Events in JavaScript -

1. Definition

- An **event** is an action or occurrence that happens in the browser.
- Examples:
  - User actions → click, key press, scroll, mouse hover
  - Browser actions → page load, resize, errors
- Events allow JavaScript to make web pages **interactive and dynamic**.

2. Types of Events

🔹 Mouse Events

- click → when element is clicked
- dblclick → double click
- mouseover → mouse moves over element
- mouseout → mouse leaves element
- mousemove → mouse moves on element
- mousedown / mouseup → press/release mouse button

```
btn.addEventListener("click", () => console.log("Clicked!"));
```

🔹 Keyboard Events

- keydown → key is pressed
- keyup → key is released
- keypress → key is pressed (deprecated, avoid)

```
document.addEventListener("keydown", (e) => console.log(e.key));
```

🔹 Form Events

- submit → form submitted
- change → input value changed
- focus → element gets focus
- blur → element loses focus
- input → whenever user types inside input

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

  e.preventDefault(); // stop page reload

  console.log("Form submitted!");

});
```

🔹 Window Events

- load → page fully loaded
- resize → browser window resized
- scroll → page is scrolled
- unload / beforeunload → page is leaving/closing

```
window.addEventListener("resize", () => console.log("Window resized!"));
```

🔹 Clipboard Events

- copy → when text copied
- cut → when text cut
- paste → when text pasted

🔹 Drag & Drop Events

- dragstart, drag, dragend
- dragenter, dragover, dragleave, drop

3. **Event Handling Methods**

Inline HTML attribute

```
<button onclick="alert('Clicked!')">Click Me</button>
```

DOM property

```
btn.onclick = () => alert("Clicked!");
```

addEventListener() (✅ Recommended)

```
btn.addEventListener("click", () => alert("Clicked!"));
```

4. **Event Object**

- Provides details about event (target element, type, key pressed, etc.).

```
document.addEventListener("click", (event) => {
  console.log(event.type);   // "click"
  console.log(event.target); // element clicked
});
```

5. **Event Propagation**

- **Bubbling (default)** → child → parent → root

- **Capturing** → root → parent → child

element.addEventListener("click", handler, true); // capturing

✍️ Short Notes (for Revision)

- **Event = action in browser** (user/browser triggered).
- **Event types**:
  - Mouse → click, dblclick, mouseover, mouseout, mousemove
  - Keyboard → keydown, keyup
  - Form → submit, change, focus, blur, input
  - Window → load, resize, scroll, unload
  - Clipboard → copy, cut, paste
  - Drag & Drop → dragstart, dragover, drop
- **Event handling methods**:
  1. Inline (onclick="...")
  2. DOM property (element.onclick = ...)
  3. addEventListener() (best)
- **Event object** → details (event.type, event.target, event.key)
- **Propagation** → Bubbling (default) & Capturing (optional).

localStorage in JavaScript -

1. **Definition**

- localStorage is part of the **Web Storage API** (BOM).
- It allows you to **store key–value pairs in the browser**.
- **Persistent storage** → data stays even after page refresh or browser restart (until manually cleared).
- Storage is **synchronous** (blocking).

2. **Properties**

- Belongs to the **window** object → window.localStorage (can be used directly as localStorage).
- Stores only **string data** (other types are converted to string).
- Maximum size → usually **5–10 MB** depending on browser.

3. **Common Methods**

Method	Description	Example
setItem(key, value)	Store data	localStorage.setItem("name", "John")
getItem(key)	Get data	localStorage.getItem("name") → "John"
removeItem(key)	Delete one item	localStorage.removeItem("name")
clear()	Delete all items	localStorage.clear()
key(index)	Get key name by index	localStorage.key(0)
length	Number of stored items	localStorage.length

4. **Examples**

Store and Retrieve Data

```
localStorage.setItem("username", "Alex");   // store
console.log(localStorage.getItem("username")); // "Alex"
```

Remove Data

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

Clear All Data

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

Loop through localStorage

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

5. **Handling Non-String Data**

Since localStorage stores only **strings**, you must use JSON.stringify() and JSON.parse() for objects/arrays.

```
let user = { name: "John", age: 25 };


// Store object

localStorage.setItem("user", JSON.stringify(user));


// Get object

let data = JSON.parse(localStorage.getItem("user"));

console.log(data.name); // John
```

6. **Difference Between localStorage and sessionStorage**

Feature	localStorage	sessionStorage
Lifetime	Permanent (until cleared)	Ends when tab/browser closed
Size	~5–10 MB	~5 MB
Shared	Across all tabs of same domain	Only current tab

✍️ Short Notes (for revision)

- **localStorage** → web storage for key–value pairs.
- Data is **permanent** (till manually cleared).
- Stores only **strings** (use JSON.stringify & JSON.parse for objects/arrays).
- **Methods**:
  - setItem(key, value) → save
  - getItem(key) → retrieve
  - removeItem(key) → delete
  - clear() → delete all

    - key(index) → get key name
    - length → count of items
- Difference from sessionStorage → localStorage persists, sessionStorage clears on tab close.

Set & set method in JS -

1. **Set (Object)**

- Introduced in **ES6**.
- A **Set** is a collection of **unique values** (no duplicates allowed).
- Values can be of any type (primitive or objects).

Example:

```
let mySet = new Set([1, 2, 3, 3, 4]);

console.log(mySet); // {1, 2, 3, 4}
```

2. **Common Set Methods**

Method	Description	Example
add(value)	Adds a new value	mySet.add(5)
delete(value)	Removes a value	mySet.delete(2)
has(value)	Checks if value exists	mySet.has(3) → true
clear()	Removes all values	mySet.clear()
size	Returns number of values	mySet.size → 3

⚡ **Note**: In Set, we use .add(), not .set().

3. **Map (Object) – where set() is used**

- A **Map** is a collection of **key–value pairs** (like objects, but keys can be of any type).

- In Map, we use .set() to insert values.

Example:

```
let myMap = new Map();


// set() → add key-value pair
myMap.set("name", "Alice");
myMap.set("age", 25);
myMap.set(true, "boolean key");


console.log(myMap.get("name")); // Alice
console.log(myMap.size);       // 3
```

4. **Map Methods (important with set)**

Method	Description	Example
set(key, value)	Adds/updates value	myMap.set("city", "Delhi")
get(key)	Returns value	myMap.get("city") → Delhi
delete(key)	Removes key-value pair	myMap.delete("city")
has(key)	Checks if key exists	myMap.has("city")
clear()	Removes all pairs	myMap.clear()
size	Number of entries	myMap.size

✍️ Short Notes

- **Set** → unique values → use .add(), not .set().
- **Map** → key-value pairs → use .set(key, value) to add/update data.
- .set() in Map can accept any type of key (string, number, object, boolean, etc.).

Date and Time in JavaScript -

1. **Date Object**

- In JavaScript, **Date** is a built-in object used to **work with dates and times**.
- It provides methods to **create, get, set, and format** dates & times.

let now = new Date();

console.log(now); // Current date & time

2. **Ways to Create Date Objects**

1. **Current Date & Time**

let now = new Date();

1. **Specific Date**

let d1 = new Date("2025-08-16"); // yyyy-mm-dd

1. **Year, Month, Day, Hour, Minute, Second, Millisecond**

let d2 = new Date(2025, 7, 16, 10, 30, 0); // month starts from 0 (0=Jan, 7=Aug)

1. **Milliseconds since Jan 1, 1970 (Unix Epoch)**

let d3 = new Date(0); // Thu Jan 01 1970

3. **Date Methods (Getters)**

Method	Description	Example
getFullYear()	Year (4 digits)	date.getFullYear() → 2025
getMonth()	Month (0–11)	date.getMonth() → 7 (August)
getDate()	Day of month (1–31)	date.getDate() → 16
getDay()	Day of week (0=Sun, 6=Sat)	date.getDay() → 6 (Saturday)
getHours()	Hours (0–23)	date.getHours()
getMinutes()	Minutes (0–59)	date.getMinutes()
getSeconds()	Seconds (0–59)	date.getSeconds()
getMilliseconds()	Milliseconds (0–999)	date.getMilliseconds()
getTime()	Milliseconds since 1970	date.getTime()

4. **Date Methods (Setters)**

Method	Description	Example
setFullYear(year)	Set year	date.setFullYear(2030)
setMonth(month)	Set month (0–11)	date.setMonth(11)
setDate(day)	Set day (1–31)	date.setDate(25)
setHours(h)	Set hours	date.setHours(22)
setMinutes(m)	Set minutes	date.setMinutes(45)
setSeconds(s)	Set seconds	date.setSeconds(30)
setMilliseconds(ms)	Set ms	date.setMilliseconds(500)

5. **Formatting Dates & Times**

- toDateString() → Human-readable date

```
console.log(new Date().toDateString()); // "Sat Aug 16 2025"
```

- toTimeString() → Human-readable time

```
console.log(new Date().toTimeString()); // "10:45:12 GMT+0530"
```

- toLocaleDateString() → Localized date

```
console.log(new Date().toLocaleDateString()); // "16/8/2025"
```

- toLocaleTimeString() → Localized time

```
console.log(new Date().toLocaleTimeString()); // "10:45:12 am"
```

- toISOString() → Standard ISO format

```
console.log(new Date().toISOString()); // "2025-08-16T05:15:12.000Z"
```

6. **Real Example: Digital Clock**

```
setInterval(() => {
  let now = new Date();
```

```
  console.log(now.toLocaleTimeString());
}, 1000);
```

✍️ Short Notes (for revision)

- Date() → built-in object to handle date & time.
- Create date → new Date(), new Date("YYYY-MM-DD"), new Date(y,m,d,h,m,s,ms).
- **Getters** → getFullYear(), getMonth(), getDate(), getDay(), getHours(), getMinutes(), getSeconds(), getTime().
- **Setters** → setFullYear(), setMonth(), setDate(), setHours(), setMinutes(), setSeconds().
- Formatting → toDateString(), toTimeString(), toLocaleDateString(), toLocaleTimeString(), toISOString().
- Month starts from **0** (0=January, 11=December).

Timing-Based Events in JavaScript -

JavaScript allows executing code **after a delay** or **repeatedly at intervals** using **timing functions**. These are part of the **Browser's Window (BOM) object**.

1. **setTimeout()**

- Executes a function **once** after a given time (in ms).

```
setTimeout(() => {
  console.log("Hello after 2 seconds");
}, 2000); // 2000ms = 2 sec
```

- **Clear Timeout** → Stop execution before it happens.

```
let timer = setTimeout(() => console.log("Won't run"), 3000);

clearTimeout(timer);
```

2. **setInterval()**

- Executes a function **repeatedly** after every given interval.

```
setInterval(() => {
  console.log("Repeating every 1 second");
}, 1000);
```

- **Clear Interval** → Stop the repeated execution.

```
let interval = setInterval(() => console.log("Tick"), 1000);
clearInterval(interval);
```

3. **requestAnimationFrame()**

- Optimized way to create animations in sync with the browser's refresh rate.

```
function animate() {
  console.log("Animating...");
  requestAnimationFrame(animate);
}
requestAnimationFrame(animate);
```

4. **Difference Between setTimeout and setInterval**

Feature	setTimeout()	setInterval()
Execution	Once after delay	Repeats at interval
Cancel Method	clearTimeout()	clearInterval()
Use Case	Delayed actions	Repeated tasks (clocks, polls)

5. **Example: Digital Clock with setInterval()**

```
setInterval(() => {
  let now = new Date();
  console.log(now.toLocaleTimeString());
```

}, 1000);

✍️ Short Notes (for revision)

- **Timing-based events** let JS run code later or repeatedly.
- setTimeout(fn, ms) → run function once after ms.
- clearTimeout(id) → stop setTimeout.
- setInterval(fn, ms) → run function repeatedly every ms.
- clearInterval(id) → stop setInterval.
- requestAnimationFrame(fn) → best for smooth animations.

Pass by Value vs Pass by Reference -

1. **Pass by Value**

- When a variable is **primitive type** (number, string, boolean, null, undefined, symbol, bigint),
- JavaScript **copies the value** and assigns it to another variable.
- Changing the new variable **does not affect** the original.

✅ Example:

```
let a = 10;

let b = a;   // copy value

b = 20;



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

console.log(b); // 20
```

2. **Pass by Reference**

- When a variable is a **non-primitive type** (object, array, function),
- JavaScript **stores a reference (memory address)**, not the actual value.
- Assigning it to another variable **copies the reference**, not the object.
- Changing one will also affect the other (because both point to same memory).

✅ Example:

```
let obj1 = { name: "Alice" };

let obj2 = obj1;   // copy reference


obj2.name = "Bob";


console.log(obj1.name); // "Bob" (changed)

console.log(obj2.name); // "Bob"
```

3. **Key Difference**

Feature	Pass by Value (Primitive)	Pass by Reference (Non-Primitive)
Type of data	Number, String, Boolean, etc.	Object, Array, Function
What is copied	Actual value	Memory reference
Change in new variable	Does NOT affect original	Affects original

4. **Trick to avoid reference issue**

If you want to **copy objects/arrays without linking**, use **spread operator** or Object.assign.

```
let arr1 = [1, 2, 3];

let arr2 = [...arr1];  // separate copy


arr2[0] = 100;


console.log(arr1); // [1,2,3] (safe)

console.log(arr2); // [100,2,3]
```

✍️ Short Notes

- **Primitive types** → Pass by Value (copy actual value).
- **Non-primitive types** → Pass by Reference (copy memory address).
- Changing reference variable changes original too.
- To avoid, use spread (...) or Object.assign.

Shallow Copy vs Deep Copy of Objects in JavaScript -

1. **Shallow Copy**

- Copies **only the first level** of properties.
- If the object has **nested objects/arrays**, they are still copied **by reference**.
- So changes in nested objects affect the original.

✅ Example:

let obj1 = { name: "Alice", address: { city: "Delhi" } };

let obj2 = { ...obj1 }; // Shallow copy

obj2.name = "Bob"; // top-level change → doesn't affect original

obj2.address.city = "Pune"; // nested change → affects original

console.log(obj1.name); // Alice (safe)

console.log(obj1.address.city); // Pune (changed 😢)

2. **Deep Copy**

- Copies the object **completely**, including nested objects/arrays.
- Original and copy are **totally independent**.

✅ Example using structuredClone:

let obj1 = { name: "Alice", address: { city: "Delhi" } };

let obj2 = structuredClone(obj1); // Deep copy

obj2.address.city = "Pune";

console.log(obj1.address.city); // Delhi (safe ✅)

console.log(obj2.address.city); // Pune

Other Ways for Deep Copy

1. JSON.parse(JSON.stringify(obj)) (works but loses methods & special values).
2. Libraries like Lodash (_.cloneDeep).

3. **Comparison Table**

Feature	Shallow Copy	Deep Copy
Nested objects	Copied **by reference**	Copied as **new object**
Independence	Not fully independent	Fully independent
Methods	Preserved	May lose (JSON method issue)
Performance	Faster (simple copy)	Slower (complete cloning)

✍️ Short Notes

- **Shallow Copy** → Copies top-level, nested still linked.
- **Deep Copy** → Fully independent copy (all levels).
- Methods: Spread (...), Object.assign = Shallow.
- Methods: structuredClone, JSON.parse(JSON.stringify()) = Deep.

this in JavaScript -

1. **What is this?**

- this is a **special keyword** that refers to the **object that is currently executing the code**.
- Its value depends on **how and where** a function is called, not where it is defined.

2. **this in Different Contexts**

(a) **Global Context**

- In browsers, this refers to the **window object** (global object).

```
console.log(this); // window
```

(b) **Inside a Function**

- In **non-strict mode**: this = window.
- In **strict mode**: this = undefined.

```
function test() {

  console.log(this);

}

test(); // window (non-strict) / undefined (strict)
```

(c) **Inside a Method (Object Function)**

- this refers to the **object that owns the method**.

```
let user = {

  name: "Alice",

  greet: function() {

    console.log(this.name);

  }

};

user.greet(); // "Alice"
```

(d) **Arrow Functions**

- Arrow functions **do not have their own this**.
- They use this from their **surrounding scope (lexical scoping)**.

```
let user = {
```

```
  name: "Alice",
  greet: () => {
    console.log(this.name);
  }
};
user.greet(); // undefined (because `this` is from global scope)
```

(e) **Event Handlers**

- this refers to the **HTML element** that received the event.

```
document.querySelector("button").addEventListener("click", function() {
  console.log(this); // button element
});
```

- But with **arrow function**: this = outer scope (not the element).

(f) **With call, apply, bind**

- We can **manually set this**.

```
function greet() {
  console.log(this.name);
}
let user = { name: "Alice" };

greet.call(user);  // "Alice"
greet.apply(user); // "Alice"

let bound = greet.bind(user);
bound(); // "Alice"
```

3. **Key Points to Remember**

- Global scope → this = window (non-strict).

- Function → this = window or undefined (strict).
- Object method → this = object.
- Arrow function → this = surrounding scope.
- Event handler → this = element.
- call, apply, bind → manually set this.

✍️ Short Notes

- this → refers to the **object that calls the function**.
- Global: this = window.
- Function: this = window (non-strict) / undefined (strict).
- Method: this = object.
- Arrow fn: inherits this from parent scope.
- Event: this = element.
- Use call, apply, bind to control this.

How JavaScript Works -

1. **JavaScript is Single-Threaded**

- JS can do **one task at a time** (synchronous).
- It runs inside the **JavaScript Engine** (like V8 in Chrome, SpiderMonkey in Firefox).
- Uses **Call Stack + Memory Heap** to execute code.

2. **Execution Context**

Whenever JS code runs, it creates an **Execution Context**.

- **Global Execution Context (GEC):** Created first, runs global code.
- **Function Execution Context (FEC):** Created when a function is called.
- Each context has:
  - **Memory (Variable Environment)** → stores variables, functions.
  - **Code (Thread of Execution)** → executes line by line.

3. **Call Stack**

- Stores execution contexts in a **stack (LIFO)**.
- When a function is called → pushed to stack.
- When function finishes → popped from stack.

✅ Example:

function a() { console.log("a"); }

function b() { a(); console.log("b"); }

b();

- Call Stack process:
  - GEC → b() → a() → print → pop a() → print → pop b().

4. **Memory Heap**

- Where objects and functions are stored in memory.
- Dynamic allocation happens here.

5. **Synchronous vs Asynchronous**

- By default, JS runs **synchronously** (line by line).
- But browsers provide APIs (setTimeout, DOM events, fetch, etc.) for **async operations**.

6. **Event Loop & Callback Queue**

- **Event Loop** makes JS non-blocking.
- Process:
  1. JS executes code in call stack.
  2. Async tasks (like setTimeout, API calls) go to **Web APIs**.
  3. When done, callbacks are pushed into **Callback Queue**.
  4. **Event Loop** checks if call stack is empty → then pushes queued callback into stack.

✅ Example:

console.log("Start");

setTimeout(() => console.log("Async Task"), 1000);

console.log("End");

Output:

Start

End

Async Task

7. **JS Engine (like V8)**

- **Parsing** → breaks code into tokens.
- **Compilation (JIT)** → converts JS into machine code quickly.
- **Execution** → runs optimized code.

✍️ Short Notes

- JS is **single-threaded** → runs one task at a time.
- Uses **Call Stack** (execution order) + **Memory Heap** (storage).
- Creates **Execution Contexts** (Global + Function).
- **Event Loop** + **Callback Queue** → handle async tasks.
- JS Engine (like V8) → parses, compiles, executes code.

JavaScript Event Loop -

1. **Why Event Loop Exists?**

- JavaScript is **single-threaded** (only one thing at a time).
- But still, JS can handle async tasks (like setTimeout, API calls, DOM events).
- Event Loop is the mechanism that **manages sync + async code together**.

2. **Main Components**

1. **Call Stack** – Executes code line by line (synchronous).
2. **Heap** – Memory storage for objects/functions.
3. **Web APIs (Browser APIs)** – Provided by browser (e.g., setTimeout, DOM events, fetch).
4. **Callback Queue (Task Queue / Message Queue)** – Holds callbacks waiting to run.
5. **Event Loop** – Continuously checks if **Call Stack is empty**, then moves tasks from Callback Queue → Call Stack.

3. **How It Works (Step by Step)**

Example:

```
console.log("1");


setTimeout(() => {
  console.log("2");
}, 1000);


console.log("3");
```

Execution Flow:

1. console.log("1") → runs immediately (stack).
2. setTimeout → sent to **Web API**, timer runs separately.
3. console.log("3") → runs immediately.
4. After 1 sec → callback (console.log("2")) moves to **Callback Queue**.
5. Event Loop checks → stack empty → pushes callback to stack.
6. Prints "2".

✅ Output:

1

3

2

4. **Microtask Queue vs Callback Queue**

- **Microtask Queue** (higher priority): Promises, MutationObserver.
- **Callback Queue (Macrotask Queue):** setTimeout, setInterval, events.
- Event Loop always clears **Microtasks first** before moving to Callback Queue.

Example:

```
console.log("A");


setTimeout(() => console.log("B"), 0);


Promise.resolve().then(() => console.log("C"));


console.log("D");
```

👉 Output:

A

D

C // microtask first

B // then macrotask

5. **Summary in Simple Words**

- Call Stack → runs code.
- Async tasks go to **Web APIs**.
- When done, callbacks → **Queues**.
- **Event Loop** moves tasks to Call Stack when it's free.
- **Microtask Queue** > **Callback Queue** (priority).

✍️ Short Notes

- JS is **single-threaded**, Event Loop helps with async tasks.
- Components: Call Stack, Heap, Web APIs, Callback Queue, Microtask Queue.
- **Execution order:** Call Stack → Microtask Queue → Callback Queue.
- Promise (microtask) runs before setTimeout (macrotask).

Hoisting in JavaScript -

1. What is Hoisting?

- **Hoisting** means: In JavaScript, **variable and function declarations are moved to the top of their scope (memory phase) before execution.**
- So, you can use variables/functions **before declaring them** (but with some differences).

2. How it works?

When JS runs:

- **Memory Creation Phase (Hoisting Phase):**
  - Variables declared with var → initialized with undefined.
  - Variables with let & const → exist in **Temporal Dead Zone (TDZ)** (not accessible until declared).
  - Function declarations → stored completely in memory.
- **Execution Phase:**
  - Code runs line by line using the values from memory.

3. Examples

(a) Hoisting with var

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

var a = 10;
```

✅ var is hoisted → declared at top, but initialized with undefined.

(b) Hoisting with let & const

```
console.log(b); // ReferenceError

let b = 20;
```

❌ let and const are hoisted but kept in **TDZ**, so they can't be used before declaration.

(c) Hoisting with Functions

```
greet(); // "Hello"


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

✅ Function declarations are fully hoisted → can be called before definition.

(d) Function Expressions & Arrow Functions

```
sayHi(); // TypeError

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

❌ Function expressions (using var, let, const) are treated like variables → not hoisted as full functions.

4. **Key Points**

- **Function Declarations** → fully hoisted.
- **Var Variables** → hoisted but set to undefined.
- **Let & Const** → hoisted but in **TDZ** (can't be accessed early).
- **Function Expressions & Arrow Functions** → behave like variables.

✍️ Short Notes

- Hoisting = JS moves **declarations** to top of scope during memory creation.
- var → hoisted with undefined.
- let & const → hoisted but in **TDZ**, not accessible before declaration.
- **Functions** → fully hoisted.
- **Function Expressions / Arrow Functions** → not hoisted (behave like variables).

Lexical Scope & Scope Chaining -

1. **What is Scope?**

- **Scope** = Region where a variable is accessible.
- Types of scope in JS:
  1. **Global Scope** → accessible everywhere.
  2. **Function Scope** → created with functions.
  3. **Block Scope** → created with {} (with let and const).

2. **What is Lexical Scope?**

- **Lexical** means "related to position in code".
- A function's scope is **decided by where it is written in code, not by where it is called**.
- Inner functions have access to variables of their outer functions.

Example:

```
function outer() {

 let a = 10;



 function inner() {

  console.log(a); // can access outer variable

 }



 inner();

}

outer(); // Output: 10
```

✅ inner has **lexical scope** to outer.

3. **What is Scope Chain?**

- When JS looks for a variable:
  1. First checks **local scope**.

2. If not found → checks parent's scope.
3. Keeps going up until **global scope**.
4. If still not found → ReferenceError.

Example:

```
let x = 1;


function a() {

 let y = 2;


 function b() {

  let z = 3;

  console.log(x, y, z);

 }


 b();

}

a();
```

✅ Output: 1 2 3

- b → finds z in local.
- If not found, goes to a's scope → finds y.
- If not found, goes to global → finds x.

4. **Key Differences**

- **Lexical Scope** → Defines where a function can access variables (depends on code position).
- **Scope Chain** → Process of searching variables step-by-step outward until global.

✍️ Short Notes

- **Scope** = area where variable is accessible.
- **Lexical Scope** = inner function can access outer function variables (based on code position).

- **Scope Chain** = variable lookup path (local → parent → global).
- If variable not found → ReferenceError.

Closures in JavaScript -

1. **What is a Closure?**

- A **closure** is created when a function **remembers and accesses variables** from its **lexical scope**, even after the outer function has finished executing.
- In simple words: **Functions + their outer scope references = Closure**.

2. **How Closures Work?**

- When an inner function is returned from an outer function:
  - The inner function keeps a **reference** to variables in the outer function's scope.
  - These variables are **not garbage collected**, because the inner function still needs them.

3. **Example**

(a) Basic Closure

```
function outer() {

 let count = 0;



 function inner() {

  count++;

  console.log(count);

 }



 return inner;

}
```

```
const counter = outer();

counter(); // 1

counter(); // 2

counter(); // 3
```

✅ Even though outer() finished execution, the inner function inner() still **remembers count**. That's a **closure**.

(b) Real-Life Example: Private Variables

```
function bankAccount() {
  let balance = 100;


  return {
    deposit: function(amount) {
      balance += amount;
      console.log(`Deposited: ${amount}, Balance: ${balance}`);
    },
    withdraw: function(amount) {
      if (amount <= balance) {
        balance -= amount;
        console.log(`Withdrew: ${amount}, Balance: ${balance}`);
      } else {
        console.log("Insufficient funds");
      }
    }
  };
}
```

```
const account = bankAccount();

account.deposit(50);   // Deposited: 50, Balance: 150

account.withdraw(30);  // Withdrew: 30, Balance: 120
```

✅ balance is private → can only be accessed via closure methods.

4. **Uses of Closures**

- **Data Privacy** (hiding variables).
- **Creating function factories** (functions that generate other functions).
- **Maintaining state** in async code.
- **Event handlers** and **callbacks**.

5. **Important Notes**

- Closures **remember reference**, not the actual value.
- Too many closures may cause **memory leaks** if not used carefully.

✍️ Short Notes

- **Closure** = Function + its lexical scope variables (even after outer function ends).
- Inner function "remembers" outer variables.
- Uses → Data privacy, function factories, state management, event handlers.
- Example: Counter function keeps count even after outer ends.
- Caution → May cause memory issues if misused.

ECMAScript 2015 – 2025 -

🌍 What is ECMAScript?

- **ECMAScript (ES)** is the standard specification of JavaScript maintained by **ECMA International**.
- Every year, a new version is released (ES2015 → ES2025).
- ES2015 (ES6) was a **big milestone**, adding modern features.

📅 ECMAScript Versions (2015 – 2025)

✅ **ES2015 (ES6)** – The Big Update 🚀

- let and const

- Arrow functions () => {}
- Template strings (`Hello ${name}`)
- Default parameters
- Destructuring assignment
- Rest & spread operators (...)
- Classes (class Person {})
- Modules (import / export)
- Promises
- for...of loop
- Symbol type
- Map & Set

✅ ES2016 (ES7)

- **Exponentiation operator** (**) → 2 ** 3 = 8
- **Array.prototype.includes()** → [1,2,3].includes(2)

✅ ES2017 (ES8)

- async / await
- Object methods: Object.values(), Object.entries(), Object.getOwnPropertyDescriptors()
- String padding: padStart, padEnd
- Shared memory & Atomics

✅ ES2018 (ES9)

- Rest/Spread properties for objects
- Asynchronous iteration (for await...of)
- Promise.finally()
- RegExp improvements

✅ ES2019 (ES10)

- Array.flat() and Array.flatMap()
- Object.fromEntries()
- String.trimStart() / trimEnd()
- Optional catch binding (catch {} without error param)

✅ ES2020 (ES11)

- BigInt (123n)
- Nullish coalescing operator (??)
- Optional chaining (obj?.prop?.value)
- Promise.allSettled()
- globalThis
- String.matchAll()
- Dynamic import()

✅ ES2021 (ES12)

- Logical assignment operators (||=, &&=, ??=)
- Numeric separators (1_000_000)
- Promise.any()
- WeakRefs & FinalizationRegistry
- String.replaceAll()

✅ ES2022 (ES13)

- Top-level await in modules
- Class fields (class Person { name = "John" })
- Private class fields (#age = 25)
- Object.hasOwn()
- Error cause (new Error("msg", { cause: err }))

✅ ES2023 (ES14)

- Array findLast() and findLastIndex()
- Symbol as WeakMap keys
- Hashbang grammar (#!/usr/bin/env node)
- ArrayBuffer improvements

✅ ES2024 (ES15)

- Promise.withResolvers() (officialized)
- Object.groupBy() & Map.groupBy()
- Set methods: union(), intersection(), difference(), isDisjointFrom()

✅ ES2025 (Upcoming)

(still proposals, subject to change – based on TC39 work-in-progress)

- Pipeline operator (|>) → experimental
- Pattern matching (like switch but more powerful)
- Records & Tuples (immutable data structures)
- More Set methods

🌟 Why It Matters?

- Each version makes JS **more powerful, readable, and modern**.
- Most modern browsers (and Node.js) support ES2015+ fully.
- Transpilers like **Babel** help older browsers run new features.

✍️ Short Notes

- **ECMAScript = JS standard**. New version every year.
- **ES2015 (ES6)** → biggest update (let, const, arrow, classes, promises, modules).

- **ES2016–ES2025** → added async/await, optional chaining, nullish coalescing, BigInt, top-level await, private fields, new Array/Set methods.
- Modern JS = mostly ES6+ features.
- Use **Babel** for backward compatibility.

Advanced JavaScript -

1. **Execution Context & Call Stack**

- JS runs inside an **Execution Context** → created in **two phases**:
  1. **Creation Phase** → Memory allocation (hoisting var/function).
  2. **Execution Phase** → Code runs line by line.
- Call Stack manages execution contexts (LIFO).

2. **Hoisting**

- Variables declared with var are **hoisted** (moved to top as undefined).
- let and const → hoisted but not initialized (Temporal Dead Zone).
- Functions → hoisted with full definition.

3. **Closures**

- A function **remembers** its lexical scope even after it's executed outside.
- Example:
- function outer() {
- let count = 0;
- return function inner() {
- count++;
- return count;
- };
- }
- const inc = outer();
- inc(); // 1
- inc(); // 2
- Used in data privacy, memoization, event handlers.

4. **Lexical Scope & Scope Chain**

- Lexical scope = where a variable is declared.
- Scope chain = inner functions can access variables of parent functions.

5. **this Keyword**

- Refers to the **execution context**.
- In global → this = window (browser).
- Inside object → this = object.
- In arrow function → this is taken from **lexical scope**.
- In strict mode → global this = undefined.

6. **Prototypes & Inheritance**

- Every JS object has a hidden property [[Prototype]].
- Prototypal inheritance → objects inherit methods from prototypes.
- Example:
- function Person(name) {
- this.name = name;
- }
- Person.prototype.sayHello = function() {
- return `Hi, I'm ${this.name}`;
- };
- const p = new Person("John");
- console.log(p.sayHello());

7. **Asynchronous JavaScript**

- JS is single-threaded but uses **Event Loop** to handle async tasks.
- **Callbacks** → functions passed into other functions.
- **Promises** → better async handling (.then, .catch).
- **async/await** → syntactic sugar for promises.

8. **Event Loop**

- Handles task queue & microtask queue.
- Priority: **Microtasks (Promises, MutationObserver)** > **Tasks (setTimeout, setInterval)**.

9. **Modules**

- **ES6 Modules** → import / export.
- Helps split code into smaller, reusable parts.

10. **Advanced Objects**

- Object.create() → create object with prototype.
- Object.keys(), Object.values(), Object.entries().
- Object.freeze() & Object.seal().

11. **Functional Programming Concepts**

- Higher-order functions (map, filter, reduce).
- Pure functions (no side effects).
- Immutability.
- Currying & Composition.

12. **Memory Management**

- Garbage Collection (GC) → automatic memory cleaning.
- Avoid memory leaks: unused event listeners, global variables, closures.

✍️ Short Notes

- **Execution Context** → Creation + Execution phase.
- **Hoisting** → var = undefined, let/const = TDZ, functions hoisted.
- **Closures** → inner fn remembers outer scope.
- **this** → depends on execution context (arrow fn = lexical).
- **Prototype** → inheritance system in JS.
- **Async JS** → callbacks → promises → async/await.
- **Event Loop** → microtasks before tasks.
- **Modules** → import/export for modular code.
- **Functional JS** → map, filter, reduce, pure functions.
- **Memory** → GC cleans unused data, avoid leaks.