Advanced JavaScript Mastery
DOM Manipulation for Interactive Web Development

Source:
https://codefinity.com/courses/v2/3ad37fbc-0d15-4d27-bee7-b107747da548/38eb68fc-e6c8-429d-a873-54d7c7d9bb2d/ed6deacc-c7bf-4a29-8635-0dfedb676131

Master DOM manipulation to create dynamic, interactive web applications. Navigate and modify the DOM's structure, dynamically update content, and implement best practices for security and performance. Differentiate between properties and attributes to optimize your code.

1. What Is the Document Object Model (DOM)?

Now that we've covered JavaScript classes let's switch our focus to the **Document Object Model (DOM)**, which is the interface that allows us to manipulate web pages. The next section will explore how JavaScript interacts with the DOM to create dynamic, interactive content.

Definition.

The **Document Object Model (DOM)** is an interface that represents the structure of an HTML document as a tree of objects. The DOM allows programming languages like JavaScript to manipulate the structure, style, and content of a webpage.

Think of the DOM like a **family tree**. Each element in the document is like a family member, connected by relationships - parents, children, and siblings.

In the DOM, elements, attributes, and text are called **nodes**, and these nodes are arranged hierarchically, just like family members in a family tree. Some nodes are parents with children, while others are siblings.

The Relationship Between HTML and the DOM

When a browser loads an HTML document, it reads the HTML code and builds a corresponding DOM tree. HTML provides the static structure of the webpage, while the DOM is the live, dynamic representation that JavaScript can manipulate.

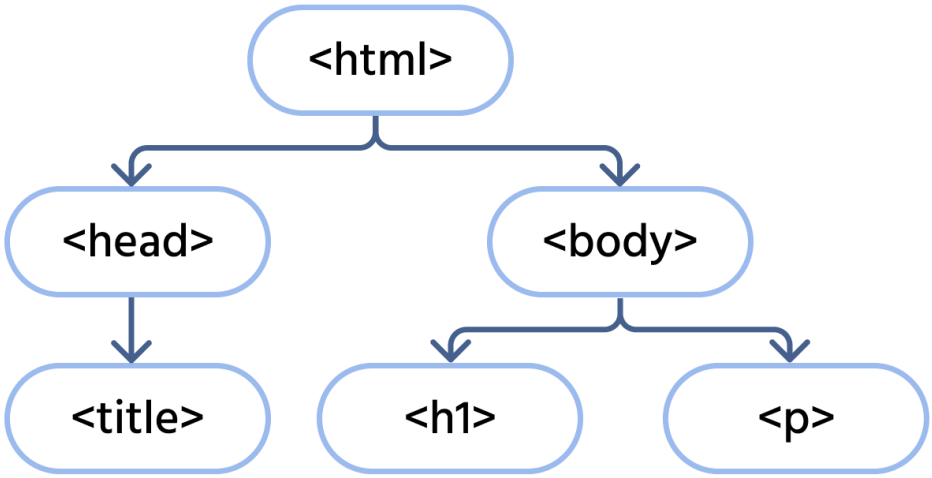
1. HTML defines the initial family tree structure of who is connected to whom, but it is static and doesn't change on its own;
2. The DOM is the family tree in real-time, and it can be changed. New members (elements) can be added or removed, and relationships (parent-child, siblings) can be updated by JavaScript.

For example:

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

```
  <head>
    <title>My Web Page</title>
  </head>
  <body>
    <h1>Welcome to My Website</h1>
    <p>This is a simple paragraph.</p>
  </body>
</html>
```

This HTML creates a basic family tree with an <html> node as the parent, and the <head> and <body> nodes as its children. Within the body, <h1> and <p> are siblings.



This structure is the DOM's live, interactive version of the HTML, where JavaScript can manipulate relationships between nodes (family members).

How JavaScript Interacts with the DOM

JavaScript can access and manipulate the DOM to make changes to the family tree in real-time. You can add new members, remove existing ones, or change their information.

1. **Accessing Family Members**: Just like you can find a specific person in a family tree, JavaScript can access individual nodes in the DOM using methods like document.getElementById() or document.querySelector();
2. **Changing Information**: Once a family member is accessed, their details can be updated. For example, JavaScript can change the text content or attributes of an element using properties like innerHTML or textContent;
3. **Modifying Relationships**: JavaScript can also add new family members to the tree or remove them. This is done by manipulating the DOM structure with methods like appendChild() or removeChild().

Example

Consider this HTML:

```
<p id="greeting">Hello, World!</p>
```

Using JavaScript, we can change the text of the paragraph, just like updating the name of a family member:

```
// Access the family member (DOM element) using its ID
const greetingElement = document.getElementById('greeting');

// Modify the family member's details (DOM content)
greetingElement.textContent = 'Hello, JavaScript!';
```

After running the JavaScript code, the family tree (DOM) is updated, and the text of the paragraph changes from "Hello, World!" to "Hello, JavaScript!"

Family Tree Before the Change	Family Tree After the Change
Hello, World!	Hello, JavaScript!

index.html

```
<!DOCTYPE html>
<html>
  <head>
    <!-- Link to the external CSS file -->
    <link rel="stylesheet" href="index.css" />
  </head>
  <body>
    <!-- This paragraph's text will be modified by JavaScript -->
    <p id="greeting">Hello, World!</p>
    <!-- Link to the external JavaScript file -->
    <script src="index.js"></script>
  </body>
</html>
```

index.js

```
// Select the DOM element with the ID 'greeting'
const greetingElement = document.getElementById('greeting');

// Change the text content of the selected element
greetingElement.textContent = 'Hello, JavaScript!';
```

index.css

```
body {
  display: flex;
  justify-content: center;
  align-items: center;
  height: 100vh;
```

```
  margin: 0;
  background-color: #1e2635;
}

p {
  font-size: 24px;
  color: #f4f4f4;
  padding: 10px;
  border: 2px solid #ff8a00;
  background-color: #4f5e77;
  margin-bottom: 20px;
```

2. Querying and Selecting Elements in the DOM

Introduction to Querying the DOM

Querying the DOM involves selecting and accessing specific elements in the document so you can manipulate or interact with them. JavaScript provides several methods to query elements, each with its own use case and behavior.

GetElementById

getElementById is used to select a single element by its unique ID. It's one of the most commonly used methods because IDs are meant to be unique within the document.

Use Case: Best for selecting a specific element when you know its ID.

Returns: A single element or null if no element is found.

```
// HTML:
<div id="header">Welcome</div>

// JS:
const headerEl = document.getElementById('header');
console.log(headerEl.textContent); // Output: Welcome
```

GetElementsByClassName

getElementsByClassName selects elements by their class name and returns a live HTMLCollection of elements.

Use Case: Useful when you need to select multiple elements with the same class.

Returns: A live HTMLCollection of matching elements, which updates if the DOM changes.

Limitations: Cannot directly use array methods; must convert to an array if needed.

```
// HTML:
<p class="text">First paragraph</p>
<p class="text">Second paragraph</p>
```

```
// JS:
const textEls = document.getElementsByClassName('text');
console.log(textEls); // Output: HTMLCollection(2) [p.text, p.text]
console.log(textEls[0].textContent); // Output: First paragraph
console.log(textEls[1].textContent); // Output: Second paragraph
```

GetElementsByTagName

getElementsByTagName selects elements by their tag name (e.g., div, p) and returns a live HTMLCollection.
Use Case: Ideal for selecting all elements of a particular type.
Returns: A live HTMLCollection of elements.
Limitations: Non-specific, as it selects all elements of a given tag within the context.

```
// HTML:
<ul>
  <li>Item 1</li>
  <li>Item 2</li>
</ul>

// JS:
const listItems = document.getElementsByTagName('li');
console.log(listItems); // Output: HTMLCollection(2) [li, li]
console.log(listItems.length); // Output: 2
```

QuerySelector

querySelector selects the first element that matches a CSS selector. It's versatile, allowing you to use any valid CSS selector to find elements.
Use Case: Great for selecting the first match of any CSS selector.
Returns: The first matching element or null if no match is found.

```
// HTML:
<div class="box" id="main-box">Content</div>

// JS:
const boxEl = document.querySelector('.box'); // Selects the first .box element
console.log(boxEl.id); // Output: main-box
console.log(boxEl.className); // Output: box
console.log(boxEl.textContent); // Output: Content
```

QuerySelectorAll

querySelectorAll selects all elements matching a CSS selector and returns a static NodeList. Unlike other methods, it doesn't update dynamically if the DOM changes.
Use Case: Perfect for selecting multiple elements with complex selectors.
Returns: A static NodeList of elements that doesn't update automatically.

```
// HTML:
<p class="content">Paragraph</p>
<span class="content">Text</span>
```

```
<div class="content">Container</div>

// JS:
const contentEls = document.querySelectorAll('.content');
console.log(contentEls); // Output: NodeList(3) [p.content, span.content,
div.content]
console.log(contentEls.length); // Output: 3
```

Differences Between These Methods
Return Types

Selector	Return Type
getElementById	Returns a single element or null.
getElementsByClassName / getElementsByTagName	Return live HTMLCollections.
querySelector	Returns the first matching element or null.
querySelectorAll	Returns a static NodeList.

Live vs. Static Collections

Selector	Live vs. Static
getElementsByClassName / getElementsByTagName	HTMLCollections are live and update when the DOM changes.
querySelectorAll	NodeLists are static and do not update automatically.

CSS Selectors

Selector	CSS Selectors

querySelector / querySelectorAll	Allow for complex CSS selectors, offering more flexibility than other methods.

Performance

Selector	Performance
getElementById	Generally the fastest because it directly accesses elements by their unique ID.
querySelector / querySelectorAll	Slower but provide greater flexibility.

Practical Example: Form Validation

Let's combine these methods in a practical scenario: highlighting invalid form fields using classes.

index.html

```
<!DOCTYPE html>
<html>
  <head>
    <link rel="stylesheet" href="index.css" />
  </head>
  <body>
    <form id="user-form">
      <input type="text" class="input-field" placeholder="Name" />
      <input type="email" class="input-field" placeholder="Email" />
      <button type="submit">Submit</button>
    </form>
    <script src="index.js"></script>
  </body>
</html>
```

index.css

```
body {
  font-family: 'Poppins', sans-serif;
  background-color: #f3f4f6;
  display: flex;
  justify-content: center;
  align-items: center;
  min-height: 100vh;
```

```css
}

#user-form {
  background-color: #ffffff;
  padding: 30px;
  border-radius: 12px;
  box-shadow: 0 10px 20px rgba(0, 0, 0, 0.1);
  display: flex;
  flex-direction: column;
  gap: 20px;
  width: 320px;
}

.input-field {
  padding: 14px;
  border: 4px solid #ddd;
  border-radius: 8px;
  font-size: 16px;
  transition: border-color 0.3s, box-shadow 0.3s;
}

.input-field:focus {
  border-color: #2563eb;
  outline: none;
  box-shadow: 0 0 5px rgba(37, 99, 235, 0.3);
}

.input-field::placeholder {
  color: #9ca3af;
}

button {
  padding: 14px;
  background-color: #2563eb;
  color: #ffffff;
  border: none;
  border-radius: 8px;
  font-size: 16px;
  cursor: pointer;
  transition: background-color 0.3s, transform 0.2s;
}

button:hover {
  background-color: #1d4ed8;
  transform: translateY(-2px);
}

button:active {
  background-color: #1e40af;
  transform: translateY(0);
}
```

index.js

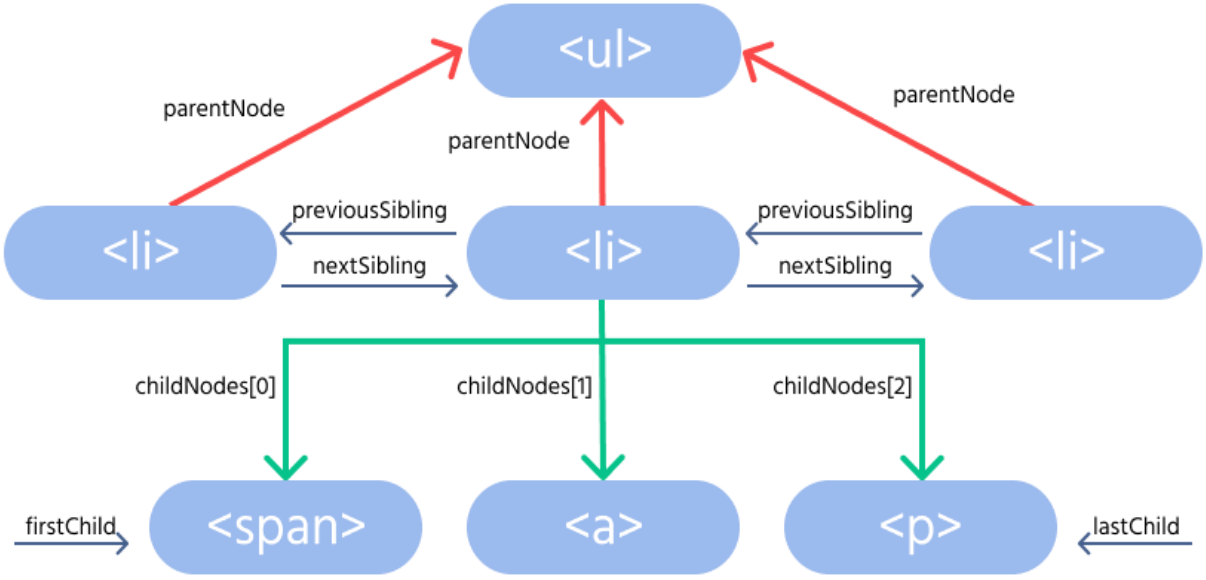
```
const formEl = document.getElementById('user-form');
const inputEls = document.querySelectorAll('.input-field');

formEl.addEventListener('submit', function (event) {
  event.preventDefault();

  // Check each input field for empty values
  Array.from(inputEls).forEach(input => {
    if (!input.value) {
      input.style.borderColor = 'red'; // Highlight empty fields
    } else {
      input.style.borderColor = 'green'; // Reset border if filled
    }
  });
});
```

The getElementById method selects the form element, while querySelectorAll retrieves all input fields with the class .input-field. An event listener on the form's "submit" event prevents the default submission and checks each input field. If a field is empty, it's highlighted with a red border, while filled fields get a green border, providing immediate visual feedback to the user.

3. Understanding the DOM Hierarchy and Relationships

DOM traversal involves moving through the document structure using JavaScript to access, modify, or interact with different nodes. The DOM is structured as a tree with nodes connected in a hierarchy, including **parent**, **child**, and **sibling** relationships:

- **Parent Node**: A node that has other nodes nested within it;
- **Child Nodes**: Nodes directly inside a parent node;
- **Sibling Nodes**: Nodes that share the same parent and are on the same level.




```
<ul>
  <li>Item 1</li>
  <li>Item 2</li>
</ul>
```

In this structure:

- is the parent of the <li> elements;
- Each <li> is a child of the <ul>;
- The two <li> elements are siblings of each other.

Navigating the DOM

JavaScript provides several properties to navigate these relationships, allowing you to move between nodes efficiently.

ParentNode

The parentNode property allows you to access the current node's parent. It's useful when you need to move upwards in the DOM tree.

index.html

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <link rel="stylesheet" href="index.css" />
  </head>
  <body>
    <ul>
      <li class="menu-item">Home</li>
      <li class="menu-item">About</li>
    </ul>
    <script src="index.js"></script>
```

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

index.js

```
const itemEl = document.querySelector('.menu-item'); // Select the first menu
item
const parentEl = itemEl.parentNode; // Access the parent <ul>

// Style the parent element
parentEl.style.border = '2px solid blue';
```

In this example, parentNode accesses the parent <ul> of the selected <li> and adds a border around the entire list.

ChildNodes

The childNodes property returns a collection of all child nodes of a given element, including text nodes (whitespace between elements).

index.html

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <link rel="stylesheet" href="index.css" />
  </head>
  <body>
    <ul id="tasks">
      <li>Task 1</li>
      <li>Task 2</li>
      <li>Task 3</li>
    </ul>
    <script src="index.js"></script>
  </body>
</html>
```

index.js

```
const taskListEl = document.getElementById('tasks');
const children = taskListEl.childNodes; // Get all child nodes
console.log(children); // Output: NodeList(7)Â [text, li, text, li, text, li,
text]

// Loop through each child node and style them if they are element nodes
children.forEach(child => {
  console.log(child);
  if (child.nodeType === 1) {
    // Check if node is an element
    child.style.padding = '10px';
  }
});
```

This example selects the task list and loops through its children, adding padding to each <li> element.

Note. Be mindful that many of the properties can return text nodes (such as whitespace). Always check nodeType === 1 to ensure you are dealing with an element.

FirstChild and LastChild

firstChild: Accesses the first child of an element, including text nodes.
lastChild: Accesses the last child of an element, including text nodes.

Suppose we need to highlight the first and last items in a shopping cart.

index.html

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <link rel="stylesheet" href="index.css" />
  </head>
  <body>
    <ul id="cart">
      <li>Product A</li>
      <li>Product B</li>
    </ul>
    <script src="index.js"></script>
  </body>
</html>
```

index.js

```
const cart = document.getElementById('cart');
console.log(cart.childNodes); // Output: NodeList(5)Â [text, li, text, li, text]

console.log(cart.firstChild); // Output: #text
// Highlight the first product added to the cart
if (cart.firstElementChild) {
  cart.firstElementChild.style.backgroundColor = 'lightgreen';
}

console.log(cart.lastChild); // Output: #text
// Highlight the last product added to the cart
if (cart.lastElementChild) {
  cart.lastElementChild.style.backgroundColor = 'lightblue';
}
```

This example highlights the first and last items in a list, demonstrating how to access and modify the first and last nodes.

Because of the fact that firstChild and lastChild often refer to text nodes (like whitespace) rather than the actual element nodes (<li>) - a better approach is to use firstElementChild and lastElementChild, which specifically target the first and last child elements, skipping over any text nodes.

PreviousSibling and NextSibling

nextSibling: Retrieves the next sibling node of the current node. previousSibling: Retrieves the previous sibling node of the current node.

Consider navigating between tasks in a to-do list, marking tasks as "up next" or "previously completed."

index.html

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <link rel="stylesheet" href="index.css" />
  </head>
  <body>
    <ul id="todo-list">
      <li>Read documentation</li>
      <li id="current">Practice coding</li>
      <li>Review code</li>
    </ul>
    <script src="index.js"></script>
  </body>
</html>
```

index.js

```
const currentTask = document.getElementById('current');

console.log(currentTask.nextSibling); // Output: #text
// Mark the next task as "up next"
const nextTask = currentTask.nextElementSibling;
if (nextTask) {
  nextTask.style.fontWeight = 'bold';
}

console.log(currentTask.previousSibling); // Output: #text
// Mark the previous task as "completed"
const previousTask = currentTask.previousElementSibling;
if (previousTask) {
  previousTask.style.textDecoration = 'line-through';
}
```

In this scenario:

- nextSibling moves to the next task and marks it as "up next";
- previousSibling marks the previous task as completed.

However, nextSibling and previousSibling often point to text nodes (such as whitespace). To target the actual element nodes (<li>), we should use nextElementSibling and previousElementSibling, which specifically skip over text nodes.

Practical Example: Task Manager

Imagine we're building a task manager where users can add, highlight, and manage tasks. We want to dynamically update tasks, mark the first task as the "priority," highlight the next task as "upcoming," and mark the last task as "completed." This example will demonstrate navigating through the DOM using parentNode, childNodes, firstElementChild, lastElementChild, nextElementSibling, and previousElementSibling.

index.html

```
<!DOCTYPE html>
<html>
  <head>
    <link rel="stylesheet" href="index.css" />
  </head>
  <body>
    <div id="task-manager">
      <button id="highlight-tasks">Highlight Tasks</button>
      <ul id="task-list">
        <li>Complete project report</li>
        <li>Attend team meeting</li>
        <li>Review code submissions</li>
      </ul>
    </div>
    <script src="index.js"></script>
  </body>
</html>
```

index.css

```
body {
  font-family: 'Verdana', sans-serif;
  background-color: #f4f6f8;
  display: flex;
  justify-content: center;
  align-items: center;
  min-height: 100vh;
}

#task-manager {
  background-color: #ffffff;
  padding: 20px;
  border-radius: 12px;
  box-shadow: 0 8px 16px rgba(0, 0, 0, 0.1);
  width: 350px;
  text-align: center;
```

```
}

#task-list {
  list-style: none;
  padding: 0;
  margin: 20px 0;
}

#task-list li {
  background-color: #e2e8f0;
  padding: 12px;
  margin-bottom: 10px;
  border-radius: 6px;
  transition: background-color 0.3s, transform 0.2s;
  cursor: pointer;
}

#task-list li:hover {
  background-color: #cbd5e1;
  transform: translateY(-3px);
}

#highlight-tasks {
  padding: 12px 20px;
  background-color: #2563eb;
  color: #ffffff;
  border: none;
  border-radius: 8px;
  font-size: 16px;
  cursor: pointer;
  transition: background-color 0.3s, transform 0.2s;
}

#highlight-tasks:hover {
  background-color: #1d4ed8;
  transform: translateY(-2px);
}

#highlight-tasks:active {
  background-color: #1e40af;
  transform: translateY(0);
}
```

index.js

```
body {
  font-family: 'Verdana', sans-serif;
  background-color: #f4f6f8;
  display: flex;
  justify-content: center;
  align-items: center;
  min-height: 100vh;
```

```css
}

#task-manager {
  background-color: #ffffff;
  padding: 20px;
  border-radius: 12px;
  box-shadow: 0 8px 16px rgba(0, 0, 0, 0.1);
  width: 350px;
  text-align: center;
}

#task-list {
  list-style: none;
  padding: 0;
  margin: 20px 0;
}

#task-list li {
  background-color: #e2e8f0;
  padding: 12px;
  margin-bottom: 10px;
  border-radius: 6px;
  transition: background-color 0.3s, transform 0.2s;
  cursor: pointer;
}

#task-list li:hover {
  background-color: #cbd5e1;
  transform: translateY(-3px);
}

#highlight-tasks {
  padding: 12px 20px;
  background-color: #2563eb;
  color: #ffffff;
  border: none;
  border-radius: 8px;
  font-size: 16px;
  cursor: pointer;
  transition: background-color 0.3s, transform 0.2s;
}

#highlight-tasks:hover {
  background-color: #1d4ed8;
  transform: translateY(-2px);
}

#highlight-tasks:active {
  background-color: #1e40af;
  transform: translateY(0);
}
```

4. Exploring DOM Properties in JavaScript

JavaScript provides several properties to access and manipulate the content. Understanding these properties allows you to dynamically update the content of web pages based on user interactions or other conditions.

InnerHTML

The innerHTML property allows you to get or set the HTML content of an element, including tags and text. It is one of the most commonly used properties for dynamically updating page content.

index.html	index.js

<!DOCTYPE html>
<html lang="en">
<head>
<link rel="stylesheet" href="index.css" />
</head>
<body>
 Hello, World!
<script src="index.js"></script>
</body>
</html>
	const contentDiv = document.getElementById('content');
	console.log(contentDiv.innerHTML); // Output: Hello, World!
	
	// Modify the content
	contentDiv.innerHTML = 'New Content';

The innerHTML retrieves all the content, including HTML tags, and allows for inserting or updating content with HTML tags.

TextContent

The textContent property gets or sets the text content of an element, stripping out any HTML tags. It's ideal when you need to work with plain text without considering the HTML structure.

index.html	index.js

<!DOCTYPE html>
<html lang="en">
<head>
<link rel="stylesheet" href="index.css" />
</head>
<body>
 Hello, World!
<script src="index.js"></script>
</body>
</html>
	const contentDiv = document.getElementById('content');
	console.log(contentDiv.textContent); // Output: Hello, World!
	
	// Modify the text content
	contentDiv.textContent = 'Plain Text Only';

The textContent retrieves text content only, ignoring HTML tags, and sets plain text, automatically escaping any tags.

Value

The value property is used with form elements like <input>, <textarea>, and <select>, allowing you to get or set the value entered by the user.

index.html	index.js

<!DOCTYPE html>
<html lang="en">
<head>
<link rel="stylesheet" href="index.css" />
</head>
<body>
<input type="text" id="name-input" value="Ray Anthony" />
<script src="index.js"></script>
</body>
</html>
	const nameInput = document.getElementById('name-input');
	console.log(nameInput.value); // Output: Ray Anthony
	
	// Set a new value
	nameInput.value = 'Eva Marie Saint';
	console.log(nameInput.value); // Output: Eva Marie Saint

The value retrieves the current value of form elements and sets a new value for input, textarea, or select elements.

5. Working with Element Attributes in the DOM

Attributes are values added to HTML elements to provide additional information or modify their behavior. JavaScript provides several methods to work with these attributes, allowing you to dynamically get, set, remove, or check for specific attributes.

GetAttribute()

getAttribute() is used to retrieve the value of a specified attribute from an element. It's useful when you need to access specific attribute values like href, src, class, etc.

index.html

```
<!DOCTYPE html>
<html lang="en">
  <head>index.html
    <link rel="stylesheet" href="index.css" />
  </head>
  <body>
    <a id="link" href="https://example.com">Visit Example</a>
    <p id="output"></p>
    <script src="index.js"></script>
  </body>
</html>
```

index.js

```
const linkEl = document.getElementById('link');
const outputEl = document.getElementById('output');

const hrefValue = linkEl.getAttribute('href');
```

```
outputEl.textContent = `The href value is: ${hrefValue}`;
```

Accesses the value of the href attribute of the anchor (<a>) element.

SetAttribute()

setAttribute() is used to add a new attribute or change the value of an existing attribute on an element.

index.html	index.js

<!DOCTYPE html>
<html lang="en">
<head>
<link rel="stylesheet" href="index.css" />
</head>
<body>

<script src="index.js"></script>
</body>
</html>
	const logo = document.getElementById('logo');
	
	logo.setAttribute('alt', 'Updated Company Logo'); // Change the `alt` text

Sets or updates the alt attribute of the image, allowing for dynamic changes based on user actions or application logic.

RemoveAttribute()

removeAttribute() removes a specified attribute from an element, making it useful for conditionally toggling attributes based on certain events or states.

index.html	index.js
<!DOCTYPE html> <html lang="en"> <head> <link rel="stylesheet" href="index.css" /> </head> <body> <button id="submit-btn" disabled>Submit</button> <script src="index.js"></script> </body> </html>	const button = document.getElementById('submit-btn'); button.removeAttribute('disabled'); // Enable the button

Removes the disabled attribute, enabling the button for interaction.

HasAttribute()

hasAttribute() checks whether an element has a specified attribute. This method is particularly useful for conditional logic, ensuring that certain attributes exist before performing further actions.

index.html	index.js

<!DOCTYPE html>
<html lang="en">
<head>
<link rel="stylesheet" href="index.css" />
</head>
<body>
<input id="email-input" type="email" required />
<script src="index.js"></script>
</body>
</html>
	const input = document.getElementById('email-input');
	if (input.hasAttribute('required')) {
	console.log('This input is required.'); // Output: This input is required.
	}

Checks if the required attribute exists on the input field and logs a message accordingly.

6. Adding Elements to the DOM Dynamically

Manipulating the DOM often involves dynamically creating and adding new elements to the document or removing existing elements based on user interactions. Here, we will consider how to create and add elements dynamically.

Creating New Elements and Adding Them to the DOM

JavaScript provides methods like createElement(), appendChild(), and insertBefore() to create new elements and add them to the DOM.

CreateElement()

The createElement() method creates a new HTML element but doesn't add it to the DOM until you explicitly do so with methods like appendChild() or insertBefore().

```
// JS:
const newDiv = document.createElement('div'); // Creates a new `<div>` element
newDiv.textContent = 'This is a new div';
```

AppendChild()

The appendChild() method adds a new child element to the end of a parent element's list of children. It's commonly used for adding elements to the bottom of a section, list, or container.

index.html

```
<!DOCTYPE html>
<html>
  <head>
    <link rel="stylesheet" href="index.css" />
  </head>
  <body>
```

```
    <ul id="list">
      <li>Existing Item 1</li>
    </ul>
    <script src="index.js"></script>
  </body>
</html>
```

index.js

```
const list = document.getElementById('list');
const newItem = document.createElement('li'); // Creates a new `<li>` element
newItem.textContent = 'New Item';
list.appendChild(newItem); // Adds the new item to the end of the list
```

InsertBefore()

The insertBefore() method inserts a new element before a specified existing child element, allowing for more precise placement within a parent.

index.html

```
<!DOCTYPE html>
<html>
  <head>
    <link rel="stylesheet" href="index.css" />
  </head>
  <body>
    <ul id="list">
      <li>Existing Item 1</li>
      <li id="item2">Existing Item 2</li>
    </ul>
    <script src="index.js"></script>
  </body>
</html>
```

index.js

```
const list = document.getElementById('list');
const newItem = document.createElement('li');
newItem.textContent = 'Inserted Item';
const referenceItem = document.getElementById('item2');
//Inserts the new item before Existing Item 2
list.insertBefore(newItem, referenceItem);
```

7. Removing Elements From the DOM

Let's dive into how to remove the elements from the DOM.

Removing Elements from the DOM

Elements can be removed dynamically using methods like removeChild() and remove().

RemoveChild()

The removeChild() method removes a specified child node from the parent node. It requires you to access both the parent and the child to remove.

index.html

```
<!DOCTYPE html>
<html>
  <head>
    <link rel="stylesheet" href="index.css" />
  </head>
  <body>
    <ul id="list">
      <li id="remove-item">Item to be removed</li>
    </ul>
    <script src="index.js"></script>
  </body>
</html>
```

index.js

```
const listEl = document.getElementById('list');
const itemToRemove = document.getElementById('remove-item');
listEl.removeChild(itemToRemove); // Removes the specified item
```

Remove()

The remove() method is a more straightforward approach to removing an element directly without needing to access the parent node explicitly.

index.html

```
<!DOCTYPE html>
<html>
  <head>
    <link rel="stylesheet" href="index.css" />
  </head>
  <body>
    <div id="remove-me">Remove this div</div>
    <script src="index.js"></script>
  </body>
</html>
```

index.js

```
const divEl = document.getElementById('remove-me');
divEl.remove(); // Removes the element directly
```

Practical Example: Dynamic To-Do List Manager

Let's build a dynamic to-do list where users can add new tasks, insert tasks at specific positions, and remove tasks as needed.

index.html

```
<!DOCTYPE html>
<html>
  <head>
    <link rel="stylesheet" href="index.css" />
  </head>
  <body>
    <div id="todo-manager">
      <input type="text" id="new-task" placeholder="Enter a new task" />
      <button id="add-task">Add Task</button>
      <ul id="task-list">
        <li>Existing Task 1 <button class="remove-btn">Remove</button></li>
      </ul>
    </div>
    <script src="index.js"></script>
  </body>
</html>
```

index.js

```
const taskList = document.getElementById('task-list');
const newTaskInput = document.getElementById('new-task');
const addTaskButton = document.getElementById('add-task');

// Function to add remove functionality to a task
function addRemoveFunctionality(taskItem) {
  const removeButton = taskItem.querySelector('.remove-btn');
  removeButton.addEventListener('click', () => {
    taskItem.remove(); // Remove the task when the button is clicked
  });
}

// Add remove functionality to existing tasks
document.querySelectorAll('#task-list .remove-btn').forEach(btn => {
  addRemoveFunctionality(btn.parentElement);
});

// Add a new task to the end of the list
addTaskButton.addEventListener('click', () => {
  const taskText = newTaskInput.value.trim();
  if (taskText) {
    const newTask = document.createElement('li');
    newTask.textContent = taskText;

    // Create a remove button for the new task
    const removeButton = document.createElement('button');
    removeButton.textContent = 'Remove';
```

```
    removeButton.classList.add('remove-btn');
    newTask.appendChild(removeButton);

    addRemoveFunctionality(newTask); // Add remove functionality to the new task
    taskList.appendChild(newTask); // Add the new task to the end of the list
    newTaskInput.value = ''; // Clear the input field
  }
});

// Insert a new task before the first existing task
const insertTaskButton = document.createElement('button');
insertTaskButton.textContent = 'Insert Task at Top';
insertTaskButton.id = 'insert-task-top';
document.getElementById('todo-manager').appendChild(insertTaskButton);

insertTaskButton.addEventListener('click', () => {
  const taskText = newTaskInput.value.trim();
  if (taskText) {
    const newTask = document.createElement('li');
    newTask.textContent = taskText;

    // Create a remove button for the inserted task
    const removeButton = document.createElement('button');
    removeButton.textContent = 'Remove';
    removeButton.classList.add('remove-btn');
    newTask.appendChild(removeButton);

    addRemoveFunctionality(newTask); // Add remove functionality to the new task

    // Insert the new task before the first child of the list
    if (taskList.firstElementChild) {
      taskList.insertBefore(newTask, taskList.firstElementChild);
    } else {
      taskList.appendChild(newTask); // If list is empty, append as the first
item
    }

    newTaskInput.value = ''; // Clear the input field
  }
});
```

index.css

```
body {
  font-family: 'Segoe UI', Tahoma, Geneva, Verdana, sans-serif;
  background-color: #1e1e2f;
  color: #e0e0e0;
  display: flex;
  justify-content: center;
  align-items: center;
```

```
  min-height: 100vh;
  padding: 20px;
}

#todo-manager {
  background: linear-gradient(145deg, #252537, #1a1a29);
  border-radius: 12px;
  box-shadow: 10px 10px 20px #171723, -10px -10px 20px #23233a;
  padding: 20px;
  width: 380px;
  display: flex;
  flex-direction: column;
  gap: 15px;
}

#new-task {
  background-color: #2c2c3a;
  color: #e0e0e0;
  border: 2px solid #333344;
  border-radius: 8px;
  padding: 12px;
  font-size: 14px;
  transition: border-color 0.3s, background-color 0.3s;
}

#new-task:focus {
  border-color: #5a5ac7;
  background-color: #32324a;
  outline: none;
}

#add-task {
  background-color: #3a3a52;
  color: #ffffff;
  border: 2px solid #444466;
  border-radius: 8px;
  padding: 12px;
  font-size: 14px;
  cursor: pointer;
  transition: background-color 0.3s, transform 0.2s;
}

#add-task:hover {
  background-color: #4d4d6b;
  transform: translateY(-3px);
}

#task-list {
  list-style: none;
  padding: 0;
  margin: 0;
```

```
  display: flex;
  flex-direction: column;
  gap: 12px;
}

#task-list li {
  background-color: #2e2e42;
  padding: 10px 15px;
  border-radius: 8px;
  display: flex;
  justify-content: space-between;
  align-items: center;
  font-size: 15px;
  transition: background-color 0.3s;
}

#task-list li:hover {
  background-color: #3d3d58;
}

.remove-btn {
  background-color: #c94f4f;
  color: #ffffff;
  border: none;
  border-radius: 6px;
  padding: 5px 10px;
  font-size: 13px;
  cursor: pointer;
  transition: background-color 0.3s, transform 0.2s;
}

.remove-btn:hover {
  background-color: #b13d3d;
  transform: translateY(-2px);
}

button#insert-task-top {
  background-color: #ff9800;
  color: #ffffff;
  border: none;
  border-radius: 8px;
  padding: 10px 15px;
  font-size: 14px;
  cursor: pointer;
  margin-top: 10px;
  transition: background-color 0.3s, box-shadow 0.2s, transform 0.2s;
}

button#insert-task-top:hover {
  background-color: #fb8c00;
  transform: translateY(-2px);
```

```
  box-shadow: 0 4px 10px rgba(255, 152, 0, 0.3);
}

button#insert-task-top:active {
  background-color: #f57c00;
  box-shadow: none;
  transform: translateY(0);
}
```

8. Modifying Element Styles with JavaScript

In web development, we often need to modify the element styles dynamically. JavaScript provides two primary ways to change the appearance of elements: updating inline styles using the style property and managing classes with classList.

Changing Inline Styles Using the style Property

The style property in JavaScript allows you to directly modify the inline CSS of an HTML element. This method gives you full control over individual CSS properties but is limited to inline styles and only affects the specific element.

We can modify individual CSS properties using dot notation on the style object of an element. The property names are written in camelCase (e.g., backgroundColor instead of background-color).

index.html

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <link rel="stylesheet" href="index.css" />
  </head>
  <body>
    <div class="container">
      <div id="box"></div>
      <button id="change-style">Change Style</button>
    </div>
    <script src="index.js"></script>
  </body>
</html>
```

index.js

```
const box = document.getElementById('box');
const changeStyleButton = document.getElementById('change-style');

// Event listener to change the style
changeStyleButton.addEventListener('click', () => {
  box.style.backgroundColor = 'coral'; // Change background color
  box.style.width = '200px'; // Change width
  box.style.border = '2px solid black'; // Add a border
});
```

index.css

```css
.container {
  display: flex;
  justify-content: space-between;
  align-items: center;
  padding: 20px;
}

#box {
  width: 100px;
  height: 100px;
  background-color: lightblue;
  border-radius: 10px;
  box-shadow: 2px 2px 10px rgba(0, 0, 0, 0.2);
  transition: all 0.5s ease;
}

button {
  padding: 10px 20px;
  background-color: #008cba;
  color: white;
  border: none;
  border-radius: 5px;
  cursor: pointer;
  transition: background-color 0.3s ease;
}

button:hover {
  background-color: #005f6b;
}
```

The style property is used to modify the inline styles of the #box element. Each CSS property is accessed as an individual JavaScript property (e.g., box.style.backgroundColor), allowing you to change specific aspects of the element's style dynamically.

Adding and Removing CSS Classes with classList

The classList property provides a more flexible and powerful way to manage the styles of an element by adding, removing, or toggling CSS classes. This method is more maintainable than directly modifying inline styles, as it allows you to take advantage of external CSS files and keep your styling separate from your JavaScript logic.

Method	Description
classList.add()	Adds a class to an element, applying all the styles associated with that class.
classList.remove()	Removes a class, thereby removing its associated styles.
classList.toggle()	Adds the class if it’s not present, or removes it if it is, making it ideal for toggling styles on and off.
classList.replace()	Replaces one class with another, which can be useful when you want to swap between two predefined states.

index.html

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <link rel="stylesheet" href="index.css" />
  </head>
  <body>
    <div class="container">
      <div id="circle" class="default-style"></div>
      <button id="toggle-style">Toggle Style</button>
    </div>
    <script src="index.js"></script>
  </body>
</html>
```

index.js

```
const circle = document.getElementById('circle');
const toggleStyleButton = document.getElementById('toggle-style');

// Event listener to toggle the highlight class
toggleStyleButton.addEventListener('click', () => {
  circle.classList.toggle('highlight'); // Toggles the 'highlight' class
});
```

index.css

```
.container {
  display: flex;
  justify-content: space-between;
  align-items: center;
```

```
  padding: 20px;
}

.default-style {
  width: 120px;
  height: 120px;
  background-color: #a8e6cf;
  border-radius: 50%;
  box-shadow: 0 4px 10px rgba(0, 0, 0, 0.1);
  transition: all 0.4s ease;
}

.highlight {
  border: 6px solid #ff6f69;
  transform: scale(1.1);
  box-shadow: 0 8px 20px rgba(0, 0, 0, 0.2);
}

button {
  padding: 12px 25px;
  background-color: #45a29e;
  color: white;
  font-size: 16px;
  border: none;
  border-radius: 25px;
  cursor: pointer;
  box-shadow: 0 4px 8px rgba(0, 0, 0, 0.1);
  transition: background-color 0.3s ease, transform 0.2s ease;
}

button:hover {
  background-color: #326a6f;
  transform: translateY(-2px);
}

button:active {
  transform: translateY(0);
}
```

classList.toggle() is used to add or remove the highlight class when the button is clicked. This method avoids cluttering the inline style attribute and relies on pre-defined CSS rules for maintainability.

Practical Example: Switching Themes

Let's take a practical example where the user can switch between a light and dark theme using classList.

index.html

```
<!DOCTYPE html>
<html>
  <head>
    <link rel="stylesheet" href="index.css" />
  </head>
  <body>
    <div id="theme-container">
      <p>This is a theme switcher example</p>
      <button id="toggle-theme">Switch Theme</button>
    </div>
    <script src="index.js"></script>
  </body>
</html>
```

index.js

```
const themeButton = document.getElementById('toggle-theme');
const bodyElement = document.body;

// Toggle between default light theme and dark theme
themeButton.addEventListener('click', () => {
  bodyElement.classList.toggle('dark-theme'); // Toggle the dark theme on and off
});
```

index.css

```
body {
  font-family: 'Segoe UI', Tahoma, Geneva, Verdana, sans-serif;
  display: flex;
  justify-content: center;
  align-items: center;
  transition: background-color 0.3s ease, color 0.3s ease;
}

#theme-container {
  background-color: #f4f4f9;
  padding: 20px;
  border-radius: 12px;
  box-shadow: 0 4px 10px rgba(0, 0, 0, 0.1);
  text-align: center;
  transition: background-color 0.3s ease;
}

#theme-container p {
  font-size: 18px;
```

```
  margin-bottom: 15px;
}

#toggle-theme {
  padding: 10px 20px;
  border: none;
  border-radius: 8px;
  background-color: #007bff;
  color: #ffffff;
  font-size: 16px;
  cursor: pointer;
  transition: background-color 0.3s ease, transform 0.2s ease;
}

#toggle-theme:hover {
  background-color: #0056b3;
  transform: translateY(-2px);
}

#toggle-theme:active {
  background-color: #004494;
  transform: translateY(0);
}

.dark-theme {
  background-color: #121212;
  color: #f4f4f9;
}

.dark-theme #theme-container {
  background-color: #1e1e2f;
  box-shadow: 0 4px 15px rgba(0, 0, 0, 0.5);
}

.dark-theme #toggle-theme {
  background-color: #ff9800;
  color: #ffffff;
}

.dark-theme #toggle-theme:hover {
  background-color: #fb8c00;
}
.dark-theme #toggle-theme:active {
  background-color: #f57c00;
}
```

classList.replace() is used to swap the light-theme and dark-theme classes based on the current state.