**DOM Structure**

The **DOM (Document Object Model)** structure represents the hierarchical structure of an HTML or XML document. It defines the way the document is accessed and manipulated through scripting languages like JavaScript. When a web page is loaded, the browser creates a DOM of the page, which can be dynamically modified using JavaScript.

Here’s a breakdown of the DOM structure:

**1. Root Element (document object):**

* The document object is the root of the DOM. It represents the whole web page and provides methods to access any element in the page.
* Example: document.getElementById(), document.querySelector(), etc.

**2. HTML Element:**

* The html element is the top-level element in the DOM. All other elements are children of this.
* Example:

html

Copy code

<html>

<!-- head and body go here -->

</html>

**3. Head Element:**

* The head element contains metadata about the document, like title, links to stylesheets, and scripts.
* Example:

html

Copy code

<head>

<title>Page Title</title>

<link rel="stylesheet" href="style.css">

</head>

**4. Body Element:**

* The body element contains the visible content of the web page, such as text, images, and interactive elements.
* Example:

html

Copy code

<body>

<h1>Main Heading</h1>

<p>This is a paragraph.</p>

</body>

**5. Nodes:**

The DOM structure is made up of different **node types**, each representing a part of the document.

* **Element Nodes**: Represent HTML elements like <div>, <p>, <a>, etc.
  + Example:

html

Copy code

<div id="container"></div>

* **Text Nodes**: Represent text inside an element.
  + Example:

html

Copy code

<p>This is a text node.</p>

* **Attribute Nodes**: Represent attributes of an element.
  + Example:

html

Copy code

<img src="image.jpg" alt="Image description">

* **Comment Nodes**: Represent HTML comments.
  + Example:

html

Copy code

<!-- This is a comment -->

**6. Parent-Child Relationship:**

* Elements in the DOM have a **parent-child** relationship. For instance, an h1 inside a div is considered the child of that div.

Example:

html

Copy code

<div>

<h1>This is the parent-child relationship</h1>

</div>

In this case, div is the parent, and h1 is the child.

**7. DOM Tree Structure:**

The DOM is often visualized as a tree where each element, attribute, and text node is a node in the tree.

Example:

html

Copy code

<html>

<head>

<title>Title of the document</title>

</head>

<body>

<h1>Welcome to the DOM</h1>

<p>This is a paragraph.</p>

</body>

</html>

The tree structure:

* The html node is the root.
* Inside it are two child nodes: head and body.
* head has a title child node, and body has h1 and p child nodes.

**8. DOM Manipulation:**

JavaScript can be used to manipulate the DOM by adding, removing, or changing elements and attributes dynamically. This allows developers to update the content and structure of a web page in response to user interactions.

Example (JavaScript):

javascript

Copy code

document.getElementById("example").innerHTML = "New content!";

This is the basic structure of the DOM, which helps create dynamic and interactive web pages.

**Selecting Elements from the DOM**

Selecting elements from the DOM is a fundamental operation when interacting with web pages using JavaScript. There are several methods for selecting elements based on various criteria like their id, class, tag name, or CSS selectors.

Methods for Selecting Elements

1. getElementById()

* Selects an element by its unique id.
* Returns a single element or null if no element is found.
* Example:

html

Copy code

<div id="header">Header Content</div>

<script>

const header = document.getElementById('header');

console.log(header); // <div id="header">Header Content</div>

</script>

2. getElementsByClassName()

* Selects all elements that have a specified class name.
* Returns a live HTMLCollection (array-like) of elements.
* Example:

html

Copy code

<div class="card">Card 1</div>

<div class="card">Card 2</div>

<script>

const cards = document.getElementsByClassName('card');

console.log(cards); // [ <div class="card">Card 1</div>, <div class="card">Card 2</div> ]

</script>

3. getElementsByTagName()

* Selects all elements with a specific tag name, such as div, p, or li.
* Returns an HTMLCollection of elements.
* Example:

html

Copy code

<ul>

<li>Item 1</li>

<li>Item 2</li>

</ul>

<script>

const listItems = document.getElementsByTagName('li');

console.log(listItems); // [ <li>Item 1</li>, <li>Item 2</li> ]

</script>

4. querySelector()

* Selects the first element that matches a CSS selector.
* Can use any valid CSS selector (e.g., #id, .class, tag, or complex selectors).
* Example:

html

Copy code

<div class="container">

<p class="text">Paragraph 1</p>

<p class="text">Paragraph 2</p>

</div>

<script>

const firstParagraph = document.querySelector('.text');

console.log(firstParagraph); // <p class="text">Paragraph 1</p>

</script>

5. querySelectorAll()

* Selects all elements that match a CSS selector.
* Returns a static NodeList (not live) of elements.
* Example:

html

Copy code

<div class="container">

<p class="text">Paragraph 1</p>

<p class="text">Paragraph 2</p>

</div>

<script>

const allParagraphs = document.querySelectorAll('.text');

console.log(allParagraphs); // [ <p class="text">Paragraph 1</p>, <p class="text">Paragraph 2</p> ]

</script>

6. getElementsByName()

* Selects elements with a specified name attribute.
* Returns an HTMLCollection of elements.
* Often used with form elements.
* Example:

html

Copy code

<input type="text" name="username" />

<input type="password" name="password" />

<script>

const usernameInput = document.getElementsByName('username');

console.log(usernameInput); // [ <input type="text" name="username" /> ]

</script>

Working with the Results

HTMLCollection vs. NodeList:

* HTMLCollection:
  + Returned by methods like getElementsByClassName() and getElementsByTagName().
  + It is live, meaning it updates automatically if the DOM changes.
  + It is array-like but not an array. You can access elements by index but can’t directly use array methods like forEach().
* NodeList:
  + Returned by methods like querySelectorAll().
  + It is static, meaning it does not change when the DOM changes.
  + It is also array-like, and you can use methods like forEach() on it.

Example of Iterating Over Elements:

html

Copy code

<ul>

<li>Item 1</li>

<li>Item 2</li>

</ul>

<script>

// Using getElementsByTagName (HTMLCollection)

const items = document.getElementsByTagName('li');

for (let i = 0; i < items.length; i++) {

console.log(items[i].textContent); // Outputs "Item 1", "Item 2"

}

// Using querySelectorAll (NodeList)

const allItems = document.querySelectorAll('li');

allItems.forEach(item => {

console.log(item.textContent); // Outputs "Item 1", "Item 2"

});

</script>

Use Cases

* getElementById() is the fastest and most commonly used method when targeting an element with a unique id.
* querySelector() and querySelectorAll() provide flexibility with complex CSS selectors and should be used when you need more specific targeting.
* getElementsByClassName() and getElementsByTagName() are useful for bulk selections by class or tag but should be iterated carefully, as they return live collections.

**DOM Manipulation**

DOM manipulation refers to the process of changing the structure, content, or styling of a web page dynamically using JavaScript. It allows developers to add, modify, or remove elements in response to user actions, making the page interactive.

Common DOM Manipulation Techniques

1. Changing Element Content

You can change the content (text or HTML) of an element using properties like innerHTML, textContent, and innerText.

* innerHTML: Modifies the HTML inside an element (allows HTML tags).

html

Copy code

<div id="content">Old content</div>

<script>

document.getElementById('content').innerHTML = '<p>New content with <b>HTML</b></p>';

</script>

* textContent: Changes the text inside an element (doesn't interpret HTML).

html

Copy code

<div id="content">Old content</div>

<script>

document.getElementById('content').textContent = 'New text content';

</script>

* innerText: Similar to textContent, but it respects the style (such as hidden elements) and renders text closer to how it's displayed.

html

Copy code

<div id="content">Old content</div>

<script>

document.getElementById('content').innerText = 'New innerText content';

</script>

2. Changing Element Attributes

You can change attributes like src, href, class, id, or custom data attributes using setAttribute() and direct property assignment.

* setAttribute(): Sets any attribute of an element.

html

Copy code

<img id="image" src="old-image.jpg" />

<script>

document.getElementById('image').setAttribute('src', 'new-image.jpg');

</script>

* Direct Property Assignment: Another way to update common attributes like id, class, src, or href.

html

Copy code

<a id="link" href="old-link.html">Click me</a>

<script>

document.getElementById('link').href = 'new-link.html';

</script>

3. Adding and Removing Classes

JavaScript provides methods like classList.add(), classList.remove(), classList.toggle(), and classList.contains() for adding, removing, or checking classes.

* Adding a class:

html

Copy code

<div id="box" class="box"></div>

<script>

document.getElementById('box').classList.add('active');

</script>

* Removing a class:

html

Copy code

<script>

document.getElementById('box').classList.remove('box');

</script>

* Toggling a class (add if it’s not there, remove if it is):

html

Copy code

<script>

document.getElementById('box').classList.toggle('highlight');

</script>

* Checking if an element has a class:

html

Copy code

<script>

if (document.getElementById('box').classList.contains('active')) {

console.log('Box is active');

}

</script>

4. Creating and Appending Elements

You can create new elements dynamically using createElement(), and then append them to the DOM using appendChild(), insertBefore(), or append().

* Creating and appending an element:

html

Copy code

<div id="container"></div>

<script>

const newElement = document.createElement('p');

newElement.textContent = 'This is a new paragraph!';

document.getElementById('container').appendChild(newElement);

</script>

* Inserting an element before another element:

html

Copy code

<div id="container">

<p id="first">First paragraph</p>

</div>

<script>

const newElement = document.createElement('p');

newElement.textContent = 'This is inserted before the first paragraph';

const container = document.getElementById('container');

const firstElement = document.getElementById('first');

container.insertBefore(newElement, firstElement);

</script>

5. Removing Elements

You can remove elements from the DOM using the remove() or removeChild() method.

* remove(): Removes an element from the DOM.

html

Copy code

<div id="box">I will be removed</div>

<script>

document.getElementById('box').remove();

</script>

* removeChild(): Removes a child element from its parent.

html

Copy code

<div id="parent">

<div id="child">Child element</div>

</div>

<script>

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

const child = document.getElementById('child');

parent.removeChild(child);

</script>

6. Modifying Styles

You can directly manipulate CSS styles of an element using the style property, or by adding/removing classes.

* Changing inline styles:

html

Copy code

<div id="box">Styled Box</div>

<script>

const box = document.getElementById('box');

box.style.backgroundColor = 'blue';

box.style.width = '100px';

box.style.height = '100px';

</script>

* Using classes for styling (preferred for complex styles):

html

Copy code

<div id="box" class="box"></div>

<script>

document.getElementById('box').classList.add('new-style');

</script>

7. Event-Driven DOM Manipulation

DOM manipulation is often tied to user events like clicks, mouse movements, or keyboard interactions. Event listeners can be attached to elements using addEventListener().

* Handling a button click to change content:

html

Copy code

<button id="myButton">Click Me!</button>

<p id="message"></p>

<script>

const button = document.getElementById('myButton');

const message = document.getElementById('message');

button.addEventListener('click', () => {

message.textContent = 'Button was clicked!';

});

</script>

Example: Interactive Button that Adds a New List Item

Here's a more complete example of how DOM manipulation can be combined with user events.

html

Copy code

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>DOM Manipulation Example</title>

</head>

<body>

<h1>My Task List</h1>

<ul id="taskList">

<li>Task 1</li>

<li>Task 2</li>

</ul>

<input type="text" id="newTask" placeholder="New task">

<button id="addTaskButton">Add Task</button>

<script>

const taskList = document.getElementById('taskList');

const addTaskButton = document.getElementById('addTaskButton');

const newTaskInput = document.getElementById('newTask');

addTaskButton.addEventListener('click', () => {

const newTaskText = newTaskInput.value;

if (newTaskText) {

const newTask = document.createElement('li');

newTask.textContent = newTaskText;

taskList.appendChild(newTask);

newTaskInput.value = ''; // Clear the input

}

});

</script>

</body>

</html>

In this example:

* A new list item is added to a task list whenever the "Add Task" button is clicked.
* The input field is cleared after adding the new task.

**Traversing the DOM**

Traversing the DOM refers to navigating through the hierarchy of HTML elements in the Document Object Model (DOM). It allows you to move between parent, child, sibling, and other related elements. JavaScript provides several methods and properties for this purpose.

DOM Traversal Properties and Methods

1. Parent Node

You can access the parent of an element using the parentNode or parentElement properties.

* parentNode: Returns the parent node of the specified node.

html

Copy code

<div id="parent">

<p id="child">This is a paragraph.</p>

</div>

<script>

const child = document.getElementById('child');

console.log(child.parentNode); // <div id="parent">...</div>

</script>

* parentElement: Similar to parentNode, but it only returns the parent if it's an element (useful for ignoring non-element nodes).

html

Copy code

<script>

console.log(child.parentElement); // <div id="parent">...</div>

</script>

2. Child Nodes

You can access child nodes of an element using childNodes or children.

* childNodes: Returns a NodeList of all child nodes, including text nodes (whitespace).

html

Copy code

<div id="parent">

<p>First child</p>

<p>Second child</p>

</div>

<script>

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

console.log(parent.childNodes); // NodeList [ text, <p>First child</p>, text, <p>Second child</p>, text ]

</script>

* children: Returns an HTMLCollection of only element nodes (ignores text nodes and comments).

html

Copy code

<script>

console.log(parent.children); // HTMLCollection [ <p>First child</p>, <p>Second child</p> ]

</script>

* firstChild / lastChild: Returns the first or last child node (including text nodes).

html

Copy code

<script>

console.log(parent.firstChild); // May return a text node (whitespace)

console.log(parent.lastChild); // May return a text node (whitespace)

</script>

* firstElementChild / lastElementChild: Returns the first or last child that is an element.

html

Copy code

<script>

console.log(parent.firstElementChild); // <p>First child</p>

console.log(parent.lastElementChild); // <p>Second child</p>

</script>

3. Siblings

You can navigate between siblings (elements at the same level in the DOM hierarchy) using nextSibling, previousSibling, nextElementSibling, and previousElementSibling.

* nextSibling: Returns the next sibling node (including text nodes).

html

Copy code

<div id="parent">

<p id="first">First</p>

<p id="second">Second</p>

</div>

<script>

const first = document.getElementById('first');

console.log(first.nextSibling); // May return a text node (whitespace)

</script>

* nextElementSibling: Returns the next sibling that is an element.

html

Copy code

<script>

console.log(first.nextElementSibling); // <p id="second">Second</p>

</script>

* previousSibling: Returns the previous sibling node (including text nodes).

html

Copy code

<script>

const second = document.getElementById('second');

console.log(second.previousSibling); // May return a text node (whitespace)

</script>

* previousElementSibling: Returns the previous sibling that is an element.

html

Copy code

<script>

console.log(second.previousElementSibling); // <p id="first">First</p>

</script>

4. Navigating to Specific Child Nodes

* childNodes[index]: Accesses a specific child node based on its index (including text nodes).

html

Copy code

<script>

console.log(parent.childNodes[1]); // <p>First child</p>

</script>

* children[index]: Accesses a specific element child by its index.

html

Copy code

<script>

console.log(parent.children[0]); // <p>First child</p>

</script>

5. Closest Ancestor Matching a Selector

* closest(selector): Traverses upwards through the DOM and returns the closest ancestor that matches the provided selector.

html

Copy code

<div class="container">

<div class="inner">

<p id="text">Some text</p>

</div>

</div>

<script>

const text = document.getElementById('text');

console.log(text.closest('.container')); // <div class="container">...</div>

</script>

6. Parent Element Matching a Selector

* matches(selector): Checks if the element itself matches a given CSS selector.

html

Copy code

<div id="box" class="container"></div>

<script>

const box = document.getElementById('box');

if (box.matches('.container')) {

console.log('Element is a container.');

}

</script>

7. Traversal Example

Here's an example that combines several traversal methods to navigate between elements.

html

Copy code

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>DOM Traversal Example</title>

</head>

<body>

<div id="parent">

<p id="first">First paragraph</p>

<p id="second">Second paragraph</p>

<p id="third">Third paragraph</p>

</div>

<script>

const second = document.getElementById('second');

// Access parent element

console.log(second.parentNode); // <div id="parent">...</div>

// Access first and last child

console.log(second.parentNode.firstElementChild); // <p id="first">First paragraph</p>

console.log(second.parentNode.lastElementChild); // <p id="third">Third paragraph</p>

// Access next and previous siblings

console.log(second.previousElementSibling); // <p id="first">First paragraph</p>

console.log(second.nextElementSibling); // <p id="third">Third paragraph</p>

</script>

</body>

</html>

Traversing the DOM Summary

* Parent Nodes: Use parentNode or parentElement to move up the DOM tree.
* Child Nodes: Use childNodes (includes all nodes) or children (only element nodes) to access child elements.
* Siblings: Use nextSibling and previousSibling (includes all nodes) or nextElementSibling and previousElementSibling (only elements) to move between sibling elements.
* Closest Matching Element: Use closest(selector) to find the closest matching ancestor based on a CSS selector.

Events and Listeners

**Events** and **listeners** are key concepts in JavaScript that allow you to interact with user actions and other dynamic occurrences in the browser, such as mouse clicks, keyboard presses, form submissions, and even custom events. Event listeners can be attached to elements to listen for and respond to specific events.

**Key Concepts**

1. **Event**: An event is an action or occurrence detected by the browser. It can be triggered by the user or the system (e.g., mouse click, keypress, page load, etc.).
2. **Event Listener**: A function or callback that is triggered in response to an event. Event listeners allow you to specify what should happen when a particular event occurs.

**Adding Event Listeners**

The most common way to handle events is by using the addEventListener() method. This method attaches an event listener to a DOM element.

**Syntax:**

javascript

Copy code

element.addEventListener('event', function, useCapture);

* event: The name of the event (like 'click', 'keydown', 'mouseover', etc.).
* function: The function or callback to be executed when the event occurs.
* useCapture (optional): A boolean indicating whether the event should be captured or bubbled (default is false).

**Common Events**

* **Mouse Events**: 'click', 'dblclick', 'mouseover', 'mouseout', 'mousedown', 'mouseup'
* **Keyboard Events**: 'keydown', 'keyup', 'keypress'
* **Form Events**: 'submit', 'change', 'input', 'focus', 'blur'
* **Document Events**: 'DOMContentLoaded', 'scroll', 'resize'
* **Window Events**: 'load', 'unload', 'error'

**Example: Adding a Click Event Listener**

Here’s a basic example where a button's click event is handled:

html

Copy code

<button id="myButton">Click Me!</button>

<p id="message"></p>

<script>

const button = document.getElementById('myButton');

const message = document.getElementById('message');

button.addEventListener('click', function() {

message.textContent = 'Button was clicked!';

});

</script>

In this example:

* The event listener is attached to the button.
* When the button is clicked, the event listener changes the text content of the paragraph to indicate the button was clicked.

**Removing Event Listeners**

To remove an event listener, you can use the removeEventListener() method. It requires the exact same function reference that was used when the listener was added.

**Syntax:**

javascript

Copy code

element.removeEventListener('event', function);

**Example:**

javascript

Copy code

function showMessage() {

message.textContent = 'Button was clicked!';

}

button.addEventListener('click', showMessage);

// To remove the listener:

button.removeEventListener('click', showMessage);

**Event Object**

When an event occurs, an event object is automatically passed to the event listener function. This object contains details about the event, such as the element that triggered it, mouse position, key pressed, etc.

**Example with Event Object:**

html

Copy code

<button id="myButton">Click Me!</button>

<p id="message"></p>

<script>

const button = document.getElementById('myButton');

const message = document.getElementById('message');

button.addEventListener('click', function(event) {

message.textContent = 'Button clicked at X: ' + event.clientX + ', Y: ' + event.clientY;

});

</script>

In this example:

* The event object (event) provides the mouse coordinates (clientX and clientY) where the click occurred.

**Event Propagation (Bubbling and Capturing)**

Event propagation refers to how events flow through the DOM. It occurs in two phases:

1. **Capturing phase**: The event starts from the root of the document and goes down to the target element.
2. **Bubbling phase**: The event bubbles up from the target element back to the root.

By default, most events bubble, meaning they propagate upward through the DOM tree. You can stop the propagation using event.stopPropagation().

**Example of Event Bubbling:**

html

Copy code

<div id="outer" style="padding: 30px; background-color: lightgray;">

Outer Div

<div id="inner" style="padding: 20px; background-color: lightblue;">

Inner Div

<button id="myButton">Click Me!</button>

</div>

</div>

<script>

const outerDiv = document.getElementById('outer');

const innerDiv = document.getElementById('inner');

const button = document.getElementById('myButton');

outerDiv.addEventListener('click', function() {

alert('Outer Div clicked!');

});

innerDiv.addEventListener('click', function() {

alert('Inner Div clicked!');

});

button.addEventListener('click', function(event) {

alert('Button clicked!');

event.stopPropagation(); // Prevents event from bubbling to parent elements

});

</script>

In this example:

* Clicking the button triggers the button's click handler, but the event doesn't bubble up to the parent divs because of event.stopPropagation().

**Event Delegation**

Event delegation is a technique where you attach a single event listener to a parent element rather than each child. This is efficient for dynamically created elements or when you have many child elements.

**Example of Event Delegation:**

html

Copy code

<ul id="myList">

<li>Item 1</li>

<li>Item 2</li>

<li>Item 3</li>

</ul>

<script>

const list = document.getElementById('myList');

list.addEventListener('click', function(event) {

if (event.target.tagName === 'LI') {

alert('You clicked on ' + event.target.textContent);

}

});

</script>

In this example:

* Instead of attaching a click listener to each li element, a single listener is attached to the ul. The event's target property is used to determine which li was clicked.

**Preventing Default Behavior**

The event.preventDefault() method prevents the default action associated with an event (e.g., following a link or submitting a form).

**Example:**

html

Copy code

<a href="https://www.example.com" id="link">Go to Example.com</a>

<script>

const link = document.getElementById('link');

link.addEventListener('click', function(event) {

event.preventDefault(); // Prevents the link from navigating

alert('Link click prevented');

});

</script>

**Summary of Key Methods**

1. **addEventListener()**: Attach an event listener to an element.
2. **removeEventListener()**: Remove an event listener.
3. **event.preventDefault()**: Prevent the default action of the event.
4. **event.stopPropagation()**: Stop the event from bubbling up the DOM.
5. **event.target**: The element that triggered the event.
6. **event.currentTarget**: The element to which the event handler is attached.

Events and listeners are central to creating interactive web pages. Understanding how to attach, remove, and manage event listeners, as well as how events propagate through the DOM, is crucial for writing effective JavaScript.

**Bubbling and Capturing**

**Event Bubbling** and **Event Capturing** are the two phases of event propagation in the DOM when an event is triggered. These phases describe how events travel through the DOM tree, from the root element to the target element and back.

**Event Propagation Phases**

1. **Capturing (Trickling) Phase**: The event starts at the root of the document and travels down the DOM tree to the target element.
2. **Target Phase**: The event reaches the target element itself.
3. **Bubbling Phase**: The event then bubbles back up from the target element to the root of the DOM tree.

By default, most events propagate in the **bubbling phase**, but you can choose to listen to events in either phase.

**Event Bubbling**

In **event bubbling**, when an event is triggered on a specific element, it first triggers the event handler for that element and then propagates upward (or "bubbles up") through its ancestors in the DOM tree.

**Example of Event Bubbling:**

html

Copy code

<div id="outer" style="padding: 30px; background-color: lightgray;">

Outer Div

<div id="inner" style="padding: 20px; background-color: lightblue;">

Inner Div

<button id="myButton">Click Me!</button>

</div>

</div>

<script>

const outerDiv = document.getElementById('outer');

const innerDiv = document.getElementById('inner');

const button = document.getElementById('myButton');

outerDiv.addEventListener('click', function() {

alert('Outer Div clicked!');

});

innerDiv.addEventListener('click', function() {

alert('Inner Div clicked!');

});

button.addEventListener('click', function() {

alert('Button clicked!');

});

</script>

* In this example, clicking the button triggers three alerts:
  1. The button's click event triggers first.
  2. The event bubbles up to the innerDiv, triggering its event handler.
  3. The event continues to bubble up to the outerDiv, triggering its event handler.

So, clicking on the button causes the click event to "bubble" up to its parent elements.

**Event Capturing**

In **event capturing** (also called **event trickling**), the event starts from the root element and propagates down to the target element. In contrast to bubbling, capturing events travel down the DOM tree.

To listen for events during the capturing phase, you can pass true as the third argument to addEventListener():

**Example of Event Capturing:**

html

Copy code

<div id="outer" style="padding: 30px; background-color: lightgray;">

Outer Div

<div id="inner" style="padding: 20px; background-color: lightblue;">

Inner Div

<button id="myButton">Click Me!</button>

</div>

</div>

<script>

const outerDiv = document.getElementById('outer');

const innerDiv = document.getElementById('inner');

const button = document.getElementById('myButton');

outerDiv.addEventListener('click', function() {

alert('Outer Div clicked!');

}, true); // Capture phase

innerDiv.addEventListener('click', function() {

alert('Inner Div clicked!');

}, true); // Capture phase

button.addEventListener('click', function() {

alert('Button clicked!');

}, true); // Capture phase

</script>

* In this case, the event listeners are set to the **capturing phase** (true is passed as the third argument).
* When you click the button, the event propagates **downward**:
  1. First, the outerDiv event listener is triggered.
  2. Then, the innerDiv listener is triggered.
  3. Finally, the button listener is triggered.

The sequence is the reverse of bubbling.

**Event Capturing vs Bubbling**

| **Feature** | **Capturing Phase** | **Bubbling Phase** |
| --- | --- | --- |
| Direction of Propagation | Top-down (from root to target) | Bottom-up (from target to root) |
| Default Phase | Not default (you need to specify it) | Default phase for most events |
| How to Enable | Set the third parameter of addEventListener to true | The third parameter of addEventListener is false or omitted |

**stopPropagation() and stopImmediatePropagation()**

* **event.stopPropagation()**: This method prevents the event from propagating further during the bubbling or capturing phase. It stops the event from moving to parent elements.

**Example:**

javascript

Copy code

button.addEventListener('click', function(event) {

alert('Button clicked!');

event.stopPropagation(); // Stops bubbling

});

* **event.stopImmediatePropagation()**: In addition to stopping the event from bubbling, it also prevents other event listeners on the same element from being executed.

**Example:**

javascript

Copy code

button.addEventListener('click', function(event) {

alert('First listener');

event.stopImmediatePropagation(); // Stops all other listeners

});

button.addEventListener('click', function() {

alert('Second listener');

});

In this example, only the first listener will execute because stopImmediatePropagation() stops the second listener.

**Use Cases**

* **Event Bubbling** is commonly used because it allows for event delegation. By attaching a single event listener to a parent element, you can handle events from its child elements.
* **Event Capturing** is less common but useful in scenarios where you want the parent element to handle the event before any child elements.

**Example: Event Delegation Using Bubbling**

Event delegation allows you to handle events for dynamically added elements or many child elements efficiently by attaching a single event listener to a parent.

html

Copy code

<ul id="myList">

<li>Item 1</li>

<li>Item 2</li>

<li>Item 3</li>

</ul>

<script>

const list = document.getElementById('myList');

list.addEventListener('click', function(event) {

if (event.target.tagName === 'LI') {

alert('You clicked on ' + event.target.textContent);

}

});

</script>

* Instead of attaching a click listener to each li item, the event listener is added to the parent ul.
* When any li is clicked, the event bubbles up to the ul, where it is handled.

**Summary**

* **Event Bubbling**: Events propagate from the target element up through its ancestors in the DOM.
* **Event Capturing**: Events propagate from the root element down to the target element.
* **stopPropagation()**: Stops the event from continuing to propagate through the DOM.
* **Event Delegation**: Uses event bubbling to handle events on child elements by attaching a single listener to a parent.

Both bubbling and capturing give you control over how events flow through the DOM, and understanding them helps you handle complex interactions in your web applications.

**Async await**

In JavaScript, the async and await keywords are used to work with asynchronous code in a more readable and synchronous-looking way, instead of using the traditional .then() syntax with Promises. They allow you to write asynchronous functions in a cleaner and more manageable manner.

**1. The async Keyword**

The async keyword is used to declare an asynchronous function. When you prefix a function with async, it automatically returns a Promise. If the function returns a value, the promise will be resolved with that value. If the function throws an error, the promise will be rejected.

**Example:**

javascript

Copy code

async function getData() {

return "Data received";

}

getData().then(result => console.log(result));

getData().then(result => console.log(result)); // Output: Data received

In this example, getData() returns a Promise that resolves with "Data received". You can handle this promise using .then().

**2. The await Keyword**

The await keyword can only be used inside an async function. It is used to pause the execution of the function until the Promise is resolved or rejected. Essentially, it waits for the Promise to complete and then returns its value.

**Example:**

javascript

Copy code

async function fetchData() {

const response = await fetch('https://api.example.com/data');//wait for resolve or reject

const data = await response.json();

console.log(data);

}

fetchData(); // Fetches the data and logs it

In this example:

* await fetch() waits for the promise returned by the fetch() function to resolve.
* Once the response is received, await response.json() waits for the response to be parsed as JSON.
* After parsing, it logs the data to the console.

**3. Error Handling with async/await**

Errors in async/await can be handled using a try-catch block. This is cleaner than using .catch() for every promise.

**Example:**

javascript

Copy code

async function fetchData() {

try {

const response = await fetch('https://api.example.com/data');

const data = await response.json();

console.log(data);

} catch (error) {

console.error('Error fetching data:', error);

}

}

fetchData();

In this example:

* If an error occurs (e.g., a network issue), the catch block will handle it.

**4. Combining Multiple await Calls**

You can use multiple await statements in an async function. They will run sequentially, waiting for each promise to resolve before moving to the next.

**Example:**

javascript

Copy code

async function processData() {

const response1 = await fetch('https://api.example.com/data1');

const data1 = await response1.json();

const response2 = await fetch('https://api.example.com/data2');

const data2 = await response2.json();

console.log('Data 1:', data1);

console.log('Data 2:', data2);

}

processData();

In this case, response1 is fetched and parsed first, then response2 is fetched and parsed after data1 is fully received.

**5. Using Promise.all with async/await**

If you need to make multiple asynchronous calls in parallel (instead of sequentially), you can use Promise.all with await.

**Example:**

javascript

Copy code

async function processMultipleData() {

const [response1, response2] = await Promise.all([

fetch('https://api.example.com/data1'),

fetch('https://api.example.com/data2')

]);

const data1 = await response1.json();

const data2 = await response2.json();

console.log('Data 1:', data1);

console.log('Data 2:', data2);

}

processMultipleData();

In this example:

* Promise.all() allows both fetch requests to run concurrently, which is faster than running them sequentially.
* await ensures that both promises resolve before moving on to the next line.

**6. Returning Values from async Functions**

Since async functions return a Promise, you can return a value from an async function and handle it using .then() or another await.

**Example:**

javascript

Copy code

async function getNumber() {

return 42;

}

getNumber().then(number => console.log(number)); // Output: 42

Or using await:

javascript

Copy code

async function displayNumber() {

const number = await getNumber();

console.log(number); // Output: 42

}

displayNumber();

**Summary**

* async makes a function return a Promise.
* await pauses the execution of an async function until the Promise is resolved.
* You can use try-catch to handle errors in async functions.
* Use Promise.all for running multiple asynchronous operations in parallel.