1. Explain Array methods in JavaScript. Specifically, demonstrate how

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

Ans:-

Array Methods in JavaScript

Arrays in JavaScript are used to store multiple values in a single variable. JavaScript provides **array methods** to add or remove elements, especially from the **beginning or end** of an array.

---

1. push()

- **Adds one or more elements to the end** of an array.
- **Modifies the original array**.
- Returns the **new length** of the array.

```
let fruits = ["apple", "banana"];

fruits.push("orange");


console.log(fruits);

// ["apple", "banana", "orange"]
```

---

2. pop()

- **Removes the last element** from an array.
- **Modifies the original array**.
- Returns the **removed element**.

```
let fruits = ["apple", "banana", "orange"];

fruits.pop();


console.log(fruits);

// ["apple", "banana"]
```

---

3. shift()

- **Removes the first element** from an array.
- **Modifies the original array**.
- Returns the **removed element**.
- Slower than pop() because all elements are re-indexed.

```
let fruits = ["apple", "banana", "orange"];

fruits.shift();


console.log(fruits);

// ["banana", "orange"]
```

---

4. unshift()

- **Adds one or more elements to the beginning** of an array.
- **Modifies the original array**.
- Returns the **new length** of the array.
- Slower than push() due to re-indexing.

```
let fruits = ["banana", "orange"];

fruits.unshift("apple");


console.log(fruits);

// ["apple", "banana", "orange"]
```

Summary Table

Method	Action	Affects Original Array	Return Value
push()	Add to end	Yes	New length
pop()	Remove from end	Yes	Removed element
shift()	Remove from beginning	Yes	Removed element
unshift()	Add to beginning	Yes	New length

Example Showing All Methods Together

```
let numbers = [2, 3];


numbers.push(4);      // [2, 3, 4]
numbers.pop();        // [2, 3]
numbers.unshift(1);   // [1, 2, 3]
numbers.shift();      // [2, 3]
```

Conclusion

- **push() and pop()** work at the **end** of an array.
- **shift() and unshift()** work at the **beginning** of an array.
- All four methods **modify the original array**.

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

Ans:-

Promises in JavaScript

A **Promise** in JavaScript is an object that represents the **eventual completion or failure of an asynchronous operation**. It allows you to handle asynchronous tasks (like fetching data, reading files, or timers) in a structured way.

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

1. **Pending** – initial state, neither fulfilled nor rejected
2. **Fulfilled** – operation completed successfully
3. **Rejected** – operation failed

Example of a Promise

```
const promise = new Promise((resolve, reject) => {
  let success = true;
  if (success) {
    resolve("Task completed");
  } else {
    reject("Task failed");
  }
});
```

Handling a Promise

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

Problems with Traditional Promise Chaining

- Multiple .then() calls can make code **harder to read**.
- Error handling becomes less intuitive in complex chains.

---

async / await

async and await are keywords introduced to **simplify working with Promises** by making asynchronous code look and behave more like synchronous code.

How They Work

- An **async function** always returns a Promise.
- **await** pauses execution until the Promise is resolved or rejected.
- Errors can be handled using **try...catch**, similar to synchronous code.

Example Using async / await

```
async function fetchData() {
  try {
    const response = await fetch("https://example.com/data");
    const data = await response.json();
    console.log(data);
  } catch (error) {
    console.error("Error:", error);
  }
}
```

Comparing Promises vs async/await

Using Promises

```
fetch("https://example.com/data")
```

```
  .then(response => response.json())
  .then(data => console.log(data))
  .catch(error => console.error(error));
```

Using async/await

```
async function getData() {
  try {
    const data = await fetch("https://example.com/data")
      .then(res => res.json());
    console.log(data);
  } catch (error) {
    console.error(error);
  }
}
```

Benefits of async / await

- ✓ Cleaner and more readable code
- ✓ Easier error handling with try...catch
- ✓ Avoids deep Promise chains (callback-like behavior)

Summary

Feature	Promises	async / await
Syntax	.then() / .catch()	try...catch
Readability	Moderate	High
Error Handling	Chain-based	Block-based
Return Type	Promise	Promise (implicitly)

Conclusion

- **Promises** manage asynchronous operations and their results.
- **async/await** provides a cleaner, more readable way to work with Promises.
- async/await is essentially **syntactic sugar** over Promises, not a replacement.

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

Ans:-

Event Delegation

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

Why Event Delegation is Used

- Improves **performance** (fewer event listeners)
- Handles **dynamically added elements**
- Makes code **cleaner and easier to maintain**

Example Without Event Delegation

```
const buttons = document.querySelectorAll("button");


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

Example With Event Delegation

```
const container = document.getElementById("container");


container.addEventListener("click", (event) => {
  if (event.target.tagName === "BUTTON") {
    console.log("Button clicked");
  }
});
```

Here:

- The event listener is attached to the **parent (container)**
- event.target identifies the **actual element clicked**
- Button clicks are handled even if buttons are added later

---

addEventListener()

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

Syntax

```
element.addEventListener(eventType, callback, useCapture);
```

Parameters

1. **eventType** – type of event (e.g., "click", "keydown")
2. **callback** – function executed when the event occurs
3. **useCapture** *(optional)* – false (default, bubbling phase) or true (capturing phase)

Example of addEventListener

```
const btn = document.getElementById("myBtn");


btn.addEventListener("click", () => {
  alert("Button clicked!");
});
```

Event Bubbling vs Capturing

- **Bubbling**: event flows from child → parent (default)
- **Capturing**: event flows from parent → child

Event delegation relies on **bubbling**.

Key Differences Summary

Concept	Description
Event Delegation	Handling events using a parent element
Event Bubbling	Events move upward in the DOM
addEventListener()	Attaches event handlers to elements

Conclusion

- **Event Delegation** uses event bubbling to manage events efficiently.
- **addEventListener()** allows multiple event handlers and better control over event flow.
- Together, they help build **scalable and dynamic web applications**.