# Working with the DOM (Browser HTML Code) in Javascript

## DOM

Loaded and rendered HTML code or to be precise the object representation of html code which browser create behind the scene. These are objects which will be expose to us as javascript objects. These objects are not limited to javascript, in python there is some documents that can access to the DOM objects.

## Window

Document is a property of window object, actually window is the root html node

## Attributes Vs Properties

**Attributes**: What you add in the HTML tag.

What browser does with tag name is creating such a DOM object base on the html tag and preconfigure some of its properties base on the attributes.

**Properties**: value stored in the object that is created based on the HTML code.

***Note 1***: Properties name and attributes are not the same all the time, for example “id” attribute and property are the same name but “class” attribute is “className” in property.

***Note 2***: Some of properties and attributes have live sync (it means that if we change the property the attribute will change as well), but some doesn’t have for example “value” attribute in the input tag.

If you do want to change attribute in the DOM, there is a method for that, “setAttribute”.

***Note:*** Don't forget that JavaScript is a "hosted language". The browser as host environment exposes this DOM API to your JS code automatically.

## Descendants

Direct or indirect child node or element. In this example <p> tag is child and descendant of <div> but <em> tag is only descendant of <div> and not its child:

<div>

<p>

<em>

Test!

</em>

</p>

</div>

In this example <div> is the ancestor of <p> and <em>.

Even spaces in the HTML are considered as text node in the DOM object created by browser (For example in the above example space between <p> tag and behind it).

const ul = document.querySelector(“ul”);

**ul.children** only shows HTML nodes but **ul.childNodes** shows all of “**ul**” childes including text nodes for spaces.

**ul.closest(“body”)** to select nearest ancestor from ul element. inside the paranthesis you can use css selector.

**ul.previousElementSibling** and **ul.previousSibling**  are the same as children and childNodes for selecting siblings.

***Note***: getting siblings is more performance than querySelector.

## Styling DOM Elements

1-Directly target individual CSS styles

2- Controls styles as inline styles on the element;

3- Style property names are based on CSS properties but have adjusted names (e.g. backgroundColor)

1. Via style property

1-Directly set the CSS classes assigned to the element;

2- Set/Control all classes at once;

3- You can also control the id or other properties.

1. Via className

1- Conveniently add, remove or toggle CSS classes;

2- Fine-grained control over classes that are added;

3- Can be used with className (with care).

1. Via classList

The className method will remove all the classes that element has and then replace it with what we added to it.

The classList has methods like: classList.remove , classList.replace, classList.add, classList.toggle.

## Adding Elements via HTML in code

The innerHTML method will replace all the contents. If you want to add another element to old elements:

const list = document.querySelector (“ul”);

list.innerHTML = list.innerHTML + “<li>Item 4</li>”;

***Note***: The above method has two downsides:

1. It will remove all the elements and then will add it again, so it is not great for performance.
2. If we had an input element by using above method, it will remove what user write in the input.

The better method for that is **insertAdjacentHTML**. This method has some methods:

***beforebegin***

***afterbegin***

***beforeend***

***afterend***

div.insertAdjacentHTML(‘beforeend’,’<p>Something</p>’)

## Adding element via createElement()

const newEle = document.createElement(“li”);

By using above line we have an element inside newEle , but it is not inserted anywhere, So we should do this:

const ul = document.querySelector(“ul”);

ul.appendChild(newEle)

There is another code for adding things in the DOM which is “append” like:

ul.append(…);

By using this method we can add string in the DOM or we can add multiple elements by using comma.

There are some methods for adding elements like : .***lastElementChild.before(newEle)***

***.lastElementChild.after(newEle)***

***.firstElementChild.replaceWith(newEle)***

Another useful method is ***“insertAdjacentElement”*** :

const list = document.querySelector(“ul”);

const secondLi = list.children[1];

const newLi = document.createElement(“li”)

secondLi.insertAdjacentElement(“afterEnd”,newLi);

Like insertAdjacentHTML this method has other methods like **beforeBegin**, …

## Cloning DOM nodes

Cloning a node by the option of deep cloning or not deep:

const newLi2 = newLi.cloneNode(true)

If we pass true it will deep clone (copy all the descendants) and by passing false it will not.

## Live Node lists vs. static Node list

Selecting elements by document.getElementBy… will be live node list and it will change by changing the DOM (it is live). Methods like querySelector is static and has a snapshot of the DOM.

## Removing a node

A way that is not supported by the “internet explorer” is node**.remove()** another way that is supported by all browsers is :

const list = document.querySelector(“ul”);

list.parentElement.removeChild(list);

## Insertion and removal methods

The main difference between these methods is browser support. For example **Safari** doesn’t support before() and after(). Safe methods are : **appendChild()** , **insertAdjacentElement()** , **replaceChild()** , **removeChild()**

## Summary: Insert, Replace, Remove

There are many ways of creating, inserting, replacing and removing DOM elements - here's a summary of the options you have.

For browser support, check the provided links and also the "Browser Support" module you find later in the course.

### Create & Insert

You got two main options: Provide an HTML snippet (e.g. via innerHTML) to a valid HTML snippet and let the browser render it OR create a DOM object in JS code and append/ insert it manually. The latter approach has the advantage of giving you direct access to the DOM object (useful for setting its properties or adding event listeners). The downside is that you have to write more code.

### **Adding HTML Code:**

1. const root = document.getElementById('root-el'); // selects something like <div id="root-el">
2. root.innerHTML = `
3. <div>
4. <h2>Welcome!</h2>
5. <p>This is all create & rendered automatically!</p>
6. </div>
7. `;

**Important**: Any existing content in root is  completely replaced when using innerHTML. If you want to append/ insert HTML code, you can use insertAdjacentHTML instead: <https://developer.mozilla.org/en-US/docs/Web/API/Element/insertAdjacentHTML>

1. const root = document.getElementById('root-el'); // selects something like <div id="root-el">
2. root.insertAdjacentHTML('afterbegin', `
3. <div>
4. <h2>Welcome!</h2>
5. <p>This is all create & rendered automatically!</p>
6. </div>
7. `);

### ****Creating & Inserting DOM Objects Manually:****

1. const someParagraph = document.createElement('p'); // creates a "p" element (i.e. a <p> element)
2. const root = document.getElementById('root-el'); // selects something like <div id="root-el">
3. root.append(someParagraph);

In this example, we create a paragraph and append it to root - append means that it's inserted at the end of root (i.e. inside of it but AFTER all other child nodes it holds).

### ****Insertion Methods:****

**append()** => <https://developer.mozilla.org/en-US/docs/Web/API/ParentNode/append>

Browser support is decent but for IE, **appendChild()** could be preferred => <https://developer.mozilla.org/en-US/docs/Web/API/Node/appendChild>

**prepend()** => <https://developer.mozilla.org/en-US/docs/Web/API/ParentNode/prepend>

Browser support is decent but for IE, **insertBefore()** could be preferred => <https://developer.mozilla.org/en-US/docs/Web/API/Node/insertBefore>

**before()**, **after()** => <https://developer.mozilla.org/en-US/docs/Web/API/ChildNode/before> & <https://developer.mozilla.org/en-US/docs/Web/API/ChildNode/after>

Browser support is okay but IE and Safari don't support it. Consider **insertBefore()** (<https://developer.mozilla.org/en-US/docs/Web/API/Node/insertBefore>) or **insertAdjacentElement()** (<https://developer.mozilla.org/en-US/docs/Web/API/Element/insertAdjacentElement>) as substitutes.

Important (no matter how you insert elements): Whenever you insert elements, you **MOVE the element** to that new place if you already inserted it before. It's **NOT copied** (you can copy an element via someElement.cloneNode(true) though).

### Replace

You can replace elements in the DOM with two main methods:

**replaceWith()** => <https://developer.mozilla.org/en-US/docs/Web/API/ChildNode/replaceWith>

**replaceChild()** => <https://developer.mozilla.org/en-US/docs/Web/API/Node/replaceChild>

replaceWith() is a bit easier to use and has decent browser support - with IE being the exception. To support that as well, consider using replaceChild().

### Remove

You can remove elements with three main methods:

**someElement.innerHTML = ''** => Clears all HTML content of someElement and hence removes any objects rendered in there.

**someElement.remove()** => Removes a single element (someElement) from the DOM (<https://developer.mozilla.org/en-US/docs/Web/API/ChildNode/remove>). Browser support is good, IE again doesn't like it though. Use removeChild (see below) instead.

**someElement.parentNode.removeChild(someElement)** =>  Removes the provided child element (NOT the element on which you call it). Provides broad browser support but of course requires a bit more code (<https://developer.mozilla.org/en-US/docs/Web/API/Node/removeChild>).

### What about Text Nodes?

You can easily create & insert text nodes in one go:

someElement.textContent = 'Hi there!';

This creates and inserts the text node with a content of 'Hi there!'.

**Want to append to existing text?**

Just use:

1. someElement.textContent = someElement.textContent + 'More text!';

***Note :*** we can use methods like querySelector on any selected document :

const addMovieModal = documnet.getElementById("add-modal");

const cancelAddMovieButton = addMovieModal.querySelector(".btn\_\_cancel");

# More on Arrays & Iterables

## What are “Iterable” and “Array-like Objects”

***Iterable*** : Objects that implement the “iterable” protocol and have an @@iterable mothod (i.e. Symbol.iterator). -Symbols are a special type of value in javascript-

For undrestand it easily, Iterables are objects where we can use for-of loops.

***Array.from*** : this method will make array from something that is not iterable.

const moreNumbers = Array.from(“Hi!”);

Another useful example is in DOM selected elements:

const listItems = document.querySelector(“li”);

//listItem is an array-like object that is not iterable

const arrayListItems = Array.from(listItems)