Assignment 8

Make use of abstract classes and interface wherever possible

Question 1: You are tasked to design a Mess Management System for your institution. The system will facilitate the management of meal plans and allow students to subscribe to these plans. Do the following as given below.

a) Create interfaces **Person, Student** and **MealPlan** with the following methods:

- **Person Interface**:
  - *`getName()`*: Returns the name of the person.
  - *`getID()`*: Returns the ID of the person.

- **`Student` Interface** :

  - *`getSubscribedMealPlans()`*: Returns a list of meal plans the student is subscribed to.
  - *`subscribeMealPlan(MealPlan mealPlan)`*: Allows a student to subscribe to a meal plan.
  - *`unsubscribeMealPlan(MealPlan mealPlan)`*: Allows a student to unsubscribe from a meal plan.

- **MealPlan Interface**:

  - *`getPlanName()`*: Returns the name of the meal plan.
  - *`getMenuItems()`*: Returns a list of menu items included in the meal plan.
  - *`getPrice()`*: Returns the total price of the meal plan.

b) Create classes **MealPLAN, MenuItem, StudentImpl and `MessController`** with the following and attributes:

- **`MealPLAN`** class implements the **`MealPlan`** interface and also includes the attributes given below.

  - *`planName:`* The name of the meal plan.
  - `menuItems`: A list to hold menu items associated with the meal plan.
  - `price`: The total price of the meal plan.
  - Implement the methods to:
    - Add a menu item.
    - Get the list of menu items.
    - Get the total price of the plan.

- **`MenuItem`** class have the following attributes:

        - `itemName:` The name of the menu item.
        - `price:` The price of the menu item.
        - Implement methods to get the item name and price.

    - **`StudentImpl`** class implements the **`Student` and Person** interface. This class should include following attributes:

        - `name:` The name of the student.
        - `id`: The student ID.
        - `subscribedMealPlans`: A list to track meal plans the student is subscribed to.
        - Implement methods to subscribe to and unsubscribe from meal plans.

    - **`Create a MessController`** class with the following method:

        - A collection of `MealPlan` objects.
        - Methods to add meal plans and allow students to subscribe or unsubscribe from them.
        - A method to print the subscription status of a student.

```
MessController controller = new MessController();
MealPLAN vegetarianPlan = new MealPLAN("Vegetarian Plan", 10.0);
vegetarianPlan.addMenuItem(new MenuItem("Salad", 5.0));
vegetarianPlan.addMenuItem(new MenuItem("Fruit", 3.0));

controller.addMealPlan(vegetarianPlan);

StudentImpl student1 = new StudentImpl("Alice", "S001");

controller.subscribeStudentToMealPlan(student1, vegetarianPlan);
controller.printSubscriptionStatus(student1);
```

Question 2

You are developing a smart logistics system for a global shipping company. The system needs to process shipments that can be categorized into various types, such as **Domestic**, **International**, and **Express** shipments. Each shipment can have different sets of parameters, including **weight**, **distance**, **shipping mode** (air, sea, road), and **customs duties** (for international shipments).

You need to implement a `calculateCost()` method that will compute the total cost of a shipment, with different overloads for each type of shipment. The system should correctly handle different combinations of inputs such as the weight, distance, mode of transport, and customs duties, and return the appropriate cost based on the overloaded method that gets invoked.

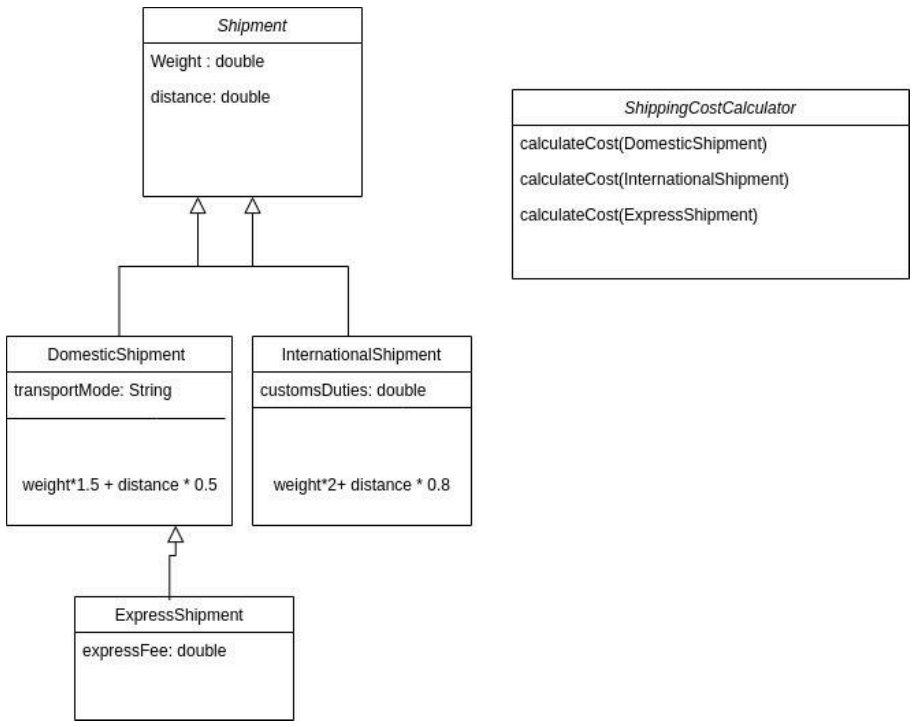
```
class ShippingCostCalculator {

        // Overload 1: Handles Domestic Shipment
        public double calculateCost(DomesticShipment shipment)
        {
        return (shipment.weight * 1.5) + (shipment.distance * 0.5);
        }

        // Overload 2: Handles International Shipment
        public double calculateCost(InternationalShipment shipment)
        {
        return (shipment.weight * 2.0) + (shipment.distance * 0.8) +
shipment.customsDuties;
        }

        // Overload 3: Handles Express Shipment (with express fee)
        public double calculateCost(ExpressShipment shipment)
        {
        return (shipment.weight * 1.5) + (shipment.distance * 0.5) +
shipment.expressFee;
        }

}
```



Question 3

You are developing a **distributed data processing framework** for a large-scale **financial data analytics system**. The system processes data from different sources, such as **Stock Market Prices**, **Banking Transactions**, and **Cryptocurrency Exchange Rates**. Each type of data presents unique challenges, requiring customized processing strategies in a distributed environment.

The framework consists of a base class `DistributedDataProcessor`, which provides methods for `loadData()`, `processData()`, and `aggregateData()`. However, each data source has specific processing needs, so these methods must be **overridden** by subclasses like `StockProcessor`, `TransactionProcessor`, and `CryptoProcessor`. Additionally, the system supports **overloaded** `processData()` methods to handle both real-time streaming data and batch data, depending on the source and use case.

For **Stock Market Prices**, the `StockProcessor` class must handle:

- **Real-time stock prices**: Streamed from multiple sources, requiring fast, parallel processing to detect fluctuations and trigger alerts in a distributed manner.
- **Historical stock data**: Processed in large batches, often used for predicting trends, requiring resource-intensive algorithms such as moving averages or anomaly detection.

Each type of data also requires a distinct **aggregation strategy**:

1. **Time-based aggregation**: For stock prices, where hourly, daily, or weekly trends are calculated.
2. **User-based aggregation**: For banking transactions, where multiple transactions from the same user are grouped and analyzed for potential fraud or risk scoring.
3. **Currency-based aggregation**: For cryptocurrency exchange rates, where data from various currencies are aggregated to track exchange performance.

These aggregation strategies must be **overridden** in the subclasses, while providing flexibility to add **custom aggregation strategies** through **method overloading**, especially in cases where aggregation must consider special conditions (e.g., different user risk profiles in transactions).

Additionally, the system supports **parallel execution**. The `processData()` method must be **overloaded** to support both **single-threaded** processing (for small or low-frequency data) and **multi-threaded distributed processing** (for high-volume real-time streams). The choice of execution mode should dynamically depend on system load and data characteristics.

Tasks:

1. Define a base class `DistributedDataProcessor` that provides a distributed `loadData()`, `processData()`, and `aggregateData()` method.
2. Implement subclasses `StockProcessor`, `TransactionProcessor`, and `CryptoProcessor` that:
   - Override the `processData()` and `aggregateData()` methods to handle the unique complexities of stock prices, banking transactions, and cryptocurrency data, respectively.
   - Overload the `processData()` method in the `StockProcessor` class to process both real-time streaming data and historical batch data, considering both single-threaded and multi-threaded modes.
3. Ensure the `aggregateData()` method in each processor handles:
   - Time-based aggregation (for stock prices).
   - User-based aggregation (for banking transactions).
   - Currency-based aggregation (for cryptocurrency data). Additionally, overload `aggregateData()` in `TransactionProcessor` to allow custom user profiling during aggregation (e.g., risk-based aggregation for users with certain behavior patterns).

Bonus: Implement a flexible strategy for dynamically switching between single-threaded and multi-threaded modes within the `processData()` method, considering system load and available computing resources. Use interfaces or abstract classes to manage distributed resource allocation and parallelism.

#Sample Main class

public class Main {

```
    public static void main(String[] args) {

        // StockProcessor instance

        List<String> stockBatch = List.of("AAPL", "GOOGL", "TSLA");

        StockProcessor stockProcessor = new StockProcessor("StockMarket", stockBatch);

        stockProcessor.loadData();

        stockProcessor.processData(); // Batch processing

        stockProcessor.aggregateData("Daily");


        // Real-time stock stream processing

        Stream<String> stockStream = Stream.of("AAPL", "AMZN", "TSLA");

        stockProcessor.processData(stockStream); // Real-time processing


        // TransactionProcessor instance

        List<String> transactionBatch = List.of("TXN1", "TXN2", "TXN3");

        TransactionProcessor transactionProcessor = new
TransactionProcessor("BankTransactions", transactionBatch);

        transactionProcessor.loadData();

        transactionProcessor.processData(); // Batch processing

        transactionProcessor.aggregateData("User123", "High Risk");


        // CryptoProcessor instance

        List<String> cryptoBatch = List.of("BTC", "ETH", "XRP");

        CryptoProcessor cryptoProcessor = new CryptoProcessor("CryptoExchange",
cryptoBatch);

        cryptoProcessor.loadData();
        cryptoProcessor.processData(); // Batch processing

        cryptoProcessor.aggregateData(); // Currency-based aggregation

    }

}
```

Question 4

You are tasked with designing a **Banking System** that simulates different types of bank accounts, each with its own unique behaviors and rules. The system must handle various account types such as **SavingsAccount**, **CheckingAccount**, and **InternationalAccount**, each with specific requirements for deposits, withdrawals, interest calculation, and generating monthly statements.

The system should implement the following core functionalities:

calculateInterest(): A method to compute interest, which will have different implementations depending on the account type.

withdraw(double amount): A method to handle withdrawals, where some account types (e.g., CheckingAccount) may apply additional fees.

deposit(double amount): A common method to add money to the account, but different accounts may impose restrictions (e.g., limits on the amount for certain account types).

getMonthlyStatement(): A method to generate a monthly statement, which will display different formats of information for different account types, such as recent transactions, interest earned, or fees charged.

System should also consider additional complexities such as:

- **Minimum Balance Requirements**: For some accounts (e.g., SavingsAccount), if the balance falls below a threshold, a penalty is applied.
- **Overdraft Facility**: For **CheckingAccount**, an overdraft limit should be set, and the system should allow withdrawal beyond the available balance up to a certain limit, charging interest on the overdraft.
- **Currency Conversion**: In the **InternationalAccount** subclass, override the **`deposit()`** method to allow deposits in multiple currencies, applying a currency conversion rate before adding to the account balance.

Implement the following classes:

- **Bank**: A base class with general properties and methods like **`calculateInterest()`**, **`withdraw()`, `deposit()`**, and **`getMonthlyStatement()`.**
- **Account**: A class that extends **`Bank`** and contains common account attributes such as **`accountNumber`, `balance`**, and **`customerName`.**
- **SavingsAccount**: A subclass of **`Account`** that overrides **`calculateInterest()`** to apply interest at a specified rate, enforces a minimum balance, and penalizes accounts that fall below it.
- **CheckingAccount**: A subclass of **`Account`** that overrides **`withdraw()`** to include a transaction fee, implements an overdraft facility, and charges interest on the overdraft amount.
- **InternationalAccount**: A subclass of **`Account`** that overrides the **`deposit()`** method to handle deposits in multiple currencies and converts them to the local currency.

Additional constraints

Interest Calculation: Savings accounts have a fixed interest rate (e.g., 5%), while checking accounts may not earn interest but could have overdraft interest.

Transaction Fees: Checking accounts charge a fee on every withdrawal, and international accounts may charge a fee for currency conversion.