1. Explain the differences between var, let, and const with respect to scope and hoisting.

1. Objective

To study and understand the behavior of JavaScript variable declarations (var, let, and const) in terms of **scope** and **hoisting**, and to analyze their practical implications in programming.

2. Theory

a) Scope

• **var** → Function-scoped. Accessible throughout the function in which it is declared. Ignores block scope.
• **let** → Block-scoped. Accessible only within the {} block where it is defined.
• **const** → Block-scoped. Similar to let, but requires initialization and cannot be reassigned.
b) Hoisting

• **var** → Hoisted to the top of its scope and initialized with undefined.
• **let** → Hoisted but not initialized. Exists in the **Temporal Dead Zone (TDZ)** until the declaration line is executed.
• **const** → Same as let (hoisted but in TDZ). Must be initialized at the time of declaration.
3. Code Demonstration

```
// Example with var console.log(a);

// Output: undefined var a = 10;

// Example with let console.log(b);

// ReferenceError let b = 20;

// Example with const console.log(c);

// ReferenceError const c = 30;
```

4. Comparison Table

Feature	var	let	const
Scope	Function-scoped	Block-scoped	Block-scoped
Hoisting	Hoisted, initialized as undefined	Hoisted, TDZ until declared	Hoisted, TDZ until declared
Reassignment	Allowed	Allowed	❌ Not allowed
Initialization	Optional	Optional	✅ Required

5. Conclusion

• var is **legacy** and can cause unexpected behavior due to function scope and hoisting.
• let and const are **modern** and preferred for predictable, block-scoped behavior.
• Best practice: Use **const** by default, and switch to **let** only when reassignment is necessary.

2. Describe the various Control Flow statements in JavaScript, specifically highlighting the difference between for, while, and do-while loops.

Ans- Control Flow Statements in JavaScript

Control flow statements determine the order in which instructions are executed. The main ones include:

• Conditional statements:
  ○ if, else if, else → execute code based on conditions.
  ○ switch → choose execution path based on multiple cases.
• Looping statements:
  ○ for
  ○ while
  ○ do-while
  ○ for...in (iterate over object properties)
  ○ for...of (iterate over iterable values like arrays, strings)
• Jump statements:
  ○ break → exit a loop or switch.
  ○ continue → skip current iteration and move to the next.
  ○ return → exit from a function.

Looping Statements: Key Differences

Loop Type	Syntax Example	Condition Check	Typical Use Case
for	**for (let i = 0; i < 5; i++) { ... }**	**Checked** before **each iteration**	**When the number of iterations is known in advance**
while	**while (i < 5) { ... }**	**Checked** before **each iteration**	**When the number of iterations is not known beforehand, depends on condition**
do-while	**do { ... } while (i < 5);**	**Checked** after **each iteration**	**Ensures the loop runs** at least once**, even if condition is false initially**

• Comparison :-

for loop

• Combines initialization, condition, and increment/decrement in one line.
• Best for **count-controlled loops** (e.g., run 10 times).
• Example:-

```
for (let i = 0; i < 3; i++) {
console.log(i); // 0, 1, 2 }
```

while loop

• Only has a condition; initialization and increment must be handled separately.
• Best for **condition-controlled loops** (e.g., keep reading until file ends).
• Example:
javascript

```
let i = 0;
while (i < 3) {
 console.log(i); // 0, 1, 2
 i++;
}
```

do-while loop

• Executes the block **first**, then checks the condition.
• Guarantees **at least one execution**.
• Example:
javascript

```
let i = 5;
do {
 console.log(i); // prints 5 even though condition fails
 i++;
} while (i < 3);
```

3. Describe the concept of Event Delegation and explain the use of addEventListener.

🎯 Event Delegation

Event Delegation is a **JavaScript design pattern** that allows you to handle events more efficiently by taking advantage of **event bubbling**.

• **Event Bubbling**: When an event (like a click) happens on an element, it "bubbles up" through its ancestors in the DOM tree until it reaches the root (document).
• **Delegation Concept**: Instead of attaching event listeners to multiple child elements, you attach a single listener to a parent element. That listener can detect events triggered by its children.

✅ Why use Event Delegation?

• **Performance**: Fewer event listeners → less memory usage and faster execution.
• **Dynamic Elements**: Works even if child elements are added/removed later (since the parent listener still catches events).
• **Cleaner Code**: Centralized event handling instead of scattered listeners.

🔧 **Example**

```
// Instead of adding a click listener to every button inside a list:

document.getElementById("list")

.addEventListener("click", function(event) {

if (event.target.tagName === "BUTTON") {

alert("Button clicked: " + event.target.innerText); } });
```

Here:

• The listener is attached to the parent
.

- When any inside is clicked, the event bubbles up to the parent.
- We check event.target to see which child triggered it.

🛠️ addEventListener

addEventListener is the method used to register event handlers on DOM elements.

Syntax

```
element.addEventListener(event, handler, options);
```

- **event**: The type of event (e.g., "click", "keydown", "mouseover").
- **handler**: The function to run when the event occurs.
- **options** (optional): Controls behavior (e.g., { once: true } to run only once, { capture: true } to use event capturing instead of bubbling).

Example

```
const button = document.querySelector("#myButton");

button.addEventListener("click", () => { console.log("Button was clicked!"); });
```

🔑 Key Takeaways

- **Event Delegation**: Attach one listener to a parent, handle events for multiple children.
- **addEventListener**: The standard way to bind event handlers in modern JavaScript.
- Together, they make your code **efficient, scalable, and easier to maintain**.