**ASSIGNMENT 3**

**Program code: PROG8580**

**Student Name: Saravanan Nadanasabesan**

**Student ID: 8949724**

Problem Description:

(*The Account class*) Design a class named Account that contains:

* A private int data field named id for the account (default 0).
* A private string data filed named first name for customer first name.
* A private string data filed named last name for customer last name.
* A private double data field named balance for the account (default 0).
* A private double data field named annualInterestRate that stores the current interest rate (default 0). Assume all accounts have the same interest rate.
* A private Date data field named dateCreated that stores the date when the account was created.
* A no-arg constructor that creates a default account.
* A constructor that creates an account with the specified id, first name, last name and initial balance.
* The accessor and mutator methods for id, name, balance, and annualInterestRate.
* The accessor method for dateCreated.
* A method named getMonthlyInterestRate() that returns the monthly interest rate.
* A method named withdraw that withdraws a specified amount from the account.
* A method named deposit that deposits a specified amount to the account.
* **You should not be able to instantiate object of type Account.** Create two subclasses for checking and saving accounts. A checking account has an overdraft limit, but a savings account cannot be overdrawn. Savings account has transaction fee applied for each transaction (withdraw / deposit).
* **ToString method should be used to display account info  
  ToString method should display all account info including overdraft limit and transaction fee.**

Draw the UML diagram for the classes. Implement the classes. Write a test program that creates a Checking and Savings Accounts objects with an account ID of 1122 & 1123, a balance of $20,000 & 100,000, and an annual interest rate of 4.5%. Use the withdraw method to withdraw $2,500, use the deposit method to deposit $3,000, and print the balance, the monthly interest, account type and the date when this account was created.

Test edge cases.

**Analysis:**

**Problem Statement:**

This assignment is to implement a banking application that manages customer accounts with distinct features for checking and savings accounts. The program must allow users to:

* Create and manage checking and savings accounts for the same customer.
* Perform basic banking operations like deposits and withdrawals.
* Enforce an overdraft limit for checking accounts, allowing customers to withdraw up to a set overdraft limit beyond their balance.
* charge a transaction fee for savings account operations (withdrawals and deposits).
* Display account information including interest rates, balances, and other details when requested.

**Assumptions:**

* The user provides all user inputs required for creating and managing accounts at runtime.
* Each customer will have both a checking and a savings account.
* The bank determines the annual interest rate, transaction fees, and overdraft limits as static constants.
* The program's operation continues until the user decides to exit.

**Inputs:**

1. The user needs to enter an integer for the Savings Account ID.
2. The user needs to enter an integer for the Checking Account ID.
3. The user needs to enter a string for their first name.
4. The user needs to enter a string for their last name.
5. The user needs to enter a real number for the initial Checking account balance.
6. The user needs to enter a real number for the initial Savings account balance.
7. The user needs to enter an integer to select the desired operation (e.g., deposit, withdrawal).
8. The user needs to enter a real number for the amount to deposit or withdraw.

**Validations:**

* **Input Type Validation:** Ensure all user inputs match the expected data types (e.g., integers for IDs, strings for names, real numbers for balances).
* **Positive Value Checks:** Account IDs and transaction amounts must be positive integers or real numbers.
* **Withdrawal Limits:** For checking accounts, allow withdrawals up to the available balance plus the overdraft limit, and for savings accounts, ensure withdrawal amounts do not exceed the balance minus the transaction fee.
* **Non-negative Balances:** Ensure balance checks and validations before processing transactions, including zero balances.

**Output:**

1. **User Prompts:** Provide clear prompts for data entry and instructions for selecting operations.
2. **Menu Options:** Offer a menu of operations (withdrawals, deposits, account summaries).
3. **Transaction and Validation Feedback:** Display relevant messages on successful transactions or validation errors (e.g., invalid input, insufficient funds).
4. **Account Summary Display:** Output formatted account details, including:
   * + Account ID
     + Name
     + Balance
     + Annual Interest Rate
     + Monthly Interest Rate
     + Date Created
     + Account Type (Checking/Savings)
     + Overdraft Limit or Transaction Fee

**Sample output:**

Enter Savings Account ID: 1123

Enter Checking Account ID: 1122

Enter First Name: selva

Enter Last Name: kumar

Enter Checking Balance: 30000

Enter Savings Balance: 200000

Select an operation:

1 - Withdraw from Savings

2 - Deposit into Savings

3 - Withdraw from Checking

4 - Deposit into Checking

5 - Print Account Statement

6 - Exit

Choose an option: 1

Enter amount to withdraw from Savings: 50000

Transaction successful. New balance: 149998.0

Select an operation:

1 - Withdraw from Savings

2 - Deposit into Savings

3 - Withdraw from Checking

4 - Deposit into Checking

5 - Print Account Statement

6 - Exit

Choose an option: 3

Enter amount to withdraw from Checking: 15000

Transaction successful. New balance: 15000.0

Select an operation:

1 - Withdraw from Savings

2 - Deposit into Savings

3 - Withdraw from Checking

4 - Deposit into Checking

5 - Print Account Statement

6 - Exit

Choose an option: 6

Exiting application and printing the final account summary.

Accounts Balance Summary:

Account ID: 1122

Name: selva kumar

Balance: $15000.0

Annual Interest Rate: 4.5%

Monthly Interest Rate: 0.375%

Date Created: Sun Nov 17 22:27:48 EST 2024

Account Type: Checking

Overdraft Limit: $2000.0

Account ID: 1123

Name: selva kumar

Balance: $149998.0

Annual Interest Rate: 4.5%

Monthly Interest Rate: 0.375%

Date Created: Sun Nov 17 22:27:48 EST 2024

Account Type: Savings

Transaction Fee: $2.0

**UML Diagram:**

A diagram of a company account

Description automatically generated

**Pseudo code:**

BEGIN

// Constants

SET ANNUAL\_INTEREST\_RATE = 4.5

SET OVERDRAFT\_LIMIT = 2000.0

SET TRANSACTION\_FEE = 2.0

// Main Method

FUNCTION main()

CALL Account.setAnnualInterestRate(ANNUAL\_INTEREST\_RATE)

INITIALIZE input AS Scanner

INITIALIZE savingsAccountId, checkingAccountId, checkingBalance, savingsBalance TO 0

INITIALIZE firstName, lastName AS null

SET validInput TO false

// Loop to collect and validate user input

WHILE validInput IS false

TRY

IF savingsAccountId IS 0 THEN

savingsAccountId = CALL validateInt(input, "Enter Savings Account ID:")

ENDIF

IF checkingAccountId IS 0 THEN

checkingAccountId = CALL validateInt(input, "Enter Checking Account ID:")

ENDIF

IF firstName IS null THEN

firstName = CALL validateString(input, "Enter First Name:")

ENDIF

IF lastName IS null THEN

lastName = CALL validateString(input, "Enter Last Name:")

ENDIF

IF checkingBalance IS 0 THEN

checkingBalance = CALL validateDouble(input, "Enter Checking Balance:")

ENDIF

IF savingsBalance IS 0 THEN

savingsBalance = CALL validateDouble(input, "Enter Savings Balance:")

ENDIF

// Create account objects

CREATE checkingAccount AS CheckingAccount(checkingAccountId, firstName, lastName, checkingBalance, OVERDRAFT\_LIMIT)

CREATE savingsAccount AS SavingsAccount(savingsAccountId, firstName, lastName, savingsBalance, TRANSACTION\_FEE)

// Perform operations on accounts

CALL performOperations(input, checkingAccount, savingsAccount)

// Print final account summary

CALL printAccountSummary(checkingAccount, savingsAccount)

SET validInput TO true

CATCH InputMismatchException

PRINT "Invalid input error message"

DISCARD input

CATCH Exception

PRINT "General error message"

ENDWHILE

END FUNCTION

// Function to perform operations

FUNCTION performOperations(input, checkingAccount, savingsAccount)

SET continueOperations TO true

WHILE continueOperations IS true

TRY

DISPLAY operation menu

SET option = CALL validateInt(input, "Choose an option:")

SWITCH (option)

CASE 1:

SET amount = CALL validateDouble(input, "Enter amount to withdraw from Savings:")

CALL savingsAccount.withdraw(amount)

CASE 2:

SET amount = CALL validateDouble(input, "Enter amount to deposit into Savings:")

CALL savingsAccount.deposit(amount)

CASE 3:

SET amount = CALL validateDouble(input, "Enter amount to withdraw from Checking:")

CALL checkingAccount.withdraw(amount)

CASE 4:

SET amount = CALL validateDouble(input, "Enter amount to deposit into Checking:")

CALL checkingAccount.deposit(amount)

CASE 5:

CALL printAccountSummary(checkingAccount, savingsAccount)

CASE 6:

SET continueOperations TO false

PRINT "Exiting application"

DEFAULT:

PRINT "Invalid selection"

ENDSWITCH

CATCH InputMismatchException

PRINT "Invalid input error message"

DISCARD input

CATCH IllegalArgumentException

PRINT "Invalid argument error message"

ENDWHILE

END FUNCTION

// Input validation functions

FUNCTION validateInt(input, prompt)

PRINT prompt

IF input IS NOT valid integer THEN

THROW InputMismatchException("Invalid input. Enter a valid integer.")

ENDIF

SET value TO input

IF value >= 0 THEN

RETURN value

ELSE

THROW InputMismatchException("Invalid input. Enter a positive integer.")

ENDIF

END FUNCTION

FUNCTION validateString(input, prompt)

PRINT prompt

RETURN trimmed input

END FUNCTION

FUNCTION validateDouble(input, prompt)

WHILE true

PRINT prompt

IF input IS NOT valid double THEN

THROW InputMismatchException("Invalid input. Enter a valid double.")

ENDIF

SET amount TO input

IF amount >= 0 THEN

RETURN amount

ELSE

THROW InputMismatchException("Amount cannot be negative.")

ENDIF

ENDWHILE

END FUNCTION

// Function to print account summaries

FUNCTION printAccountSummary(checkingAccount, savingsAccount)

PRINT "Accounts Balance Summary"

PRINT checkingAccount

PRINT savingsAccount

END FUNCTION

END

**Flow Chart:**

A black screen with white squares

Description automatically generated

A black and white diagram

Description automatically generated

A diagram of a function

Description automatically generated with medium confidence A black background with white squares

Description automatically generated A white rectangle with black text

Description automatically generated A white rectangular sign with black text

Description automatically generated

**Coding:**

1. **Account:**

**package** Assignment03;

**import** java.util.Date;

**public** **abstract** **class** Account {

**private** **int** id;

**private** String firstName;

**private** String lastName;

**private** **double** balance;

**private** **static** **double** *annualInterestRate*;

**private** Date dateCreated;

// No-argument constructor

**public** Account() {

**this**.dateCreated = **new** Date();

}

// Constructor with input parameters

**public** Account(**int** id, String firstName, String lastName, **double** balance) {

**this**.setId(id);

**this**.setFirstName(firstName);

**this**.setLastName(lastName);

**this**.setBalance(balance);

**this**.dateCreated = **new** Date();

}

// Setter methods

**public** **void** setId(**int** id) {

**if** (id <= 0) {

**throw** **new** IllegalArgumentException("ID must be a positive integer.");

}

**this**.id = id;

}

**public** **void** setFirstName(String firstName) {

**if** (firstName == **null** || firstName.trim().isEmpty()) {

**throw** **new** IllegalArgumentException("First name cannot be empty.");

}

**this**.firstName = firstName;

}

**public** **void** setLastName(String lastName) {

**if** (lastName == **null** || lastName.trim().isEmpty()) {

**throw** **new** IllegalArgumentException("Last name cannot be empty.");

}

**this**.lastName = lastName;

}

**public** **void** setBalance(**double** balance) {

**if** (balance < 0) {

**throw** **new** IllegalArgumentException("Balance cannot be negative.");

}

**this**.balance = balance;

}

**public** **static** **void** setAnnualInterestRate(**double** rate) {

*annualInterestRate* = rate;

}

// Getter methods

**public** **int** getId() {

**return** id;

}

**public** String getFirstName() {

**return** firstName;

}

**public** String getLastName() {

**return** lastName;

}

**public** **double** getBalance() {

**return** balance;

}

**public** **static** **double** getAnnualInterestRate() {

**return** *annualInterestRate*;

}

**public** Date getDateCreated() {

**return** dateCreated;

}

**public** **double** getMonthlyInterestRate() {

**return** *annualInterestRate* / 12;

}

// Abstract methods for withdrawal and deposit

**public** **abstract** **void** withdraw(**double** amount);

**public** **abstract** **void** deposit(**double** amount);

// Overridden toString method

@Override

**public** String toString() {

**return** "Account ID: " + id + "\n" +

"Name: " + firstName + " " + lastName + "\n" +

"Balance: $" + balance + "\n" +

"Annual Interest Rate: " + *annualInterestRate* + "%\n" +

"Monthly Interest Rate: " + getMonthlyInterestRate() + "%\n" +

"Date Created: " + dateCreated + "\n";

}

}

1. **CheckingAccount:**

**package** Assignment03;

**public** **class** CheckingAccount **extends** Account {

**private** **double** overdraftLimit;

// Constructor with input parameters

**public** CheckingAccount(**int** id, String firstName, String lastName, **double** balance, **double** overdraftLimit) {

**super**(id, firstName, lastName, balance);

**this**.overdraftLimit = overdraftLimit;

}

// Setter method for overdraft limit

**public** **void** setOverdraftLimit(**double** overdraftLimit) {

**this**.overdraftLimit = overdraftLimit;

}

// Getter method for overdraft limit

**public** **double** getOverdraftLimit() {

**return** overdraftLimit;

}

// Override withdraw method

@Override

**public** **void** withdraw(**double** amount) {

**if** (getBalance() == -overdraftLimit) {

**throw** **new** IllegalArgumentException("Cannot withdraw. Balance has reached the overdraft limit.");

} **else** **if** (amount < 0) {

**throw** **new** IllegalArgumentException("Withdrawal amount must be positive.");

} **else** **if** (amount <= getBalance() + overdraftLimit) {

setBalance(getBalance() - amount);

System.***out***.println("Transaction successful. New balance: " + getBalance());

} **else** {

**throw** **new** IllegalArgumentException("Withdrawal exceeds available balance plus overdraft limit: " + (getBalance() + overdraftLimit));

}

}

// Override deposit method

@Override

**public** **void** deposit(**double** amount) {

**if** (amount < 0) {

**throw** **new** IllegalArgumentException("Deposit amount must be positive.");

} **else** {

setBalance(getBalance() + amount);

System.***out***.println("Transaction successful. New balance: " + getBalance());

}

}

// Overridden toString method

@Override

**public** String toString() {

**return** **super**.toString() + "Account Type: Checking\n" +

"Overdraft Limit: $" + overdraftLimit + "\n";

}

}

1. **SavingsAccount:**

**package** Assignment03;

**public** **class** SavingsAccount **extends** Account {

**private** **double** transactionFee;

// Constructor with input parameters

**public** SavingsAccount(**int** id, String firstName, String lastName, **double** balance, **double** transactionFee) {

**super**(id, firstName, lastName, balance);

**this**.transactionFee = transactionFee;

}

// Setter method for transaction fee

**public** **void** setTransactionFee(**double** transactionFee) {

**this**.transactionFee = transactionFee;

}

// Getter method for transaction fee

**public** **double** getTransactionFee() {

**return** transactionFee;

}

// Override withdraw method

@Override

**public** **void** withdraw(**double** amount) {

**if** (amount < 0) {

**throw** **new** IllegalArgumentException("Withdrawal amount must be positive.");

} **else** **if** (amount + transactionFee <= getBalance()) {

setBalance(getBalance() - amount - transactionFee);

System.***out***.println("Transaction successful. New balance: " + getBalance());

} **else** {

**throw** **new** IllegalArgumentException("Insufficient funds including transaction fee.");

}

}

// Override deposit method

@Override

**public** **void** deposit(**double** amount) {

**if** (amount < 0) {

**throw** **new** IllegalArgumentException("Deposit amount must be positive.");

} **else** {

setBalance(getBalance() + amount - transactionFee);

System.***out***.println("Transaction successful. New balance: " + getBalance());

}

}

// Overridden toString method

@Override

**public** String toString() {

**return** **super**.toString() + "Account Type: Savings\n" +

"Transaction Fee: $" + transactionFee + "\n";

}

1. **TestAccount:**

**package** Assignment03;

**import** java.util.Scanner;

**public** **class** TestAccount {

**private** **static** **final** **double** ***ANNUAL\_INTEREST\_RATE*** = 4.5;

**private** **static** **final** **double** ***OVERDRAFT\_LIMIT*** = 2000.0;

**private** **static** **final** **double** ***TRANSACTION\_FEE*** = 2.0;

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

Account.*setAnnualInterestRate*(***ANNUAL\_INTEREST\_RATE***); // Set the annual interest rate for all accounts

Scanner input = **new** Scanner(System.***in***); // Scanner for user input

**int** savingsAccountId = 0;

**int** checkingAccountId = 0;

String firstName = **null**;

String lastName = **null**;

**double** checkingBalance = 0;

**double** savingsBalance = 0;

**boolean** validInput = **false**;

// Collect user input

**while** (!validInput) {

**try** {

**if** (savingsAccountId == 0) {

savingsAccountId = *validateInt*(input, "Enter Savings Account ID: ");

}

**if** (checkingAccountId == 0) {

checkingAccountId = *validateInt*(input, "Enter Checking Account ID: ");

}

**if** (firstName == **null**) {

firstName = *validateString*(input, "Enter First Name: ");

}

**if** (lastName == **null**) {

lastName = *validateString*(input, "Enter Last Name: ");

}

**if** (checkingBalance == 0) {

checkingBalance = *validateDouble*(input, "Enter Checking Balance: ");

}

**if** (savingsBalance == 0) {

savingsBalance = *validateDouble*(input, "Enter Savings Balance: ");

}

// Create the CheckingAccount and SavingsAccount

CheckingAccount checkingAccount = **new** CheckingAccount(checkingAccountId, firstName, lastName, checkingBalance, ***OVERDRAFT\_LIMIT***);

SavingsAccount savingsAccount = **new** SavingsAccount(savingsAccountId, firstName, lastName, savingsBalance, ***TRANSACTION\_FEE***);

// Perform operations

*performOperations*(input, checkingAccount, savingsAccount);

// Print final account summaries

*printAccountSummary*(checkingAccount, savingsAccount);

validInput = **true**; // Input successfully validated

} **catch** (InputMismatchException ime) {

System.***out***.println(ime.getMessage());

input.next(); // Discard invalid input

} **catch** (Exception e) {

System.***out***.println(e.getMessage());

}

}

}

// Method for performing operations like withdrawal, deposit, etc.

**public** **static** **void** performOperations(Scanner input, CheckingAccount checkingAccount, SavingsAccount savingsAccount) {

**boolean** continueOperations = **true**;

**while** (continueOperations) {

**try** {

// Display menu options

System.***out***.println("Select an operation:\n1 - Withdraw from Savings\n2 - Deposit into Savings\n3 - Withdraw from Checking\n4 - Deposit into Checking\n5 - Print Account Statement\n6 - Exit");

**int** option = *validateInt*(input, "Choose an option: ");

**double** amount;

// Switch case to handle different options

**switch** (option) {

**case** 1:

amount = *validateDouble*(input, "Enter amount to withdraw from Savings: ");

savingsAccount.withdraw(amount); // Perform withdrawal

**break**;

**case** 2:

amount = *validateDouble*(input, "Enter amount to deposit into Savings: ");

savingsAccount.deposit(amount); // Perform deposit

**break**;

**case** 3:

amount = *validateDouble*(input, "Enter amount to withdraw from Checking: ");

checkingAccount.withdraw(amount); // Perform withdrawal

**break**;

**case** 4:

amount = *validateDouble*(input, "Enter amount to deposit into Checking: ");

checkingAccount.deposit(amount); // Perform deposit

**break**;

**case** 5:

*printAccountSummary*(checkingAccount, savingsAccount); // Print account details

**break**;

**case** 6:

continueOperations = **false**; // Exit the loop

System.***out***.println("Exiting application and printing the final account summary.");

**break**;

**default**:

System.***out***.println("Invalid selection. Try again.");

}

}

//exceptional handling for input mis-match

**catch**(InputMismatchException ime) { System.***out***.println(ime.getMessage());

System.***out***.println("----Returning to the operation selection----"); input.next();

//exceptional handling for illegal arguments.

}**catch** (IllegalArgumentException iae) { System.***out***.println(iae.getMessage());

System.***out***.println("----Returning to the operation selection----");

}

}

}

// Method to validate and get an integer input

**public** **static** **int** validateInt(Scanner input, String prompt) {

System.***out***.print(prompt);

**if** (!input.hasNextInt()) {

**throw** **new** InputMismatchException("Invalid input. Please enter a valid integer.");

}

**int** value = input.nextInt();

**if** (value >= 0) {

**return** value;

}

**throw** **new** InputMismatchException("Invalid input. Please enter a positive integer.");

}

// Method to validate and get a string input

**public** **static** String validateString(Scanner input, String prompt) {

System.***out***.print(prompt);

**return** input.next().trim();

}

// Method to validate and get a double input

**public** **static** **double** validateDouble(Scanner input, String prompt) {

**double** amount;

**while** (**true**) {

System.***out***.print(prompt);

// Check if the next input is a valid double

**if** (!input.hasNextDouble()) {

**throw** **new** InputMismatchException("Invalid input. Please enter a valid integer.");

}

amount = input.nextDouble();

// Check if the amount is positive

**if** (amount >= 0) {

**return** amount; // Return the valid input

} **else** {

**throw** **new** InputMismatchException("Amount cannot be negative. Please enter a valid positive number.");

}

}

}

// Method to print account summary

**public** **static** **void** printAccountSummary(CheckingAccount checkingAccount, SavingsAccount savingsAccount) {

System.***out***.println("Accounts Balance Summary:");

System.***out***.println(checkingAccount); // Print Checking account details

System.***out***.println(savingsAccount); // Print Savings account details

}

}

**Testing:**

Case 1: Withdraw and Deposit without Exception

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

Case 2: Withdraw and Deposit with Exception

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

Case 3: Input Validation.

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated