Project 1 (Day 1): Basic Account Class

Description

Students will create a foundational `Account` class that represents a bank account. This class should include basic operations such as depositing money, withdrawing funds, and checking the account balance. The goal is to practice encapsulation and basic method writing.

Attributes

- **accountNumber** (String): A unique identifier for the account.
- **accountHolder** (String): The name (or ID) of the account holder.
- **balance** (double): The current monetary balance in the account.

Methods

1. **deposit(amount)**

    - **Purpose:** Add a specified amount to the balance.
    - **Parameters:** `amount` (double) – Must be a positive number.
    - **Return:** No value (void).
    - **Behavior:** Increase `balance` by `amount` if `amount > 0` .

2. **withdraw(amount)**

    - **Purpose:** Subtract a specified amount from the balance if sufficient funds exist.
    - **Parameters:** `amount` (double) – Must be positive and less than or equal to the current balance.
    - **Return:** Boolean – `true` if the withdrawal is successful, `false` otherwise.
    - **Behavior:** Decrease `balance` by `amount` if there is enough balance.

3. **getBalance()**

    - **Purpose:** Retrieve the current balance.
    - **Return:** The current `balance` (double).

Manual Test Cases (Using if/else Logic)

- **Test Case 1: Deposit Valid Amount**

    - **Action:** Create an account with an initial balance of 1000. Call `deposit(500)` .

    - **If/Else Check:**
        - **If** the new balance equals 1500, **then** print “Deposit successful.”
        - **Else** print “Error: Deposit did not update balance correctly.”

- **Test Case 2: Withdraw Valid Amount**
    - **Action:** Using the account with balance 1500, call `withdraw(300)` .
    - **If/Else Check:**
        - **If** `withdraw(300)` returns `true` and the new balance equals 1200, **then** print “Withdrawal successful.”
        - **Else** print “Error: Withdrawal failed or balance incorrect.”

- **Test Case 3: Withdraw Exceeding Balance**
    - **Action:** Attempt to withdraw 2000 from the account with balance 1200.
    - **If/Else Check:**
        - **If** `withdraw(2000)` returns `false` , **then** print “Properly handled insufficient funds.”
        - **Else** print “Error: Withdrawal allowed without sufficient funds.”

- **Test Case 4: Deposit Negative Amount**
    - **Action:** Attempt to deposit a negative value (e.g., `deposit(-100)`).
    - **If/Else Check:**
        - **If** the balance remains unchanged, **then** print “Negative deposit rejected.”
        - **Else** print “Error: Negative deposit affected the balance.”

Project 2 (Day 2): Extending Account – SavingsAccount & CurrentAccount

Description

Extend the basic `Account` class by creating two specialized account types:

- **SavingsAccount:** Includes an interest rate and a method to calculate interest.
- **CurrentAccount:** Includes an overdraft limit and an overridden withdrawal method that permits overdraft up to a limit.

Additional Attributes

- **For SavingsAccount:**

    - **interestRate** (double): The interest rate (e.g., 0.05 for 5%).

- **For CurrentAccount:**
    - **overdraftLimit** (double): The extra amount allowed for withdrawal beyond the current balance.

Additional Methods

1. **SavingsAccount: calculateInterest()**
    - **Purpose:** Compute the interest based on the current balance.
    - **Return:** The interest amount (double) calculated as `balance * interestRate` .
2. **CurrentAccount: withdraw(amount)**
    - **Purpose:** Override the withdrawal to allow withdrawing more than the current balance up to `balance + overdraftLimit` .
    - **Return:** Boolean – `true` if the withdrawal is successful, `false` if the requested amount exceeds this limit.

Manual Test Cases (Using if/else Logic)

- **Test Case 1: SavingsAccount Interest Calculation**
    - **Action:** Create a SavingsAccount with an initial balance of 1000 and an interest rate of 0.05.
    - **If/Else Check:**
        - **If** calling `calculateInterest()` returns 50 (i.e., 1000 * 0.05), **then** print “Interest calculated correctly.”
        - **Else** print “Error: Incorrect interest calculation.”
- **Test Case 2: CurrentAccount Withdrawal Within Overdraft Limit**
    - **Action:** Create a CurrentAccount with a balance of 1000 and an overdraft limit of 500. Attempt to withdraw 1300.
    - **If/Else Check:**
        - **If** `withdraw(1300)` returns `true` and the new balance is (1000 - 1300) = -300, **then** print “Overdraft withdrawal successful.”
        - **Else** print “Error: Withdrawal within overdraft limit failed.”
- **Test Case 3: CurrentAccount Withdrawal Exceeding Limit**
    - **Action:** From the same account, attempt to withdraw an amount that exceeds (balance +

overdraftLimit), for example, 2000.

- **If/Else Check:**
  - **If** `withdraw(2000)` returns `false` , **then** print “Properly rejected excessive withdrawal.”
  - **Else** print “Error: Excessive withdrawal was allowed.”

Project 3 (Day 3): Bank Class for Account Management & Transfer

Description

Develop a `Bank` class that manages a collection of various `Account` objects. This class will allow the addition of accounts and provide a method to transfer funds between any two accounts (demonstrating polymorphism as it handles multiple account types).

Attributes

- **accounts** (Collection of Account objects): A list or array holding all account instances managed by the bank.

Methods

1. **addAccount(account)**

   - **Purpose:** Add an `Account` (or any of its subclasses) to the bank’s collection.
   - **Return:** Void.

2. **findAccount(accountNumber)**

   - **Purpose:** Search for an account using its `accountNumber` .
   - **Return:** The found `Account` object or `null` if not found.

3. **transfer(fromAccountNumber, toAccountNumber, amount)**

   - **Purpose:** Facilitate a transfer by withdrawing the amount from the source account and depositing it into the target account.
   - **Return:** Boolean – `true` if the transfer is successful, `false` otherwise.
   - **Behavior:**
     - Use `findAccount` to get both accounts.
     - If the withdrawal from the source account succeeds, deposit the amount into the

target account.

Manual Test Cases (Using if/else Logic)

- **Test Case 1: Add and Find Account**
  - **Action:** Create two accounts (e.g., Account A with balance 1000 and Account B with balance 500) and add them to the Bank.
  - **If/Else Check:**
    - **If** `findAccount("A_Number")` returns Account A, **then** print "Account A found successfully."
    - **Else** print "Error: Account A not found."
- **Test Case 2: Successful Transfer**
  - **Action:** Transfer 300 from Account A (balance 1000) to Account B (balance 500).
  - **If/Else Check:**
    - **If** `transfer("A_Number", "B_Number", 300)` returns `true` , **then** check that Account A's balance is now 700 and Account B's is 800; print "Transfer successful."
    - **Else** print "Error: Transfer failed."
- **Test Case 3: Failed Transfer Due to Insufficient Funds**
  - **Action:** Attempt to transfer 1200 from Account A (balance 700 after the previous transfer) to Account B.
  - **If/Else Check:**
    - **If** `transfer("A_Number", "B_Number", 1200)` returns `false` , **then** print "Transfer rejected due to insufficient funds."
    - **Else** print "Error: Transfer should have failed but succeeded."

Project 4 (Day 4): Advanced Features – LoanAccount, Transaction Logging, & Person Class

Description

Enhance the system by introducing:

- A **LoanAccount** that models a loan, including repayment and interest calculation.
- A **Transaction** class to record each operation.
- A **Person** class to represent an account holder, who may own one or more bank accounts and

may have relationships (such as a spouse or co-owner).

Additional Attributes

- **LoanAccount:**
    - **loanAmount** (double): The total amount of the loan.
    - **interestRate** (double): The interest rate applied on the loan.
- **Transaction:**
    - **transactionID** (String): A unique identifier for the transaction.
    - **accountNumber** (String): The account involved.
    - **type** (String): The kind of transaction (e.g., “DEPOSIT”, “WITHDRAWAL”, “TRANSFER”).
    - **amount** (double): The transaction amount.
    - **transactionDate** (Date): The date/time of the transaction.
- **Person:**
    - **personID** (String): A unique identifier for the person.
    - **name** (String): The person’s name.
    - **accounts** (Collection of Account objects): Accounts owned by the person.
    - **relationships** (Collection of Person objects): Other persons related to this person (e.g., spouse, family members).

Additional Methods

1. **LoanAccount:**
    - **repay(amount)**
        - **Purpose:** Deduct a repayment amount from the outstanding loan.
        - **Parameters:** `amount` (double) – must be positive and less than or equal to the outstanding loan.
        - **Return:** Void.
    - **calculateInterest()**
        - **Purpose:** Compute the interest on the remaining loan.
        - **Return:** Interest amount (double).
2. **Person:**
    - **addAccount(account)**

        - **Purpose:** Link an account to the person's account list.
        - **Return:** Void.
    - **addRelationship(person)**
        - **Purpose:** Establish a relationship with another person (e.g., to represent joint account holders).
        - **Return:** Void.

3. **Transaction:**

    - **toString()**
        - **Purpose:** Provide a string representation of the transaction details.
        - **Return:** A formatted String showing date, type, amount, and account involved.

Manual Test Cases (Using if/else Logic)

- **Test Case 1: LoanAccount Repayment & Interest**

    - **Action:** Create a LoanAccount with a `loanAmount` of 5000 and an `interestRate` of 0.10.
    - **Steps:**
        - Call `repay(1000)` on the LoanAccount.
        - Then call `calculateInterest()` .
    - **If/Else Check:**
        - **If** the outstanding loan is now 4000 and `calculateInterest()` returns 400 (i.e., 4000 * 0.10), **then** print "Loan repayment and interest calculation successful."
        - **Else** print "Error: Loan repayment or interest calculation incorrect."

- **Test Case 2: Linking a Person with an Account**

    - **Action:** Create a Person with a specific `personID` and name. Then create an account (of any type) for that person.
    - **If/Else Check:**
        - **If** the person's account list contains the newly created account, **then** print "Account successfully linked to person."
        - **Else** print "Error: Account not linked."

- **Test Case 3: Transaction Logging**

    - **Action:** After performing a deposit or withdrawal, create a Transaction record capturing the operation details.
    - **If/Else Check:**

        - **If** the Transaction's string representation (via `toString()`) correctly displays the expected details, **then** print "Transaction logged correctly."
        - **Else** print "Error: Transaction log details are incorrect."

Project 5 (Day 5): Complete Command-Line Banking Application

Description

Integrate all previous components into a complete command-line application. This application should allow users to:

- Create Persons.
- Create various Account types (SavingsAccount, CurrentAccount, LoanAccount).
- Perform transactions such as deposits, withdrawals, transfers, and loan repayments.
- Log each transaction.
- Establish relationships between Persons (for joint accounts or family relationships).

The focus is on user interaction via the command line (using menus and prompts) and robust input validation (using if/else conditions).

Key Functional Methods

1. **createPerson()**
    - **Purpose:** Prompt the user for personal details and create a new Person.
2. **createAccount()**
    - **Purpose:** Allow the user to choose the account type and enter details (such as account number, initial balance, interest rate or overdraft limit) to create an account.
    - **Note:** Ensure the account is linked to an existing Person or create a new Person if needed.
3. **deposit(), withdraw(), transfer(), repayLoan()**
    - **Purpose:** Facilitate the corresponding banking operations by calling the methods already defined in the Account classes.
    - **Validation:**
        - Use if/else conditions to check for invalid input (e.g., negative amounts, insufficient funds).

4. **displayMenu()**
    - **Purpose:** Present a menu of options to the user and capture their selection.
5. **logTransaction()**
    - **Purpose:** Record every transaction (deposit, withdrawal, transfer, loan repayment) in a Transaction log.

Manual Test Cases (Using if/else Logic)

- **Test Case 1: Valid Deposit Operation**
    - **Action:**
        - User selects “Deposit” from the menu.
        - Enters a valid account number and a positive deposit amount.
    - **If/Else Check:**
        - **If** the deposit is processed and the account balance increases accordingly, **then** print “Deposit successful.”
        - **Else** print “Error: Deposit failed.”
- **Test Case 2: Invalid Withdrawal Operation**
    - **Action:**
        - User selects “Withdraw” and enters an amount greater than the account balance (or beyond overdraft limits for CurrentAccount).
    - **If/Else Check:**
        - **If** the system rejects the withdrawal (e.g., by returning false or displaying an error), **then** print “Withdrawal correctly rejected.”
        - **Else** print “Error: Withdrawal should not have been permitted.”
- **Test Case 3: Successful Transfer**
    - **Action:**
        - User selects “Transfer,” enters valid source and destination account numbers, and a transfer amount that is within the source’s available funds.
    - **If/Else Check:**
        - **If** the transfer is completed (source account balance decreases and destination account balance increases by the correct amount), **then** print “Transfer completed successfully.”
        - **Else** print “Error: Transfer did not complete as expected.”

- **Test Case 4: Loan Repayment and Interest Check**
    - **Action:**
        - User selects “Repay Loan” for a LoanAccount and enters a valid repayment amount.
    - **If/Else Check:**
        - **If** the outstanding loan amount is reduced correctly and the new interest (if recalculated) is correct, **then** print “Loan repayment successful.”
        - **Else** print “Error: Loan repayment processing failed.”
- **Test Case 5: Input Validation**
    - **Action:**
        - Simulate invalid inputs (e.g., non-existent account number, negative deposit amounts).
    - **If/Else Check:**
        - **If** the system catches the invalid input and displays an error message, **then** print “Input validation working.”
        - **Else** print “Error: Invalid input was not handled correctly.”