Class 5: Interactive Cart Page with DOM Manipulation

Session Overview

Class Topics:

- DOM Manipulation
- Event Handling
- Form Validation

Learning Objectives: By the end of this session, students will be able to:

1. Create an interactive shopping cart page using HTML and CSS.
2. Use JavaScript for dynamic content updates and event handling.
3. Implement form validation.
4. Understand the integration between front-end design and dynamic functionality.

Session

DOM Manipulation

Overview

This example demonstrates a shopping cart page where items are added from the product page. The user can adjust item quantities, remove items, and view the total cost. The quantities can be adjusted from both the product page and the cart page. Responsive design is applied to ensure the layout adapts to different screen sizes.

HTML Structure

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Shopping Cart</title>
    <link rel="stylesheet" href="styles.css">
</head>
<body>
    <div class="cart-container">
        <h1 class="cart-title">Shopping Cart</h1>
        <div class="cart-items-container" id="cart-items">
            <!-- Cart items will be appended here -->
        </div>
        <div class="cart-summary">
            <p id="cart-total">Total: $0.00</p>
            <button class="cart-checkout-button"
id="checkout-button">Checkout</button>
        </div>
    </div>
    <script>
document.addEventListener('DOMContentLoaded', () => {
    const cartItemsContainer = document.getElementById('cart-items');
    const cartTotalElement = document.getElementById('cart-total');
    const checkoutButton = document.getElementById('checkout-button');

    // Function to load cart items from localStorage and display them in the
cart
    function loadCartItems() {
        const cart = JSON.parse(localStorage.getItem('cart')) || [];
        if (cart.length > 0) {
            cartItemsContainer.innerHTML = ''; // Clear any existing items

            cart.forEach((product, index) => {
                const cartItem = document.createElement('div');
                cartItem.className = 'cart-item';

                cartItem.innerHTML = `
                    <img src="${product.image}" alt="${product.name}">
                    <div class="cart-item-details">
                        <h3 class="cart-item-title">${product.name}</h3>
                        <p
class="cart-item-price">$${product.price.toFixed(2)}</p>
                        <div class="cart-item-actions">
                            <input type="number" value="${product.quantity}"
min="1" class="quantity-input">
                            <button class="remove-button">Remove</button>
                        </div>
                    </div>
                `;

                cartItemsContainer.appendChild(cartItem);

                // Add event listeners
                const removeButton = cartItem.querySelector('.remove-button');
                const quantityInput = cartItem.querySelector('.quantity-input');

                removeButton.addEventListener('click', () => {
                    cart.splice(index, 1);
                    localStorage.setItem('cart', JSON.stringify(cart));
                    loadCartItems();
                });

                quantityInput.addEventListener('change', (event) => {
                    product.quantity = parseInt(event.target.value);
                    localStorage.setItem('cart', JSON.stringify(cart));
                    updateCartTotal();
                });
            });

            updateCartTotal();
        } else {
            cartItemsContainer.innerHTML = '<p>Your cart is empty</p>';
        }
    }

    // Function to update the cart total
    function updateCartTotal() {
        let total = 0;
        const cart = JSON.parse(localStorage.getItem('cart')) || [];
        cart.forEach(item => {
            total += item.price * item.quantity;
        });
        cartTotalElement.textContent = `Total: $${total.toFixed(2)}`;
    }

    // Initialize cart items
    loadCartItems();

    // Handle checkout button click
    checkoutButton.addEventListener('click', () => {
        alert('Proceeding to checkout');
    });
});
    </script>
</body>
</html>
```

CSS Styling (styles.css)

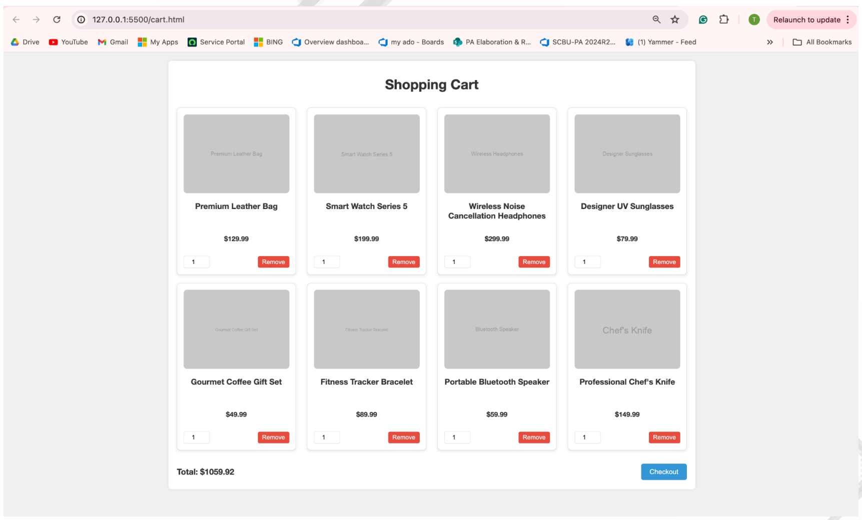
```
.cart-container {
  max-width: 1200px;
  margin: 20px auto;
  padding: 20px;
  background: #fff;
  border-radius: 8px;
  box-shadow: 0 0 10px rgba(0, 0, 0, 0.1);
}

.cart-title {
  text-align: center;
  color: #333;
  margin-bottom: 20px;
  font-size: 2rem;
```

```css
}

.cart-items-container {
  display: flex;
  flex-wrap: wrap;
  gap: 20px;
  justify-content: space-between;
}

.cart-item {
  background: #fff;
  border: 1px solid #ddd;
  border-radius: 8px;
  padding: 15px;
  box-shadow: 0 2px 5px rgba(0, 0, 0, 0.1);
  display: flex;
  flex-direction: column;
  align-items: center;
  position: relative;
  box-sizing: border-box;
  width: calc(25% - 20px); /* Ensure four items per row */
}

.cart-item img {
  max-width: 100%;
  border-radius: 8px;
  height: auto;
}

.cart-item-details {
  display: flex;
  flex-direction: column;
  align-items: center;
  width: 100%;
  margin-top: 10px;
}

.cart-item-title {
  font-weight: bold;
  color: #333;
  text-align: center;
  min-height: 3em; /* Fixed height for title */
}

.cart-item-price {
  font-weight: bold;
  color: #333;
  text-align: center;
  margin-top: 5px;
  min-height: 2em; /* Fixed height for price */
}
```

```css
.cart-item-actions {
  display: flex;
  justify-content: space-between;
  align-items: center;
  width: 100%;
  margin-top: 10px;
}

.quantity-input {
  width: 50px;
  text-align: center;
  border: 1px solid #ddd;
  border-radius: 4px;
  font-size: 0.9rem;
  padding: 5px;
}

.remove-button {
  background-color: #e74c3c;
  color: #fff;
  border: none;
  border-radius: 4px;
  padding: 5px 10px;
  cursor: pointer;
  font-size: 0.9rem;
}

.cart-checkout-button {
  background-color: #3498db;
  color: #fff;
  border: none;
  border-radius: 5px;
  padding: 10px 20px;
  cursor: pointer;
  font-size: 1rem;
}

.cart-summary {
  display: flex;
  justify-content: space-between;
  align-items: center;
  margin-top: 30px;
}

#cart-total {
  font-size: 1.2rem;
  font-weight: bold;
}

/* Responsive styles */
```

```
@media (max-width: 1024px) {
  .cart-item {
    width: calc(33.33% - 20px); /* Three items per row on medium screens */
  }
}

@media (max-width: 768px) {
  .cart-item {
    width: calc(50% - 20px); /* Two items per row on small screens */
  }
}

@media (max-width: 480px) {
  .cart-item {
    width: 100%; /* One item per row on very small screens */
  }
}
```

Output Screenshot:



Shopping Cart

Premium Leather Bag
$129.99
1 Remove

Smart Watch Series 5
$199.99
1 Remove

Wireless Noise Cancellation Headphones
$299.99
1 Remove

Designer UV Sunglasses
$79.99
1 Remove

Gourmet Coffee Gift Set
$49.99
1 Remove

Fitness Tracker Bracelet
$89.99
1 Remove

Portable Bluetooth Speaker
$59.99
1 Remove

Professional Chef's Knife
$149.99
1 Remove

Total: $1059.92
Checkout

HTML Structure

1. **Cart Container**:
   ○ This is the main wrapper for the cart page, containing the title, items, and summary. It uses CSS to center and style the content with padding and shadows.
2. **Cart Title**:
   ○ Displays a heading at the top of the cart page, indicating the section is for the shopping cart. It is centered and styled to be prominent.
3. **Cart Items Container**:

- This section holds the individual cart items. Initially empty, it will be populated dynamically with items from the JSON file. Each item is displayed inside this container.

4. **Cart Summary**:
   - Shows the total cost of all items in the cart and includes a checkout button. It provides a summary of the cart's state and allows the user to proceed to checkout.

JavaScript Functionality

1. **DOMContentLoaded Event Listener**:
   - Ensures that the JavaScript code runs only after the entire DOM (Document Object Model) has been fully loaded. This is crucial for manipulating the DOM elements correctly.
2. **loadCartItems Function:**
   - This function loads product data from localStorage.
   - Once the data is retrieved, it clears any previous content in the cart item container and populates it with new items.
   - For each product, a new HTML structure is created that includes an image, title, price, and action buttons (quantity input and remove button).
   - Event listeners are attached to the remove button and quantity input to handle user interactions, such as removing an item or updating quantities.
3. **updateCartTotal Function**:
   - Calculates the total cost of all items in the cart.
   - It loops through each cart item, retrieves the price and quantity, and computes the total.
   - The total amount is then displayed in the cart summary section.
4. **Checkout Button Event Listener**:
   - This function handles the click event on the checkout button.
   - When clicked, it displays an alert to notify the user that the checkout process is starting. This is a placeholder for further checkout functionality.
5. **addToCart Function (Product Page):**
   - Adds products to the cart with a specified quantity and updates localStorage.
   - Checks if the product already exists in the cart and updates the quantity accordingly, or adds it as a new item.
   - Displays an alert to notify the user that the product has been added to the cart.

CSS Styling

1. **Layout and Styling**:
   - The cart-container class provides a maximum width, margin, padding, and styling for the cart's appearance, including background color and shadow.
   - The cart-items-container uses Flexbox to arrange items in rows with gaps between them.
   - Each cart-item is styled to ensure it has a fixed width, background, border, and shadow, making it look like a card.
2. **Responsive Design**:

    - Media queries adjust the layout for different screen sizes.
    - On larger screens, four items are displayed per row. On medium screens, three items per row. On smaller screens, two items per row, and on very small screens, one item per row.

DOM Manipulation

- **Element Creation**:
    - New elements such as div, img, and button are created programmatically using JavaScript to build the HTML structure for each cart item.
- **Element Insertion**:
    - Elements are added to the DOM using methods like appendChild, which inserts newly created elements into existing containers.
- **Element Updates**:
    - Existing elements are updated to reflect the current state of the cart. For example, the total cost is recalculated and updated based on user interactions.
- **Event Handling**:
    - Event listeners are attached to elements such as buttons and inputs to respond to user actions, like removing an item or changing the quantity. This allows the page to react dynamically to user input.
    - This is explained in detail in the next section.

In summary, the code provides a dynamic shopping cart experience by fetching product data, creating and displaying cart items, and handling user interactions to update the cart's contents and total cost.

Event Handling

DOMContentLoaded Event Listener:

- This ensures that the JavaScript code only runs after the entire DOM has fully loaded. It is crucial for safely manipulating DOM elements since it guarantees that all elements are available for scripting.

```
document.addEventListener('DOMContentLoaded', () => {
  // JavaScript code to run after the DOM is fully loaded
});
```

Event Listener for Remove Button:

- Each cart item has a remove button. An event listener is added to this button to handle click events. When the button is clicked, the corresponding cart item is removed from the DOM, and the total cost is updated.

```
removeButton.addEventListener('click', () => {
    cartItem.remove();
    updateCartTotal();
});
```

This code is executed within the fetchCartItems function, which creates and appends the cart items to the container.

Event Listener for Quantity Input:

- Each cart item has a quantity input field. An event listener is attached to this input field to handle changes. When the quantity is changed, the updateCartTotal function is called to recalculate and display the new total cost.

```
quantityInput.addEventListener('change', () => {
    updateCartTotal();
});
```

This allows users to modify the quantity of each item and see the updated total immediately.

Event Listener for Checkout Button:

- The checkout button has an event listener attached to handle click events. When the button is clicked, an alert is shown to indicate that the checkout process is starting. This placeholder can be replaced with actual checkout logic.

```
checkoutButton.addEventListener('click', () => {
    alert('Proceeding to checkout');
});
```

Summary of Event Handling

- **Remove Button**: Handles the removal of items from the cart and updates the total.
- **Quantity Input**: Updates the total cost when the quantity is adjusted.
- **Checkout Button**: Provides feedback on checkout initiation.

Event handling in this code is used to create an interactive and dynamic user experience, ensuring that the cart responds to user actions like removing items, changing quantities, and proceeding to checkout.

Form Validation

Purpose:

Form validation is used to ensure that user input is correct and meets specified criteria before it is submitted. This helps in preventing invalid data from being processed.

Types:

- **Client-Side Validation**:
    - Uses HTML5 attributes and JavaScript to validate data before it is submitted to the server.
    - Provides immediate feedback to users, improving the user experience.
- **Server-Side Validation**:
    - Validates data on the server before processing it.
    - Ensures that data meets the specified criteria even if client-side validation is bypassed.

Updated Form Code

```
<!DOCTYPE html>
<html lang="en" class="form-html">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Sign Up / Sign In</title>
    <link rel="stylesheet" href="styles.css">
    <script>
        // Function for client-side validation
        function validateForm(event) {
            const form = event.target;
            const email = form.querySelector('input[name="email"]').value;
            const password = form.querySelector('input[name="password"]').value;

            if (form.id === 'signup-form') {
                const confirmPassword =
form.querySelector('input[name="confirmPassword"]').value;

                if (password !== confirmPassword) {
                    alert('Passwords do not match');
                    event.preventDefault(); // Prevent form submission
                    return false;
                }

                // Store user details in localStorage
                localStorage.setItem('userEmail', email);
                localStorage.setItem('userPassword', password);
                alert('Sign up successful!');
            }

            if (form.id === 'signin-form') {
                const storedEmail = localStorage.getItem('userEmail');
                const storedPassword = localStorage.getItem('userPassword');
                if (email !== storedEmail || password !== storedPassword) {
                    alert('Invalid email or password');
                    event.preventDefault(); // Prevent form submission
                    return false;
                }

                alert('Sign in successful!');
            }

            return true;
        }
    </script>
</head>
<body class="form-body">
    <div class="form-container">
        <!-- Signin/Signup Toggle -->
        <input type="radio" id="signin-tab" name="tab" checked>
        <label class="form-label" for="signin-tab">Sign In</label>

        <input type="radio" id="signup-tab" name="tab">
        <label class="form-label" for="signup-tab">Sign Up</label>

        <!-- Signin Form -->
        <div class="form signin-form">
            <h2>Sign In</h2>
            <form id="signin-form" action="#" method="POST" onsubmit="return
validateForm(event)">
                <input type="email" name="email" placeholder="Email Address"
required>
                <input type="password" name="password" placeholder="Password"
required>
                <button type="submit">Sign In</button>
            </form>
            <p class="form-text">Don't have an account? <label
class="form-label" for="signup-tab">Sign Up Here</label></p>
        </div>

        <!-- Signup Form -->
        <div class="form signup-form">
            <h2>Sign Up</h2>
            <form id="signup-form" action="#" method="POST" onsubmit="return
validateForm(event)">
                <input type="text" name="fullname" placeholder="Full Name"
required>
                <input type="email" name="email" placeholder="Email Address"
required>
                <input type="password" name="password" placeholder="Password"
required>
                <input type="password" name="confirmPassword"
placeholder="Confirm Password" required>
```

```
                <button type="submit">Sign Up</button>
            </form>
            <p class="form-text">Already have an account? <label
class="form-label" for="signin-tab">Sign In Here</label></p>
        </div>
    </div>
</body>
</html>
```

Explanation

Sign-In Form:

- Fields: Includes fields for email and password.
- Validation: Uses the onsubmit attribute to call the validateForm function for client-side validation.

Sign-Up Form:

- Additional Field: Adds a confirmPassword field for password confirmation.
- Form ID: The form is given an ID of signup-form, which is referenced in the validateForm function.
- Validation: The validateForm function checks if the password and confirmation match before allowing form submission.
- LocalStorage: Stores the user's email and password in localStorage upon successful sign-up.

How Validation is Implemented

Client-Side Validation:

- HTML5 Attributes:
    - Use of required ensures basic validation that fields cannot be left empty.
- JavaScript:
    - The validateForm function ensures that the passwords match in the sign-up form.
    - It provides immediate feedback without requiring a page reload.
    - Stores user data in localStorage for sign-up and validates against it during sign-in.

Event Handling:

- Form Submission:
    - The onsubmit attribute on the form calls the validateForm function.
    - This function is responsible for preventing form submission if validation fails.

Summary: The updated code includes JavaScript-based client-side validation to check if passwords match in the sign-up form and validate user credentials during sign-in using localStorage. This ensures that users correctly confirm their password before submitting the form and that sign-in credentials are validated against stored data. The validateForm function is triggered on form submission, providing a smoother and more user-friendly experience.

Final Code Files

Cart.html

Index.html

styles.css

Interview and FAQ References

DOM Manipulation

Understanding DOM Manipulation:

- **DOM (Document Object Model)**: The DOM represents the structure of a web page as a tree of objects. Each element, attribute, and piece of text is an object that can be accessed and manipulated programmatically.
- **DOM Manipulation**: Involves changing the structure, style, or content of the web page by interacting with these objects using JavaScript. This includes adding, removing, or modifying elements and their attributes.

Interview Questions and Answers:

What is DOM manipulation, and why is it important?

- **Answer:** DOM manipulation refers to the process of dynamically altering the document structure, content, and styles using JavaScript. It is crucial for creating interactive and dynamic web applications where content can change in response to user actions without requiring a page reload.

How do you add a new element to the DOM using JavaScript?

- **Answer:** To add a new element, you first create it using document.createElement(), then set its content or attributes, and finally append it to an existing element using methods like appendChild() or insertBefore(). Example:

```
const newDiv = document.createElement('div');
newDiv.textContent = 'Hello World';
document.body.appendChild(newDiv);
```

Explain how to remove an element from the DOM.

- **Answer:** To remove an element, select it using methods like document.querySelector() or getElementById(), then call the remove() method on the element or its parent element using removeChild(). Example:

```
const element = document.querySelector('#elementId');
element.remove(); // or
element.parentNode.removeChild(element);
```

What is event delegation in DOM manipulation?

- **Answer**: Event delegation involves attaching a single event listener to a parent element instead of multiple listeners to individual child elements. This is efficient and allows the listener to handle events from dynamically added child elements. Example:

```
document.querySelector('#parent').addEventListener('click', function(event) {
  if (event.target.matches('.child')) {
    console.log('Child element clicked');
  }
});
```

Advanced Topics:

- **How does the virtual DOM (e.g., in React) differ from direct DOM manipulation?**
  - **Answer**: The virtual DOM is a lightweight in-memory representation of the actual DOM. Libraries like React use it to efficiently update the real DOM by comparing changes and minimizing direct manipulation, improving performance.

Event Handling

Understanding Event Handling:

- **Event Handling**: Involves responding to user interactions or browser events (like clicks, keypresses, etc.) through event listeners. This allows developers to execute code in response to specific actions, enhancing user interactivity.

Interview Questions and Answers:

1. **What are event listeners, and how are they used in JavaScript?**
   - **Answer**: Event listeners are functions attached to HTML elements that execute code when a specified event occurs (e.g., click, keydown). They are added using methods like addEventListener(). Example:

```
document.querySelector('#myButton').addEventListener('click', () => {
  alert('Button clicked');
});
```

Explain the difference between addEventListener and onclick property.

- **Answer**: addEventListener allows multiple event handlers for the same event type and provides more control over event propagation. The onclick property is a simpler method for assigning a single handler to an element. Example:

```
// Using addEventListener
element.addEventListener('click', handler);

// Using onclick
element.onclick = handler;
```

1. **What is event bubbling, and how can it be controlled?**
   - **Answer**: Event bubbling is the process where an event propagates from the target element up through its ancestors in the DOM hierarchy. It can be controlled using event.stopPropagation() to prevent the event from reaching parent elements.
2. **How can you handle events for dynamically added elements?**
   - **Answer**: Use event delegation by attaching a single event listener to a parent element that will handle events from dynamically added child elements.

Advanced Topics:

- **How does event delegation improve performance in web applications?**
  - **Answer**: Event delegation reduces the number of event listeners by handling events at a higher level in the DOM, which is especially beneficial for applications with many dynamically added elements or high interactivity.

Form Validation

Understanding Form Validation:

- **Form Validation**: Ensures that user input in forms is accurate and complete before submission. This can be achieved through HTML5 attributes and client-side JavaScript.

Interview Questions and Answers:

1. **What are the basic HTML5 form validation attributes?**
   - **Answer**: HTML5 provides attributes like required, minlength, maxlength, pattern, and type (e.g., type="email"). These attributes enforce constraints and provide browser-level validation.
2. **How do you implement client-side validation using JavaScript?**
   - **Answer**: JavaScript validation involves writing custom scripts to check input values before form submission. This can be done using event listeners on the form's submit

event to validate fields and display error messages. Example:

```
document.querySelector('form').addEventListener('submit', function(event) {
  const password = document.querySelector('input[name="password"]').value;
  const confirmPassword =
document.querySelector('input[name="confirmPassword"]').value;
  if (password !== confirmPassword) {
    alert('Passwords do not match');
    event.preventDefault(); // Prevent form submission
  }
});
```

1. **What is the purpose of event.preventDefault() in form validation?**
    - **Answer**: event.preventDefault() stops the form from submitting if validation fails. It ensures that users correct errors before the form is sent to the server.
2. **How can you ensure accessibility in form validation?**
    - **Answer**: Use semantic HTML and ARIA roles/attributes to provide context and feedback to users with disabilities. For instance, use aria-invalid to indicate invalid fields and ensure error messages are accessible.

Advanced Topics:

- **How can you combine HTML5 and JavaScript validation for robust form validation?**
    - **Answer**: Use HTML5 attributes for basic validation and provide additional checks with JavaScript for more complex scenarios. HTML5 handles basic constraints, while JavaScript can manage more sophisticated validations and custom messages.
- **What are the benefits of server-side validation in addition to client-side validation?**
    - **Answer**: Server-side validation is essential for security as it ensures that all data submitted from the client is validated before processing. It prevents invalid or malicious data from affecting the server or database, complementing client-side checks.

These references and explanations will help in preparing for interviews and understanding key concepts related to DOM manipulation, event handling, and form validation.