COMSATS UNIVERSITY ISLAMABAD

ATTOCK CAMPUS

MOBILE APPLICATION DEVELOPMENT

ASSIGNMENT # 01

SUBMITTED TO:

SIR MUHAMMAD KAMRAN

SUBMITTED BY:

AFSANA SAJID (048)

QUESTION NO 01:

INTRODUCTION:

The JavaScript shopping cart feature is designed to simulate a basic e-commerce cart experience, allowing users to add, update, and remove items seamlessly. This implementation employs ES6 syntax and modern JavaScript practices to ensure clarity, efficiency, and maintainability.

Objectives

The primary goals of this shopping cart feature include:

1. **User-Friendly Item Management**: Simplify the process of adding and removing products while allowing easy updates to quantities.
2. **Accurate Cost Calculation**: Provide accurate total pricing based on item quantities and individual prices.
3. **Dynamic Cart Summary**: Display a clear summary of the cart's contents, highlighting only relevant items.
4. **Discount Application**: Offer a straightforward way to apply discount codes, enhancing the shopping experience.

Key Components

1. **ShoppingCart Class**:
   a. Central to the implementation, this class encapsulates all cart-related functionalities.
   b. It maintains the cart as an array of product objects.
2. **Product Object Structure**:
   a. Each product in the cart is represented as an object with the following properties:
      i. productId: A unique identifier for the product.
      ii. productName: The name of the product.
      iii. quantity: The number of items of that product in the cart.

iv. price: The price per unit of the product.

Functionalities

1. **Add Items to the Cart**:
   a. The addItem method allows users to add new products. It constructs a product object and appends it to the cart using the push method.
2. **Remove and Update Items**:
   a. **Removing Items**: The removeItem method utilizes the filter method to create a new array, excluding the specified product ID.
   b. **Updating Quantity**: The updateQuantity method modifies the quantity of an existing product using the map method, ensuring that only the targeted product is updated.
3. **Calculate Total Cost**:
   a. The calculateTotalCost method calculates the total price of the items in the cart. It uses the reduce method to iterate through each product, multiplying the quantity by the price and accumulating the total.
4. **Display Cart Summary**:
   a. The displayCartSummary method provides a formatted summary of the cart's contents. It uses map to create an array of summaries for each product, and filter to exclude items with a quantity of zero.
5. **Apply Discount Code**:
   a. The applyDiscount method checks for valid discount codes and adjusts the total cost accordingly. It demonstrates how to manage simple discount logic and provides users with immediate feedback on their savings.

OPERATION THAT ARE IMPLEMENTED:

Adding Items to the Cart:

- **Method**: `addItem(productId, productName, quantity, price)`

- **Operation**: This method takes product details as parameters and creates a product object. It then uses the `push` method to add this object to the `cart` array.

Adding Items to the Cart:

- **Method**: `addItem(productId, productName, quantity, price)`
- **Operation**: This method takes product details as parameters and creates a product object. It then uses the `push` method to add this object to the `cart` array.

Updating Item Quantity:

- **Method**: `updateQuantity(productId, newQuantity)`
- **Operation**: This method updates the quantity of a specific product. It uses the `map` method to iterate over the `cart` array, returning a new array where the targeted product's quantity is updated while keeping the rest unchanged.

Calculating Total Cost:

- **Method**: `calculateTotalCost()`
- **Operation**: This method calculates the total price of all items in the cart. It utilizes the `reduce` method to sum the products' total costs, which is determined by multiplying each product's price by its quantity.

Displaying Cart Summary:

- **Method**: `displayCartSummary()`
- **Operation**: This method generates a summary of the cart contents. It uses the `map` method to create an array of objects that include each product's name, quantity, and total price. It then uses `filter` to exclude items with a quantity of zero before displaying the results.

Applying Discount Code:

- **Method**: `applyDiscount(code)`
- **Operation**: This method checks if the provided discount code exists in a predefined set of codes. If valid, it calculates the discount amount and applies it to the total cost, returning the final price after the discount is applied.

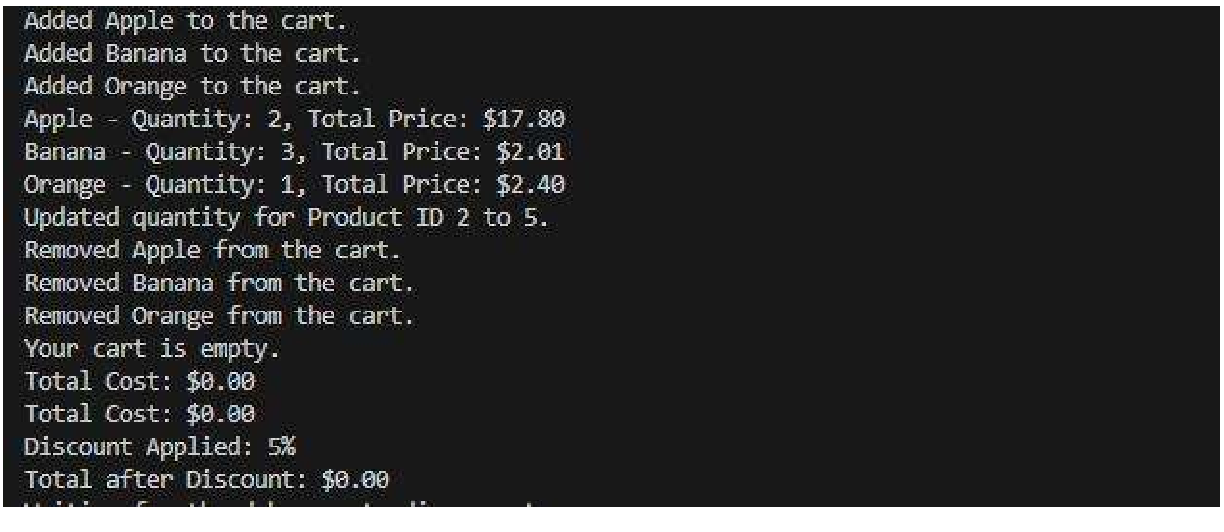
Logic Behind Each Operation

1. **Adding Items to the Cart**:
   a. **Logic**: The `addItem` method is designed to create a new product object with the provided details (ID, name, quantity, price) and append it to the cart array. The use of the `push` method allows for dynamic addition, meaning users can continuously add products without the need to reinitialize the cart.
   b. **Implementation Insight**: By encapsulating product details in an object, the code maintains a clean structure, allowing for easier future enhancements, such as adding more attributes (like images or descriptions).
2. **Removing Items from the Cart**:
   a. **Logic**: The `removeItem` method employs the `filter` method to create a new array that excludes the item with the specified `productId`. This approach ensures immutability, as it does not modify the original array but rather creates a new one.
   b. **Implementation Insight**: Filtering out products by ID is efficient and scalable, as it keeps the cart organized without needing to manually search for and remove items, which could be error-prone.
3. **Updating Item Quantity**:
   a. **Logic**: The `updateQuantity` method uses the `map` function to traverse the cart array and apply changes only to the targeted product. If a product matches the specified ID, a new object is returned with the updated quantity; otherwise, the original object remains unchanged.
   b. **Implementation Insight**: This method ensures that the cart remains immutable by returning a new array. It helps prevent side effects that could arise from directly modifying objects, which is a best practice in functional programming.
4. **Calculating Total Cost**:
   a. **Logic**: The `calculateTotalCost` method uses `reduce` to accumulate the total price of items in the cart. For each product, it multiplies the price by the quantity and adds this value to a running total.
   b. **Implementation Insight**: This method demonstrates a functional approach by leveraging `reduce` to consolidate values into a single output. It keeps the logic clear and concise, providing a straightforward way to compute totals.
5. **Displaying Cart Summary**:
   a. **Logic**: The `displayCartSummary` method first maps over the cart to create a summary object for each product, including the product name, quantity, and total price for that product. It then filters out any products with a quantity of zero to ensure only relevant items are displayed.
   b. **Implementation Insight**: This combination of `map` and `filter` promotes readability and ensures the display logic is clean. By separating the mapping and filtering steps, it makes the code easier to understand and maintain.

6. **Applying Discount Code**:
   a. **Logic**: The `applyDiscount` method checks the provided discount code against a predefined set of codes. If a valid code is found, it calculates the discount based on the total cost and subtracts it from the original total.
   b. **Implementation Insight**: This operation encapsulates the logic for discount application, which can be expanded later (e.g., adding new discount rules or types). It allows for easy integration of promotional features without altering the core pricing logic.

Overall Logic and Design Principles

- **Encapsulation**: The use of a class (`ShoppingCart`) encapsulates all cart-related logic, providing a clear interface for interacting with the cart while hiding the internal implementation details.
- **Immutability**: By creating new arrays or objects instead of modifying existing ones, the code minimizes potential side effects. This principle is crucial for maintaining state consistency, especially in applications where multiple components might interact with the cart.
- **Functional Programming**: The extensive use of higher-order array methods (`map`, `filter`, `reduce`) reflects a functional programming style that promotes cleaner, more predictable code. This approach makes it easier to reason about the transformations happening at each step.
- **Scalability**: The structure of the code is designed to be easily extendable. New features, such as additional item properties or new discount rules, can be integrated with minimal disruption to existing functionality.



```
Added Apple to the cart.
Added Banana to the cart.
Added Orange to the cart.
Apple - Quantity: 2, Total Price: $17.80
Banana - Quantity: 3, Total Price: $2.01
Orange - Quantity: 1, Total Price: $2.40
Updated quantity for Product ID 2 to 5.
Removed Apple from the cart.
Removed Banana from the cart.
Removed Orange from the cart.
Your cart is empty.
Total Cost: $0.00
Total Cost: $0.00
Discount Applied: 5%
Total after Discount: $0.00
```

Conclusion

The JavaScript shopping cart feature serves as a robust and efficient solution for managing items in a mobile e-commerce application. By utilizing modern JavaScript practices, including ES6 syntax and functional programming concepts, this implementation provides a seamless user experience while maintaining code clarity and maintainability.

Key takeaways from the implementation include:

1. **User-Centric Design**: The cart operations—adding, removing, updating items, calculating totals, and applying discounts—are designed to be intuitive and responsive to user actions, ensuring a smooth shopping experience.
2. **Encapsulation and Modularity**: The use of a dedicated `ShoppingCart` class encapsulates all relevant functionalities, promoting modularity. This structure allows developers to manage cart operations without affecting other components of the application.
3. **Functional Programming Techniques**: Leveraging higher-order array methods such as `map`, `filter`, and `reduce` leads to concise, readable, and expressive code. This approach simplifies complex data transformations and calculations.
4. **Immutability**: By avoiding direct modifications to existing data structures, the code minimizes potential side effects and enhances the predictability of state changes. This practice is especially important in applications where state management is critical.
5. **Scalability and Flexibility**: The current design allows for easy future enhancements, such as adding more product attributes, implementing additional discount codes, or integrating with external services. This flexibility makes the shopping cart adaptable to changing business requirements.

Overall, this shopping cart feature exemplifies best practices in software development, providing a solid foundation for building more complex e-commerce functionalities. It not only addresses immediate user needs but also sets the stage for future growth and enhancements in the application. As e-commerce continues to evolve, such implementations will remain vital for delivering efficient and effective user experiences.