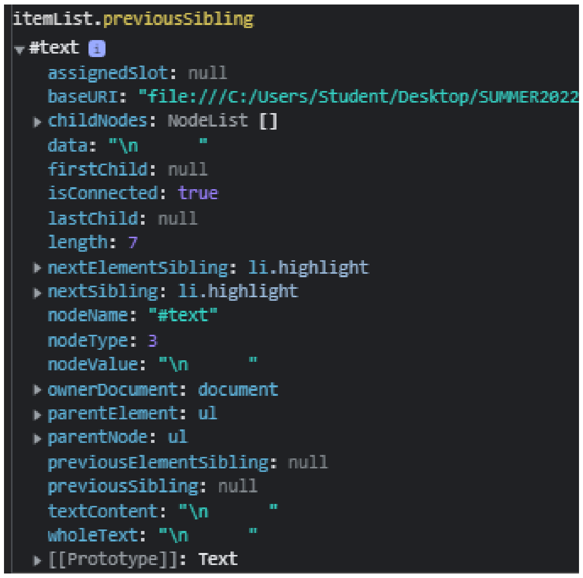
```
itemList.nextSibling
▼#text
    assignedSlot: null
    baseURI: "file:///C:/Users/Student/Desktop/SUMMER2022/
  ▶childNodes: NodeList []
    data: "\n      "
    firstChild: null
    isConnected: true
    lastChild: null
    length: 7
  ▶nextElementSibling: li
  ▶nextSibling: li
    nodeName: "#text"
    nodeType: 3
    nodeValue: "\n      "
  ▶ownerDocument: document
  ▶parentElement: ul
  ▶parentNode: ul
  ▶previousElementSibling: li.highlight
  ▶previousSibling: li.highlight
    textContent: "\n      "
    wholeText: "\n      "
  ▶[[Prototype]]: Text
```

Property `.nextElementSibling` returns the element the content of the next adjacent element of the same type:



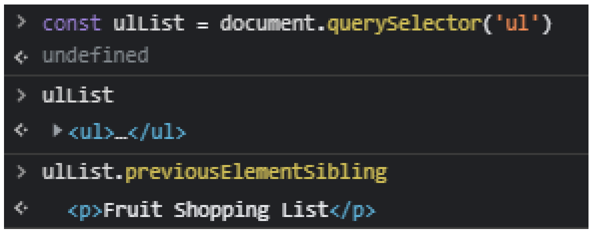
```
> itemList.nextElementSibling
<·  ▼<li class>
       ::marker
       "Orange"
     </li>
```

`previousSibling` property



```
itemList.previousSibling
▾#text i
    assignedSlot: null
    baseURI: "file:///C:/Users/Student/Desktop/SUMMER2022
  ▸ childNodes: NodeList []
    data: "\n      "
    firstChild: null
    isConnected: true
    lastChild: null
    length: 7
  ▸ nextElementSibling: li.highlight
  ▸ nextSibling: li.highlight
    nodeName: "#text"
    nodeType: 3
    nodeValue: "\n      "
  ▸ ownerDocument: document
  ▸ parentElement: ul
  ▸ parentNode: ul
    previousElementSibling: null
    previousSibling: null
    textContent: "\n      "
    wholeText: "\n      "
  ▸ [[Prototype]]: Text
```

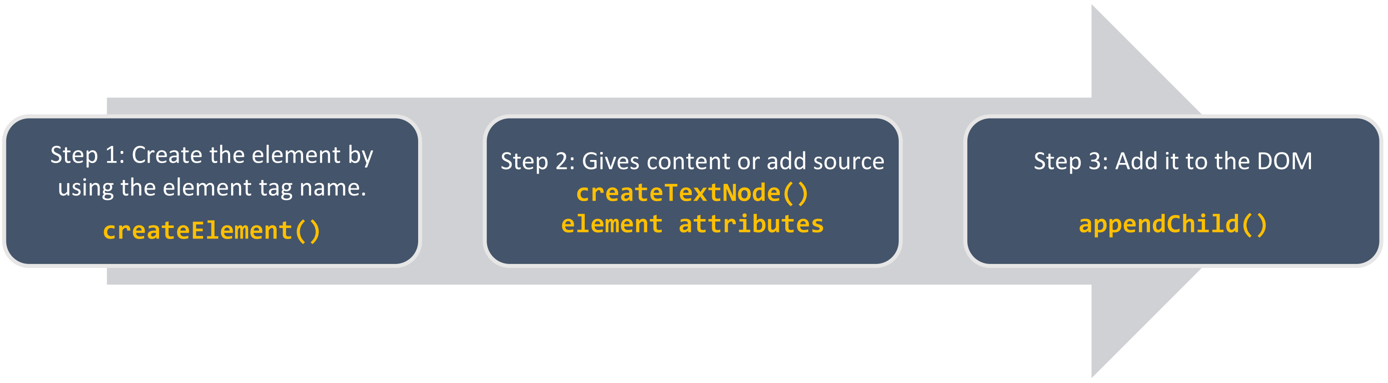
`previousElementSiblings`



```
> const ulList = document.querySelector('ul')
<· undefined
> ulList
<·  ▸<ul>…</ul>
> ulList.previousElementSibling
<·    <p>Fruit Shopping List</p>
```

<u>Creating Elements and Text Nodes</u>

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.


Step 1: Create the element by using the element tag name.
createElement()
Step 2: Gives content or add source
createTextNode()
element attributes
Step 3: Add it to the DOM
appendChild()

Example 16) create an image node and append it to an existing element in the DOM

```
const newImg = document.createElement('img')
// create a content
newImg.src = 'iceland.jpg'
// append newImg to the end of body element
document.body.appendChild(newImg)
newImg.style.width = '20%';
```

Example 17) create a paragraph and append it after the `<div>` element

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

<script type="text/javascript">
// step 1: create a new paragraph
   let myPara = document.createElement('p');
// step 2: give it content
   let 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>
```

Example 18) Create a subtitle using <h2> and append it next to an element with class name .n1

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

<script>
       const newSubtitle = document.createElement('h2')
       newSubtitle.innerHTML ='Appending elements to the DOM';
       document.querySelector('.n1).appendChild(newSubtitle)
</script>
```

Example 19) create 20 buttons and add them inside `<div>` with `id="container"`

```
<div id="container"> </div>

<script>
  for (let button = 1; button<=20; button++){
    let b = document.createElement('button')
    b.innerHTML =`Button ${button}`
    document.body.appendChild(b)
}
</script>
```

We also add element to the end of an element by using the method **`.append().`** We can add element to the beginning to an element by using **`prepend()`**

Note: IE does not support `append()` and `prepend()`

Example 20) append a message and an image stores in variable `newImg` from Example 16 to the `<div>` element with `id="special"`

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

<script>
    document.querySelector('#special').append('HELLO THERE!',newImg)
</script>
```

Example 21) append a paragraph to the beginning of `<div id="special">`

```
document.querySelector('#special').prepend('HELLO THERE!',newImg)
```

We can also append element to the adjacent elements, for example in the middle of elements. This can be done using an `.insertAdjacentElement().` For this, we need to specify a position (`beforebegin, afterbegin, beforeend, afterend`) of the appended element.

Example 22)

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

<script>
    const subtitle = document.createElement('h2')
    subtitle.append("THINGS TO DO IN NYC!")
    let nextSubtitle = document.querySelector('#special')
    nextSubtitle.insertAdjacentElement('afterend',subtitle)
</script>
```

Removing Elements

There are two ways to remove a child from a DOM: `removeChild()` and `remove()`

The original method is `removeChild(),` it has all browser support but it is more complicated to use it. It does not remove the selected element; it removes the child of the selected element.

Example 23)

```
<div>
  <p class = 'p1'>PARAGRAPH 1</p>
  <p>PARAGRAPH 2</p>
</div>

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

Or we can write

```
<script type="text/javascript">
    let par1 = document.querySelector('.p1');
    par1.parentElement.removeChild(par1);
</script>
```

A new method to remove an element in the DOM is called **`remove()`**

Example 24) using the previous example, remove `<p class = 'p1'>` using the `remove()` method

```
let par1 = document.querySelector('.p1');
par1.remove()
```

READING

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