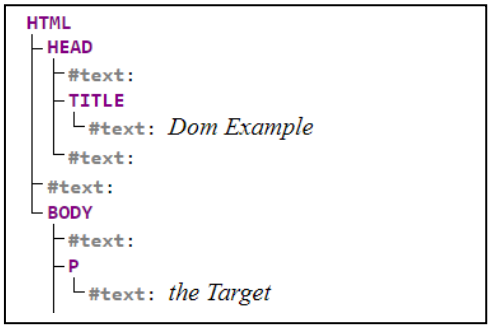
DOM (Document Object Model)

The Document Object Model (DOM) is application programming interface (API) for HTML and XML. It represents the content of html or xml document as tree structure so that programs read, access and change the document structure, style and content.

The Dom is an Object-Oriented representation of the web page, which can be manipulated with a scripting language such as JavaScript.

e.g.

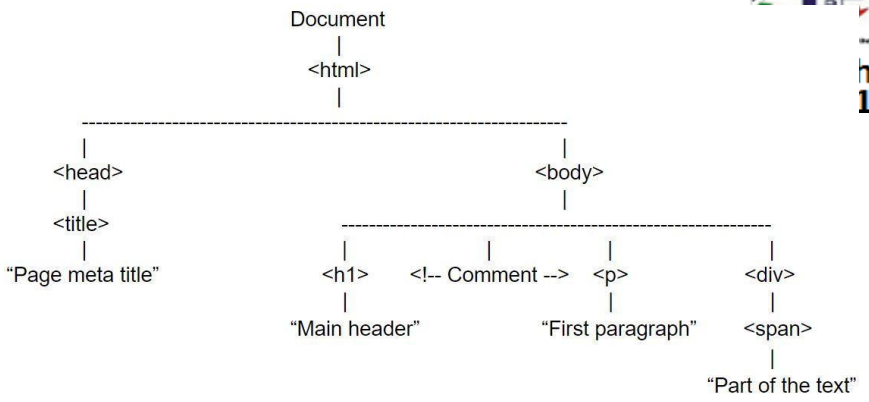
```
<html>
<head>
   <title> Dom Example</title>
</head>
<body>
   <p>the Target </p>
</body>
</html>
```



The DOM tree consists of several types of nodes, including:

- **Document Node:** This represents the entire document and serves as the root of the DOM tree.
- **Element Nodes:** These represent HTML elements, such as , <p>, , etc. Element nodes can have child nodes, such as other elements, text nodes, comments, etc.
- **Attribute Nodes:** These represent attributes of elements. For example, if you have an <a> element with an href attribute, there will be an attribute node representing that href attribute.
- **Text Nodes:** These represent the text content of elements. For example, the text within <p> or elements.
- **Comment Nodes:** These represent comments within the HTML document.

The browser parses an HTML document, it constructs the DOM tree by creating nodes for each element, attribute, text, and comment encountered in the document. The relationships between these nodes reflect the structure of the document.



POWERS OF THE DOM

The powers of the DOM (Document Object Model) lie in its ability to interactively manipulate the structure, style, and content of web pages. Here are some specific powers of the DOM:

Dynamic manipulation

- Dynamically change all the HTML elements in the page
- Dynamically change all the HTML attributes in the page
- Dynamically change all the CSS styles in the page
- Dynamically remove existing HTML elements and attributes
- Dynamically add new HTML elements and attributes
- Dynamically react to all existing HTML events in the page
- Dynamically create new HTML events in the

- **Accessing Elements**: The DOM allows developers to access elements within a webpage using various methods such as getElementById, getElementsByClassName, querySelector, and querySelectorAll. This enables developers to target specific elements and manipulate them as needed.

- **Manipulating Elements:** With the DOM, developers can dynamically create, modify, or remove HTML elements and their attributes. This allows for the dynamic updating of content based on user actions, server responses, or other events.

- **Changing Styles:** The DOM enables developers to dynamically change the style of HTML elements using JavaScript. This includes modifying CSS properties such as color, size, position, and visibility, allowing for dynamic styling based on user interactions or other conditions.

- **Handling Events:** DOM events allow developers to respond to user actions such as clicks, key presses, mouse movements, and form submissions. Event handling in the DOM enables the creation of interactive and responsive web applications.

- **Modifying Content**: The DOM provides methods for modifying the content of HTML elements, including setting text content, appending or removing child nodes, and manipulating HTML attributes. This allows developers to dynamically update the content of web pages based on user input or other factors.

- **Dynamic Loading:** The DOM enables dynamic loading of content into a webpage without requiring a full-page refresh. This includes loading content via AJAX (Asynchronous JavaScript and XML) requests or dynamically injecting content into the DOM based on user interactions.

- **Traversal and Navigation:** The DOM represents HTML documents as a hierarchical tree structure, which allows developers to traverse and navigate the document easily. This includes moving between parent, child, and sibling nodes, as well as navigating up and down the DOM tree.

ACCESSING ELEMENTS

Accessing elements refers to the process of retrieving specific HTML elements from a webpage using JavaScript through the DOM (Document Object Model). This is a fundamental capability in web development, as it allows developers to interactively manipulate and work with the content and structure of a webpage.

- **getElementById:** The getElementById method is a fundamental part of the DOM (Document Object Model) API in JavaScript. As its name suggests, it allows developers to retrieve a reference to a specific HTML element on a webpage by its unique ID attribute. This method is

particularly useful when you need to target and manipulate a single element that has a unique identifier within the document.

e.g.

```
var element = document.getElementById("myElementId");
```

e.g.

```
<body>
  <p id ="elementID"></p>
  <button onclick="test()">hit</button>
  <script>
    function test(){
        const element = window.document.getElementById("elementID");
        element.innerHTML = "Target";
    }
  </script>
</body>
```

- **getElementsByClassName**: The getElementsByClassName method is another essential part of the DOM (Document Object Model) API in JavaScript. As its name implies, it allows developers to retrieve a collection of HTML elements that have a specified class name. This method is useful when you want to target multiple elements that share a common class attribute and perform operations on them collectively.

  Note:
  The getElementsByClassName method returns a **HTMLCollection** containing all elements in the document that have the specified class name. If no elements with the specified class name are found, an empty HTMLCollection is returned.

  e.g.-1

```
var elements = document.getElementsByClassName("myClassName");
for (var i = 0; i < elements.length; i++) {
  //process  elements[i]
}
```

  e.g.-2

```
<body>
  <p class="MERN"></p>
  <p class="MERN"></p>
  <p class="HTML"></p>
  <p class="MERN"></p>
  <p class="HTML"></p>
  <button onclick="test()">hit</button>
  <script>
    function test(){
      var element = window.document.getElementsByClassName("MERN");
      for( i = 0 ; i < element.length ; i++ )
        element[i].innerHTML = "MERN-paragraph-" + (i + 1) ;


      var element = window.document.getElementsByClassName("HTML");
      for( i = 0 ; i < element.length ; i++ )
        element[i].innerHTML = "HTML-paragraph-" + (i + 1) ;
      }
  </script>
```

```
</body>
```

NOTE:
getElementsByClassName () return HTMLCollection

- **getElementsByTagName:** The getElementsByTagName method is a crucial part of the DOM (Document Object Model) API in JavaScript, allowing developers to retrieve a collection of HTML elements that have a specified tag name. This method is particularly useful when you want to target multiple elements of the same type (e.g., <p>, ,) and perform operations on them collectively.
  Note:

  The getElementsByTagName method returns a live HTMLCollection containing all elements in the document that have the specified tag name. If no elements with the specified tag name are found, an empty HTMLCollection is returned.
  e.g.-1

```
var elements = document.getElementsByTagName("p");
for (var i = 0; i < elements.length; i++) {
   //process elements[i]
}
```

  e.g.-2

```
<body>
  <p></p>
  <p></p>
  <p></p>
  <p></p>
  <button onclick="test()">hit</button>
  <script>
    function test(){
      const element = window.document.getElementsByTagName("p");
 // console.log(element) returns HTMLCollection
      for( i = 0 ; i < element.length ; i++ )
        element[i].innerHTML = "paragraph-" + (i + 1) ;
    }
  </script>
</body>
```

NOTE:
getElementsByTagName() return HTMLCollection

querySelector: The querySelector method is a versatile and powerful feature of the DOM (Document Object Model) API in JavaScript. It allows developers to select and retrieve the first HTML element that matches a specified CSS selector. This method is particularly useful when you need to target a single element or the first element that matches a complex selector pattern.

Note:
The querySelector method returns the first **HTML element** in the document that matches the specified CSS selector. If no element matches the selector, it returns null.

e.g.-1

var element = document.querySelector(".myClass"); // Selects the first element with class "myClass"

e.g.-2
var element = document.querySelector("#myId"); // Selects the element with ID "myId"

e.g.-3
var element = document.querySelector("div > p"); // Selects the first <p> element that is a direct child of a

e.g.-4
var element = document.querySelector("input[type='text']"); // Selects the first <input> element with type "text"

e.g. -5

```
<body>
  <p class="MERN"></p>
  <p class="MERN"></p>
  <p class="HTML"></p>
  <p class="MERN"></p>
  <p class="HTML"></p>
  <button onclick="test()">hit</button>
  <script>
    function test(){
      const element = document.querySelectorAll("p.MERN");
     console.log(element);
      for( i = 0 ; i < element.length ; i++ )
        element[i].innerHTML = "MERN-paragraph-" + (i + 1) ;
    }
  </script>
</body>
```

NOTE:
document.querySelectorAll() return NODElist

"children" property

The "children" property specifically refers to the collection of child nodes that belong to a particular parent node in the DOM. These child nodes can include elements, text nodes, comments, etc., that are direct descendants of the parent node.

e.g.

```
<html>
<head>
   <title> Dom Example</title>
</head>
<body>
   <div id="parent">
      <p>Child paragraph 1</p>
      <p>Child paragraph 2</p>
   </div>
   <script>
      const parentElement = document.getElementById("parent");
      const children = parentElement.children;
```

```
    console.log(children); // This will log an HTMLCollection of the child <p> elements
  </script>
</body>
</html>
```

parentNode property: The parentNode property provides a way to access the parent node of a given DOM node

e.g.

```
<html>
<head>
  <title> Dom Example</title>
</head>
<body>
  <div id="parent">
    <p>Child paragraph 1</p>
    <p>Child paragraph 2</p>
  </div>
  <script>
    const childNode = document.querySelector('p'); // Selecting the first <p> element
    const parentNode = childNode.parentNode;
    console.log(parentNode); // This will log the <div> element with the id "parent"

    // Add a class to the parent node
    parentNode.classList.add("highlight");

    // Alternatively, you can directly modify the style
     parentNode.style.backgroundColor = "lightblue";

  </script>
</body>
</html>
```

previousSibling and nextSibling properties

The previousSibling and nextSibling properties are part of the DOM (Document Object Model) API in JavaScript, allowing developers to access the sibling nodes (elements, text nodes, or comments) of a specified node within the DOM tree.

previousSibling: This property returns the previous sibling node of a specified node. It represents the node immediately preceding the specified node in the DOM tree.

e.g.

```
var previousSiblingNode = node.previousSibling;
```

nextSibling: This property returns the next sibling node of a specified node. It represents the node immediately following the specified node in the DOM tree.

e.g.

```
var nextSiblingNode = node.nextSibling;
```

e.g.

```
<html >
<head>
<title>Previous and Next Sibling Demo</title>
```

```
</head>
<body>
  <ul>
    <li id="item1">Coffee (first item)</li><li id="item2">Tea (second item)</li>
  </ul>
<script>
let text = document.getElementById("item2").previousSibling.innerHTML;
console.log(text)
</script>
</body>
</html>
```

firstChild and lastChild properties

firstChild and lastChild are properties of a DOM node that refer to its first and last child nodes, respectively.

- **firstChild:** This property returns the first child node of the specified node. It is useful for accessing and manipulating the first child node within a parent node. If the specified node has no child nodes, firstChild returns null.

- **lastChild:** This property returns the last child node of the specified node. Similar to firstChild, it is used to access and manipulate the last child node within a parent node. If the specified node has no child nodes, lastChild returns null.

e.g.

```
<html>

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

<body>
  <div id="parent">
    <p>First child paragraph</p>
    <p>Middle child paragraph</p>
    <p>Last child paragraph</p>
  </div>
  <script>
    // Select the parent node
    var parentElement = document.getElementById("parent");
    console.log(parentElement)
    // Access the first element child node
    const firstElementChild = parentElement.firstElementChild;

    // Access the last element child node
    const lastElementChild = parentElement.lastElementChild;

    console.log(firstElementChild.textContent); // Output: "First child paragraph"
    console.log(lastElementChild.textContent); // Output: "Last child paragraph"
```

```
  </script>
</body>

</html>
```

MANIPULATING ELEMENTS

Manipulating Elements" refers to the ability to dynamically create, modify, or remove HTML elements and their attributes using JavaScript and the Document Object Model (DOM). The DOM represents the structure of an HTML document as a tree of objects, where each element, attribute, and text node is an object that can be accessed and manipulated programmatically.

Creating Elements: You can dynamically create new HTML elements using JavaScript. This is done using the document.createElement() method, where you specify the type of element you want to create. (e.g., "div", "p", "span").

document.createElement() Method: This method is provided by the Document Object Model (DOM) interface in JavaScript. It allows you to create a new HTML element node programmatically.
Syntax

```
var newElement = document.createElement(tagName);
```

e.g.

```
// Create a new paragraph element
var paragraph = document.createElement("p");
```

appendChild() : The appendChild method is a fundamental feature of the DOM (Document Object Model) API in JavaScript, used to append a new child node to an existing parent node in the DOM tree. This method is commonly used to dynamically add new elements or content to a webpage, allowing developers to update the structure and content of the document dynamically.
e.g.- 1

```
// Get the container element by its ID
var container = document.getElementById("container");
// Create a new paragraph element
var paragraph = document.createElement("p");
paragraph.textContent = "New paragraph";
// Append the paragraph to the container
container.appendChild(paragraph);
```

e.g.-2

```
<html>
<head>
   <title> Dom Example</title>
</head>
<body>
   <div id = "container"></div>
   <script>
      // Get the container element by its ID
      var container = document.getElementById("container");
      // Create a new paragraph element
      var paragraph = document.createElement("p");
```

```
    paragraph.textContent = "New paragraph";
    // Append the paragraph to the container
    container.appendChild(paragraph);
  </script>
</body>
<html>
```

insertBefore()

The insertBefore() method in JavaScript is used to insert a new node before a specified reference node within the DOM (Document Object Model) tree. This method provides a way to dynamically add content to a web page and control its position relative to existing content.

Syntax

```
parentNode.insertBefore(newNode, referenceNode);
```

e.g.

```
<html>
<body>
 <div id="parentElement">
  <p id="referenceElement">This is the reference paragraph.</p>
 </div>
 <button onclick="addParagraphBeforeReference()">Add Paragraph Before Reference</button>
 <script>
  function addParagraphBeforeReference() {
   // Get a reference to the parent element
   var parentElement = document.getElementById("parentElement");

   // Get a reference to the reference element
   var referenceElement = document.getElementById("referenceElement");

   // Create a new paragraph element
   var newParagraph = document.createElement("p");
   newParagraph.textContent = "New paragraph content";

   // Insert the new paragraph before the reference element as a child of the parent element
   parentElement.insertBefore(newParagraph, referenceElement);
  }
 </script>
</body>
</html>
```

removeChild()

The removeChild() method in JavaScript is used to remove a specified child node from its parent node in the DOM (Document Object Model). It allows developers to dynamically delete elements from the DOM tree based on various conditions or user interactions.

Syntax

```
parentNode.removeChild(childNode);
```

e.g.

```
<html>
  <body>
```

```
    <div id="parentElement">
      <p id="childElementToRemove">This is the element to be removed.</p>
    </div>
    <button onclick="removeChildElement()">Remove Child Element</button>
    <script>
      function removeChildElement() {
        // Get a reference to the parent element
        var parentElement = document.getElementById("parentElement");

        // Get a reference to the child element to be removed
        var childElementToRemove = document.getElementById("childElementToRemove");

        // Remove the child element from its parent
        parentElement.removeChild(childElementToRemove);
      }
    </script>
  </body>
</html>
```

replaceChild()

The replaceChild() method in JavaScript is used to replace one child node of a specified parent node with another node. This method is particularly useful when you need to dynamically update the content or structure of a specific portion of your document without affecting other elements.

Syntax

```
parentNode.replaceChild(newNode, oldNode);
```

e.g.

```
<html>
<body>
  <div id="parentElement">
    <p id="oldChildElement">This is the old content.</p>
  </div>

  <button onclick="replaceChildElement()">Replace Child Element</button>
  <script>
    function replaceChildElement() {
      // Get a reference to the parent element
      var parentElement = document.getElementById("parentElement");
      // Get a reference to the old child element to be replaced
      var oldChildElement = document.getElementById("oldChildElement");
      // Create a new paragraph element with new content
      var newParagraph = document.createElement("p");
      newParagraph.textContent = "This is the new content.";
      // Replace the old child element with the new paragraph element
      parentElement.replaceChild(newParagraph, oldChildElement);
    }
  </script>
```

```
</body>
</html>
```

cloneNode()

The cloneNode() method in JavaScript is used to create a shallow copy of a node in the Document Object Model (DOM). This method allows developers to duplicate an existing node, including its attributes, but it does not recursively copy its children.

Syntax:

```
var clonedNode = node.cloneNode(deep);
```

- **node:** This is the node you want to clone.
- **deep (optional):** A boolean value specifying whether to also clone the node's children. If true, it creates a deep copy; if false or omitted, it creates a shallow copy. The default value is false.

e.g.

```
<html>
<body>
  <div id="originalElement">
    <p>This is the original content.</p>
  </div>
  <button onclick="cloneAndInsert()">Clone and Insert</button>
  <script>
    function cloneAndInsert() {
      // Get a reference to the original element
      var originalElement = document.getElementById("originalElement");
      // Clone the original element (shallow copy)
      var clonedElement = originalElement.cloneNode(true); // Passing true creates a deep copy with children.

      // Modify the cloned element (optional)
      clonedElement.querySelector("p").textContent = "This is the cloned content.";
      // Insert the cloned element after the original element
      originalElement.parentNode.insertBefore(clonedElement, originalElement.nextSibling);
    }
  </script>
</body>
</html>
```

insertAdjacentHTML()

The insertAdjacentHTML method is a powerful feature of the DOM (Document Object Model) API in JavaScript, allowing developers to insert HTML markup at a specified position relative to an element. This method provides fine-grained control over where new HTML content is inserted in relation to existing content within an element.

Syntax

```
element.insertAdjacentHTML(position, html);
```

e.g. -1

```
// Get the reference to an element
var element = document.getElementById("myElement");

// Insert HTML content before the element
```

```
element.insertAdjacentHTML("beforebegin", "<div>New content before the element</div>");

// Insert HTML content as the first child of the element
element.insertAdjacentHTML("afterbegin", "<span>New content as the first child</span>");

// Insert HTML content as the last child of the element
element.insertAdjacentHTML("beforeend", "<p>New content as the last child</p>");

// Insert HTML content after the element
element.insertAdjacentHTML("afterend", "<div>New content after the element</div>");
```

e.g.-2

```
<html>
<head>
  <title> Dom Example</title>
</head>
<body>
  <div id="myElement"></div>
  <script>
    // Get the reference to an element
    var element = document.getElementById("myElement");
    // Insert HTML content before the element
    element.insertAdjacentHTML("beforebegin", "<div>New content before the element</div>");

    // Insert HTML content as the first child of the element
    element.insertAdjacentHTML("afterbegin", "<span>New content as the first child</span>");

    // Insert HTML content as the last child of the element
    element.insertAdjacentHTML("beforeend", "<p>New content as the last child</p>");

    // Insert HTML content after the element
    element.insertAdjacentHTML("afterend", "<div>New content after the element</div>");
  </script>
</body>
</html>
```

Modifying Element Attributes: Once you have created an element, you can modify its attributes (such as id, class, src, href, etc.) using JavaScript. This is done by directly accessing the element's properties like element.id, element.className, etc., or by using methods like setAttribute().

- **Accessing Element Properties:** Every HTML element in the DOM has various properties that define its behavior and appearance. These properties can be accessed and modified directly through JavaScript.
- **Directly Accessing Properties:** You can directly access an element's properties by referencing them through the element object. For instance:
    - **element.id:** This property represents the "id" attribute of the element. You can access or modify it directly.

- **element.className:** This property represents the "class" attribute of the element. You can access or modify it directly as well.
- Other properties like src, href, title, innerText, etc., can also be accessed and modified directly.

e.g.

```
<html>
<body>
  <img id="myElementId" src="./images/image1.jpg" onclick="change()">
  <script>
    function change() {
      // Accessing and modifying element properties directly
      var myElement = document.getElementById("myElementId");
      myElement.src = "./images/image2.jpg";
    }
  </script>
</body>
</html>
```

- **setAttribute() Method:** The setAttribute() method in JavaScript is used to dynamically set or modify attributes of HTML elements within the Document Object Model (DOM). The primary purpose of setAttribute() is to dynamically change attributes of HTML elements based on certain conditions or user interactions in a web application.

Syntax:

```
element.setAttribute(attributeName, attributeValue);
```

e.g.

```
<html>
<body>
  <img id="myElementId" src="./images/image1.jpg" onclick="change()">
  <script>
    function change() {
      /// Assuming there's an existing element with id "myElement"
      var myElement = document.getElementById("myElementId");

      // Setting attributes using setAttribute()
      myElement.setAttribute("class", "newClass");
      myElement.setAttribute("title", "New Title");
      myElement.setAttribute("src", "./images/image2.jpg");
      myElement.setAttribute("data-custom", "123");
    }
  </script>
</body>
</html>
```

Benefits of setAttribute(): Using setAttribute() provides flexibility because you can dynamically set any attribute, even if it doesn't have a corresponding property. It's particularly useful when dealing with custom attributes or attributes that JavaScript doesn't provide direct access to.

- **The getAttribute() method** in JavaScript is used to retrieve the value of a specific attribute on an HTML element. It allows developers to access the values of both standard and custom attributes of HTML elements within the Document Object Model (DOM).

e.g.

```
<html>
<body>
  <img id="myElementId" src="./images/image1.jpg" onclick="change()">
  <script>
    function change() {
      /// Assuming there's an existing element with id "myElement"
      var myElement = document.getElementById("myElementId");

      // Setting attributes using setAttribute()
      myElement.setAttribute("class", "newClass");
      myElement.setAttribute("title", "New Title");
      myElement.setAttribute("src", "./images/image2.jpg");
      myElement.setAttribute("data-custom", "123");
      x = myElement.getAttribute("data-custom")
      console.log(x)
    }
  </script>
</body>
</html>
```

Modifying Element Content: Changing content in the DOM allows developers to update text, HTML, or other content within HTML elements dynamically using JavaScript. This capability is crucial for creating dynamic and interactive web applications where content needs to be updated based on user actions, server responses, or other events.

- **Text Content:** To change the text content of an element, you can use the textContent property. This property represents the text content of a node and its descendants as plain text.

  e.g. - 1

```
// Get the element by its ID
var element = document.getElementById("myElement");

// Change the text content
element.textContent = "New text content";
```

  e.g.-2

```
<html>
<head>
  <title> Dom Example</title>
</head>
<body>
  <p id = "myElement"></p>

  <script>
    var element = document.getElementById("myElement");
    element.textContent = "New text content";
  </script>
</body>
</html>
```

- **Inner HTML:** To change the HTML content of an element, you can use the innerHTML property. This property represents the HTML content inside an element, including any HTML tags.
  e.g. - 1

```
// Get the element by its ID
var element = document.getElementById("myElement");

// Change the inner HTML
element.innerHTML = "<strong>New</strong> HTML content";
```

  e.g.-2

```
<html>
<head>
  <title> Dom Example</title>
</head>
<body>
  <p id="myElement"></p>
  <script>
    var element = document.getElementById("myElement");
    element.innerHTML = "<strong>New</strong> HTML content";
  </script>
</body>
</html>
```

Note:

Be cautious when using innerHTML, as it can potentially introduce security vulnerabilities if the content being inserted is not properly sanitized.

Modifying Styles: Once you have a reference to the element, you can modify its styles by accessing the style property. This property provides access to the inline CSS styles of the element, allowing you to directly set or modify specific CSS properties. (**The style property is an object that represents all the CSS properties and their corresponding values defined in the element's style attribute).**

Syntax:

```
var element = document.getElementById("myElement");
var styleObject = element.style;
```

e.g.-1

```
// Change the color of the element's text
element.style.color = "red";

// Modify the font size
element.style.fontSize = "20px";

// Set the background color
element.style.backgroundColor = "lightblue";

// Set the width , height & padding
element.style.width = "200px";
element.style.height = "50%";
```

Important Considerations:

- When accessing the style property, you are only modifying inline styles. If the CSS property is defined externally (in a stylesheet), modifying the style property won't have any effect.
- Using the style property directly adds inline styles to the element. This can override existing styles and affect specificity, so use it judiciously.
- If you need to apply complex or reusable styles, it's often better to define them in a separate CSS stylesheet and then **add or remove CSS classes using the classList property or by modifying the className attribute.**

element.style.padding = "10px";

e.g.-2

```
<html>
<body>
<div id="myDiv">This is a div element.</div>
<button onclick="changeStyles()">Change Styles</button>
<script>
function changeStyles() {
  // Get a reference to the element
  var element = document.getElementById("myDiv");
  // Modify styles
  element.style.color = "blue";         // Change text color
  element.style.fontSize = "24px";      // Change font size
  element.style.fontWeight = "bold";    // Make text bold
  element.style.padding = "10px";       // Add padding
  element.style.border = "2px solid red";// Add border
  element.style.backgroundColor = "lightgray"; // Change background color
}
</script>
</body>
</html>
```

- **classList property :** The **classList property** is a useful feature of the DOM (Document Object Model) in JavaScript that provides an interface to access and manipulate the list of CSS classes of an HTML element. It allows you to easily add, remove, toggle, and check for the presence of CSS classes on an element.
    - **Adding a Class:** You can add a CSS class to an element using the add() method. If the class already exists, it will not be duplicated.
      e.g.

      ```
      element.classList.add("newClass");
      ```

    - **Removing a Class:** You can remove a CSS class from an element using the remove() method.
      e.g.

      ```
      element.classList.remove("oldClass");
      ```

    - **Toggling a Class:** You can toggle the presence of a CSS class on an element using the toggle() method. If the class is present, it will be removed; if it's not present, it will be added.
    - **e.**g.
        - element.classList.toggle("active");

- **Checking for the Presence of a Class:** You can check whether an element has a particular CSS class using the contains() method. It returns true if the class is present, and false otherwise.
  e.g.

  ```
  if (element.classList.contains("highlight")) {
    // Do something
  }
  ```

e.g.

```
<html >
<head>
<title>classList Example</title>
<style>
  .highlight {
    background-color: yellow;
  }
</style>
</head>
<body>

<div id="myElement">This is a div element.</div>
<button onclick="addHighlight()">add class - Highlight</button>
<button onclick="removeHighlight()">remove class -highlight </button>
<button onclick="toggleHighlight()">toggle class - highlight </button>

<script>
function addHighlight() {
  var element = document.getElementById("myElement");
  element.classList.add("highlight");
}

function removeHighlight() {
  var element = document.getElementById("myElement");
  element.classList.remove("highlight");
}

function toggleHighlight() {
  var element = document.getElementById("myElement");
  element.classList.toggle("highlight");
}
</script>
</body>
</html>
```

Handling Events: DOM events allow developers to respond to user actions such as clicks, key presses, mouse movements, and form submissions. Event handling in the DOM enables the creation of interactive and responsive web applications.

Handling events in the DOM refers to the process of writing JavaScript code to respond to various actions or interactions that occur within a web page. These actions, known as events, can include user interactions such as clicks, mouse movements, key presses, form submissions, and more. By handling events, developers can create interactive and responsive web applications that react to user input in real-time.

- **Event Listener Registration:** To respond to events, developers typically register event listeners on specific elements in the DOM. Event listeners are JavaScript functions that are triggered when a particular event occurs on an element. You can attach event listeners using various methods such as addEventListener or by assigning event handler attributes directly in HTML.

e.g.

```
// Example using addEventListener
element.addEventListener('click', handleClick);
// Example using event handler attribute in HTML
<button onclick="handleClick()">Click me</button>
```

Event Handling Function:

The event handling function, also known as the event handler, is the JavaScript function that gets executed when the event occurs. This function receives an event object as a parameter, which contains information about the **event** that occurred, such as the target element, mouse coordinates, key code, etc.

e.g.

```
function handleClick(event) {
   // Handle the click event
   console.log('Button clicked!');
   console.log('Event target:', event.target);
}
```

e.g.

```
<html>
<body>
 <button id="myButton">Click me</button>
 <script>
  // Get a reference to the button element
  var button = document.getElementById("myButton");
  // Define an event listener function
  function handleClick(event) {
  alert("Button clicked!");
  }
  // Register the event listener for the click event on the button
  button.addEventListener("click", handleClick);
 </script>
</body>
</html>
```

Note

- **don't use the "on" prefix for the event; use "click"** instead of "onclick".
- You can easily remove an event listener by using the removeEventListener() method.

event.target : the event.target property refers to the DOM element that triggered the event. This property is particularly useful when handling events on elements with nested structure or when an event propagates through multiple elements.

e.g.-2

```
<html>

<body>
 <button id="myButton1">Click me</button>
 <button id="myButton2">Click me</button>
 <button id="myButton3">Click me</button>
 <script>
  // Get a reference to the button element
  var button1 = document.getElementById("myButton1");
  var button2 = document.getElementById("myButton2");
  var button3 = document.getElementById("myButton3");
  // Define an event listener function

  function handleClick(event) {
  // Handle the click event
  console.log('Button clicked!');
  console.log('Event target:', event.target);
  x = event.target
  x.style.color = "red"
 }
```

```
  // Register the event listener for the click event on the button
  button1.addEventListener("click", handleClick);
  button2.addEventListener("click", handleClick);
  button3.addEventListener("click", handleClick);
 </script>
</body>

</html>
```

Preventing Default Behavior: In some cases, you may want to prevent the default behavior associated with certain events, such as preventing a form from being submitted or preventing a link from navigating to a new page. You can do this by calling the preventDefault() method on the event object within your event handling function.

Page Reload or Navigation:

Upon receiving the response from the server, the browser may perform a page reload or navigate to a new page, depending on the server's response and the browser's configuration.

```
function handleSubmit(event) {
 event.preventDefault();
   // Perform custom form submission logic
}
```

e.g.

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<title>Preventing Default Behavior Example</title>
</head>
<body>

<form id="myForm">
 <label for="textInput">Enter text:</label>
 <input type="text" id="textInput" name="textInput">
 <button type="submit">Submit</button>
</form>

<script>
// Get a reference to the form element
var form = document.getElementById("myForm");

// Define an event listener function for form submission
function handleSubmit(event) {
   // Prevent the default form submission behavior
   event.preventDefault();

   // Perform custom form submission logic (in this case, just log the submitted text)
   var textInput = document.getElementById("textInput").value;
   console.log("Submitted text:", textInput);
}

// Register the event listener for the form submission event
form.addEventListener("submit", handleSubmit);
```

```
</script>

</body>
</html>
```

ACCESSING COLLECTIONS

Accessing collections in the context of the Document Object Model (DOM) refers to retrieving groups of DOM elements that share common characteristics or belong to specific categories. These collections provide convenient ways to access multiple elements at once, enabling developers to perform bulk operations or iterate over elements for various purposes.

- **document.forms** is a property of the Document Object Model (DOM) that provides access to a collection of all <form> elements within the document. It allows developers to access and manipulate forms and their elements programmatically using JavaScript.
    - **Accessing Forms:** You can access individual form elements within the collection using numeric indices or by using their **name** or **id** attributes as properties of the **document.forms** object.

      e.g.

```
// Accessing the first form element in the document
var firstForm = document.forms[0];

// Accessing a form element by its name
var myForm = document.forms["myFormName"];

// Accessing a form element by its id
var anotherForm = document.forms["myFormId"];
```

e.g.

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<title>Document.forms Example</title>
</head>
<body>

<!-- Example Form 1 -->
<form name="loginForm">
  <label for="username">Username:</label>
  <input type="text" id="username" name="username">
  <label for="password">Password:</label>
  <input type="password" id="password" name="password">
  <button type="submit">Login</button>
</form>

<!-- Example Form 2 -->
<form name="registrationForm">
  <label for="fullname">Full Name:</label>
  <input type="text" id="fullname" name="fullname">
  <label for="email">Email:</label>
```

```
  <input type="email" id="email" name="email">
  <button type="submit">Register</button>
</form>

<script>
// Accessing and working with the first form
// var loginForm = document.forms[0];
var loginForm = document.forms["loginForm"];
loginForm.addEventListener("submit", function(event) {
event.preventDefault(); // Prevent form submission for demonstration
var username = loginForm.elements["username"].value;
  var password = loginForm.elements["password"].value;
  console.log("Username:", username);
  console.log("Password:", password);
});

// Accessing and working with the second form
// var loginForm = document.forms[1];
var registrationForm = document.forms["registrationForm"];
registrationForm.addEventListener("submit", function(event) {
event.preventDefault(); // Prevent form submission for demonstration
var fullname = registrationForm.elements["fullname"].value;
  var email = registrationForm.elements["email"].value;
  console.log("Full Name:", fullname);
  console.log("Email:", email);
});
</script>
</body>
</html>
```

- **document.anchors** property is a property of the Document Object Model (DOM) that provides access to a collection of all anchor elements (<a>) within the document that have a name attribute. These anchor elements are often referred to as "named anchors."

```
// Accessing the first anchor element in the document
var firstAnchor = document.anchors[0];

// Iterating over the anchors collection
for (var i = 0; i < document.anchors.length; i++) {
   var anchor = document.anchors[i];
   // Do something with each anchor element
}
```

- **document.images** property is a property of the Document Object Model (DOM) that provides access to a collection of all <img> elements within the document. This collection allows developers to programmatically interact with and manipulate images present in the document using JavaScript.

```
// Accessing the first image element in the document
var firstImage = document.images[0];

// Iterating over the images collection
for (var i = 0; i < document.images.length; i++) {
```

```
    var image = document.images[i];
    // Do something with each image element
}
```

- **document.body** property is a property of the Document Object Model (DOM) that provides direct access to the <body> element of an HTML document. This property allows developers to programmatically interact with and manipulate the content within the <body> tag using JavaScript.

```
// Accessing the body element of the document
var bodyElement = document.body;

// Manipulating the body element
bodyElement.style.backgroundColor = "lightblue"; // Changing background color
bodyElement.innerHTML = "<h1>Hello, World!</h1>"; // Changing inner HTML content
e.g.
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<title>Document.body Example</title>
<style>
  body {
    font-family: Arial, sans-serif;
    background-color: #f0f0f0;
    padding: 20px;
  }
</style>
</head>
<body>

<h1>Hello, World!</h1>
<p>This is a simple example demonstrating the usage of <code>document.body</code>.</p>
<button onclick="changeBodyContent()">Change Body Content</button>

<script>
// Function to change body content and style
function changeBodyContent() {
  // Accessing the body element
  var bodyElement = document.body;

  // Changing background color
  bodyElement.style.backgroundColor = "#c0c0c0";

  // Changing inner HTML content
  bodyElement.innerHTML = `
    <h1>New Content</h1>
    <p>This is the updated content of the body element.</p>
    <button onclick="resetBodyContent()">Reset Body Content</button>
  `;
}
```

```
// Function to reset body content and style
function resetBodyContent() {
  // Accessing the body element
  var bodyElement = document.body;

  // Resetting background color
  bodyElement.style.backgroundColor = "#f0f0f0";

  // Resetting inner HTML content
  bodyElement.innerHTML = `
    <h1>Hello, World!</h1>
    <p>This is a simple example demonstrating the usage of <code>document.body</code>.</p>
    <button onclick="changeBodyContent()">Change Body Content</button>
  `;
}
</script>

</body>
</html>
```

- d**ocument.documentElement** property is a property of the Document Object Model (DOM) that provides access to the root element of the HTML document, which is typically the <html> element. This property allows developers to programmatically interact with and manipulate attributes of the root element using JavaScript.

```
// Accessing the root element of the document
var htmlElement = document.documentElement;

// Manipulating the root element
htmlElement.style.fontSize = "16px"; // Changing font size
htmlElement.setAttribute("lang", "en"); // Setting lang attribute

e.g.
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<title>Document.documentElement Example</title>
</head>
<body>

<h1>Hello, World!</h1>
<p>This is a simple example demonstrating the usage of <code>document.documentElement</code>.</p>
<button onclick="changeLanguage()">Change Language</button>

<script>
// Function to change the language attribute of the root element
```

```
function changeLanguage() {
  // Accessing the root element of the document
  var htmlElement = document.documentElement;

  // Setting the lang attribute to 'fr' (French)
  htmlElement.setAttribute("lang", "fr");
}
</script>

</body>
</html>
```

- **document.title** property is a property of the Document Object Model (DOM) that represents the title of the HTML document. This property allows developers to programmatically access and modify the title of the document using JavaScript.

```
// Accessing the current title of the document
var currentTitle = document.title;
console.log("Current title:", currentTitle);

// Changing the title of the document
document.title = "New Title";
```

e.g.

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<title>Document.title Example</title>
</head>
<body>

<h1>Hello, World!</h1>
<p>This is a simple example demonstrating the usage of <code>document.title</code>.</p>
<button onclick="changeDocumentTitle()">Change Document Title</button>

<script>
// Function to change the document title
function changeDocumentTitle() {
  // Accessing and modifying the title of the document
  document.title = "New Document Title";
}
</script>

</body>
</html>
```

- **document.scripts** property is a property of the Document Object Model (DOM) that provides access to a collection of all <script> elements within the document. This collection allows developers to programmatically interact with and manipulate script elements present in the document using JavaScript.

```
// Accessing the first script element in the document
var firstScript = document.scripts[0];

// Iterating over the scripts collection
for (var i = 0; i < document.scripts.length; i++) {
   var script = document.scripts[i];
   // Do something with each script element
}
e.g.
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<title>Document.scripts Example</title>
</head>
<body>

<h1>Hello, World!</h1>
<p>This is a simple example demonstrating the usage of
<code>document.scripts</code>.</p>

<script>
// Accessing and manipulating script elements
var scriptElements = document.scripts;

// Displaying the number of script elements in the document
console.log("Number of script elements:", scriptElements.length);

// Iterating over the script elements and displaying their src attribute
for (var i = 0; i < scriptElements.length; i++) {
   console.log("Script element", i+1, "- src:", scriptElements[i].src);
}
</script>

<script >

// Displaying the number of script elements in the document
console.log("Number of script elements:", scriptElements.length);

// Iterating over the script elements and displaying their src attribute
for (var i = 0; i < scriptElements.length; i++) {
   console.log("Script element", i+1, "- src:", scriptElements[i].src);
}
</script>

</body>
</html>
```

- **document.embeds** property is a property of the Document Object Model (DOM) that provides access to a collection of all <embed> elements within the document. This collection allows developers to programmatically interact with and manipulate embed elements present in the document using JavaScript.

```
// Accessing the first embed element in the document
var firstEmbed = document.embeds[0];

// Iterating over the embeds collection
for (var i = 0; i < document.embeds.length; i++) {
   var embed = document.embeds[i];
   // Do something with each embed element
}
```

- **document.links** property is a property of the Document Object Model (DOM) that provides access to a collection of all anchor elements (<a>) with an href attribute within the document. This collection allows developers to programmatically interact with and manipulate anchor elements present in the document using JavaScript.

e.g.

```
// Accessing the first anchor element in the document
var firstLink = document.links[0];

// Iterating over the links collection
for (var i = 0; i < document.links.length; i++) {
   var link = document.links[i];
   // Do something with each anchor element
}
```

e.g.

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<title>Document.links Example</title>
</head>
<body>

<h1>Document.links Example</h1>
<p>This is a simple example demonstrating the usage of <code>document.links</code>.</p>

<ul id="linkList">
  <li><a href="https://www.example.com">Example Website</a></li>
  <li><a  href="https://www.google.com">Google</a></li>
  <li><a  href="https://www.github.com">GitHub</a></li>
</ul>

<button onclick="changeLinkURL()">Change Link URL</button>
```

```
<script>
// Function to change the URL of the first link
function changeLinkURL() {
  // Accessing the first anchor element in the document
  var firstLink = document.links[0];

  // Changing the href attribute of the first link
  firstLink.href = "https://www.newurl.com";
}
</script>

</body>
</html>
```

nodeName property : The nodeName property is a read-only property of DOM (Document Object Model) nodes that represents the name of the node. It returns the name of the node as a string, which varies depending on the type of node it is.

e.g.

```
var element = document.getElementById("myElement");
console.log(element.nodeName); // Outputs the tag name of the element (e.g., "DIV")
```

e.g.

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<title>nodeName Property Example</title>
</head>
<body>

<!-- Example elements -->
<div id="myDiv">This is a div</div>
<p>This is a paragraph</p>
<a href="#">This is a link</a>

<script>
// Accessing and displaying node names
var divNode = document.getElementById("myDiv");
console.log("Div Node Name:", divNode.nodeName);

var paragraphNode = document.querySelector("p");
console.log("Paragraph Node Name:", paragraphNode.nodeName);

var linkNode = document.querySelector("a");
console.log("Link Node Name:", linkNode.nodeName);
</script>

</body>
</html>
```

nodeValue property: nodeValue property is a property of DOM (Document Object Model) nodes that represents the value of the node. It varies depending on the type of node it is in the document tree.

```
var textNode = document.createTextNode("Example Text");
console.log(textNode.nodeValue); // Outputs "Example Text"

var attributeNode = document.createAttribute("class");
attributeNode.value = "example-class";
console.log(attributeNode.nodeValue); // Outputs "example-class"
```

e.g.

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<title>nodeValue Property Example</title>
</head>
<body>

<script>
// Creating a text node
var textNode = document.createTextNode("Example Text");

// Retrieving and logging the value of the text node
console.log("Text Node Value:", textNode.nodeValue); // Outputs "Example Text"

// Creating an attribute node
var attributeNode = document.createAttribute("class");
attributeNode.value = "example-class";

// Retrieving and logging the value of the attribute node
console.log("Attribute Node Value:", attributeNode.nodeValue); // Outputs "example-class"
</script>
```

- **“this” refer the current element**

e.g.

```
<!DOCTYPE html>
<html>
<body>

<div onmousedown="mDown(this)" onmouseup="mUp(this)"
style="background-color:red;width:90px;height:20px;padding:40px; border-radius:20px 30px 25px
40px ; color:white;">
Click Me</div>

<script>
function mDown(obj) {
  obj.style.backgroundColor = "blue";
  obj.innerHTML = "Release Me";
}

function mUp(obj) {
  obj.style.backgroundColor="yellow";
  obj.style.color="blue";
  obj.style.border="3px solid red";
  obj.innerHTML="Thank You";
}
</script>

</body>
</html>
```

e.g.

```
<!DOCTYPE html>
<html>
<head>
<script>
function myFunction(x) {
  x.style.background = "yellow";
```

```
}
</script>
</head>
<body>
Enter your name: <input type="text" onfocus="myFunction(this)">
</body>
</html>
```

Akhilesh Kumar Gupta
(Technical Training specialist)
TechCarrel LLP