CS23304- Java programming Laboratory
on the spot program

Date: 12.8.26 Marks: 25

Problem Statement: Multi-Threaded Smart Banking System

Objective

Design and implement a robust, package-structured Java application for a modern banking platform. The application must simulate core banking operations while explicitly demonstrating the implementation of fundamental and advanced Object-Oriented Programming (OOP) concepts, including classes, methods, inheritance, interfaces, inner classes, exception handling, packages, and multithreading.

1. Package Structure (`com.bank`)

Organize your source code into a clean, hierarchical package structure:

- `com.bank.model`: Contains the core account classes and interfaces.
- `com.bank.exceptions`: Contains custom user-defined exception classes.
- `com.bank.threads`: Contains classes handling concurrent transactions using multithreading.
- `com.bank.main`: Contains the executable driver class with the `main` method.

2. Classes, Methods, and Inheritance (`com.bank.model`)

- **Base Class (`Account`):** Create an abstract or concrete class containing common attributes (`accountNumber`, `accountHolderName`, `balance`) and concrete methods (`deposit(double amount)`, `getBalance()`).
- **Derived Classes:** Create two subclasses that inherit from `Account`:

`SavingsAccount` Class

Attributes (Fields)

- **`interestRate`** *(double)*: The fixed percentage rate used to calculate annual or monthly interest earned on the accumulated balance.
- **`minimumBalance`** *(double)*: The mandatory threshold amount that must be maintained in the account at all times to avoid penalty fees or restricted withdrawals.

Methods

- **`calculateInterest()`** *(void/double)*: Computes the interest accrued based on the current balance and the `interestRate`, then automatically credits it back to the account balance.
- **`withdraw(double amount)`** *(synchronized boolean - Overridden)*:
    - *Specialized Logic:* Checks if the requested withdrawal amount would cause the resulting balance to fall below the `minimumBalance`.
    - If it violates the minimum balance rule, it throws an `InsufficientFundsException`. Otherwise, it safely updates the balance.
- **`printStatement()`** *(void - Overridden/Implemented)*: Formats and displays account details along with savings-specific metrics like accrued interest and current minimum balance status via the inner `Passbook` class.

2. `CurrentAccount` Class

Attributes (Fields)

- **`overdraftLimit`** *(double)*: The maximum negative credit limit extended to the account holder, allowing them to withdraw funds even when the actual balance hits zero.
- **`overdraftFee`** *(double)*: A flat fee or penalty charged to the account whenever the balance dips below zero and the overdraft facility is actively utilized.

Methods

- **`withdraw(double amount)`** *(synchronized boolean - Overridden)*:
    - *Specialized Logic:* Unlike a savings account, a current account permits withdrawals exceeding the current balance, provided the negative amount does not cross `balance + overdraftLimit`.
    - If the limit is breached, it throws an `InsufficientFundsException`. If a withdrawal drops the account into a negative balance, it automatically triggers the `chargeOverdraftFee()` method.
- **`chargeOverdraftFee()`** *(void)*: Deducts the pre-established `overdraftFee` from the account balance whenever an overdraft occurs.
- **`printStatement()`** *(void - Overridden/Implemented)*: Formats and prints comprehensive business statements, explicitly itemizing current overdraft utilization and accumulated penalties via the inner `Passbook` class.

3. Interfaces (`com.bank.model`)

- Define an interface named `TransactionOperations` that declares abstract methods like `transfer(Account destination, double amount)` and `printStatement()`.
- Implement this interface within your account classes or a dedicated transaction processor class.

4. Inner and Nested Classes (`com.bank.model`)

- **Static Nested Class:** Implement a static nested class named `BankPolicy` inside the main system or account module to hold global configuration constants (e.g., maximum daily transaction limits, base interest rates).
- **Member Inner Class:** Implement a non-static inner class named `Passbook` inside the `Account` class that has direct access to the account's private balance and transaction logs to print historical statements.

5. Exception Handling (`com.bank.exceptions`)

Create custom checked and unchecked exceptions to handle erroneous banking states gracefully:

- `InsufficientFundsException`: Thrown when a withdrawal or transfer exceeds the available balance (or allowed overdraft).
- `InvalidAmountException`: Thrown when a negative or zero value is passed for deposits or withdrawals.
- Use `try-catch-finally` blocks and `throws`/`throw` keywords during transaction execution to catch these exceptions without crashing the program.

6. Multithreading (`com.bank.threads`)

- Implement multithreading to simulate concurrent user access (e.g., multiple ATMs or online users performing simultaneous deposits and withdrawals on the same account).
- Create a task class implementing `Runnable` (or extending `Thread`) named `TransactionTask`.
- Use synchronization blocks (`synchronized`) on critical methods (like `withdraw` and `deposit`) to prevent race conditions and ensure thread safety for the account balance.