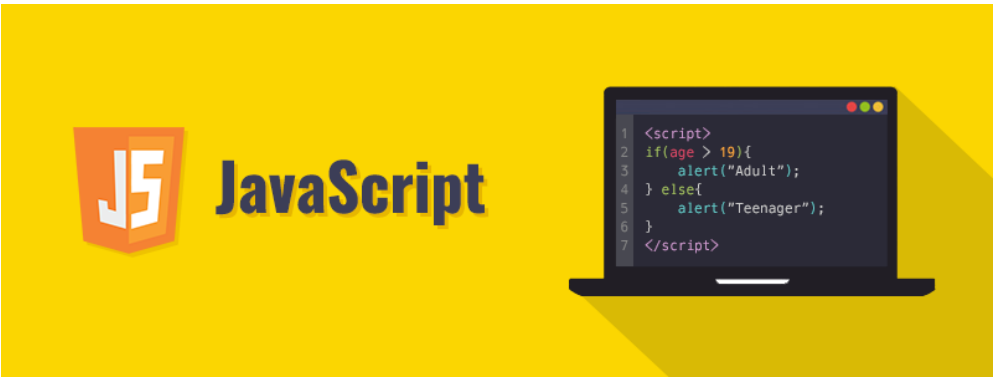
JavaScript Foundation – Training Day-6



Course content and duration:

Duration: 16 hours | Schedule: 8 days @ 2 hours/day

S. No	Day	Module	Topics
1	WED (10/12/2025)	Module 1: Introduction to JavaScript and Basics	JavaScript Overview, Syntax, and Variables
2	FRI (12/12/2025)	Module 2: Control Flow and Loops	Conditional Statements and Iteration
3	MON (15/12/2025)	Module 3: Functions and Scope	Function Declaration, Expressions, and Scope
4	TUE (16/12/2025)	Module 4: Arrays and Array Methods	Array Manipulation and Higher-Order Functions
5	WED (17/12/2025)	Module 5: Objects and Object-Oriented Programming	Objects, Properties, Methods, and Prototypes
6	THU (18/12/2025)	Module 6: DOM Manipulation and Events	Document Object Model and Event Handling
7	FRI (19/12/2025)	Module 7: Asynchronous JavaScript	Callbacks, Promises, and Async/Await
8	MON (22/12/2025)	Module 8: ES6+ Features and Best Practices	Modern JavaScript Features and Code Quality

Day6: Document Object Model and Event Handling (JavaScript)

Contents
Understanding the DOM tree structure
Selecting elements (getElementById, querySelector)
Modifying element content and attributes
Creating and removing elements
Event listeners and event handling
Event object and event propagation
Common events (click, submit, keypress, load)
Form handling and validation

Understanding the DOM Tree Structure 🌳

What is the DOM?

DOM (Document Object Model) is a **tree-like representation** of an HTML document created by the browser.

- When a browser loads an HTML page:
    1. It reads the HTML
    2. Converts it into a **DOM tree**
    3. JavaScript interacts with this tree to **read, change, add, or remove elements**

👉 **JavaScript does NOT directly work with HTML files**
👉 It works with the **DOM in memory**

Why is it called a "Tree"?

Because HTML elements have a **parent–child relationship**, just like a tree.

Example HTML:

```
<html>
  <head>
    <title>My Page</title>
  </head>
  <body>
    <h1>Hello</h1>
    <p>Welcome</p>
  </body>
</html>
```

DOM Tree Structure:

Document

└── html

  ├── head

  │ └── title

  │ └── "My Page"

  └── body


```
├── h1
│    └── "Hello"
└── p
     └── "Welcome"
```

DOM Nodes – Everything is a Node

In the DOM, **everything is a node**:

Node Type	**Example**
Document node	document
Element node	, <p>
Text node	"Hello"
Attribute node	class="box"
Comment node	<!-- comment -->

Example:

```
<p class="msg">Hello</p>
```

DOM Nodes:

- p → element node
- class="msg" → attribute node
- "Hello" → text node

Parent, Child & Sibling Relationships

DOM nodes are related like family members.

```
<ul>
  <li>One</li>
  <li>Two</li>
</ul>
```

- → **parent**
- → **children**
- One and Two → **siblings**

JavaScript access:

```
const ul = document.querySelector("ul");
ul.parentNode;        // parent
ul.children;          // HTMLCollection of li
ul.firstElementChild; // first li
ul.lastElementChild;  // last li
```

Document Object – The Root

The **entry point** to the DOM is the document object.

document

From document, you can access:

```
document.documentElement; // <html>
document.head;            // <head>
document.body;            // <body>
```

Visualizing DOM in Browser

Open **Chrome DevTools**:

- Right click → Inspect
- Elements tab = **DOM tree**
- Expand/collapse nodes to show hierarchy

👉 This directly shows the **tree structure**.

DOM vs HTML

HTML	DOM
Static file	Live structure in memory
Written by developer	Created by browser
Cannot change itself	Can be changed via JS

Example:

document.body.style.background = "lightblue";

➞ HTML file unchanged
➞ DOM updated instantly

Why DOM Tree Knowledge is Important

Understanding DOM structure helps to:

- Select correct elements
- Avoid wrong selectors
- Handle events efficiently
- Improve performance
- Debug UI issues faster

Simple Classroom Analogy 🏫

Think of DOM as:

- **Document** → School
- **Elements** → Classrooms
- **Text nodes** → Students
- **Attributes** → Student properties

Key Takeaways:

- DOM is a **tree representation** of HTML
- Every part of HTML becomes a **node**
- Parent–child–sibling relationships exist
- document is the root access point
- JavaScript manipulates DOM, not HTML files

Next topic: Selecting HTML Elements

Selecting HTML Elements (DOM Selection Methods) 🎯

Selecting elements is the **first and most important step** before manipulating the DOM. JavaScript provides **multiple methods** to select HTML elements from the DOM tree.

---

getElementById() – Select by ID (Most Common)

Syntax

document.getElementById("id")

Example

```
<p id="msg">Hello</p>
const para = document.getElementById("msg");
console.log(para.textContent); // Hello
```

Key Points

- Returns **ONE element**
- IDs must be **unique**
- Fastest method
- Returns null if not found

✅ Best for **single, unique elements**

---

getElementsByClassName() – Select by Class

Syntax

document.getElementsByClassName("className")

Example

```
<div class="box">A</div>
<div class="box">B</div>
const boxes = document.getElementsByClassName("box");
console.log(boxes[0].textContent); // A
```

Key Points

- Returns **HTMLCollection**
- Live collection (updates automatically)
- Access using index

---

getElementsByTagName() – Select by Tag

Syntax

document.getElementsByTagName("tag")

Example

```
<p>One</p>
<p>Two</p>
const paragraphs = document.getElementsByTagName("p");
console.log(paragraphs.length); // 2
```

Key Points

- Returns **HTMLCollection**
- Live collection
- Useful for bulk tag selection

---

querySelector() – Modern & Powerful ⭐

Syntax

document.querySelector("CSS selector")

Example

```
<div class="card">
  <h2>Title</h2>
</div>
const heading = document.querySelector(".card h2");
console.log(heading.textContent);
```

Key Points

- Returns **FIRST matching element**
- Uses **CSS selectors**
- Very flexible

✅ Recommended for most use cases

querySelectorAll() – Select Multiple Elements ⭐⭐

Syntax

document.querySelectorAll("CSS selector")

Example

```
<li class="item">One</li>
<li class="item">Two</li>
const items = document.querySelectorAll(".item");

items.forEach(item => {
  console.log(item.textContent);
});
```

Key Points

- Returns **NodeList**
- Not live (static)

Selecting Elements Inside Another Element

You can scope selection to a specific parent.

```
<div id="menu">
  <button>Save</button>
</div>
const menu = document.getElementById("menu");
const btn = menu.querySelector("button");
```

✅ Improves performance
✅ Avoids accidental matches

HTMLCollection vs NodeList (Important)

Feature	HTMLCollection	NodeList
Returned by	getElements*	querySelectorAll
Live	Yes	No
forEach	❌	✅
Array methods	❌	❌ (convert needed)

Common Beginner Mistakes ❌

```
document.getElementById("#box"); // ❌ wrong
document.querySelector("box");   // ❌ missing dot
Correct:
document.getElementById("box");  // ✅
document.querySelector(".box"); // ✅
```

Performance & Best Practices

- Use getElementById for unique elements
- Use querySelector(All) for complex selectors
- Store selected elements in variables
- Avoid repeated DOM queries

🔑 Quick Cheat Sheet

```
getElementById("id")         // single
getElementsByClassName("c")// multiple
getElementsByTagName("p")  // multiple
querySelector(".box")      // first match
querySelectorAll(".box")   // all matches
```

Next topic: Modifying element content and attributes

Modifying Element Content and Attributes 🛠️

Once elements are selected, the **next core skill** is modifying their **content, attributes, and styles** using JavaScript.

Modifying Text Content

textContent (Recommended ✅)

- Changes **only text**
- Safe from XSS (XSS, or **cross-site scripting**, is a web security vulnerability letting attackers inject malicious scripts (usually JavaScript) into web pages)
- Fast

```
<p id="msg">Hello</p>
const msg = document.getElementById("msg");
msg.textContent = "Welcome to JavaScript!";
Output:
<p>Welcome to JavaScript!</p>
```

---

innerText

- Respects CSS (hidden text not returned)
- Triggers reflow → slower

```
<p id="msg">Hello</p>
const msg = document.getElementById("msg");
msg.innerText = "Hello User";
📌 Output:
<p>Hello User</p>
```

---

innerHTML (Powerful but Dangerous ⚠️)

- Parses HTML
- Can insert tags

```
<p id="msg">Hello</p>
const msg = document.getElementById("msg");
msg.innerHTML = "<strong>Hello</strong> World";
📌 Output:
<p><strong>Hello</strong> World</p>
```

❌ **Security risk** if data comes from users
❌ Slower than textContent

📊 Comparison Table

Property	Allows HTML	Safe	Fast
textContent	❌	✅	✅
innerText	❌	⚠️	❌
innerHTML	✅	❌	⚠️

Modifying Attributes

getAttribute() / setAttribute()

```
<img id="logo" src="logo.png">
const img = document.getElementById("logo");

img.getAttribute("src"); // logo.png
img.setAttribute("src", "new.png");
img.setAttribute("alt", "Company");
```

---

removeAttribute()

```
<img id="logo" src="logo.png">
const img = document.getElementById("logo");

img.getAttribute("src"); // logo.png
img.setAttribute("src", "new.png");
img.setAttribute("alt", "Company");

img.removeAttribute("alt");
```

Direct Property Access (Preferred ⭐)

Some attributes have **direct JS properties**:

```
<input id="email" type="text">
const input = document.getElementById("email");

input.value = "test@example.com";
input.type = "email";
input.disabled = true;
```

📌 Faster & cleaner than setAttribute()

Modifying CSS Styles

Inline Styles

```
<p id="msg">Hello</p>
const msg = document.getElementById("msg");
msg.style.color = "red";
msg.style.fontSize = "20px";
msg.style.backgroundColor = "yellow";
```

⚠️ Inline styles override CSS

Class-based Styling (Best Practice ✅)

```
<p id="note" class="info">Note</p>
.highlight {
  color: white;
  background: blue;
}
const note = document.getElementById("note");
note.classList.add("highlight");
note.classList.remove("info");
note.classList.toggle("active");
note.classList.contains("active"); // true/false
```

✅ Clean
✅ Reusable
✅ Maintainable

Working with Form Elements

```
<input id="name">
const nameInput = document.getElementById("name");
nameInput.value = "Srikanth";
nameInput.placeholder = "Enter name";
nameInput.readOnly = true;
```

Modifying Data Attributes (data-*)

```
<button id="btn" data-user-id="101"></button>
const btn = document.getElementById("btn");

btn.dataset.userId;       // "101"
btn.dataset.userId = "202";
```

📌 Auto converts:

- data-user-id → dataset.userId

Common Mistakes ❌

```
element.textContent("<b>Hello</b>"); // ❌ wrong
element.innerHTML = userInput;       // ❌ unsafe
element.style = "color:red";         // ❌ overwrites style
Correct:
element.textContent = "Hello";        // ✅
element.classList.add("red");         // ✅
```

Performance & Best Practices

- Prefer textContent over innerHTML
- Modify classes instead of inline styles
- Avoid frequent DOM updates inside loops
- Cache DOM elements in variables

Summary

- Use textContent for text
- Use innerHTML carefully
- Prefer direct properties for attributes
- Use classList for styles
- Avoid inline CSS

Next topic: Creating and removing elements

Creating Elements in JavaScript

What is “Creating Elements”?

Creating elements means **dynamically generating new HTML elements using JavaScript**, instead of writing them directly in HTML.

JavaScript uses the **DOM API** to create elements while the page is already loaded.

Why do we create elements dynamically?

We create elements dynamically when:

- Content depends on **user interaction**
- Data comes from an **API or server**
- We want to **avoid page reloads**
- The number of elements is **not known in advance**

✓ Example: Adding items to a to-do list, chat messages, notifications, dynamic forms.

When should you use element creation?

Use it when:

- Buttons add items to a list
- Forms generate new UI sections
- Data is loaded asynchronously
- You want reusable, flexible UI

❌ Avoid when:

- Content is static and never changes
- HTML can be written directly

Core Methods for Creating Elements

✅ document.createElement()

📌 What?

Creates a **new HTML element node** in memory.

```
const p = document.createElement("p");
```

⚠️ Note: The element is **not visible yet**.

📌 Example: Create a paragraph

```
const para = document.createElement("p");
para.textContent = "Hello, I was created using JavaScript!";
```

✅ Setting attributes and content

```
para.id = "dynamicPara";
para.className = "highlight";
para.style.color = "blue";
```

✅ appendChild() / append()

📌 What?

Adds the created element to the DOM.

```
document.body.appendChild(para);
```

or

```
document.body.append(para);
```

Removing Elements in JavaScript

What is “Removing Elements”?

Removing elements means **deleting existing HTML elements from the DOM**.

Why remove elements?

We remove elements to:

- Clear old content
- Remove unnecessary UI
- Improve performance
- Respond to user actions

✓ Example: Removing a to-do item, closing a popup, deleting messages.

When should you remove elements?

Use it when:

- Users delete items
- Content must update dynamically
- UI state changes

Methods to Remove Elements

✅ **element.remove()**

📌 **What?**

Directly removes the element from the DOM.

element.remove();

✅ parent.removeChild(child)

📌 What?

Removes a child element using its parent.

```
parentElement.removeChild(childElement);
```

Best Practices

✅ Do This

✓ Use createElement instead of innerHTML
✓ Attach event listeners after creating elements
✓ Remove unused elements to avoid memory leaks

❌ Avoid This

❌ Replacing large HTML blocks unnecessarily
❌ Creating elements inside tight loops without cleanup
❌ Using innerHTML += repeatedly (performance issue)

Summary Table

Concept	What	Why	When
createElement	Creates new DOM node	Dynamic UI	Unknown content
appendChild	Adds node to DOM	Display content	After creation
remove	Deletes element	Cleanup UI	User action
removeChild	Removes child via parent	Compatibility	Controlled removal

---

Tip 🎯

“HTML builds the structure once,
JavaScript **keeps changing it**.”

Next topic: Event listeners and event handling

Event Listeners & Event Handling in JavaScript

What are Events?

An **event** is an action that happens in the browser, usually triggered by the **user** or the **browser itself**.

📌 **Common events:**

- Mouse: click, dblclick, mouseover
- Keyboard: keydown, keyup
- Form: submit, change, input
- Page: load, DOMContentLoaded

✓ Example: Clicking a button, typing in a textbox, submitting a form.

What is Event Handling?

Event handling is the process of **detecting an event and responding to it using JavaScript code**.

```
// Handle a click event
function handleClick() {
  console.log("Button clicked");
}
```

Why do we need event handling?

Without event handling:
❌ Web pages would be static
❌ No interaction with users

With event handling:
✓ Buttons work
✓ Forms submit data
✓ UI responds instantly

When should you use event handling?

Use event handling when:

- User interaction is required
- UI should react dynamically
- Data needs validation before submission

Ways to Handle Events

Inline Event Handlers (Not Recommended)

Event code written **directly in HTML attributes**.

```
<button onclick="sayHello()">Click</button>

<script>
function sayHello() {
  alert("Hello!");
}
</script>
```

Why is this discouraged?

❌ Mixes HTML and JavaScript
❌ Hard to maintain
❌ Not scalable

When can it be used?

✓ Quick demos
✓ Beginner learning
❌ Not for production apps

DOM Property Event Handlers

Assigning a function to an event property.

```
btn.onclick = function () {
  console.log("Clicked");
};
```

Why use it?

✓ Cleaner than inline
✓ Simple to understand

🔷 Limitation

❌ Only **one handler per event**

---

addEventListener() (Best Practice)

Attaches one or more event handlers to an element.

```
element.addEventListener("click", handler);
```

Multiple handlers for the **same event**

```
<button id="myBtn">Hover or Click</button>

<script>
  const myBtn = document.getElementById("myBtn");
  myBtn.addEventListener("click", () => {
    console.log("Button clicked");
  });
  myBtn.addEventListener("mouseenter", () => {
    console.log("Mouse entered button");
  });
  myBtn.addEventListener("mouseleave", () => {
    console.log("Mouse left button");
  });
</script>
```

Why use addEventListener?

✓ Multiple handlers allowed
✓ Separation of concerns
✓ Better control
✓ Can remove listeners later

🔷 When should you use it?

✓ Always in modern JavaScript
✓ Production code
✓ Framework-free apps

🧠 Example

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

<script>
const btn = document.getElementById("btn");

btn.addEventListener("click", function () {
  alert("Button clicked!");
});
</script>
```

Removing Event Listeners

Stops listening to an event.

```
element.removeEventListener("click", handler);
```

Why remove event listeners?

✓ Prevent memory leaks
✓ Disable UI when needed
✓ Improve performance

Important Rule

The **same function reference** must be used.

```
function greet() {
  console.log("Hello");
}

btn.addEventListener("click", greet);
btn.removeEventListener("click", greet);
```

Event Object

When an event occurs, JavaScript creates an **event object** containing details about the event.

```
btn.addEventListener("click", function (event) {
  console.log(event);
});
```

Useful properties

Property	Meaning
event.target	Element that triggered event
event.type	Event name
event.preventDefault()	Stops default behavior

🧠 Example

```
link.addEventListener("click", function (e) {
  e.preventDefault();
  console.log("Link click prevented");
});
```

Event Handling Flow

1️⃣ User interacts
2️⃣ Browser detects event
3️⃣ Event listener runs
4️⃣ JavaScript executes handler

Best Practices ⭐

✅ Do This

✓ Use addEventListener
✓ Keep event logic in JS
✓ Use named functions for reusable handlers

❌ Avoid This

❌ Inline event handlers
❌ Anonymous functions if removal is needed
❌ Too many listeners on many elements

Summary Table

Concept	What	Why	When
Event	User/browser action	Interaction	Any UI action
Event handling	Responding to events	Dynamic UI	User input
addEventListener	Attach handler	Best practice	Always
removeEventListener	Remove handler	Cleanup	Disable UI

⭐

“Events make the web **interactive**.
Event listeners tell JavaScript **how to react**.”

Next topic: Event object and event propagation

Event Object & Event Propagation in JavaScript

Event Object

What is the Event Object?

When an event occurs, JavaScript automatically creates an **event object** and passes it to the event handler.

This object contains **all information about the event**:

- What happened
- Where it happened
- Which element triggered it
- Mouse/keyboard details

Why do we need the event object?

Without the event object:
❌ You wouldn't know **which element** triggered the event
❌ You couldn't stop default behavior
❌ You couldn't read keyboard or mouse data

With it:
✓ You can build dynamic, reusable handlers
✓ You can control browser behavior

When should you use the event object?

Use it when:

- One handler is used for multiple elements
- You need mouse/keyboard info
- You want to stop default browser actions
- You want to control event flow

Common Event Object Properties

Property	What it means	Example use
event.target	Element that triggered event	Button clicks
event.currentTarget	Element listener attached to	Delegation
event.type	Event name	Logging
event.key	Key pressed	Keyboard events
event.clientX/Y	Mouse position	Drag & drop

🧠 Example – event.target

```
<button>Button 1</button>
<button>Button 2</button>

<script>
document.addEventListener("click", function (e) {
  console.log("Clicked:", e.target.textContent);
});
</script>
```

✓ One handler works for multiple buttons.

Preventing Default Behavior

Stops the browser's **default action**.

event.preventDefault();

Why use it?

✓ Stop form submission
✓ Stop page reload
✓ Stop link navigation

🧠 Example

```
<a href="https://google.com" id="link">Go</a>

<script>
link.addEventListener("click", function (e) {
  e.preventDefault();
  alert("Navigation prevented");
});
</script>
```

Event Propagation

What is Event Propagation?

Event propagation defines **how an event travels through the DOM tree** when it occurs.

There are **three phases**:

1️⃣ Capturing phase
2️⃣ Target phase
3️⃣ Bubbling phase

Why is propagation important?

Without understanding propagation:

❌ Unexpected multiple handlers run
❌ Bugs in nested elements
❌ Hard-to-debug UI behavior

Understanding it allows:

✓ Precise control
✓ Cleaner code
✓ Event delegation

When should you care about propagation?

You should care when:

- Elements are nested
- Multiple listeners exist
- You use event delegation
- You need performance optimization

Bubbling Phase (Default Behavior)

Event starts from the **target element** and bubbles **upwards** to parent elements.

```
<div id="parent">
  <button id="child">Click</button>
</div>

<script>
parent.addEventListener("click", () => console.log("Parent"));
child.addEventListener("click", () => console.log("Child"));
</script>
```

🧠 **Output:**

Child

Parent

✓ This is the default behavior.

Capturing Phase

Event flows from **root** → **target** before bubbling.

How to enable capturing?

Pass true as third parameter.

```
parent.addEventListener("click", handler, true);
```

🧠 Example

```
parent.addEventListener("click", () => console.log("Parent"), true);
child.addEventListener("click", () => console.log("Child"));
```

Output:

Parent

Child

Stopping Event Propagation

Stops event from moving further in the DOM.

```
event.stopPropagation();
```

Why use it?

✓ Prevent parent handlers
✓ Avoid duplicate logic
✓ Control complex UI

🧠 Example

```
child.addEventListener("click", function (e) {
  e.stopPropagation();
  console.log("Child clicked only");
});
```

Event Delegation (Real-World Use Case)

Attaching **one event listener to a parent** instead of many children.

Why use event delegation?

✓ Better performance
✓ Handles dynamically added elements
✓ Less memory usage

When should you use it?

✓ Large lists
✓ Dynamic content
✓ Unknown number of children

🧠 Example – List Click Handling

```
<ul id="list">
  <li>Item 1</li>
  <li>Item 2</li>
</ul>
<script>
list.addEventListener("click", function (e) {
  if (e.target.tagName === "LI") {
    console.log("Clicked:", e.target.textContent);
  }
});
</script>
```

---

9 Summary Table

Concept	What	Why	When
Event object	Info about event	Control behavior	Dynamic handling
preventDefault	Stops default action	Custom logic	Forms, links
Bubbling	Event goes upward	Default flow	Nested UI
Capturing	Event goes downward	Control order	Advanced cases
stopPropagation	Stops flow	Prevent duplicates	Complex UI
Event delegation	One listener for many	Performance	Large lists

"The event object tells **what happened**.
Propagation tells **where it travels**."

Next topic: Common events (click, submit, keypress, load)

Common Events & Event Methods in JavaScript

Common Events

---

click Event

What is the click event?

The click event fires when a user **clicks on an element** using a mouse, touch, or keyboard (Enter/Space).

Why do we use click?

To:

- Trigger actions
- Handle buttons
- Toggle UI
- Perform user-driven logic

Without click, buttons would do nothing.

When should you use click?

Use it when:

- A button is pressed
- An icon is clicked
- A menu is toggled
- A card or list item is selected

🧠 Example

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

<script>
btn.addEventListener("click", () => {
  alert("Button clicked");
});
</script>
```

submit Event

What is the submit event?

The submit event fires when a **form is submitted**, either by:

- Clicking a submit button
- Pressing Enter inside an input

Why do we use submit?

To:

- Validate form data
- Prevent page reload
- Send data using JavaScript
- Control form submission behavior

When should you use submit?

Use it when:

- Validating forms
- Sending data via AJAX / Fetch
- Preventing default form reload

🧠 Example

```
<form id="myForm">
  <input required />
  <button type="submit">Submit</button>
</form>

<script>
myForm.addEventListener("submit", function (e) {
  e.preventDefault();
  alert("Form submitted using JS");
});
</script>
```

---

keyboard events (keypress / keydown / keyup)

What are keyboard events?

Keyboard events fire when the user interacts with the keyboard.

Event	When it fires
keydown	Key is pressed down
keypress	Key produces a character (deprecated)
keyup	Key is released

⚠️ keypress is **deprecated** → use keydown instead.

Why do we use keyboard events?

To:

- Capture user input
- Build shortcuts
- Validate typing
- Create games or editors

When should you use them?

Use them when:

- Detecting Enter/Escape
- Real-time input handling
- Keyboard navigation

🧠 **Example**

```
<input id="input" placeholder="Type something">

<script>
input.addEventListener("keydown", function (e) {
  console.log("Key pressed:", e.key);
});
</script>
```

---

load Event

What is the load event?

The load event fires when:

- Page
- Images
- Stylesheets
- External resources

are **fully loaded**.

Why do we use load?

To:

- Ensure DOM & assets are ready
- Run code after page load
- Avoid accessing undefined elements

When should you use load?

Use it when:

- Working with images
- Measuring layout sizes
- Initializing heavy scripts

🧠 **Example**

```
window.addEventListener("load", () => {
  console.log("Page fully loaded");
});
```

---

stopImmediatePropagation()

Stops:

- Bubbling
- AND other handlers on the same element

When to use?

“Use stopImmediatePropagation() when one event handler must completely take control and prevent any other handlers on the same element and its parents from running.”

🧠 **Example**

```
btn.addEventListener("click", e => {
  e.stopImmediatePropagation();
});
```

once option (Modern JS)

Runs event handler **only once**.

Why use it?

✓ Cleaner than manual removal

✓ Avoid repeated execution

🧠 **Example**

```
btn.addEventListener("click", () => {
  alert("Runs once");
}, { once: true });
```

How { once: true } will work

1. The click handler runs **only one time**
2. After the first click, the event listener is **automatically removed**
3. No need to call removeEventListener()

Summary Table

Event / Method	What	Why	When
click	Mouse click	Trigger actions	Buttons
submit	Form submission	Validate data	Forms
keydown	Key pressed	Keyboard input	Shortcuts
load	Page loaded	Safe execution	Initialization

Next topic: Form Handling & Validation in JavaScript

Form Handling & Validation in JavaScript

What is Form Handling?

Form handling is the process of:

- Capturing user input from form fields
- Processing the data using JavaScript
- Submitting or rejecting the data based on rules

Forms are the **primary way users send data** to applications.

Why do we need form handling?

Without form handling:
❌ Invalid or empty data is submitted
❌ Poor user experience
❌ Security risks

With form handling:
✓ Clean, structured data
✓ Real-time feedback
✓ Controlled submissions

When should you use form handling?

Use form handling when:

- Collecting user information
- Submitting login/registration data
- Sending data to APIs
- Performing client-side checks

Accessing Form Elements

JavaScript can access form fields using DOM methods.

```
const form = document.getElementById("myForm");
const email = document.getElementById("email");
```

Why to access Form Elements?

To:

- Read user input
- Validate data
- Modify values dynamically

When?

Whenever user input needs to be processed.

🧠 Example

```
console.log(email.value);
```

What is Form Validation?

Form validation ensures that user input meets **defined rules** before submission.

✓ Required fields
✓ Correct formats
✓ Length limits

Why is validation important?

Without validation:

❌ Empty submissions
❌ Invalid data
❌ Poor UX
❌ Security vulnerabilities

When should validation happen?

✓ Before form submission
✓ While typing (real-time validation)

Client-Side vs Server-Side Validation

Type	What	Why	When
Client-side	JS in browser	Fast feedback	First line of defense
Server-side	Backend validation	Security	Always required

⚠️ **Client-side validation is NOT enough alone.**

Preventing Default Form Submission

Stops the browser's default form submit behavior.

```
event.preventDefault();
```

Why?

✓ Prevent page reload
✓ Validate using JavaScript
✓ Send data via AJAX / Fetch

When?

Always when handling forms using JavaScript.

🧠 Example

```
form.addEventListener("submit", e => {
  e.preventDefault();
});
```

Basic Validation Techniques

Required Field Validation

Checks if input is empty.

🧠 Example

```
if (username.value.trim() === "") {
  alert("Username is required");
}
```

Length Validation

Ensures minimum or maximum characters.

🧠 Example

```
if (password.value.length < 6) {
  alert("Password must be at least 6 characters");
}
```

Pattern (Regex) Validation

Validates format (email, phone, etc.)

🧠 Example – Email Validation

```
const emailPattern = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;

if (!emailPattern.test(email.value)) {
  alert("Invalid email format");
}
```

HTML5 Built-in Validation (Very Important)

HTML provides built-in validation attributes.

```
<input type="email" required minlength="5">
```

Why use it?

✓ No JavaScript needed
✓ Fast & accessible
✓ Browser-handled

When?

For simple validations.

Displaying Validation Errors (User-Friendly)

Showing error messages near inputs.

🧠 Example

```
<span id="error"></span>

<script>
error.textContent = "Invalid input";
error.style.color = "red";
</script>
```

Best Practices ⭐

✅ Do This

✓ Validate on submit AND on input
✓ Show clear error messages
✓ Use HTML5 + JavaScript together

❌ Avoid This

❌ Relying only on client-side validation
❌ Alert-only error messages
❌ Submitting without validation

🔑 Summary Table

Concept	What	Why	When
Form handling	Processing user input	Data control	Forms
Validation	Data correctness	UX & security	Before submit
preventDefault	Stops reload	JS handling	JS forms
Required check	Empty fields	Data integrity	Mandatory inputs
Pattern check	Format validation	Correct data	Emails, phones