Lecture 1 – History of JavaScript (JS)

🔹 Pehli Baat: JavaScript Me C++ Ya

Doosri Languages Ka Concept Mat Lagao

- **JavaScript** ek alag language hai — iska **syntax aur working** C++, Java, Python se kaafi alag hai.
- Agar tum **C++** jaise languages ke concepts yahan apply karoge, to **confusion ho sakta hai.**
- **JavaScript ki apni duniya hai** — use usi tarah samajhna hoga.

📌 *Beginner Tip*: Har language ka apna **syntax aur execution model** hota hai — blindly compare mat karo.

🔹 JavaScript Ki Zarurat Kyu Hai?

- **HTML** – Structure (Jaise: Skeleton)
- **CSS** – Style (Jaise: Clothes & Colors)
- **JavaScript** – Logic & Functionality (Jaise: Dimaag aur Actions)

JavaScript Se Kya Kya Kar Sakte Hain?

- Webpage me content ko **manipulate** kar sakte ho (DOM Manipulation).
- **Mathematical calculations** kar sakte ho.
- **Dynamic behavior** add kar sakte ho (form validation, animations, etc.)

📌 *Note*: **HTML** aur **CSS** se calculations ya dynamic tasks nahi ho sakte — yeh sirf **JavaScript** se possible hai.

🔹 JavaScript Ka Background (Story Style)

- Jab hum **console me 2 + 3** likhte hain, to browser JS ki madad se calculation karta hai.

- Har browser ke andar ek **JavaScript Engine** hota hai.
- **Google Chrome** me ye engine ka naam hai: **V8 Engine**

🔹 Kahani JavaScript Ki: 10 Din Ka Kamal

- 1995 me **Netscape** naam ki company ne ek scripting language banayi — sirf **10 din** me!
- Iska naam tha **LiveScript**, jo baad me **JavaScript** bana.

❌ **Galti Kya Hui?**

- Netscape ne kuch features ko **paid** bana diya.

🔹 Microsoft Ka JavaScript Copy-Paste Khel

- Microsoft ne **Internet Explorer** banaya.
- Netscape ke JS code ko **copy kiya**, thoda modify kiya aur launch kiya **JScript**.

📉 Is wajah se Netscape ka **downfall** shuru ho gaya.

🔹 Confusion: JavaScript vs JScript

.js
Example

```
// Netscape interpretation:
3++4  // 3^4 = 81

// Internet Explorer interpretation:
3++4  // 3^-4 = 1/81
```

- Alag browsers alag results de rahe the.
- Developers confused ho gaye: Kaunsa rule follow karein?

🔹 ECMA Ki Entry: Standardization

- Yeh confusion solve karne ke liye **ECMA (European Computer Manufacturers Association)** aayi.
- ECMA ne ek standard banaya: **ECMAScript**
- Aaj har browser **ECMAScript standards** ko follow karta hai.

📌 *Yaad Rakho*: JavaScript = Language, ECMAScript = Uska Standard

---

🔹 Netscape Ka End Aur Mozilla Ka Start

- Netscape doob rahi thi, to usne apna browser ka code **open-source** kar diya.
- **Mozilla** naam ke organization ne us code ko liya, improve kiya — aur ban gaya **Firefox**.

🔹 Google Chrome Aur V8 Engine Ka Jadoo

- **Google Chrome** ne entry ki with superfast **V8 Engine**
- **V8 Engine** JavaScript code ko **machine code** me convert karta hai – super fast!

📌 **V8 Engine** use hota hai:

- **Google Chrome** me
- **Node.js** me (browser ke bahar JS run karne ke liye)

🔹 Browser Ke Bahar JavaScript Kaise Chalayein?

- Browser ke andar to JS chalti hi hai.
- Lekin agar hume apne **computer ya server pe JS chalani ho**, to chahiye: **Node.js**

🔹 Node.js Kya Hai?

Node.js = V8 Engine + Extra Functionalities

- JavaScript ko **browser ke bahar** run karne ke liye use hota hai.
- File system, networking, database access jaise features deta hai.

🔹 Node.js Kaise Install Karein?

1. Visit: https://nodejs.org
2. Do versions milenge:
    - **LTS (Long Term Support)** – ✅ Recommended, stable
    - **Current Version** – Testing stage me hota hai
3. **LTS version download karo.**
4. CMD me likho:

bash
CopyEdit
```
node --version
```

Agar version show ho gaya, to installation successful hai.

🔹 Node.js Ka Use Kaha Hota Hai?

- **Frontend frameworks** me bhi (React, Angular)
- **Backend development** ke liye (Express.js, server creation)
- APIs, real-time applications, etc.

🔹 VS Code JavaScript Run Kyu Nahi Karta By Default?

- **VS Code ek text editor hai, compiler nahi.**
- Agar ye JS run kare to har language ke compiler ko bhi support karna padega (C++, Python, Java, etc.)
- Isse VS Code bohot **heavy** ho jaata.
- Isliye aapko **Node.js** alag se install karna padta hai.

🔁 Summary – Ek Nazar Me Revision

Topic	Key Point
HTML	Structure banata hai
CSS	Style deta hai
JavaScript	Logic aur functionality add karta hai
V8 Engine	Chrome ka JS engine – fast and powerful
JavaScript Banaya	Netscape ne 10 din me banaya
Microsoft JScript	JS ka copy, compatibility issues laaye
ECMA	JavaScript ko standard banane wali organization
Mozilla & Firefox	Netscape ke code ko use karke Firefox banaya
Node.js	JS ko browser ke bahar chalane ka solution
Node.js Install	nodejs.org se LTS version install karo
VS Code JS Issue	Text editor hone ke wajah se compiler included nahi

Lecture 2 – JavaScript Data Types (Part 1)

Why Are Data Types Needed?

JavaScript me data types isliye important hain kyunki ye decide karte hain ki **kisi variable me kis type ka data store hoga** aur uske sath kaunse operations possible hain.

Example:

- Instagram comment → *string*
- Like count → *number*
- Block status → *boolean*

.js
Example

```
let comment = "Nice pic!";
console.log(comment);           // Output: Nice pic!
console.log(typeof comment);    // Output: string
```

Types of Data in JavaScript

1. **Primitive Data Types** – *Simple & immutable*
2. **Non-Primitive Data Types** – *Complex like Arrays, Objects, Functions*

👉 Is lecture me sirf **Primitive Data Types** cover kiye gaye hain.

---

1. Number

- Represents **integers and decimals**
- **Use Case:** Age, balance, likes

.js
Example

```
let balance = 5000.75;
console.log(balance);           // Output: 5000.75
console.log(typeof balance);    // Output: number
```

---

2. String

- Represents **textual data** inside `' '`, `" "`, or `` ` ` ``
- **Use Case:** Username, comments, messages

.js
Example

```
let username = "Harshal";
console.log(username);           // Output: Harshal
console.log(typeof username);    // Output: string
```

.js
Example

```
let post = "Rohit is a bad boy";
console.log(post);               // Output: Rohit is a bad boy
console.log(typeof post);        // Output: string
```

---

3. Boolean

- Represents **true/false**
- **Use Case:** Login status, block status, admin check

.js
Example

```
let isBlocked = true;
console.log(isBlocked);          // Output: true
console.log(typeof isBlocked);   // Output: boolean
```

---

4. Undefined

- Variable is **declared but not assigned**
- **Use Case:** Input field empty, data not received

.js
Example

```
let searchInput;
console.log(searchInput);        // Output: undefined
```

```
console.log(typeof searchInput); // Output: undefined
```

---

5. Null

- **Intentionally empty** value
- **Use Case:** Server down response, unavailable data
  Jab **Server Down Hoga** Tab **0** (Zero) Ko Reply me Dega Nahi Isliye Waha Pe **Null** Send Kar Dega

.js
Example

```
let result = null;
console.log(result);            // Output: null
console.log(typeof result);     // Output: object
```

⚠️ **Note:**
`typeof null` returns `"object"` → **JavaScript bug** preserved for backward compatibility

🧠 `typeof null` → "object" क्यों होता है?

- **JavaScript** का एक पुराना **bug** है।
- पहले type **memory address** देखकर **detect** होता था — और `null` भी **object-type memory pattern** follow करता था।
- अगर इसे अब ठीक किया जाए तो **existing** पुराना **code** टूट सकता है।
- इसलिए **backward compatibility** के लिए इसे वैसे ही रखा गया है।

🟩 **Conclusion:**
`typeof null` **"object"** दिखाता है **—** ये एक पुरानी गलती है जिसे **compatibility** की वजह से अब भी **maintain** किया जाता है।

---

6. BigInt

- Used for **very large integers**
- **Use Case:** Aadhaar number, cryptographic keys

.js
Example

```
let bigValue = 1234567891234567891234567890n;
console.log(bigValue);           // Output:
1234567891234567891234567890n
console.log(typeof bigValue);    // Output: bigint
```

---

7. Symbol

- Represents a **unique and immutable identifier**
- **Use Case:** Object keys, hidden/private values

.js
Example

```
let userId = Symbol("id");
console.log(userId);             // Output: Symbol(id)
console.log(typeof userId);      // Output: symbol
```

Summary Table

Data Type	Definition	Example	typeof
Number	Integer or decimal values	`let x = 45.6`	`"number"`
String	Text in quotes	`let name = "Raj"`	`"string"`
Boolean	True/false value	`let isLive = false`	`"boolean"`
Undefined	Declared but not assigned	`let y;`	`"undefined"`
Null	Intentional absence of value	`let z = null;`	`"object"`
BigInt	Large integer values (suffix `n`)	`let big = 123n`	`"bigint"`
Symbol	Unique identifier	`let key = Symbol("uid")`	`"symbol"`

✨ Lecture 2: JavaScript Data Types – Part 2

🔹 (2) Non-Primitive Data Types (Reference Types)

➤ 1. Array

- **Definition:** Array ek ordered collection hota hai values ka — jo **index (0-based)** se access hoti hain.
- **Need:** Jab humein **multiple values** store karni ho **single variable** me, especially same type ya mixed values.

.js
Example

```
let arr = [10, 20, 30.2, "rohit", "mohit"];
console.log(arr);           // → [10, 20, 30.2, 'Harshal', 'Deepak']
console.log(typeof arr);    // → object
```

🔸 **Note:** `typeof arr` → `"object"` aata hai kyunki **array bhi JavaScript object** par hi based hota hai.

➤ 2. Object

- **Definition:** Object is a **collection of key-value pairs**. Har property ek key aur uski value se banti hai.
- **Need:** Jab humein **real-world entities** ko represent karna ho (e.g. user, car, product).

.js
Example

```
let obj = {
  user_name: "Harshal",
  account_number: 31242314213,
  balance: 420
};
console.log(obj);           // → {user_name: 'Rohit', account_number:
..., balance: 420}
console.log(typeof obj);    // → object
```

➤ 3. Function

- **Definition:** Function ek block of code hota hai jo **specific task** perform karta hai.
- **Need:** Jab code ko **reusable** banana ho ya **logic ko repeat** karna ho.

.js
Example

```
let fun = function() {
  console.log("Hello coder Army");
};
fun();                        // → Hello coder Army
console.log(typeof fun);      // → function
```

🔸 Type Conversions in JavaScript

Definition: Jab hum ek data type ke value ko **dusre type me convert** karte hain.

🧠 Types of Conversion:

🔹 1. Implicit Type Conversion (Type Coercion)

Automatically JavaScript karta hai

.js

Example

```
console.log("5" + 2);       // → '52' (number 2 converted to string)

console.log("5" - 2);       // → 3   (string "5" converted to number)

console.log(true + 1);      // → 2   (true → 1)
```

🔹 2. Explicit Type Conversion

Hum manually convert karte hain using functions like `Number()`, `String()`, `Boolean()`

✅ 1. String to Number

.js
Example
```
let account_balance = "100";
let num = Number(account_balance);
console.log(typeof account_balance);   // → string
console.log(typeof num);               // → number
```

❌ **Invalid Conversion**

.js
Example
```
let account = "100xs";
console.log(Number(account));          // → NaN
```

✅ 2. Boolean to Number

.js
Example
```
let x = true;
console.log(Number(x));                // → 1
```

✅ 3. Null to Number

.js
Example
```
let x1 = null;
console.log(Number(x1));               // → 0
```

✅ 4. Undefined to Number

.js
Example
```
let x2;
console.log(Number(x2));               // → NaN
```

✅ 5. Number to String

.js
Example
```
let ab = 20;
console.log(String(ab));               // → '20'
```

✅ 6. Boolean to String

.js
Example

```
let ax = true;
console.log(String(ax));                // → 'true'
```

✅ 7. String to Boolean

.js
Example

```
let abc = "str";
console.log(Boolean(abc));              // → true

console.log(Boolean(""));               // → false
console.log(Boolean(" "));              // → true
```

🔚 Short Summary: Key Learnings

- ✅ **Array**, **Object**, aur **Function** non-primitive types hain jo reference ke through kaam karte hain.
- ✅ `typeof` operator arrays ke liye `"object"` aur function ke liye `"function"` return karta hai.
- ✅ **Type Conversion** me:
  - `"100"` → number me convert hota hai.
  - `"100abc"` → NaN deta hai.
  - `true` → 1, `false` → 0
  - `null` → 0, `undefined` → NaN
  - `"hello"` → `true` (Boolean), `""` → `false`
  - `" "` (space) bhi `true` return karta hai Boolean conversion me.

✨📘 LECTURE 3 & 4 – JAVASCRIPT OPERATORS

🔹 Operators Kya Hote Hain?

Operators wo **symbols** hote hain jo **values ya variables par operations** perform karte hain.
👉 Example: `+`, `-`, `*`, `==`, `&&`, etc.

🔹 1. Arithmetic Operators

👉 Mathematical calculations ke liye use hote hain.

Operator	Description	Example	Result
`+`	Addition	`5 + 3`	8
`-`	Subtraction	`5 - 3`	2
`*`	Multiplication	`5 * 3`	15
`/`	Division	`10 / 2`	5
`%`	Modulus (Remainder)	`10 % 3`	1
`++`	Increment	`x++` / `++x`	-
`--`	Decrement	`x--` / `--x`	-
`**`	Exponentiation	`2 ** 3`	8

📌 **Order of Priority (BODMAS Rule):**

- Multiply, Divide → Left to Right
- Add, Subtract → Left to Right

```
console.log(6 * 3 + 18 / (6 - 9));        // Bad way
console.log(((6 * (3 + 18)) / (6 - 9)));  // Better way
```

👉 **Increment & Decrement Example:**

```
let sum = 20;
```

```
console.log(sum++); // 20 (post increment)
console.log(sum--); // 21 (post decrement)

let num = 23;
++num;
console.log(num);   // 24 (pre increment)
console.log(--num); // 23 (pre decrement)
```

🔹 2. Assignment Operators

Variable me **value assign/update** karne ke liye use hote hain.

Operator	Description	Example	Equivalent To
`=`	Assign	`x = 5`	`x = 5`
`+=`	Add & assign	`x += 3`	`x = x + 3`
`-=`	Subtract & assign	`x -= 3`	`x = x - 3`
`*=`	Multiply & assign	`x *= 4`	`x = x * 4`
`/=`	Divide & assign	`x /= 2`	`x = x / 2`
`%=`	Modulus & assign	`x %= 3`	`x = x % 3`
`**=`	Power & assign	`x **= 2`	`x = x ** 2`

🔹 3. Comparison Operators

👉 Do values ko compare karte hain.

```
let a1 = 10;
let a2 = 20;

console.log(a1 == a2);   // false
console.log(a2 > a1);    // true
console.log(a2 < a1);    // false
console.log(a1 >= a2);   // false
console.log(a1 <= a2);   // false
```

📌 **Loose vs Strict Equality:**

- `==` → Type conversion ke baad compare karta hai.
- `===` → Type + Value dono compare karta hai.

```
let a1 = 10;
let str = "10";

console.log(a1 == str);   // true (string → number)
console.log(a1 === str);  // false (number ≠ string)

console.log(null == undefined);   // true
console.log(null === undefined);  // false
```

⚠ **Special Case:**

```
console.log(null <= 0); // true
console.log(null >= 0); // true
console.log(undefined == 0); // false
```

🔹 4. Logical Operators

Logical operators wo operators hote hain jo multiple conditions (true/false values) ko combine karke ek single boolean result (true ya false) dete hain.

👉 Ye mainly **decision making** aur **conditional statements** (if-else, loops) me use hote hain.

Operator	Name	Description	Example	Result
`&&`	Logical AND	True if both true	`5 > 2 && 2 > 1`	true
||	Logical OR	`	Logical OR	True if one true
`!`	Logical NOT	Inverts value	`!true`	false

🔹 5. Bitwise Operators

Bitwise operators wo operators hote hain jo numbers ke binary (0 aur 1) representation par operation perform karte hain, bit by bit.

Operator	Name	Description	Example	Result

`&`	AND	1 if both bits 1	`5 & 3`	1
|	OR	OR	1 if any bit 1	`5
`^`	XOR	1 if bits different	`5 ^ 3`	6
`~`	NOT	Inverts bits	`~5`	-6
`<<`	Left Shift	Shifts bits left	`5 << 1`	10
`>>`	Right Shift	Shifts bits right	`5 >> 1`	2

🔹 6. NaN Comparison

⚠ **Important:** `NaN == NaN → false`

```
let str3 = "rohit";
let str4 = "mohit";

console.log(Number(str3) == Number(str4));
// NaN == NaN → false
```

🔹 7. Multiple Comparisons

```
let abc1 = 123;
let abc2 = "123";
let abc3 = 123;

console.log(abc1 == abc2 == abc3);
// Step 1: 123 == "123" → true
// Step 2: true == 123 → false

abc3 = true;
console.log(abc1 == abc2 == abc3);
// Step 1: 123 == "123" → true
// Step 2: true == true → true
```

🔁 SHORT SUMMARY – EK NAZAR ME

✅ **Arithmetic Operators** → Maths ke liye (+, -, *, /, %, ++, --, **)
✅ **Assignment Operators** → Value assign/update (+=, -=, *=, etc.)
✅ **Comparison Operators** → ==, ===, <, >, <=, >=

✅ **Logical Operators** → &&, | |, !
✅ **Bitwise Operators** → Bits par operations (&, |, ^, <<, >>)
✅ **null & undefined** → Special cases in comparison
✅ **NaN** → Kabhi equal nahi hota (even with itself)
✅ **Multiple comparisons** → Left to right evaluate hote hain

✨📘 LECTURE 5 – MEMORY IN JAVASCRIPT

🔹 Primitive vs Non-Primitive Data Types

- **Primitive (Immutable)** → Value **change** नहीं होती, new memory ban जाती है.
- **Non-Primitive (Mutable)** → Value **change** हो सकती है, same memory par update hota hai.

Example:

```
let obj1 = { id: 20, naming: "Rohit" };
let obj2 = obj1;

obj2.id = 30;

console.log(obj1); // {id:30, naming:"Rohit"}
console.log(obj2); // {id:30, naming:"Rohit"}
```

👉 Both point to **same memory address**, so changes reflect in both.

🔹 Stack vs Heap Memory

- **Stack** → Primitive data type (Number, String, Boolean, null, undefined, Symbol, BigInt).
- **Heap** → Non-primitive data type (Array, Object, Function).

Example:

```
let a = 10;
let b = a;
b = 50;

console.log(a); // 10 (No change)
console.log(b); // 50
```

👉 **Call by Value** → Changes affect only `b`, not `a`.

```
let obj1 = { id: 20, name: "Rohit" };
let obj2 = obj1;
```

```
obj2.id = 30;

console.log(obj1); // {id:30, name:"Rohit"}
```

👉 **Call by Reference** → Both share same memory address.

🔹 Why Not Store Primitives in Heap?

- **Stack** → छोटी memory, fast access.
- **Heap** → बड़ी memory, flexible storage.
- Primitive → छोटे होते हैं, इसलिए **Stack me**.
- Non-Primitive → बड़े होते हैं, इसलिए **Heap me**.

🔹 Example: Const with Primitive vs Non-Primitive

```
// Primitive
const num = 10;
num = 20; // ❌ Error (immutable)

// Non-Primitive
const obj = { id: 10, balance: 234 };
obj.id = 20; // ✅ Allowed
console.log(obj);
```

👉 Primitive const **re-assign** नहीं कर सकते,
Non-primitive const → **properties change** कर सकते हो.

🔹 Why Primitive Data Types Are Immutable?

```
let a = 10;
let c = a;
c = 50; // New memory created for c
```

👉 Agar new value zyada memory लेती है (e.g., string), JS नया memory location create करता है.

⚡ In **C++/Java** → fixed type hote hain (int = only integer),
इसलिए waha JS jaisa memory management नहीं होता.

🔹 Why Do We Need Address?

- Memory is **byte addressable** → हर byte का unique address होता है.
- Address se hum fast access kar paate hain aur duplicate values ka confusion nahi hota.

Scenario 1: Accessing 78

- Without address → line by line search → **slow**.
- With address → direct access → **fast**.

Scenario 2: Duplicate Values

- Agar 78 do jagah hai → confusion.
- Address ensures → unique identification.

⚡ **Summary:**

- Primitive → Stored in **Stack**, immutable.
- Non-Primitive → Stored in **Heap**, mutable.
- Stack = fast, Heap = flexible.
- Addressing system makes memory access efficient.

✨📘 LECTURE 6 – STRING IN JAVASCRIPT

JavaScript me **string ek sequence of characters (text)** hota hai jo `'single quotes'`, `"double quotes"`, ya `` `backticks` `` ke andar likha jata hai.

🔹 Ways to Declare String

```
let singleQuote = 'Hello, world';

let doubleQuote = "Hello, world";

let backticks = `Hello, world`;
```

👉 Backticks (`) me string ke andar variable interpolation hota hai.

```
let price = 80;

console.log(`Price of the tomato is ${price}`);

// Price of the tomato is 80
```

🔹 String Concatenation

```
let s1 = "hello";

let s2 = " world";

let s3 = s1 + s2;


console.log(s3);  // "hello world"
```

🔹 Length of String

```
console.log(s1.length); // 5
```

```
console.log(s2.length); // 6

console.log(s3.length); // 11
```

🔹 Access Characters

```
let special = "Rohit";


console.log(special[0]);        // 'R'

console.log(special.charAt(3)); // 'i'
```

🔹 Change Case

```
console.log(special.toLowerCase()); // 'rohit'

console.log(special.toUpperCase()); // 'ROHIT'
```

🔹 Searching in Strings

- `indexOf(substring)` → Pehli occurrence ka index
- `lastIndexOf(substring)` → Aakhri occurrence ka index
- `includes(substring)` → Exists ya nahi check karta hai

```
let hero = "Hello Coder Coder Army";


console.log(hero.indexOf("Coder"));     // 6

console.log(hero.lastIndexOf("Coder")); // 12

console.log(hero.includes("Coder"));    // true
```

🔹 Extracting Substrings

- `slice(start, end)` → Negative index allowed
- `substring(start, end)` → Negative index allowed **nahi**

```
let newstr = "HelloDon";


console.log(newstr.slice(0, 3));      // 'Hel'
console.log(newstr.substring(0, 3)); // 'Hel'
console.log(newstr.slice(-6, -3));   // 'llo'
console.log(newstr.slice(-2, 4));    // ''
```

🔹 Replacing Contents

```
let str10 = "hello world world acha hai";


console.log(str10.replace("world", "duniya"));
// "hello duniya world acha hai"


console.log(str10.replaceAll("world", "duniya"));
// "hello duniya duniya acha hai"
```

🔹 Splitting Strings

```
let str2 = "money, honey, sunny, funny";
```

```
console.log(str2.split(", "));

// ['money', 'honey', 'sunny', 'funny']
```

🔹 Trimming Strings

```
let str3 = "   Coder Army   ";


console.log(str3.trim());      // "Coder Army"

console.log(str3.trimStart()); // "Coder Army   "

console.log(str3.trimEnd());   // "   Coder Army"
```

🔹 Extra Useful Methods

```
let text = "JavaScript is fun";


console.log(text.startsWith("Java")); // true

console.log(text.endsWith("fun"));    // true

console.log(text.repeat(2));          // "JavaScript is funJavaScript
is fun"


console.log("7".padStart(3, "0"));    // "007"

console.log("7".padEnd(3, "0"));      // "700"
```

🔹 New Way to Create String

```
let latestString = new String("hello coder army");
```

```
console.log(typeof latestString); // object
```

👉 Ye string **Heap memory** me store hoti hai, primitive string se different.

🔁 SHORT SUMMARY – EK NAZAR ME

✅ Declare string → `'single'`, `"double"`, `` `backtick` ``
✅ Concatenation → `+` operator se
✅ Length → `.length`
✅ Access → `str[index]`, `charAt()`
✅ Case change → `toUpperCase()`, `toLowerCase()`
✅ Search → `indexOf`, `lastIndexOf`, `includes`
✅ Extract → `slice`, `substring`
✅ Replace → `replace`, `replaceAll`
✅ Split → `split()` array banata hai
✅ Trim → `trim`, `trimStart`, `trimEnd`
✅ Extra → `startsWith`, `endsWith`, `repeat`, `padStart`, `padEnd`
✅ New way → `new String()` (object banata hai)

✨📘 LECTURE 7 – NUMBERS & MATH IN JAVASCRIPT

🔹 Numbers in JavaScript

👉 JavaScript me numbers **primitive type** hote hain. Do tarike se declare kar sakte ho:

```
let num1 = 231;

console.log(typeof num1); // "number"


let num2 = new Number(231);

console.log(typeof num2); // "object"
```

📌 **Difference:**

- **num1** ek primitive number hai.
- **num2** ek object hai (Number wrapper class se bana).

```
let num1 = 231;

let num2 = new Number(231);

let num3 = new Number(231);


console.log(num1 == num2); // true (object → primitive convert)

console.log(num2 == num3); // false (alag memory address)
```

🔹 Number Methods

1️⃣ toFixed(n) → Fixed decimal places me convert karta hai

```
let num = 231.689;
```

```
console.log(num.toFixed(1)); // "231.7"
```

2️⃣ toPrecision(n) → Number ko specified length tak format karta hai

```
let num = 231.689;

console.log(num.toPrecision(5)); // "231.69"
```

3️⃣ toString() → Number ko string me convert karta hai

```
let num = 42;

console.log(num.toString()); // "42"
```

4️⃣ toExponential(n) → Exponential (scientific) notation me convert karta hai

```
let num = 12345.6789;

console.log(num.toExponential(2)); // "1.23e+4"
```

🔹 Math Object in JavaScript

👉 JavaScript ek global **Math object** deta hai jo advanced calculations ke liye use hota hai.

📌 Constants

```
console.log(Math.PI);   // 3.14159...

console.log(Math.E);    // 2.718 (Euler's number)

console.log(Math.LN10); // 2.302 (log base e of 10)
```

📌 Rounding Methods

```
console.log(Math.round(4.6)); // 5 (nearest integer)

console.log(Math.floor(4.9)); // 4 (round down)

console.log(Math.ceil(4.1));  // 5 (round up)
```

📌 Power & Square Root

```
console.log(Math.pow(2, 3));  // 8

console.log(Math.sqrt(16));   // 4
```

📌 Min & Max

```
console.log(Math.min(5, 2, 9)); // 2

console.log(Math.max(5, 2, 9)); // 9
```

🔹 Math.random()

👉 0 (inclusive) se 1 (exclusive) ke beech ek pseudo-random number generate karta hai.

```
console.log(Math.random());          // 0.123456...

console.log(Math.random() * 10);     // 0 se 9 ke beech

console.log(Math.floor(Math.random() * 10)); // integer 0–9

console.log(Math.floor(Math.random() * 10) + 1);  // integer 1–10

console.log(Math.floor(Math.random() * 10) + 11); // integer 11–20
```

📌 General Formula:

```
Math.floor(Math.random() * (max - min + 1)) + min
```

Example:

```
console.log(Math.floor(Math.random() * (40 - 30 + 1)) + 30);

// Random number between 30 and 40
```

🔁 SHORT SUMMARY – EK NAZAR ME

✅ **Numbers** → Primitive (`typeof = number`) aur Object (`new Number()`)
✅ **Number Methods** → toFixed(), toPrecision(), toString(), toExponential()
✅ **Math Object** → Constants (PI, E, LN10) + Methods (round, floor, ceil, pow, sqrt, min, max)
✅ **Math.random()** → Random number generate karne ke liye
✅ **Formula** → `(Math.random() * (max - min + 1)) + min`

✨📘 LECTURE 8 – ARRAYS IN JAVASCRIPT

🔹 What is an Array?

👉 Array ek **dynamic data structure** hai jo **ordered collection of values** ko store karta hai.
👉 Values kisi bhi type ki ho sakti hain → **numbers, strings, objects, even other arrays**.

```
let arr = [2, 3, 4, 1, 89, "Harshal", true];
```

💡 **Tip**: Array ke indexes **0 se start** hote hain.

🔹 Array Length

👉 **Definition**: Array me kitne elements hain uska count deta hai.

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

console.log(arr.length); // 3
```

⚠️ **Note**: `length` property **mutable hai**, agar tum `arr.length = 1` karoge to array truncate ho jayega.

Agar tum array ki length ko **chhota set karte ho**, to array ke extra elements **delete** ho jate hain.

🔹 Accessing Elements

👉 **Definition**: Array ke element ko index number (0 se start) ya `.at()` method se access karte hain.

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

console.log(arr[0]);     // 1

console.log(arr.at(1));  // 2

console.log(arr.at(-1)); // 3 (last element)
```

💡 **Tip**: `.at(-1)` ka use karke easily **last element** le sakte ho.

🔹 Copying Arrays

👉 **Definition**: Agar ek array ko dusre variable me assign karo to reference copy hoti hai (same address).

```
const arr = [1, 2, 3];

const newArr = arr;

console.log(newArr == arr); // true
```

👉 **Independent copy** (different memory):

```
const newArr = structuredClone(arr);

console.log(newArr == arr); // false
```

⚠️ **Important**: Direct assignment karne se array **by reference copy hota hai**, isliye ek array me change dusre me bhi reflect hoga.

🔹 Common Array Methods

1️⃣ push()

👉 **Definition**: Array ke **end me element add karta hai**.

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

arr.push(40);

console.log(arr); // [10, 20, 30, 40]
```

2️⃣ pop()

👉 **Definition**: Array ke **last element ko remove karta hai**.

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

```
arr.pop();

console.log(arr); // [10, 20]
```

3️⃣ unshift()

👉 **Definition**: Array ke **starting me element add karta hai**.

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

arr.unshift(50);

console.log(arr); // [50, 10, 20, 30]
```

4️⃣ shift()

👉 **Definition**: Array ke **pehle element ko remove karta hai**.

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

arr.shift();

console.log(arr); // [20, 30]
```

5️⃣ delete (⚠️ Not Recommended)

👉 **Definition**: Specific index se element delete karta hai, lekin empty slot chhod deta hai.

```
let arr = [10, 20, 30, 40];

delete arr[0];

console.log(arr); // [empty, 20, 30, 40]
```

⚠️ **Note**: `delete` avoid karo, kyunki array ke length ko change nahi karta aur hole (undefined) chhod deta hai.

6️⃣ indexOf()

👉 **Definition**: Array me element ki **first occurrence ka index return karta hai**.

```
let arr = [2, 1, 4, 8, 1];

console.log(arr.indexOf(1)); // 1
```

7️⃣ lastIndexOf()

👉 **Definition**: Array me element ki **last occurrence ka index return karta hai**.

```
let arr = [2, 1, 4, 1, 8, 1];

console.log(arr.lastIndexOf(1)); // 5
```

8️⃣ includes()

👉 **Definition**: Check karta hai ki element array me exist karta hai ya nahi.

```
let arr = [2, 4, 6, 8];

console.log(arr.includes(6)); // true
```

🔹 slice() vs splice()

slice()

👉 **Definition**: Array ka ek **part copy karke new array return karta hai** (original change nahi hota).

```
let arr = [2, 4, 6, 8, 10, 12];

console.log(arr.slice(2, 5)); // [6, 8, 10]
```

💡 **Tip**: `slice(-n)` last ke `n` elements nikalne ka shortcut hai.

🔹 splice()

Array ko modify karta hai (add, remove, ya replace).

⚠️ **Important: splice() original array ko modify karta hai, jabki slice() nahi kart**

```
let arr = [2, 4, 6, 8, 10];


// 1️⃣ Remove elements

console.log(arr.splice(2, 2)); // [6, 8] → ye removed elements hain

console.log(arr);              // [2, 4, 10] → original array modify
ho gaya


// 2️⃣ Add elements

arr.splice(1, 0, "coder");

console.log(arr); // [2, "coder", 4, 10]


// 3️⃣ Replace elements

arr.splice(2, 1, "JS");

console.log(arr); // [2, "coder", "JS", 10]
```

⚡ Explanation

1. **`arr.splice(2, 2)`**
    - Index `2` se shuru karke `2` elements remove kiye (→ `[6, 8]`).
    - Original array ab `[2, 4, 10]`.
2. **`arr.splice(1, 0, "coder")`**
    - Index `1` par `"coder"` insert hua.
    - Koi element remove nahi hua (deleteCount = 0).
    - Array ban gaya `[2, "coder", 4, 10]`.

3. `arr.splice(2, 1, "JS")`
    - Index `2` par ek element (value `4`) delete karke `"JS"` se replace kiya.
    - Array ban gaya `[2, "coder", "JS", 10]`.

👉 Is tarah `splice()` **remove + add + replace** sab kaam kar sakta hai.

🔹 toString() & join()

toString()

👉 Array ko **comma-separated string** me convert karta hai.

```
let arr = [2, 4, 6];

console.log(arr.toString()); // "2,4,6"
```

join()

👉 Custom separator ke sath array elements ko string banata hai.

```
let arr = [2, 4, 6];

console.log(arr.join(" * ")); // "2 * 4 * 6"
```

🔹 concat(), 2D Array & flat(), isArray()

concat()

👉 Do ya zyada arrays ko merge karta hai.

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

let arr2 = [3, 4];

console.log(arr1.concat(arr2)); // [1, 2, 3, 4]
```

2D Array

👉 Array ke andar array (Matrix jaisa structure).

```
let arr = [[1,2,3], [4,5,6], [7,8,9]];

console.log(arr[2][2]); // 9
```

---

flat()

👉 Nested array ko flatten karta hai.

```
let arr = [1, [2, 3], [4, 5]];

console.log(arr.flat()); // [1, 2, 3, 4, 5]
```

⚡ Advanced:

- `flat(1)` → 1 level flatten
- `flat(2)` → 2 levels flatten (3D ko 1D)
- `flat(Infinity)` → kitne bhi depth flatten kar dega

3D Array

```
let arr3D = [[[1,2],[3,4]], [[5,6],[7,8]], [9,10]];

console.log(arr3D.flat(2));

// [1,2,3,4,5,6,7,8,9,10]
```

---

isArray()

👉 Check karta hai ki value array hai ya nahi.

```
console.log(Array.isArray([1,2,3])); // true

console.log(Array.isArray("hello")); // false
```

🔹 Other Ways to Create Array (⚠️ Not Recommended)

```
let arr1 = new Array(2, 3, 4);

console.log(arr1); // [2, 3, 4]


let arr2 = new Array(5);

console.log(arr2); // [empty × 5]
```

⚠️ Agar ek hi number diya to wo length treat hota hai, element nahi.

🔹 Memory Difference (C++ vs JS)

- **C++ Arrays:** Fixed size, contiguous memory.
- **JavaScript Arrays:** Heap me store hote hain, contiguous memory ki guarantee nahi hoti.

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

arr[0] = "Hello";  // Allowed
```

🔁 SHORT SUMMARY – EK NAZAR ME

✅ push() → end me add
✅ pop() → last remove
✅ unshift() → start me add
✅ shift() → start remove
✅ slice() → part copy (no modify)
✅ splice() → part modify (add/remove)
✅ concat() → merge arrays
✅ 2D Array → arrays ke andar array (matrix jaisa)
✅ flat() → nested ko flat (Infinity = any depth)
✅ join() / toString() → array → string
✅ indexOf / lastIndexOf / includes → searching

✅ isArray() → check array or not
✅ delete → avoid karo (hole banata hai)
✅ JS arrays flexible hote hain, mixed data store kar sakte hain.

📘 LECTURE 9 – DATE IN JAVASCRIPT

🔹 What is Date in JavaScript?

👉 JavaScript me **Date object** ka use **date aur time ke saath work karne ke liye** hota hai.
👉 Ye alag-alag methods deta hai dates ko **create, manipulate aur format** karne ke liye.

```
const d = new Date();

console.log(d);

// Example: 2024-11-29T15:34:51.729Z
```

⚠️ **Note**: `Date` object ka **typeof** → **"object"** hota hai.

🔹 From Where Does Date Come?

👉 JavaScript apne **system clock** se date nikalta hai.
👉 Internally ye time ko **milliseconds me count karta hai** →
`Jan 1, 1970, 00:00:00 UTC (Unix Epoch)` se.

```
const d1 = new Date(1000);

console.log(d1);

// 1970-01-01T00:00:01.000Z (1000 ms later)
```

💡 **Tip**: Agar `new Date(2000)` doge → 2 sec ke baad ka time dikhayega.

🔹 Creating Date

1️⃣ Current Date

```
const d = new Date();

console.log(d.toString());
```

2️⃣ Custom String Date

```
const d = new Date("2022-10-20");

console.log(d); // 2022-10-20T00:00:00.000Z
```

3️⃣ **Custom Date + Time**

```
const d = new Date("2022-10-20T10:10:12");
```

4️⃣ **Year, Month, Date Format**

```
const d = new Date(2024, 4, 28);

// Month 0-based → 4 = May

console.log(d.toDateString()); // Tue May 28 2024
```

⚠️ **Note**: `new Date(year, month, date, hours, minutes, seconds, ms)`

- **First 2 compulsory** (year & month).
- Agar ek hi value doge → milliseconds maana jayega.

🔹 Formatting Dates

```
const d = new Date();

console.log(d.toDateString()); // Fri Nov 29 2024

console.log(d.toString());     // Fri Nov 29 2024 21:06:20 GMT+0530

console.log(d.toISOString());  // 2024-11-29T15:36:49.659Z
```

💡 **Tip**: `.toISOString()` API development me useful hota hai kyunki ye **universal format** deta hai.

🔹 Common Methods

```
const d = new Date();
```

```
console.log(d.getFullYear()); // 2024
console.log(d.getMonth());    // 10 (Nov, 0-based)
console.log(d.getDate());     // 29
console.log(d.getDay());      // 5 (Friday, 0=Sunday)
console.log(d.getHours());    // 21
console.log(d.getMinutes());  // 06
console.log(d.getSeconds());  // 20
console.log(d.getMilliseconds()); // 729
```

⚠️ **Important**:

- `getMonth()` → 0 = Jan, 11 = Dec.
- `getDay()` → 0 = Sunday, 6 = Saturday.

🔹 Milliseconds (Why Important?)

👉 Date object internally sab kuch **milliseconds** me calculate karta hai.

```
console.log(d.getTime());
// Milliseconds since Jan 1, 1970
```

👉 **Use Case**:
Ticket booking me, agar do log ek hi din booking karen:

- Person 1 → `23815679 ms`
- Person 2 → `23815678 ms`

➡ Person 2 ne **1 ms pehle** booking ki, jo only milliseconds se hi pata chalega.

✅ **Milliseconds ka use**:

- Precise time difference

- Easy calculations
- Sorting time-based data

```
const now = Date.now();

console.log(now); // Current timestamp in ms
```

🔹 Setting Date Components

👉 Date ko modify bhi kar sakte ho.

```
const d = new Date();

d.setDate(20);

d.setFullYear(2021);

d.setMonth(3); // (April, 0-based)


console.log(d.toString());

// Tue Apr 20 2021 21:41:24 GMT+0530
```

💡 **Tip**: Agar tum **range se bahar values** doge to JS automatically adjust kar deta hai.
Example:

```
let d = new Date(2024, 0, 35);

console.log(d.toDateString()); // Mon Feb 04 2024
```

🔹 Short Summary – Ek Nazar Me

✅ `new Date()` → Current Date
✅ `new Date(ms)` → From Epoch milliseconds
✅ `new Date("YYYY-MM-DD")` → From string
✅ `getFullYear()` / `setFullYear()` → Year
✅ `getMonth()` / `setMonth()` → Month (0–11)

✅ `getDate()` / `setDate()` → Day of month (1–31)
✅ `getDay()` → Weekday (0–6)
✅ `getTime()` / `Date.now()` → Milliseconds
✅ `toDateString()` / `toISOString()` → Formatting

📘 Lecture 10 – Objects in JavaScript (Part-1)

🔹 What is an Object?

👉 **Object ek collection of key-value pairs hota hai.**

- Keys → properties ke naam (always string type internally)
- Values → data (string, number, boolean, object, function, etc.)
- JS me real world entities ko represent karne ke liye object ka use hota hai.

Example:

```
const obj = {
  name: "Bhupendar Jogi",
  "account-balance": 150,
  gender: "male",
  age: 20
};
console.log(obj);
```

⚡ **Important:**

- Keys hamesha **string form me store** hoti hain backend me.
- Agar key me **space ya special char** ho to usse " " me likhna zaroori hai.

❌ `account number: 10500`
✅ `"account number": 10500`

🔹 Accessing Object Properties

1️⃣ **Dot Notation**

```
console.log(obj.name);    // "Bhupendar Jogi"
console.log(obj.age);     // 20
```

2️⃣ **Bracket Notation**

```
console.log(obj["gender"]);   // "male"
console.log(obj["account-balance"]); // 150
```

⚡ **Tips:**

- Agar key me **space/special char** ho → bracket notation use karna hi padega.
- Bracket notation me key ko " " me likhna padta hai.

🔹 Number Keys in Objects

```
const obj = {
  0: "zero",
  1: "one"       // O index Value →Key - Value Number to [] square bracket se access hoga
};
console.log(obj[0]); // "zero"
```

🔹 Special Keys

```
const obj = {
  undefined: 50,
  null: "Harshal"
};

console.log(obj.undefined);  // 50
console.log(obj["null"]);    // "Harshal"
```

🔹 Ways to Create Objects

🔹 1. Object Literal

👉 **Sabse common aur simple tarika** object banane ka. Direct { } braces use karke key-value pairs define karte hain.

```
const person = {
```

```
  name: "Harshal",
  age: 30
};

console.log(person.name); // Harshal
console.log(person.age);  // 30
```

✅ Pros

- Simple & quick
- Readable syntax
- Small data ke liye perfect

❌ Cons

- Same structure ke multiple objects banana ho to repetitive code likhna padta hai.

🔹 2. Object Constructor (Built-in Object class)

👉 `new Object()` use karke ek **empty object** banate hain, aur phir properties manually add karte hain.

```
const person = new Object();
person.name = "Alice";
person.age = 30;

console.log(person);
// { name: 'Alice', age: 30 }
```

✅ Pros

- Flexibility: pehle object banao, baad me properties add kar sakte ho.
- Dynamic object creation ke liye useful.

❌ Cons

- Syntax thoda verbose hai.
- Object literal ke comparison me zyada use nahi hota.

🔹 3. Constructor Function (Custom Function)

👉 JavaScript me ek **custom function** banakar uska use object banane ke liye karte hain. `new` keyword ke sath call karte ho.

```
function Person(name, age) {
  this.name = name;
  this.age = age;
}

let p1 = new Person("Alice", 20);
let p2 = new Person("Bob", 30);

console.log(p1); // { name: 'Alice', age: 20 }
console.log(p2); // { name: 'Bob', age: 30 }
```

✅ Pros

- Multiple objects create karne ke liye reusable.
- Constructor function ek **template** ki tarah kaam karta hai.

❌ Cons

- Prototype chaining samajhna thoda mushkil ho sakta hai beginners ke liye.

🔹 4. Class (ES6 OOPs style)

👉 `class` syntax ek **syntactic sugar** hai jo constructor functions aur prototypes ke upar bana hai.
👉 Ye **modern, cleaner & OOP style** provide karta hai.

```
class People {
  constructor(name, age, gender) {
    this.name = name;
    this.age = age;
    this.gender = gender;
  }
}

let p1 = new People("Alice", 20, "male");
```

```
let p2 = new People("Bob", 30, "male");

console.log(p1);
// { name: 'Alice', age: 20, gender: 'male' }
```

✅ Pros

- OOPs style → `constructor`, `methods`, `inheritance` support karta hai.
- Code reusable aur clean hota hai.
- Industry standard (modern JS).

❌ Cons

- Thoda advanced concept hai beginners ke liye.

🔹 Modifying Objects

```
let person = { name: "Saurav", age: 30 };

// Add
person.gender = "male";

// Update
person.age = 31;

// Delete
delete person.name;

console.log(person);   // { age: 31, gender: "male" }
```

🔹 Common Object Methods

1️⃣ **Object.keys(obj)** → properties ka array deta hai

```
Object.keys(person); // ["age", "gender"]
```

2️⃣ **Object.values(obj)** → values ka array deta hai

```
Object.values(person); // [31, "male"]
```

3️⃣ **Object.entries(obj)** → key-value pairs ka array deta hai

```
Object.entries(person);
// [["age", 31], ["gender", "male"]]
```

4️⃣ **Object.assign(target, source)** → ek object ki properties dusre me copy karta hai

```
const obj1 = { a: 1 };
const obj2 = { b: 2 };
const obj3 = Object.assign(obj1, obj2); // Aise obj1 me change ho jata
ha isiliye {} ka use karte hai

const obj4 = Object.assign({}, obj1, obj2); // obj1 aur obj ki value
ko combine kar ke {} empty object me add kar dega
console.log(obj4); // { a: 1, b: 2 }
```

⚠ Agar `target` me direct object pass kar do to wo bhi change ho jata hai.
Isliye hamesha `{}` pass karo.

🔹 Spread Operator

👉 Spread operator (`...`) ka use karke hum easily **multiple objects ko merge** kar sakte hain.

```
let obj1 = { name: "Harshal", age: 21 };
let obj2 = { city: "Varanasi", country: "India" };

// Spread operator se merge
let obj3 = { ...obj1, ...obj2 };

console.log(obj3);
```

Output:

```
{ name: 'Harshal', age: 21, city: 'Varanasi', country: 'India' }
```

🔹 Conflict Case (same property overwrite hota hai)

```
let objA = { name: "Harshal", age: 21 };
let objB = { age: 25, city: "Mumbai" };

let merged = { ...objA, ...objB };
```

```
console.log(merged);
```

Output:

```
{ name: 'Harshal', age: 25, city: 'Mumbai' }
```

👉 Yahan `age` property pehle `21` thi (`objA` me),
but `objB` ne overwrite karke `25` kar diya.
⚡ Rule: **Jo baad me spread hota hai, wahi final value hoti hai.**

💡 **Tip:** Spread operator object cloning me bhi use hota hai, jisse shallow copy ban jata hai.

🔷 Object.freeze() vs Object.seal()

Feature	Object.freeze() ❄️	Object.seal() 🔒
Add properties	❌ Not allowed	❌ Not allowed
Delete properties	❌ Not allowed	❌ Not allowed
Modify values	❌ Not allowed	✅ Allowed
Nested objects	Manual freeze required	Manual seal required

✅ Examples

1️⃣ `Object.freeze()`

```
let user = { name: "Harshal", age: 21 };

Object.freeze(user);

user.age = 22;         // ❌ Change not allowed
user.city = "Varanasi"; // ❌ Add not allowed
delete user.name;      // ❌ Delete not allowed

console.log(user);
```

Output:

```
{ name: "Harshal", age: 21 }
```

👉 Freeze ke baad **kuch bhi change nahi hota**.

2️⃣ `Object.seal()`

```
let person = { name: "Harshal", age: 21 };

Object.seal(person);

person.age = 22;       // ✅ Allowed (update ho jayega)
person.city = "Mumbai"; // ❌ Add not allowed
delete person.name;    // ❌ Delete not allowed

console.log(person);
```

Output:

```
{ name: "Harshal", age: 22 }
```

👉 Seal ke baad **sirf update allowed hai**, add/delete nahi.

🔹 Shallow Copy vs Deep Copy

- **Shallow Copy:** sirf reference copy hota hai → dono object ek hi memory ko point karte hain.

```
let obj1 = { a: 1 };
let obj2 = obj1;
obj2.a = 10;
console.log(obj1.a); // 10
```

- **Deep Copy:** alag memory address create hota hai.

```
let obj1 = { a: 1, b: 2 };
let obj2 = structuredClone(obj1);
obj2.a = 20;

console.log(obj1, obj2);
// { a: 1, b: 2 } { a: 20, b: 2 }
```

⚠ **Nested Objects me shallow copy dikkat deta hai**.
Isliye hamesha `structuredClone()` ya JSON methods use karo deep copy ke liye.

✅ **Summary / Tips:**

- Object JS ka sabse **fundamental building block** hai.
- **Dot notation** fast hai but **bracket notation** flexible hai.
- Always dhyan rakho **copy shallow hai ya deep**.
- `Object.freeze()` aur `Object.seal()` immutability ke liye important hain.

✅ Summary / Tips (Quick Revision)

- **Object** = Key-Value pairs (real-world entities ko represent karne ke liye).
- **Keys** hamesha string form me store hoti hain (special chars/space ke liye `" "` use karo).
- **Accessing** → Dot Notation (fast) + Bracket Notation (flexible).
- **Creating Objects → Object Literal `{}`, new Object() ,Constructor Function , Class (ES6)**

- **Modifying Objects** → Add, Update, Delete.
- **Methods** →
    - `Object.keys(obj)` → properties array
    - `Object.values(obj)` → values array
    - `Object.entries(obj)` → key-value pairs array
    - `Object.assign()` / Spread → copy/combine objects
- **Freeze vs Seal** → Freeze ❄️ = no change at all | Seal 🔒 = update allowed but no add/delete.
- **Copying** →

- Shallow Copy → same reference, risky in nested objects.
- Deep Copy → `structuredClone()` / `JSON` methods, safe for independence.

⚡ **Important Tip:** Objects har jagah use hote hain (functions, arrays, JSON sab object hi hote hain JS me). Inko solid samajhna bahut zaroori hai.

📘 Lecture 11 – Objects in JavaScript (Part-2)

---

1. Destructuring of Objects

👉 **Definition:** Destructuring ek syntax hai jisme hum **object ke properties ko directly variables me unpack** kar lete hain.
Ye code ko short aur readable banata hai.

Basic Object Destructuring

```
let obj = {
  name: "Sourav",
  money: 420,
  balance: 30,
  age: 20,
  aadhaar: "74729826543"
};
// Normal Access
let n = obj.name;
console.log(n);   // Sourav

// With Destucturing
const { name, balance, age } = obj;
console.log(name, balance, age);
// Sourav 30 20
```

Assigning New Variable Names

👉 Hum variables ko naya naam bhi de sakte hain.

```
const { name: fullname, age: years } = obj;
console.log(fullname, years);
// Sourav 20
```

⚡ Ab `name` aur `age` directly use nahi kar sakte, kyunki unka naam badal gaya hai.

Using Rest Operator

👉 Jo properties hum destructure nahi karte, wo `...` rest operator se ek naya object me chale jate hain.

```
const { name, age, ...obj1 } = obj;
console.log(name, age);   // Sourav 20
console.log(obj1);
// { money: 420, balance: 30, aadhaar: "74729826543" }
```

⚡ Step by Step Explanation:

1. **`{ name, age, ...obj1 } = obj;`**
    - Yahan JS bolega → `obj` ke andar se `name` aur `age` properties nikal lo aur **alag variables** bana lo.
    - Matlab:
        - `name` variable → `"Sourav"`
        - `age` variable → `20`
2. 🔹 Ye values **copy** karke nayi variables me store hoti hain, `obj` se delete nahi hoti.

---

2. **`...obj1`**
    - Rest operator (`...`) bolta hai:
      "jo properties tumne destructure (nikali) nahi ki hain, un sab ko ek **naya object** bana kar `obj1` me daal do."
    - Matlab:
        - Baaki bachi properties = `{ money: 420, balance: 30, aadhaar: "74729826543" }`

---

2. Destructuring Of An Arrays

👉 Same concept arrays me bhi lagta hai.

```
const arr = [3, 2, 1, 5, 10];

const [first, second] = arr;
console.log(first, second);
// 3 2

const [a, b, , c] = arr;
```

```
console.log(a, b, c);
// 3 2 5

const [x, y, ...rest] = arr;
console.log(x, y);   // 3 2
console.log(rest);   // [1, 5, 10]
```

---

3. Destructuring Nested Objects

👉 Nested object ke andar ke properties bhi destructure kar sakte hain.

```
let obj = {
  name: "Harshal",
  age: 20,
  aadhaar: "45863072",
  address: {
    pincode: 802113,
    city: "Varanasi",
    state: "UP"
  }
};

const { address: { city, pincode } } = obj;
console.log(city, pincode);
// Varanasi 802113
```

👉 Array inside object:

```
let obj = {
  arr: [90, 40, 60, 80]
};

const { arr: [first] } = obj;
console.log(first);
// 90
```

- Yahan `arr:` ka matlab hai → obj ke andar ki `arr` property ko access karo.
- `[first]` ka matlab hai → us array ka **pehla element nikal kar** variable `first` me store karo.

4. Prototype Chaining

Definition:

👉 Prototype chaining ek **mechanism hai jisme objects dusre objects ke properties aur methods inherit karte hain.**
👉 Is wajah se humein inbuilt functions milte hain jaise `toString()`, `push()`, `pop()` etc.

Example

```
let obj = {
  name: "Harshal",
  amount: 420,
  greet: function() {
    return 10;
  }
};

console.log(obj.toString()); // Output: [object Object]
// function inherited from Object.prototype
```

👉 Humne `toString()` banaya hi nahi, fir bhi access kar pa rahe hain → kyunki **Object.prototype** se inherit hota hai.

Prototype Chain Flow

1. Har object ke paas ek hidden property hota hai: `__proto__`.
2. Agar property khud object me nahi milti to JS uske prototype me search karta hai.
3. Ye process chain form me chalta hai → isse **prototype chaining** kehte hain.

Object Linking Example

```
let user1 = { name: "Harsh", age: 20 };
let user2 = { amount: 150, money: 20 };

user2.__proto__ = user1;

console.log(user2.name);
// Harsh (inherited from user1)
```

👉 Yaha **prototype chaining** set ki gayi hai. Matlab:

- Agar `user2` ke andar koi property **directly nahi milegi**, tab JavaScript usko `user1` ke andar dhoondhega.
- `__proto__` ek hidden link hota hai jo ek object ko dusre object ke saath connect karta hai (inheritance ke liye).

```
console.log(user2.name);
```

👉 Ab `user2` me `name` property direct nahi hai ❌.
To JS automatically check karega `user2.__proto__` me (yaani `user1`).
Waha `name: "Harsh"` milta hai ✅.

Prototype Chain in Arrays

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

console.log(arr.__proto__ === Array.prototype); // true
console.log(arr.__proto__.__proto__ === Object.prototype); // true
console.log(Object.prototype.__proto__ === null); // true
```

Diagram (Prototype Chain)

```
null
 ↑
Object.prototype → { toString(), valueOf() }
 ↑
Array.prototype  → { push(), pop(), includes(), indexOf() }
 ↑
arr = [10, 20, 30]
```

✅ Key Points (Tips for Interviews)

- Har **object** ke sath `Object.prototype` attach hota hai.
- Har **array** ke sath `Array.prototype` attach hota hai (jo khud Object.prototype se inherit karta hai).

- Isi wajah se `typeof array === "object"`.
- Prototype chaining ka use karke hum methods inherit kar sakte hain bina har object me likhe.

📌 Summary

- **Destructuring** → Objects & Arrays ke values ko direct variables me nikalne ka short syntax.
- **Rest Operator** → Baaki properties ko ek naya object me store kar leta hai.
- **Nested Destructuring** → Inner objects & arrays ke values bhi extract kar sakte ho.
- **Prototype Chaining** → Ek object doosre object ke properties/methods inherit karta hai via `__proto__`.
- **Array methods (push, pop, etc.)** Array.prototype se aate hain, aur wo Object.prototype se inherit karta hai. **Prototype chain end hoti hai `null` par.**

✨📘 LECTURE 12 – FUNCTIONS IN JAVASCRIPT

🔹 What is a Function?

👉 **Function** ek reusable block of code hota hai jo ek specific task perform karta hai.
👉 Functions code ko **modular, reusable, readable** banate hain.

🔹 Function Declaration

👉 `function` keyword ke saath function define kiya jata hai.

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

greet(); // function calling
```

👉 Example with parameters:

```
function sum(a, b) {   // parameters
  return a + b;
}

let result = sum(3, 5); // arguments
console.log(result); // 8
```

🔹 Function Expression

👉 Function ko variable me assign karna → Function Expression.

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

console.log(add(2, 3)); // 5
```

⚠️ **Note:**
`return` ke baad likha hua code **kabhi execute nahi hota**.

```
const fun = function() {
  console.log("Hello Codes Army");
  return "money";
  console.log("Ye kabhi execute nahi hoga");
};

console.log(fun()); // Output:
// Hello Codes Army
// money
```

🔹 Arrow Function (ES6+)

👉 Short & clean syntax. Specially useful for callbacks.

```
const sum = (a, b) => {
  return a + b;
};
console.log(sum(3, 4)); // 7
```

✅ One-liner:

```
const sum = (a, b) => a + b;
console.log(sum(3, 4)); // 7
```

✅ Single parameter:

```
const cube = a => a * a * a;
console.log(cube(3)); // 27
```

👉 🔹 Rest Parameters (`...rest`)

Definition:
👉 Rest parameter multiple arguments ko **ek single array** me collect karta hai.

Use Case:
Jab hume **pehle se nahi pata hota** ki function me kitne arguments aayenge, tab `rest operator` ka use karte hain.

```
const sum = function(...numbers) {
  let sum = 0;
```

```
  for (let i = 0; i < numbers.length; i++) {
    sum += numbers[i];
  }
  return sum;
};

console.log(sum(2, 3)); // 5
console.log(sum(2, 3, 4)); // 9
console.log(sum(10, 18, 1, 19)); // 48
```

🔹 Functions with Objects

👉 Passing object directly:

```
let obj = {
  name: "Sourav",
  age: 20,
  amount: 420
};

function fun(obj) {
  console.log(obj.name, obj.age);
}
fun(obj); // Sourav 20
```

👉 Destructuring in function:

```
function fun2({ name, amount }) {
  console.log(name, amount);
}
fun2(obj); // Sourav 420
```

🔹 Prototype Chain in Functions & Objects

Example:

```
let obj = { name: 'Harshal', age: 20 };

obj.__proto__ === Object.prototype; // true
Object.prototype.__proto__ === null; // true
```

👉 **Functions bhi ek object hote hain.**

```
const first = (a, b) => { return a + b; };

console.log(first.__proto__ === Function.prototype); // true
console.log(Function.prototype.__proto__ === Object.prototype); //
true
console.log(Object.prototype.__proto__ === null); // true
```

🔹 Built-in Prototype Methods

- **Object.prototype**
  - `toString()`, `valueOf()`
- **Array.prototype**
  - `push()`, `pop()`, `includes()`, `indexOf()`
- **Function.prototype**
  - sabhi functions inherit karte hain

🔁 SHORT SUMMARY – EK NAZAR ME

✅ Function Declaration → `function greet() {}`
✅ Function Expression → `const f = function() {}`
✅ Arrow Function → Short, clean syntax `(a,b)=>a+b`
✅ Rest Parameter → `(...args)` multiple arguments collect karta hai
✅ Functions ke andar Objects ko pass kar sakte hain + destructure kar sakte hain
✅ Har function internally ek **object hai** aur `Function.prototype` se inherit karta hai
✅ **Prototype Chain:** Function → Function.prototype → Object.prototype → null

📘 Lecture 13: Conditional Statements, Loops & Scope in JavaScript

---

🔹 Conditional Statements

👉 Code में decision लेने के लिए use होते हैं।

✅ if Statement

```
const age = 20;

if (age >= 18) {

  console.log("You are eligible to vote");

}

// Output: You are eligible to vote
```

✅ if-else Statement

```
const age = 16;

if (age >= 18) {

  console.log("You are eligible to vote");

} else {

  console.log("You are not eligible to vote");

}

// Output: You are not eligible to vote
```

✅ if-else if Ladder

```
let age = 19;

if (age < 18) {
```

```
  console.log("KID");
} else if (age > 45) {
  console.log("OLD");
} else {
  console.log("ADULT");
}
// Output: ADULT
```

✅ Switch Statement

```
switch (new Date().getDay()) {
  case 0: console.log("Sunday"); break;
  case 1: console.log("Monday"); break;
  case 2: console.log("Tuesday"); break;
  case 3: console.log("Wednesday"); break;
  case 4: console.log("Thursday"); break;
  case 5: console.log("Friday"); break;
  case 6: console.log("Saturday"); break;
  default: console.log("Invalid day");
}
```

💡 **Tip:** Switch ज़्यादा readable होता है जब multiple fixed values check करनी हों।

🔹 Loops in JavaScript

👉 बार-बार same code चलाने के लिए।

✅ For Loop

```
for (let i = 1; i <= 5; i++) {
  console.log("Hello World");
}
// Output: Hello World (5 times)
```

👉 Example: Array values print करना

```
const arr = [10, 20, 30, 40, 50];
for (let i = 0; i < arr.length; i++) {
  console.log(arr[i]);
}
// Output: 10 20 30 40 50
```

✅ While Loop

```
let i = 0;
while (i < 5) {
  console.log("Hello World");
  i++;
}
// Output: Hello World (5 times)
```

✅ Do...While Loop

```
let i = 0;
do {
  console.log("Hello World");
  i++;
```

```
} while (i < 5);

// Output: Hello World (5 times)
```

👉 **Note:** कम से कम 1 बार ज़रूर चलेगा।

✅ Nested Loops

```
const matrix = [
  [1, 2, 3],
  [4, 5, 6],
  [7, 8, 9]
];


for (let i = 0; i < matrix.length; i++) {
  for (let j = 0; j < matrix[i].length; j++) {
    console.log(matrix[i][j]);
  }
}
// Output: 1 2 3 4 5 6 7 8 9
```

🔹 Scope in JavaScript

👉 Scope decide करता है कि variable या function कहाँ तक accessible है।

✅ Global Scope

- Pure Code me kahi bhi access

```
let a = 10;

var b = 20;
```

```
const c = 30;


function greet() {
  console.log(a, b, c); // 10 20 30
}


console.log(a, b, c); // 10 20 30
```

✅ Local Scope (Function Scope)

- Function ke inside access jo bhi variable define hoge

```
function greet() {
  let a = 10;
  var b = 20;
  const c = 30;
  console.log(a, b, c); // 10 20 30
}


greet();
console.log(a); // ❌ Error
```

✅ Block Scope (let, const)

- {} ke inside access

```
if (true) {
  let a = 10;
  const c = 30;
```

```
  var b = 20;


  console.log(a); // 10
  console.log(c); // 30
}
console.log(b); // 20 (var leak हो गया बाहर)
console.log(a); // ❌ Error
```

💡 **Tip:** हमेशा `let` और `const` का use करो, `var` avoid करो।

❌ Why Not Use var?

1. Block scope follow नहीं करता → बाहर से access हो सकता है।
2. Redeclaration allowed → bugs create करता है।
3. Hoisting problem → variable declare करने से पहले भी undefined value मिलती है।

```
console.log(x); // undefined (due to hoisting)
var x = 10;
```

🔹 Functions & Hoisting

✅ **Function Declaration → Hoisting में पहले call कर सकते हैं।**

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

❌ **Function Expression** → पहले **call** नहीं कर सकते।

```
meet(); // ❌ Error

const meet = function () {

  console.log("Hello Meet");

}
```

💡 **Note:** Function expressions variables में stored होते हैं, और variables hoist होकर `undefined` बनते हैं declaration से पहले।

📌 Final Short Summary – Ek Nazar Me

✅ **Conditional Statements** → if, if-else, if-else-if, switch
✅ **Loops** → for, while, do-while, nested
✅ **Scope** → global, local (function), block
✅ **Avoid var** → block scope + hoisting issues
✅ **Functions & Hoisting** → declaration chalega, expressionनहीं

📘 Lecture 14: Advanced Loops in JavaScript

🔹 `for-in` Loop (Objects ke liye)

👉 `for-in` loop ka use object ke keys iterate karne ke liye hota hai।

```
let obj = {
  name: "Rohan",
  age: 23,
  gender: "male",
  city: "Banaras"
};

for (let key in obj) {
  console.log(key);
}
// Output: name, age, gender, city
```

👉 Values access karne ke liye:

```
for (let key in obj) {
  console.log(key, obj[key]);
}
// name Roham
// age 23
// gender male
// city Kotdwara
```

- `key` me property ka naam hota hai (jaise `"name"`).
- `obj[key]` likhne ka matlab → object se us key ki **value nikalna**.

Step by Step:

1. Pehli iteration: `key = "name"`
    - `obj["name"]` = `"Rohan"`
2. Dusri iteration: `key = "age"`
    - `obj["age"]` = `23`
3. Teesri iteration: `key = "gender"`
    - `obj["gender"]` = `"male"`
4. Chauthi iteration: `key = "city"`
    - `obj["city"]` = `"Banaras"`

🔹 Difference: `Object.keys()` vs `for-in`

```
let obj1 = { name: "Harsh", age: 20 };
let obj2 = Object.create(obj1);
obj2.money = 420;
obj2.id = "Ron";

console.log(Object.keys(obj2));
// ['money', 'id']

for (let key in obj2) {
  console.log(key);
}
// money, id, name, age
```

🔹 **Short Explanation:**

- **`Object.keys(obj2)`** → sirf **obj2 ke own keys** deta hai → `['money', 'id']`.
- **`for-in`** → obj2 ke **own + inherited keys (obj1 se)** sabko iterate karega → `money, id, name, age`.

👉 Simple Rule:

- `Object.keys()` = apne ghar ke saman.
- `for-in` = apne + papa ke ghar ka inherited saman bhi.

🔹 Why `Object.prototype` ke properties for-in se access nahi hote?

```
console.log(Object.getOwnPropertyDescriptor(Object.prototype, 'toString'));

/*

{

  value: [Function: toString],

  writable: true,

  enumerable: false,  ❌ not enumerable

  configurable: true

}

*/
```

👉 Kyunki **toString enumerable: false** hai, isliye `for-in` usse ignore karta hai।

🔹 Property Descriptors in JS

Every property ke 3 hidden attributes hote hain:

- **writable** → value change kar sakte hain ya nahi.
- **enumerable** → iteration (for-in, Object.keys) me show hoga ya nahi.
- **configurable** → delete ya modify kar sakte ho ya nahi.

✅ Writable Example

```
let obj = {};

Object.defineProperty(obj, 'name', {

  value: "rohit",

  writable: false,   // ❌ value cannot be changed

  enumerable: true,

  configurable: true

});


console.log(obj.name); // rohit

obj.name = "mohit";    // fail

console.log(obj.name); // rohit
```

✅ Configurable Example

```
let obj = {};

Object.defineProperty(obj, 'name', {

  value: "Harshal",

  writable: true,

  enumerable: true,
```

```
  configurable: false
});

obj.name = "mohit";
console.log(obj.name); // mohit ✅

Object.defineProperty(obj, 'name', { writable: false });
// writable can be changed true → false
obj.name = "Saurav";
console.log(obj.name); // mohit ❌
```

👉 **configurable: false** hone par property ko delete/modify nahi kar सकते (except writable true → false)।

✅ Enumerable Example

```
const customer = {
  name: "Saurav",
  age: 21,
  account_number: 122436789,
  account_balance: 4820
};

for (let key in customer) {
  console.log(key);
}
// name, age, account_number, account_balance
```

```
Object.defineProperty(customer, 'name', { enumerable: false });


for (let key in customer) {
  console.log(key);
}
// age, account_number, account_balance
```

👉 **enumerable: false** → property loop me nahi aati, but directly access kar sakte hain।

👉 ek **easy & best example** deta hoon jisme `writable`, `enumerable`, `configurable` teenon ka use ekdum clear ho jaye 👇

Example:

```
let person = {};


// define property
Object.defineProperty(person, "id", {
  value: 101,
  writable: false,    // ❌ value change nahi kar sakte
  enumerable: false,  // ❌ loop me nahi dikhai degi
  configurable: false // ❌ delete/modify nahi kar sakte
});


// 1️⃣ Try to change value
person.id = 202;
console.log(person.id);
// Output: 101 (value change nahi hui)
```

```
// 2️⃣ Try to loop
for (let key in person) {
  console.log(key);
}
// Output: (blank, kuch nahi aaya)

// 3️⃣ Try to delete
delete person.id;
console.log(person.id);
// Output: 101 (delete bhi nahi hua)
```

🔹 Short Explanation:

1. **writable: false** → value change nahi hoti.
2. **enumerable: false** → property loop me hidden ho jati hai.
3. **configurable: false** → delete ya aur modify karna allowed nahi.

👉 Real-life socho:

- `id` ek **Aadhar number** jaisa hai →
    - Ek baar ban gaya to change nahi kar sakte (writable ❌).
    - Public list me nahi dikhana (enumerable ❌).
    - Aur delete bhi nahi kar sakte (configurable ❌).

🔹 Why not use `for-in` with Arrays?

```
const arr = [10, 20, 30, 40];

arr.name = "saurav";

arr.age = "30";


for (let key in arr) {

  console.log(key, arr[key]);

}

// 0 10

// 1 20

// 2 30

// 3 40

// name saurav ❌

// age 30 ❌
```

👉 Array bhi object hai, isliye `for-in` normal indexes ke sath **extra properties** bhi print kar dega।
👉 Isliye array iterate karne ke liye use karo:

- `for` loop
- `for-of` loop
- `forEach()`

🔹 Inherited Properties ke sath Example

```
let customer = {

  name: "Saurav",

  age: 21,
```

```
  account_number: 122436789,
  account_balance: 4820
};

let customer2 = Object.create(customer);
customer2.city = "Haridwar";
customer2.place = "Delhi";

for (let key in customer2) {
  console.log(key);
}
// place, city, name, age, account_number, account_balance
```

👉 `for-in` inherited enumerable properties ko bhi access karta hai।

📌 Final Short Summary – Ek Nazar Me

✅ **for-in loop** → object keys iterate karta hai (own + inherited enumerable).
✅ **Object.keys()** → sirf own keys deta hai।
✅ **Property Descriptors** → `writable`, `enumerable`, `configurable` object property behavior decide karte hain।
✅ **for-in loop arrays ke liye avoid karo** → kyunki extra properties bhi iterate ho jati hain।
✅ **Prototype properties (like toString)** access nahi hoti kyunki `enumerable: false` hoti hai।

Lecture 15 – Callback Functions, for…of, forEach, filter & map (Desi Version)

1️⃣ for…of Loop

- **Kya hai:** `for…of` iterable cheezon (arrays, strings, sets, maps) ke **values** pe loop lagane ka simple tareeka hai.
- Syntax:

```
for (variable of iterableObject) {
  // yahan code chalega
}
```

- `variable` → har baar loop me current value store karta hai.

Example – Array

```
const arr = [10, 20, 11, 18, 13];
for (let value of arr) {
  console.log(value); // 10 20 11 18 13
}
```

Example – String

```
let str = "Saurav";
for (let char of str) {
  console.log(char); // S a u r a v
}
```

Important Tip:

- Plain objects pe **for…of** nahi chalta, kyunki objects iterable nahi hote, kyuki phle wale key location pe pahuch gya to aage ka memory location nahi pata hai .
- Object ko iterate karna ho → `Object.keys()`, `Object.values()`, `Object.entries()` ka use karo.

👉 **Agar ekdam chull machi hai iterate krne ki tab 😀**

```
let user = { name: "Harshal", age: 21 };
```

- **Keys Loop**

```
for (let key of Object.keys(user)) {
  console.log(key);
}
// Output: name, age
```

- **Values Loop**

```
for (let value of Object.values(user)) {
  console.log(value);
}
// Output: Harshal, 21
```

- **Entries Loop**

```
for (let [key, value] of Object.entries(user)) {
  console.log(key, value);
}
```

- Matlab:
  - Pehla inner array → `["name", "Harshal"]`
  - Dusra inner array → `["age", 21]`

```
// Output: name Harshal
//         age 21
```

👉 Trick:

`Object.keys()` → keys ke liye

`Object.values()` → values ke liye

`Object.entries()` → dono ek sath

🔹 Object Iteration Shortcut

- **Object.keys(obj)** → object ke **keys ko array** bana deta hai.
- **Object.values(obj)** → object ke **values ko array** bana deta hai.
- **Object.entries(obj)** → object ke **[key, value] pairs ko array** bana deta hai.

👉 `for..of` loop array pe chalta hai, isliye ye tino ke saath use kar sakte ho ✅

📘 Callback Function Notes (Final 🔥)

🔹 Definition

Callback = **Ek function ko dusre function ke argument ke roop me dena**, aur baad me usse chalana.
👉 Matlab: “Function ke andar dusra function.”

🔹 Flow

➡ Main function → apna kaam karega.
➡ Fir baad me callback function ko call karega.

🔹 Use Cases

- **Asynchronous code** → setTimeout, API calls
- **Events** → button click, user input
- **Array methods** → map, filter, forEach

🔹 Example 1 – Simple

```
function greet(callback) {
  console.log("Main greet function hu");
  callback(); // yaha dusra function chalega
}

function hello() {
  console.log("Main callback function hu");
}

greet(hello);

/*
Output:
Main greet function hu
Main callback function hu
*/
```

🔹 Example 2 – Real Life (Teacher–Student)

```
function teacher(checkHomework) {
  console.log("Teacher: Pahle main padhata hu...");
  checkHomework(); // baad me student ka homework check hota hai
}

function student() {
  console.log("Student: Homework dikhata hu!");
}

teacher(student);  // yaha pe student ka reference pass kar rahe hai

/*
Output:
```

```
Teacher: Pahle main padhata hu...
Student: Homework dikhata hu!
*/
```

👉 Matlab: **Teacher apna kaam pehle karta hai, fir student ka function call hota hai.**

🔹 Shortcut Versions

```
// Arrow function callback
greet(() => console.log("Main callback hu"));

/*
Output:
Main greet function hu
Main callback hu
*/

// Function expression callback
const fun = function() {
  console.log("Main callback hu");
};
greet(fun);
/*
Output:
Main greet function hu
Main callback function hu
*/
```

✅ Super Tip

- `callback` → function ko **pass karo** (reference ke roop me).
- `callback()` → function turant **execute ho jayega**.

👉 Hamesha **reference pass karo**, taaki wo baad me run ho.

🔑 Ek Line Me Yaad Rakhne Ka Tarika

Callback = “Dusre ke kaam khatam hone ke baad chalne wala function.”

3️⃣ `forEach()` – Array Loop

🔹 `forEach()` ek **array method** hai jo **array ke har element par ek function run karta hai**.
👉 Ye mainly tab use hota hai jab hame array ke elements ko ek-ek karke process karna ho.

✅ Syntax

```
array.forEach((value, index, array) => {
  // code yaha chalega
});
```

- `value` → current element
- `index` → us element ka index
- `array` → pura array

🔸 Different Ways to Use `forEach()`

1️⃣ Normal Function

```
let arr = [10, 20, 30, 40];


arr.forEach(function(num) {
// forEach array se ek value uthata hai
  // aur function(num) me bhejta hai
  console.log(num);
});
/*
Output:
10
20
30
40
*/
```

🛠️ Behind the Scenes (Kaise kaam karta hai):

```
for (let i = 0; i < arr.length; i++) {
  let num = arr[i];
  console.log(num);
}
```

✅ forEach() bhi kuch aisa hi kaam karta hai – lekin short cut style me .

2️⃣Arrow Function

```
arr.forEach((num) => {
  console.log(num);
});
/*
Output:
10
20
30
40
*/
```

3️⃣Single-line Arrow Function

```
arr.forEach(num => console.log(num));
/* Output:
```

```
10

20

30

40 */
```

🔸 Multiple Arguments

✅ Index ka use

```
arr.forEach((num, indx) => console.log(num, indx));

/*

Output:

10 0

20 1

30 2

40 3

*/
```

✅ Original Array ka use

```
arr.forEach((num, indx, a) => {

  console.log(num, indx, a);

});

/*

Output:

10 0 [10,20,30,40]

20 1 [10,20,30,40]

30 2 [10,20,30,40]

40 3 [10,20,30,40]
```

```
*/
```

🔸 Modify Array Example

```
let arr = [10, 20, 30, 40];


arr.forEach((num, idx, a) => {
  a[idx] = num + 2;
});


console.log(arr);
// Output: [12, 22, 32, 42]
```

🔁 Ek Real Life Example:

Socho tumhare paas ek **thali (plate)** hai jisme 4 **rotiyan** hain:

```
let thali = [1, 1, 1, 1]; // 1 = simple roti
```

Tum chahte ho ki har roti pe **makhan laga do**:

```
thali.forEach((roti, i, t) => {
  t[i] = roti + 1; // 1 = makhan added
});
```

Ab tumhari original **thali** ban gayi:

```
[2, 2, 2, 2] // sab rotiyon pe makhan lag gaya 😋
```

⚠️ Important Notes

- `forEach()` **kuch return nahi karta** → hamesha `undefined`.
- `break` / `continue` use nahi kar sakte.
- Sirf **array ke har element par function chalane** ke liye best hai.
- Agar **naya array chahiye** ho → `map()` use karo.

👉 **Shortcut yaad rakhne ka tarika**
`forEach()` = "Array ke sab elements pe function chalao, par kuch return mat lo."

⚠️ 🔁 Kya hum `forEach()` se banaye hue naya result **naye array me store** kar sakte hain?

🔴 Seedha jawab:

Nahi bhai, `forEach()` ka kaam sirf loop chalana hota hai.
Woh **kuch return nahi karta**, isliye **naye array me store nahi kar sakte.**

✅ Agar tumhe naya array banana hai to `map()` use karo!

4️⃣ filter()

`filter()` Kya Hai?

- `filter()` ek **array method** hai.
- Ye **original array ko touch nahi karta** (original array safe rehta hai).

- Ye **condition ke basis par** ek **naya array banata hai** jisme sirf wo elements hote hain jo condition ko satisfy karte hain.

2️⃣ Syntax:

```
let newArray = originalArray.filter(callbackFunction);
```

- `callbackFunction` har element pe chalega.
- Jo elements **`true` return karenge**, wahi `newArray` me rahenge.

3️⃣ Example 1: Numbers Array me Even Numbers filter karna

```
let arr = [10, 22, 33, 41, 50];

const even = arr.filter(num => num % 2 === 0);

console.log(even);  // Output: [10, 22, 50]
```

Explanation:

- Har number pe `num % 2 === 0` (matlab even hai ya nahi) check karta hai.
- Jo even hain, wahi new array `even` me aaye.

4️⃣ Example 2: Array of Objects me Filter karna

```
const students = [
  {name: "rohan", marks: 40},
  {name: "mohan", marks: 80},
  {name: "saurav", marks: 90}
];

const passed = students.filter(s => s.marks > 50);

console.log(passed);
// Output:
```

```
// [
//   {name: "mohan", marks: 80},
//   {name: "saurav", marks: 90}
// ]
```

Explanation:

- Students jinke `marks` 50 se jyada hain, unko filter karke naya array `passed` me store kiya.

5️⃣ Pro Tip: Destructuring Use Karke Short Code

```
const passed2 = students.filter(({marks}) => marks > 50);
```

- Yahan `{marks}` directly object me se `marks` property nikal raha hai.
- Code zyada concise aur clean ho jaata hai.

6️⃣ Summary:

Feature	Description
Type	Array method
Return	Naya filtered array
Original array	Safe, no change
Use case	Filter items based on condition
Callback function argument	Current element of array
Callback returns	`true` to keep element, `false` to ignore

5️⃣ map()

`map()` Kya Hai?

- `map()` ek **array method** hai.

- Ye array ke **har element par operation** (transformation) apply karta hai.
- Har transformed value se **naya array banata hai**.
- **Original array safe rehta hai**, koi badlav nahi hota.

2️⃣ Syntax:

```
let newArray = originalArray.map(callbackFunction);
```

3️⃣ Example: Square of Numbers

```
const arr = [1, 2, 3, 4];

const squares = arr.map(num => num * num);

console.log(squares);  // [1, 4, 9, 16]
```

Explanation:
Har number ka square nikal ke ek naya array `squares` ban gaya.

4️⃣ Combined Use: `filter()` + `map()`

```
let arr2 = [1, 2, 3, 4, 5, 6];

const result = arr2
  .filter(n => n % 2 === 0)  // Even numbers
  .map(n => n * n);          // Unka square

console.log(result);  // [4, 16, 36]
```

Explanation:

- Pehle filter se even numbers liye: `[2, 4, 6]`
- Fir unka square kiya: `[4, 16, 36]`

5️⃣ Real-Life Example (Objects ke saath)

```
const users = [
  { name: "Amit", age: 20 },
  { name: "Sumit", age: 25 }
];

const names = users.map(user => user.name);

console.log(names);  // ["Amit", "Sumit"]
```

6️⃣ Summary Table:

Feature	Description
Method Type	Array method
Return	New transformed array
Original Array	Unchanged
Purpose	Transform each element
Callback Returns	Modified value for each element
Use With `filter()`	Yes, first filter, then map

✅ Tip:

- `filter()` → **select** elements
- `map()` → **transform** elements

🔚 Final Line:

`map()` har item ke saath kaam karta hai, usme kuch badlav karta hai, aur naya array banata hai — original array bilkul safe.

💡 Quick Tips

- `for…of` → iterable pe use karo (arrays, strings, sets, maps). Objects pe nahi.
- `forEach` → simple array iteration.
- `filter` → conditional naya array banata hai.
- `map` → array elements modify ya transform karta hai.
- Callback → function ko function me pass karna aur later run karna.
- Object iterate karna ho → `Object.keys/values/entries` ka use karo.

🔁 Summary – Ek Nazar Me

Feature	Kya karta hai	Example
for…of	Iterable ke values iterate	Array/String
Callback	Function ko argument me pass karna	greet(hello)
forEach	Array me har element pe function	arr.forEach(num => …)
filter	Condition satisfy element ka new array	arr.filter(n => n>10)
map	Array elements transform	arr.map(n => n*n)
Object.keys/values/entries	Object iterate karna	for-of ke saath

✨ Quick & Crisp Summary — `forEach`, `filter`, `map`, `reduce`

🔹 `forEach()`

“Kaam kar, return mat kar!”
📌 Use: Har element pe kaam (like print, alert), kuch return nahi chahiye
🧠 Action-only loop

🔹 `filter()`

“Jo pasand ho, wahi rakh!”
📌 Use: Condition ke basis par chhantna (like even numbers)
🧠 Returns selected items in a new array

🔹 `map()`

“Sabko badal, naya roop de!”
📌 Use: Har item ko change/transform karke naya array banana
🧠 Same length, new values

🔹 `reduce()`

“Sabko jodo, ek result nikalo!”
📌 Use: Array ko ek single value me convert karna (sum, total, etc.)
🧠 Value compressor

🎯 Super-Short Memory Line:

- 🔸 `forEach` – Kaam kar, return nahi kar
- 🔸 `filter` – Jo chahiye, wahi rakh
- 🔸 `map` – Sabko badal, naya array bana
- 🔸 `reduce` – Sabko mila, ek hi value bana

🔁 Quick Example Table:

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

Method	Use Case	Output
`forEach`	Print all	1 2 3 4 (console)
`filter`	Even numbers	[2, 4]
`map`	Square each number	[1, 4, 9, 16]
`reduce`	Total sum	10

🧠 Lecture 16 - Reduce , Map & Set

🥇 Kya hai `reduce()`?

> 💬 Socho ek **dabba (box)** hai — tum har item us dabbe me **dalte ja rahe ho**, aur har step pe us dabbe ka content **update hota ja raha hai**.
> **Last me ek final value banti hai.**
> Bas wahi hai `reduce()`.

✅ Kaam:

👉 Ye har array element pe function chalata hai
👉 Aur **ek hi final result** banata hai — value ya object
👉 Har step pe ek **accumulator** me result banta rehta hai

⚙️ Reduce() Syntax & Parameters Explained Clearly

```
array.reduce((accumulator, currentValue, index, array) => {
  // logic
  return updatedAccumulator;
}, initialValue);
```

🔍 Parameters:

Naam	Matlab/Explanation
accumulator (acc)	Pichle step ka result, har baar update hota hai
currentValue (curr)	Abhi process ho raha current element
index	(Optional) Current element ka index
array	(Optional) Original array jispe reduce chal raha hai
initialValue	Shuruaat me accumulator ki value, zaroori hoti hai

---

🔥 1. Sabse Simple Example – SUM

```
const arr = [10, 20, 30, 40, 50];
```

```
const sum = arr.reduce((acc, curr) => acc + curr, 0);

console.log(sum); // 150
```

🧠 **Logic:**

- `acc = 0` (initial)
- `curr = 10 → acc = 0 + 10 = 10`
- `curr = 20 → acc = 10 + 20 = 30`
- ...
- Finally acc = 150

> 💬 **Yani reduce ne array ko jod jod ke ek hi number bana diya!**

🍎 2. Real-Life Example – Fruits Counter (Frequency Count)

```
let fruits = ["orange", "apple", "banana", "orange", "apple",
"orange", "grapes"];

const result = fruits.reduce((acc, fruit) => {
  if (acc.hasOwnProperty(fruit)) {
    // Agar fruit pehle se object me hai, count badhao
    acc[fruit] = acc[fruit] + 1;
  } else {
    // Agar fruit pehli baar mil raha hai, 1 se shuru karo
    acc[fruit] = 1;
  }

  return acc;
}, {}); // Initial value: empty object {}

console.log(result);
```

🧠 **Output:**

```
{
  orange: 3,
  apple: 2,
  banana: 1,
  grapes: 1
}
```

🧠 Ek Line Me Yaad Rakhne Ka Tareeka:

> 🔹 **"Sabko mila, ek cheez bana!"**
> Chahe woh number ho, string ho, object ho, array ho... reduce sabko jod ke **ek nateeja** banata hai!

2️⃣ Map (Collection)

Map is a built-in JavaScript object that stores **key-value pairs**.

✅ Key Points:

- Keys can be **any data type** – numbers, strings, objects, functions, etc. (Unlike objects, which only allow strings or symbols as keys.)
- Maintains the **insertion order** of elements.
- Provides methods for **fast access**, **insertion**, **deletion**, and **iteration**.

🔧 Creating a Map

✅ Method 1: Using `new Map()` (Most Common)

```
const myMap = new Map();
myMap.set(10, 'key');
myMap.set('naman', 'newname');
```

✅ Method 2: Creating Map with Initial Data

```
const myMap = new Map([
  ['name', 'Naman'],
  [100, 'Number key'],
  [true, 'Boolean key']
```

```
]);
```

📥 Adding Values – `.set(key, value)`

```
myMap.set('language', 'JavaScript');
myMap.set(99, 'marks');
```

📤 Getting Values – `.get(key)`

```
console.log(myMap.get('language'));  // JavaScript
```

✅ Checking Keys – `.has(key)`

```
console.log(myMap.has(99)); // true
```

❌ Deleting Keys – `.delete(key)`

```
myMap.delete(99);  // removes key 99
```

🧹 Clearing the Map – `.clear()`

```
myMap.clear();  // removes all entries
```

📏 Map Size – `.size`

```
console.log(myMap.size);  // Number of key-value pairs
```

🔁 Looping through Map

✅ 1. `forEach()`

```
myMap.forEach((value, key) => {
  console.log(`${key}: ${value}`);
});
```

✅ 2. `for...of` Loop (with Destructuring)

```
for (let [key, value] of myMap) {
  console.log(`${key}: ${value}`);
}
```

📊 Map vs. Object – Which One to Use?

Situation	Use Map	Use Object
Keys can be any data type	✅	❌ (Only string/symbol)
Maintain insertion order	✅	❌
Frequent adding/removing	✅	❌ (Less efficient)
Simple key-value store needed	❌	✅
Need built-in methods like `.set()` / `.get()`	✅	❌

🧠 Tips:

- Don't use `map[key] = value` — that's for **objects**, not Maps.
- Always use `.set()` and `.get()` with Maps.
- Map keys can be objects too:

```
const objKey = { id: 1 };
myMap.set(objKey, 'value for object key');
```

📌 Summary:

Feature	Map
Keys	Any data type
Order	Maintains insertion order
Performance	Fast for frequent changes

Methods	`.set()`, `.get()`, `.has()`, `.delete()`, `.clear()`, `.size`
Use case	Advanced key-value storage

🪄 Super Short Yaad Rakhne Wali Line:

> **"Map tab use karo jab keys sirf string nahi, kuch bhi ho sakti ho!"**

3️⃣ Set

🔹 What is a Set?

Set is a special JavaScript object that stores **only unique values**.

✅ Key Features:

- **Duplicate values allowed nahi hote**
- **Insertion order maintain** karta hai
- Values can be of **any data type**
- Built-in methods ke through fast operations milte hain

🛠️ Creating a Set

✅ 1. From an Array:

```
const set1 = new Set([10, 20, 20, 30, 40, 10]);
console.log(set1);
// Set(4) {10, 20, 30, 40}
```

✅ 2. By adding values manually:

```
const set2 = new Set();
set2.add(4);
set2.add("Harshal");
set2.add(6);
```

```
console.log(set2);
// Set(3) {4, "Harshal", 6}
```

🔧 Common Methods in Set (With Examples)

Method	Kya Karta Hai	Example (Output)
`add(value)`	Naya value add karta hai	`set.add(10)` → `Set {10}`
`delete(value)`	Specific value hata deta hai	`set.delete(10)` → `true/false`
`has(value)`	Check karta hai value present hai ya nahi	`set.has("apple")` → `true/false`
`clear()`	Set ke sare elements hata deta hai	`set.clear()` → `Set {}`
`size`	Total unique elements ka count	`set.size` → `number`

🔍 Examples for Methods:

```
const s = new Set([1, 2, 3]);

s.add(4);                  // Add 4
s.delete(2);               // Remove 2
console.log(s.has(1));     // true
console.log(s.size);       // 3
s.clear();                 // Empty the set
console.log(s);            // Set {}
```

🔁 Converting Between Set & Array

🔄 Array → Set (To Remove Duplicates):

```
const arr = [10, 20, 10, 30];
const set = new Set(arr);
console.log(set);  // Set {10, 20, 30}
```

🔄 Set → Array:

```
const arr2 = [...set];
console.log(arr2);  // [10, 20, 30]
```

🔁 Looping Over a Set

✅ Using `for...of`

```
for (let item of set) {
  console.log(item);
}
```

✅ Using `forEach()`

```
set.forEach((value) => {
  console.log(value);
});
```

🔥 Real-Life Use Cases

Use Case	Why Set is Useful
Remove duplicates from array	Automatically removes repeats
Store unique tags/user IDs	Ensures no duplicate entries
Fast lookup of existence	`.has()` is faster than `.includes()`
Compare lists (union/intersection)	Great for set theory operations

---

➕ Set Operations (Manually)

✅ Union (Combine Unique Elements):

```
const a = new Set([1, 2, 3]);
const b = new Set([3, 4, 5]);
const union = new Set([...a, ...b]);
console.log(union);  // Set {1, 2, 3, 4, 5}
```

✅ Intersection (Common Elements):

```
const intersection = new Set([...a].filter(x => b.has(x)));
console.log(intersection);  // Set {3}
```

✅ Difference (Only in A, not in B):

```
const difference = new Set([...a].filter(x => !b.has(x)));
console.log(difference);  // Set {1, 2}
```

💡 Tips to Remember

- **Best tool to remove duplicates** from an array
  ➤ `const unique = [...new Set(arr)]`
- **Fast membership check** → `.has(value)`
- **Not index-based** like arrays (no `.map()`, `.filter()` directly)
- Use `Set` when you care about:
  - **Uniqueness**
  - **Speed**
  - **Clean data**

📌 Summary Table

Feature	Set
Duplicate allowed	❌ No
Order preserved	✅ Yes
Data types allowed	✅ All (number, string, object, etc.)
Key-Value pairs	❌ Only values (not key-value)
Loopable	✅ Yes (`for...of`, `.forEach()`)
Useful for	Uniqueness, fast lookup, filtering

🧠 One-Line Definition to Remember:

"Set sirf ek baar value rakhta hai, baar-baar nahi!"

🔁 Summary – Ek Nazar Me

Feature	Kya karta hai	Best Use

reduce()	Array → Single value/object	Sum, count, flatten, max/min
Map	Key-Value pairs, keys can be any type	Dynamic keys, ordered data
Set	Unique values only	Remove duplicates, membership check

🔥 **Master Tip:**

- `reduce` = **calculation & aggregation**
- `Map` = **dictionary/lookup table**
- `Set` = **unique collection / fast membership check**

Lecture 18 – How JS Code Works & Hoisting in JS

1️⃣ JavaScript Nature

- **Synchronous** → ek time pe ek hi kaam karta hai.
- **Single-threaded** → ek hi main thread hota hai jo **line by line** code chalata hai.

👉 Matlab: Ek line complete hogi tabhi agli line chalegi.

2️⃣ Execution Context

Har JS code chalne se pehle **Execution Context (EC)** banta hai.

EC ke 2 phases hote hain:

1. **Memory Creation Phase (Hoisting Phase)**
    - Variables aur functions ke liye memory allocate hoti hai.
    - `var` → **undefined** assign hota hai.
    - `let` & `const` → memory allocate hoti hai but **value assign nahi hoti** (yehi TDZ hota hai).
    - Functions → pura function ka code memory me store hota hai.
2. **Code Execution Phase**
    - Ab actual code line by line chalti hai.
    - Variables ko values assign hoti hai.
    - Functions call hote hain.

Example

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

```
console.log(y); // ReferenceError
let y = 20;

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

Execution Context Table

Memory	Code
x: undefined	console.log(x)
y: (TDZ)	console.log(y)
	x = 10
	y = 20

3️⃣ Temporal Dead Zone (TDZ)

- **Definition:** Jab tak `let` aur `const` ko value assign nahi hoti, tab tak wo TDZ me rehte hain.
- Agar TDZ me variable ko access kiya → **ReferenceError** milta hai.

👉 Example:

```
console.log(y); // ❌ Cannot access 'y' before initialization
let y = 20;
```

✅ **Tip:**
Always declare variables at the **top** of their scope to avoid TDZ issues.

4️⃣ Hoisting

- Hoisting ka matlab hai **variables aur functions ki declaration** ko unke scope ke top pe “lift” kar dena.
- Sirf **declaration hoist hoti hai, initialization nahi.**

Case 1: var

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

👉 Yaha `var x;` hoist ho gaya, aur value baad me assign hoti hai.

Case 2: let/const

```
console.log(y); // ReferenceError
let y = 20;
```

👉 Yaha hoist to hota hai, par TDZ me rehta hai jab tak value assign na ho.

Case 3: Function Declaration

```
greet(); // ✅ Works
function greet() {
  console.log("Hello from greet");
}
```

👉 Pure function code memory me pehle se hota hai, isliye bina problem chalega.

Case 4: Function Expression (let)

```
meet(); // ❌ ReferenceError
let meet = function() {
  console.log("meet");
};
```

👉 `let` ke saath TDZ issue.

Case 5: Function Expression (var)

```
meet(); // ❌ TypeError: meet is not a function
var meet = function() {
  console.log("meet");
};
```

👉 Yaha `meet = undefined` hoist hua, isliye `meet()` call karte time undefined hai.

5️⃣ Example with Function & Variables

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

function add(num1, num2) {
  let result = num1 + num2;
  return result;
}

var ans = add(a, b);
console.log(ans); // 30
```

Execution Context

Memory	Code
a: undefined	a = 10
b: undefined	b = 20
add: function	ans = add(a, b)
ans: undefined	console.log(ans)

👉 Pehle memory me sab store, baad me code chalke result **30** print karega.

6️⃣ Quick Scenarios

✅ Scenario 1 (Function Declaration hoisted)

```
greet();
function greet() {
  console.log("Hello from greet");
}
// Output: Hello from greet
```

❌ Scenario 2 (Function Expression with let)

```
meet();
let meet = function() {
  console.log("meet");
};
// Error: Cannot access 'meet' before initialization
```

❌ Scenario 3 (Function Expression with var)

```
meet();
var meet = function() {
  console.log("meet");
};
// Error: meet is not a function
```

7️⃣ Summary (Ek Nazar Me)

- JS ek **Synchronous, Single-threaded** language hai.
- **Execution Context = Memory Phase + Code Execution Phase**.
- **Hoisting:** Declarations top pe move ho jati hai.
    - `var` → hoist hota hai with `undefined`.
    - `let/const` → hoist hote hain but TDZ me rehte hai.
    - Function Declaration → pura code memory me hota hai (works).
    - Function Expression → var ke sath → `undefined`, let/const ke sath → TDZ.
- **TDZ:** let/const ka wo period jab tak unko initialize nahi kiya gaya.

🔥 **Master Tip (Interview Point):**

- **"Hoisting doesn't mean moving code up"** → It just means **JS pehle memory allocate karta hai declarations ke liye**.
- Confusion avoid karne ke liye → Always declare variables/functions at the **top of their scope**.

📘 Lecture 18 – `this` Keyword in JavaScript

🌍 1. Global Object

👉 JavaScript code run hone ke liye ek **execution environment** chahiye hota hai.
Is environment ka **sabse bada object** hota hai → **Global Object**.

- **Browser me:** `window`
- **Node.js me:** `global`
- **Universal (ES2020+):** `globalThis`

❓ `console.log` aur `Math.random` kaha se aate hain?

- C++ me → `#include <iostream>` likhna padta hai.
- JavaScript me → Sab kuch **default global object** ke andar hota hai.

✅ Example:

```
console.log("Hello World");   // console global object ka part hai
console.log(Math.random());   // Math bhi global object ka part hai
```

👉 `console`, `Math`, `setTimeout`, `setInterval` → sab **global object properties** hain.
👉 `var` se declare kiya variable bhi global object ka part hota hai,
lekin `let` & `const` global object ke part **nahi** bante.

✅ Universal Access

```
console.log(window.Math.random());    // Browser
console.log(global.Math.random());    // Node.js
console.log(globalThis.Math.random()); // Har jagah
```

💡 **Tip:** Interview me puchha jata hai →
"console.log kaise kaam karta hai?"
Answer → `console` ek global object ki property hai aur `log()` uska method.

🔑 2. `this` in Different Contexts

✅ a) Global Context

```
console.log(this);

// Browser → window
// Node.js → {}
console.log(globalThis); // universal → window ya global
```

✅ b) Inside a Function

```
function greet() {
  console.log(this);
}
greet();
// Browser → window
// Node.js → global

"use strict";
function greetStrict() {
  console.log(this);
}
greetStrict(); // undefined
```

✅ c) Inside an Object Method

```
const obj = {
  name: "Harshal",
  sayName() {
    console.log(this.name);
  }
};
obj.sayName(); // Harshal
```

👉 Yaha `this` → jis object pe method call hua, usko point karega.

✅ d) Arrow Functions

👉 Arrow functions ka **apna `this` hota hi nahi**.
Wo **lexical parent se `this` inherit karte hain**.

```
const obj = {
  name: "Harshal",
  arrow: () => {
    console.log(this); // ❌ Global object, not obj
// output : // In browser: window ,  In Node.js: {} or global

  }
};
obj.arrow();
```

Lexical binding example:

```
const obj = {
  name: "Rohit",
  greet() {
    const arrow = () => console.log(this.name);
    arrow(); // Rohit
  }
};
obj.greet();
```

- **Arrow function**: `this` upar se inherit karta hai (lexical).
- **Regular function**: `this` depend karta hai **kaise call kiya**.

✅ e) Inside Class / Constructor

```
class Person {
  constructor(name) {
    this.name = name;
  }
}
const p1 = new Person("Saurav");
console.log(p1.name); // Saurav
```

👉 Constructor me `this` naya object ko point karta hai.

✅ f) `this` in setTimeout

```
setTimeout(function() {
  console.log(this); // window (browser)
}, 1000);

setTimeout(() => {
  console.log(this); // lexical (depends on parent)
}, 1000);
```

✅ g) `this` in Event Listeners

```
const btn = document.querySelector("button");

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

btn.addEventListener("click", () => {
  console.log(this); // lexical parent (not button)
});
```

📌 Important Tips

✔ Arrow functions ka apna `this` nahi hota → parent se inherit karte hain.
✔ `var` se declare variable global object ka part banta hai, `let` & `const` nahi.
✔ Browser vs Node.js me global object alag hota hai → universal → `globalThis`.

🔁 Summary – Ek Nazar Me

1. **Global Object:** Browser → `window`, Node.js → `global`, universal → `globalThis`.
2. **console.log & Math.random:** Global object ke andar ke methods.
3. **this:** Context ke hisaab se badalta hai:
    - Global → window/global
    - Function → global (strict me undefined)
    - Object method → jis object pe call hua

- Arrow → parent ka `this`
- Class/constructor → new object

4. **Tip:** Agar confuse ho jao → `console.log(globalThis)` karo.

📘 LECTURE 19 — DOM in JavaScript

First-Thought Principle (samjho seedha)

- **HTML** = ghar ka ** नक्शा** (structure)
- **CSS** = ghar ki ** सजावट** (style)
- **JavaScript + DOM** = ghar me **kaam karne wale log** (dynamic changes)
  Browser tumhara HTML ko **Object Tree (DOM)** me badalta hai, aur JS us tree ko **read / change / add / delete** kar sakti hai.

Base HTML (sirf ek <h1> ka example)

```
<!DOCTYPE html>
<html>
  <head>
    <title>Lecture 19</title>
  </head>
  <body>
    <h1 id="title" class="heading">Hello Bhai</h1>
  </body>
</html>
```

DOM Tree (high-level)

```
window
 └─ document
     └─ html
        └─ body
           └─ h1#title.heading ("Hello Bhai")
```

🌎 Global Object & Document

- **window** → Browser ka global object.
- **document** → Window ke andar HTML ka object.
- Example access:

```
console.log(window.document);   // Pura document
console.log(document.body);     // <body> content
```

```
console.log(document.head);     // <head> content
console.log(document.title);    // Page ka title
console.log(document.URL);      // Page URL
```

💡 **Tip**: JS me sab kuch **object ke form me hota hai**.
Jaise C++ me `#include<iostream>` se functions milte hai, waise JS me `window` object se `console.log`, `Math.random` milte hai.

Jab hum Live Server (ya browser) me HTML file run karte hain, to **poora HTML tree ek `document` object me convert ho jata hai**, jise DOM kehte hain — aur JavaScript se ise access, modify, ya update kiya ja sakta hai.

Elements ko Access kaise karein

① By ID (single element)

```
const el = document.getElementById('title');
console.log(el.innerHTML);  // "Hello Bhai"
```

② By Class Name (multiple)

```
const list = document.getElementsByClassName('heading');
console.log(list[0]);       // <h1 id="title" class="heading">Hello Bhai</h1>
```

⚡ **Note:** `getElementsByClassName` → **HTMLCollection** (array-like).
Agar array methods chahiye, to: `Array.from(list)`

QUESTION -> 100 elements same class ke ho to sab access ho jayenge?

ANSWER -> Haan, loop ya index se access kar sakte ho.

EXAMPLE :

```
<div class="box">One</div>

<div class="box">Two</div>

<div class="box">Three</div>


<script>
```

```
  const items = document.getElementsByClassName('box');

  Array.from(items).forEach(item => {

    console.log(item.innerHTML);

  });

</script>
```

OUTPUT:

```
One

Two

Three
```

“Behind the Scenes” — yeh <h1> object ban kar kaisa dikhta hai?

Console me aise socho (conceptual view — idea samajhne ke liye):

🔹 console.dir(el)

- **Element ko JavaScript object ki tarah dikhata hai.**
- Isme tum **saare properties, methods, prototype chain** dekh sakte ho.
- Zyada useful hota hai jab tum JavaScript-level details samajhna chahte ho.

```
// Console me: console.dir(el)
{
  id: "title",
  className: "heading",
  tagName: "H1",
  innerHTML: "Hello Bhai",
  innerText: "Hello Bhai",
```

```
  textContent: "Hello Bhai",
  style: CSSStyleDeclaration {...},
  // prototype chain:
  __proto__: HTMLHeadingElement → HTMLElement → Element → Node →
EventTarget → Object
}
```

> Yaani browser is `<h1>` ko **HTMLHeadingElement** object ke through expose karta hai jisme methods/properties ready milti hain.

Chhota demo (property change):

```
el.innerText = "Namaste Bhai";          // Text change
el.style.color = "crimson";             // Inline CSS color
el.className = "heading hero";          // All classes replaced
el.classList.add('new');                // "new" class added
el.classList.remove('heading');         // "heading" class removed
```

📦 Final Output:

```
<h1 id="title" class="hero new" style="color: crimson;">
  Namaste Bhai
</h1>
```

❓ “HTML elements ko object me hi kyu convert kiya jata hai?”

🟢 Seedha Jawaab:

> **JavaScript sirf objects ke saath kaam kar sakti hai**, isliye browser HTML elements ko **object me convert** karta hai — taaki hum unhe JS se control kar sakein.

📚 Short Notes:

- JS ko **object ke properties/methods** samajh aate hain.
- Har HTML element ko browser **DOM object** bana deta hai.
- Fir hum use JS se control karte hain:
  - `.innerText` → text change
  - `.style` → CSS change

    - `.appendChild()` → naya element add

- Yeh object browser ke andar hota hai, asli code C++/Rust me hota hai.
- JS ko sirf ek **wrapper object** milta hai.

`document` ko dekhna (pure page ka DOM object)

```
console.log(document);   // Document object (pure page ka tree)
console.dir(document);   // properties/methods ka detailed view
```

> **Common mistake fix:** `console.log(documnet)` ❌ → sahi: `console.log(document)` ✅

Mini “Why/How” Recap (ek nazar me)

- Browser HTML padhta hai → **DOM (Object Tree)** banata hai.
- JS DOM ko **objects** ki tarah handle karti hai (properties/methods).
- `getElementById` → **single** object; `getElementsByClassName` → **live collection**.
- Console me elements **HTMLHeadingElement** jaise types ke objects hote hain.
- `console.log(document)` se **poora tree** dikhta hai; wahi tumhara playground hai.

Quick Tips

- **Text vs HTML**: `innerText`/`textContent` sirf text; `innerHTML` me tags bhi.
- **Classes**: `classList.add/remove/toggle` use karo; `className` poora replace karta hai.
- **Collections**: HTMLCollection ko array banana ho to `Array.from(...)`.
- **Debug**: `console.dir(element)` structure dekhne ke liye best.

🔁 Summary – Ek Nazar Me

- DOM = Browser ka **Object Model** of HTML.
- Har element (jaise `<h1>`) JS me **object** ban jata hai (methods + properties ke saath).
- Access basics: **ID** (single), **Class** (HTMLCollection).
- `document` print karke **poora DOM** dekh sakte ho.
- Objects me convert isliye hota hai kyunki JS **objects ko manipulate** karke page ko dynamic banati hai.

📘 Lecture 20 – Accessing DOM Elements

🌱 **First Thought Principle (Desi Style)**

DOM = ek joint family ka ped 🌳

Parent → Child → Sibling bilkul ghar ke rishton ki tarah.

- **innerHTML** = aadmi ke kapde + style bhi dikhte hain 👕
- **innerText** = sirf saamne dikhne wali baat 👀
- **textContent** = andar ka sach bhi milta hai (hidden bhi).

🏗️ Example HTML

```
<!DOCTYPE html>

<html>

<head>

  <title>Access DOM</title>

</head>

<body>

  <h1 id="title" class="heading">Hello DOM</h1>

  <h1 class="heading">Second Heading</h1>

  <h1>Third Heading</h1>


  <p id="first">First Para</p>

  <p>Second Para</p>


  <ul id="list">

    <li>HTML</li>

    <li>CSS</li>

    <li>JavaScript</li>
```

```
  </ul>

</body>

</html>
```

🔎 1. Accessing by CSS Selectors

(a) Single Element → `querySelector()`

Definition:
CSS selector ka use karke **document me se pehla matching element** return karta hai.

Example:

```
const h1 = document.querySelector('#title');

console.log(h1.innerHTML);

// Output: Hello DOM
```

👉 Sirf **pehla match** return karega.

(b) Multiple Elements → `querySelectorAll()`

- **Definition:**
  CSS selector se **saare matching elements ka static NodeList** return karta hai.
- **Example:**

```
const allH1 = document.querySelectorAll('h1');


// For loop

for (let i = 0; i < allH1.length; i++) {

  console.log(allH1[i].innerText);

}

// Output:
```

```
// Hello DOM

// Second Heading

// Third Heading
```

🔹 `allH1 = document.querySelectorAll('h1')`
➡️ Ye ek **NodeList** deta hai

🔹 NodeList kya hota hai?

- **Definition:**
  NodeList ek **array-jaisi collection** hoti hai jo multiple DOM nodes rakhti hai.
  Isme indexing hoti hai aur ye loop ke through easily iterate ho sakta hai.

- **Example NodeList Structure:**

```
NodeList(3) [
  0: <h1 id="title" class="heading">Hello DOM</h1>,
  1: <h1 class="heading">Second Heading</h1>,
  2: <h1>Third Heading</h1>
]
```

NodeList ko Array me convert kaise karein?

- **Why?**
  NodeList me sab array methods (like map, filter) nahi hote.

- **Methods:**

```
const arr = Array.from(allH1);

const arr2 = [...allH1];
```

- Ab array methods easily use kar sakte ho.

```
// forEach
allH1.forEach(val => console.log(val.innerText));
// Output:
// Hello DOM
// Second Heading
// Third Heading

// for...of loop
for (let val of allH1) {
  console.log(val.innerText);
}
// Output:
// Hello DOM
// Second Heading
// Third Heading
```

💡 **NodeList** → `forEach` chal jata hai.
Agar array methods use karne ho → `Array.from(allH1)`.

🔎 2. Accessing Elements by Tag Name

- **Tag Name:**
  HTML element ka naam jaise `h1`, `p`, `li`, `div`, `a`, etc.

- **Method:**
  `getElementsByTagName('tagname')` sab matching elements ko **live HTMLCollection** me return karta hai.

- **Example:**

```
let liItems = document.getElementsByTagName('li');


for (let i = 0; i < liItems.length; i++) {
  liItems[i].style.color = "blue";
}
// Output: Sare <li> blue color me ho jayenge
```

⚡ Return → **HTMLCollection (live).**

👨‍👩‍👧 3. Accessing by Relationship

(a) Parent & Child Nodes

- **parentNode / parentElement:**
  Kisi element ka **parent node ya element** return karta hai.

- **children:**
  Sirf **child elements** return karta hai (ignores text nodes).

- **childNodes:**
  Saare children return karta hai — elements + text nodes (like spaces, line breaks).

- **Example:**

```
const ul = document.getElementById('list');


console.log(ul.parentNode);
// Output: <body>...</body>


console.log(ul.children);
// Output: HTMLCollection(3) [li, li, li]


console.log(ul.childNodes);
```

```
// Output: NodeList(7) [text, li, text, li, text, li, text]
```

`childNodes` me ye sab nodes aate hain: elements + text nodes (whitespace, newlines)Isliye total count **7 hota hai** (3 `<li>` elements + 4 text nodes)

(b) First & Last Child

Property	Meaning	Output Example
`firstChild`	Pehla **node** (element ya text)	`#text` (line break bhi count)
`firstElementChild`	Pehla **child element**	`<li>HTML</li>`
`lastChild`	Last **node** (element ya text)	`#text` (line break)
`lastElementChild`	Last **child element**	`<li>JavaScript</li>`

Example:

```
console.log(ul.firstChild);

// Output: #text (line break bhi count hoga)


console.log(ul.firstElementChild);

// Output: <li>HTML</li>


console.log(ul.lastChild);

// Output: #text (line break)


console.log(ul.lastElementChild);

// Output: <li>JavaScript</li>
```

(c) Sibling Nodes

Property	Meaning	Example Output

`nextSibling`	Next **node** (element or text)	`#text` (whitespace)
`nextElementSibling`	Next **element node** only	`<p>Second Para</p>`
`previousSibling`	Previous **node** (element or text)	`#text` (whitespace)
`previousElementSibling`	Previous **element node** only	`null` (agar pehla element ho)

> 💡 **Best Practice:** Hamesha **ElementSibling** properties use karo, taaki text nodes se bach sako.

```
const p = document.getElementById('first');


console.log(p.nextSibling);

// Output: #text


console.log(p.nextElementSibling);

// Output: <p>Second Para</p>


console.log(p.previousSibling);

// Output: #text


console.log(p.previousElementSibling);

// Output: null (kyunki yeh pehla para hai)
```

💡 Best Practice → hamesha **ElementSibling** use karo.

📝 4. innerHTML vs innerText vs textContent

- **innerHTML:**
  Element ke andar ka pura **HTML markup + text** return karta hai.

- **innerText:**
  Sirf visible text content return karta hai (jo CSS se hidden hai, wo nahi aata).

- **textContent:**
  Element ke andar ka **saara text** return karta hai, chahe wo visible ho ya hidden.

```
<p id="demo">Hello <b style="display:none">Hidden</b> World</p>
```

```
console.log(demo.innerHTML);
// Output: Hello <b style="display:none">Hidden</b> World
```

```
console.log(demo.innerText);
// Output: Hello World
```

```
console.log(demo.textContent);
// Output: Hello Hidden World
```

✅ Quick Recap (With Definitions)

✅ Quick Recap (With Definitions)

Topic	Definition / Behavior
Selectors	
`querySelector()`	Pehla matching element return karta hai.

`querySelectorAll()`	Saare matching elements return karta hai (NodeList – static).
`getElementsByTagName()`	Saare given tag ke elements return karta hai (HTMLCollection – live).

Relationships

`parentNode` / `parentElement`	Element ka parent (node ya element).
`children`	Sirf child elements return karta hai (HTMLCollection).
`childNodes`	Sab nodes (text + comments + elements) return karta hai (NodeList).
`firstChild` / `lastChild`	Pehla/Last node (text bhi count hota hai).
`firstElementChild` / `lastElementChild`	Pehla/Last child element (text ignore hota hai).
`nextElementSibling` / `previousElementSibling`	Next/Previous sirf element nodes return karta hai.

Text Access

`innerHTML`	Pura HTML markup + text return karta hai.

`innerText`	Sirf visible text return karta hai (hidden ignore).
`textContent`	Saara text return karta hai (hidden bhi include).

Ab jab bhi ye notes dekho, bas itna yaad rakhna:
DOM elements ek tree structure me hote hain, jisko hum alag-alag selectors aur properties se access karte hain — text access ke liye innerHTML, innerText aur textContent alag-alag tarike se content dikhate hain.

📘 Lecture 21 – Creating & Modifying DOM Elements (Best Notes)

🧱 1. `createElement()` + `appendChild()`

➕ Create element and add to parent

```
<ul id="root"></ul>


<script>

const li = document.createElement("li");

li.innerHTML = "Hello World";


const parent = document.getElementById("root");

parent.appendChild(li);

</script>
```

✅ Output:

```
<ul id="root">

  <li>Hello World</li>

</ul>
```

- 🔹 `createElement()` → Creates a new element in memory
- 🔹 `appendChild()` → Appends it as **last child** of a parent

👨‍👩‍👧‍👦 2. `append()` – Add Multiple Elements

```
<ul id="root"></ul>
```

```
<script>
const li1 = document.createElement("li");
li1.innerHTML = "HTML";

const li2 = document.createElement("li");
li2.innerHTML = "CSS";

document.getElementById("root").append(li1, li2);
</script>
```

✅ **Output:**

```
<ul id="root">
  <li>HTML</li>
  <li>CSS</li>
</ul>
```

- `append()` → Can add **multiple nodes or strings**

⬅️ 3. `prepend()` – Add at Start

```
<ul id="root">
  <li>JavaScript</li>
</ul>

<script>
const li = document.createElement("li");
li.innerHTML = "HTML";
```

```
document.getElementById("root").prepend(li);
</script>
```

✅ **Output:**

```
<ul id="root">
  <li>HTML</li>   <!-- Added first -->
  <li>JavaScript</li>
</ul>
```

- `prepend()` → Adds the node at **beginning**

🧾 4. `createTextNode()`

```
<ul id="root"></ul>

<script>
const text = document.createTextNode("Node Text");
const li = document.createElement("li");

li.appendChild(text);
document.getElementById("root").appendChild(li);
</script>
```

✅ **Output:**

```
<ul id="root">
  <li>Node Text</li>
</ul>
```

- 🔸 `createTextNode()` → Creates only text (without HTML)

🔄 5. `replaceChild()`

```
<ul id="root">
  <li>Old</li>
</ul>

<script>
const newLi = document.createElement("li");
newLi.innerHTML = "New";

const root = document.getElementById("root");
root.replaceChild(newLi, root.children[0]);
</script>
```

✅ **Output:**

```
<ul id="root">
  <li>New</li>   <!-- Old replaced -->
</ul>
```

- 🔸 `replaceChild(new, old)` → Replaces a child node with new one

💻 6. `innerHTML` – Direct HTML Set

```
<ul id="root"></ul>
```

```
<script>
const root = document.getElementById("root");
root.innerHTML += "<li>React</li>";
root.innerHTML += "<li>Node.js</li>";
</script>
```

✅ **Output:**

```
<ul id="root">
  <li>React</li>
  <li>Node.js</li>
</ul>
```

⚠️ **Note:**
`innerHTML` re-parses the full content — may affect performance in large DOMs.

📍 7. `insertAdjacentElement()` – Precise Placement

```
<ul id="root">
  <li>JavaScript</li>
</ul>
```

```
<script>
const li = document.createElement("li");
li.innerHTML = "HTML";

const root = document.getElementById("root");
root.insertAdjacentElement("beforebegin", li);
</script>
```

✅ **Output:**

```
<li>HTML</li>     <!-- Outside UL -->
<ul id="root">
  <li>JavaScript</li>
</ul>
```

🔸 **Options for `insertAdjacentElement(position, element)`:**

- `"beforebegin"` → Outside, before element
- `"afterbegin"` → Inside, at beginning
- `"beforeend"` → Inside, at end
- `"afterend"` → Outside, after element

🗑️ 8. `remove()` – Direct Element Removal

```
<ul id="root">
  <li>HTML</li>
  <li>CSS</li>
</ul>

<script>
document.querySelector("li").remove();
</script>
```

✅ **Output:**

```
<ul id="root">
  <li>CSS</li>   <!-- First removed -->
```

```
</ul>
```

- `element.remove()` → Directly removes selected element

❌ 9. `removeChild()` – Remove Specific Child

```
<ul id="root">
  <li>HTML</li>
  <li>CSS</li>
</ul>

<script>
const root = document.getElementById("root");
const secondLi = root.children[1];
root.removeChild(secondLi);
</script>
```

✅ **Output:**

```
<ul id="root">
  <li>HTML</li>
</ul>
```

- `removeChild(child)` → Requires reference to child node

🧾 10. Attribute Methods

```
<h1 id="title" class="heading">Hello</h1>
```

```
<script>

const el = document.getElementById("title");


console.log(el.getAttribute("class"));   // heading


el.setAttribute("style", "color:red;");

// <h1 style="color:red">Hello</h1>


el.removeAttribute("style");

// Style removed

</script>
```

Method	What it does
`getAttribute(name)`	Gets value of attribute
`setAttribute(name, value)`	Sets or updates an attribute
`removeAttribute(name)`	Removes the attribute completely

🔁 Quick Summary – Ek Nazar Mein

Method	Use Case
`createElement()`	Create new HTML element

`createTextNode()`	Create pure text (no tags)
`appendChild()`	Add element as **last child**
`append()`	Add **multiple elements**
`prepend()`	Add element at **start**
`replaceChild(new, old)`	Replace existing child element
`innerHTML`	Set raw HTML inside element
`insertAdjacentElement()`	Insert at specific position
`remove()`	Remove element directly
`removeChild()`	Remove known child from parent
`getAttribute()`	Read attribute
`setAttribute()`	Add/modify attribute
`removeAttribute()`	Remove attribute

🧠 Tip to Remember:

DOM elements sirf dikhte nahi, **dynamicly banaye, badle, hataaye ja sakte hain.** Use the **right method for the right task** – and avoid `innerHTML` when working with many elements.

📘 Lecture 22 – Events in JavaScript

🌱 First Thought Principle (Desi Style)

Soch lo web page ek **mela (fair)** hai 🎪

- Button dabaya → `click` 🖱️
- Form bhara → `submit` 📝
- Keyboard bajaya → `keydown` ⌨️

👉 **Event = koi ghatna jo page pe hoti hai**, aur JavaScript kaam me lag jaata hai.

🏷️ What is an Event?

Event = Browser me hone wali **action/ghatna**, jaise:

- Click , Scroll ,Typing ,Form bharna ,Mouse ghoomna etc.

JavaScript in sab events ko **sunta hai** aur **react karta hai** using `addEventListener`.

🖱️ Mouse Events (Event types for mouse)

Event	Trigger kab hota hai
`click`	Jab mouse se ek baar click karo
`dblclick`	Jab double click hota hai
`mousedown`	Mouse button dabate hi
`mouseup`	Mouse button chhodte hi

`mouseover`	Mouse element ke upar aata hai
`mouseout`	Mouse element se bahar jata hai
`mousemove`	Mouse hil raha ho element ke andar

✅ Example:

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

<script>

const btn = document.getElementById("btn");


btn.addEventListener("click", () => console.log("Button clicked!"));

btn.addEventListener("dblclick", () => console.log("Double clicked!"));

btn.addEventListener("mousedown", () => console.log("Pressed!"));

btn.addEventListener("mouseup", () => console.log("Released!"));

btn.addEventListener("mouseover", () => console.log("Entered!"));

btn.addEventListener("mouseout", () => console.log("Left!"));

</script>
```

⌨️ Keyboard Events

Event	Description

`keydown`	Key dabate hi trigger
`keyup`	Key chhodte hi trigger

✅ Example:

```
<input type="text" id="box">

<script>

const box = document.getElementById("box");


box.addEventListener("keydown", (e) => console.log("Pressed:", e.key));

box.addEventListener("keyup", (e) => console.log("Released:", e.key));

</script>
```

📝 Form Events

Event	Kya karta hai
`submit`	Form submit par trigger
`change`	Input value badalne par
`focus`	Cursor input me aate hi
`blur`	Cursor input se jaate hi

✅ Example:

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

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

form.addEventListener("submit", (e) => {
  e.preventDefault(); // 🚫 Page reload nahi hoga
  console.log("Form submission prevented");
});

form.addEventListener("change", (e) => console.log("Changed:", e.target.value));
form.addEventListener("focus", () => console.log("Input focused"), true);
form.addEventListener("blur", () => console.log("Input blurred"), true);
</script>
```

🎧 Event Listener Methods

Method	Kya karta hai

`addEventListener()`	Event sunta hai
`removeEventListener()`	Event listener hata deta hai

✅ Example:

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

<script>

function showAlert() {

  alert("Clicked!");

}


const btn = document.getElementById("btn");

btn.addEventListener("click", showAlert);


// Remove after 5 sec

setTimeout(() => {

  btn.removeEventListener("click", showAlert);

}, 5000);

</script>
```

🎁 Event Object

Jab koi event hota hai, JavaScript ek **event object** deta hai — jisme saari info hoti hai:

Property	Description
`type`	Kis type ka event tha (e.g. 'click')
`target`	Kis element pe event hua
`currentTarget`	Jispe listener laga tha

✅ Example:

```
btn.addEventListener("click", function (event) {
  console.log("Type:", event.type);          // click
  console.log("Target:", event.target);      // <button>...</button>
});
```

🎛️ Event Object Methods

1. `preventDefault()` – Default kaam rokta hai

Example: Link open hone se rokna

```
link.addEventListener("click", (e) => {
  e.preventDefault();
  console.log("Navigation stopped");
});
```

---

2. `stopPropagation()` – Event bubbling rokta hai

Example: Child pe click se parent trigger na ho

```
child.addEventListener("click", (e) => {
  e.stopPropagation();
  console.log("Only child clicked");
});
```

🖱️ Mouse Event Properties

Property	Description
`clientX`	X position (viewport based)
`clientY`	Y position (viewport based)
`pageX`	X position (page based)
`pageY`	Y position (page based)
`button`	Konsa mouse button (0,1,2)

✅ Example:

```
div.addEventListener("mousemove", (e) => {
  console.log("Mouse at:", e.clientX, e.clientY);
});
```

⌨️ Keyboard Event Properties

Property	Description
`key`	Konsi key press hui

✅ Example:

```
input.addEventListener("keydown", (e) => {
  console.log("Pressed:", e.key);
});
```

✅ QUICK RECAP (1-min Revision)

👉 Event = Page pe hone wali koi ghatna (action)

👉 addEventListener() = Sunta hai event

👉 removeEventListener() = Hataata hai event

👉 Mouse Events = click, dblclick, mouseover, mouseout

👉 Keyboard Events = keydown, keyup

👉 Form Events = submit, change, focus, blur

👉 preventDefault() = Default kaam roko

👉 stopPropagation() = Parent ko bubble jane se roko

👉 Event Object = Info deta hai (type, target, key, etc.)

👉 Mouse Properties = clientX, pageX, button

👉 Keyboard Properties = key

💡 Real-life Analogy

Action	Web Event Equivalent
Darwaza khola	click
Light switch daba	mousedown
Switch chhoda	mouseup
Button ke paas gaya	mouseover
Keyboard dabaya	keydown
Keyboard chhoda	keyup
Naam likha	change
Form submit kiya	submit

🌍 Real Life Example: Door Bell System 🛎️

🔸 Scenario:

- Ek **ghar** hai jisme ek **door bell** lagi hai.
- Koi **guest aata hai aur bell bajata hai**.
- Ghar ke andar ka system us bell bajne pe **react karta hai** (jaise koi aake gate kholta hai).

🔁 Web Page version of same:

Real Life	Web Page Equivalent

Bell Button	HTML Button (`<button>`)
Guest ne bell bajayi	User ne button click kiya (event)
Bell ka sound aaya	JavaScript ne `event` detect kiya
Gharwala darwaza khola	JS ne kuch kaam kiya (function run)

✅ Code Example (Bell Click Simulation):

```
<button id="bell">🔔 Ring Bell</button>


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

  // Ye function react karega jab bell bajegi
  function openGate() {
    console.log("Gate opened! Welcome Guest.");
  }

  // Event Listener laga diya
  bell.addEventListener("click", openGate);
</script>
```

Yaha `openGate` likha hai, **`openGate()` nahi**, kyuki hum usse turant nahi chahte, **bas reference de rahe hain**.

📦 Output:

Jab user 🔔 `Ring Bell` button pe click karega:

```
Gate opened! Welcome Guest.
```

✅ Event Object in Real Life

Jab bell bajti hai (event trigger hota hai), toh system ye bhi jaan sakta hai:

Real-life Info	JS me kya hota hai (`event` object se)
Kisne bell bajayi?	`event.target`
Kab bajayi?	`event.timeStamp`
Kitni baar bajayi?	Count logic laga sakte ho
Kahan se bajayi (left/right hand)?	`event.button` (mouse events)

✅ Event Object Example:

```
bell.addEventListener("click", function (event) {
  console.log("Bell clicked by:", event.target);
  console.log("Event type:", event.type);
});
```

✅ Summary – Ek Line me:

> **Event** = Web page pe koi *action ya ghatna*, jise JS *detect* karke kuch *reaction* karti hai.
> Jaise bell bajana = event, aur gate kholna = JavaScript ka response.

📘 Lecture 23 – Event Bubbling, Event Capturing & Delegation

🔴 Event Bubbling (Default Behavior)

👉 Matlab: Jab tu kisi **andar wale element** (child) pe click karta hai, to event **upar ki taraf bubble** hota hai (child → parent → grandparent).
Ye default hota hai JS me.

Example:

```
<div id="grandparent" style="padding:30px; background:lightblue;">
  Grandparent
  <div id="parent" style="padding:20px; background:lightgreen;">
    Parent
    <button id="child" style="padding:10px;">Child</button>
  </div>
</div>


<script>
document.getElementById("child").addEventListener("click", () => {
  console.log("Child clicked");
}, false);


document.getElementById("parent").addEventListener("click", () => {
  console.log("Parent clicked");
}, false);


document.getElementById("grandparent").addEventListener("click", () =>
{
```

```
  console.log("Grandparent clicked");
}, false);
</script>
```

Output (Child pe click karne par):

```
Child clicked
Parent clicked
Grandparent clicked
```

👉 Isko **“bubble”** isliye kehte hain kyunki event andar se bahar ki taraf **bubble hota hai**.

🟢 Event Capturing (Trickle Down)

👉 Matlab: Event sabse **bahar wale element (grandparent)** se shuru hota hai, aur **andar tak descend** karta hai.
Ye default nahi hota, iske liye `true` likhna padta hai.

Example:

```
<script>
document.getElementById("child").addEventListener("click", () => {
  console.log("Child clicked");
}, true);

document.getElementById("parent").addEventListener("click", () => {
  console.log("Parent clicked");
}, true);

document.getElementById("grandparent").addEventListener("click", () =>
{
  console.log("Grandparent clicked");
```

```
}, true);

</script>
```

Output (Child pe click karne par):

```
Grandparent clicked

Parent clicked

Child clicked
```

👉 Isko **“capturing / trickling”** kehte hain kyunki event bahar se andar ki taraf **trickle hota hai**.

🟡 Difference: Bubbling vs Capturing

- **Bubbling (false / default)** → andar se bahar
- **Capturing (true)** → bahar se andar

🟣 Event Delegation

👉 Matlab: Har ek child element pe alag listener lagane ki jagah, ek **parent pe listener** lagado aur check karo ki kaunsa child click hua.
Isse code clean ho jata hai aur performance fast hoti hai.

Example:

```
<div id="menu">

  <button id="home">Home</button>

  <button id="about">About</button>

  <button id="contact">Contact</button>

</div>


<script>

document.getElementById("menu").addEventListener("click", (event) => {
```

```
  if (event.target.tagName === "BUTTON") {
    console.log(event.target.id + " clicked");
  }
});
</script>
```

Output:

- Agar **Home button** dabaya → `home clicked`
- Agar **About button** dabaya → `about clicked`
- Agar **Contact button** dabaya → `contact clicked`

👉 Matlab: Ek hi event listener se sab button handle ho gaye.

✅ Quick Recap

- **Event Bubbling** → andar se bahar (default).
- **Event Capturing** → bahar se andar (true use karke).
- **Event Delegation** → Parent pe ek hi listener lagake child elements handle karna.

⚡ Desi Soch:

- Bubbling = **"ghar ke andar se shor macha → poore mohalla me ghoom gaya"**
- Capturing = **"mohalla ke bahar se shor start hua → andar ghar tak aa gaya"**
- Delegation = **"Mohalle ke gate pe ek chowkidar rakha, woh decide karega kis ghar ka guest aya"** 😅

📘 Lecture 24 – FORM HANDLING in javJavaScript

🧾 Form = User ke data lene ka sabse bada source

(Jaise: naam, email, password, feedback, etc.)

⚙️ Agar tu form ka data **sahi se handle karna seekh gaya** to:
✅ Login System
✅ Signup Form
✅ Search Box
➡️ Sab easy ban jaayenge! 🚀

🔑 1. Input Event – 🧠 *Live Typing*

📌 **Jab user type karta hai**, turant event trigger hota hai.

```
<form id="myForm">

  <input type="text" placeholder="Type something..." />

</form>


<script>

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

form.addEventListener("input", (event) => {

  console.log("Typed:", event.target.value);

});

</script>
```

🖥️ **Output**
Typing `"Hello"`:

```
H

He

Hel
```

```
Hell

Hello
```

🧠 Use it for **live feedback, search suggestions, etc.**

🔑 2. Change Event – 🔁 *Change + Focus Lost*

📌 Trigger hota hai **jab value change hoti hai + focus chhodte ho**

```
<form id="myForm">
  <input type="text" placeholder="Enter name" />
</form>


<script>
form.addEventListener("change", (event) => {
  console.log("Changed Value:", event.target.value);
});
</script>
```

🖥️ **Output**

> Input me `"Harshal"` likha → bahar click kiya
> ➡️ `Changed Value: Harshal`

🧠 Use it for **final value validation** (e.g. email, phone)

🔑 3. Focus / Focusin Event – 🔦 *Light Aayi*

- `focus` → Sirf current element (🚫 bubble nahi hota)
- `focusin` → Bubble hota hai ✔ (form level pe use kar sakte ho)

```
<form id="myForm">
```

```
  <input type="text" placeholder="Name" />
  <input type="email" placeholder="Email" />
</form>

<script>
form.addEventListener("focusin", (event) => {
  console.log(event.target.placeholder, "got focus");
});
</script>
```

🖥️ **Output**
➡️ Clicked on Email field
`Email got focus`

🧠 Use it to highlight active input box

🔑 4. Blur / Focusout Event – 🌑 *Light Gayi*

- `blur` → Non-bubbling ❌
- `focusout` → Bubbling ✔

```
<form id="myForm">
  <input type="text" placeholder="Name" />
  <input type="email" placeholder="Email" />
</form>

<script>
form.addEventListener("focusout", (event) => {
  console.log(event.target.placeholder, "lost focus");
});
```

```
</script>
```

🖥️ **Output**
➡️ Name field se bahar nikla

```
Name lost focus
```

🧠 Use it to hide error messages or UI hints

🔑 5. Click / Double Click Event – 🖱️ *Mouse Interaction*

📌 Form ke andar kahin bhi **click ya double click** detect karo.

```
<form id="myForm">

  <button>Click Me</button>

</form>


<script>

form.addEventListener("click", () => {

  console.log("Form clicked");

});


form.addEventListener("dblclick", () => {

  console.log("Form double clicked");

});

</script>
```

🖥️ **Output**

```
Form clicked
Form double clicked
```

🧠 Use it for triggering animations, confirmations, etc.

🔑 6. Submit Event – 📤 *Form Bhejna*

📌 Jab form submit hota hai, ye trigger hota hai.
⚠️ By default, **page reload ho jaata hai** → Rokne ke liye `preventDefault()` lagao.

```
<form id="myForm">

  <input type="text" placeholder="Enter name" />

  <button type="submit">Submit</button>

</form>


<script>

form.addEventListener("submit", (event) => {

  event.preventDefault(); // Page reload rokna

  console.log("Form submitted!");

});

</script>
```

🖥️ **Output**

```
Form submitted!
```

🧠 Submit karte waqt backend call yahi se hota hai

🔑 7. Reset Event – 🧹 *Form Clean Karo*

📌 Jab **reset button** press hota hai, saare inputs clear ho jaate hain.

```
<form id="myForm">

  <input type="text" placeholder="Name" />

  <button type="reset">Reset</button>

</form>


<script>
```

```
form.addEventListener("reset", () => {
  console.log("Form reset, inputs cleared!");
});
</script>
```

🖥️ **Output**

```
Form reset, inputs cleared!
```

🧠 Helpful in clear button or cancel forms

🔑 8. FormData API – 📦 *Form Ka Dabba*

📌 Ye form ke sabhi inputs ko **key-value pair** me collect karta hai — boss level tool 🔥

```
<form id="myForm">
  <input type="text" name="username" placeholder="Name" />
  <input type="email" name="email" placeholder="Email" />
  <button type="submit">Submit</button>
</form>

<script>
form.addEventListener("submit", (event) => {
  event.preventDefault();

  const data = new FormData(form);

  for (let [key, value] of data.entries()) {
    console.log(key, ":", value);
  }
```

```
  console.log("Keys:", Array.from(data.keys()));
  console.log("Values:", Array.from(data.values()));
});
</script>
```

🖥️ **Output** (agar Name = Harshal, Email = test@gmail.com):

```
username : Harshal
email : test@gmail.com
Keys: ["username", "email"]
Values: ["Harshal", "test@gmail.com"]
```

🧠 Use it to send data to backend via `fetch()` or `axios`

✅ Quick Recap Table

🔑 Event	🧠 Description
`input`	Type karte hi trigger
`change`	Value change + focus lost
`focus/focusin`	Input box par focus aaya (light on)
`blur/focusout`	Input box chhoda (light off)
`click/dblclick`	Mouse interactions

`submit`	Form bhejna (preventDefault zaroori)
`reset`	Saare inputs clear
`FormData`	Saare data ko ek dabbe me collect karna

⚡ Desi Tricks (Yaad Rakhne ke Liye)

- 🧠 **Input** = Live typing
- 🔁 **Change** = Typing + focus bahar
- 🔦🌑 **Focus/Blur** = Light aayi / Light gayi
- 📤 **Submit** = Form bhejna
- 🧹 **Reset** = Form saaf karna
- 📦 **FormData** = Form ka dabba, saara saman andar 😄

📘 Lecture 25 –Callback Hell in JavaScript

🌱 Callback Kya Hai?

Callback function ek aisi function hoti hai jo dusri function ko ek **argument** ke roop me di jaati hai, taaki jab ek kaam complete ho jaaye to use **call back** kiya ja sake.

👉 Example soch:
"Hey JavaScript, jab tum user ka data fetch kar lo, tab mujhe call kar lena, taaki main greeting dikhau."

🔑 Example Without Callback

```
function fetchUser() {


    console.log("Fetching the user details.....");

    setTimeout(() => {

    console.log("Data fetched successfully");

    const name = "Saurav";

    greet(name);

    meet(name);

  }, 2000);

}


function greet(name) {

  console.log(`Hello ${name}`);

}


function meet(name) {

  console.log(`Hello ${name}, I will meet you in Delhi`);
```

```
}


fetchUser();
```

🖥️ Output:

```
Fetching the user details.....

Data fetched successfully

Hello Saurav

Hello Saurav, I will meet you in Delhi
```

⚠️ Limitation → Fixed hai ki data fetch hone ke baad `greet` aur `meet` hi chalenge. Flexibility nahi hai.

🔑 Example With Callback

```
function fetchUser(callback) {

  console.log("Fetching the user details.....");


  setTimeout(() => {

    console.log("Data fetched successfully");

    const name = "Saurav";

    callback(name); // Yaha hum decide karenge ki kaunsa function chalana hai

  }, 2000);

}


function greet(name) {

  console.log(`Hello ${name}`);
```

```
}


function meet(name) {
  console.log(`Hello ${name}, I will meet you in Delhi`);
}


// Choice hamare haath me hai
fetchUser(greet);
// ya
fetchUser(meet);
```

🖥️ Output (agar greet pass karein):

```
Fetching the user details.....
Data fetched successfully
Hello Saurav
```

🖥️ Output (agar meet pass karein):

```
Fetching the user details.....
Data fetched successfully
Hello Saurav, I will meet you in Delhi
```

✅ Flexibility → Ab hum decide kar sakte hain ki data fetch hone ke baad **kya karna hai**.

🍕 Callback Hell – Pizza Example

Soch, tu Domino's se pizza order karta hai. Steps:
1️⃣ Order place karna
2️⃣ Pizza prepare karna

3️⃣ Delivery boy order pickup karega
4️⃣ Delivery boy ghar laake dega

```
function placeOrder (callback) {
  console.log("Talking with Domino's");
  setTimeout(() => {
    console.log("Order placed successfully");
    callback();
  }, 2000);
}

function preparingOrder (callback) {
  console.log("Pizza preparation started...");
  setTimeout(() => {
    console.log("Pizza preparation done!");
    callback();
  }, 3000);
}

function pickupOrder (callback) {
  console.log("Reaching restaurant to pick up the order...");
  setTimeout(() => {
    console.log("Order picked up by the delivery boy");
    callback();
  }, 2000);
}

function deliverOrder () {
```

```
  console.log("Delivery boy is on the way...");
  setTimeout(() => {
    console.log("Order delivered successfully");
  }, 3000);
}

// Nested callback
placeOrder(() => {
  preparingOrder(() => {
    pickupOrder(() => {
      deliverOrder();
    });
  });
});
```

🖥️ Output:

```
Talking with Domino's
Order placed successfully
Pizza preparation started...
Pizza preparation done!
Reaching restaurant to pick up the order...
Order picked up by the delivery boy
Delivery boy is on the way...
Order delivered successfully
```

😵 Problems of Callback Hell

1. **Readability khatam** → Itna nested code dekh ke lagta hai coding nahi *maggi ban rahi hai*.
2. **Debugging mushkil** → Error aaya to samajhna badi problem.
3. **Dependency issues** → Har ek step previous step pe depend hai. Agar ek atka to sab atka.

🧵 JavaScript Nature

🔹 Single-Threaded

JS ek hi time pe ek hi task karta hai (single-thread).
Code sequence me execute hota hai.

```
console.log(10);

const times = Date.now();

while (Date.now() - times < 2000) {

   // wait 2 seconds

}

console.log(20);

console.log(30);
```

🖥️ Output:

```
10

20

30
```

👉 Yaha `while loop` ne 2 sec ke liye poora code rok diya (blocking).

🔹 Asynchronous Behaviour

Jab heavy kaam (jaise timer, API call) hota hai to JS usko **Web API** ko de deti hai aur khud free rehti hai.

```
console.log(10);


setTimeout(() => {
  console.log(20);
}, 2000);


console.log(30);
```

🖥️ Output:

```
10
30
20
```

⚡ Kyu aisa hua?

1. `console.log(10)` → sabse pehle chal gaya.
2. `setTimeout()` → Browser ke Web API ko de diya.
3. `console.log(30)` → bina wait kiye turant chal gaya.
4. 2 sec ke baad Web API ne callback bheja → Event loop ne dekha stack khali hai → fir `20` print hua.

🔑 Key Points

- `setTimeout`, `Event listeners`, `HTTP requests` **JavaScript ke part nahi** → ye **Browser Web APIs** handle karti hai.
- Async kaam hone ke baad, callback **Event Loop** ke through execute hota hai.
- Isi wajah se JS **non-blocking** hai aur smooth lagti hai.

📌 **One Line Recap:**
Callback hell me code unreadable ho jata hai.
JS ek single-threaded language hai lekin Web APIs ke wajah se asynchronous kaam kar leti hai bina block kiye.

📘 Lecture 26: Async Task Working & Web APIs

🔹 JavaScript Nature

- JS ek **single-threaded language** hai → ek time pe sirf ek hi kaam kar sakti hai.
- Lekin JS ke paas **asynchronous power** bhi hoti hai → jaise `setTimeout`, `setInterval`, `fetch`, events, etc.
- Ye sab JS ka hissa nahi hote, balki **Web APIs (browser ke features)** handle karte hain.

🔹 Call Stack

- Call stack ek **execution box** hai jisme JS ke functions line by line chalte hain.
- Jab ek function khatam hota hai → stack se remove ho jata hai (pop).

Example:

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

function greet() {

  console.log("Namaste");

}

greet();

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

Output:

```
Hello

Namaste

End
```

-

🔹 Web APIs

- Browser ke paas kuch extra features hote hain jise JS use karti hai.
- Example:
    - `setTimeout()`
    - `setInterval()`
    - `fetch()`
    - DOM events (`addEventListener`)

Ye sab directly JS ke andar nahi, balki **Web APIs** me bante hain.

🔹 Callback Queue

- Jab Web API ka kaam khatam hota hai (jaise 2 sec ka timer pura), uska result ek **callback function** ke roop me `Callback Queue` me ch चला jata hai.
- Lekin wo turant execute nahi hota → pehle **Call Stack khali hona chahiye**.

🔹 Event Loop

- Event loop ek **chowkidar** hai jo dekhta hai:
    - Agar **Call Stack khali** hai,
    - to **Callback Queue** se function utha kar **Call Stack** me daal deta hai.

🔹 Example 1: setTimeout

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

setTimeout(() => {
  console.log("Callback after 2 sec");
}, 2000);
```

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

Output:

```
Hello

End

Callback after 2 sec
```

👉 Kyun aisa?

- `setTimeout()` Web API me chala gaya.
- 2 sec ke baad callback **Callback Queue** me gaya.
- Lekin pehle "End" execute hua kyunki Call Stack khali nahi tha.

🔹 Example 2: setInterval

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

setInterval(() => {

  console.log("I am fast");

}, 2000);

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

Output:

```
Start

End

I am fast   (after 2 sec)

I am fast   (after 4 sec)

I am fast   (after 6 sec) ...
```

👉 `setInterval()` bar-bar ek fixed interval pe callback queue me bhejta hai.

🎯 Event Listener & Web APIs

- Humare paas 4 buttons hai aur sab par `click` event listener lagaya hai.
- Lekin JS **single-threaded** hai → ek time pe ek hi kaam sun sakti hai.
- Matlab JS direct sabhi buttons ko ek saath **nahi sun paati**.
- Isliye **event listener ka kaam Web APIs sambhalti hain** → wo sabhi buttons pe nazar rakhti hain.
- Jab button click hota hai → Web API callback ko JS ke paas bhej deti hai (Event Loop ke through).

👉 Simple bolna ho to: **JS ek banda hai, Web APIs uske helpers hai jo saare events sunke usko bataati hain.**

🔹 Important Points

1. `setTimeout` / `setInterval` **JS ka part nahi hai**, ye **Web APIs** ke through aate hain.
2. Callback hamesha **tabhi chalega jab call stack khali hoga**, chahe time pura ho chuka ho.
3. Isi wajah se JS me **race condition** nahi hoti → har device pe ek hi output aata hai.
4. Event listeners (click, input, etc.) bhi Web APIs ke part hain.

🔑 Short Recap

- **Call Stack** → Functions execute hote hain.
- **Web APIs** → setTimeout, fetch, DOM, etc. handle karte hain.
- **Callback Queue** → Callback functions rakhte hain.
- **Event Loop** → Callback Queue se function uthakar Call Stack me dalta hai.

👉 Bas itna yaad rakho:
JavaScript ke asynchronous tasks hamesha **Web APIs → Callback Queue → Event Loop → Call Stack** ke flow se chalte hain.

🚀 Lecture 27 – Promises in JavaScript

---

1. Promise Kya Hai?

- Promise ek **object** hai jo batata hai ki koi async kaam (jaise data fetch karna) aakhir me **success** hoga ya **fail**.
- Matlab → "Abhi pending hai, baad me result milega."

---

2. Fetch API (real-life use)

```
fetch("url");
```

👉 Ye ek HTTP request bhejta hai aur hamesha ek **Promise** return karta hai.

---

3. Promise ka Structure

```
let promise = new Promise((resolve, reject) => {

   // async kaam

   if (success) {

      resolve(value);   // ✅ kaam sahi hua

   } else {

      reject(error);    // ❌ kaam fail hua

   }

});
```

- **resolve** → kaam sahi hua toh call hota hai.
- **reject** → kaam fail hua toh call hota hai.

---

4. Promise States

1. **pending** → shuru me, abhi result nhi aaya.
2. **fulfilled** → kaam success, resolve call hua.
3. **rejected** → kaam fail, reject call hua.

---

5. Promise Handling

```
const promise = fetch("url");


promise.then((response) => {
   return response.json(); // ye bhi ek async promise return karta hai
}).then((data) => {
   console.log(data); // yaha actual data milta hai
}).catch((error) => {
   console.log("Error:", error);
});
```

👉 `then()` → success result ko handle karta hai.
👉 `catch()` → error handle karta hai.

---

6. Promise Chaining

Promise chaining ka matlab hai ek ke baad ek async kaam karna **pipe line jaisa**.

Example:

```
fetch("url")
   .then((response) => response.json())  // pehla async kaam
   .then((data) => console.log(data))    // dusra async kaam
   .catch((error) => console.log(error));// agar fail hua
```

👉 Isse code clean aur readable ho jata hai.

⚡ Tips (Desi Tadka)

- Promise = "Aadat se majboor dost" → abhi reply nhi dega, lekin future me pakka batayega (ya success ya fail).
- Always **use `.catch()`** → warna error aayega aur crash ho sakta hai.
- **Chaining best hai** → callback hell se bachne ka asaan tarika.

📌 Summary

- **Promise** = async kaam ka result (success ya fail).
- States → **pending**, **fulfilled**, **rejected**.
- Handle karne ke liye → `.then()` & `.catch()`.
- **Promise chaining** → multiple async kaam ko clean tareeke se likhne ka tarika.
- `fetch()` hamesha promise return karta hai.

🔴 Callback Hell ki Problem (Promise ki Need)

Callback hell tab hota hai jab **nested callback functions** ek ke andar ek ghus jaate hain. Code aisa lagta hai jaise **ladder ya pyramid of doom** ban gaya ho.

👉 Example: Pizza Delivery with Callback Hell

```
placeOrder(() => {
  preparingOrder(() => {
```

```
    pickupOrder(() => {
      deliverOrder(() => {
        console.log("🎉 Enjoy your pizza!");
      });
    });
  });
});
```

⚠️ Problems:

1. **Readability khatam** – Code dekh kar hi dar lagta hai. Ek ke andar ek nested callback.
2. **Debugging mushkil** – Agar error aaye to samajhna tough hota hai ki kaunse level pe dikkat hai.
3. **Maintain karna mushkil** – Agar beech me ek aur step add karna ho, to pura nesting todna padta hai.
4. **Error handling kharab** – Har ek callback me alag se error check karna padta hai.

🟢 Promise ka Solution

Promises ne yehi problem solve kiya. Har function ek **Promise object** return karega, jisme success hoga to `resolve()` chalega aur error hoga to `reject()`.
Isse hum **chaining** kar sakte hain, matlab ek ke baad ek straight line me likh sakte hain.

👉 Example: Pizza Delivery with Promises

```
function placeOrder() {
  return new Promise((resolve) => {
    console.log("Talking with Domino's...");
    setTimeout(() => {
      console.log("✅ Order placed successfully");
      resolve();
    }, 2000);
```

```
  });
}

function preparingOrder() {
  return new Promise((resolve) => {
    console.log("👨‍🍳 Pizza preparation started...");
    setTimeout(() => {
      console.log("✅ Pizza preparation done!");
      resolve();
    }, 3000);
  });
}

function pickupOrder() {
  return new Promise((resolve) => {
    console.log("🛵 Reaching restaurant to pick up the order...");
    setTimeout(() => {
      console.log("✅ Order picked up by delivery boy");
      resolve();
    }, 2000);
  });
}

function deliverOrder() {
  return new Promise((resolve) => {
    console.log("🚚 Delivery boy is on the way...");
    setTimeout(() => {
```

```
      console.log("✅ Order delivered successfully!");
      resolve();
    }, 3000);
  });
}

// 👉 Promise chaining
placeOrder()
  .then(() => preparingOrder())
  .then(() => pickupOrder())
  .then(() => deliverOrder())
  .then(() => console.log("🎉 Enjoy your Pizza!"))
  .catch((err) => console.log("❌ Something went wrong:", err));
```

✅ Why Promises are Better:

1. **Readable Code** → Nested se bach gaye, ek straight line me chaining ho gayi.
2. **Easy Debugging** → Agar error aaye to `catch()` se easily pakad sakte ho.
3. **Maintain karna easy** → Beech me ek step add karna ho to sirf ek `.then()` jodna hai.
4. **Error Handling ek jaga** → Saare errors ek hi `catch()` me aa jaate hain.

👉 Simple shabdo me:

- Callback hell = **"Chipka chipki wala code"**
- Promise = **Chipk Seedha aur Clean Code “**

🚀 Lecture 27 – Async & Await in JavaScript

Async/await ka kaam hai **asynchronous code ko aise likhna jaise synchronous ho**.
Matlab aapko `.then()` ka lamba chain nahi banana padega → bas ek line me `await` laga do.

🔑 1. `async function`

- Agar kisi function ke aage `async` likh diya → wo **hamesha promise return karega**.
- Agar normal value return kare → JS usko bhi `Promise.resolve()` me wrap kar deta hai.

👉 Example:

```
async function myFun() {
  return "Hello!";
}

myFun().then((val) => console.log(val));
```

Output:

```
Hello!
```

💡 **Tip:** `async` bina `await` ke adhura hai – dono dost hai 😅

🔑 2. `await` keyword

- Sirf `async` function ke andar use hota hai.
- Ye **execution ko pause** kar deta hai jab tak promise resolve/reject nahi ho jata.
- Resolve hone ke baad uski value return hoti hai.

👉 Example:

```
const p1 = new Promise((resolve) => {

  setTimeout(() => resolve("Data loaded"), 3000);

});


async function getData() {

  console.log("Fetching...");

  const result = await p1; // yahan ruk jaayega jab tak p1 resolve
nahi hota

  console.log(result);

}


getData();
```

Output:

```
Fetching...

(3 sec baad)

Data loaded
```

🔄 3. Callback Hell vs Promise vs Async/Await

👉 Callback Hell:

```
placeOrders(cost, (order) => {

  prepareOrders(order, (food) => {
```

```
    pickupOrders(food, (location) => {
      deliverOrders(location);
    });
  });
});
```

👉 Promise Chain:

```
placeOrders(cost)
  .then(order => prepareOrders(order))
  .then(food => pickupOrders(food))
  .then(location => deliverOrders(location))
  .catch(err => console.log(err));
```

👉 Async/Await (cleanest):

```
async function greet() {
  const order = await placeOrders(cost);
  const food = await prepareOrders(order);
  const location = await pickupOrders(food);
  await deliverOrders(location);
  console.log("🎉 Pizza delivered!");
}
greet();
```

💡 **Tip:** `await` = `.then()` ka shortcut jo code ko readable banata hai.

🧪 4. Real Examples

Example 1: Sequential execution

```
const p1 = new Promise(resolve => {
  setTimeout(() => resolve("First promise resolved"), 8000);
});

const p2 = new Promise(resolve => {
  setTimeout(() => resolve("Second promise resolved"), 5000);
});

async function greet() {
  const data1 = await p1;
  console.log(data1);     // waits 8 sec

  const data2 = await p2;
  console.log(data2);     // already resolved, prints immediately
}
greet();
```

Output (after 8 sec):

```
First promise resolved
Second promise resolved
```

Example 2: Function ke andar naya promise

```
function test1() {
  return new Promise(resolve => {
    setTimeout(() => resolve("First resolved"), 5000);
  });
```

```
}

function test2() {
  return new Promise(resolve => {
    setTimeout(() => resolve("Second resolved"), 5000);
  });
}

async function greet() {
  const data1 = await test1();  // waits 5 sec
  console.log(data1);

  const data2 = await test2();  // waits another 5 sec
  console.log(data2);
}
greet();
```

Output (10 sec total):

```
First resolved
Second resolved
```

💡 **Tip:** Agar sequential me call karte ho to total wait time jyaada ho jaata hai.

Example 3: Same Promise dobara await karna

```
const p1 = new Promise(resolve => {
  setTimeout(() => resolve("Hello Everyone"), 5000);
});
```

```
async function greet() {
  const data1 = await p1;
  console.log(data1);    // waits 5 sec

  const data2 = await p1;
  console.log(data2);    // instantly print karega, kyunki pehle se resolved hai
}
greet();
```

Output:

```
Hello Everyone
Hello Everyone
```

Example 4: Parallel Execution

Agar do promises ek sath chahiye ho → use **Promise.all()**

```
async function greet() {
  const [data1, data2] = await Promise.all([p1, p2]);
  console.log(data1, data2);
}
```

💡 **Tip:** Ye dono promises ko parallel run karega, wait time kam ho jayega.

📝 Summary

- `async` → function ko promise return karwata hai.
- `await` → promise resolve hone tak rukta hai.
- Async/Await = **Readable + Clean code** (callback hell aur `.then()` se bachata hai).
- Sequential execution me zyada time lagta hai, parallel ke liye `Promise.all()` best hai.
- Async/Await ka use karke **real projects me APIs call karna easy ho jata hai**.

📘 Lecture 30 – Try, Catch aur Promise.all in JS

🔹 Try & Catch (Error Handling)

- Kabhi bhi async function me error aa sakta hai (network fail, API down, etc).
- **try{}** block me code likhte hain. Agar error aaya → seedha **catch{}** me chala jaata hai.

👉 Example:

```
async function greet() {
  try {
    const data1 = await test1();
    console.log(data1);

    const data2 = await test2();
    console.log(data2);
  } catch (error) {
    console.log("Error aya:", error);
  }
}

greet();
```

✅ **Working**

1. Pehle test1() resolve hoga.
2. Fir test2() chalega.

3. Agar kahin error hua → sidha catch block.

🔹 Problem → Sequential Execution

- Upar ke code me test1() khatam hone ke baad hi test2() start hota hai.
- Matlab **time zyada lagta hai**, kyunki dono ek ek karke execute ho rahe hain.

🔹 Solution → Promise.all()

- Agar tum chahte ho ki **dono ek sath (parallel) chalein** → `Promise.all()` use karo.
- Ye **dono ko ek sath start karega** aur wait karega jab tak dono complete na ho jaayein.

👉 Example:

```
async function greet() {
  try {
    console.log("Hello I greet you");

    const [data1, data2] = await Promise.all([test1(), test2()]);
    console.log(data1);
    console.log(data2);
  } catch (error) {
    console.log("Error aya:", error);
  }
}

greet();
```

✅ **Working**

1. test1() aur test2() **ek sath start hote hain**.
2. Jo promise zyada time lega, uske complete hone ke baad hi result milega.
3. Matlab total time = **max(test1, test2)** ⏱️

⚡ Quick Tips

- **try/catch** hamesha async functions me lagao → warna error crash kar dega.
- Agar **multiple promises** ek sath chalana ho → `Promise.all()`.
- Agar ek promise fail hua → `Promise.all()` sidha error dega (catch me chala jayega).

📝 Summary

1. **try/catch** → async functions me error handle karne ke liye.
2. Normal `await` → sequential (ek ke baad ek).
3. `Promise.all()` → parallel execution (dono ek sath).
4. Time taken = **sabse bada wala promise** jitna time lega.

👉 Bhai, chaahe exam ho ya interview, bas ye 3 points yaad rakhna →
Try/Catch = error control | await = sequence | Promise.all = parallel 🚀

`try/catch` aur `Promise.all()` (Real Life Example)

1. `try` aur `catch` kya hai?

Soch le tu ghar pe **online pizza order** kar raha hai.

- Tu order karta hai → `try` block me code run hota hai.
- Agar pizza aa gaya → masti hai 🍕.
- Agar delivery wale ka tyre puncture ho gaya ya network down ho gaya → seedha `catch` block me error aa jayega.

👉 Code example:

```
async function orderPizza() {
  try {
    let pizza = await getPizza();   // pizza order karna
    console.log(pizza);

    let coke = await getCoke();     // coke order karna
    console.log(coke);
  } catch (error) {
    console.log("Kuch gadbad ho gayi:", error);
  }
}
```

Samajh le:

- Pehle pizza order hoga.
- Pizza aane ke baad hi coke order hoga.
- Agar kahin error aaya (jaise coke khatam ho gaya), to `catch` block chalega.

---

2. Problem: Ye sequential hai (ek ke baad ek)

Soch le:

- Pizza banane me 10 min lage.
- Coke lane me 5 min lage.
- Total time = 10 + 5 = **15 min** ⏱️

Matlab, tu unnecessarily wait kar raha hai.

---

3. Solution: `Promise.all()` (Parallel Execution)

Ab soch le tu **Domino's app me ek sath pizza + coke order** karta hai.

- Dono order ek sath nikal gaye 🚴
- Ab time = **jo bada time lega wahi final time hoga**.
    - Pizza (10 min), Coke (5 min).
    - Matlab total = **10 min** (na ki 15 min).

👉 Code example:

```
async function orderPizzaAndCoke() {
  try {
    console.log("Order placed: Pizza + Coke");

    const [pizza, coke] = await Promise.all([getPizza(), getCoke()]);
    console.log(pizza);
    console.log(coke);
  } catch (error) {
    console.log("Kuch gadbad ho gayi:", error);
  }
}
```

---

4. Key Samajhne Wali Baat

- **`try/catch`** → galti/error pakadne ke liye.
- **`await` ek ke baad ek** → sequential (zyada time lega).
- **`Promise.all()`** → dono ek sath chalega (time bachega).

⚡ Tips Yaad Rakho

- Agar kaam ek dusre pe depend karte hain → normal `await` use karo.
  👉 (jaise: pehle base bana, fir topping dal).

- Agar kaam independent hain → `Promise.all()` use karo.
  👉 (jaise: pizza ban raha hai aur coke fridge se aa rahi hai, dono alag cheez).

📝 Short Summary

- `try/catch` = Error handling (safe code).

- Normal `await` = Ek ke baad ek (slow).

- `Promise.all()` = Ek sath (fast).
- Time taken = sabse **bada wala promise** jitna time lega.

🚀 JavaScript Closure — The Ultimate Simple Guide (with Memory Magic!)

🔥 Closure Kya Hai?

Closure matlab:

🔹 **Ek inner function apne outer function ke variables ko yaad rakhta hai, chahe outer function ka kaam khatam ho chuka ho.**

🎯 Real Life Example:

Mummy kitchen mein khana bana rahi hai (outer function).
Tumne bola:

"Mummy, jab khana ready ho jaye, mujhe bula dena." (inner function)

Khana ready hone ke baad, mummy ne bula liya — kyunki unhone tumhara kehna yaad rakha!

Ye yaad rakhna = Closure!

💻 Simple JavaScript Code:

```
function mummy() {
  let khana = "Rajma Chawal";

  function beta() {
    console.log("Mujhe mila: " + khana);
  }

  return beta;
}

let bulao = mummy();  // Outer function khatam
```

```
bulao();              // Inner function ab bhi outer ke variable ko
yaad rakhta hai
```

Output:

```
Mujhe mila: Rajma Chawal
```

🧠 Memory Magic Behind Closure:

- Jab `mummy()` function chalta hai, to `khana` variable **stack memory** mein banta hai.
- Normally function khatam hone par ye variable delete ho jata.
- Par kyunki `beta()` function use karta hai, JS engine is variable ko **stack se heap memory me shift kar deta hai** taaki inner function use kar sake.
- Is process ko **Closure** kehte hain.

🔧 Closure Kab Use Hota Hai?

1. **Private Data Chhupane ke liye**

```
function counter() {
  let count = 0; // Private variable

  return function() {
    count++;
    console.log(count);
  };
}

const c = counter();
c(); // 1
c(); // 2
```

2. **Callbacks aur Asynchronous Code me**

```
function greet(name) {
  setTimeout(() => {
    console.log("Hello " + name);
  }, 1000);
}

greet("Amit"); // 1 second baad "Hello Amit"
```

⚡ 2 Line Me Closure Ka Summary:

- Inner function apne outer function ke variables ko yaad rakhta hai.
- Outer function ke khatam hone ke baad bhi wo variables accessible rehte hain.

💡 Pro Tip:

Closure powerful hai, lekin samajh ke use karo!
Galat use se memory leak ho sakta hai.

✍️ Bonus - Best Analogy:

Office band ho gaya (outer function finish),
par employee ke paas chabi (inner function) hai,
toh wo ab bhi office ke documents (variables) tak pahuch sakta hai.

✨ Made with ❤️ and Hardwork by **Harshal Chauhan** 🚀🔥

NEXUS+ COURSE

MENTOR = ROHIT NEGI