Below is a **clean, simplified, internally explained, and NoteLM-friendly version** of `var` **in JavaScript**, written in the **same depth + format** you've been following.
I've corrected language, removed redundancy, and added **internal behavior explanations** where needed (important for exams + interviews).

🔹 `var` Keyword in JavaScript

The `var` **keyword** is used to declare variables in JavaScript.
Before **ES6 (2015)**, `var` was the **only way** to declare variables.
After ES6, `let` and `const` were introduced to fix many problems of `var`.

📌 Modern JavaScript Recommendation

- Prefer `let` and `const`
- Use `var` **only for legacy browser support or old codebases**

📌 What is a Variable?

A **variable** is a container used to store data in memory.

JavaScript variables can store:

- Number
- String
- Boolean
- Object
- Array
- Function

📘 Syntax of `var`

```
js

var identifier;
var identifier = value;
```

- `identifier` must follow valid variable naming rules
- Initialization is optional

🧪 Basic Example

```
html

<script>
 var num = 30;
 var text = "Hello World";
```

```
  var flag = true;

  console.log(num);
  console.log(text);
  console.log(flag);
</script>
```

Output

```
arduino

30
Hello World
true
```

🌍 Scope of `var`

A variable declared with `var` has **function scope**, not block scope.

Types of Scope with `var`

1. Global Scope
2. Function Scope

❌ `var` does **NOT** have block scope

🔹 Function Scope (Very Important)

If a variable is declared **anywhere inside a function**, it is accessible **throughout the function**, even outside blocks like `if`, `for`, `{}`.

Example: Function Scope

```
html

<script>
function demo() {
  var x = 10;

  {
    var y = 20;
  }

  console.log(x); // 10
  console.log(y); // 20
}
```

```
demo();
</script>
```

📌 Reason (Internal Explanation)

JavaScript ignores blocks `{}` for `var`.

Only functions create scope boundaries for `var`.

❌ Contrast with `let`

```
js

{
  let a = 10;
}
console.log(a); // ❌ ReferenceError
```

🌐 Global Scope with `var`

If `var` is declared **outside any function**, it becomes a **global variable**.

Global `var` variables:

- Attach to the `window` object (in browsers)
- Can be accessed anywhere

Example

```
html

<script>
var num1 = 10;

function sum(num2) {
  console.log(num1 + num2);
  console.log(window.num1 + num2);
}

sum(20);
</script>
```

Output

```
30
30
```

📌 `var` pollutes the global scope → **dangerous**

🚀 Hoisting Behavior of `var`

Variables declared with `var` are **hoisted** to the top of their scope.

What is Hoisting?

> JavaScript moves variable declarations to the top of the scope **before execution**

⚠️ Only the **declaration** is hoisted, **not the value**

Example: Hoisting

```
html

<script>
function test() {
  a = 98;
  console.log(a);
  var a;
}
test();
</script>
```

Output

```
98
```

Internally, JavaScript reads it as:

```
js

function test() {
  var a;
  a = 98;
  console.log(a);
}
```

⚠️ Dangerous Example

```js
console.log(x);
var x = 10;
```

Output:

```javascript
undefined
```

❌ This causes bugs

✔️ `let` and `const` fix this using **Temporal Dead Zone (TDZ)**

🔁 Redeclaration with `var`

`var` allows **multiple redeclarations** of the same variable.

Example

```html
<script>
var a = 10;
var a = 20;
var a;

console.log(a);
</script>
```

Output

```
20
```

📌 Redeclaration does **not reset value**

📌 This is **not allowed** with `let` or `const`

🔍 Shadowing with `var`

```html
<script>
var num = 10;
```

```
function show() {
  var num = 20;
  console.log(num); // 20
}

show();
console.log(num); // 10
</script>
```

✔️ Inner `var` hides outer `var`

✔️ Both exist in different scopes

🔢 Multiple Declarations in One Statement

```
js

var a, b, c;
a = 10;
b = 20;
c = 30;
```

Or:

```
js

var a = 10, b = 20, c = 30;
```

✔️ Valid

✔️ Common in old JavaScript code

🔁 `var` in Loops (Classic Trap)

```
html

<script>
for (var i = 0; i < 3; i++) {
  console.log(i);
}
console.log(i);
</script>
```

Output

```
0
1
2
3
```

📌 `i` is accessible **outside loop**

📌 This causes bugs in async code

✔️ `let` fixes this by block scoping

📦 Destructuring with `var`

```
html

<script>
var arr = [4, 5, 6];
var [a, b, c] = arr;

console.log(a, b, c);
</script>
```

Output

```
4 5 6
```

✔️ Works

❌ Still function-scoped

⚠️ Major Problems with `var`

Problem	Explanation
No block scope	Leaks outside `{}`
Hoisting bugs	Access before declaration
Global pollution	Attaches to `window`
Redeclaration allowed	Silent overwrites
Loop issues	Breaks async logic

🟢 Comparison Summary

Feature	var	let	const
Scope	Function	Block	Block
Hoisted	Yes	Yes (TDZ)	Yes (TDZ)
Redeclare	✅	❌	❌
Reassign	✅	✅	❌
Global object	Yes	No	No

🎯 Final Takeaway (Exam Ready)

> `var` declares function-scoped variables that are hoisted and can be redeclared, which often leads to bugs. Modern JavaScript prefers `let` and `const`.

If you want, next we can do:

- `let` in same depth
- `const` internals
- `var vs let vs const` traps
- Interview MCQs
- Memory & execution context view

Just say **next** 🚀