1. **JavaScript Basics & Setup Scenario:**
   Set up your community portal to use JavaScript. Objective: Configure environment and test basic script functionality.
   Task:
   • Use <script src = “main.js”></script> in HTML
   • Log "Welcome to the Community Portal" using console.log()
   • Use an alert to notify when the page is fully loaded

   main.js

```
console.log("Welcome to the Community Portal");

window.onload = function () {
  alert("Page fully loaded");
};
```

   HTML

```
<script src="main.js"></script>
```

2. **Syntax, Data Types, and Operators Scenario:**
   Store event details like name, date, and available seats.
   Objective: Use proper data types and operations.
   Task:
   • Use const for event name and date, let for seats
   • Concatenate event info using template literals
   • Use ++ or -- to manage seat count on registration

```
const eventName = "Art Expo";
const eventDate = "2025-06-01";
let availableSeats = 50;

console.log(`Event: ${eventName} on ${eventDate} - Seats Available:
${availableSeats}`);
availableSeats--; // After registration
```

   O/P:

```
Event: Art Expo on 2025-06-01 - Seats Available: 50
```

3. <u>**Conditionals, Loops, and Error Handling Scenario:**</u>

   Only show valid events and limit registrations.

   Objective: Apply conditions and handle invalid data.

   User Story: As a user, I want only upcoming events with seats to be displayed.

   Task:

   • Use if-else to hide past or full events

   • Loop through the event list and display using forEach()

   • Wrap registration logic in try-catch to handle errors

```
const events = [
  { name: "Music Fest", date: "2025-06-10", seats: 0 },
  { name: "Art Fair", date: "2025-07-01", seats: 10 },
];

events.forEach(event => {
  if (new Date(event.date) > new Date() && event.seats > 0) {
    console.log(`Upcoming: ${event.name}`);
  } else {
    console.log(`Event ${event.name} is either past or full.`);
  }
});

try {
  let userRegistered = true;
  if (!userRegistered) throw "Registration failed";
  console.log("Registration successful");
} catch (e) {
  console.error(e);
}
```

   O/P :

```
Event Music Fest is either past or full.
Upcoming: Art Fair
Registration successful
```

4. <u>**Functions, Scope, Closures, Higher-Order Functions Scenario:**</u>

   Create reusable functions for event operations.

   Objective: Encapsulate logic and use closures.

   Task:

   • Create addEvent(), registerUser(), filterEventsByCategory()

   • Use closure to track total registrations for a category

   • Pass callbacks to filter functions for dynamic search

```
function addEvent(name, date) {
  return { name, date };
}

function registerUser(event) {
  console.log(`Registered for ${event.name}`);
}

function filterEventsByCategory(events, category, callback) {
  return events.filter(e => callback(e, category));
}

function categoryCounter() {
  let count = 0;
  return function () {
    count++;
    console.log(`Total registered: ${count}`);
  };
}

const trackMusic = categoryCounter();
trackMusic(); // 1
trackMusic(); // 2
```

O/P:

```
Total registered: 1
Total registered: 2
```

5. **<u>Objects and Prototypes Scenario:</u>**
   Each event is an object with properties and methods.
   Objective: Model real-world entities using objects.
   Task:
   • Define Event constructor or class
   • Add checkAvailability() to prototype
   • List object keys and values using Object.entries()

```
function Event(name, seats) {
  this.name = name;
  this.seats = seats;
}
```

```
Event.prototype.checkAvailability = function () {
  return this.seats > 0;
};

const e1 = new Event("Food Fest", 20);
console.log(e1.checkAvailability()); // true
console.log(Object.entries(e1));
```

O/P:

```
true
[["name", "Food Fest"], ["seats", 20]]
```

6. <u>**Arrays and Methods Scenario:**</u>
   Manage an array of all community events.
   Objective: Use array methods for CRUD operations.
   Task:
   • Add new events using .push()
   • Use .filter() to show only music events
   • Use .map() to format display cards (e.g., "Workshop on Baking")

```
let events = [];

events.push({ name: "Dance Show", category: "music" });
events.push({ name: "Coding Workshop", category: "tech" });

let musicEvents = events.filter(e => e.category === "music");
let formatted = events.map(e => `Event: ${e.name}`);

console.log(musicEvents);
console.log(formatted);
```

O/P:

```
[ { name: "Dance Show", category: "music" } ]
[ "Event: Dance Show", "Event: Coding Workshop" ]
```

7. <u>**DOM Manipulation Scenario:**</u>
   Display all events dynamically on the webpage.
   Objective: Render events using JS.
   Task:
   • Access DOM elements using querySelector()

• Create and append event cards using createElement()

• Update UI when user registers or cancels

```
const container = document.querySelector("#eventContainer");
const card = document.createElement("div");
card.textContent = "Dance Show - Register Now";
container.appendChild(card);
```

8. **<u>Event Handling Scenario:</u>**

   Add interactive elements like buttons and filters.

   Objective: Respond to user actions.

   Task:

   • Use onclick for "Register" buttons

   • Use onchange to filter events by category

   • Use keydown to allow quick search by name

```
document.getElementById("registerBtn").onclick = () => alert("Registered!");

document.getElementById("categoryFilter").onchange = function () {
  console.log("Filter by:", this.value);
};

document.getElementById("searchBox").onkeydown = (e) => {
  console.log("Typing:", e.key);
};
```

   O/P:

```
Filter by: music
Typing: D
```

9. **<u>Async JS, Promises, Async/Await Scenario:</u>**

   Fetch event data from a mock API.

   Objective: Use asynchronous logic for remote operations.

   Task:

   • Fetch events from a mock JSON endpoint

   • Use .then() and .catch() to handle results

   • Rewrite using async/await and show loading spinner

```
// Using .then()
fetch("events.json")
  .then(res => res.json())
  .then(data => console.log("Events:", data))
  .catch(err => console.error(err));

// Using async/await
async function fetchEvents() {
  document.getElementById("loading").style.display = "block";
  try {
    const res = await fetch("events.json");
    const data = await res.json();
    console.log("Events:", data);
  } catch (e) {
    console.error("Error:", e);
  } finally {
    document.getElementById("loading").style.display = "none";
  }
}
```

O/P:

```
Events: [ { name: "Dance Show", ... }, ... ]
```

10. **<u>Modern JavaScript Features Scenario:</u>**
    Refactor code to be concise and maintainable.
    Objective: Use ES6+ features.
    Task:
    • Use let, const, default parameters in functions
    • Use destructuring to extract event details
    • Use spread operator to clone event list before filtering

```
const showEvent = ({ name, date }) => {
  console.log(`${name} on ${date}`);
};

const event = { name: "Tech Talk", date: "2025-06-25" };
showEvent(event);

const eventList = [{ name: "A" }, { name: "B" }];
const cloneList = [...eventList];
```

O/P:

```
Tech Talk on 2025-06-25
```

11. <u>**Working with Forms Scenario:**</u>

Create a registration form for event sign-up.

Objective: Connect form inputs to JavaScript.

Task:

• Capture name, email, and selected event using form.elements

• Prevent default form behavior using event.preventDefault()

• Validate inputs and show errors inline

```
<form id="regForm">
  <input name="name" required>
  <input name="email" required>
  <select name="event"><option value="music">Music</option></select>
  <button type="submit">Submit</button>
</form>
<p id="errorMsg"></p>

<script>
document.getElementById("regForm").addEventListener("submit", function(e) {
  e.preventDefault();
  let name = this.elements.name.value;
  let email = this.elements.email.value;
  if (!name || !email) {
    document.getElementById("errorMsg").textContent = "Please fill all fields!";
  } else {
    alert(`Registered: ${name}`);
  }
});
</script>
```

12. <u>**AJAX & Fetch API Scenario:**</u>

Send user registration to the server.

Objective: Simulate backend communication.

Task:

• Use fetch() to POST user data to a mock API

• Show success/failure message after submission

• Use setTimeout() to simulate a delayed response

```
const user = { name: "Alex", email: "a@mail.com" };
```

```
setTimeout(() => {
  fetch("https://mockapi.io/endpoint", {
    method: "POST",
    headers: { "Content-Type": "application/json" },
    body: JSON.stringify(user)
  })
    .then(res => res.json())
    .then(data => console.log("Success", data))
    .catch(err => console.error("Fail", err));
}, 1000);
```

O/P:

```
Success { id: 1, name: "Alex", ... }
```

13. <u>jQuery and JS Frameworks Scenario:</u>

Use jQuery to simplify DOM tasks.

Objective: Understand and use jQuery.

Task: •

Use $('#registerBtn').click(...) to handle click events

• Use .fadeIn() and .fadeOut() for event cards

• Mention one benefit of moving to frameworks like React or Vue

```
$("#registerBtn").click(() => alert("Registered via jQuery!"));
$(".eventCard").fadeOut().fadeIn();
```

Benefit of React/Vue:

- Efficient UI updates through Virtual DOM (React) or reactive bindings (Vue)