BONIFACE MWANGI WARIGI

REG NO: SCT212-0726/2022

UNIT CODE: BIT 2203

UNIT: ADVANCED PROGRAMMING

ASSIGNMENT ONE

Question 1 - Extending Interface in Concrete Class

ANSWER

To solve the problem, we will follow these steps:

1. **Define the Interface**: First, we need to define the TransactionInterface which will declare the methods that must be implemented by any class that implements this interface.

```
import java.util.Calendar;


public interface TransactionInterface {
   double getAmount();
   Calendar getDate();
   String getTransactionID();
   void printTransactionDetails();
   void apply(BankAccount ba);
}
```

2. **Create the BaseTransaction Class**: Next, we create the BaseTransaction class that implements the TransactionInterface. This class will provide concrete implementations of the methods declared in the interface.

```
public class BaseTransaction implements TransactionInterface {
   private double amount;
   private Calendar date;
   private String transactionID;


   public BaseTransaction(double amount, Calendar date, String transactionID) {
      this.amount = amount;
      this.date = date;
```

```
        this.transactionID = transactionID;
    }

    @Override
    public double getAmount() {
        return amount;
    }

    @Override
    public Calendar getDate() {
        return date;
    }

    @Override
    public String getTransactionID() {
        return transactionID;
    }

    @Override
    public void printTransactionDetails() {
        System.out.println("Transaction ID: " + transactionID);
        System.out.println("Date: " + date.getTime());
        System.out.println("Amount: " + amount);
    }

    @Override
    public void apply(BankAccount ba) {
        // Basic implementation, to be overridden in derived classes
        System.out.println("Applying base transaction.");
    }
}
```

3. **Implement Derived Classes**: We will now create two derived classes, DepositTransaction and WithdrawalTransaction, which extend BaseTransaction and override the apply() method.

```
public class DepositTransaction extends BaseTransaction {
    public DepositTransaction(double amount, Calendar date, String transactionID) {
        super(amount, date, transactionID);
    }

    @Override
    public void apply(BankAccount ba) {
        ba.deposit(getAmount());
        System.out.println("Deposited: " + getAmount());
    }
}

public class WithdrawalTransaction extends BaseTransaction {
    public WithdrawalTransaction(double amount, Calendar date, String transactionID) {
        super(amount, date, transactionID);
    }

    @Override
    public void apply(BankAccount ba) {
        ba.withdraw(getAmount());
        System.out.println("Withdrawn: " + getAmount());
    }
}
```

4. **BankAccount Class**: We assume a simple BankAccount class exists which has deposit() and withdraw() methods.

```
public class BankAccount {
    private double balance;

    public BankAccount(double balance) {
```

```
        this.balance = balance;
    }

    public void deposit(double amount) {
        balance += amount;
    }

    public void withdraw(double amount) {
        balance -= amount;
    }

    public double getBalance() {
        return balance;
    }
}
```

5. **Testing the Implementation**: Finally, we can create a simple test to demonstrate the functionality.

```
import java.util.GregorianCalendar;

public class Main {
    public static void main(String[] args) {
        BankAccount ba = new BankAccount(1000);
        BaseTransaction deposit = new DepositTransaction(200, new GregorianCalendar(2023, 2, 25), "TX1001");
        BaseTransaction withdrawal = new WithdrawalTransaction(150, new GregorianCalendar(2023, 2, 26), "TX1002");

        deposit.apply(ba);
        withdrawal.apply(ba);

        deposit.printTransactionDetails();
        withdrawal.printTransactionDetails();
```

```
        System.out.println("Final Balance: " + ba.getBalance());
    }
}
```

Question 2 - Differentiate functionality of DepositTransaction and WithdrawalTransaction

Answer

To solve the problem, we will implement two classes, DepositTransaction and WithdrawalTransaction, which are subclasses of a hypothetical superclass Transaction. These classes will interact with another class, BankAccount, which maintains the balance of a bank account. The key functionality to implement is the reverse() method in the WithdrawalTransaction class, which will restore the balance of the bank account to its state before the withdrawal was made. Deposits, once made, are irreversible according to the problem statement.

1. Define the BankAccount class:
    - This class will have at least one attribute, balance, to store the current balance.
    - It will have methods to deposit(amount) and withdraw(amount) to modify the balance.
2. Define the Transaction superclass:
    - This class might include attributes such as amount and a reference to the BankAccount it affects.
    - It could also include a method apply() to apply the transaction.
3. Define the DepositTransaction subclass:
    - This class inherits from Transaction.
    - The apply() method will call the deposit(amount) method of BankAccount.
4. Define the WithdrawalTransaction subclass:
    - This class also inherits from Transaction.
    - The apply() method will call the withdraw(amount) method of BankAccount.
    - It will include a reverse() method that restores the original balance by depositing the withdrawn amount back into the account.
5. Implement the reverse() method in WithdrawalTransaction:

- Check if the transaction has already been reversed to prevent double reversal.
- If not reversed, call the deposit(amount) on the associated BankAccount to restore the withdrawn amount.
- Mark the transaction as reversed.

6. Testing:
   - Create an instance of BankAccount.
   - Perform a series of deposits and withdrawals.
   - Attempt to reverse a withdrawal and check if the balance is correctly restored.

```
class BankAccount {
    private double balance;

    public BankAccount(double initialBalance) {
        this.balance = initialBalance;
    }

    public void deposit(double amount) {
        balance += amount;
    }

    public void withdraw(double amount) {
        balance -= amount;
    }

    public double getBalance() {
        return balance;
    }
}

abstract class Transaction {
    protected BankAccount account;
```

```
    protected double amount;

    public Transaction(BankAccount account, double amount) {
        this.account = account;
        this.amount = amount;
    }

    abstract void apply();
}

class DepositTransaction extends Transaction {
    public DepositTransaction(BankAccount account, double amount) {
        super(account, amount);
    }

    @Override
    void apply() {
        account.deposit(amount);
    }
}

class WithdrawalTransaction extends Transaction {
    private boolean reversed = false;

    public WithdrawalTransaction(BankAccount account, double amount) {
        super(account, amount);
    }

    @Override
    void apply() {
        account.withdraw(amount);
```

```
    }

    public boolean reverse() {
        if (!reversed) {
            account.deposit(amount);
            reversed = true;
            return true;
        }
        return false;
    }
}
```

Question 3 - Exception Handling and Client Codes

ANSWER

To solve the problem, we will follow these steps:

1. **Create the Exception Class**:
    - Define a custom exception class named InsufficientFundsException that extends the Java Exception class. This will allow us to throw this exception when specific conditions (like insufficient funds) are met during bank transactions.
2. **Implement the WithdrawalTransaction Class**:
    - Create a class named WithdrawalTransaction that will handle withdrawal operations.
    - Implement the apply() method which will throw the InsufficientFundsException if the withdrawal amount is greater than the account balance.
    - Overload the apply() method to handle cases where the balance is positive but less than the withdrawal amount, allowing the withdrawal of the entire balance and recording the amount not withdrawn.
3. **Exception Handling**:
    - Use a try-catch-finally block within the apply() method to handle exceptions and ensure that all necessary actions are taken regardless of whether an exception occurs.

      Detailed Implementation:

      Step 1: Creating the InsufficientFundsException Class

```
public class InsufficientFundsException extends Exception {
    public InsufficientFundsException(String message) {
        super(message);
    }
}
```

Step 2: Implementing the WithdrawalTransaction Class

```
public class WithdrawalTransaction {
    private BankAccount account;

    public WithdrawalTransaction(BankAccount account) {
        this.account = account;
    }

    // Method to apply withdrawal and handle exceptions
    public void apply(double amount) throws InsufficientFundsException {
        if (amount > account.getBalance()) {
            throw new InsufficientFundsException("Insufficient funds for the
requested withdrawal.");
        }
        account.withdraw(amount);
    }

    // Overloaded method to handle partial withdrawals
    public void apply(double amount, boolean allowPartial) {
        try {
            if (account.getBalance() > 0 && account.getBalance() < amount) {
                System.out.println("Partial withdrawal of available balance: " +
account.getBalance());
                account.withdraw(account.getBalance()); // Withdraw all available
balance
```

```
                double shortfall = amount - account.getBalance();
                System.out.println("Amount not withdrawn due to insufficient
funds: " + shortfall);
            } else {
                apply(amount); // Call the original apply method
            }
        } catch (InsufficientFundsException e) {
            System.out.println(e.getMessage());
        } finally {
            System.out.println("Transaction completed.");
        }
    }
}
```

Step 3: Exception Handling

The exception handling is integrated within the apply(double amount, boolean allowPartial) method using a try-catch-finally block. The try block attempts the withdrawal, the catch block handles any InsufficientFundsException thrown when funds are insufficient, and the finally block ensures that the transaction completion message is printed regardless of the outcome.

Question 4 - Writing the Client Code

ANSWER

To solve the problem described in the question, we need to write client code that tests the functionality
of DepositTransaction and WithdrawalTransaction classes, which are presumably subclasses of a superclass (perhaps Transaction). We will follow these steps:

1. **Assume Class Definitions**: Since the actual implementations of DepositTransaction, WithdrawalTransaction, and their superclass Transaction are not provided, we will assume typical behaviors:
    - Transaction: A base class with an apply() method that might modify an account balance.
    - DepositTransaction: A subclass of Transaction that increases the account balance.

- WithdrawalTransaction: A subclass of Transaction that decreases the account balance.

2. **Create Test Classes**: We will create simple implementations of these classes to use in our testing.

```
class Transaction {
    protected double amount;

    public Transaction(double amount) {
        this.amount = amount;
    }

    public void apply(Account account) {
        // Base implementation does nothing
    }
}

class DepositTransaction extends Transaction {
    public DepositTransaction(double amount) {
        super(amount);
    }

    @Override
    public void apply(Account account) {
        account.balance += this.amount;
    }
}

class WithdrawalTransaction extends Transaction {
    public WithdrawalTransaction(double amount) {
        super(amount);
    }
```

```
    @Override
    public void apply(Account account) {
        account.balance -= this.amount;
    }
}

class Account {
    public double balance;

    public Account(double initialBalance) {
        this.balance = initialBalance;
    }
}
```

3. **Write Main Class to Test**: We will write a Main class that creates instances of DepositTransaction and WithdrawalTransaction, and tests their apply() methods.

```
public class Main {
    public static void main(String[] args) {
        Account myAccount = new Account(1000);  // Starting with $1000

        // Test DepositTransaction
        Transaction deposit = new DepositTransaction(200);
        deposit.apply(myAccount);
        System.out.println("Balance after deposit: " + myAccount.balance);  // Expected: 1200

        // Test WithdrawalTransaction
        Transaction withdrawal = new WithdrawalTransaction(150);
        withdrawal.apply(myAccount);
        System.out.println("Balance after withdrawal: " + myAccount.balance);  // Expected: 1050
```

```
        // Demonstrating polymorphism
        Transaction genericTransaction;
        genericTransaction = deposit;  // Referencing DepositTransaction
        genericTransaction.apply(myAccount);
        System.out.println("Balance after second deposit: " +
myAccount.balance);  // Expected: 1400


        genericTransaction = withdrawal;  // Referencing WithdrawalTransaction
        genericTransaction.apply(myAccount);
        System.out.println("Balance after second withdrawal: " +
myAccount.balance);  // Expected: 1250
    }
}
```

4. **Explanation of Code**:
    - An Account object is created with an initial balance of $1000$1000.
    - A DepositTransaction object is created and applied to the account, increasing the balance by $200$200.
    - A WithdrawalTransaction object is created and applied, decreasing the balance by $150$150.
    - Polymorphism is demonstrated by using a Transaction reference to point to both a DepositTransaction and a WithdrawalTransaction, showing that the correct apply() method is called in each case.
5. **Final Output**:
    - After the first deposit: $1200$1200
    - After the first withdrawal: $1050$1050
    - After the second deposit: $1400$1400
    - After the second withdrawal: $1250$1250