PROBLEM STATEMENT

Secure Banking Withdrawal System with Defensive Programming

Context

You are developing a **secure banking module** for a financial application.
The system must handle **account creation**, **withdrawal operations**, **exception handling**, and **error logging**, while following **defensive programming practices** and **custom exception design**.

The application must ensure **data integrity**, **clear error reporting**, and **proper exception propagation**.

MODULE 1: BANK ACCOUNT MANAGEMENT

TASK 1 – Namespace and Class Design

Student Must Do

- Create a namespace named **`BankingSystem`** (or equivalent logical banking namespace).
- Inside this namespace, define a class named **`BankAccount`**.

Expected Outcome

- All banking-related logic is logically grouped.
- The `BankAccount` class represents a real-world bank account entity.

TASK 2 – Data Members and Access Control

Student Must Do

Define the following data members inside the `BankAccount` class:

Data Member Name	Data Type	Access Modifier	Purpose
`AccountNumber`	`string`	`public` (getter), `private` (setter)	Stores unique account identifier
`Balance`	`decimal`	`public` (getter), `private` (setter)	Stores current account balance

Expected Outcome

- Account number and balance can be **read** externally.
- Account number and balance **cannot be modified directly** outside the class.
- Encapsulation is strictly enforced.

TASK 3 – Constructor with Validation

Method Specification

- **Method Name:** `BankAccount`
- **Return Type:** Constructor
- **Parameters:**
  - `string accountNumber`
  - `decimal initialBalance`

Student Must Do

- Validate that the account number is **not null, empty, or whitespace**.
- Validate that the initial balance is **not negative**.
- Throw appropriate exceptions for invalid input.
- Initialize the account number and balance only if validation succeeds.

Expected Outcome

- Invalid account creation is prevented.
- Bank accounts always start in a valid state.
- Defensive programming is applied at object creation time.

TASK 4 – Withdrawal Operation

Method Specification

- **Method Name:** `Withdraw`
- **Return Type:** `void`
- **Parameters:**
    - `decimal amount`

Student Must Do

- Validate that the withdrawal amount is **greater than zero**.
- If the withdrawal amount exceeds the current balance:
    - Throw a **custom insufficient balance exception** with a meaningful message.
- If the withdrawal is valid:
    - Deduct the amount from the balance.
    - Display a success message with the updated balance.

Expected Outcome

- Invalid withdrawal amounts are rejected immediately.
- Overdraft attempts are detected and blocked.
- Successful withdrawals update the balance correctly.

TASK 5 – Defensive Exception Handling inside Withdraw

Student Must Do

Inside the withdrawal operation:

- Handle **insufficient balance exceptions** separately.
- Log exception details before rethrowing them.
- Catch any unexpected exceptions:
    - Log them.
    - Wrap them inside a higher-level banking exception.

    - Preserve the original exception as the inner exception.

Expected Outcome

- Known business exceptions are handled cleanly.
- Unexpected failures are logged and escalated safely.
- Exception chaining is implemented correctly.

TASK 6 – Exception Logging Mechanism

Method Specification

- **Method Name:** `LogException`
- **Return Type:** `void`
- **Parameters:**
    - `Exception ex`

Student Must Do

- Create a private logging method.
- Log the following details to a file:
    - Date and time
    - Account number
    - Complete exception information
- Append logs to a text file instead of overwriting.

Expected Outcome

- All errors are permanently recorded.
- Logs contain sufficient diagnostic information.
- Logging is hidden from external classes.

MODULE 2: CUSTOM EXCEPTION HANDLING

TASK 7 – Insufficient Balance Exception

Class Specification

- **Class Name:** `InsufficientBalanceException`
- **Base Class:** `Exception`

Student Must Do

- Create a custom exception to represent insufficient balance errors.
- Accept a descriptive error message through the constructor.
- Pass the message to the base exception class.

Expected Outcome

- Business-rule violations are clearly identified.
- Exception type clearly communicates the failure reason.

TASK 8 – Bank Operation Exception with Inner Exception

Class Specification

- **Class Name:** `BankOperationException`
- **Base Class:** `Exception`

Student Must Do

- Create a custom exception for unexpected banking errors.
- Accept:
  - A custom message
  - An inner exception
- Preserve the root cause using exception chaining.

Expected Outcome

- Higher layers receive meaningful error context.
- Root causes are not lost.
- Exception hierarchy is maintained.

MODULE 3: APPLICATION EXECUTION & TESTING

TASK 9 – Program Execution and Exception Handling

Student Must Do

- Create a `Program` class with a `Main` method.
- Create a bank account with an initial balance.
- Attempt a withdrawal that exceeds the balance.
- Handle the following exceptions separately:
  - Insufficient balance exception
  - Bank operation exception (including inner exception)
  - Any other unexpected exception

Expected Outcome

- Insufficient balance message is displayed to the user.
- Bank operation errors show both high-level and root-cause messages.
- Application does not crash abruptly.

OVERALL EXPECTED LEARNING OUTCOMES

By completing this problem, the student demonstrates:

- Defensive programming skills
- Custom exception design
- Proper use of inner exceptions
- Exception logging strategies
- Secure financial operation handling
- Real-world object-oriented design