JavaScript Variables: Behind The Scenes

1. JavaScript Memory Phases (Hoisting in Action)

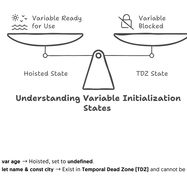
Example Code:

```
console.log(age);   // undefined
var age = 25;

console.log(name);  // ReferenceError
let name = "John";

console.log(city);  // ReferenceError
const city = "New York";
```

Step 1: Memory Creation Phase (Before Execution)

- **JavaScript scans the code first and sets up memory for variables.** - **Memory Table Before Execution:**

```
┌──────────┬──────────┬──────────┐
| Variable | Value    | State    |
├──────────┼──────────┼──────────┤
| age      | undefined| Hoisted  |
| name     |  -       | TDZ (Blocked) |
└──────────┴──────────┴──────────┘
```



- **var age** → Hoisted, set to **undefined**.
- **let name & const city** → Exist in **Temporal Dead Zone (TDZ)** and cannot be used.

Step 2: Execution Phase (Line by Line Execution)

- Now, JavaScript starts executing line by line:
- **console.log(age);** ✅ → Prints **undefined** because **var** is hoisted.🔼
- **console.log(name);** ❌ → **ReferenceError! let** is still in TDZ.🔼 **console.log(city);** ❌ → **ReferenceError! const** is also in TDZ.🔼 **var age = 25;** ✅ → Updates **age** to **25**.🔼
- **let name = "John";** ✅ → Removes **name** from TDZ and assigns **"John"**.🔼 **const city = "New York";** ✅ → Removes **city** from TDZ and assigns **"New York"**.

Final Memory Table (After Execution)

```
┌──────────┬──────────┬──────────┐
| Variable | Value    | State    |
├──────────┼──────────┼──────────┤
| age      | 25       | Assigned |
| name     | "John"   | Assigned |
└──────────┴──────────┴──────────┘
```

2. How JavaScript Stores Variables (Stack vs Heap Memory)

Example Code:

```
let x = 10;   // Stored in Stack
let y = x;    // Copy of value stored separately

let obj1 = { value: 10 };  // Stored in Heap
let obj2 = obj1;  // Reference to the same object
obj1.value = 20;  // Changes reflect in obj2 as well
```

Step 1: Stack Memory (For Primitive Types)

- Stack memory stores **primitive values directly**.

Stack Memory Before Execution

```
┌──────┐
| x | 10 |
| y | 10 |
└──────┘
```

- When **y = x**, a **copy** of **x** is created. Changing **x** later **does NOT affect y**.

After **x = 20**

```
┌──────┐
| x | 20 |
| y | 10 |
└──────┘
```

✅ **Values are independent!**

Step 2: Heap Memory (For Objects)

- Heap memory stores **objects** and only **stores a reference in Stack**.

Heap & Stack Memory Before Execution

```
Stack:          Heap:
┌──────┐        ┌──────────────┐
| obj1 | <REF_1> | ------> | {value: 10} |
| obj2 | <REF_1> | ------> |              |
└──────┘        └──────────────┘
```

- **obj1** and **obj2** both point to the **same object in Heap**.

After **obj1.value = 20;**

```
Stack:          Heap:
┌──────┐        ┌──────────────┐
| obj1 | <REF_1> | ------> | {value: 20} |
| obj2 | <REF_1> | ------> |              |
└──────┘        └──────────────┘
```

✅ **Both obj1 and obj2 are updated!**

3. Block Scope & Function Memory Behavior

Example Code:

```
{
    var a = 1;
    let b = 2;
    const c = 3;
}

console.log(a);  // ✅ 1  (var is global)
console.log(b);  // ❌ ReferenceError
console.log(c);  // ❌ ReferenceError
```

Memory Visualization:

Inside Block Scope:

```
Block Scope:
┌──────┐
| b | 2 |
| c | 3 |
└──────┘
```

🚀 **Variables b and c exist only inside the block!**

Global Memory After Block Ends:

```
Global Scope:
┌──────┐
| a | 1 |
└──────┘
```

✅ **Only a survives because var is function-scoped!**

4. Function Calls & Memory Allocation

Example Code:

```
function createUser(name) {
    let userId = Math.random();
    const timestamp = Date.now();
    var isActive = true;

    return { userId, name, timestamp, isActive };
}

let user1 = createUser("John");
let user2 = createUser("Jane");
```

Step 1: First Call (createUser("John"))

```
Function Call 1:
┌─────────────┬──────────────┐
| userId      | "id_12345"   |
| timestamp   | 1612345678   |
| isActive    | true         |
└─────────────┴──────────────┘
```

Step 2: Second Call (createUser("Jane"))

```
Function Call 2:
┌─────────────┬──────────────┐
| userId      | "id_67890"   |
| timestamp   | 1612345680   |
| isActive    | true         |
└─────────────┴──────────────┘
```

🚀 **Each function call creates a new memory space!**

Final Takeaways

✅ **Variables behave differently based on var, let, and const.** ✅ **Primitive values are stored in Stack (copied when assigned).** ✅ **Objects & Arrays are stored in Heap (shared reference).** ✅ **Function calls create new memory space each time.** ✅ **Block scope (let & const) clears memory after execution.**

Would you like more examples or animations to explain these concepts further? 🚀