3.1 Explain the Differences Between var, let, and const with respect to Scope and Hoisting

In JavaScript, **var**, **let**, and **const** are used to declare variables. Although they are used for similar purposes, they differ significantly in terms of scope and hoisting, which affects how variables behave in a program.

1. Scope

var
Variables declared using var are function-scoped or global if declared outside a function. They do not follow block scope, meaning they can be accessed outside blocks such as if or for statements.

Example:

```
if (true) {
 var x = 10;
}
console.log(x); // 10 (accessible outside the block)
```

let

Variables declared using let are block-scoped. They are accessible only within the block {} where they are declared, making them safer to use in loops and conditional statements.

Example:

```
if (true) {
 let y = 20;
}
console.log(y); // Error: y is not defined
```

const

Variables declared using const are also block-scoped like let. They must be initialized at the time of declaration and cannot be reassigned later. However, objects and arrays declared with const can still be modified.

Example:

```
const z = 30;
z = 40; // Error
```

2. Hoisting

var
Variables declared with var are hoisted to the top of their scope and initialized with the value undefined.

Example:

```
console.log(a); // undefined
var a = 5;
```

let

let declarations are hoisted but not initialized. Accessing them before declaration results in a ReferenceError due to the Temporal Dead Zone.

Example:

```
console.log(b); // ReferenceError
let b = 10;
```

const

const has the same hoisting behavior as let. It is hoisted but remains in the Temporal Dead Zone until it is initialized.

Example:

```
console.log(c); // ReferenceError
const c = 15;
```

3.2 Describe the Various Control Flow Statements in JavaScript specifically highlighting the difference between for,while , and do-while loops

Control flow statements in JavaScript determine the order in which statements are executed in a program. They allow the program to make decisions, repeat tasks, and jump to different parts of the code based on conditions. Control flow statements are mainly divided into decision-making statements, looping statements, and jump statements.

1. Decision-Making Statements

(a) if Statement
Executes a block of code when a condition is true.

```
if (age >= 18) {
 console.log("Eligible to vote");
}
```

(b) if...else Statement

Executes one block of code if the condition is true and another block if the condition is false.

```
if (marks >= 40) {
 console.log("Pass");
} else {
 console.log("Fail");
}
```

(c) switch Statement

Selects one block of code from multiple options based on a matching case.

```
switch (day) {
 case 1: console.log("Monday"); break;
 case 2: console.log("Tuesday"); break;
 default: console.log("Invalid day");
}
```

2. Looping Statements

Looping statements are used to execute a block of code repeatedly until a specified condition is met.

(a) for Loop

Used when the number of iterations is known in advance. The condition is checked before executing the loop body.

```
for (let i = 1; i <= 5; i++) {
 console.log(i);
}
```

(b) while Loop

Used when the number of iterations is not known beforehand. The condition is checked before each iteration.

```
let i = 1;
while (i <= 5) {
 console.log(i);
 i++;
}
```

(c) do...while Loop

The loop body is executed at least once even if the condition is false. The condition is checked after execution.

```
let i = 1;
do {
 console.log(i);
 i++;
} while (i <= 5);
```

Difference Between for, while, and do...while Loops

for Loop: Condition is checked before execution and is best used when the number of iterations is known. **while Loop:** Condition is checked before execution and is used when iterations are unknown. **do...while Loop:** Condition is checked after execution and guarantees at least one execution.

3. Jump Statements

Jump statements alter the normal flow of execution in a program.

(a) break

Terminates the loop or switch statement immediately.

(b) continue

Skips the current iteration of a loop and continues with the next iteration.

3.3. What is the Document Object Model (DOM)?Explain how to select elements and modfiy their content using innerText and innerHTML

Document Object Model (DOM) The Document Object Model (DOM) is a programming interface that represents an HTML (or XML) document as a tree-like structure of objects. In this structure, every element, attribute, and piece of text in the web page is treated as a node. The DOM allows JavaScript to access, modify, add, or remove HTML elements dynamically, making web pages interactive. In simple words, the DOM acts as a bridge between HTML and JavaScript, enabling JavaScript to control and change the content, structure, and style of a web page after it has loaded.

Selecting Elements in the DOM JavaScript provides several methods to select HTML elements from the DOM:

1. getElementById()
Selects an element using its unique id.
let heading = document.getElementById("title");

2. getElementsByClassName()
Selects elements by their class name.
let items = document.getElementsByClassName("menu");

3. getElementsByTagName()
Selects elements by their tag name.
let paragraphs = document.getElementsByTagName("p");

4. querySelector() and querySelectorAll()
Select elements using CSS selectors.
let box = document.querySelector(".container");
let allBoxes = document.querySelectorAll(".box");

Modifying Content Using innerText and innerHTML
1. innerText Used to get or change only the visible text content of an element. Ignores HTML tags.
Example: <p id="msg">Hello World</p>
document.getElementById("msg").innerText = "Welcome to JavaScript";

2. innerHTML
Used to get or change the HTML content, including tags, inside an element. Allows adding formatted text and new HTML elements.
Example: <div id="content"></div>
document.getElementById("content").innerHTML = "<h2>Hello</h2><p>This is DOM</p>";