

Lab Assignment 4

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

Ans:-

➤ **Modifying the End of the Array**

These methods deal with the "tail" of the array (the highest index).

push() — Add to the End

- **Action:** Adds one or more elements to the end of an array.
- **Return Value:** Returns the new length of the array.

Ex:-

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

```
// Add "Orange" to the end
```

```
let newLength = fruits.push("Orange");
```

```
console.log(fruits); // Output: ["Apple", "Banana", "Orange"]
```

```
console.log(newLength); // Output: 3
```

pop() — Remove from the End

- **Action:** Removes the last element from an array.
- **Return Value:** Returns the element that was removed.

Ex:-

```
let fruits = ["Apple", "Banana", "Orange"];
```

```
// Remove the last item ("Orange")
```

```
let removedItem = fruits.pop();
```

```
console.log(fruits); // Output: ["Apple", "Banana"]
```

```
console.log(removedItem); // Output: "Orange"
```

➤ **Modifying the Start of the Array**

These methods deal with the "head" of the array (index 0). Note that these operations are generally slower than push/pop because the computer has to re-index every other element in the array.

unshift() — Add to the Start

- **Action:** Adds one or more elements to the beginning of an array.
- **Return Value:** Returns the new length of the array.

Ex:-

```
let fruits = ["Banana", "Orange"];
```

```
// Add "Apple" to the start
```

```
let newLength = fruits.unshift("Apple");
```

```
console.log(fruits); // Output: ["Apple", "Banana", "Orange"]
```

```
console.log(newLength); // Output: 3
```

shift() — Remove from the Start

- **Action:** Removes the first element from an array.
- **Return Value:** Returns the element that was removed.

Ex:-

```
let fruits = ["Apple", "Banana", "Orange"];
```

```
// Remove the first item ("Apple")
```

```
let removedItem = fruits.shift();
```

```
console.log(fruits); // Output: ["Banana", "Orange"]
```

```
console.log(removedItem); // Output: "Apple"
```

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

Ans:-

What is a Promise?

In JavaScript, a Promise is an object that represents the eventual completion (or failure) of an asynchronous operation and its resulting value. Think of it like a real-life "receipt" for an order: you don't have the item yet, but the receipt promises that you will eventually get either the item (success) or an explanation of why it's out of stock (failure).

A Promise has three distinct states:

- **Pending:** The initial state; the operation has not started or finished yet.
- **Fulfilled (Resolved):** The operation completed successfully, and the promise now holds a value.
- **Rejected:** The operation failed, and the promise now holds a reason (error).

❖ How to use a Promise

Traditionally, you access the result of a Promise using `.then()` for success and `.catch()` for errors.

Code:

```
// A function that returns a Promise
function fetchData() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      const success = true;
      if (success) {
        resolve("Data received! 📦");
      } else {

```

```

        reject("Connection failed ❌");
    }
}, 2000);
});
}

```

```

// Consuming the Promise
console.log("1. Requesting data...");

```

```

fetchData()
    .then((data) => {
        console.log("2. Success:", data);
    })
    .catch((error) => {
        console.error("2. Error:", error);
    })
    .finally(() => {
        console.log("3. Operation finished.");
    });

```

❖ The Problem: "Promise Chaining"

While Promises are better than old-school callbacks, they can still become messy when you need to perform multiple steps in a sequence. This is often called "Promise Chaining."

- **Scenario:** User logs in ,Get User ID , Get User Settings.

Using Raw Promises:

Code:

```

loginUser()
    .then(user => {
        return getUserId(user); // Returns a new Promise
    })
    .then(userId => {
        return getUserSettings(userId); // Returns a new Promise
    })

```

```
})
.then(settings => {
  console.log(settings);
})
.catch(err => {
  console.error(err);
});
```

This is readable, but the indentation and repeated `.then()` calls can get visually cluttered, and sharing variables between different `.then()` blocks is difficult.

❖ How **async/await** Simplifies It

Introduced in ES2017, `async` and `await` are "syntactic sugar" built on top of Promises. They allow you to write asynchronous code that looks and behaves like synchronous code.

- **async:** Placing this keyword before a function tells JavaScript that this function will always return a Promise.
- **await:** This keyword can only be used inside an `async` function. It makes JavaScript "pause" the execution of that specific function until the Promise is resolved.

The Simplification

Here is the same logic as above, rewritten with `async/await`.

Code:

```
async function initializeUserSettings() {
  try {
    // The code "pauses" at each await line until the result is ready
    const user = await loginUser();
    const userId = await getUserId(user);
    const settings = await getUserSettings(userId);

    console.log(settings);
  } catch (err) {
```

```
// Standard try/catch handles errors for ALL steps above  
console.error(err);}}
```

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

Ans:-

Event Delegation and addEventListener

In JavaScript, handling user interactions efficiently is key to building performant web applications. Event Delegation is a pattern that relies on the foundational method addEventListener to manage events smartly.

- **addEventListener**

The addEventListener() method is the standard way to register an event handler (a function that runs when an event happens) to a specific DOM element.

Syntax:

element.addEventListener(type, listener, options);

- **type:** A string representing the event type to listen for (e.g., 'click', 'keydown', 'submit').
- **listener:** The function (callback) that runs when the event occurs.
- **options (optional):** An object or boolean specifying characteristics like capture (whether to use the capturing phase) or once (whether the listener should be invoked at most once)

Ex:-

```
const btn = document.querySelector('#myBtn');
```

```
btn.addEventListener('click', function(event) {  
  console.log('Button clicked!');  
});
```

The Concept of Event Delegation

Event Delegation is a technique where you attach a single event listener to a parent element to manage events for all of its descendants (children), rather than attaching individual listeners to each child.

This works because of Event Bubbling.

How it Works (The Mechanics)

- **Event Bubbling:** When an event (like a click) happens on an element, it doesn't just stay there. It "bubbles" up the DOM tree, triggering the same event on the parent, the grandparent, and so on, all the way to the window.
- **The Parent Listens:** Because the event bubbles up, a listener on a parent element will eventually be triggered by a click on any of its children.
- **Identifying the Target:** Inside the parent's event listener, you use `event.target` to pinpoint exactly which child element was actually clicked.

Why use Event Delegation?

- **Memory Efficiency:** Instead of creating 100 separate functions for a list of 100 items, you create only one function attached to the container.
- **Dynamic Elements:** If you add new items to the list after the page loads (e.g., via an API call), the parent listener automatically handles them. You don't need to re-attach listeners to new elements.

Comparison Example

Imagine a generic "To-Do List" where users can delete items.

Without Delegation (Inefficient)

You would have to loop through every list item and add a listener. If you add a new Task later, this code won't work for it unless you run it again.

Ex:-

```
const items = document.querySelectorAll('.list-item');
```

```
items.forEach(item => {  
  item.addEventListener('click', e => {  
    console.log('Item clicked');  
  });  
});
```

With Delegation (Efficient)

You attach the listener once to the parent .

// 1. Select the parent

```
const list = document.querySelector('#todo-list');
```

// 2. Add ONE listener to the parent

```
list.addEventListener('click', function(event) {
```

// 3. Check if the actual clicked element (event.target) is a list
 item

// The 'closest' method is useful to handle clicks on icons inside
 the li

```
    const clickedItem = event.target.closest('.list-item');
```

```
    if (clickedItem) {
```

```
      console.log('List item clicked:', clickedItem.textContent);
```

```
      // Perform action, e.g., toggle 'completed' class
```

```
      clickedItem.classList.toggle('done');
```

```
    }
```

```
  });
```