JOHN MWEGA MAINA

REG NO: SCT212-0055/2021
UNIT CODE: BITZ2203
UNIT: ADVANCED PROGRAMMING
ASSIGNMENT ONE

Question 1 - Extending Interface in Concrete Class

Answer:

Define the Interface: The TransactionInterface declares the methods that any class implementing this interface must provide.

```
import java.util.Calendar;

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

2. **Create the BaseTransaction Class**: The BaseTransaction class implements the TransactionInterface, providing concrete implementations for the methods declared in the interface.

```java
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;
    }
```

```java
    @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);
    }
```

```java
    @Override
    public void apply(BankAccount ba) {
        System.out.println("Applying base transaction.");
    }
}
```

3. **Implement Derived Classes**: The DepositTransaction and WithdrawalTransaction classes extend BaseTransaction and override the apply() method to provide specific functionality for deposit and withdrawal operations.

```
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**: This class holds the account balance and provides methods to deposit and withdraw money.

```java
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**: We can now test the implementation by creating a BankAccount instance and applying transactions.

```java
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,
        BaseTransaction withdrawal = new WithdrawalTransaction(150, new GregorianCalendar(

        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:

1. **DepositTransaction**: This class handles the functionality of depositing money into a BankAccount. When the apply() method is called, it increases the account balance by the specified amount.
2. **WithdrawalTransaction**: This class handles the functionality of withdrawing money from a BankAccount. The apply() method decreases the account balance by the specified amount. Additionally, it can include a reverse() method to restore the account balance if a withdrawal needs to be undone.

Question 3 - Exception Handling and Client Codes

Answer:

1. **Create the Exception Class**: We define a custom exception called InsufficientFundsException to be thrown when a withdrawal amount exceeds the account balance.

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

2. **Implement the WithdrawalTransaction Class**: The apply() method throws an InsufficientFundsException if the withdrawal amount exceeds the account balance. Additionally, we create an overloaded apply() method to handle partial withdrawals.

```java
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 wit
        }
        account.withdraw(amount);
    }
```

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

3. **Exception Handling**: The exception handling ensures that any InsufficientFundsException is caught and managed, allowing for the transaction to complete without failure.

Question 4 - Writing the Client Code

Answer:

1. **Create Test Classes**: Assuming typical implementations of Transaction, DepositTransaction, and WithdrawalTransaction, we create simple test classes for them.

```
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;
    }
}
```

2. **Write Main Class to Test**: We write a Main class to test the functionality of the DepositTransaction and WithdrawalTransaction.

```
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: 12

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

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

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

3. **Explanation of Code**: This client code tests the functionality of DepositTransaction and WithdrawalTransaction by creating instances and applying them to the account. It also demonstrates polymorphism by using a Transaction reference to invoke methods of both DepositTransaction and WithdrawalTransaction.
4. **Final Output**:
    - After the first deposit: $1200
    - After the first withdrawal: $1050
    - After the second deposit: $1400

- After the second withdrawal: $1250