🧠 LECTURE 5 — MEMORY IN JAVASCRIPT

📘 **JavaScript Internals — Memory • Stack • Heap**
🧠 *First Principles • Visualization • Zero Confusion*

⬛ FIRST PRINCIPLE — "MEMORY HOTI KYA HAI?"

🔷 Simple Soch (Real-Life)

Socho tumhara **mobile phone**:

- Contacts kahin save
- Photos kahin save
- Apps kahin save

👉 Ye sab **memory** me stored hota hai.

🟨 **Same cheez JavaScript me hoti hai**

Variable = Naam
Memory = Jagah jahan value rakhi jaati hai

⬛ PRIMITIVE vs NON-PRIMITIVE (FIRST PRINCIPLE)

🔷 Primitive Data Types

- **Immutable** (original value change nahi hoti)
- Change par **nayi memory create hoti hai**

Includes:

Number, String, Boolean, null, undefined, Symbol, BigInt

🔷 Non-Primitive Data Types

- **Mutable** (same memory me value ka change)
- **Reference ke through kaam karte hain**

Includes:

Array, Object, Function

⬛ REAL-LIFE EXAMPLE — REFERENCE CONCEPT

```
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"}
```

🟥 **Observation**

Dono variables **same memory address** ko point kar rahe hain
Isliye ek change → dono me reflect

🟨 **KEY IDEA**

Non-Primitive = **Address copy hota hai, value nahi**

⬛⬛⬛ STACK vs HEAP MEMORY ⬛⬛⬛

① STACK MEMORY

🔷 Used For

- Primitive data types

🔷 Nature

- Small size
- Fast access
- Call by Value

```
let a = 10;

let b = a;

b = 50;
```

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

console.log(b); // 50
```

🟨 **Explanation**

b ko a ki **copy** mili
Change sirf b me hua

---

② HEAP MEMORY

🔷 Used For

- Non-Primitive data types

🔷 Nature

- Large size
- Flexible
- Call by Reference

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

let obj2 = obj1;


obj2.id = 30;


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

🟨 **Explanation**

obj1 & obj2 → same memory address
Isliye change dono me

⬛ CALL BY VALUE vs CALL BY REFERENCE

Concept	Primitive	Non-Primitive
Memory	Stack	Heap
Copy	Value copy	Address copy
Change effect	Original safe	Original affected

⬛ WHY PRIMITIVES NOT STORED IN HEAP?

🔷 First-Principle Reason

- Stack = fast & small
- Heap = large & flexible

🟨 Logic

Primitive values chhoti hoti hain
Isliye **Stack me efficient**

Non-primitives bade/dynamic hote hain
Isliye **Heap me stored**

⬛ CONST BEHAVIOR — TRICKY BUT IMPORTANT

🔷 Const with Primitive

```
const num = 10;

num = 20; // ❌ Error
```

🟥 Reason

Const primitive = value constant

🔷 Const with Non-Primitive

```
const obj = { id: 10, balance: 234 };

obj.id = 20;      // ✅ Allowed

console.log(obj);
```

🟨 Reason

Const object = **reference constant**
Properties change ho sakti hain
Reference change ❌

⬛ WHY PRIMITIVE DATA TYPES ARE IMMUTABLE?

```
let a = 10;

let c = a;

c = 50;
```

🧠 Behind the scenes

- `c = 50`
- JS ne **nayi memory location** create ki
- `a` purani memory pe hi raha

🟨 Special Case

Agar nayi value zyada memory leti hai
(jaise long string)
JS **new memory allocate karta hai**

⬛ C++ / JAVA vs JAVASCRIPT (MEMORY DIFFERENCE)

Language	Type System
C++ / Java	Fixed types

JavaScript Dynamic types

🟨 **Isliye**

JavaScript ka memory management zyada flexible hota hai

⬛ WHY DO WE NEED MEMORY ADDRESS?

🔷 First Principle

Memory **byte-addressable** hoti hai
👉 Har byte ka **unique address**

🟦 Scenario 1 — Fast Access

- Without address → line by line search ❌
- With address → direct access ✅

🟦 Scenario 2 — Duplicate Values

- 78 do jagah stored
- Address ke bina confusion
- Address se **unique identity**

🟨 Conclusion

Addressing system = fast + accurate access

⬛FINAL POWER SUMMARY ⬛

🟨 ONE-GLANCE REVISION

- Primitive → Stack → Immutable → Value copy
- Non-Primitive → Heap → Mutable → Reference copy
- Stack = Fast, Heap = Flexible
- Const primitive → change ❌
- Const object → properties change ✅
- Address = Fast access + No confusion

⬛⬛⬛ FINAL THOUGHT ⬛⬛⬛

🧠 **JavaScript memory samajh li →**
bugs, confusion, unexpected output sab kam