

Perfect. Continuing systematically, NotebookLM-ready, with NOTES and CODE strictly separated.

TOPIC: Scope & Hoisting in JavaScript

(Execution Context • Memory Allocation • `var` / `let` / `const` • TDZ)

NOTE 1: What “scope” means internally

- **Scope** = where a variable is accessible
- Scope is determined **at compile time**, not runtime
- JavaScript uses **lexical (static) scoping**
- Scope is tied to **code structure**, not call order

NOTE 2: Types of scope in JavaScript

1. Global Scope
2. Function Scope
3. Block Scope (`{ }` with `let` / `const`)
4. Lexical Scope (scope chain concept)

CODE 1: Global scope

```
js

let x = 10;

function show() {
  console.log(x); // accessible
}

show();
console.log(x); // accessible
```

NOTE 3: Function scope

- Variables declared inside a function:
 - Exist only within that function
- `var` is **function-scoped**
- `let` / `const` are **block-scoped**

CODE 2: Function scope example

```
js

function test() {
  var a = 10;
  let b = 20;
}
```

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

NOTE 4: Block scope (`let` / `const`)

- Block = `{ }` inside:
 - `if`
 - `for`
 - `while`
- `let` and `const` live inside the block only
- `var` ignores block scope

CODE 3: Block scope difference

```
js

if (true) {
  var x = 10;
  let y = 20;
  const z = 30;
}

console.log(x); // 10
console.log(y); // ❌ ReferenceError
console.log(z); // ❌ ReferenceError
```

NOTE 5: Lexical scope (VERY IMPORTANT)

- Inner functions can access outer variables
- Outer functions CANNOT access inner variables
- Scope chain is fixed at **definition time**

CODE 4: Lexical scope example

```
js

function outer() {
  let a = 10;

  function inner() {
    console.log(a); // accessible
  }

  inner();
}
```

```
outer();
```

NOTE 6: What is hoisting (real meaning)

- Hoisting is **not moving code**
- JavaScript does a **memory allocation phase** before execution
- Declarations are registered before execution starts

NOTE 7: Two phases of execution context

1. Creation Phase

- Memory allocated
- Variables & functions registered

2. Execution Phase

- Code runs line by line

CODE 5: Hoisting with `var`

```
js

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

Why?

- `var a` → allocated in memory with value `undefined`
- Assignment happens later

NOTE 8: `let` and `const` hoisting (IMPORTANT)

- `let` and `const` ARE hoisted
- But NOT initialized
- Access before initialization → **ReferenceError**

This period is called **Temporal Dead Zone (TDZ)**

CODE 6: `let` hoisting behavior

```
js

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

NOTE 9: Temporal Dead Zone (TDZ)

- Time between:
 - Scope entry
 - Variable initialization

- Accessing variable in TDZ causes error
- Prevents unsafe access

CODE 7: TDZ illustration

```
js

{
  // TDZ starts here
  // console.log(x); ❌ ReferenceError
  let x = 10;
  console.log(x); // 10
}
```

NOTE 10: `const` rules (deep)

- Must be initialized at declaration
- Cannot be reassigned
- Object contents CAN be mutated

CODE 8: `const` behaviors

```
js

const a = 10;
// a = 20; ❌ TypeError

const obj = { x: 1 };
obj.x = 2; // allowed
```

NOTE 11: Hoisting of functions

Function Declarations

- Fully hoisted
- Stored with function body

Function Expressions

- Hoisting depends on variable type

CODE 9: Function hoisting

```
js

sayHi(); // works

function sayHi() {
```

```
console.log("Hi");
}
```

js

```
sayHello(); // ❌ ReferenceError
```

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

NOTE 12: var vs let vs const (INTERVIEW TABLE)

Feature	var	let	const
Scope	Function	Block	Block
Hoisted	Yes	Yes	Yes
Initialized	undefined	No (TDZ)	No (TDZ)
Reassign	Yes	Yes	No
Redeclare	Yes	No	No

NOTE 13: Global scope & window binding

- `var` creates property on global object
- `let` / `const` do NOT

CODE 10: Global object behavior

js

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

```
window.a; // 10
window.b; // undefined
```

NOTE 14: Common interview traps

- `let` is hoisted but uninitialized
- TDZ exists until declaration line
- `var` ignores block scope
- Functions are hoisted differently

- Scope is lexical, not dynamic

FINAL MENTAL MODEL

- Scope = access rules decided by code structure
- Hoisting = memory allocation phase
- `var` → hoisted + initialized to `undefined`
- `let` / `const` → hoisted + TDZ
- Functions create new scopes
- Lexical scope builds scope chain

STOP HERE

Next recommended continuation:

- 1 Closures (deep, memory-level + real use)
- 2 Array Methods (`map` , `filter` , `reduce`)
- 3 Objects (creation, access, reference behavior)

Reply with **next** or the **number** you want to continue.