LAB ASSIGNMENT UNIT -4

4.1.Explain Array methods in JavaScript. Specifically, demonstrate how push(), pop(), shift(), and unshift() modify an array.

Answer: Arrays in JavaScript are used to store multiple values in a single variable. JavaScript provides several built-in methods to modify arrays dynamically. Among these, **push()**, **pop()**, **shift()**, and **unshift()** are commonly used to add or remove elements from an array. These methods directly change the original array.

1. push() Method

The **push()** method adds one or more elements to the **end of an array**.

Example:

```
let arr = [1, 2, 3];

arr.push(4);
```

Result:

[1, 2, 3, 4]

- Adds elements at the last index
- Returns the new length of the array

2. pop() Method

The **pop()** method removes the **last element** from an array.

Example:

let colors = ["Red", "Blue", "Green"];

colors.pop();

Result:

["Red", "Blue"]

- Removes the last element
- Returns the removed element

3. shift() Method

The **shift()** method removes the **first element** of an array.

Example:

let cities = ["Delhi", "Mumbai", "Chennai"];

cities.shift();

Result:

["Mumbai", "Chennai"]

- First element is removed
- Remaining elements shift to the left

4. unshift() Method

The **unshift()** method adds one or more elements to the **beginning of an array**.

Example:

```
let animals = ["Dog", "Cat"];

animals.unshift("Elephant");
```

Result:

```
["Elephant", "Dog", "Cat"]
```

- Adds elements at index 0
- Returns the new length of the array

Table:-

Method	Action	Position
push()	Add element(s)	End
pop()	Remove element	End
shift()	Remove element	Beginning
unshift()	Add element(s)	Beginning

The array methods push(), pop(), shift(), and unshift() are essential for modifying arrays in JavaScript. They allow easy insertion and removal of elements from both ends of an array. Understanding these methods helps in efficient data handling and is important for both exams and practical JavaScript programming.

4.2. What are Promises in JavaScript, and how do async/await simplify working with asynchronous code?

answer: JavaScript is a **single-threaded** language, but it can perform **asynchronous operations** such as fetching data from a server, reading files, or waiting for timers. To handle these operations without blocking the main thread, JavaScript provides **Promises** and later introduced **async/await**, which simplifies working with asynchronous code.

What are Promises in JavaScript?

A **Promise** is an object that represents the **eventual completion or failure** of an asynchronous operation and its resulting value.

In simple words, a Promise is a guarantee that a value will be available **now, later, or never**.

States of a Promise

A Promise can be in one of three states:

1. **Pending** – Initial state, operation not completed yet
2. **Fulfilled** – Operation completed successfully
3. **Rejected** – Operation failed due to an error

Creating a Promise

```
let promise = new Promise((resolve, reject) => {
   let success = true;
   if (success) {
      resolve("Operation Successful");
   } else {
      reject("Operation Failed");
   }
});
```

- resolve() → called when the operation is successful
- reject() → called when the operation fails

Consuming a Promise

Promises are handled using .then(), .catch(), and .finally().

```
promise
 .then(result => console.log(result))
 .catch(error => console.log(error))
 .finally(() => console.log("Done"));
```

Problems with Traditional Promise Chaining

When multiple asynchronous operations are chained using .then(), the code can become:

- Hard to read

- Difficult to debug
- Similar to callback hell (nested structure)

Example:

```
fetchData()
  .then(data => processData(data))
  .then(result => saveResult(result))
  .catch(error => console.log(error));
```

What is async/await?

The **async/await** syntax is a modern way to work with Promises that makes asynchronous code look and behave like **synchronous code**.

- async is used before a function
- await pauses execution until the Promise resolves or rejects
- **async Function**

```
async function myFunction() {
   return "Hello";
}
```

- An async function **always returns a Promise**

Using await

```
async function fetchData() {
   let response = await fetch(url);
   let data = await response.json();
```

```
  console.log(data);
}
```

- await waits for the Promise to resolve
- Makes code easier to read and understand

Error Handling with async/await

Instead of .catch(), errors are handled using try...catch.

```
async function example() {
  try {
    let result = await promise;
    console.log(result);
  } catch (error) {
    console.log(error);
  }
}
```

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

Answer: JavaScript allows web pages to respond to user actions such as clicks, key presses, mouse movements, and form submissions. These actions are called events. To handle events efficiently, JavaScript provides the addEventListener method. Additionally, a powerful

concept known as Event Delegation helps manage events for multiple elements in an optimized way.

What is addEventListener?

The addEventListener method is used to attach an event handler to a specific HTML element. It listens for a particular event and executes a function when that event occurs.

Syntax

element.addEventListener(event, function, useCapture);

- event → Type of event (e.g., "click", "mouseover")
- function → Function to be executed when the event occurs
- useCapture → Optional (default is false)

Example:

```
let button = document.getElementById("btn");

button.addEventListener("click", function () {
  console.log("Button clicked");
});
```

Advantages of addEventListener

- Allows multiple event handlers on the same element
- Does not overwrite existing event handlers
- Supports event bubbling and capturing
- Separates HTML and JavaScript logic

Event Propagation:-

Before understanding Event Delegation, it is important to know event propagation.

Event propagation has two main phases:

1. Capturing Phase – Event travels from the root to the target
2. Bubbling Phase – Event travels from the target back to the root

By default, JavaScript uses event bubbling.

What is Event Delegation?

Event Delegation is a technique where a single event listener is added to a parent element instead of adding listeners to multiple child elements. The parent handles events for its children using event bubbling.

Why Event Delegation is Needed:

- Reduces memory usage

- Improves performance
- Handles dynamically added elements easily
- Simplifies code

How Event Delegation Works

When an event occurs on a child element:

- The event bubbles up to the parent element
- The parent checks which child triggered the event
- Appropriate action is performed

Example Without Event Delegation

```
let buttons = document.querySelectorAll(".item");

buttons.forEach(btn => {
  btn.addEventListener("click", () => {
    console.log("Item clicked");
  });
});
```

This approach is inefficient when many elements exist.

Example Using Event Delegation

```
let list = document.getElementById("list");
```

```
list.addEventListener("click", function (event) {
   if (event.target.tagName === "LI") {
      console.log("List item clicked");
   }
});
```

- Only one event listener is used
- Works even if new <li> elements are added dynamically

Role of event.target:

- event.target refers to the actual element that triggered the event
- It is used in Event Delegation to identify child elements