DOM Manipulation

Application Programming Interface

- An Application Programming Interface (API) is a set of rules and protocols that allows different software applications to communicate and interact with each other.
- Web APIs specifically refer to the APIs provide a standardized way for different software components to exchange data and perform specific functions.

Web APIs

As we have seen in Lecture 4, web APIs are an integral part of web development. They allow developers to access and interact with web-based services, retrieve data, and perform various operations.

Types of Web APIs:

Web APIs can be broadly categorized into two types:

a. **Browser APIs:**

- These APIs are built into web browsers and provide capabilities for accessing and manipulating various browser-related features.
- Browser APIs are provided by web browsers and enable developers to interact with browser features and functionalities.
- Some common browser APIs are accessible through the `**window**` object, which is a global object in the browser environment.
- Examples of browser APIs include:

- **Console API:** The Console API, accessible through the **`window.console`** property, provides methods to output messages and debug information to the browser console. It is commonly used for logging and debugging purposes.
- **Alert API:** The Alert API, accessible through the **`window.alert`** method, displays a simple dialog box with a message to the user. It is often used to show important notifications or prompts.
- **DOM API (Document Object Model):**

  The DOM API allows developers to access and manipulate the elements of an HTML or XML document. It represents the document as a tree-like structure, where each element is a node.

  The Document object, accessible through the **`window.document`** property, represents the web page loaded in the browser and provides methods and properties to interact with the document's content.

b. Third-Party APIs:

- These APIs are developed by external providers and allow developers to access their services and integrate their functionalities into their applications.
- Examples include:
  - **Google Maps API:** Google Maps API provides developers with the ability to embed and interact with maps within their web applications. It offers various services like geocoding, directions, and map customization.

Document Object Model

- The Document Object Model (DOM) is a programming interface for HTML and XML documents, representing them as a structured tree-like model.
- The DOM allows developers to access, manipulate, and navigate the elements and content of a web page dynamically.

Representation of the DOM:

- The DOM represents an HTML or XML document as a hierarchical structure known as the DOM tree.
- The DOM tree consists of various nodes, where each node represents an element, attribute, or text within the document.
- The relationship between nodes is defined by parent-child relationships, with each node having zero or more child nodes.
- **DOM Tree Representation:** It represents the structure of an HTML or XML document. Here's an example of a simplified DOM tree for an HTML document:

- The topmost node in the DOM tree is the `**document**` object, representing the entire HTML document.
- The `**documentElement**` node represents the `**<html>**` element, which serves as the root of the DOM tree.
- The `**documentElement**` node has child nodes, such as the `**head**` and `**body**` elements.
- The `**head**` element contains child nodes, including the `**title**` element, which represents the title of the document.
- The `**body**` element represents the document's body and can contain various elements, such as headings (`**h1**`, `**h2**`, etc.), paragraphs (`**p**`), images (`**img**`), and more.
- The DOM tree continues to branch out, with each element having its own child nodes, forming a hierarchical structure.

Node Access using the Document Object

The Document object provides various methods and properties to access different nodes within the DOM tree using the dot operator.

a. `**document.body**`

- Returns the `**<body>**` element of the document.
- Example:

```
var bodyElement = document.body;
```

b. `**firstElementChild**`

- Returns the first immediate child element of the document.
- Example:

```
var firstChild = document.firstElementChild;
```

c. `**lastElementChild**`

- Returns the last immediate child element of the document.
- Example:

```
var lastChild = document.lastElementChild;
```

d. `children`

- Returns a collection of immediate child elements of the document.
- Example:

```
var children = document.children;
```

These are just a few examples of the many methods available for the Document object. Each method provides different ways to interact with and manipulate the elements and content within the DOM tree.

Selectors in JavaScript

- Selectors in JavaScript are used to fetch data or access particular nodes within the Document Object Model (DOM) of an HTML document.
- Selectors allow developers to target specific elements, classes, or IDs to perform operations on them dynamically.

Commonly Used Selectors:

- **`document.querySelector(selector)`**
  - Returns the first element that matches the specified CSS selector.
  - Example:
    ```
    var element = document.querySelector(".myClass");
    ```
- **`document.querySelectorAll(selector)`**
  - Returns a collection of elements that match the specified CSS selector.
  - Example:
    ```
    var elements = document.querySelectorAll("div");
    ```

- **`document.getElementById(id)`**
    - Returns the element with the specified ID attribute.
    - Example:
      ```
      var element = document.getElementById("myElement");
      ```
- **`document.getElementsByClassName(className)`**
    - Returns a collection of elements with the specified class name.
    - Example:
      ```
      var elements =
      document.getElementsByClassName("myClass");
      ```
- **`document.getElementsByTagName(tagName)`**
    - Returns a collection of elements with the specified tag name.
    - Example:
      ```
      var elements = document.getElementsByTagName("div");
      ```

Best Practices for Placement of Script Tag

When including JavaScript code in an HTML document, it's important to consider the placement of the **`<script>`** tag for optimal performance and functionality.

- **Placing `<script>` in the `<head>`:**
    - Placing the **`<script>`** tag in the **`<head>`** section allows the script to be loaded and executed before the HTML content is parsed and rendered.
    - This approach is suitable when the script needs to be executed early or when it requires elements in the **`<head>`** section (e.g., modifying the document's metadata).

- Example:

```
<!DOCTYPE html>
<html>
  <head>
    <script src="script.js"></script>
  </head>
  <body>
    <!-- HTML content -->
  </body>
</html>
```

- **Placing `<script>` before the closing `</body>` tag:**
  - Placing the **`<script>`** tag just before the closing **`</body>`** tag allows the HTML content to load and render before the script is executed.
  - This approach improves the perceived page load time since the user can see and interact with the page while the script is being fetched and executed.
  - Example:

```
<!DOCTYPE html>
<html>
  <head>
    <!-- HTML content -->
  </head>
  <body>
    <!-- HTML content -->
    <script src="script.js"></script>
  </body>
</html>
```

It's generally recommended to place scripts just before the closing </body> tag to ensure faster initial page load and better user experience.

Styling Fetched Data from Selectors

- Using JavaScript, we can dynamically fetch elements with selectors and apply custom styles to enhance their appearance or modify their properties based on specific conditions.
- Here's an example of how you can add styling to fetched data using selectors:

```
<!DOCTYPE html>
<html>
  <head>
    <style>
      .highlight {
        background-color: yellow;
      }

      .bold {
        font-weight: bold;
      }
    </style>
  </head>
  <body>
    <h1 class="highlight">Welcome to My Website</h1>
    <p>This is a paragraph with some content.</p>
    <ul>
      <li class="bold">Item 1</li>
      <li>Item 2</li>
      <li class="highlight">Item 3</li>
    </ul>

    <script>
```

```
      // Fetching and styling elements
      var highlightedElements =
document.getElementsByClassName("highlight");
      for (var i = 0; i < highlightedElements.length; i++) {
        highlightedElements[i].style.color = "red";
      }

      var boldElement = document.querySelector("li.bold");
      boldElement.style.fontSize = "20px";
    </script>
  </body>
</html>
```

- In the above example:
    - The CSS styles defined within the `**<style>**` tag specify a yellow background color for elements with the class `**highlight**` and bold font weight for elements with the class `**bold**`.
    - Inside the `**<body>**` section, there are heading, paragraph, and list elements with various classes applied.
    - The JavaScript code within the `**<script>**` tag selects elements using selectors and applies additional styling to them.
    - The `**highlightedElements**` variable fetches all elements with the class `**highlight**`, and a loop sets their color to red.
    - The `**boldElement**` variable uses the selector `**li.bold**` to fetch the specific list item and modifies its font size.

JavaScript Element Manipulation

1. Creating Elements:

In JavaScript, you can dynamically create new HTML elements using the `document.createElement(tagName)` method. Here's an example:

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

// Adding text content to the paragraph
paragraph.textContent = "This is a dynamically created
paragraph.";

// Adding a CSS class to the paragraph
paragraph.classList.add("highlight");

// Appending the paragraph to the body element
document.body.appendChild(paragraph);
```

- In the above example:
    - The `document.createElement("p")` method creates a new `<p>` element.
    - The `textContent` property sets the text content of the paragraph.
    - The `classList.add("highlight")` adds the CSS class "highlight" to the paragraph.
    - The `appendChild` method appends the paragraph as a child of the `document.body` element.

2. Appending Elements:

To append an existing element as a child to another element, you can use the **`appendChild(childElement)`** method. Here's an example:

```
// Creating a new <li> element
var listItem = document.createElement("li");
listItem.textContent = "New Item";

// Selecting the parent <ul> element
var parentList = document.getElementById("myList");

// Appending the new <li> element to the parent <ul>
parentList.appendChild(listItem);
```

- In the above example:
    - The **`document.createElement("li")`** method creates a new **`<li>`** element.
    - The **`textContent`** property sets the text content of the list item.
    - The **`getElementById("myList")`** method selects the parent **`<ul>`** element.
    - The **`appendChild(listItem)`** method appends the list item as a child to the parent **`<ul>`** element.

3. Removing Elements:

To remove an element from the DOM, you can use the remove() method or manipulate its parent element using the **`removeChild(childElement)`** method. Here's an example:

```
// Selecting the element to remove

var elementToRemove = document.getElementById("myElement");
```

```
// Removing the element using the remove() method
elementToRemove.remove();


// Alternatively, removing the element using the parent's removeChild() method
var parentElement = document.getElementById("parentElement");
parentElement.removeChild(elementToRemove);
```

- In the above example:
    - The `**`getElementById("myElement")`**` method selects the element to be removed.
    - The `**`remove()`**` method removes the element directly from the DOM.
    - Alternatively, the `**`getElementById("parentElement")`**` method selects the parent element, and the `**`removeChild(elementToRemove)`**` method removes the specified child element.

Event Listeners

- Event listeners in JavaScript allow you to respond to various actions or events triggered by user interactions or system events.
- By attaching event listeners to elements, you can execute specific code or functions in response to those events.

Using addEventListener():

- The `**addEventListener()**` method is commonly used to attach event listeners to DOM elements. It allows you to specify the event type and the function to be executed when the event occurs.
- The syntax for `**addEventListener()**` is as follows:

```
element.addEventListener(eventType, callbackFunction);
```

  - **element:** The DOM element to which the event listener is attached.
  - **eventType:** The type of event to listen for (e.g., "click", "mouseover", "keydown").
  - **callbackFunction:** The function that will be executed when the event occurs.

Some examples that demonstrate the usage of addEventListener() for different event types:

a) Handling a Click Event:

```
var button = document.getElementById("myButton");

button.addEventListener("click", function() {
  console.log("Button clicked!");
});
```

In this example, when the button with the ID "myButton" is clicked, the anonymous function is executed, which logs a message to the console.

b) Handling a Mouseover Event:

```
var image = document.getElementById("myImage");

image.addEventListener("mouseover", function() {
  image.src = "hover-image.jpg";
});
```

Here, when the mouse cursor hovers over the image with the ID "myImage," the anonymous function is executed, changing the image source to "hover-image.jpg."

c) Handling a Keydown Event:

```
var inputField = document.getElementById("myInput");

inputField.addEventListener("keydown", function(event) {
  console.log("Key pressed: " + event.key);
});
```

In this example, as a key is pressed within the input field with the ID "myInput," the anonymous function is executed, logging the pressed key to the console.

Note: The event object (e.g., event in the examples) can be accessed within the callback function, providing additional information about the event that occurred.

Event Propagation

- Event propagation refers to the process by which events are handled and propagated through the DOM tree.
- Understanding event propagation is essential for managing event flow and handling events effectively.

Event Phases:

Event propagation occurs in two phases: capturing phase and bubbling phase.

Capturing Phase:

During the capturing phase, the event starts at the root of the DOM tree and traverses through each parent element down to the target element.

Bubbling Phase:

In the bubbling phase, after reaching the target element, the event propagates back up the DOM tree, triggering event handlers on each ancestor element.

Example:

Let's consider the following HTML structure for the examples:

```
<div id="outer">
    <div id="middle">
      <div id="inner">
        Click Me
      </div>
    </div>
  </div>
```

a) Bubbling Example:

When a click event occurs on the innermost element, the event bubbles up through the parent elements. You can listen to the event at different levels of the DOM tree:

```
var outer = document.getElementById("outer");
var middle = document.getElementById("middle");
var inner = document.getElementById("inner");

outer.addEventListener("click", function() {
  console.log("Outer div clicked!");
});
```

```
middle.addEventListener("click", function() {

  console.log("Middle div clicked!");

});



inner.addEventListener("click", function() {

  console.log("Inner div clicked!");

});
```

If you click the "Click Me" text, the following output will be logged:

```
Inner div clicked!
Middle div clicked!
Outer div clicked!
```

b) Capturing Example:

You can also listen to events during the capturing phase by setting the third parameter of `**addEventListener()**` to `**true**`:

```
outer.addEventListener("click", function() {

  console.log("Outer div clicked during capturing phase!");

}, true);



middle.addEventListener("click", function() {

  console.log("Middle div clicked during capturing phase!");

}, true);
```

```
inner.addEventListener("click", function() {

  console.log("Inner div clicked during capturing phase!");

}, true);
```

If you click the "Click Me" text, the following output will be logged:

```
Outer div clicked during capturing phase!
Middle div clicked during capturing phase!
Inner div clicked during capturing phase!
```

Stop Propagation:

To prevent event propagation to further elements, you can use the stopPropagation() method. Here's an example:

```
inner.addEventListener("click", function(event) {
  event.stopPropagation();
  console.log("Inner div clicked! Event propagation stopped.");
});
```

With the **`stopPropagation()`** method, only the innermost element's event handler will execute. The output will be:

```
Inner div clicked! Event propagation stopped.
```

By utilizing capturing and bubbling phases and using the **`stopPropagation()`** method, you can control event flow and create more robust event handling mechanisms in your JavaScript applications.

Summarizing it

In this lecture, we have covered the following topics:

- API: We have covered Web API (accessing external services), including Browser API (built-in browser functionality) and third-party API.
- DOM: We have explored DOM representation (hierarchical tree structure) and the concept of the DOM tree.
- Selectors: We have learned about selectors and how to access specific elements using criteria such as ID, class, or tag name.
- DOM Manipulation: We have discussed creating, appending, and removing elements dynamically in the DOM.
- Event Listeners: We have covered event listeners and how to handle user/system events like clicks, mouse movements, or key presses.
- Event Propagation: We have learned about event propagation, including the capturing and bubbling phases in the DOM tree.

References

- Document Object Model: **Link**
- Events in JS: **Link**