MODULE 2

Document Object Model: DOM Manipulation, Selecting Elements, Working with DOM Nodes, Updating Element Content & Attributes, Events, Different Types of Events, How to Bind an Event to an Element, Event Delegation, Event Listeners.

Document Object Model

The HTML DOM (Document Object Model) is a programming interface that represents the structure of a web page in a way that programming languages like JavaScript can understand and manipulate.

Think of it as a tree of objects where each part of your HTML document (elements, attributes, text) is represented as a node, allowing you to dynamically change or interact with the content and structure of the page.

What Does the HTML DOM Look Like?

Imagine your webpage as a tree

- The document is the root.
- HTML tags like <html>, <head>, and <body> are branches.
- Attributes, text, and other elements are the leaves.

Why is DOM Required?

The DOM is essential because

- Dynamic Content Updates: Without reloading the page, the DOM allows content updates (e.g., form validation, AJAX responses).
- User Interaction: It makes your webpage interactive (e.g., responding to button clicks, form submissions).

- Flexibility: Developers can add, modify, or remove elements and styles in real-time.
- Cross-Platform Compatibility: It provides a standard way for scripts to interact with web documents, ensuring browser compatibility.

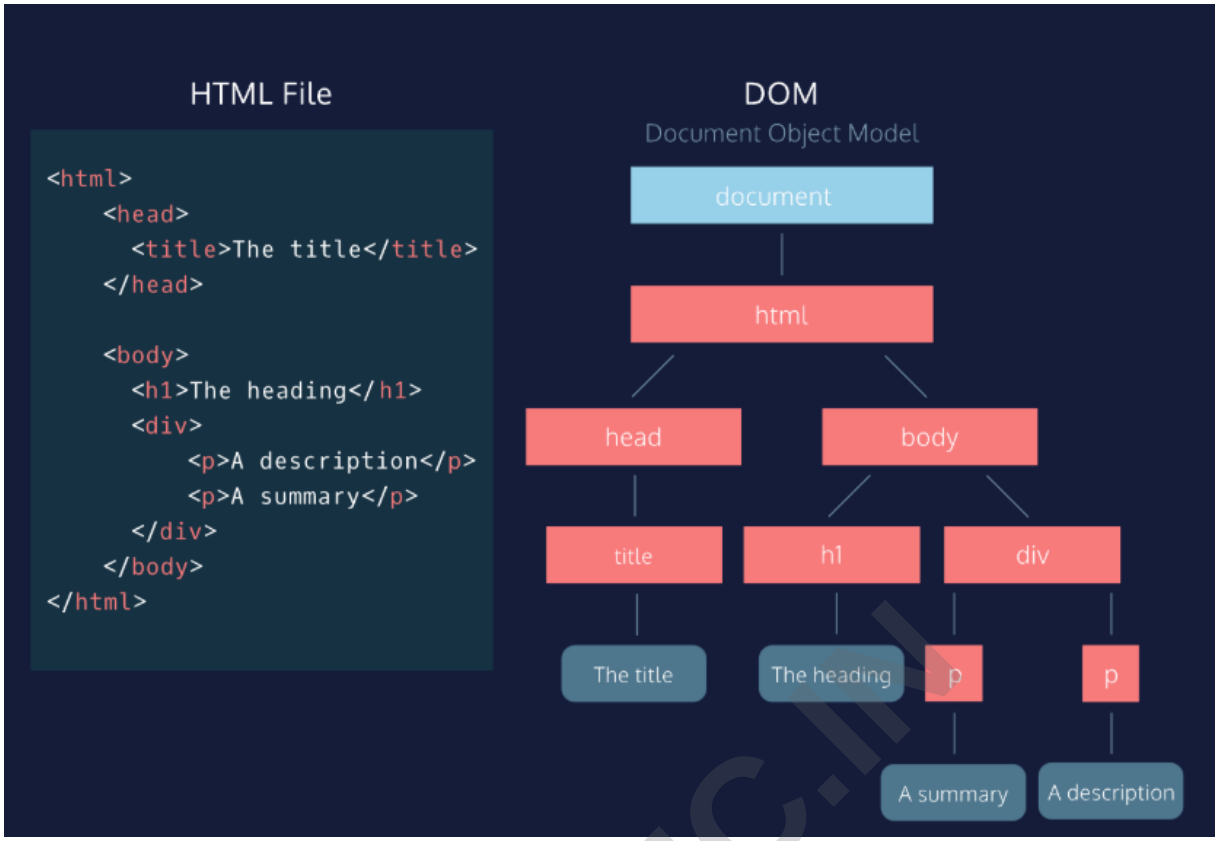
How the DOM Works?

The DOM connects your webpage to JavaScript, allowing you to:

- Access elements (like finding an <h1> tag).
- Modify content (like changing the text of a <p> tag).
- React to events (like a button click).
- Create or remove elements dynamically.

Properties of the DOM

- Node-Based: Everything in the DOM is represented as a node (e.g., element nodes, text nodes, attribute nodes).
- Hierarchical: The DOM has a parent-child relationship, forming a tree structure.
- Live: Changes made to the DOM using JavaScript are immediately reflected on the web page.
- Platform-Independent: It works across different platforms, browsers, and programming languages.



Methods to Select HTML Elements in JavaScript

JavaScript provides multiple methods to **select** HTML elements from the **DOM**. The most commonly used methods are:

Method	Description
getElementById()	Selects an element by its **ID**.
getElementsByClassName()	Selects **all elements** with a given **class name**.
getElementsByTagName()	Selects **all elements** with a given **tag name**.
querySelector()	Selects the **first matching** element using a **CSS selector**.
querySelectorAll()	Selects **all matching** elements using a **CSS selector**.

1. document.getElementById() (Select by ID)

- This method selects an element using its **unique id**.
- Returns a **single element**.

Example: Changing Text of an Element

```
<p id="demo">Hello, World!</p>
<button onclick="changeText()">Click Me</button>
<script>
   function changeText() {
      let element = document.getElementById("demo"); // Select element by ID
      element.textContent = "Text Changed!"; // Modify text content
   }
</script>
```

- getElementById("demo") selects the <p> element.
- textContent changes the text inside the element

2. document.getElementsByClassName() (Select by Class)

- This method selects **all elements** with a specific class.
- Returns a **collection (HTMLCollection)** of elements.

Example: Changing the Style of Multiple Elements

```
<p class="myClass">First Paragraph</p>
<p class="myClass">Second Paragraph</p>
<button onclick="changeColor()">Change Color</button>
```

```
<script>
  function changeColor() {
    let elements = document.getElementsByClassName("myClass"); // Select elements by class
    for (let i = 0; i < elements.length; i++) {
      elements[i].style.color = "red"; // Change text color
    }
  }
</script>
```

- getElementsByClassName("myClass") selects all <p> elements with class="myClass".
- A for loop is used to apply changes to all elements.

3. document.querySelector() (Select First Matching Element)

- This method selects the **first element** that matches a **CSS selector**.
- It allows more flexible selection compared to getElementById().

Example: Changing Font Size

```
<p class="example">Hello World</p>
<p class="example">Namaste</p>

<button onclick="changeSize()">Change Font Size</button>
<script>
  function changeSize() {
    let element = document.querySelector(".example"); // Select first element with class 'example'
    element.style.fontSize = "20px"; // Change font size
  }
```

</script>

- querySelector(".example") selects the **first** <p> element with class "example".
- style.fontSize = "20px" changes its font size.

4. getElementsByTagName(tagName) (Select by Tag Name)

- This method selects **all elements** with a given tag name (e.g., p, div, span).
- It returns an **HTMLCollection**, which is a live collection of elements.

Example: Changing the Background Color of All Paragraphs

```
<p>First paragraph</p>

<p>Second paragraph</p>

<button onclick="changeBg()">Change Background</button>

<script>
    function changeBg() {
        let paragraphs = document.getElementsByTagName("p"); // Select all <p> elements


        for (let i = 0; i < paragraphs.length; i++) {
            paragraphs[i].style.backgroundColor = "yellow"; // Change background color
        }
    }
</script>
```

- getElementsByTagName("p") selects **all** <p> elements.
- The for loop iterates through each <p> and changes the background color.

5. querySelectorAll(selector) (Select Multiple Elements with CSS Selector)

- This method selects **all elements** matching a **CSS selector**.
- It returns a **NodeList**, which is a static collection (does not update when DOM changes).

Example: Changing Font Size of All Elements with Class "text"

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

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

<button onclick="changeFontSize()">Change Font Size</button>

<script>

  function changeFontSize() {

    let elements = document.querySelectorAll(".text"); // Select all elements with class "text"

    elements.forEach(element => {

      element.style.fontSize = "18px"; // Change font size

    });

  }

</script>
```

- querySelectorAll(".text") selects **all** elements with class="text".
- The forEach() method is used to apply styles to each selected element

DOM Manipulation

DOM (Document Object Model) manipulation allows us to dynamically **change HTML content**, **modify CSS styles**, and **update attributes** of HTML elements using JavaScript.

1. Changing HTML Content (innerHTML and textContent)

You can change the **text or HTML content** of an element using:

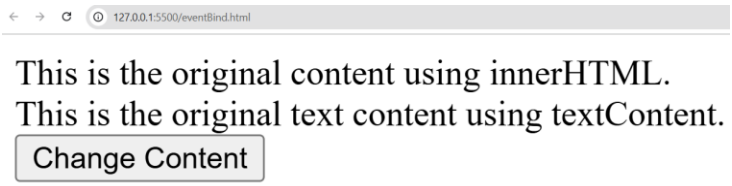
- innerHTML → Updates an element's **HTML content** (including child elements).
- textContent → Changes **only text**, ignoring HTML tags.

Example: Changing Content on Button Click

```
<html>
<body>
<div id="example1">This is the original content using innerHTML.</div>
<div id="example2">This is the original text content using textContent.</div>

<button onclick="changeContent()">Change Content</button>
</body>
<script>
// Function to change content
function changeContent() {
document.getElementById("example1").innerHTML =  "<b>This is changed using innerHTML!</b>";

document.getElementById("example2").textContent =  "This is changed using textContent!";
}
</script>

    </html>
```

Output

before



After button click

- innerHTML changes the entire content of an element, including HTML tags. In this case, we replace the content of the first div with bold text using <b>.
- textContent changes only the text inside the element, ignoring any HTML tags. The second div is updated with plain text, without any HTML formatting.
- The first div shows "This is the original content using innerHTML."
- The second div shows "This is the original text content using textContent."
- After clicking the "Change Content" button.
- The first div will display "This is changed using innerHTML!" with bold text.
- The second div will display "This is changed using textContent!" with plain text.

2. Setting CSS Styles (style.property)

You can change the **appearance** of an element dynamically using JavaScript.

Example: Changing Text Color and Background

```
<p id="styledText">Change my style!</p>
<button onclick="applyStyles()">Apply Styles</button>
<script>
    function applyStyles() {
        let text = document.getElementById("styledText");
        text.style.color = "white";        // Change text color
        text.style.backgroundColor = "blue"; // Change background color
        text.style.padding = "10px";       // Add padding
        text.style.fontSize = "20px";      // Increase font size
    }
</script>
```

Output

Before

After button click

- JavaScript **modifies the inline CSS** of the <p> element, changing color, background, padding, and font size.

3. Modifying Element Attributes (setAttribute() and getAttribute())

You can update or retrieve **attributes** such as src, href, alt, class, etc.

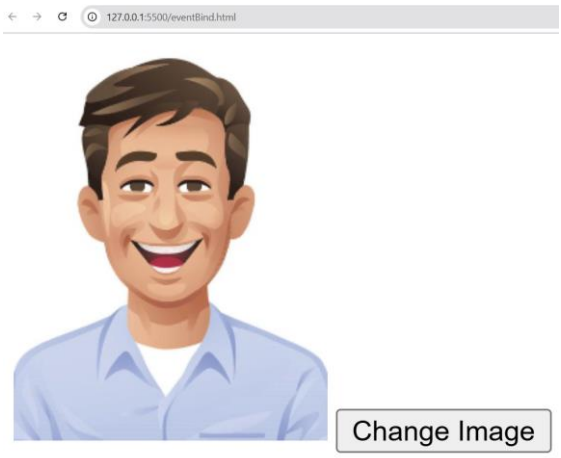
Example: Changing Image Source and Alt Text

```
<img id="image" src="old.jpg" alt="Old Image" width="150">
<button onclick="changeImage()">Change Image</button>
    <script>
        function changeImage() {
            let img = document.getElementById("image");
            img.setAttribute("src", "new.jpg"); // Change image source
            img.setAttribute("alt", "New Image"); // Update alt text

        }

    </script>
```

- The setAttribute() method **updates the src** to "new.jpg", changing the displayed image

Output

before



After button click

Manipulating the DOM with JavaScript is a core aspect of web development. Understanding how to select, modify, and manipulate DOM elements enables you to create dynamic, interactive web applications. With the power to change text, styles, attributes, and even structure in real-time, JavaScript empowers developers to build rich, user-responsive websites.

Working with DOM Nodes

In JavaScript, the Document Object Model (DOM) represents the structure of an HTML document as a tree of nodes. Each element, attribute, and piece of text in an HTML document is a node that can be accessed and manipulated using JavaScript.

Types of DOM Nodes

1. **Element Nodes** – Represent HTML elements (e.g., , , <ul>).
2. **Attribute Nodes** – Represent attributes of elements (e.g., class="box", id="header").
3. **Text Nodes** – Represent text within elements (e.g., inside Hello , "Hello" is a text node).

4. **Comment Nodes** – Represent HTML comments (e.g., <!-- This is a comment -->).

Accessing DOM Nodes

To work with DOM nodes, we can use various methods:

1. **document.getElementById(id)** – Selects an element by its id.

Ex,

```
let title = document.getElementById("heading");
console.log(title); // Logs the element with id "heading"
```

2. **document.getElementsByClassName(className)** – Selects all elements with a specific class.

Ex,

```
let items = document.getElementsByClassName("list-item");
console.log(items[0]); // Logs the first item with class "list-item"
```

3. **document.getElementsByTagName(tagName)** – Selects all elements of a specific tag.

Ex,

```
let paragraphs = document.getElementsByTagName("p");
console.log(paragraphs.length); // Logs the number of <p> elements
```

4. **document.querySelector(selector)** – Selects the first matching element.

Ex,

```
let firstItem = document.querySelector(".list-item");
console.log(firstItem); // Logs the first element with class "list-item"
```

5. **document.querySelectorAll(selector)** – Selects all matching elements.

Ex,

```
let allItems = document.querySelectorAll(".list-item");
console.log(allItems); // Logs a NodeList of all elements with class "list-item"
```

Manipulating DOM Nodes

Once a node is selected, we can modify it.

1. **Changing Content (textContent and innerHTML)**

Ex,

```
let title = document.getElementById("heading");
title.textContent = "New Title"; // Changes only text
title.innerHTML = "<span>New Title</span>"; // Supports HTML
```

2. **Changing Attributes (setAttribute and getAttribute)**

Ex,

```
let link = document.getElementById("myLink");
link.setAttribute("href", "https://www.example.com"); // Change URL
console.log(link.getAttribute("href")); // Get attribute value
```

3. **Modifying CSS Styles**

Ex,

```
let box = document.getElementById("box");
box.style.color = "blue"; // Changes text color
box.style.backgroundColor = "lightgray"; // Changes background
```

Creating and Appending Nodes

New elements can be created dynamically using JavaScript.

1. **Creating and Appending an Element**

Ex,

```
let newPara = document.createElement("p"); // Create a <p> element
newPara.textContent = "This is a new paragraph"; // Add text content
document.body.appendChild(newPara); // Append to the document
```

2. **Removing a Node**

Ex,

```
let removeMe = document.getElementById("removeMe");
removeMe.parentNode.removeChild(removeMe); // Removes the element
```

3. **Replacing an Element**

Ex,

```
let oldPara = document.getElementById("oldText");
let newPara = document.createElement("p");
newPara.textContent = "This is updated text";
oldPara.parentNode.replaceChild(newPara, oldPara);
```


Traversing the DOM

DOM traversal means navigating through elements in the document tree.

1. **Parent Node (parentNode)**

Ex,

```
let child = document.getElementById("child");
console.log(child.parentNode); // Logs the parent element
```

2. **Child Nodes (childNodes and children)**

Ex,

```
let list = document.getElementById("list");
console.log(list.childNodes); // Logs all child nodes (including text and comment nodes)
console.log(list.children); // Logs only element nodes
```

3. **First and Last Child**

Ex,

```
console.log(list.firstElementChild.textContent); // Logs first list item
console.log(list.lastElementChild.textContent); // Logs last list item
```

4. **Sibling Nodes (nextElementSibling and previousElementSibling)**

Ex,

```
console.log(child.previousElementSibling); // Logs previous sibling
console.log(child.nextElementSibling); // Logs next sibling
```

Example

```
<!DOCTYPE html>
<html lang="en">
<head>
  <title>DOM Example</title>
</head>
<body>
  <div id="container">
    <h1 id="heading">Hello World</h1>
    <p>This is a paragraph.</p>
    <ul id="list">
      <li class="list-item">Item 1</li>
      <li class="list-item" id="removeMe">Item 2</li>
      <li class="list-item">Item 3</li>
```

```
        </ul>
        <button onclick="addParagraph()">Add Paragraph</button>
        <button onclick="removeItem()">Remove Item 2</button>
    </div>

    <script>
        function addParagraph() {
            let newPara = document.createElement("p");
            newPara.textContent = "New paragraph added!";
            document.getElementById("container").appendChild(newPara);
        }

        function removeItem() {
            let item = document.getElementById("removeMe");
            item.parentNode.removeChild(item);
        }
    </script>
</body>
</html>
```

Events and Different Types of Event

JavaScript events are actions or occurrences that happen in the web browser, such as clicking a button, pressing a key, or loading a page. JavaScript can detect these events and respond to them using event listeners.

What is an Event?

An event is a signal that something has happened on a web page. Common events include:

- Clicking a button (click event)
- Typing in a text field (keydown event)
- Moving the mouse (mousemove event)
- Submitting a form (submit event)

To handle events, we use event listeners.

Types of Events

1. Mouse Events

Mouse events occur when a user interacts with a webpage using a mouse.

Event	Description
click	Triggered when the user clicks an element
dblclick	Triggered when the user double-clicks
mousedown	Triggered when a mouse button is pressed
mouseup	Triggered when a mouse button is released
mousemove	Triggered when the mouse moves
mouseover	Triggered when the mouse hovers over an element

Event	Description
mouseout	Triggered when the mouse leaves an element

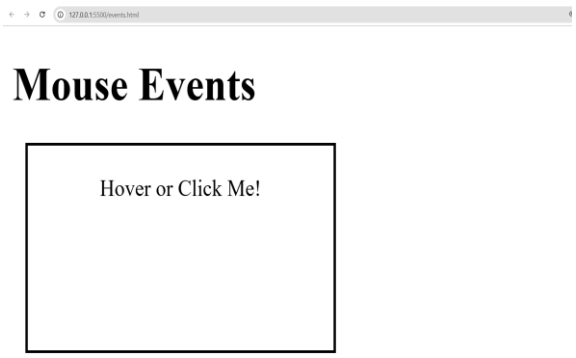
Example

```
<html >
<head>
   <title>Mouse Events</title>
   <style>
      #mouseBox {
         width: 200px;
         height: 100px;
         margin: 10px;
         padding: 20px;
         border: 2px solid black;
         text-align: center;
      }
   </style>
</head>
<body>

   <h1>Mouse Events </h1>
   <div id="mouseBox"
      onmouseover="mouseOver()"
      onmouseout="mouseOut()"
      onclick="mouseClick()"
      ondblclick="mouseDoubleClick()"
      onmousedown="mouseDown()"
      onmouseup="mouseUp()">
      Hover or Click Me!
   </div>
```

```
    <script>
        function mouseOver()
        {
            document.getElementById("mouseBox").style.backg
roundColor = "lightblue";
        }
        function mouseOut()
        {
            document.getElementById("mouseBox").style.backg
roundColor = "white";
        }
        function mouseClick()
        {
            alert("Mouse Clicked!");
        }
        function mouseDoubleClick()
        {
            alert("Mouse Double Clicked!");
        }
        function mouseDown()
        {
            document.getElementById("mouseBox").style.border
= "5px solid red";
         }
        function mouseUp()
         {
            document.getElementById("mouseBox").style.border
= "2px solid black";
         }
    </script>

</body>
</html>
```

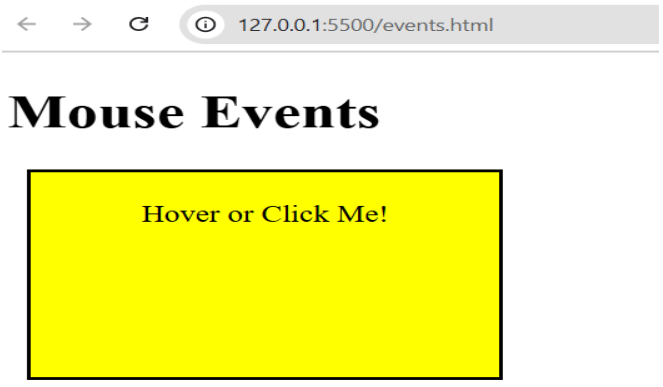
OUTPUT

1. Output without before event

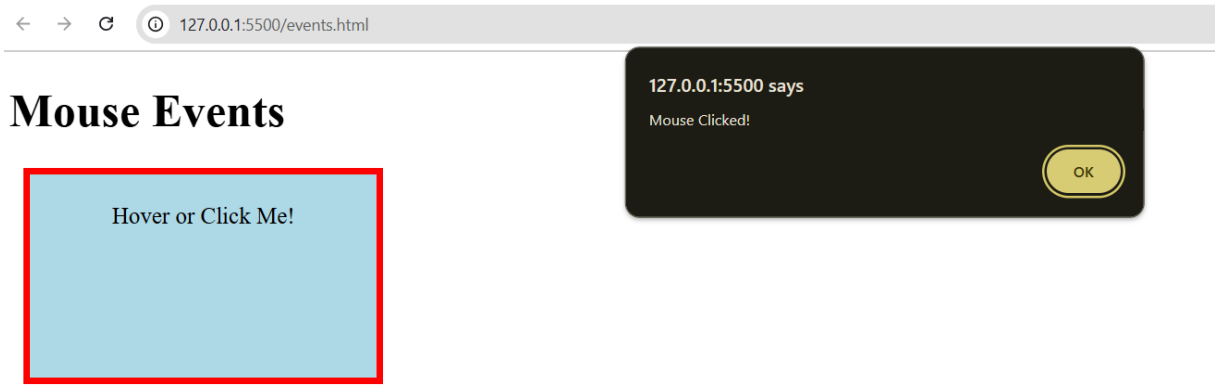


2. Output for mouseover event

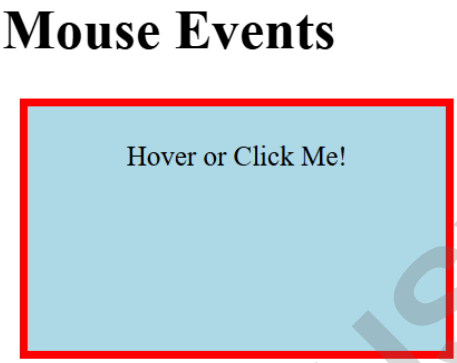
3. Output for mouseout event



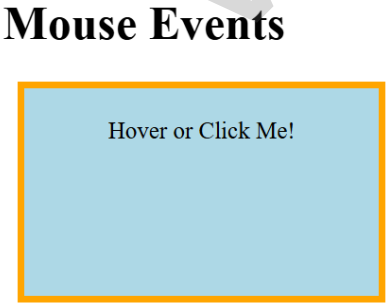
4. Oclick event



5. Mouseup



6. Mousedown



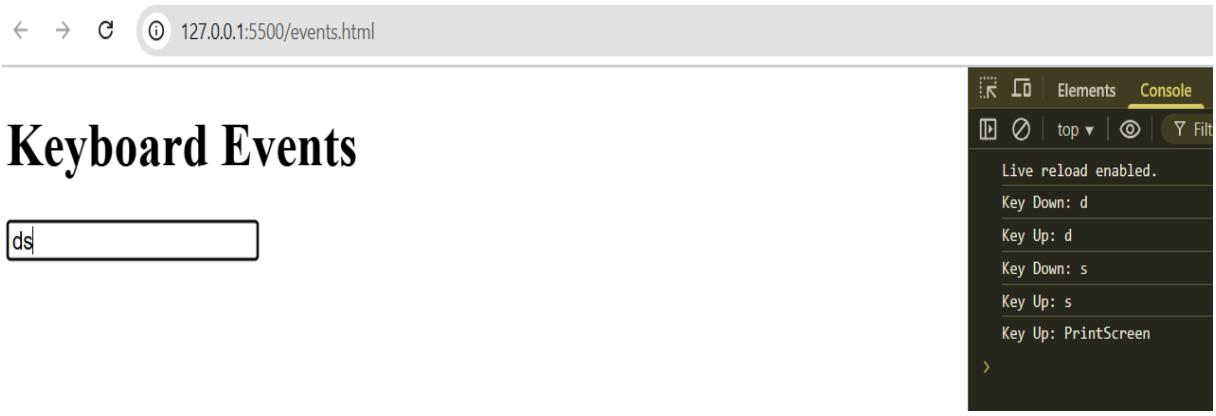
2. Keyboard Events

Keyboard events occur when a user presses a key.

Event	Description
keydown	Triggered when a key is pressed down
keyup	Triggered when a key is released
keypress	(Deprecated) Triggered when a key is pressed (except special keys)

1. Output for keyUp event

2. Output for KeyDown



3.Output for keyPress

3.Form Events

Form events are related to form elements like input fields, checkboxes, and buttons.

Event	Description
submit	Triggered when a form is submitted
change	Triggered when an input value changes
input	Triggered when user types in an input field
focus	Triggered when an element gets focus
blur	Triggered when an element loses focus

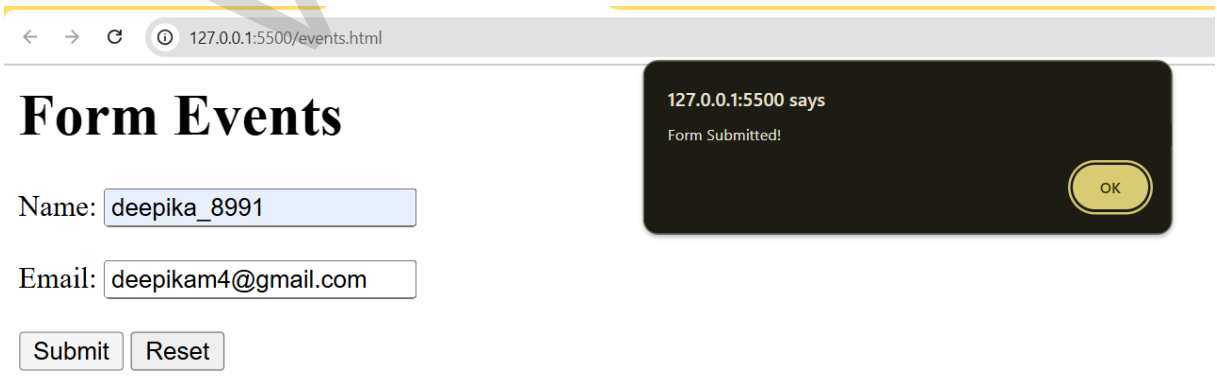
Example

```
<html>
   <head>
   <title>Form Events</title>

</head>
<body>

   <h1>Form Events </h1>

   <form onsubmit="submitEvent(event)"
onreset="resetEvent()">
      <label for="name">Name:</label>
      <input type="text" id="name" name="name"
           onfocus="focusEvent()"
           onblur="blurEvent()"
           oninput="inputEvent()"
           onchange="changeEvent()">
      <br><br>

      <label for="email">Email:</label>
      <input type="email" id="email" name="email">
      <br><br>

      <button type="submit">Submit</button>
      <button type="reset">Reset</button>
   </form>

</body>
<script>
   function focusEvent() {
      console.log("Input field focused!");
   }

   function blurEvent() {
      console.log("Input field lost focus!");
   }
```

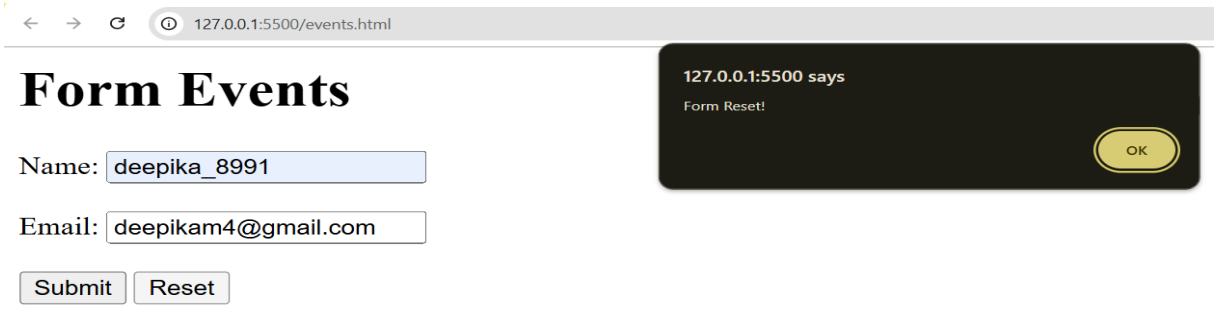
```
    function inputEvent() {
        console.log("User is typing...");
    }

    function changeEvent() {
        console.log("Value changed!");
    }

    function submitEvent(event) {
        event.preventDefault(); // Prevent actual form
submission
        alert("Form Submitted!");
    }

    function resetEvent() {
        alert("Form Reset!");
    }
</script>
</html>
```

OUTPUT

1. **Form submit**



2. Form reset



3. Focus and blur

4. Window Events

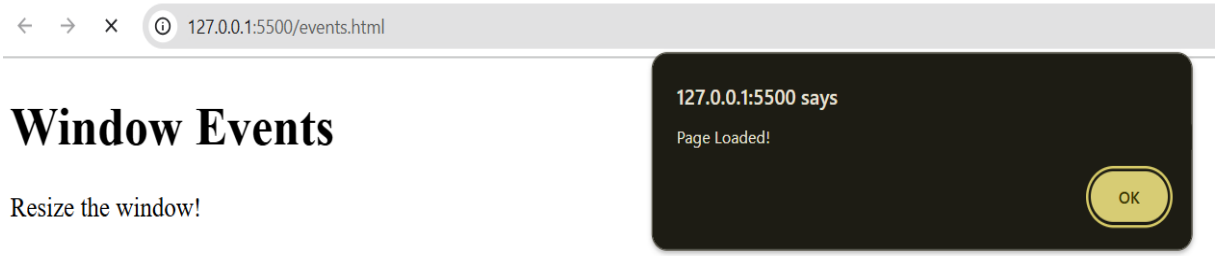
Window events occur at the browser window level.

Event	Description
load	Triggered when the page fully loads
resize	Triggered when the window is resized
scroll	Triggered when the user scrolls the page

Example,

```
<html>
<head>
   <title>Window Events</title>
</head>
<body>

   <h1>Window Events</h1>
   <div id="resizeBox">Resize the window!</div>

   <script>
      window.onload = function()
       {
         alert("Page Loaded!");
       }
      window.onresize = function()
      {
         document.getElementById("resizeBox").innerText =
"Window Resized!";
      }
      window.onscroll = function()
      {
         alert("Page Scrolled!");
      }
   </script>

</body>
</html>
```

OUTPUT 1



OUTPUT 2

Event Binding, Event Delegation & Event Listener

JavaScript provides different ways to handle events, which allow users to interact with web pages. Below, we will explain event binding, event delegation, and event listeners in detail, along with practical examples.

1. Event Binding

Event binding refers to the process of attaching an event (such as click, mouseover, or keydown) directly to an HTML element. When the event occurs, a function executes in response.

Ways to Bind Events

- Inline Event Binding (Inside HTML)
- JavaScript Event Binding (Using onclick)
- Using addEventListener (Modern approach)

Example 1: Inline Event Binding

```
<button onclick="showMessage()">Click Me</button>

<script>
   function showMessage() {
      alert("Button Clicked!");
   }
</script>
```

Drawback: Not recommended because it mixes HTML and JavaScript, reducing code readability and maintainability.

Example 2: JavaScript Event Binding

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

<script>
   document.getElementById("myButton").onclick =
function() {
      alert("Button Clicked!");
   };
</script>
```

Drawback: If another function is assigned to onclick, it replaces the existing one.

Example 3: Using addEventListener (Best Practice)

```
<button id="btn">Click Me</button>

<script>

document.getElementById("btn").addEventListener("click",
function() {
      alert("Event Listener Clicked!");
   });
</script>
```

Advantages:

- Supports multiple event handlers on the same element.
- More flexible than the onclick property.

2. Event Delegation

Event delegation is a technique where we attach a single event listener to a parent element instead of individual child elements. This is useful for dynamically added elements or optimizing event handling for large lists.

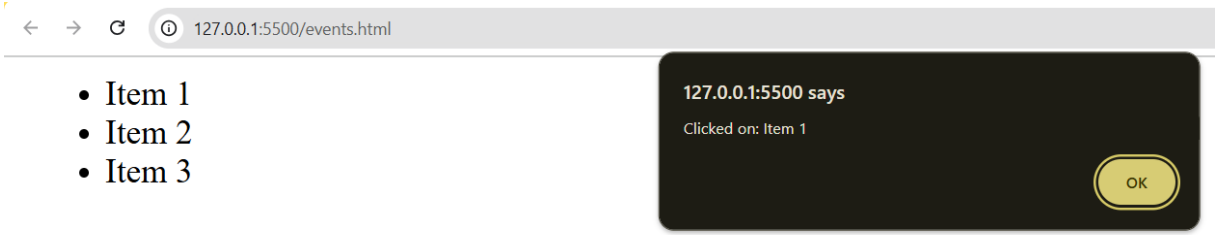
Why Use Event Delegation?

- Reduces memory usage by minimizing event listeners.
- Useful when elements are created dynamically.
- More efficient than binding events to multiple elements separately.

Example: Event Delegation for a List

```
<ul id="myList">
   <li>Item 1</li>
   <li>Item 2</li>
   <li>Item 3</li>
</ul>

<script>

document.getElementById("myList").addEventListener("clic
k", function(event) {
       if (event.target.tagName === "LI") { // Ensures only
<li> elements trigger the event
          alert("Clicked on: " + event.target.textContent);
       }
   });
</script>
```

Output



Advantages of Event Delegation:

- Works on both existing and dynamically added elements.
- More efficient for handling multiple elements.

3.Event Listener (addEventListener)

The addEventListener method allows us to attach multiple event listeners to a single element without overwriting existing events.

Syntax:

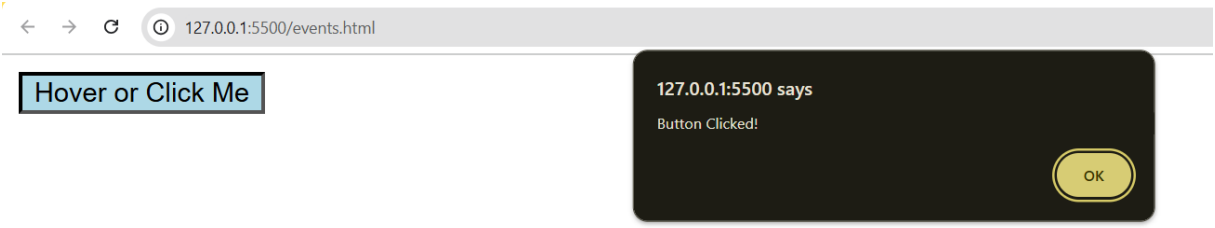
element.addEventListener(event, function, useCapture);

- **event**: The event type (e.g., "click", "mouseover").
- **function:** The function to execute when the event occurs.
- **useCapture:** (Optional) Determines event flow (default is false).

Example: Using addEventListener for Different Events

```
<button id="eventBtn">Hover or Click Me</button>

<script>
  let button = document.getElementById("eventBtn");

  button.addEventListener("click", function() {
    alert("Button Clicked!");
  });

  button.addEventListener("mouseover", function() {
    button.style.backgroundColor = "lightblue";
  });

  button.addEventListener("mouseout", function() {
    button.style.backgroundColor = "";
  });
</script>
```

OUTPUT



Advantages:

- Allows multiple event listeners.
- Provides better flexibility than inline event handlers.
- Can be easily removed using removeEventListener().

Comparison Table

Feature	Event Binding	Event Delegation	Event Listener (`addEventListener`)
Definition	Directly binding an event to an element	Assigning an event to a parent and delegating it to child elements	Using `addEventListener` to attach events
Performance	Less efficient for many elements	More efficient for multiple elements	Flexible and optimized
Supports Multiple Handlers?	❌ No	✅ Yes (via parent)	✅ Yes
Works for Dynamic Elements?	❌ No	✅ Yes	❌ No (unless reattached)