**Adapter Pattern Implementation Report**

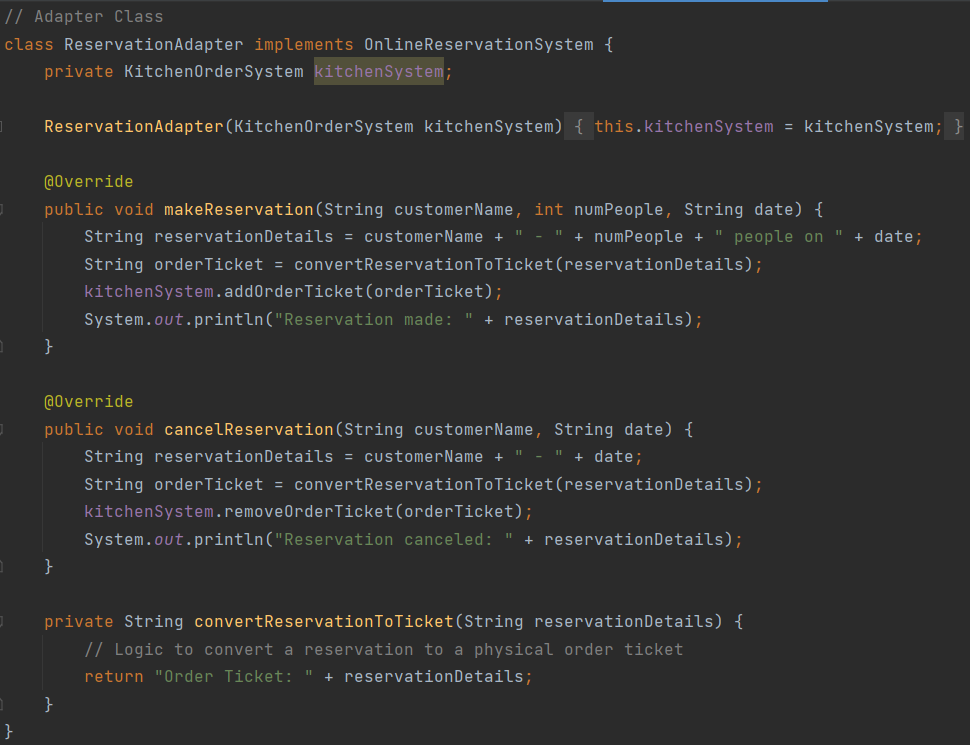
**1. Brief Explanation of the Adapter Pattern:** The Adapter Pattern is a structural design pattern that allows objects with incompatible interfaces to work together. It involves creating an adapter class that acts as an intermediary to translate requests from one interface to another. This pattern is particularly useful when you have two systems or components that need to collaborate but have different interfaces.

**2. Description of the Chosen Real-World Scenario:** In this implementation, I applied the Adapter Pattern to a restaurant scenario. The scenario involves two systems:

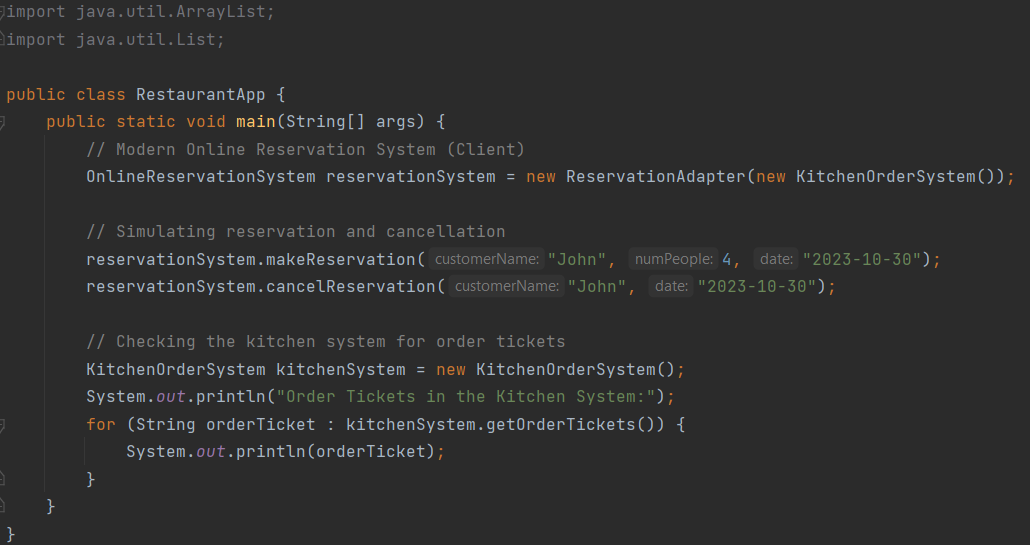
* A modern online reservation system (the client) that allows customers to make and cancel reservations.
* An older kitchen order processing system (the adaptee) that processes orders using physical printed tickets.

**3. Explanation of How and Why the Adapter Pattern was Applied:** The Adapter Pattern was applied to bridge the gap between the modern online reservation system and the older kitchen order processing system. These two systems had different interfaces and were unable to communicate directly. We used the Adapter Pattern to create an **ReservationAdapter** class that allowed the modern reservation system to make reservations and cancel them while translating these actions into the language of the kitchen system (i.e., creating and removing physical order tickets).

**4. Code Screenshots with Explanations:**

**The Adapter Class (ReservationAdapter):** 

In this screenshot, you can see the **ReservationAdapter** class, which implements the **OnlineReservationSystem** interface. It adapts the methods **makeReservation** and **cancelReservation** to work with the **KitchenOrderSystem**. It also contains a method to convert reservations into physical order tickets.

**Main Class (RestaurantApp):** 

Here's the main class that serves as the entry point for our application. It creates instances of **ReservationAdapter** and **KitchenOrderSystem**. Inside the loop, user interactions are handled, allowing reservations to be made and canceled, and order tickets to be viewed.

**5. Test Cases and Their Outcomes:** Here are some example test cases and their expected outcomes:

* **Test Case 1: Making a Reservation**
* Input: "John", 4, "2023-10-30"
* Expected Outcome: "Reservation made: John - 4 people on 2023-10-30"
* **Test Case 2: Canceling a Reservation**
* Input: "John", "2023-10-30"
* Expected Outcome: "Reservation canceled: John - 2023-10-30"
* **Test Case 3: Viewing Order Tickets**
* Expected Outcome: List of order tickets in the kitchen system

**6. Reflection on the Challenges and Benefits:**

* **Challenges:**
* Initially, there was an issue with canceling reservations, which was resolved by modifying the code to remove reservations directly.
* **Benefits:**
* The Adapter Pattern effectively bridged the gap between the modern online reservation system and the older kitchen order processing system.
* It provided a clear separation between the client and adaptee, making the code more maintainable.
* The design demonstrated how the Adapter Pattern can be used to adapt real-world systems with minimal code changes.

In summary, the Adapter Pattern proved effective in adapting the interfaces of the online reservation and kitchen order processing systems. It's a valuable pattern when dealing with legacy systems and modern systems that need to work together.