Name: Ayush Padhy

Assignment: 3-Banking System

TASK 1: CREATING CLASSES

1. Customer Class:

```python
class Customer:
    def __init__(self, customerID: int, firstName: str, lastName: str, email: str, phone: str, address: str):
        self.customerID = customerID
        self.firstName = firstName
        self.lastName = lastName
        self.email = email
        self.phone = phone
        self.address = address

    4 usages (1 dynamic)
    @property
    def customerID(self):
        return self.customerID

    @property
    def firstName(self):
        return self.firstName

    @property
    def lastName(self):
        return self.lastName

    @property
    def email(self):
        return self.email

    @property
    def phone(self):
        return self.phone

    @property
    def address(self):
        return self.address

    3 usages (1 dynamic)
```

```python
    3 usages (1 dynamic)
    @customerID.setter
    def customerID(self, customerID):
        self.customerID = customerID

    @firstName.setter
    def firstName(self, firstName):
        self.firstName = firstName

    @lastName.setter
    def lastName(self, lastName):
        self.lastName = lastName

    @email.setter
    def email(self, email):
        self.email = email

    @phone.setter
    def phone(self, phone):
        self.phone = phone

    @address.setter
    def address(self, address):
        self.address = address

    def getCustomerDetails(self):
        print("Customer Details: ")
        print(f"ID: {self.customerID}")
        print(f"Name: {self.firstName} {self.lastName}")
        print(f"Email: {self.email}")
        print(f"Phone: {self.phone}")
        print(f"Address: {self.address}")
```

2. Accounts Class:

```python
class Account(Customer):
    def __init__(self, account_num: int, account_type: str, balance: float, lastAccNo=None, customerID=None,
                 lastName=None, email=None, firstName=None, phone=None):
        super().__init__(customerID, firstName, lastName, email, phone, address)
        self.account_num = account_num
        self.account_type = account_type
        self.balance = balance
        self.lastAccNo = lastAccNo

    @property
    def accountNum(self):
        return self.account_num

    1 usage
    @property
    def accountType(self):
        return self.account_type

    @property
    def balance(self):
        return self.balance

    2 usages
    @property
    def lastAccNo(self):
        return self.lastAccNo

    @accountNum.setter
    def accountNum(self, accountNum):
        self.account_num = accountNum
```

```python
    @accountType.setter
    def accountType(self, type):
        self.account_type = type

    @balance.setter
    def balance(self, bal):
        self.balance = bal

    2 usages
    @lastAccNo.setter
    def lastAccNo(self, lastAccNo):
        self.lastAccNo = lastAccNo

    def getAccountDetails(self):
        print("Account Details: ")
        print(f"Account Number: {self.account_num}")
        print(f"Account Type: {self.account_type}")
        print(f"Customer ID: {self.customerID}")
        print(f"Customer Name: {self.firstName} {self.lastName}")
        print(f"Balance: {self.balance}")

    1 usage (1 dynamic)
    def depositAmount(self, amount):
        if amount > 0:
            self.balance += amount
            return self.balance
        else:
            return "Invalid"

    1 usage (1 dynamic)
    def withdrawAmount(self, amount):
        if amount > self.balance:
            return "Insufficient Balance"
        else:
            self.balance -= amount
```

3. Transaction Class:

```python
class Transaction:
    def __init__(self, account, desctiption, dateTime, transactionType, transactionAmount):
        self.account = account
        self.description = desctiption
        self.dateTime = dateTime
        self.transactionType = transactionType
        self.transactionAmount = transactionAmount
        self.type = ("Withdraw", "Deposit", "Transfer")

    def getTransactionDetails(self):
        print("Transaction Details: ")
        print(f"Account ID: {self.account}")
        print(f"Description: {self.description}")
        print(f"Date and Time: {self.dateTime}")
        print(f"Transaction Type: {self.transactionType}")
        print(f"Transaction Amount {self.transactionAmount}")
```

TASK 2: CREATING SUBCLASSES

1. Creating Savings Account:

```python
from PyCharm.Assignments.BankingSystemFinal.Bean.Account import Account


class SavingsAccount(Account):
    def __init__(self, accountNum):
        super().__init__(accountNum, account_type: 'Savings', balance: 500)
        self.interestRate = 4.5
        print("Savings Account Activated!!!")

    def calculateInterest(self):
        interest = self.balance * (self.interestRate / 100)
        self.balance += interest
        print(f"After {interest} Interest, account balance is: {self.balance}")
```

2. Current Account:

```python
from PyCharm.Assignments.BankingSystemFinal.Bean.Account import Account


2 usages
class CurrentAccount(Account):
    LIMIT = 5000
    def __init__(self, accountNum, balance):
        super().__init__(accountNum, account_type: 'Current', balance)
        self.overdraftLimit = self.LIMIT
        print("Current Account Activated!!!")
```

3. Zero Balance Account:

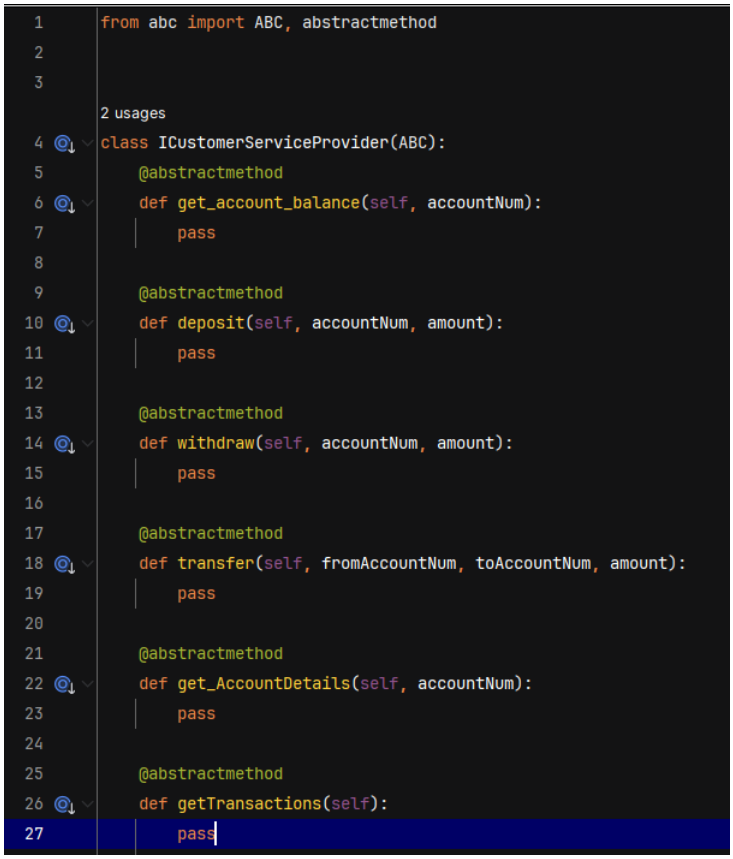
```python
from PyCharm.Assignments.BankingSystemFinal.Bean.Account import Account


class ZeroBalanceAccount(Account):
    def __init__(self, accountNum):
        super().__init__(accountNum, account_type: 'Zero Balance', balance: 0)
        print("Zero Balance Account Activated!!!")
```

TASK 3: CREATE ABSTRACT CLASSES

1. ICustomerServiceProvider class:

```python
from abc import ABC, abstractmethod


2 usages
class ICustomerServiceProvider(ABC):
    @abstractmethod
    def get_account_balance(self, accountNum):
        pass

    @abstractmethod
    def deposit(self, accountNum, amount):
        pass

    @abstractmethod
    def withdraw(self, accountNum, amount):
        pass

    @abstractmethod
    def transfer(self, fromAccountNum, toAccountNum, amount):
        pass

    @abstractmethod
    def get_AccountDetails(self, accountNum):
        pass

    @abstractmethod
    def getTransactions(self):
        pass
```

2. IBankServiceProvider:



```python
from abc import ABC, abstractmethod


2 usages
class ICustomerServiceProvider(ABC):
    @abstractmethod
    def get_account_balance(self, accountNum):
        pass

    @abstractmethod
    def deposit(self, accountNum, amount):
        pass

    @abstractmethod
    def withdraw(self, accountNum, amount):
        pass

    @abstractmethod
    def transfer(self, fromAccountNum, toAccountNum, amount):
        pass

    @abstractmethod
    def get_AccountDetails(self, accountNum):
        pass

    @abstractmethod
    def getTransactions(self):
        pass
```

TASK 4: IMPLEMENTATION CLASS

1. CustomerServiceProviderImpl:

```python
from PyCharm.Assignments.BankingSystemFinal.Service.ICustomerServiceProvider import ICustomerServiceProvider
from PyCharm.Assignments.BankingSystemFinal.Classes.CurrentAccount import CurrentAccount


2 usages
class CustomerServiceProviderImpl(ICustomerServiceProvider):
    def __init__(self, dbUtil):
        self.dbUtil = dbUtil

    5 usages
    def get_account_balance(self, accountNum):
        query = "Select balance from accounts where accountID = %s"
        value = (accountNum,)
        result = self.dbUtil.fetchOne(query, value)
        return result[0]

    def deposit(self, accountNum, amount):
        if amount <= 0:
            print("Invalid Amount")
        else:
            if self.get_AccountDetails(accountNum) is not None:
                balance = self.get_account_balance(accountNum)
                newBal = float(balance) + amount
                query = "Update accounts set balance=%s where accountID = %s"
                values = (newBal, accountNum)
                self.dbUtil.executeQuery(query, values)
                print(f"New Balance is: {newBal}")
            else:
                raise Exception("InvalidAccountIDException")
```

```python
    2 usages
    def withdraw(self, accountNum, amount):
        currentBal = self.get_account_balance(accountNum)
        accountType = self.getAccountType(accountNum)

        if self.get_AccountDetails(accountNum) is not None:
            if amount <= 0:
                return "Invalid Amount"
            else:
                if accountType == 'savings':
                    if amount > currentBal:
                        raise Exception("InsufficientFundException")
                    else:
                        if float(currentBal) - amount < 500.00:
                            raise Exception("MinimumBalanceLimitException")
                        else:
                            query = "Update accounts set balance=balance-%s where accountID = %s"
                            values = (amount, accountNum)
                            self.dbUtil.executeQuery(query, values)
                            result = self.get_account_balance(accountNum)
                            return result
                elif accountType == 'current':
                    if amount > currentBal and amount - currentBal > CurrentAccount.LIMIT:
                        raise Exception("OverdraftLimitExceededException")
                    else:
                        query = "Update accounts set balance=balance-%s where accountID = %s"
                        values = (amount, accountNum)
                        self.dbUtil.executeQuery(query, values)
                        result = self.get_account_balance(accountNum)
                        return result
                else:
                    raise Exception("ZeroBalanceAccountException")
        else:
            raise Exception("InvalidAccountIDException")
```

```
    def transfer(self, fromAccountNum, toAccountNum, amount):
        if self.get_AccountDetails(fromAccountNum) and self.get_AccountDetails(toAccountNum):
            try:
                self.withdraw(fromAccountNum, amount)
                self.deposit(toAccountNum, amount)
            except Exception as e:
                return f"Transfer Failed!!! Error: {e}"

    6 usages
    def get_AccountDetails(self, accountNum):
        query = "Select * from accounts join customers on accounts.customerID = customers.customerID where accountID=%s"
        value = (accountNum,)
        result = self.dbUtil.fetchall(query, value)
        return result

    1 usage
    def getTransactions(self):
        accountNum = input("Enter your accountID: ")
        fromDate = input("From: ")
        toDate = input("To: ")
        if self.get_AccountDetails(accountNum) is not None:
            query = "Select * from transactions where transaction_date between %s and %s and accountID = %s"
            values = (fromDate, toDate, accountNum)
            result = self.dbUtil.fetchall(query, values)
            return result
        else:
            raise Exception("InvalidAccountIDException")

    2 usages
    def getAccountType(self, accountNum):
        value = (accountNum,)
        result = self.dbUtil.fetchOne("Select account_type from accounts where accountID = %s", value)
        return result[0]
```

2. BankServiceProviderImpl Class:

```
from PyCharm.Assignments.BankingSystemFinal.Service.IBankServiceProvider import IBankServiceProvider


2 usages
class BankServiceProviderImpl(IBankServiceProvider):
    def __init__(self, dbUtil):
        self.dbUtil = dbUtil

    1 usage
    def create_account(self):
        customerID = input("Please enter your customerID: ")
        if self.existingCustomer(customerID) is None:
            raise Exception("CustomerNotRegisteredException")
        else:
            print("Please fill up the account details: ")
            account = {
                'accountID': self.generateUniqueAccountID(),
                'customerID': customerID,
            }
            print("What type of account do you want to create?")
            print("1. Savings Account")
            print("2. Current Account")
            print("3. Zero Balance Account")
            ch = int(input("Enter your choice: "))

            match ch:
                case 1:
                    account['account_type'] = 'savings'
                    account['balance'] = float(input("Enter the amount: "))
                    return self.AddAccountIntoDatabase(account)

                case 2:
                    account['account_type'] = 'current'
                    account['balance'] = float(input("Enter the amount: "))
                    return self.AddAccountIntoDatabase(account)
```

```
1 usage
def listAccounts(self):
    query = "SELECT accountID, customerID, account_type, balance FROM accounts"
    result = self.dbUtil.fetchall(query, values=None)
    print(result)

def get_AccountDetails(self, accountNum):
    query = "Select * from accounts where accountID=(%s)"
    value = (accountNum,)
    return self.dbUtil.fetchall(query, value)

def calculateInterest(self, inRate=4.5):
    query = "select balance * %s as inRate from accounts"
    return self.dbUtil.fetchall(query, inRate)

1 usage
def get_no_of_accounts(self):
    query = "Select count(*) from accounts"
    result = self.dbUtil.fetchOne(query)
    return result[0]

1 usage
def generateUniqueAccountID(self):
    concat = ('BO', str(self.get_no_of_accounts()+ 1))
    return "".join(concat)

1 usage
def existingCustomer(self, customerID):
    query = "Select * from Customers where customerID = %s"
    value = (customerID,)
    return self.dbUtil.fetchOne(query, value)
```

```
3 usages
def AddAccountIntoDatabase(self, accounts):
    query = "insert into accounts values(%s, %s, %s, %s)"
    values = (accounts['accountID'], accounts['customerID'], accounts['account_type'], accounts['balance'])
    return self.dbUtil.executeQuery(query, values)
```

TASK 5: DATABASE CONNECTION CLASS

```python
from mysql.connector import connect


2 usages
class DBUtil:
    def __init__(self, host, user, password, port, database):
        self.connection = connect(
            host=host,
            user=user,
            password=password,
            port=port,
            database=database
        )
        self.cursor = self.connection.cursor()

    23 usages (23 dynamic)
    def executeQuery(self, query, values=None):
        try:
            self.cursor.execute(query, values)
            self.connection.commit()
        except Exception as e:
            print(f"Query Execution Error! {e}")
            self.connection.rollback()

    def fetchall(self, query, values=None):
        try:
            self.cursor.execute(query, values)
            return self.cursor.fetchall()
        except Exception as e:
            print(f"FetchAll Error: {e}")
            self.connection.rollback()

    def fetchOne(self, query, values=None):
        try:
            self.cursor.execute(query, values)
            return self.cursor.fetchone()
        except:
            print(f"FetchOne Error!")
            self.connection.rollback()

    1 usage
    def closeConnection(self):
        self.cursor.close()
        self.connection.close()
```

TASK 6: MAIN APP

```python
from PyCharm.Assignments.BankingSystemFinal.Connection.DBUtil import DBUtil
from PyCharm.Assignments.BankingSystemFinal.Bean.BankServiceProviderImpl import BankServiceProviderImpl
from PyCharm.Assignments.BankingSystemFinal.Bean.CustomerServiceProviderImpl import CustomerServiceProviderImpl



1 usage
class BankApp:
    def main(self):
        try:
            dbutil = DBUtil(host='localhost', user='root', password='SQL@bunny11', port='3306', database='hmbank')
            bsp = BankServiceProviderImpl(dbutil)
            csp = CustomerServiceProviderImpl(dbutil)
        except Exception as e:
            raise Exception(f"Error Connecting Server: {e}")

        print("Welcome to HMBank App!")
        while True:
            print("What do you want to do?")
            print("1. Create an Account")
            print("2. Deposit Money")
            print("3. Withdraw Money")
            print("4. Get Balance")
            print("5. Transfer Money")
            print("6. Get your Account Details")
            print("7. List all Accounts")
            print("8. Get Transaction Details")
            print("9. Get ACCount Type")
            print("10. Exit")
            ch = int(input("Enter your choice: "))
```

```
        ch = int(input("Enter your choice: "))

        if ch == 1:
            bsp.create_account()
        elif ch == 2:
            accountNum = input("Enter the account number in which you want to deposit:")
            amount = float(input("Enter the amount: "))
            csp.deposit(accountNum, amount)
        elif ch == 3:
            accountNum = input("Enter the account number from which you want to withdraw:")
            amount = float(input("Enter the amount: "))
            csp.withdraw(accountNum, amount)
        elif ch == 4:
            accountNum = input("Enter the account number to get balance:")
            print(csp.get_account_balance(accountNum))
        elif ch == 5:
            fromAcc = input("AccountID of the sender: ")
            toAcc = input("AccountID of the receiver: ")
            amount = float(input("Enter the amount: "))
            csp.transfer(fromAcc, toAcc, amount)
        elif ch == 6:
            accountNum = input("Enter the account number to get details:")
            print(csp.get_AccountDetails(accountNum))
        elif ch == 7:
            bsp.listAccounts()
        elif ch == 8:
            print(csp.getTransactions())
        elif ch == 9:
            accountNum = input("Enter your accountID: ")
            print(csp.getAccountType(accountNum))
        elif ch == 10:
            dbutil.closeConnection()
            break  # Exit the loop
```

WORKING OF THE APP:

```
Welcome to HMBank App!
What do you want to do?
1. Create an Account
2. Deposit Money
3. Withdraw Money
4. Get Balance
5. Transfer Money
6. Get your Account Details
7. List all Accounts
8. Get Transaction Details
9. Get ACCount Type
10. Exit
Enter your choice: 7
[('A001', 'C001', 'savings', Decimal('25000.00')), ('A002', 'C002', 'current', Decimal('50000.00')), ('A004', 'C004', 'savings', Decimal('135800.00')), ('A005'
What do you want to do?
1. Create an Account
2. Deposit Money
3. Withdraw Money
4. Get Balance
5. Transfer Money
6. Get your Account Details
7. List all Accounts
8. Get Transaction Details
9. Get ACCount Type
10. Exit
Enter your choice: 10

Process finished with exit code 0
```

TASK 7: CREATING AND MAINTAINING DIRECTORIES