Shopping Cart solution

To: Nadia Rasiet
Date: 03 October 2025
Sanlam FinTech Assessment

Detailed outline of my approach

My approach to solving this shopping cart code implementation was to look at core OOP principles that Java is built on, including encapsulation, abstraction, inheritance and polymorphism. Since these principles ensures that software is clean, reusable maintainable and highly scalable. To improve and analyze the performance of the algorithms, I used Big O notation and it helped me recognize inefficiencies in the data structures which affected the space and time complexity or the solution.

How were OOP principles violated

- The controller class directly manipulated raw data structures (Map<String, Object>) instead of using a proper model or a domain class.
- The business logic like calculating total was embedded inside of the controller class instead of being encapsulated in a separate class.
- The code had poor abstraction since there was no clear separation of concerns between controller, data model and the business logic.
- Hence the old approach led the code to be hard to maintain since everything was coupled together making it difficult for the team to read, maintain or make updates.

My approach was to redo the code using OOP concepts and MVC structure

- Keeping the controller class focused only on REST endpoints
- Created my own specialized objects (ShoppingItem,ShoppingCart) to prevent using raw objects, since a user can submit a form with invalid data and cause the system to crash due to class cast runtime error.
- I retained the initial business functionality with improved functionality, like calling methods to run calculations instead of using code redundantly.
- Before the solution was concatenating the item’s name and its price as the primary key which was not a good approach at all since items can have prices going up or down depending on demand, which would've resulted in system breaking and running slow since there’s no unique id. I used only the name for this solution but for real world production using uuid for each item is best the best approach.

Implementaion Choices

- I realized that the old code has a good data structure a HashMap, hence I kept this data structure. HashMap has O(1) average time complexity for insertion and lookup, which is efficient for storing cart items.
- I created two new classes ShoppingItem and ShoppingCart to keep the code modular and readable while exercising Encapsulation.
- To safely handle data in case a user enters invalid data in the form, I replaced the generic raw object Map<String,Object> to prevent unsafe casting and removed itemName_price which caused parsing errors and primary key conflics.
- There was an unused import of BigDecimal which I utilized instead of double or a float since BigDecimal ensures accuracy for finencial values and prevents conversion errors.
- I used a for-each loop to find items and sum totals since it is safer and less prone to common index related errors.
- Instead of concatenating item name and price as key, I stored items by their name directly to prevent breaking the system when prices updates. In a real system using specialized imports for generating unique id(like UUID/SKU) would be a great approach.

Unclear library usage and Code Correctness

- BigDecimal was imported in the original code but not used, I applied it properly and since it is a standard way to represent currencies in Java, and to avoid rounding issues with floating points.
- HashMap is still in use since it runs at a constant space and time complexity making sure that the data retrievals and insertions happen efficiently.
- Spring annotations remain the same now and mostly they are only responsible for routing requests and not for holding business logic.
- For code correctness I removed the use of raw Object in the map which required unsafe casting resulting in ClassCastException. I also Replaced double with BigDecimal which avoids rounding errors. And the total calculation is encapsulated inside of ShoppingCart class O(n) operations instead of duplicating same method which results in O(n.m) operations causing overhead and the degrades perfomance if the input size increases.

Kind Regards.
Mkhatshwa Akhona