LAB ASSIGNMENT-4

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

⇨ **Array Methods in JavaScript:-**

An **array** in JavaScript is a special variable used to store **multiple values in a single variable**. JavaScript provides many **built-in array methods** to add, remove, and manipulate elements.
Some commonly used array methods are **push()**, **pop()**, **shift()**, and **unshift()**. These methods modify the original array.

---

1. push() Method
Definition:
The push() method **adds one or more elements to the end of an array** and returns the new length of the array.

Example:

```
let fruits = ["Apple", "Banana"];
fruits.push("Mango");

console.log(fruits);
```

Output:
["Apple", "Banana", "Mango"]
Explanation:

- "Mango" is added at the **end** of the array.

---

2. pop() Method
Definition:
The pop() method **removes the last element** from an array and returns that element.
Example:

```
let fruits = ["Apple", "Banana", "Mango"];
fruits.pop();
```

console.log(fruits);

Output:

["Apple", "Banana"]

Explanation:

- The last element "Mango" is removed from the array.

---

3. shift() Method

Definition:

The shift() method **removes the first element** of an array and shifts the remaining elements to lower indexes.

Example:

let fruits = ["Apple", "Banana", "Mango"];

fruits.shift();

console.log(fruits);

Output:

["Banana", "Mango"]

Explanation:

- "Apple" (first element) is removed.
- The index of remaining elements changes.

---

4. unshift() Method

Definition:

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

Example:

let fruits = ["Banana", "Mango"];

fruits.unshift("Apple");

console.log(fruits);

Output:

["Apple", "Banana", "Mango"]

Explanation:

- "Apple" is added at the **start** of the array.

Summary Table

Method	Action	Position
push()	Adds element(s)	End
pop()	Removes one element	End
shift()	Removes one element	Start
unshift()	Adds element(s)	Start

Conclusion

The array methods push(), pop(), shift(), and unshift() are used to **add or remove elements** from an array. These methods help in managing data efficiently and are commonly used in JavaScript programming.

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

⇨ **Promises in JavaScript:-**

What is Asynchronous JavaScript?

JavaScript is **single-threaded**, but it can perform **asynchronous operations** such as:

- Fetching data from a server
- Reading files
- Timers (setTimeout)

To handle these operations, JavaScript uses **Promises** and **async/await**.

What is a Promise?

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

States of a Promise:

A Promise can be in **three states**:

1. **Pending** – Initial state, operation not completed
2. **Fulfilled** – Operation completed successfully
3. **Rejected** – Operation failed

Creating a Promise

Example:

```
let promise = new Promise(function (resolve, reject) {
  let success = true;

  if (success) {
    resolve("Data fetched successfully");
  } else {
    reject("Error while fetching data");
  }
});
```

Consuming a Promise using .then() and .catch()

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

Output:

Data fetched successfully

Problems with Promises

- Multiple .then() calls can make code **hard to read**
- Leads to **promise chaining**
- Difficult to handle errors in complex logic

To solve this, **async/await** was introduced.

---

async/await in JavaScript

What is async/await?

- async and await are **keywords** introduced in **ES6**
- They make asynchronous code look like **synchronous code**
- Built on top of Promises

Using async/await

Example:

```
function fetchData() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      resolve("Data received");
    }, 2000);
  });
}

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

```
}

getData();
```

Output:

Data received

How async/await Simplify Asynchronous Code

Without async/await (Promise chaining):

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

With async/await:

```
async function getData() {
  let result = await fetchData();
  console.log(result);
}
```

Advantages of async/await

1. Improves **code readability**
2. Easier **error handling** using try...catch
3. Avoids long promise chains
4. Looks similar to synchronous code
5. Makes debugging easier

Difference Between Promises and async/await

Promises	async/await
Uses .then() and .catch()	Uses await and try...catch
Code can be lengthy	Code is cleaner
Harder to read	Easier to understand

Conclusion

Promises are used to handle asynchronous operations in JavaScript. The async/await syntax simplifies working with Promises by making the code cleaner, more readable, and easier to maintain.

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

⇨ **Event Delegation in JavaScript:-**

What is an Event?

An **event** is an action performed by the user or browser, such as:

- Clicking a button
- Typing in an input field
- Moving the mouse

JavaScript handles events using **event listeners**.

Concept of Event Delegation

Definition:

Event Delegation is a technique in JavaScript where **a single event listener is attached to a parent element** to handle events for its **child elements**, instead of attaching listeners to each child.
This works because of **event bubbling**, where events propagate from the child element up to its parent.

How Event Delegation Works

1. An event occurs on a child element
2. The event bubbles up to the parent
3. The parent's event listener detects the event
4. The target element is identified using event.target

Example Without Event Delegation (Not Efficient)

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

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

Problem:

- Multiple event listeners
- Not efficient for large or dynamic lists

Example With Event Delegation (Efficient)

HTML:

```
<ul id="list">
  <li>Item 1</li>
  <li>Item 2</li>
```

```
  <li>Item 3</li>
</ul>
```

JavaScript:

```
document.getElementById("list").addEventListener("click",
function (event) {
  if (event.target.tagName === "LI") {
    console.log("Clicked:", event.target.innerText);
  }
});
```

Output:

Clicked: Item 1

Advantages of Event Delegation

1. Improves performance
2. Uses fewer event listeners
3. Works with dynamically added elements
4. Cleaner and more maintainable code

---

addEventListener() in JavaScript

Definition:

addEventListener() is a method used to **attach an event handler to an element** without overwriting existing event handlers.

Syntax:

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

- **event** – type of event (e.g., "click", "mouseover")
- **function** – function to execute
- **useCapture** – optional (default is false)

Example of addEventListener()

```
<button id="myBtn">Click Me</button>
document.getElementById("myBtn").addEventListener("click", function () {
   alert("Button Clicked!");
});
```

Benefits of addEventListener()

1. Allows multiple event handlers on one element
2. Better control over event propagation
3. Separates JavaScript from HTML
4. Supports event delegation

Relationship Between Event Delegation and addEventListener

- Event Delegation **uses addEventListener()**
- The listener is attached to a **parent element**
- Child events are handled using event.target

Conclusion

Event Delegation is an efficient technique that uses event bubbling to handle events on multiple child elements with a single event listener. The addEventListener() method provides a flexible and powerful way to handle events in JavaScript.