1. **Javascript DOM:-**
2. **WHAT IS DOM?**

DOM is a representation of a web page as a tree-like structure, where each element (e.g., headings, paragraphs, buttons, images) is a "node" in the tree, and they are organized in a hierarchical manner.

1. **Console.log(document) vs console.dir(document) in console.**

**console.log(document):**

console.log() is a method that outputs the specified data to the console. When you use console.log(document), it will display the document object as a summary or a simple representation, typically showing its properties and values.

**console.dir(document):**

console.dir() is another method that displays the properties of a specified JavaScript object in a more detailed and interactive way. When you use console.dir(document), it will display the document object with its properties as a navigable tree-like structure.

1. DOM selecting methods

In JavaScript, selecting methods is a common operation used to access and manipulate elements in the Document Object Model (DOM) or perform various actions on arrays, strings, and other objects. The "selecting" process typically involves retrieving specific elements or data from a larger collection or structure

1. **selectElementById**

This method is used to select a single element from the DOM by its unique ID attribute.

**Syntax:-**

const element = document.getElementById("elementId");

1. **HTML Collection**

an HTMLCollection is a live, ordered collection of DOM elements that share a common tag name or are part of a specific group, such as elements with a particular class name. It represents a list of elements within an HTML document that match a certain criterion.

HTMLCollections are not regular JavaScript arrays, but they are "array-like" objects. This means they have a numeric length property and can be accessed using numeric indices like an array. However, they lack many of the array methods that are available in regular arrays (e.g., forEach, map, reduce). They also do not inherit from the Array prototype.

HTMLCollections are automatically updated when the underlying DOM changes. This means that if you use a DOM method to add, remove, or modify elements that match the criterion for the HTMLCollection, the collection will immediately reflect these changes without requiring any additional update.

Note: we can use incides and iterable properties on the html collection, but cannot use the array methods like map reduce etc.

For example: if we want an element at position 5, we can use:

const image5 = document.getElementsById(“img”);

console.log(image5[4]);

**Note:** When you log an HTMLCollection to the console, the browser's console will typically display a representation of the DOM elements contained within the HTMLCollection. This representation includes a list of the DOM elements along with some of their properties and attributes

1. **getElementsByTagName**

const elementsByTagName = document.getElementsByTagName("tagname");

Note: if we want to select one html element and target one of its property or attribute, we have to use the dot notation

**example:-**

const element = document.getElementById("elementId"); // Replace "elementId" with the actual ID of the element

const propertyValue = element.property; // Replace "property" with the desired property or attribute. example:- (element .width)

1. **getElementsByClassName**

const elementsByTagName = document.getElementsByClassName("classname");

1. **query selector**

querySelector is a powerful DOM method in JavaScript that allows you to select and retrieve the first element that matches a specified CSS selector.

**Syntax:-**

const element = document.querySelector(selector);

**selector:** A string representing the CSS selector to match the element(s) you want to select. It can be an element name (e.g., "div"), a class name (e.g., ".my-class"), an ID (e.g., "#my-element"), or any other valid CSS selector.

Note:- The method returns the first element that matches the specified selector, or null if no matching elements are found.

Here are some examples of using querySelector:

* **Selecting by Tag Name:**

const firstDiv = document.querySelector("div"); // Selects the first <div> element

* **Selecting by Class Name:**

const firstElementWithClass = document.querySelector(".my-class"); // Selects the first element with class "my-class"

* **Selecting by ID:**

const elementById = document.querySelector("#my-element"); // Selects the element with ID "my-element"

* **Selecting Nested Elements:**

const nestedElement = document.querySelector("div .nested"); // Selects the first element with class "nested" inside a <div>

* **Selecting Elements Based on Attributes:**

const inputWithValue = document.querySelector("input[value='hello']"); // Selects the first <input> element with value "hello"

Note: When using attribute selectors with querySelector or querySelectorAll, you need to mention the attribute within square brackets.

The attribute selector syntax is [attribute='value'], where:

**atttribute:** The name of the attribute you want to target.

**'value':** The value you want the attribute to match.

* **Selecting Elements by Pseudo-Classes:**

const firstLink = document.querySelector("a:first-child"); // Selects the first <a> element that is the first child of its parent

1. **query selector all**

querySelectorAll is a DOM method in JavaScript that allows you to select multiple elements that match a specified CSS selector. It returns a NodeList, which is a collection of nodes (elements) that match the given selector.

Syntax:- const elements = document.querySelectorAll(selector);

1. **console.log vs console.dir**

**console.log():** console.log() is used to print a message or any JavaScript value to the console.

It provides a textual representation of the value being logged.

If you log an object or an array using console.log(), it will display the object or array in a collapsed form by default, and you can expand it to view its properties or elements.

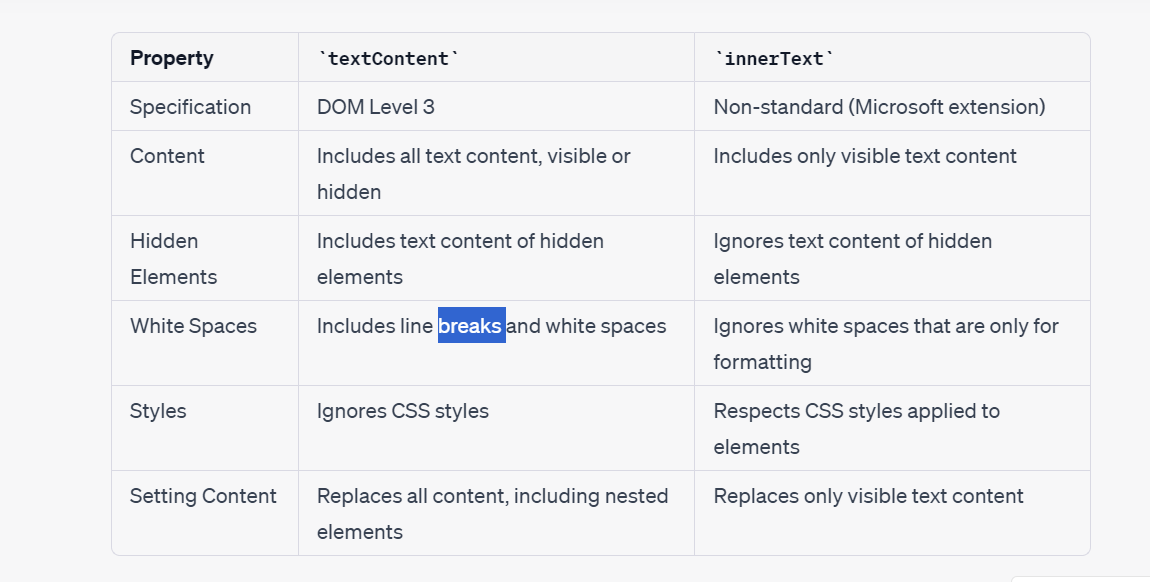
If you log an HTML element using console.log(), it will display the element in a collapsed form, showing its tag name and the value of its most important properties.

**console.dir()** is used to display the properties of an object in an interactive and expandable way.

It shows a detailed representation of the object with all its properties and methods expanded, making it easier to inspect complex objects.

If you log an HTML element using console.dir(), it will display all the properties and methods of the element in an expanded form, giving you a comprehensive view of the element's details.

1. **textContent vs innerText**



1. **innerText vs innerHtml**
2. **select attribute methods**

* **querySelector:** document.querySelector() allows you to select an element based on a CSS selector, including attribute selectors.

It returns the first element that matches the specified selector.

**Example:-**

const element = document.querySelector('input[type="text"]');

* **getAttribute:**

element.getAttribute(attributeName) allows you to retrieve the value of a specific attribute of an element.

It returns the attribute value as a string.

**Example:-**

const imageElement = document.querySelector('img');

const altValue = imageElement.getAttribute('alt');

console.log(altValue); // Output: "Sample Image"

* **setAttribute:**

element.setAttribute(attributeName, attributeValue) is used to set or modify the value of a specific attribute of an element.

If the attribute does not exist, it will be created with the specified name and value.

**Example:-**

const linkElement = document.getElementById('myLink');

linkElement.setAttribute('href', 'https://example.com');

1. **getElementsByTagName**

jdfgjdrjgdrjg

1. **selecting and modifying style attribute in js**

**Syntax:-**

**element.style.width = "50vw";**

Note: When setting CSS properties using JavaScript, you need to use camelCase notation for property names. Additionally, hyphens ("-") should be removed from property names and converted to camelCase.

(background-color)………….. 👎

(backgroundColor)…………… 👍

Note : When using the dot notation to modify an attribute of a particular element, like the style attribute, subsequent modifications(more than 1) will overwrite the previous ones. This is because each modification using the dot notation directly sets a new value to the attribute, and it does not take into account the previous styles.

Therefore, the best way to add multiple multiple modifications on one attribute, adding class is the recommended option.

1. **classList**

The classList property allows you to access and manipulate the class names of an element. It provides methods to add, remove, toggle, and check for class names.

**Here's a brief overview of how you can use classList:**

**add(className):** Adds a class to the element.

**remove(className):** Removes a class from the element.

**toggle(className):** Toggles the presence of a class. If the class is present, it removes it, and if it's not present, it adds it.

**contains(className):** Checks if the element has a specific class and returns true or false.

1. **parentElement property:-**

the parentElement property is a direct property of a DOM element that provides a reference to the parent element of the current element.

Syntax:- childElement.parentElement;

Example:-

<div id="parent">

<p>Child element</p>

</div>

<script>

const childElement = document.querySelector('p');

const parentElement = childElement.parentElement;

console.log(parentElement.tagName); // Output: "DIV"

</script>

Note:- Every element can have 1 single parent element. But one parent element have multiple child elements.

1. **childElementCount and children property**

**childElementCount:**

The childElementCount property returns the number of child elements (elements that are direct descendants) of a given parent element. It excludes any non-element nodes, such as text nodes or comments, that may exist between the child elements.

example:-

<div id="parent">

<p>Child element 1</p>

<p>Child element 2</p>

</div>

<script>

const parentElement = document.getElementById('parent');

const count = parentElement.childElementCount;

console.log(count); // Output: 2

</script>

**children:**

The children property returns a live HTMLCollection of the child elements of the specified parent element. An HTMLCollection is an array-like object that contains all the child elements as elements (nodes), allowing you to access them using numeric indices.

Example:-

<div id="parent">

<p>Child element 1</p>

<p>Child element 2</p>

</div>

<script>

const parentElement = document.getElementById('parent');

const childrenCollection = parentElement.children;

console.log(childrenCollection.length); // Output: 2

// You can also access individual child elements

console.log(childrenCollection[0].textContent); // Output: "Child element 1"

console.log(childrenCollection[1].textContent); // Output: "Child element 2"

</script>

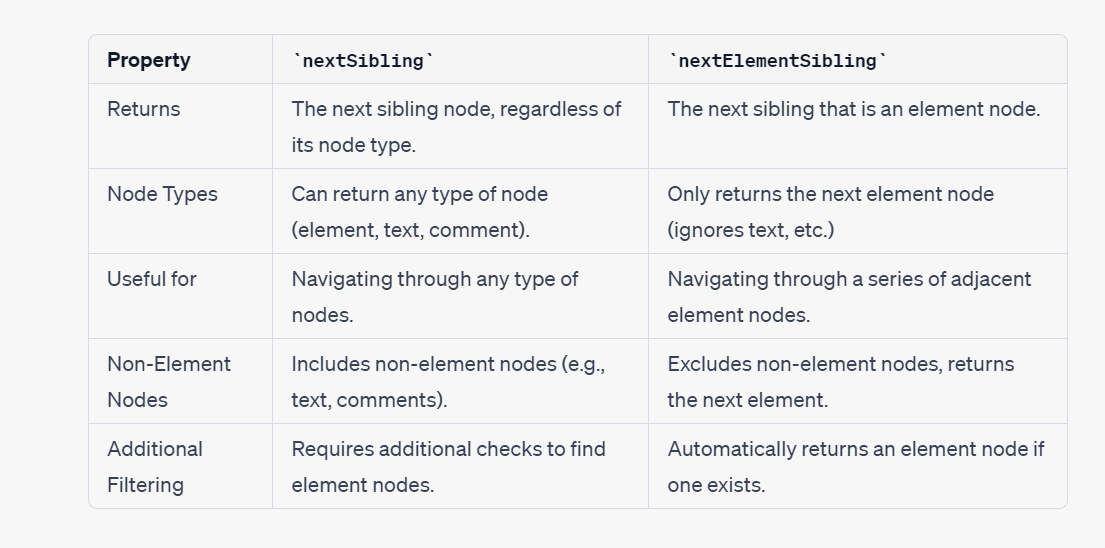
1. **siblings(nextElementSibling and previousElementSibling)**

In the JavaScript DOM, nextElementSibling and previousElementSibling are properties that allow you to access the following sibling and the previous sibling, respectively, of a given element. These properties return the next or previous sibling element nodes, excluding any non-element nodes (such as text nodes or comments).

syntax:-

const nextSibling = secondParagraph.nextElementSibling;

1. **nextSibling vs nextElementSibling**



1. **createElement method**

In the Document Object Model (DOM), the createElement method is used to dynamically create a new HTML element. This method allows you to programmatically generate and add elements to the web page using JavaScript.

**The syntax for createElement is as follows:**

document.createElement(tagName)

**Example:-**

<!DOCTYPE html>

<html>

<head>

<title>createElement Example</title>

</head>

<body>

<script>

// Create a new div element

var newDiv = document.createElement("div");

// Set some properties and content for the new div

newDiv.style.backgroundColor = "lightblue";

newDiv.textContent = "This is a dynamically created div element.";

// Append the new div to an existing element (e.g., the body)

document.body.appendChild(newDiv);

</script>

</body>

</html>

1. **appendChild method:-**

In the Document Object Model (DOM), the appendChild method is used to add a newly created or an existing element as a child to another element. This method allows you to dynamically add elements to the web page or move elements around within the DOM hierarchy.

**The syntax for appendChild is as follows:**

parentElement.appendChild(childElement);

1. **append method**

The Element.append() method inserts a set of Node objects or string objects after the last child of the Element. String objects are inserted as equivalent Text nodes.

**Differences from Node.appendChild():**

* Element.append() allows you to also append string objects, whereas Node.appendChild() only accepts Node objects.
* Element.append() has no return value, whereas Node.appendChild() returns the appended Node object.
* Element.append() can append several nodes and strings, whereas Node.appendChild() can only append one node.

**Syntax:-**

1. **Appending an element**

let div = document.createElement("div");

let p = document.createElement("p");

div.append(p);

console.log(div.childNodes); // NodeList [ <p> ]

1. **Appending a TEXT:**

let div = document.createElement("div");

div.append("Some text");

console.log(div.textContent); // "Some text"

1. **Appending a STRING & TEXT:**

let div = document.createElement("div");

let p = document.createElement("p");

div.append("Some text", p);

console.log(div.childNodes); // NodeList [ #text "Some text", <p> ]

1. **insertAdjacentElement() method**

The insertAdjacentElement() method of the Element interface inserts a given element node at a given position relative to the element it is invoked upon.

**Syntax:-**

insertAdjacentElement(position, element)

[**Parameters**](https://developer.mozilla.org/en-US/docs/Web/API/Element/insertAdjacentElement#parameters)

**position**

A string representing the position relative to the targetElement; must match (case-insensitively) one of the following strings:

* 'beforebegin': Before the targetElement itself.
* 'afterbegin': Just inside the targetElement, before its first child.
* 'beforeend': Just inside the targetElement, after its last child.
* 'afterend': After the targetElement itself.

**element**

The element to be inserted into the tree.

1. **removeChild():-**

The removeChild() method of the Node interface removes a child node from the DOM and returns the removed node.

**Note:-** As long as a reference is kept on the removed child, it still exists in memory, but is no longer part of the DOM. It can still be reused later in the code.

If the return value of removeChild() is not stored, and no other reference is kept, it will be automatically deleted from memory after a short time.

use cases:-

* To remove a specified element when knowing its parent node:

let d = document.getElementById("top");

let d\_nested = document.getElementById("nested");

let throwawayNode = d.removeChild(d\_nested);

* To remove a specified element without having to specify its parent node(not a recommended process)

let node = document.getElementById("nested");

if (node.parentNode) {

node.parentNode.removeChild(node);

}

1. **remove() method:-**

The Element.remove() method removes the element from the DOM.

**Syntax:-**

const element = document.getElementById("div-02");

element.remove(); // Removes the div with the 'div-02' id

1. **a**

a

1. **a**

a

1. **a**

a

1. **a**

a

a

1. **a**

a

a

1. **a**

a

a

1. **a**

a

a

1. **a**

a

a