Reflection Log: Accounts Program

Project Description

In this project, I developed a Java program to simulate different types of bank accounts:

1. **Base Class**: `Account`
2. **Derived Classes**: `PersonalAcct` and `BusinessAcct`

The program demonstrated **inheritance**, where `PersonalAcct` and `BusinessAcct` inherited common functionality (e.g., deposit, withdraw, and getBalance) from the `Account` class while adding specific rules, such as penalties for withdrawals below a minimum balance.

What I Learned

1. **Inheritance**:
    - I utilized inheritance to define shared properties and methods in the `Account` class, avoiding redundancy in subclasses.
    - Both `PersonalAcct` and `BusinessAcct` inherited core functionality, making the code easier to maintain and extend.
2. **Overriding Methods**:
    - I learned how to override base class methods to customize behavior in subclasses.
    - For instance, the `withdraw` method in both `PersonalAcct` and `BusinessAcct` was overridden to enforce penalties for dropping below the minimum balance.
3. **Encapsulation**:
    - The `balance` field was declared private, and public methods were provided to modify and access it.
    - This ensured proper validation and controlled interactions with the account's balance.
4. **Constants Usage**:
    - I replaced hardcoded values (e.g., penalties and minimum balances) with constants, making the code more maintainable and easier to update.

Challenges

1. **Validating Withdrawals**:
    - Initially, I overlooked validating if the withdrawal amount exceeded the balance before applying penalties, leading to incorrect results.
    - Fixing this issue emphasized the importance of validating operations carefully to ensure logical correctness.
2. **Penalty Application**:
    - At first, I repeated penalty logic in both subclasses, increasing redundancy.
    - Refactoring this logic into a utility method reduced repetition, making the code more concise and reusable.
3. **Testing Edge Cases**:
    - It was challenging to test scenarios like withdrawals that precisely hit the minimum balance or situations where penalties might stack.
    - Writing thorough test cases uncovered bugs and improved the program's reliability.

Improvements

1. **Enhanced User Interaction**:
    - In the future, I could add dynamic user input to allow for account creation, transaction processing, and balance displays interactively, rather than relying on hardcoded data.
2. **Custom Exceptions**:
    - Adding custom exceptions (e.g., `OverWithdrawalException`, `InvalidDepositException`) would improve error handling and make the program more descriptive and user-friendly.
3. **Persistent Storage**:
    - Implementing file I/O or database storage to save and load account data would make the program more practical for real-world applications.

Reflection

This project significantly enhanced my understanding of object-oriented design principles, particularly **inheritance**, **method overriding**, and **encapsulation**. Key takeaways include:

- The importance of **validation** in ensuring logical operations.
- The value of **refactoring** repetitive code to improve maintainability.
- How using **constants** simplifies updates and enhances readability.

These skills will prove invaluable for future projects, especially those involving business logic and complex data models. This experience has reinforced my ability to design and implement robust, reusable, and maintainable software.