WEEK-7 NOTES

▼ How JavaScript works

- **JS engine**: V8 (Chrome, Node), SpiderMonkey (Firefox). It parses source → creates AST → compiles to optimized machine code (JIT).
- **Single-threaded**: There's one main thread executing JS code. But I/O/OS can be handled by background threads (browser or Node) — JS interacts via callbacks/promises.
- **Call stack**: Where function execution contexts are pushed/popped. Synchronous code uses the call stack.
- **Heap**: Memory for objects, closures.
- **Event loop**: Coordinates async tasks.
  - **Call stack** executes tasks.
  - **Task queue (macrotasks)**: setTimeout, setInterval, I/O callbacks.
  - **Microtask queue**: Promises `.then` / `queueMicrotask` — these run **before** the next macrotask when the stack is emptied.
- **Execution contexts**:
  - Global execution context created at program start.
  - Function execution contexts for each function call.
  - Each has its own **scope chain**, **variable environment**, `this` **binding**.

Execution order: parsing, hoisting, and creation/execution

phases

When JS runs a script or function it roughly does:

1. **Creation phase**: variables and function declarations are placed into memory.
   - **Function declarations** are hoisted with their body.
   - `var` **variables** are hoisted with `undefined`.
   - `let` **and** `const` are hoisted but stay in a *Temporal Dead Zone* until initialization.
2. **Execution phase**: code runs top to bottom, values assigned, functions invoked.

Example (hoisting):

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

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

hoistedFn(); // "hello"
function hoistedFn(){ console.log("hello"); }
```

▼ var-let-const

var

Key Characteristics

1. **Function-scoped**
   - If declared inside a function → available everywhere inside that function.
   - If declared inside a block (`if {}`, `for {}`) → NOT block-scoped (escapes the block).
2. **Hoisted**

- `var` declarations are hoisted to the top of their scope.
- During hoisting, JS sets them to `undefined`.

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

3. **Can be redeclared** in the same scope.

```
var a = 10;
 var a = 20; // allowed
```

4. **No Temporal Dead Zone (TDZ)**

   Using `var` before declaration does NOT throw an error — it gives `undefined`.

let

Key Characteristics

1. **Block-scoped**
   - Accessible only inside `{...}` where it is defined.
   - Blocks: if, for, while, switch, plain braces.
2. **Hoisted BUT in Temporal Dead Zone (TDZ)**
   - Variable exists in memory before execution…
   - But is NOT initialized → cannot be accessed until declaration line is executed.

```
console.log(b); //  ReferenceError
 let b = 10;
```

3. **Cannot be redeclared** in the same scope.

```
let x = 10;
 let x = 20; // error
```

4. **Can be reassigned.**

```
let a = 5;
 a = 10; // allowed
```

const

Key Characteristics

1. **Block-scoped** (like `let`)
2. **Hoisted but in TDZ** (like `let`)
3. **Must be initialized at declaration**

```
const a; // error (missing initializer)
```

4. **Cannot be reassigned**

```
const x = 10;
 x = 20; //  TypeError
```

5. **BUT objects & arrays can be mutated**

```
const obj = { name: "Nisharg" };
 obj.name = "Soni";  // allowed
```

▼ JavaScript Data Types

JavaScript has **two categories of data types**:

A. Primitive Types (7 types)

Primitives are:

- **stored by value**
- **immutable** (cannot be changed once created)
- **copied by value**

- live in **stack memory**

The seven primitive types are:

1. undefined

Meaning: a variable exists but has no value.

Automatically assigned by JS.

```
let a;
console.log(a); // undefined
```

2. null

Meaning: intentional absence of value.

```
let b = null;
```

`null` is used when we want to say “this has no value”.

3. boolean

`true` or `false`.

```
let isLoggedIn = true;
```

4. number

- JS has **one** number type → 64-bit floating point.
- Includes:
  - `NaN` (Not a Number)
  - `Infinity`
  - `Infinity`

Example:

```
typeof 42;     // "number"
typeof NaN;    // "number"
```

5. bigint

For very large integers (beyond $2^{53}-1$).

```
const big = 1234567890123456789012345n;
typeof big; // "bigint"
```

6. string

Sequence of characters. Immutable.

Modifying a string produces a **new** string.

```
let name = "Nisharg";
name[0] = "X"; // does not work
```

7. symbol

Unique and immutable identifiers.

```
const id = Symbol("id");
```

B. Objects (non-primitive)

Objects are:

- **mutable**
- **stored by reference**
- **copied by reference**
- live in **heap memory**

Examples:

Objects include:

- Object
- Array
- Function

Example:

```
const obj = { a: 1 };
const arr = [1, 2, 3];
```

Even arrays and functions are technically objects.

Primitive vs Object

Primitive:

```
let a = 10;
let b = a;
b = 20;

console.log(a); // 10
```

`a` and `b` hold **separate values**.

Object:

```
let obj1 = { x: 10 };
let obj2 = obj1;
obj2.x = 99;

console.log(obj1.x); // 99
```

Both point to **same memory location** (reference copy).

Checking Types

`typeof` operator

Value	typeof result
10	`"number"`
"hi"	`"string"`
true	`"boolean"`

Value	typeof result
undefined	`"undefined"`
Symbol()	`"symbol"`
10n	`"bigint"`
function(){}	`"function"`
{}	`"object"`
[]	`"object"`

▼ Operators in JavaScript

Types of Operators in JavaScript

JavaScript provides these main groups:

1. **Arithmetic operators**
2. **Assignment operators**
3. **Comparison (relational) operators**
4. **Logical operators**
5. **Unary operators**
6. **Ternary operator**
7. **String operators**
8. **Bitwise operators**
9. **Type operators** (`typeof`, `instanceof`)
10. **Nullish Coalescing (`??`)**
11. **Optional Chaining (`?.`)**

Arithmetic Operators

Operator	Meaning	Example	Output
`+`	Addition	5 + 2	7
`-`	Subtraction	5 - 2	3
`*`	Multiplication	5 * 2	10
`/`	Division	5 / 2	2.5

Operator	Meaning	Example	Output
`%`	Modulus (remainder)	5 % 2	1
`**`	Exponentiation	5 ** 2	25
`++`	Increment	a++	a + 1
`--`	Decrement	a--	a - 1

Assignment Operators

Operator	Meaning	Example
`=`	Assign	a = 10
`+=`	Add & assign	a += 5 → a = a + 5
`-=`	Subtract & assign	
`*=`	Multiply & assign	
`/=`	Divide & assign	
`%=`	Modulus assign	
`**=`	Exponent assign	

Comparison Operators

Operator	Meaning
`==`	Equal (loose, type conversion)
`===`	Strict equal (no type conversion)
`!=`	Not equal (loose)
`!==`	Strict not equal
`>`	Greater than
`<`	Less than
`>=`	Greater or equal
`<=`	Less or equal

Logical Operators

1. Logical AND → `&&`

```
true && true  // true
false && true // false
```

Short-circuit behavior:

```
false && console.log("Hi"); // Does NOT run
```

2. Logical OR → `||`

```
true || false // true
```

Short-circuit:

```
true || console.log("Hi"); // Does NOT run
```

3. Logical NOT → `!`

```
!true  // false
!!"hi" // true (double NOT converts to boolean)
```

Unary Operators

Operator	Meaning
`typeof`	Returns type
`delete`	Removes property from object
`+` (unary plus)	Converts to number
`-` (unary minus)	Negates number

Ternary Operator (Conditional Operator)

Format:

```
condition ? valueIfTrue : valueIfFalse;
```

Example:

```
let age = 20;
let result = age >= 18 ? "Adult" : "Minor";
```

Type Operators

typeof

```
typeof 123;      // "number"
typeof []        // "object"
typeof null      // "object"
```

instanceof

Checks if object is instance of constructor.

```
[] instanceof Array; // true
{} instanceof Object; // true
```

Nullish Coalescing Operator: ??

Returns right side **only if left is null or undefined**.

```
let user = null;
let name = user ?? "Guest"; // "Guest"
```

Optional Chaining Operator: ?.

Safely access nested properties.

```
const user = {};
console.log(user.address?.city); // undefined (no error)
```

Without it:

```
user.address.city; // TypeError
```

▼ Control Flow

IF, ELSE IF, ELSE

```
if (condition) {
// code runs when condition is true
} else if (anotherCondition) {
// code runs when another condition is true
} else {
// runs when none above are true
}
```

SWITCH STATEMENT

```
switch(value) {
case option1:
// code
break;
case option2:
// code
break;
default:
// code
}
```

LOOPS

1. FOR LOOP

```
for (initialization; condition; increment) {
// code
}
```

2. WHILE LOOP

```
while (condition) {
// code
}
```

3. DO...WHILE LOOP

```
do {
// code
} while (condition);
```

4. **FOR...OF LOOP** (Used for Arrays, Strings, Maps, Sets)

```
for (let value of iterable) {
// code
}

let colors = ["red", "green", "blue"];

for (let c of colors) {
   console.log(c);
}


for (let ch of "JS") {
   console.log(ch);
}
```

5. **FOR...IN LOOP** (Used for Objects)

```
for (let key in object) {
// code
}

let person = {
   name: "Nisharg",
   age: 20,
```

```
    city: "Anand"
};

for (let key in person) {
    console.log(key, "=", person[key]);
}
```

▼ Functions

Function Declaration

1. Named Function

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

2. Function Expression

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

3. Arrow Functions (ES6)
    not contain their own `this`

```
const add = (a, b) => a + b;
```

Default Parameters

```
function greet(name = "Guest") {
console.log("Hello", name);
}
greet();        // Hello Guest
greet("Nisharg"); // Hello Nisharg
```

```
const power = (x, y = 2) => x ** y;
console.log(power(5)); // 25
```

Return Statement

Used to return values from a function.

```
function multiply(a, b) {
return a * b;
}

console.log(multiply(4, 5)); // 20

//Automatic semicolon insertion
// ERROR
return -> return;
{
   name: "JS"
};



//FIX
return {
   name: "JS"
};
```

Function Hoisting

Function declaration is hoisted

```
sayHi(); // Works

function sayHi() {
   console.log("Hi");
}
```

Function expression is NOT hoisted

```
sayHi(); // Error

const sayHi = function() {
   console.log("Hi");
}
```

Arrow function also NOT hoisted

```
hello(); // Error
const hello = () => console.log("Hello");
```

Callbacks

A **function passed as an argument** to another function.

```
function greet(name) {
console.log("Hello " + name);
}
function processUser(callback) {
callback("Nisharg");
}
processUser(greet);
```

Higher-Order Functions (HOF)

A function that:

takes another function as argument
or returns a function
or both

```
// 1
function calculate(a, b, operation) {
return operation(a, b);
}
const add = (x, y) => x + y;
```

```
console.log(calculate(5, 3, add));


// 2
function multiplier(x) {
   return function(y) {
      return x * y;
   };
}

const double = multiplier(2);

console.log(double(10)); // 20
```

▼ Array

An **array** is an ordered collection of values.

```
let arr = [10, 20, 30];
```

BASIC ARRAY METHODS

push() → Add at end

```
let arr = [1, 2];
arr.push(3);
console.log(arr); // [1, 2, 3]
// Returns new length
// Modifies original array
```

pop() → Remove from end

```
let arr = [1, 2, 3];
arr.pop();
console.log(arr); // [1, 2]
```

```
// Returns removed item
// Modifies original array
```

shift() → Remove from start

```
let arr = [1, 2, 3];
arr.shift();
console.log(arr); // [2, 3]
// Returns removed item
```

unshift() → Add at start

```
let arr = [2, 3];
arr.unshift(1);
console.log(arr); // [1, 2, 3]
// Returns new length
```

slice() → Creates a Copy (Non-destructive)

```
let arr = [10, 20, 30, 40];
let part = arr.slice(1, 3); // arr.slice(start,end) end not included
console.log(part); // [20, 30]
console.log(arr);  // [10, 20, 30, 40]
// Does NOT modify original array

arr.slice() // returns full copy of array
```

splice() → Add/Remove Items (Destructive)

```
// array.splice(start, deleteCount, item1, item2, ...);

// Remove elements
let arr = [10, 20, 30];
arr.splice(1, 1);   // remove 1 item at index 1
console.log(arr);   // [10, 30]

// Add elements
let arr = [10, 30];
```

```
arr.splice(1, 0, 20); // at index 1, remove 0, add 20
console.log(arr);     // [10, 20, 30]

// Replace elements
let arr = [10, 40];
arr.splice(1, 1, 20, 30);
console.log(arr); // [10, 20, 30]
```

ARRAY ITERATION METHODS

`map()` → Transform each item & return NEW array

```
let nums = [1, 2, 3];
let doubled = nums.map(n => n * 2);
console.log(doubled); // [2, 4, 6]

//Does notr modify original
//Best for transforming data
```

`filter()` → Returns items matching condition

```
let nums = [1, 2, 3, 4];

let even = nums.filter(n => n % 2 === 0);
console.log(even); // [2, 4]

// Returns smaller or equal-sized array
// Does NOT modify original
```

`reduce()` → Convert array → single value

```
let nums = [1, 2, 3, 4];
let sum = nums.reduce((acc, curr) => acc + curr, 0);
console.log(sum); // 10
```

`find()` → Returns FIRST matching item

```
let users = [
{ id: 1, name: "A" },
{ id: 2, name: "B" }
];
let user = users.find(u => u.id === 2);
console.log(user); // { id: 2, name: "B" }
```

`some()` → Returns TRUE if at least one match

```
let nums = [1, 3, 5];
console.log(nums.some(n => n % 2 === 0)); // false
```

`every()` → TRUE only if ALL match

```
let nums = [2, 4, 6];
console.log(nums.every(n => n % 2 === 0)); // true
```

`sort()` → Sorts array (modifies original)
Default sort treats values as STRINGS (special case)

```
let nums = [10, 1, 5];
nums.sort();
console.log(nums); // [1, 10, 5]  wrong numerically


//Fix using comparator
nums.sort((a, b) => a - b);
console.log(nums); // [1, 5, 10]
```

`reverse()` → Reverses array

```
let arr = [1, 2, 3];
arr.reverse();
console.log(arr); // [3, 2, 1]
```

ARRAY DESTRUCTURING

```
const arr = [10, 20, 30];
const [a, b, c] = arr;
console.log(a); // 10
console.log(b); // 20
console.log(c); // 30
```

SPREAD OPERATOR ...

```
// Copy array
let arr1 = [1, 2];
let arr2 = [...arr1];

//Merg0e arrays
let a = [1, 2];
let b = [3, 4];
let c = [...a, ...b];
console.log(c); // [1, 2, 3, 4]


let arr = [2, 3];
let newArr = [1, ...arr, 4];
console.log(newArr); // [1, 2, 3, 4]


console.log([..."NISHARG"]);
// ["N", "I", "S", "H", "A", "R", "G"]
```

▼ Object

```
const person = {
name: "Nisharg",
age: 21,
city: "Anand"
};

const user = {};
```

```
user.name = "John";
user.age = 30;

const obj = new Object();
obj.a = 10;

function createUser(name, age) {
  return { name, age };
}
const u = createUser("Jay", 20);
```

Dot Notation vs Bracket Notation

```
console.log(person.name);
console.log(person.age);


const car = {
  "model name": "Swift",
  2025: "Future Model"
};

console.log(car["model name"]);
console.log(car[2025]);
```

The `this` keyword

`this` refers to **the object that is calling the method.**

Example:

```
const student = {
  name: "Raj",
  showName() {
    console.log(this.name);
  }
};
```

```
student.showName(); // "Raj"
```

Object Destructuring

```
const person = {
name: "Nisharg",
age: 21,
city: "Anand"
};
const { name, age, city } = person;

console.log(name); // Nisharg
console.log(age);  // 21
```

Optional Chaining `?.`

```
const user = {};
console.log(user.address.city); // Error

console.log(user.address?.city); // undefined
```

Rest Operator `...` in Objects

Used to collect **remaining properties** of an object.

```
const personX = {
  name: "Nisharg",
  age: 21,
  city: "Anand",
  country: "India"
};

const { name: n, ...rest } = personX;
```

```
console.log(n);    // "Nisharg"
console.log(rest);  // { age: 21, city: "Anand", country: "India" }
```

▼ DOM

The DOM is a tree-like representation of an HTML document. The browser converts HTML into nodes (elements, text nodes, comments). JavaScript can read and change that tree — add/remove elements, change content, change styles, attach events, etc.

Important nodes:

- `document` — the root
- `document.documentElement` — the `<html>` element
- `document.head`, `document.body`
- Element nodes (tags), Text nodes (strings inside elements)

1. Selecting elements

getElementById(id)

- Returns a single element with the matching `id` (or `null` if not found).

```
<div id="title">Hello</div>
```

```
const title = document.getElementById('title');
console.log(title.textContent);
```

Special case: `id` must be unique in the page. If not unique, only the first will be returned.

querySelector(selector)

- Very powerful: accepts any CSS selector and returns the **first** matching element (or `null`).
- Examples: `'#id'`, `'.class'`, `'div > p'`, `'input[name=email]'`.

```
const firstBtn = document.querySelector('.btn');
const emailInput = document.querySelector('input[name="email"]');
```

When to use: use `getElementById` when you have an id (micro-optimisation). Use `querySelector` for CSS-like flexibility.

querySelectorAll(selector)

- Returns a **static NodeList** (array-like) of **all** matches. You can iterate it with `for...of`, `forEach`, or convert to array with `[...nodelist]`.

```
const items = document.querySelectorAll('.item');
items.forEach(el => console.log(el.textContent));
```

Pitfall: `getElementsByClassName` and `getElementsByTagName` return *live HTMLCollections* (they update automatically if DOM changes), while `querySelectorAll` returns a *static* NodeList.

2. Editing content

innerText vs textContent vs innerHTML

- `textContent` — gets/sets raw text of an element (fast). Preserves spacing as text nodes. Does **not** parse HTML.
- `innerText` — returns what would be visible to the user (considers CSS like `display:none`) and is slower because it triggers a layout calculation. Setting `innerText` escapes HTML.
- `innerHTML` — gets/sets the HTML markup inside the element.

Examples

```
<div id="demo">
    <b>Hello</b>
    <span style="display:none">hidden</span>
</div>
```

```
const el = document.getElementById('demo');
console.log(el.textContent); // "Hello hidden"
console.log(el.innerText);   // "Hello" (hidden is not visible)
console.log(el.innerHTML);   // "<b>Hello</b> <span style="display:none">hidden</span>"
```

Best practice: use `textContent` to set plain text. Use `innerHTML` only when you need to inject markup and you control the source (or sanitize input).

3. Editing styles

`.style` (inline styles)

- `element.style.property = value` sets an inline style.
  CSS property names use camelCase in JS (`background-color` → `backgroundColor`).

```
const box = document.querySelector('.box');
box.style.backgroundColor = 'yellow';
box.style.width = '200px';
```

CSS classes

Use the class system to toggle or set groups of styles — keeps JS and CSS concerns separated.

`element.classList` methods

- `.add(name)`, `.remove(name)`, `.toggle(name[, force])`, `.contains(name)`, `.replace(old, new)`

```
el.classList.add('active');
el.classList.toggle('hidden'); // adds if missing; removes if present
if (el.classList.contains('error')) { /* ... */ }
el.classList.replace('old', 'new');
```

4. Creating & removing elements

document.createElement(tagName)

Creates an element in memory (not yet attached).

```
const li = document.createElement('li');
li.textContent = 'New item';
li.classList.add('todo-item');
```

Inserting elements

- `.append(childOrNode)` — appends to the end (accepts nodes and strings).
- `.prepend(childOrNode)` — inserts at the start.
- `.appendChild(node)` — older API, only nodes allowed.
- `.insertBefore(newNode, referenceNode)` — insert before reference.

```
const ul = document.querySelector('ul');
ul.append(li);        // add to end
ul.prepend(document.createElement('li')); // add to start
```

Note: `.append` accepts strings; `.appendChild` does not.

Removing elements

- `.remove()` — remove the element itself.
- `.removeChild(childNode)` — remove a child from a parent (older API).

```
li.remove(); // removes itself from DOM
```

5. DOM tree understanding & traversal

Nodes vs Elements

- Every element is a node, but there are other node types (text nodes, comment nodes).

- `node.nodeType` helps you check type (`1` = Element, `3` = Text).

Traversal properties & methods

- `parentNode` / `parentElement`
- `children` — element children (HTMLCollection)
- `childNodes` — all child nodes including text nodes (NodeList)
- `firstElementChild`, `lastElementChild`
- `firstChild`, `lastChild` (may be text nodes)
- `nextElementSibling`, `previousElementSibling`
- `nextSibling`, `previousSibling` (may be text nodes)

Example

```
const item = document.querySelector('.item');
const parent = item.parentElement;
const next = item.nextElementSibling;
const children = parent.children; // collection of element children
```

▼ Events

addEventListener

Syntax

```
element.addEventListener(type, listener, options);
```

- `type`: string like `"click"`, `"keydown"`.
- `listener`: function `(event) => { ... }`.
- `options`: boolean `useCapture` or an object `{ capture, once, passive }`.

Mouse Events

Event	Description
`click`	Fired when mouse button is clicked (press + release)
`dblclick`	Double-click

Event	Description
`mousedown`	Mouse button pressed
`mouseup`	Mouse button released
`mousemove`	Mouse moves
`mouseenter`	Mouse enters element (does NOT bubble)
`mouseleave`	Mouse leaves element (does NOT bubble)
`mouseover`	Mouse enters element (bubbles)
`mouseout`	Mouse leaves element (bubbles)
`contextmenu`	Right-click event
`wheel`	Mouse wheel scroll
`mouseover`	Pointer enters (can hover child elements)
`mouseout`	Pointer leaves

Keyboard Events

Event	Description
`keydown`	Key pressed down (fires repeatedly while holding)
`keyup`	Key released
`keypress` (deprecated)	Key pressed — old event, avoid using

Clipboard Events

Event	Description
`copy`	User copies content
`cut`	User cuts content
`paste`	User pastes content
`beforecopy`	Before copying
`beforecut`	Before cutting
`beforepaste`	Before pasting

Input & Form Events

Event	Description
input	Fires on every change of input value
change	Fires when input loses focus after value change
submit	Form submitted
reset	Form reset
invalid	Validation failed
select	Text selection inside input
focus	Element receives focus
blur	Element loses focus

Drag & Drop Events

Event	Description
dragstart	Start dragging
drag	During drag
dragend	Dragging ends
dragenter	Draggable enters drop zone
dragleave	Leaves drop zone
dragover	Element is dragged over drop zone
drop	Dropped

Window Events

Event	Description
load	Page fully loaded
beforeunload	Before page refresh/exit
unload	When page is closed (obsolete)
resize	Window size changed
scroll	Scroll happens
error	JavaScript errors

Event	Description
`hashchange`	URL hash changed (`#section`)
`popstate`	Browser back/forward navigation
`online`	Internet connected
`offline`	Internet disconnected

Media Events (Audio/Video)

Event	Description
`play`	Media starts playing
`pause`	Paused
`ended`	Finished playing
`timeupdate`	Playback time changed
`volumechange`	Volume changed
`seeking`	User is seeking
`seeked`	Seek completed
`loadeddata`	Media data loaded
`error`	Video/audio error

Event object

The handler receives an `Event`.

Useful properties:

- `event.type` — string event type, e.g. `"click"`.
- `event.target` — the deepest element that triggered the event (where the event originated).
- `event.currentTarget` — the element the listener is attached to.
- `event.preventDefault()` — stop default browser action (e.g., link navigation, form submit).
- `event.stopPropagation()` — stop the event from bubbling further.
- `event.stopImmediatePropagation()` — also prevents other listeners of same type on the same element from running.

- `event.clientX`, `event.clientY` — mouse coordinates relative to viewport.
- `event.detail` — number of clicks for `click`/`dblclick` (1, 2).

Event bubbling & capturing

Events travel in three phases:

1. **Capture phase**: Document → down to target (top→down)
2. **Target phase**: Event arrives at the target element (listeners there run).
3. **Bubble phase**: Event bubbles up from target → document (bottom→top).

Default: most listeners run in the bubble phase.

▼ Web Storage

Web Storage API allows websites to store data inside the browser in the form of **key–value pairs**.

It provides two types of storage:

1. **localStorage** → persistent storage
2. **sessionStorage** → tab-specific temporary storage

Both:

- store data as **strings**
- have simple **set / get / remove / clear** APIs
- have much **larger storage** space than cookies (~5 MB)
- do **not** send data to the server (unlike cookies)

localStorage

Characteristics

- Data **remains permanently** (unless manually removed).
- Survives:

- Page reload
- Browser restart
- System restart

- Shared across all **tabs/windows** of the same site.
- Capacity: ~5–10MB (varies by browser).

sessionStorage

Characteristics

- Data survives:
  - Page reload
- But cleared when:
  - Tab is closed
  - Browser window is closed
- Not shared across tabs (new tab = empty session).

Store data

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

Retrieve data

```
let value = localStorage.getItem("key");
```

If key does not exist → returns **null**.

Remove a key

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

Clear everything

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

Storing & Retrieving Objects

Store object

```
const user = { name: "Nisharg", age: 21 };
localStorage.setItem("user", JSON.stringify(user));
```

Retrieve object

```
const obj = JSON.parse(localStorage.getItem("user"));
console.log(obj.name);
```

Common pattern

```
let data = JSON.parse(localStorage.getItem("cart")) || [];
```

Iterating over storage keys

```
for (let i = 0; i < localStorage.length; i++) {
  const key = localStorage.key(i);
  console.log(key, localStorage.getItem(key));
}
```

Storage Event

When **localStorage** changes in *one tab*,

other tabs on the **same domain** receive a `storage` event.

```
window.addEventListener("storage", (e) => {
  console.log("Key changed:", e.key);
  console.log("Old:", e.oldValue);
```

```
  console.log("New:", e.newValue);
});
```

▼ Cookies

What Are Cookies?

Cookies are **small pieces of data** stored in the browser.

They are mainly used for:

- Authentication
- Session management
- Tracking user behavior
- Remembering preferences

Cookies are sent **automatically** to the server with every HTTP request.

Creating / Setting Cookies in JavaScript

Cookies are created using:

```
document.cookie = "key=value";
```

Example:

```
document.cookie = "name=Nisharg";
```

Adding Expiry to a Cookie

Expires on specific date:

```
document.cookie = "user=Nisharg; expires=Tue, 20 Jan 2026 12:00:00 UTC";
```

Expires after X seconds:

```
document.cookie = "token=12345; max-age=3600"; // 1 hour
```

Cookies options

expires=DATE

Defines the **exact date and time** when the cookie should be deleted.

Example:

```
expires=Wed, 20 Jan 2026 12:00:00 UTC
```

max-age=SECONDS

Defines how long (in seconds) the cookie should live.

Example:

```
max-age=3600   // 1 hour
```

path=PATH

Defines **which pages** can access the cookie.

Example:

```
path=/        // accessible everywhere on site
path=/admin    // only admin section
```

secure

Cookie is **sent only over HTTPS**.

Example:

```
secure
```

HttpOnly

Prevents JavaScript from reading the cookie.

Example:

```
HttpOnly
```

SameSite=Lax | Strict | None

Controls whether cookie can be sent in cross-site requests.

SameSite=Lax (default)

```
SameSite=Lax
```

Sent on normal navigation (click links)

Not sent in POST requests, iframes, AJAX

SameSite=Strict (most secure)

```
SameSite=Strict
```

Only sent from same site

Not sent even if user clicks link from Google

SameSite=None (allows cross-site)

```
SameSite=None;
```