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

Specifically, demonstrate how push(), pop(), shift(), and unshift() modify an array.

Answer:

Arrays in JavaScript are dynamic collections of data, and JavaScript provides several built-in methods to traverse and modify these collections. The four primary methods used to add or remove elements from the start or end of an array are `push()`, `pop()`, `shift()`, and `unshift()`. These are known as "mutator" methods because they modify the original array directly rather than creating a new one.

1. push() — Adding to the End The `push()` method appends one or more elements to the very end of an array.

- **Modification:** It increases the length of the array.
- **Return Value:** It returns the *new length* of the array (an integer).
- **Usage:** This is commonly used when building a list of data where order matters and new items arrive sequentially.

Example:

JavaScript

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

console.log(fruits);      // Output: ["Apple", "Banana",
"Orange", "Mango"]
console.log(newLength);   // Output: 4
```

2. pop() — Removing from the End The `pop()` method removes the last element from an array.

- **Modification:** It decreases the length of the array by one.
- **Return Value:** It returns the specific element that was removed. If the array is empty, it returns `undefined`.
- **Usage:** This is often used in "Stack" data structures (Last-In, First-Out).

Example:

JavaScript

```
let numbers = [10, 20, 30];
let removedItem = numbers.pop();
```

```
console.log(numbers);     // Output: [10, 20]
console.log(removedItem); // Output: 30
```
3. shift() — Removing from the Beginning The `shift()` method removes the first element from an array.

- **Modification:** It decreases the length of the array by one. Crucially, it also changes the indices of every remaining element; the element at index 1 moves to index 0, index 2 moves to index 1, and so on.
- **Return Value:** It returns the element that was removed.
- **Usage:** Used in "Queue" structures (First-In, First-Out) to process the oldest item next.

Example:

JavaScript

```
let tasks = ["Task 1", "Task 2", "Task 3"];
let currentTask = tasks.shift();

console.log(tasks);       // Output: ["Task 2", "Task 3"]
console.log(currentTask); // Output: "Task 1"
```
4. unshift() — Adding to the Beginning The `unshift()` method adds one or more elements to the beginning of an array.

- **Modification:** It increases the length of the array and shifts all existing elements to higher indices (index 0 becomes index 1, etc.) to make room for the new items.
- **Return Value:** It returns the *new length* of the array.

Example:

JavaScript

```
let queue = ["User B", "User C"];
queue.unshift("User A");

console.log(queue); // Output: ["User A", "User B", "User C"]
```

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

Answer:

Promises in JavaScript A **Promise** is a JavaScript object that represents the eventual completion (or failure) of an asynchronous operation and its resulting value. In simpler terms, it is a placeholder for a value that is not yet known when the code starts executing but will be resolved in the future.

A Promise exists in one of three states:

1. **Pending:** The initial state; the operation has not yet completed.
2. **Fulfilled (Resolved):** The operation completed successfully, and the promise now holds a "value."
3. **Rejected:** The operation failed, and the promise now holds a "reason" (usually an error).

Promises were introduced to solve the problem of "Callback Hell," where nested callbacks made code difficult to read and debug. However, using Promises still requires chaining `.then()` and `.catch()` methods, which can become verbose.

How async/await Simplifies Asynchronous Code Introduced in ES2017, the keywords `async` and `await` act as "syntactic sugar" built on top of Promises. They do not change the underlying logic of Promises but strictly improve the syntax and readability of the code.

They simplify coding in three major ways:

1. **Synchronous Appearance:** The most significant advantage is that `async/await` allows developers to write asynchronous code that looks and behaves like synchronous (top-to-bottom) code. You pause the execution of a function at the `await` keyword until the Promise settles, rather than putting the subsequent code inside a `.then()` callback block.
2. **Better Error Handling:** With standard Promises, you must use `.catch()` to handle errors. With `async/await`, you can use standard JavaScript `try...catch` blocks. This unifies error handling for both synchronous and asynchronous code in the same function.
3. **Clean Code Structure:** It removes the visual clutter of chained functions. Complex logic involving conditionals or loops (like `for` loops) is much harder to implement inside `.then()` chains but is trivial with `await`.

Comparison Example:

Using Standard Promises:

JavaScript

```
function getData() {
    fetch('https://api.example.com/data')
        .then(response => response.json())
        .then(data => console.log(data))
        .catch(error => console.error(error));
}
```

Using Async/Await (Simplified):

JavaScript

```
async function getData() {
    try {
```

```
        const response = await fetch('https://
api.example.com/data');
        const data = await response.json();
        console.log(data);
    } catch (error) {
        console.error(error);
    }
}
```

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

Answer:

Event Delegation Event Delegation is a performance optimization pattern in JavaScript that allows developers to handle events for multiple elements by attaching a single event listener to a common parent element, rather than attaching individual listeners to every single child element.

This technique works due to **Event Bubbling**. When an event (like a 'click') occurs on a specific element (the target), it doesn't stop there; it bubbles up through the DOM tree to its parent, then the grandparent, and so on.

Key Benefits:

1. **Memory Efficiency:** Instead of creating 100 event listeners for a list of 100 items (which consumes significant memory), you create only one listener on the container `<ul>` or `<div>`.
2. **Dynamic Content Handling:** If you add new elements to the DOM after the page has loaded (e.g., adding a new item to a to-do list via JavaScript), you do not need to attach a new listener to the new item. The parent listener will automatically capture events from the new child because of bubbling.

Mechanism: Inside the parent's event listener, you use `event.target` to identify exactly which child element triggered the event.

The Use of addEventListener The `addEventListener()` method is the modern, standard way to register an event handler on a generic HTML element, Document, or Window.

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

Why it is the preferred method:

1. **Multiple Handlers:** Unlike the older `onclick` property (e.g., `element.onclick = func`), which overwrites any existing event handlers, `addEventListener` allows you to attach *multiple* different functions to the same event type on the same element.
2. **Separation of Concerns:** It keeps JavaScript logic separate from HTML markup (unlike inline HTML attributes like `<button onclick="...">`).

3. **Event Flow Control:** It accepts a third optional parameter (options object or boolean) that allows you to control whether the event should be triggered during the "capturing" phase or the "bubbling" phase.

Example of Event Delegation using addEventListener:

JavaScript

```
// Select the parent list
const list = document.getElementById('myList');

// Attach ONE listener to the parent
list.addEventListener('click', function(event) {
    // Check if the actual clicked element (event.target) is
a list item
    if (event.target.tagName === 'LI') {
        console.log('You clicked on item:',
event.target.textContent);
        // Perform action, e.g., strike through the text
        event.target.style.textDecoration = 'line-through';
    }
});
```