LAB ASSIGNMENT UNIT -3

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

Answer:- JavaScript provides three keywords for declaring variables:

var, **let**, and **const**. Although they serve the same basic purpose of storing data, they behave very differently in terms of **scope** and **hoisting**. These differences affect how variables are accessed, initialized, and managed during program execution. A clear understanding of these concepts helps developers write cleaner, safer, and more predictable code.

Scope

Scope defines the region of a program where a variable can be accessed. In JavaScript, scope mainly exists in two forms:

- **Function Scope**
- **Block Scope**

1. var Keyword

Scope of var: Variables declared using var are **function-scoped**. This means:

- If declared inside a function, the variable is accessible anywhere within that function.

- If declared outside a function, it becomes a **global variable**.
- var does **not** support block scope.

Example:

```
function testVar() {
  if (true) {
    var a = 10;
  }
  console.log(a);
}
testVar();
```

Although a is declared inside the if block, it is accessible outside the block because var ignores block boundaries. This can cause accidental overwriting of values and unexpected behavior in large programs.

Hoisting with var

Hoisting is JavaScript's default behavior of moving declarations to the top of their scope.

- Variables declared with var are **hoisted and initialized with undefined**.
- This allows access to the variable before its declaration.

Example:

```
console.log(x);
var x = 5;
```

Explanation:

Internally, JavaScript interprets the code as:

```
var x;
console.log(x);
x = 5;
```

The output is undefined, not an error. While this may seem convenient, it often leads to logical bugs and makes code harder to debug.

2. let Keyword

Scope of let: The let keyword introduces **block scope**. A variable declared with let is accessible only within the block {} in which it is defined.

Example:

```
function testLet() {
  if (true) {
    let b = 20;
  }
  console.log(b);
}
testLet();
```

The variable b exists only inside the if block. Attempting to access it outside the block results in a **ReferenceError**. This behavior prevents unintended variable access and makes the code more predictable.

Hoisting with let

- Variables declared with let are hoisted but **not initialized**.
- They remain in the **Temporal Dead Zone (TDZ)** from the start of the block until the declaration is executed.
- Accessing the variable during this phase causes an error.

Example:

```
console.log(y);

let y = 10;
```

Explanation:

Although y is hoisted, it cannot be accessed before declaration. JavaScript throws a **ReferenceError**, enforcing better coding discipline and reducing bugs.

3. const Keyword

Scope of const: The const keyword is also **block-scoped**, just like let.

Example:

```
if (true) {
```

```
    const z = 30;
}
console.log(z);
```

Explanation:

The variable z is limited to the block in which it is declared. Accessing it outside the block causes an error.

Hoisting with const

- const variables are hoisted but remain in the **Temporal Dead Zone**.
- They must be initialized at the time of declaration.
- Accessing them before declaration results in a **ReferenceError**.

Example:

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

Reassignment Behavior

- Variables declared with const **cannot be reassigned**.
- However, for objects and arrays, the **contents can be modified**, but the reference cannot change.

Example:

```
const arr = [1, 2, 3];
arr.push(4);    // Allowed
```

```
arr = [5, 6];   // Error
```

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

Answer:- Control Flow statements in JavaScript determine the order in which statements are executed in a program. By default, JavaScript executes code line by line from top to bottom. However, in real-world programming, decision-making and repetition are required. Control Flow statements allow developers to:

- Make decisions
- Execute code conditionally
- Repeat blocks of code efficiently

These statements play a crucial role in building logical and dynamic JavaScript applications.

➢ Types of Control Flow Statements in JavaScript

Control Flow statements in JavaScript can be broadly classified into the following categories:

1. Conditional Statements
2. Looping (Iteration) Statements
3. Jump Statements

Conditional Control Flow Statements :

Conditional statements execute different blocks of code based on a condition.

1.if Statement :

The if statement executes a block of code only when the given condition evaluates to true.

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

2. if–else Statement :

The if–else statement provides an alternative path when the condition is false.

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

3. else if Ladder :

Used when multiple conditions need to be checked.

```
if (score >= 90) {
   grade = "A";
```

```
} else if (score >= 75) {
   grade = "B";
} else {
   grade = "C";
}
```

4. switch Statement :

The switch statement is used when multiple discrete values need to be compared.

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

Looping Control Flow Statements

Loops allow a block of code to be executed repeatedly until a specified condition is met. JavaScript provides several looping constructs, among which for, while, and do-while are the most commonly used.

1. for Loop

The for loop is used when the number of iterations is known in advance. It is a count-controlled loop.

Syntax:

```
for (initialization; condition; increment/decrement) {
  // code to be executed
}
```

- Working

1. Initialization is executed once.
2. Condition is checked before every iteration.
3. If the condition is true, the loop body executes.
4. After execution, the increment/decrement statement runs.
5. The process repeats until the condition becomes false.

Example:-

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

}

Use Case:-

- When loop counter is required
- When iteration count is predetermined

2.while Loop

The while loop is used when the number of iterations is not known beforehand. It is a condition-controlled loop.

Syntax:

```
while (condition) {
   // code to be executed
}
```

Working:

1. Condition is checked before entering the loop.
2. If the condition is true, the loop body executes.
3. If false, the loop body is skipped completely.

Example:

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

Use Case

- Reading user input
- Looping until a condition is satisfied

3. do-while Loop

The do-while loop is similar to the while loop, except that the condition is checked after the loop body executes. Therefore, the loop runs at least once, regardless of the condition.

Syntax:

```
do {
  // code to be executed
} while (condition);
```

Working:

1. Loop body executes first.
2. Condition is checked after execution.
3. If the condition is true, the loop repeats.
4. If false, the loop terminates.

Example:

```
let i = 6;
do {
  console.log(i);
  i++;
} while (i <= 5);
```

Even though the condition is false initially, the loop executes once.

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

Answer: The **Document Object Model (DOM)** is a fundamental concept in JavaScript that allows programmers to interact with and manipulate web pages dynamically. When a web page is loaded in a browser, the browser creates a structured representation of the page, which JavaScript can access and modify. This representation is known as the **Document Object Model**.

The DOM enables JavaScript to change HTML elements, attributes, styles, and content without reloading the page, making modern web applications interactive and dynamic.

What is the Document Object Model (DOM)?

The **Document Object Model (DOM)** is a **programming interface** for HTML and XML documents. It represents the structure of a web page as a **tree of objects**, where each node corresponds to an element, attribute, or piece of text in the document.

In simple terms, the DOM:

- Converts HTML elements into JavaScript objects

- Organizes these objects in a hierarchical tree structure
- Allows JavaScript to access, modify, add, or remove elements dynamically

DOM Tree Structure

Consider the following HTML:

```
<html>
 <body>
  <h1>Hello World</h1>
  <p>This is a paragraph.</p>
 </body>
</html>
```

The DOM tree for this document is:

- Document
  - html
    - body
      - h1
      - p

Each tag becomes a **node** in the DOM tree, and JavaScript can navigate this structure to manipulate the web page.

Importance of DOM in JavaScript

The DOM is important because it allows:

- Dynamic content updates

- User interaction handling
- Form validation
- Animation and style changes
- Single Page Application (SPA) behavior

Without the DOM, JavaScript would not be able to directly modify the content of a webpage after it has loaded.

Selecting Elements in the DOM

To manipulate elements, JavaScript must first **select** them. The DOM provides several methods for selecting HTML elements.

Selecting Elements by ID

The getElementById() method selects an element with a specific id.

let heading = document.getElementById("title");

- Returns a **single element**
- IDs must be unique

Selecting Elements by Class Name

The getElementsByClassName() method selects elements with a given class name.

let items = document.getElementsByClassName("item");

- Returns an **HTMLCollection**
- Multiple elements can be selected

Selecting Elements by Tag Name

The getElementsByTagName() method selects elements by tag name.

let paragraphs = document.getElementsByTagName("p");

Selecting Elements using CSS Selectors

The modern and most flexible methods are:

querySelector()

Selects the **first matching element**.

let firstPara = document.querySelector("p");

querySelectorAll()

Selects **all matching elements**.

let allParas = document.querySelectorAll("p");

Returns a **NodeList**.

Modifying Content using innerText

What is innerText?

The innerText property is used to:

- Get or set the **visible text content** of an element
- Ignore HTML tags
- Respect CSS styling (such as display: none)

Example using innerText

Hello World

let para = document.getElementById("demo");

para.innerText = "Welcome to JavaScript DOM";

Explanation:

- The existing text inside the paragraph is replaced
- Only text is inserted, not HTML

Security Advantage of innerText

Since innerText does not parse HTML, it helps prevent **Cross-Site Scripting (XSS)** attacks when inserting user-generated content.

Modifying Content using innerHTML

What is innerHTML?

The innerHTML property:

- Gets or sets the **HTML content** inside an element
- Allows insertion of HTML tags
- Replaces all existing child elements

Example using innerHTML

let div = document.getElementById("container");

div.innerHTML = "<h2>DOM Manipulation</h2> This is dynamic content ";

Explanation:

- New HTML elements are created inside the div
- Both structure and content are modified dynamically