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

Array Methods in JavaScript

In JavaScript, an array is a data structure used to store multiple values in a single variable. JavaScript provides many built-in array methods that help in adding, removing, and managing elements easily. Among these, **push()**, **pop()**, **shift()**, and **unshift()** are commonly used to modify arrays.

1. push() Method

Function: Adds one or more elements to the end of an array.
Modifies: The original array.
Returns: The new length of the array.

Example: let fruits = ["Apple", "Banana"];
fruits.push("Mango");
console.log(fruits);
Output: ["Apple", "Banana", "Mango"]

2. pop() Method

Function: Removes the last element from an array.
Modifies: The original array.
Returns: The removed element.

Example: let fruits = ["Apple", "Banana", "Mango"];
fruits.pop();
console.log(fruits);
Output: ["Apple", "Banana"]

3. shift()

Method Function: Removes the first element from an array.
Effect: Shifts all remaining elements to a lower index.
Returns: The removed element.

Example: let fruits = ["Apple", "Banana", "Mango"];
fruits.shift();
console.log(fruits);
Output: ["Banana", "Mango"]

4. unshift()

Method Function: Adds one or more elements to the beginning of an array.
Effect: Shifts existing elements to a higher index.
Returns: The new length of the array.

Example: let fruits = ["Banana", "Mango"];
fruits.unshift("Apple");
console.log(fruits);
Output: ["Apple", "Banana", "Mango"]

Conclusion: These array methods are essential for adding and removing elements from arrays efficiently and are commonly used in JavaScript programming.

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

Promises in JavaScript

In JavaScript, Promises are used to handle asynchronous operations such as fetching data from a server, reading files, or performing time-consuming tasks. A Promise represents a value that may be available now, later, or never.

States of a Promise

- **Pending** – The operation is still in progress
- **Fulfilled** – The operation completed successfully
- **Rejected** – The operation failed

Creating a Promise

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

Handling a Promise

Promises are handled using **.then()** and **.catch()** methods.

```
promise
  .then(result => {
    console.log(result);
  })
  .catch(error => {
 console.log(error);
});
```

Problems with Traditional Promise Handling

- Code becomes hard to read when many **.then()** calls are chained
- Error handling can be confusing in complex asynchronous flows

Async/Await in JavaScript

async and **await** are modern JavaScript keywords that make working with Promises simpler and more readable.

async

Declares a function that always returns a Promise.

await Pauses the execution of the async function until the Promise is resolved or rejected.

Example Using Async/Await async function fetchData() { try { let result = await promise; console.log(result); } catch (error) { console.log(error); } } fetchData(); **Real-World Example** async function getUser() { let response = await fetch("https://api.example.com/user"); let data = await response.json(); console.log(data); }

Example Using Async/Await

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

      fetchData();
```

Real-World Example

```
    async function getUser() {
let response = await  fetch("https://api.example.com/user");
    let data = await response.json();
        console.log(data);
        }
```

4.3. Describe the Concept of Event Delegation and Explain the Use of addEventListener

Event Delegation in JavaScript

Event Delegation is a JavaScript technique where a single event listener is attached to a parent element instead of adding separate event listeners to multiple child elements. It works because of **event bubbling**, where an event triggered on a child element bubbles up to its parent elements.

Using event delegation improves performance, reduces memory usage, and makes it easier to handle dynamically added elements.

Example Without Event Delegation

```
<button>Button 1</button>
<button>Button 2</button>
<button>Button 3</button>

let buttons = document.querySelectorAll("button");
   buttons.forEach(btn => {
       btn.addEventListener("click", () => {
         console.log("Button clicked");
       });
   });
```

Multiple event listeners are created.

Example With Event Delegation

```
<div id="container">
<button>Button 1</button>
<button>Button 2</button>
<button>Button 3</button>
</div>

document.getElementById("container").addEventListener("click", function (event) {
  if (event.target.tagName === "BUTTON") {
console.log("Button clicked:", event.target.innerText);
  }
});
```

Only one event listener handles all button clicks.

Advantages of Event Delegation

- Better performance (fewer event listeners)
- Works with dynamically added elements
- Cleaner and more maintainable code

addEventListener() in JavaScript

The **addEventListener()** method is used to attach an event handler to an HTML element.

Syntax

```
element.addEventListener(event, function, useCapture);
```

Explanation of Parameters

event The event type (e.g., "click", "mouseover", "keydown"
function - The function to execute when the event occurs
useCapture (optional) - Boolean that defines event flow (default is false,meaning bubbling)

Example of addEventListener()

```
document.getElementById("btn").addEventListener("click", function () {
alert("Button clicked!");
});
```

Why Use addEventListener() Instead of Inline Events?

Allows multiple event handlers on the same element
Separates JavaScript from HTML
Supports event delegation
More flexible and powerful

Relationship Between Event Delegation and addEventListener()

Event delegation uses **addEventListener()** on a parent element to listen for events from child elements. The event object **(event.target)** is used to identify which child triggered the event.