Q1. Responsive Card Layout with CSS Flexbox

Here's a responsive card layout using CSS Flexbox. This design includes a title, an image, and a description. The layout adjusts for different screen sizes, stacking vertically on smaller screens and arranging horizontally on larger ones.

HTML (index.html)

HTML

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Responsive Card Layout</title>
    <link rel="stylesheet" href="style.css">
</head>
<body>
    <div class="card-container">
        <div class="card">
            <img src="https://via.placeholder.com/300" alt="Card Image" class="card-image">
            <div class="card-content">
                <h2 class="card-title">Card Title 1</h2>
                <p class="card-description">This is a description for the first card. It provides some details about the content.</p>
            </div>
        </div>
        <div class="card">
            <img src="https://via.placeholder.com/300" alt="Card Image" class="card-image">
            <div class="card-content">
                <h2 class="card-title">Card Title 2</h2>
                <p class="card-description">This is a description for the second card. It provides some details about the content.</p>
            </div>
        </div>
        <div class="card">
```

```
        <img src="https://via.placeholder.com/300" alt="Card Image" class="card-image">
        <div class="card-content">
            <h2 class="card-title">Card Title 3</h2>
            <p class="card-description">This is a description for the third card. It provides some details about the content.</p>
        </div>
    </div>
  </div>
</body>
</html>
```

CSS (style.css)

CSS

```
body {
  font-family: Arial, sans-serif;
  background-color: #f0f2f5;
  margin: 0;
  padding: 20px;
}

.card-container {
  display: flex;
  flex-wrap: wrap;
  gap: 20px;
  justify-content: center;
}

.card {
  background-color: #fff;
  border-radius: 8px;
  box-shadow: 0 4px 8px rgba(0, 0, 0, 0.1);
  overflow: hidden;
  width: 100%;
  max-width: 300px;
  display: flex;
  flex-direction: column;
  transition: transform 0.3s ease;
}
```

```css
.card:hover {
    transform: translateY(-5px);
}

.card-image {
    width: 100%;
    height: auto;
    display: block;
}

.card-content {
    padding: 16px;
}

.card-title {
    margin-top: 0;
    font-size: 1.5em;
    color: #333;
}

.card-description {
    color: #666;
    font-size: 1em;
}

@media (min-width: 768px) {
    .card {
        width: calc(33.333% - 20px);
    }
}
```

Explanation:

- The **.card-container** uses display: flex and flex-wrap: wrap to allow cards to wrap to the next line. gap: 20px creates space between them.
- Each **.card** is a flex container with flex-direction: column to stack its content (image and text) vertically.
- The @media query ensures that on screens wider than 768px, the cards occupy calc(33.333% - 20px) of the container's width, creating a three-column layout. On smaller screens, they take up 100% width, stacking vertically.

Q2. Filtering an Array of Objects & Closures

Part A: Filtering an Array of Objects

You can filter an array of objects in JavaScript using the **Array.prototype.filter()** method. This method creates a new array with all elements that pass the test implemented by the provided function.

Example: Filtering books by a specific property
Let's filter an array of book objects to find all books published after the year 2000.

JavaScript

```
const books = [
  { title: 'Book A', author: 'Author 1', year: 1998 },
  { title: 'Book B', author: 'Author 2', year: 2005 },
  { title: 'Book C', author: 'Author 3', year: 2010 },
  { title: 'Book D', author: 'Author 4', year: 1995 },
];

// Function to filter books by a specific year
function filterBooksByYear(bookArray, year) {
  return bookArray.filter(book => book.year > year);
}

const modernBooks = filterBooksByYear(books, 2000);

console.log(modernBooks);
/*
Output:
[
  { title: 'Book B', author: 'Author 2', year: 2005 },
  { title: 'Book C', author: 'Author 3', year: 2010 }
]
*/
```

Explanation:

- The filterBooksByYear function takes an array and a year as arguments.
- The filter() method iterates over each book object in the array.
- The **arrow function book => book.year > year** is the test condition. It returns true for books published after the specified year, and false otherwise.
- filter() returns a **new array** containing only the objects that met the condition.

Part B: The Concept of Closures in JavaScript

A **closure** is the combination of a function and the lexical environment within which that function was declared. In simpler terms, a closure gives you access to an outer function's scope from an inner function. A closure "remembers" the variables and arguments of the outer function even after the outer function has finished executing.

Example of a Closure

JavaScript

```
function makeGreeter(greeting) {
  // `greeting` is a variable in the outer function's scope
  return function(name) {
    // This inner function "closes over" the `greeting` variable
    console.log(`${greeting}, ${name}!`);
  };
}

const greetHello = makeGreeter('Hello');
const greetHi = makeGreeter('Hi');

greetHello('Alice'); // Output: Hello, Alice!
greetHi('Bob');     // Output: Hi, Bob!
```

Explanation:

1. The **makeGreeter** function is called, and it returns a new, anonymous function.
2. The returned function "remembers" the greeting variable from its parent scope, even though makeGreeter has already finished running.

3. When we call **greetHello('Alice')**, the anonymous function still has access to the greeting value 'Hello'.
4. Each call to makeGreeter creates a new closure, each with its own independent greeting variable. This is why greetHello and greetHi have different greeting values.

Q3. HTML Form with CSS Styling

Here is a simple HTML login form with username, password, and a "Remember me" checkbox, styled with CSS.

HTML (form.html)

HTML

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Login Form</title>
    <link rel="stylesheet" href="form-style.css">
</head>
<body>
    <div class="login-container">
        <form class="login-form">
            <h2>Login</h2>
            <div class="form-group">
                <label for="username">Username</label>
                <input type="text" id="username" name="username" required>
            </div>
            <div class="form-group">
                <label for="password">Password</label>
                <input type="password" id="password" name="password" required>
            </div>
            <div class="form-group-checkbox">
                <input type="checkbox" id="rememberMe" name="rememberMe">
                <label for="rememberMe">Remember me</label>
            </div>
```

```
        <button type="submit" class="login-button">Sign In</button>
    </form>
  </div>
</body>
</html>
```

CSS (form-style.css)

CSS

```
body {
  font-family: Arial, sans-serif;
  background-color: #f4f4f9;
  display: flex;
  justify-content: center;
  align-items: center;
  height: 100vh;
  margin: 0;
}

.login-container {
  background-color: #fff;
  padding: 40px;
  border-radius: 10px;
  box-shadow: 0 4px 10px rgba(0, 0, 0, 0.1);
  width: 100%;
  max-width: 400px;
  text-align: center;
}

.login-form h2 {
  margin-bottom: 25px;
  color: #333;
}

.form-group {
  margin-bottom: 20px;
  text-align: left;
}
```

```css
.form-group-checkbox {
    margin-bottom: 20px;
    text-align: left;
    display: flex;
    align-items: center;
}

.form-group label {
    display: block;
    margin-bottom: 8px;
    font-weight: bold;
    color: #555;
}

.form-group input[type="text"],
.form-group input[type="password"] {
    width: 100%;
    padding: 10px;
    border: 1px solid #ddd;
    border-radius: 5px;
    box-sizing: border-box;
    font-size: 16px;
}

.form-group input[type="text"]:focus,
.form-group input[type="password"]:focus {
    border-color: #007bff;
    outline: none;
    box-shadow: 0 0 5px rgba(0, 123, 255, 0.25);
}

.form-group-checkbox input[type="checkbox"] {
    margin-right: 10px;
}

.login-button {
    width: 100%;
    padding: 12px;
    background-color: #007bff;
    color: #fff;
    border: none;
    border-radius: 5px;
    font-size: 1em;
```

```
    cursor: pointer;
    transition: background-color 0.3s ease;
}

.login-button:hover {
    background-color: #0056b3;
}
```

Q4. The name Attribute in Form Submission

The **name attribute** is crucial for form submission because it acts as the identifier for the data that a user enters. When a form is submitted, the data is sent to the server as key-value pairs, where the **name attribute of a form element becomes the key**, and the user's input becomes the value.

Role in Form Submission:

- It creates a **key-value pair** for each form control (like an input, select, or textarea).
- Without a name attribute, a form field's data **will not be sent** to the server during submission.

Role in Server-Side Processing:

- On the server side, the submitted data is received as a collection (e.g., an associative array or a dictionary).
- The server-side script (e.g., PHP, Node.js, Python) uses the **name** as the **variable name or key** to access the corresponding value. This makes it possible to retrieve and process the user's input.

Example
Let's consider a simple HTML form and a hypothetical PHP script for processing.
HTML (contact.html)

```
HTML
```

```
<form action="submit_form.php" method="POST">
    <label for="user_name">Name:</label>
    <input type="text" id="user_name" name="name">
```

```
    <label for="user_email">Email:</label>
    <input type="email" id="user_email" name="email">

    <button type="submit">Submit</button>
</form>
```

PHP Server-Side Processing (submit_form.php)

PHP

```
<?php
// Check if the form was submitted using the POST method
if ($_SERVER["REQUEST_METHOD"] == "POST") {
    // Access the submitted data using the 'name' attributes
    $userName = $_POST['name'];
    $userEmail = $_POST['email'];

    echo "Hello, " . htmlspecialchars($userName) . "!<br>";
    echo "Your email is: " . htmlspecialchars($userEmail);
}
?>
```

Explanation:

- In the HTML form, the name attribute of the first input is name, and the second is email.
- When the form is submitted, the browser sends the data as name=...&email=... to the server.
- The PHP script uses **$_POST['name']** and **$_POST['email']** to access the submitted values. The string inside the brackets ('name', 'email') must exactly match the **name** attribute from the HTML form.

Q5. Promises in JavaScript

A **Promise** in JavaScript is an object that represents the eventual completion (or failure) of an asynchronous operation and its resulting value. It's a way to handle asynchronous code more cleanly than with traditional callbacks. A Promise can be in one of three states:

1. **pending**: The initial state, neither fulfilled nor rejected.

2. **fulfilled**: The operation completed successfully.
3. **rejected**: The operation failed.

Key Methods:

- **.then()**: A method that is called when the Promise is fulfilled. It takes a callback function to handle the successful result.
- **.catch()**: A method that is called when the Promise is rejected. It takes a callback function to handle the error.
- **.finally()**: A method that is called when the Promise is either fulfilled or rejected.

Function to Simulate Data Fetching with a Delay
This function uses setTimeout to simulate a network request that resolves after a specified delay. It returns a new Promise.

JavaScript

```
function fetchDataWithDelay(delayInMs, shouldSucceed = true) {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      if (shouldSucceed) {
        const data = { message: 'Data fetched successfully!', id: 123 };
        resolve(data); // Resolve the promise with the data
      } else {
        const error = new Error('Failed to fetch data!');
        reject(error); // Reject the promise with an error
      }
    }, delayInMs);
  });
}

// Example usage: Successful fetch
console.log("Fetching data...");
fetchDataWithDelay(2000)
  .then(data => {
    console.log("Success:", data.message);
  })
  .catch(error => {
    console.error("Error:", error.message);
  })
  .finally(() => {
    console.log("Fetch operation complete.");
```

```
  });

// Example usage: Failed fetch
console.log("\nAttempting to fetch data that will fail...");
fetchDataWithDelay(2000, false)
  .then(data => {
    console.log("Success:", data.message);
  })
  .catch(error => {
    console.error("Error:", error.message);
  })
  .finally(() => {
    console.log("Failed fetch operation complete.");
  });
```

Q6. CSS position Property

The **position property** in CSS controls how an element is positioned on a web page. It works in conjunction with the top, right, bottom, and left properties to determine the element's final location.

position: static

- **Default value** for all HTML elements.
- An element with position: static is positioned according to the normal document flow.
- The top, right, bottom, and left properties have **no effect** on a static element.

Example:

CSS

```
.static-box {
    position: static;
    top: 50px; /* This will be ignored */
```

```
  background-color: lightblue;
}
```

position: relative

- The element is positioned relative to its **normal position**.
- Using top, right, bottom, or left will shift the element from where it would normally be.
- The space the element would normally occupy is **preserved**, and other elements do not move to fill that space.

Example:

CSS

```
.relative-box {
  position: relative;
  top: 20px;
  left: 30px;
  background-color: lightgreen;
}
```

position: absolute

- The element is removed from the normal document flow.
- It is positioned relative to its **closest positioned ancestor** (an ancestor with position: relative, absolute, fixed, or sticky).
- If no positioned ancestor exists, it is positioned relative to the initial containing block (the <html> element).
- The space it would normally occupy is **not preserved**, allowing other elements to move into its space.

Example:

CSS

```css
.parent {
  position: relative; /* This is the positioning context */
  width: 200px;
  height: 200px;
  border: 2px solid black;
}
.absolute-child {
  position: absolute;
  bottom: 10px;
  right: 10px;
  background-color: lightcoral;
}
```

position: fixed

- The element is removed from the normal document flow.
- It is positioned relative to the **viewport** (the browser window).
- It **stays in the same place** even when the page is scrolled.
- Often used for navigation bars or social media buttons that should always be visible.

Example:

CSS

```css
.fixed-nav {
  position: fixed;
  top: 0;
  left: 0;
  width: 100%;
  background-color: #333;
  color: white;
}
```

position: sticky

- The element behaves like relative until a certain scroll position is met.
- Once that position is reached, it becomes "stuck" and acts like a fixed element.
- It requires a top, right, bottom, or left property to determine where it should stick.

Example:

CSS

```css
.sticky-header {
  position: sticky;
  top: 0;
  background-color: lightgray;
  padding: 10px;
}
```

As the user scrolls, this element will remain at the top of the viewport.

Q7. To-Do List with DOM Manipulation

This JavaScript function implements a simple to-do list with "add," "delete," and "mark as completed" functionality. It demonstrates **DOM manipulation** by dynamically creating, modifying, and removing HTML elements.

HTML (todo.html)

HTML

```html
<!DOCTYPE html>
<html lang="en">
```

```
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Simple To-Do List</title>
    <link rel="stylesheet" href="todo.css">
</head>
<body>
    <div class="container">
        <h1>My To-Do List</h1>
        <div class="input-container">
            <input type="text" id="todoInput" placeholder="Add a new task...">
            <button id="addBtn">Add</button>
        </div>
        <ul id="todoList"></ul>
    </div>
    <script src="todo.js"></script>
</body>
</html>
```

JavaScript (todo.js)

JavaScript

```
document.addEventListener('DOMContentLoaded', () => {
    const todoInput = document.getElementById('todoInput');
    const addBtn = document.getElementById('addBtn');
    const todoList = document.getElementById('todoList');

    addBtn.addEventListener('click', addTask);
    todoInput.addEventListener('keypress', (e) => {
        if (e.key === 'Enter') {
            addTask();
        }
    });

    function addTask() {
        const taskText = todoInput.value.trim();
        if (taskText === "") {
            return;
        }
```

```
        // 1. Create new elements
        const li = document.createElement('li');
        const span = document.createElement('span');
        const deleteBtn = document.createElement('button');
        const completeBtn = document.createElement('button');

        // 2. Set content and attributes
        span.textContent = taskText;
        deleteBtn.textContent = 'Delete';
        completeBtn.textContent = 'Complete';
        deleteBtn.classList.add('delete-btn');
        completeBtn.classList.add('complete-btn');

        // 3. Add event listeners
        deleteBtn.addEventListener('click', () => {
            // Remove the parent <li> element
            todoList.removeChild(li);
        });

        completeBtn.addEventListener('click', () => {
            // Toggle a class for styling
            li.classList.toggle('completed');
        });

        // 4. Append elements to the DOM
        li.appendChild(span);
        li.appendChild(completeBtn);
        li.appendChild(deleteBtn);
        todoList.appendChild(li);

        // 5. Clear the input field
        todoInput.value = '';
    }
});
```

Explanation of DOM Manipulation:

- **document.createElement()**: We use this method to **create new HTML elements** (<li>, , <button>) in memory. They are not yet visible on the page.
- **element.textContent**: We set the text content of the and <button> elements.
- **element.classList.add()**: We add CSS classes to the buttons for styling.
- **element.addEventListener()**: We attach event listeners to the buttons to handle user

interactions.

- **element.removeChild()**: When the delete button is clicked, we find its parent <li> element and remove it from the <ul> element, effectively deleting the task from the list.
- **element.classList.toggle()**: This method adds a class if it's not present or removes it if it is, which is perfect for toggling the "completed" state and applying a strikethrough style.
- **element.appendChild()**: We use this to insert the newly created and <button> elements **as children** of the <li> element.
- **todoList.appendChild(li)**: Finally, we insert the completed <li> element into the <ul> list, making it visible on the page.

Q8. Responsive Navigation Bar

Here's how to design a responsive navigation bar using HTML and CSS. The navigation bar will collapse into a hamburger menu on smaller screens, which is a common pattern for mobile-friendly designs.

HTML (nav.html)

HTML

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Responsive Navigation</title>
    <link rel="stylesheet" href="nav-style.css">
    <script src="nav-script.js" defer></script>
</head>
<body>
    <header class="navbar">
        <a href="#" class="logo">Logo</a>
        <button class="hamburger-menu" aria-label="Toggle navigation menu">
            <span class="bar"></span>
            <span class="bar"></span>
            <span class="bar"></span>
        </button>
        <nav class="nav-links">
```

```
        <a href="#">Home</a>
        <a href="#">About</a>
        <a href="#">Services</a>
        <a href="#">Contact</a>
      </nav>
    </header>
</body>
</html>
```

CSS (nav-style.css)

CSS

```
body {
    font-family: Arial, sans-serif;
    margin: 0;
}

.navbar {
    display: flex;
    justify-content: space-between;
    align-items: center;
    background-color: #333;
    color: white;
    padding: 1rem 2rem;
}

.logo {
    font-size: 1.5rem;
    font-weight: bold;
    color: white;
    text-decoration: none;
}

.nav-links {
    display: flex;
    gap: 1.5rem;
}

.nav-links a {
```

```
    color: white;
    text-decoration: none;
    font-size: 1rem;
    transition: color 0.3s ease;
}

.nav-links a:hover {
    color: #ffd700;
}

.hamburger-menu {
    display: none; /* Hidden by default on large screens */
    flex-direction: column;
    justify-content: space-around;
    width: 25px;
    height: 20px;
    background: transparent;
    border: none;
    cursor: pointer;
    padding: 0;
}

.hamburger-menu .bar {
    width: 100%;
    height: 3px;
    background-color: white;
    transition: transform 0.3s ease;
}

/* Mobile styles */
@media (max-width: 768px) {
    .navbar {
        flex-direction: column;
        align-items: flex-start;
    }

    .hamburger-menu {
        display: flex;
        position: absolute;
        top: 1rem;
        right: 2rem;
    }
```

```
    .nav-links {
        display: none; /* Hide nav links by default on mobile */
        flex-direction: column;
        width: 100%;
        text-align: center;
        margin-top: 1rem;
    }

    .nav-links.active {
        display: flex; /* Show nav links when 'active' class is added */
    }

    .nav-links a {
        padding: 10px 0;
        border-bottom: 1px solid #444;
    }
}
```

JavaScript (nav-script.js)

```
JavaScript
```

```
const hamburgerMenu = document.querySelector('.hamburger-menu');
const navLinks = document.querySelector('.nav-links');

hamburgerMenu.addEventListener('click', () => {
    navLinks.classList.toggle('active');
});
```

Explanation:

- The **.navbar** uses **Flexbox (display: flex)** to align the logo, hamburger menu, and navigation links.
- The **.nav-links** are displayed horizontally on larger screens.
- The **.hamburger-menu** is hidden on desktop using display: none.
- The **@media (max-width: 768px)** query is the key to responsiveness.
    - It changes the .navbar to a column layout.
    - It **shows the .hamburger-menu** (display: flex) and **hides the .nav-links** (display: none).
    - The JavaScript code listens for a click on the hamburger button and **toggles the**

.active class on the nav-links.
- The nav-links.active CSS rule then sets display: flex, making the menu visible.

Q9. How Flexbox Improves Layout Alignment

Flexbox (Flexible Box Layout) is a one-dimensional layout model in CSS designed for arranging items in a single row or column. It provides a powerful and efficient way to align and distribute space among items in a container, regardless of their size or the viewport.

How it Improves Layout Alignment:

1. **Simplifies Centering:** Flexbox makes it incredibly easy to center items, both horizontally and vertically, with just a few properties.
   - align-items: center; (vertical alignment)
   - justify-content: center; (horizontal alignment)
2. **Flexible Sizing:** It allows items to grow (flex-grow) or shrink (flex-shrink) to fill available space, ensuring a responsive design.
3. **Equal Spacing:** Properties like justify-content: space-between or space-around automatically distribute space between items, eliminating the need for complex margin calculations.
4. **Direction Control:** The flex-direction property lets you easily switch between a row (row) and a column (column) layout without changing the HTML structure.
5. **Order Control:** The order property allows you to change the visual order of items without affecting their order in the source HTML, which is great for accessibility and mobile-first design.

Example:
Imagine you want to center three boxes horizontally and vertically inside a container.
Flexbox Solution:

HTML

```
<div class="flex-container">
  <div class="box">1</div>
  <div class="box">2</div>
  <div class="box">3</div>
</div>
```

CSS

```css
.flex-container {
  display: flex;
  height: 200px;
  border: 2px solid #ccc;
  /* Center horizontally */
  justify-content: center;
  /* Center vertically */
  align-items: center;
}
.box {
  width: 50px;
  height: 50px;
  background-color: dodgerblue;
  color: white;
  margin: 5px;
}
```

Traditional (Non-Flexbox) Approach:
This would require a combination of techniques like display: table-cell with vertical-align, or using position: absolute with transformations, which are more complex and less maintainable. Flexbox provides a single, clear, and modern solution for these common layout problems.

Q10. Login Form Validation (Check Empty Fields)

This JavaScript program validates a login form to ensure that the username and password fields are not empty before submission.

HTML (login.html)

HTML

```html
<!DOCTYPE html>
```

```html
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Login Validation</title>
</head>
<body>
    <form id="loginForm">
        <h2>Login</h2>
        <div class="form-group">
            <label for="username">Username:</label>
            <input type="text" id="username" name="username">
            <small class="error-message" id="username-error"></small>
        </div>
        <div class="form-group">
            <label for="password">Password:</label>
            <input type="password" id="password" name="password">
            <small class="error-message" id="password-error"></small>
        </div>
        <button type="submit">Submit</button>
    </form>
    <script src="validate.js"></script>
</body>
</html>
```

JavaScript (validate.js)

JavaScript

```
document.addEventListener('DOMContentLoaded', () => {
    const loginForm = document.getElementById('loginForm');
    const usernameInput = document.getElementById('username');
    const passwordInput = document.getElementById('password');
    const usernameError = document.getElementById('username-error');
    const passwordError = document.getElementById('password-error');

    loginForm.addEventListener('submit', function(event) {
        let isValid = true;

        // Clear previous error messages
```

```
    usernameError.textContent = '';
    passwordError.textContent = '';

    // Check if username is empty
    if (usernameInput.value.trim() === '') {
      usernameError.textContent = 'Username is required.';
      isValid = false;
    }

    // Check if password is empty
    if (passwordInput.value.trim() === '') {
      passwordError.textContent = 'Password is required.';
      isValid = false;
    }

    // If not valid, prevent form submission
    if (!isValid) {
      event.preventDefault();
    }
  });
});
```

Explanation:

- The JavaScript code listens for the **submit event** on the form.
- The **event.preventDefault()** method is crucial. It stops the form from being submitted to the server immediately, allowing the JavaScript code to run its validation checks first.
- **.value.trim()** is used to get the input value and remove any leading or trailing whitespace.
- If the isValid flag is set to false, the form submission is prevented, and an error message is displayed next to the respective input field.

Q11. How Event Handling Works in JavaScript

Event handling is the process by which JavaScript responds to actions and events that occur on a web page, such as a user clicking a button, a page loading, or a key being pressed.

The Core Concepts:

1. **Events:** These are actions or occurrences that happen in the browser. Examples include click, mouseover, submit, load, and change.
2. **Event Listeners:** A function that "listens" for a specific event to occur on a target

element. When the event happens, the listener's callback function is executed.

3. **Event Object:** When an event is triggered, a special object is created and passed to the event handler function. This object contains useful information about the event, like the target element (event.target) or the type of event (event.type).

The Mechanism:

- **Binding a Listener:** You use the **element.addEventListener(event, handler)** method to attach a listener to an element.
- **Event Loop:** JavaScript has a single-threaded event loop. When an event occurs, it's placed in an event queue. The event loop continuously checks this queue and, when the call stack is empty, it processes the next event and executes its handler.
- **Event Propagation (Bubbling & Capturing):**
  - **Bubbling:** The default behavior. The event first triggers on the element it happened on, then "bubbles up" to its parent, and its parent's parent, and so on, all the way to the <html> and document objects.
  - **Capturing:** The opposite of bubbling. The event first triggers on the outermost element (document) and travels down to the target element. You can enable this by passing a third argument (true) to addEventListener.

Example:

JavaScript

```
const myButton = document.getElementById('myButton');

// Add a click event listener to the button
myButton.addEventListener('click', function(event) {
  console.log('Button was clicked!');
  console.log('Event target:', event.target);

  // Prevent the default action (e.g., a link from navigating)
  event.preventDefault();
});
```

In this example, when the user clicks the button, the anonymous function (the event handler) is executed, logging a message to the console. The event object provides details about the interaction.

Q12. HTML Form with CSS Only Styling

Here is a simple HTML form with fields for name and email, and a submit button. The styling is done exclusively with CSS.

HTML (form-css-only.html)

HTML

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>CSS-Only Form</title>
    <link rel="stylesheet" href="form-styles-2.css">
</head>
<body>
    <div class="form-container">
        <form class="simple-form">
            <h2>Contact Us</h2>
            <div class="form-field">
                <label for="name">Name</label>
                <input type="text" id="name" name="name" required>
            </div>
            <div class="form-field">
                <label for="email">Email</label>
                <input type="email" id="email" name="email" required>
            </div>
            <button type="submit" class="submit-btn">Send</button>
        </form>
    </div>
</body>
</html>
```

CSS (form-styles-2.css)

CSS

```css
body {
    font-family: 'Segoe UI', Tahoma, Geneva, Verdana, sans-serif;
    background-color: #e9ecef;
    display: flex;
    justify-content: center;
    align-items: center;
    min-height: 100vh;
    margin: 0;
}

.form-container {
    background-color: #fff;
    padding: 30px;
    border-radius: 8px;
    box-shadow: 0 5px 15px rgba(0, 0, 0, 0.1);
    width: 100%;
    max-width: 450px;
}

.simple-form h2 {
    text-align: center;
    color: #495057;
    margin-bottom: 25px;
}

.form-field {
    margin-bottom: 20px;
}

.form-field label {
    display: block;
    font-weight: 600;
    margin-bottom: 8px;
    color: #495057;
}

.form-field input {
    width: 100%;
    padding: 12px;
```

```
    border: 1px solid #ced4da;
    border-radius: 4px;
    box-sizing: border-box;
    font-size: 16px;
    transition: border-color 0.3s;
}

.form-field input:focus {
    border-color: #007bff;
    outline: none;
    box-shadow: 0 0 0 0.2rem rgba(0, 123, 255, 0.25);
}

.submit-btn {
    width: 100%;
    padding: 12px;
    background-color: #007bff;
    color: #fff;
    border: none;
    border-radius: 4px;
    font-size: 1em;
    font-weight: 600;
    cursor: pointer;
    transition: background-color 0.3s;
}

.submit-btn:hover {
    background-color: #0056b3;
}
```

Q13. Importance of Semantic HTML in Forms

Semantic HTML is the use of HTML tags to convey meaning and structure, not just presentation. For forms, using semantic tags like <form>, <label>, <input>, <button>, <fieldset>, and <legend> is crucial for several reasons:

1. **Accessibility:** Semantic tags make forms accessible to users with disabilities who rely on assistive technologies like screen readers. A screen reader can interpret a <label> tag and announce it in a way that clearly associates it with its corresponding <input> field, enabling the user to understand what information is required.

2. **SEO (Search Engine Optimization):** Search engines use semantic tags to understand the content and structure of a page. Properly structured forms can help search engines index the page correctly, even if the form is part of a larger content area.
3. **Improved Usability:** Clear, logical structure makes the form easier for **all users** to understand and complete. The <label> tag, for instance, provides a larger clickable area for its associated input, improving the user experience on touch devices.
4. **Maintainability and Readability:** Semantic HTML code is easier for other developers to read and understand. When you see a <label> and an <input>, you instantly know what their relationship is, which is far better than relying on generic tags with custom classes.
5. **Built-in Browser Features:** Browsers have built-in styling and behavior for semantic elements. For example, clicking a <label> automatically puts the cursor in its linked <input> field, a behavior that would require extra JavaScript to replicate with non-semantic tags.

Example:
Semantic Form

HTML

```
<form action="/submit-data">
  <label for="email">Email Address:</label>
  <input type="email" id="email" name="email">
</form>
```

Non-Semantic Equivalent

HTML

```
<div class="form-container">
  <p class="form-label">Email Address:</p>
  <div class="form-input-box">
    <input type="text" class="email-field" id="email" name="email">
  </div>
</div>
```

The semantic version is much cleaner, more accessible, and more robust.

Q14. Asynchronous Programming & Data Fetching

Asynchronous programming is a programming paradigm that allows a program to start a long-running task and then continue executing other tasks without waiting for the long-running task to complete. In web development, this is essential for tasks like:

- **Fetching data** from an API.
- **Reading files** from a server.
- **Handling user input** and animations without freezing the UI.

In JavaScript, asynchronous behavior is handled primarily using **Promises** and the async/await syntax, which is built on top of Promises.

async/await is syntactic sugar that makes asynchronous code look and feel like synchronous code, making it easier to read and debug.

- An **async function** is a function that always returns a Promise.
- The **await keyword** can only be used inside an async function. It pauses the execution of the async function until the Promise it's waiting for is resolved or rejected.

Function to Fetch and Display Data from an API
This function uses the fetch() API and async/await to get data from a public API and display it.

JavaScript

```
// URL for a public API (JSONPlaceholder)
const API_URL = 'https://jsonplaceholder.typicode.com/posts/1';

async function fetchAndDisplayData() {
    try {
        console.log('Fetching data...');
        // Use await to wait for the fetch request to complete
        const response = await fetch(API_URL);

        // Check if the response was successful (status code 200-299)
        if (!response.ok) {
            throw new Error(`HTTP error! status: ${response.status}`);
        }
```

```
        // Use await to parse the JSON data
        const data = await response.json();

        // Display the fetched data on the page
        const outputDiv = document.createElement('div');
        outputDiv.innerHTML = `
            <h3>Title: ${data.title}</h3>
            <p>Body: ${data.body}</p>
        `;
        document.body.appendChild(outputDiv);

        console.log('Data fetched and displayed successfully!');
    } catch (error) {
        // Handle any errors that occurred during the fetch
        console.error('There was a problem with the fetch operation:', error);
    }
}

// Call the async function
fetchAndDisplayData();
```

Explanation:

- The fetchAndDisplayData function is declared as async.
- **await fetch(API_URL)** pauses the function until the network request completes and a Response object is returned.
- **await response.json()** pauses again until the data from the response body is fully parsed as a JSON object.
- The try...catch block gracefully handles any potential errors (e.g., network issues, invalid URL, or non-200 HTTP status codes).

Q15. CSS Grid with a Practical Layout Example

CSS Grid Layout is a two-dimensional layout system that allows you to design web pages with rows and columns. Unlike Flexbox, which is one-dimensional, Grid is ideal for creating complex, multi-column layouts and overall page structures.

Key Concepts:

- **Grid Container:** The element on which display: grid is applied.
- **Grid Items:** The direct children of the grid container.
- **Grid Lines, Tracks, and Cells:** The lines that form the grid, the space between the lines

(rows and columns), and the individual boxes created by the intersection of rows and columns.

Practical Layout Example: A Simple Dashboard
This example creates a responsive dashboard with a header, sidebar, and main content area.
HTML (grid-layout.html)

HTML

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>CSS Grid Layout Example</title>
    <link rel="stylesheet" href="grid-style.css">
</head>
<body>
    <div class="grid-container">
        <header class="header">Header</header>
        <aside class="sidebar">Sidebar</aside>
        <main class="main-content">Main Content</main>
        <footer class="footer">Footer</footer>
    </div>
</body>
</html>
```

CSS (grid-style.css)

CSS

```
body {
    font-family: Arial, sans-serif;
    margin: 0;
    height: 100vh;
}
```

```css
.grid-container {
    display: grid;
    height: 100%;
    gap: 10px;
    padding: 10px;
    background-color: #f0f2f5;

    /* Define grid structure for larger screens */
    grid-template-areas:
        "header header"
        "sidebar main"
        "footer footer";
    grid-template-columns: 200px 1fr; /* 200px for sidebar, rest for main */
    grid-template-rows: auto 1fr auto;
}

.header { grid-area: header; background-color: #3498db; }
.sidebar { grid-area: sidebar; background-color: #2ecc71; }
.main-content { grid-area: main; background-color: #ecf0f1; }
.footer { grid-area: footer; background-color: #34495e; }

.grid-container > div {
    padding: 20px;
    color: white;
    font-size: 1.2rem;
    display: flex;
    justify-content: center;
    align-items: center;
    border-radius: 8px;
}
.main-content { color: #333; }

/* Responsive design for small screens */
@media (max-width: 768px) {
    .grid-container {
        /* Redefine grid for a single column layout */
        grid-template-areas:
            "header"
            "sidebar"
            "main"
            "footer";
        grid-template-columns: 1fr;
        grid-template-rows: auto auto 1fr auto;
```

```
  }
}
```

Explanation:

- The **grid-template-areas** property is used to give names to areas of the grid, making the layout structure easy to visualize.
- **grid-template-columns** defines the width of each column. 1fr means "one fraction" of the available space, which makes the main content area flexible.
- The @media query demonstrates how to create a responsive layout by simply redefining the grid-template-areas and grid-template-columns for smaller screens. The sidebar and main content now stack on top of each other.

Q16. Dynamically Adding and Removing List Items

This JavaScript code snippet demonstrates how to dynamically add and remove list items (<li>) from an unordered list (<ul>). This is a classic example of **DOM manipulation**.

HTML (dynamic-list.html)

HTML

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Dynamic List</title>
</head>
<body>
  <div class="list-container">
    <input type="text" id="itemInput" placeholder="New item...">
    <button id="addBtn">Add Item</button>
    <ul id="itemList"></ul>
  </div>
  <script src="list-script.js"></script>
</body>
</html>
```

JavaScript (list-script.js)

JavaScript

```
document.addEventListener('DOMContentLoaded', () => {
    const itemInput = document.getElementById('itemInput');
    const addBtn = document.getElementById('addBtn');
    const itemList = document.getElementById('itemList');

    // Add item function
    function addItem() {
        const itemText = itemInput.value.trim();
        if (itemText === '') {
            return;
        }

        // 1. Create the new list item element
        const newListItem = document.createElement('li');
        newListItem.textContent = itemText;

        // 2. Create the remove button
        const removeButton = document.createElement('button');
        removeButton.textContent = 'Remove';
        removeButton.style.marginLeft = '10px';

        // 3. Add an event listener to the remove button
        removeButton.addEventListener('click', function() {
            // Technique 1: Using parentNode
            // this.parentNode.remove();

            // Technique 2: Using remove() on the element itself
            newListItem.remove();
        });

        // 4. Append the button to the list item
        newListItem.appendChild(removeButton);

        // 5. Append the list item to the unordered list
        itemList.appendChild(newListItem);
```

```
        // 6. Clear the input field
        itemInput.value = '';
    }

    addBtn.addEventListener('click', addItem);
    itemInput.addEventListener('keypress', (e) => {
        if (e.key === 'Enter') {
            addItem();
        }
    });
});
```

Techniques for DOM Manipulation:

1. **Adding Items (appendChild()):**
    - The **document.createElement('li')** method creates a new <li> node.
    - We set its text content and add a button.
    - The **itemList.appendChild(newListItem)** method then inserts the newly created <li> element as the last child of the <ul> element, making it appear on the page.
2. **Removing Items (element.remove() or parentNode.removeChild()):**
    - **element.remove()**: This is the modern, simpler way to remove an element. It removes the element directly from the DOM tree. newListItem.remove() removes the list item itself.
    - **parentNode.removeChild(child)**: This is an older but still valid technique. It involves selecting the parent of the element you want to remove (this.parentNode in the code, where this refers to the button) and calling removeChild on it, passing the child element as an argument.