Week 7 - Javascript

Created by Dakshil Gorasiya

Variable

JavaScript provides **three ways** to declare variables: `var`, `let`, and `const`.

Each behaves differently in terms of scope, hoisting, and reassignment.

1. `var` — Function-Scoped / Globally-Scoped

- `var` is **function-scoped** (or global if declared outside a function).
- Ignores block scope (`{ }`).
- Can be **redeclared** and **reassigned**.
- Gets hoisted with default value `undefined`.

Example:

```
var name = "Alice";
console.log("Using var:", name); // alice

{
  var name = "Bob"; // Overwrites outer variable
  console.log("Using var inside block:", name); // bob
}

console.log("Using var outside block after reassignment:", name); // bob
```

2. `let` — Block-Scoped

- `let` is **block-scoped** (`{ }`).
- Cannot be redeclared in the same scope.
- Can be **reassigned**.
- Hoisted but not initialized (Temporal Dead Zone).

Example:

```
let age = 25;
console.log("Using let:", age); // 25
```

```
{
  let age = 30; // Different variable (block-scoped)
  console.log("Using let inside block:", age); // 30
}

console.log("Using let outside block:", age); // 25
```

3. `const` — Block-Scoped & Read-Only

- Block-scoped like `let`.
- **Must be initialized** during declaration.
- **Cannot be reassigned**.
- The value inside a const object/array can still be mutable.

Example:

```
const country = "USA";
console.log("Using const:", country);
```

Hoisting & Global Scope

Hoisting of Variables

JavaScript "hoists" variable declarations, but **not** their initial values.

`var` Hoisting

```
console.log(a); // undefined
console.log(window.a); // undefined
var a = 10;
```

- `var a` is hoisted to the top of the scope.
- Its value is set to `undefined` until the assignment happens.
- `var` becomes a **property of the global object** (`window.a` in browsers).

`let` and `const` Hoisting

```
// console.log(b); // ReferenceError: Cannot access 'b' before initialization
let b = 20;
```

- The declaration is hoisted but not initialized.
- They stay in the **Temporal Dead Zone (TDZ)** until execution reaches their line.
- Accessing them early results in **ReferenceError**, not `undefined`.

Undeclared Variables

```
c = 30; // Implicit global variable
console.log(c); // Accessible globally
```

- Assigning to a variable without `var`, `let`, or `const` creates a **global variable**.
- This applies even inside functions.

  Not recommended — leads to accidental globals.

Hoisting of Functions

JavaScript treats two kinds of functions differently:

Function Declaration

```
fun1(); // Works

function fun1() {
  console.log("Inside fun1");
}
```

- Fully hoisted with function body.
- Can be called **before** the declaration.

Function Expression

```
// fun2(); // ReferenceError: Cannot access 'fun2' before initialization

fun2 = function () {
  console.log("Inside fun2");
};
```

- Behaves like a variable assignment.
- Not hoisted with its function body.

Data Types

JavaScript has two main categories of data types:

- **Primitive (stack memory)**
- **Non-primitive / Reference types (heap memory)**

1. Primitive Data Types

Primitive values are **immutable** and stored directly in memory (stack).

1. Number

```
let a = 10;
let b = 10.2;
```

2. String

```
let c = "abc";
let d = "abc";
let e = `abc${a}`; // string interpolation
```

3. Boolean

```
let f = false;
```

4. Undefined

Declared but not assigned any value.

```
let g;
```

5. Null

Intentional empty value.

```
let h = null;
```

6. BigInt

Used for very large integers.

```
let i = 9007199254740991n;
```

7. Symbol

Unique and immutable identifier.

```
let j = Symbol("id");
```

2. Non-Primitive (Reference) Data Types

Stored in heap memory; variables hold a **reference**.

Object

```
let k = {
  name: "dakshil",
  college: "bvm",
};
```

Array

```
let l = [1, 2, 3];
```

Function

```
let m = function () {
  console.log("FUNCTION");
};
```

Memory Allocation Difference

Primitive — values are copied

```
let name = "a";
let anotherName = name;

anotherName = "b";

console.log(name); // a
console.log(anotherName); // b
```

Changing one doesn't affect the other.

Reference Types — reference (address) is copied

```
let ob = { name: "a" };
let anotherOb = ob;

anotherOb.name = "b";

console.log(ob);        // { name: "b" }
console.log(anotherOb); // { name: "b" }
```

Both variables point to the **same object** in memory.

Type Conversion

String → Number

```
let s = "5";
let num = Number(s); // 5
```

Undefined → Number

```
let a = undefined;
num = Number(a); // NaN
```

Type of NaN

```
console.log(typeof NaN); // "number"
```

Other Conversion

Value	Number() Output
`"123"`	123
`""`	0
`"abc"`	NaN
`null`	0
`undefined`	NaN
`true`	1
`false`	0

Boolean Conversion

```
let boolean = Boolean("");
console.log(boolean); // false
```

Falsy Values in JS

- false
- undefined
- null
- 0
- NaN
- "" (empty string)

String + Number Behavior

JavaScript evaluates left to right.

```
console.log("1" + 1 + 1); // "111"
// string → string → string

console.log(1 + 1 + "1"); // "21"
// number + number = 2 → "2" + "1" = "21"
```

Comparison: `==` vs `===`

`==` → loose comparison (type conversion allowed)

`===` → strict comparison (no type conversion)

```
console.log(2 == "2");  // true
console.log(2 === "2"); // false
```

Control Flow

`switch` statement

Use `switch` to branch on discrete values. Don't forget `break` to avoid fall-through.

```
const month = 3;

switch (month) {
  case 1:
    console.log("January");
    break;
  case 2:
    console.log("February");
    break;
  case 3:
    console.log("March");
    break;
  case 4:
    console.log("April");
    break;
  default:
    console.log("Default");
    break;
}
```

Single-line `if` and the comma

If you write an `if` without braces, **only the first statement** after it is controlled by the `if`.

```
const balance = 1000;

// WARNING: only the first statement is conditional
if (balance > 500) console.log("test"), console.log("test2");
// parsed as:
// if (balance > 500) console.log("test");
// console.log("test2");  // runs regardless of condition
```

Always use braces for clarity:

```
if (balance > 500) {
  console.log("test");
  console.log("test2");
}
```

Truthy / Falsy values

Falsy values (evaluate to `false` in boolean context):

- `false`
- `0`
- `0n` (BigInt zero)
- `""` (empty string)
- `null`
- `undefined`
- `NaN`

Truthy examples:

- `"0"`, `"false"`, `" "` (non-empty strings)
- `[]` (empty array)
- `{}` (empty object)
- `function(){}` (functions)

> Anything non-empty or non-zero is generally truthy.

Check empty array / object

```
const userEmail = [];
if (userEmail.length === 0) {
  console.log("Array is empty");
}

const emptyObj = {};
if (Object.keys(emptyObj).length === 0) {
  console.log("Object is empty");
}
```

Nullish coalescing operator `??`

`??` returns the first operand that is **not** `null` or `undefined`.

```
let val1;

val1 = 5 ?? 10;        // 5
val1 = null ?? 10;      // 10
```

```
val1 = undefined ?? 15; // 15

val1 = null ?? 10 ?? 20;        // 10
val1 = null ?? undefined ?? 11; // 11
```

Use `??` when you want to treat `0`, `""`, or `false` as valid values (unlike `||` which treats them as falsy).

Logical OR Operator `||`

`||` returns the first non `falsy` value

```
val1 = 5 || 10; // 5
val1 = 0 || 10; // 10
val1 = "" || 15; // 15
val1 = null || 10 || 20; // 10
val1 = undefined || 0 || 11; // 11
```

Ternary operator

Short inline `if`:

```
// condition ? valueIfTrue : valueIfFalse
const iceTeaPrice = 100;
iceTeaPrice <= 80 ? console.log("Less than 80") : console.log("More than 80");
```

Loop

Array Iteration

1) Classic `for` loop

Useful when you need full control over index and step.

```
for (let i = 0; i < arr.length; i++) {
  console.log(`Index: ${i}, Value: ${arr[i]}`);
}
```

2) `for...of` loop

Best for iterating **values** of arrays.

```
for (let value of arr) {
  console.log(value);
}
```

- Gives **values**

- Cleaner than the classic loop

3) `for...in` loop (Not ideal for arrays)

Iterates **keys (indexes)**.

```
for (let index in arr) {
  console.log(arr[index]);
}
```

4) `forEach()`

Clean and functional-style iteration.

```
arr.forEach((value, index) => {
  console.log(`Index: ${index}, Value: ${value}`);
});
```

Object Iteration

1) `for...in` loop

Iterates over **keys**.

```
for (let key in obj) {
  console.log(`Key: ${key}, Value: ${obj[key]}`);
}
```

2) Using `Object.keys()` + `forEach()`

Gives an array of keys.

```
Object.keys(obj).forEach((key) => {
  console.log(`Key: ${key}, Value: ${obj[key]}`);
});
```

3) Using `Object.entries()` + `for...of`

Gives `[key, value]` pairs.

```
for (let [key, value] of Object.entries(obj)) {
  console.log(`Key: ${key}, Value: ${value}`);
}
```

While & Do...While Loops

While Loop

Runs when condition is **true**.

```
while (count < 5) {
  console.log(count);
  count++;
}
```

Do...While Loop

Executes **at least once**, even if condition is false initially.

```
let num = 0;
do {
  console.log(num);
  num++;
} while (num < 5);
```

`break` and `continue`

`continue` → Skip current iteration

`break` → Exit loop immediately

```
for (let i = 0; i < 10; i++) {
  if (i % 2 === 0) continue; // skip even numbers

  console.log(`Odd Number: ${i}`);

  if (i === 7) break; // stop loop
}
```

Objects

Using Symbols as Object Keys

Symbols create **unique, non-enumerable** keys.

```
const mySyn = Symbol("key1");

const user = {
  name: "dakshil",
  "full name": "gorasiya dakshil r",
  [mySyn]: "mykey1",  // symbol key (needs brackets)
};
```

Why brackets?

- `mySyn: "mykey1"` would create a **string key** `"mySyn"`
- `[mySyn]: "mykey1"` correctly uses the **Symbol** itself as the key

Accessing Object Properties

```
console.log(user.name);
console.log(user["full name"]);
console.log(user[mySyn]); // using symbol
console.log(user.lastLoginDays[0]);
```

Three access methods:

- Dot notation → `user.name`
- Bracket notation for multi-word keys → `user["full name"]`
- Bracket notation for symbol keys → `user[mySyn]`

Object.freeze() — Make an Object Immutable

```
Object.freeze(user);
user.name = "abc"; // will NOT work
```

- After freezing, no properties can be **added, changed, or deleted**.

Adding Methods to Objects

```
user.greeting = function () {
  console.log("hello user");
};
```

Creating Object Using `new Object()`

```
const ob = new Object();
ob.fun = function () {
  console.log("hello from ob");
};
ob.fun();
```

- Same as `{}`, but more explicit.

Merging Objects

1) Using `Object.assign()`

```
const obj3 = Object.assign({}, obj1, obj2);
```

2) Using Spread Operator `...`

```
const obj4 = { ...obj1, ...obj2 };
```

- Cleaner, modern syntax.

Object Utility Methods

```
Object.keys(obj3);    // ['1', '2', '3', '4']
Object.values(obj3);  // ['a', 'b', 'a', 'b']
Object.entries(obj3); // [['1','a'],['2','b'],...]
obj3.hasOwnProperty("1"); // true
```

- `keys` → returns array of property names
- `values` → returns array of values
- `entries` → returns key-value pairs
- `hasOwnProperty` → checks if a key exists

Object Destructuring

Extract values directly into variables.

```
const obj1 = {
  1: "a",
  2: "b",
};

const { 1: firstProp, 2: secondProp } = obj1;

console.log(firstProp);   // a
console.log(secondProp);  // b
```

- Works even with number-like keys.
- Syntax: `{ keyName: newVariableName }`

Functions

Function Hoisting

JavaScript hoists function declarations **fully**, but not function expressions or arrow functions.

```
console.log(add(2, 3)); // ✅ Works
// console.log(sub(5, 2)); // ❌ Error
// console.log(mul(2, 3)); // ❌ Error
```

Three Ways to Declare Functions

1) Function Declaration

Fully hoisted.

```
function add(a, b) {
  return a + b;
}
```

2) Function Expression

Stored in a variable.

```
let sub = function (a, b) {
  return a - b;
};
```

Not callable before definition.

3) Arrow Function

Shorter syntax.

```
let mul = (a, b) => {
  return a * b;
};
```

IIFE — Immediately Invoked Function Expression

Runs as soon as it is created.

```
(function () {
  console.log("IIFE function executed");
})();
```

IIFEs are used to:

- Avoid polluting global namespace
- Run setup code immediately

Scope & Closures

A closure is created when an inner function remembers variables from its outer function even after the outer function has returned.

```
function outer() {
  let outerVar = "I am from outer function";

  function inner() {
    console.log(outerVar); // outerVar is still accessible
  }

  return inner;
}
```

```
let innerFunc = outer();
innerFunc(); // "I am from outer function"
```

Closure Key Points:

- Inner functions can access outer function variables
- Those variables stay “alive” even after outer function execution
- Useful for data privacy, function factories, and more

Types of arguments

1. Pass by Value

- Applies to **primitive types** (number, string, boolean, etc.)
- A copy of the value is passed to the function.
- Changing it **does not affect** the original variable.

2. Pass by Reference

- Applies to **objects, arrays, functions**.
- A reference is passed, so modifying the object **changes the original**.

3. Default Parameters

- Allows setting default values for function parameters.

```
function greet(name = "Guest") {}
```

4. Rest Parameter (`...args`)

- Collects unlimited arguments into an **array**.
- Useful when the number of inputs is unknown.

```
function sum(...args) {}
```

5. `arguments` Object

- Array-like object containing all passed arguments.
- Not available in arrow functions.
- Does **not** support array methods directly.

```
function show() { console.log(arguments); }
```

6. Named Arguments via Object Destructuring

- Allows cleaner parameter handling when passing many values.
- Order does not matter.

```
function createUser({ name, age, city }) {}
```

7. call, apply, bind

Used to manually set `this` inside a function.

`call(thisArg, arg1, arg2, ...)`

- Passes arguments **individually**.

```
intro.call(user, 21, "anand");
```

`apply(thisArg, [args])`

- Passes arguments as an **array**.

```
intro.apply(user, [21, "anand"]);
```

`bind(thisArg, arg1, arg2, ...)`

- Returns a **new function** with `this` permanently set.

```
const fn = intro.bind(user, 21, "anand");
fn();
```

Popup boxes

1. `alert()`

Displays a **message box** with an OK button.

Used for **notifications** or warnings.

Always returns **undefined**.

2. `confirm()`

- Shows a dialog with **OK** and **Cancel** buttons.
- Returns:
  - `true` → user clicked **OK**
  - `false` → user clicked **Cancel**

3. `prompt()`

Displays a dialog with:

- A message
- A text input field
- OK / Cancel buttons

Returns:

- The **entered text**

- `null` → if user clicked **Cancel**

DOM Manipulation

Selecting Elements

`querySelector()`

Returns the **first element** that matches a CSS selector.

```
h1tag = document.querySelector(".box h1");
```

`querySelectorAll(selector)` — returns *all* matches as a `NodeList`.

```
const listItems = document.querySelectorAll(".list");
```

`getElementById()`

Selects an element using its **id**.

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

`getElementsByClassName()`

Returns an **HTMLCollection** (array-like) of elements.

```
optionList = document.getElementsByClassName("optionsList")[0];
```

First / Last child

`firstElementChild` / **`lastElementChild`** — first/last **element** node.

```
container.firstElementChild;
container.lastElementChild;
```

`firstChild` / **`lastChild`** — first/last **child node** (could be text).

```
container.firstChild;
container.lastChild;
```

Parent:

- `parentNode` and `parentElement` — usually return the same parent element.

  ```
  firstDiv.parentNode;
  firstDiv.parentElement;
  ```

Siblings:

- `nextElementSibling` / `previousElementSibling` — next/previous **element** nodes.
- `nextSibling` / `previousSibling` — next/previous **child nodes** (may be text).

```
firstDiv.nextElementSibling;
firstDiv.nextSibling;
firstDiv.previousElementSibling;
firstDiv.previousSibling;
```

`closest(selector)`

- Finds the **closest ancestor** (including the element itself) that matches the selector.

```
paragraph.closest("div");
paragraph.closest("#container");
```

Changing Text Content

`innerText`

Sets/gets visible text only.

```
mainContent = document.getElementById("mainContent");
mainContent.innerText = "Lorem ipsum dolor sit amet...";
```

`innerHTML`

Replaces HTML inside the element.

```
mainContent.innerHTML = "<strong>Lorem ipsum dolor sit amet</strong>...";
```

> ⚠️ Use innerHTML carefully to avoid XSS.

Changing CSS with JavaScript

You can modify styles directly through `.style`:

```
button.style.padding = "10px 20px";
button.style.fontSize = "16px";
button.style.cursor = "pointer";
button.style.borderRadius = "5px";
```

Adding Event Listeners

Use `.addEventListener()` to react to user actions:

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

Advantages:

- Multiple listeners allowed
- Keeps HTML clean

Creating New Elements

`createElement()`

Creates a new element dynamically.

```
newLi = document.createElement("li");
newLi.innerText = "Option 4";
```

Appending a Child

```
optionList.appendChild(newLi);
```

Adds the new `<li>` at the end of the `ul`.

Removing Elements

Remove a specific child:

```
optionList.removeChild(optionList.children[1]); // Removes second item
```

Creating & Adding a Link

```
googleLink = document.createElement("a");
googleLink.setAttribute("href", "https://www.google.com");
googleLink.setAttribute("target", "_blank");
googleLink.innerText = "Go to Google";

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

Key notes:

- `setAttribute()` is used to set attributes like `href`, `target`, `id`, etc.
- `_blank` opens the link in a new tab.

Attributes

1. `getAttribute(name)`

- Used to read the value of an HTML attribute.
- Returns the attribute value **or** `null` if not present.

2. `setAttribute(name, value)`

- Creates a new attribute or updates an existing one.

3. `removeAttribute(name)`

- Deletes an attribute from an element.
- After removal, `getAttribute` will return `null`.

4. `hasAttribute(name)`

- Checks whether an attribute **exists**.
- Returns `true` or `false`.

Difference Between `children` and `childNodes`

1. `children`

- Returns **only element nodes** (HTML tags).
- Ignores text nodes, whitespace, comments.
- Gives an **HTMLCollection** (live list).

2. `childNodes`

- Returns **all types of nodes**, including:
    - Element nodes
    - Text nodes (spaces, newlines)
    - Comment nodes
- Gives a **NodeList**

Events

Event Bubbling

- Default behavior of events in JavaScript.
- Event flow:

  Child → Parent → Document
- When an event occurs on a child element, it automatically propagates to its parent elements.
- Commonly used in event delegation.

Example Flow:

Button → Div → Body → Document

Event Capturing

- Opposite of event bubbling.
- Event flow:

  Parent → Child
- Must be enabled manually using `{ capture: true }`.

Order of execution

If `div>button` and both have both bubble and capture listeners then order of execution will be

1. div capture
2. button capture
3. button bubble
4. div bubble

`event.target` vs `event.currentTarget`

`event.target`

- Refers to the **actual element that triggered the event**.
- Changes depending on where the user clicks.

`event.currentTarget`

- Refers to the **element on which the event listener is attached**.
- Always remains constant for that handler.

Event Delegation

- Technique where a **single parent element** handles events of multiple child elements.
- Uses the concept of **event bubbling**.

Instead of:

- Adding many event listeners to child elements

We add One listener to the parent

Advantages:

- Better performance
- Less memory usage
- Works with dynamically created elements

`addEventListener()`

- Modern method for attaching events.
- Allows:
  - Multiple event handlers on the same element
  - Capturing or bubbling control
  - Better separation of logic and UI

Syntax supports:

- Bubbling (default)
- Capturing (`{ capture: true }`)

`stopPropagation()`

It prevents the event from reaching parent elements but allows other listeners on the same element to execute.

`stopImmediatePropagation()`

It stops the event completely — no parent handlers and no remaining handlers on the same element will run.

Important events

Mouse events

Event Name	When It Fires
click	When the user presses and releases the left mouse button on an element.
dblclick	When the user performs two quick click actions on the same element.
mousedown	When the user presses any mouse button down.
mouseup	When the user releases a pressed mouse button.
mousemove	Whenever the mouse pointer moves over the element.
mouseenter	When the mouse pointer enters the element **(does NOT bubble)**.
mouseleave	When the mouse pointer leaves the element **(does NOT bubble)**.
mouseover	When mouse enters the element **or any of its children** (bubbles).
mouseout	When mouse leaves the element **or moves to its children** (bubbles).
contextmenu	When the user right-clicks (before the browser shows the context menu).
wheel	When the user scrolls the mouse wheel over the element.
dragstart	When dragging begins on a draggable element.
drag	Fires continuously while dragging.
dragend	When dragging stops (mouse release).
dragenter	When a draggable element enters a drop target.
dragleave	When a draggable element leaves a drop target.
dragover	Fires repeatedly while hovered over a drop target (**must call preventDefault to allow dropping**).
drop	When a draggable element is released over a drop target.

Keyboard events

Event Name	When It Fires
keydown	When a key is pressed down (fires before the character appears). Repeats if the key is held.
keyup	When a pressed key is released.
keypress *(deprecated)*	When a **printable character** key is pressed (does NOT fire for keys like Shift, Ctrl, Escape).

Form events

Event Name	When It Fires
input	Whenever the value changes (every keystroke, paste, delete).
change	When the user changes the value **and then leaves the input** (blur).
focus	When the input or textarea receives focus (clicked or tabbed in).
blur	When the input or textarea loses focus.
beforeinput	Right *before* the value is modified (typing, paste, delete).
select	When the user selects text inside the input (drag/select, shift+arrow).
submit	When the form is submitted (button click, Enter key).
reset	When the form is reset (reset button or form.reset()).
formdata	When FormData is created during form submission (after submit event but before navigation).

Window events

Event Name	When It Fires
online	When the browser regains network connectivity.
offline	When the browser loses network connectivity.
focus	When the window/tab becomes active (clicked or switched to).
blur	When the window/tab loses focus (switch tab, open other app).
resize	When the browser window size changes.
scroll	When the window is scrolled (vertical or horizontal).
beforeunload	Right before the page is about to unload (refresh, close tab, navigate away). Allows showing a confirmation dialog.
unload	When the page is being unloaded from memory (cannot cancel).
DOMContentLoaded	When the DOM is fully parsed (no waiting for images/videos).
load	When the entire page and all resources have finished loading.
freeze	When the page is being frozen (put into BFCache or suspended).
resume	When the page is restored after being frozen (loaded from BFCache).

Media events

Event Name	When It Fires
loadstart	When the browser starts loading the video file.
progress	While the browser is downloading/buffering video data.
suspend	When the browser intentionally stops downloading data (e.g., enough buffered).
abort	When the loading was aborted (e.g., the user navigated away or changed source).
stalled	When the browser tries to download but no data is received.
loadedmetadata	When metadata like duration, width, height becomes available.
loadeddata	When the first frame of the video is loaded and ready to display.

Event Name	When It Fires
durationchange	When the video duration becomes known or changes.
canplay	Video is ready to play, but may still buffer later.
canplaythrough	Video is fully buffered enough to play to the end without stopping.
play	When playback is requested (click play or JS `.play()`).
playing	When the video actually starts playing (after buffering).
pause	When the video is paused.
ended	When the video reaches its end.
waiting	When playback stops because more data is needed (buffering).
seeking	When the user or JS jumps to a new playback time.
seeked	When the video finishes jumping to the new time.
timeupdate	Fires repeatedly while the video is playing or its time changes.
ratechange	When playback speed changes (e.g., 1× → 2×).
volumechange	When volume or mute/unmute changes.
emptied	When the video element is reset (e.g., changing to a new source).
error	When any loading or playback error occurs.

Clipboard events

Event Name	When It Fires
copy	When the user copies selected text (Ctrl+C, Command+C, or right-click → Copy).
cut	When the user cuts selected text (Ctrl+X, Command+X, or right-click → Cut).
paste	When the user pastes text into an editable element (Ctrl+V, Command+V, or right-click → Paste).

Document events

Event Name	When It Fires
DOMContentLoaded	When the HTML document is fully parsed (scripts may still be loading).
readystatechange	When `document.readyState` changes (`loading → interactive → complete`).
load	When the entire page and all resources (images, scripts, CSS) are fully loaded.
visibilitychange	When the tab becomes **visible** or **hidden** (switching tabs, minimizing window).
selectstart	When the user begins selecting text (mouse drag or shift+arrow).
fullscreenchange	When the document enters or exits fullscreen mode.
fullscreenerror	When a fullscreen request fails (permissions blocked, invalid element).
pointerlockchange	When the pointer becomes locked or unlocked (mouse captured by the page).
pointerlockerror	When request to lock the pointer fails.

Navigation events

Event Name	When It Fires
hashchange	When the part of the URL after `#` changes (e.g., `#abc` → `#xyz`). Fired even without a page reload.
popstate	When the user navigates through the history stack (Back, Forward) or when JS calls `history.back()` / `history.forward()`.

Custom events

1. Creating a Custom Event

Use `new CustomEvent(eventName, options)`

```
const loginEvent = new CustomEvent("userLoggedIn", {
  bubbles: true,
  detail: dataObject,   // custom data sent with the event
});
```

Key Options:

- `detail` → Payload you want to send with the event
- `bubbles` → Allows event to bubble up through DOM (default=false)
- `cancelable` → If `true`, listeners can call `event.preventDefault()` (default=false)

2. Dispatching a Custom Event

Trigger the event using:

```
document.dispatchEvent(loginEvent);
```

This will notify all listeners waiting for that event.

3. Listening for a Custom Event

Use `addEventListener` with the custom event name:

```
document.addEventListener("userLoggedIn", (e) => {
  const payload = e.detail;
});
```

Accessing Data:

Custom event data appears inside `e.detail`.

Cookie

- Small pieces of text stored by the browser.
- Used to remember user data between requests.
- Sent automatically in HTTP requests (depending on flags).
- Size limit: ~4 KB per cookie, ~20–50 cookies per domain.

To set cookie

```
document.cookie = "key=value; expires=DATE; path=/; secure; samesite=Lax";
```

To set multiple cookie use document.cookie multiple time but not try to set multiple cookie in same statement first will be saved and other will be treated as flag for that cookie

To read cookie

```
console.log(document.cookie);
```

To delete cookie set cookie with max-age=0 or expires to previous date

Cookie flags

1. expires

- Fixed date/time when the cookie should expire.
- Format: GMT string (HTTP-date).
- Example:
  `expires=Tue, 31 Dec 2025 23:59:59 GMT`

2. max-age

- Defines lifetime in **seconds**.
- Overrides `expires` if both exist.
- Example:
  `max-age=3600` (1 hour)

3. path

- Restricts cookie availability based on URL path.
- Common setting:
  `path=/` (available everywhere)

4. domain

- Defines which domain/subdomains can access the cookie.
- Example: `domain=example.com` → usable in `app.example.com`
- If omitted: cookie is bound to exact domain.

5. secure

- Cookie sent **only over HTTPS**.
- Protects from network sniffing.
- Should always be used in production.

6. httponly

- Cannot be accessed by JavaScript
- Must be set by server only
- Protects from **XSS** since JS cannot read it.
- Example server header:
  `Set-Cookie: session=abc; HttpOnly`

7. samesite

Controls cross-site cookie sending (important for CSRF).

Values:

- **Strict**

  Only sent on same-site requests (best security).
- **Lax**

  Sent on top-level navigations (default behavior).
- **None**

  Sent on cross-site requests → requires `secure`.

Setting cookie from server

To set cookie using server set Set-Cookie header is used

```
Set-Cookie: sessionId=abc123; Path=/; Max-Age=3600; Secure; HttpOnly; SameSite=Lax
```

CSRF (Cross site request forgery)

- Attack where a malicious site triggers actions on another site using the user's cookies.
- Example: perform banking transfer because cookies are automatically sent.

How Cookies Are Involved

- Browsers automatically attach cookies with requests → attacker can exploit this.

Prevention

- Use `SameSite=Lax` or `SameSite=Strict`.
- Use CSRF tokens in forms/POST requests.
- Validate Referer / Origin headers.

XSS (Cross site scripting)

- Attacker injects JavaScript that runs in your site.

How Cookies Are Involved

- XSS can steal cookies **unless they are HttpOnly**.

Prevention

- Use `HttpOnly` flag for sensitive cookies.
- Never insert raw HTML from user data.

Localstorage

- A browser storage mechanism for storing key–value data.
- Data persists **even after closing browser**, until manually cleared.
- Stores **string data only** (objects must be JSON-stringified).
- Available per **origin** (protocol + domain + port).
- If multiple tab is open locaStorage of all will be same

When to Use LocalStorage

Use LocalStorage for:

- Saving UI preferences (theme, layout)
- Caching lightweight API responses
- Remembering form data
- Storing shopping cart data (non-sensitive)
- Storing app state for offline apps

Not for:

- Passwords
- Authentication tokens
- Sensitive or personal data

localstorage apis

To store

```
localStorage.setItem(key, value);
```

Here key and value are string only

To get

```
localStorage.getItem(key);
```

To remove

```
localStorage.removeItem(key);
```

To clear entire localStorage

```
localStorage.clear();
```

Session Storage

- A browser storage mechanism similar to LocalStorage.
- Stores key–value pairs (string-based).
- Data persists **only for the duration of the tab session**.
- Data is cleared when:

    - The tab/window is closed
    - The browser restarts (in most cases)
    - The tab navigates to a different domain
- If multiple tab is open SessionStorage of each will be different
- Available per **origin** (protocol + domain + port).

When to Use SessionStorage

Use SessionStorage for **temporary and tab-specific** data like:

- Wizard/form step data
- Temporary user input
- In-progress checkout state
- Search filters or UI state (per tab)
- One-time flags

Avoid SessionStorage for:

- Authentication tokens (XSS risk)
- Data needed across tabs or after tab close
- Sensitive personal data

Session Storage apis

To set item

```
sessionStorage.setItem(key, value);
```

To get item

```
sessionStorage.getItem(key);
```

To delete item

```
sessionStorage.removeItem(key);
```

To clear

```
sessionStorage.clear();
```

Feature	**Cookies**	**localStorage**	**sessionStorage**
Storage Limit	~4 KB	~5–10 MB	~5–10 MB
Lifetime	Custom (via `expires` / `max-age`)	Permanent until cleared	Until tab/window is closed
Sent to Server Automatically	Yes (with every request)	No	No
Accessible from JavaScript	Yes (except HttpOnly cookies)	Yes	Yes
Secure Flag Support	Yes (`Secure`, `HttpOnly`, `SameSite`)	No	No
Use Cases	Authentication, sessions, CSRF tokens	Preferences, caching, persistent app data	Per-tab temporary data, form steps
Bound to	Domain + Path	Origin (protocol + domain + port)	Origin + **Tab**
Shared Across Tabs	Yes	Yes	No
Expires Automatically	If set	No	When tab closes
Server Can Set	Yes (`Set-Cookie` header)	No	No
Server Can Read	Yes (sent in request)	No	No