Assignment 3 – Banking System

Control Structure

Task 1:

```
credit_score = int(input("Enter your credit score: "))
annual_income = float(input("Enter your annual income: $"))

credit_score_threshold = 700
income_threshold = 50000

if credit_score > credit_score_threshold and annual_income >=
income_threshold:
    print("Congratulations! You are eligible for a loan.")
else:
    print("Sorry, you are not eligible for a loan.")
```

```
Enter your credit score: 500
Enter your annual income: 1000000
Sorry, you are not eligible for a loan.
```

Task 2:

```
def display_options():
    print("ATM Options:")
    print("1. Check Balance")
    print("2. Withdraw")
    print("3. Deposit")
    print("4. Exit")
def check_balance(balance):
    print("Your current balance is: Rs. {}".format(balance))
def withdraw_money(balance):
    amount = int(input("Enter the amount to withdraw: Rs."))
    if amount % 100 == 0 or amount % 500 == 0:
        if amount <= balance:
            balance -= amount
            print("Withdrawal successful. Remaining balance:
Rs{}".format(balance))
        else:
            print("Insufficient funds.")
    else:
        print("Withdrawal amount must be in multiples of 100 or 500.")
def deposit_money(balance):
    amount = int(input("Enter the amount to deposit: Rs."))
    if amount > 0:
        balance += amount
        print("Deposit successful. New balance: Rs.{}".format(balance))
    else:
        print("Invalid deposit amount.")

def main():
```

```
    balance = float(input("Enter your current balance: Rs. "))

    while True:
        display_options()
        choice = input("Enter your choice (1-4): ")

        if choice == '1':
            check_balance(balance)
        elif choice == '2':
            withdraw_money(balance)
        elif choice == '3':
            deposit_money(balance)
        elif choice == '4':
            print("Thank you for using our ATM. Have a nice day!")
            break
        else:
            print("Invalid choice. Please enter a number between 1 and 4.")


if __name__ == "__main__":
    main()
```

```
Enter your current balance: Rs. 100000
ATM Options:
1. Check Balance
2. Withdraw
3. Deposit
4. Exit
Enter your choice (1-4): 1
Your current balance is: Rs. 100000.0
ATM Options:
1. Check Balance
2. Withdraw
3. Deposit
4. Exit
Enter your choice (1-4): 2
Enter the amount to withdraw: Rs.5000
Withdrawal successful. Remaining balance: Rs95000.0
```

Task 3:

```
def calculate_future_balance(initial_balance, annual_interest_rate, years):
    future_balance = initial_balance * (1 + annual_interest_rate/100) **
years
    return future_balance
def main():
    num_customers = int(input("Enter the number of customers: "))

    for i in range(1, num_customers + 1):
        print("\nCustomer", i)
        initial_balance = float(input("Enter the initial balance for
customer {}: Rs".format(i)))
        annual_interest_rate = float(input("Enter the annual interest rate
for customer {} (%): ".format(i)))
        years = int(input("Enter the number of years for customer {}:
".format(i)))
```

```
        future_balance = calculate_future_balance(initial_balance,
annual_interest_rate, years)
        print("Future balance for customer {}: Rs.{:.2f}".format(i,
future_balance))

if __name__ == "__main__":
    main()
```

```
Enter the number of customers: 3

Customer 1
Enter the initial balance for customer 1: Rs1000
Enter the annual interest rate for customer 1 (%): 10
Enter the number of years for customer 1: 2
Future balance for customer 1: Rs.1210.00

Customer 2
Enter the initial balance for customer 2: Rs2000
Enter the annual interest rate for customer 2 (%): 20
Enter the number of years for customer 2: 3
Future balance for customer 2: Rs.3456.00

Customer 3
Enter the initial balance for customer 3: Rs3000
Enter the annual interest rate for customer 3 (%): 5
Enter the number of years for customer 3: 1
Future balance for customer 3: Rs.3150.00
```

Task 4:

```
accounts = {
    '123456': 1000.00,
    '234567': 2000.00,
    '345678': 1500.00,
    '456789': 3000.00
}
def is_valid_account(account_number):
    return account_number in accounts

def main():
    while True:
        account_number = input("Enter your account number (or 'q' to quit):
")

        if account_number.lower() == 'q':
            print("Exiting the program. Goodbye!")
            break

        if is_valid_account(account_number):
            print("Your account balance is:
Rs{:.2f}".format(accounts[account_number]))
        else:
            print("Invalid account number. Please try again.")
```

```
if __name__ == "__main__":
    main()
```

```
Enter your account number (or 'q' to quit): 1234
Invalid account number. Please try again.
Enter your account number (or 'q' to quit): 123456
Your account balance is: Rs1000.00
Enter your account number (or 'q' to quit): q
Exiting the program. Goodbye!
```

Task 5:

```
def validate_password(password):
    if len(password) < 8:
        return False, "Password must be at least 8 characters long."

    if not any(char.isupper() for char in password):
        return False, "Password must contain at least one uppercase
letter."

    if not any(char.isdigit() for char in password):
        return False, "Password must contain at least one digit."

    return True

def main():
    password = input("Create a password for your bank account: ")
    is_valid = validate_password(password)
    if is_valid:
        print("Valid password")

if __name__ == "__main__":
    main()
```

```
Create a password for your bank account: passWord1
Valid password
```

Task 6:

```
def display_transaction_history(transaction_history):
    print("\nTransaction History:")
    print("Type\t|\tAmount")
    print("-------------------------------")
    for transaction in transaction_history:
        print(transaction[0] + "\t|\tRs." + str(transaction[1]))

def main():
    transaction_history = []

    while True:
        print("\nOptions:")
        print("1. Add Deposit")
```

```
        print("2. Add Withdrawal")
        print("3. Exit")

        choice = input("Enter your choice: ")

        if choice == '1':
            amount = float(input("Enter deposit amount: Rs."))
            transaction_history.append(("Deposit", amount))
        elif choice == '2':
            amount = float(input("Enter withdrawal amount: Rs."))
            transaction_history.append(("Withdrawal", amount))
        elif choice == '3':
            print("Exiting...")
            break
        else:
            print("Invalid choice. Please try again.")

    display_transaction_history(transaction_history)

if __name__ == "__main__":
    main()
```

```
Options:
1. Add Deposit
2. Add Withdrawal
3. Exit
Enter your choice: 1
Enter deposit amount: Rs.1000

Options:
1. Add Deposit
2. Add Withdrawal
3. Exit
Enter your choice: 3
Exiting...

Transaction History:
Type      |    Amount
-----------------------------
Deposit   |    Rs.1000.0
```

Implementation pf oops and database connectivity:

CUSTOMER CLASS:

```
class Customer:
    __customer_id_counter = 1

    def __init__(self, first_name=None, last_name=None, email=None,
phone_number=None, address=None):
        self.__customer_id = Customer.__customer_id_counter
        Customer.__customer_id_counter += 1
        self.__first_name = first_name
        self.__last_name = last_name
        self.__email = email
        self.__phone_number = phone_number
        self.__address = address

    # Getter methods
    def get_customer_id(self):
        return self.__customer_id

    def get_first_name(self):
        return self.__first_name

    def get_last_name(self):
        return self.__last_name

    def get_email(self):
        return self.__email

    def get_phone_number(self):
        return self.__phone_number

    def get_address(self):
        return self.__address

    # Setter methods
    def set_customer_id(self, customer_id):
        self.__customer_id = customer_id

    def set_first_name(self, first_name):
        self.__first_name = first_name

    def set_last_name(self, last_name):
        self.__last_name = last_name

    def set_email(self, email):
        self.__email = email

    def set_phone_number(self, phone_number):
        self.__phone_number = phone_number

    def set_address(self, address):
        self.__address = address

    # Method to print all information of the customer
    def print_info(self):
        print("Customer ID:", self.__customer_id)
        print("First Name:", self.__first_name)
        print("Last Name:", self.__last_name)
```

```
        print("Email Address:", self.__email)
        print("Phone Number:", self.__phone_number)
        print("Address:", self.__address)
```

ACCOUNTS CLASS:

```
from models.Customers import Customer


class Account:
    # Static variable to generate unique account numbers
    __last_account_number = 1000

    def __init__(self, account_type, account_balance, customer):
        # Generating unique account number
        Account.__last_account_number += 1
        self.__account_number = Account.__last_account_number

        self.__account_type = account_type
        self.__account_balance = account_balance
        self.__customer = customer

    # Getter methods
    def get_account_number(self):
        return self.__account_number

    def get_account_type(self):
        return self.__account_type

    def get_account_balance(self):
        return self.__account_balance

    def get_customer(self):
        return self.__customer

    # Setter methods
    def set_account_balance(self, new_balance):
        self.__account_balance = new_balance

    # Method to deposit money into the account
    def deposit(self, amount):
        self.__account_balance += amount

    # Method to withdraw money from the account
    def withdraw(self, amount):
        if self.__account_balance >= amount:
            self.__account_balance -= amount
            print("Withdrawal successful. Remaining balance:",
self.__account_balance)
        else:
            print("Insufficient funds")

    # Method to calculate interest
    def calculate_interest(self):
        # Assuming fixed interest rate of 4.5%
        interest = self.__account_balance * 0.045
        self.deposit(interest)  # Adding interest to the account balance

    # Method to print all information of the account
    def print_info(self):
```

```
        print("Account Number:", self.__account_number)
        print("Account Type:", self.__account_type)
        print("Account Balance:", self.__account_balance)
        print("Customer:", self.__customer.get_first_name(),
self.__customer.get_last_name())
```

TRANSACTION CLASS

```
class Transaction:
    def __init__(self, account, description, transaction_type,
transaction_amount, date_time):
        self.account = account
        self.description = description
        self.transaction_type = transaction_type
        self.transaction_amount = transaction_amount
        self.date_time = date_time

    # Method to print all information of the transaction
    def get_transaction_details(self):
        print("Account:", self.account.get_account_number())
        print("Description:", self.description)
        print("Transaction Type:", self.transaction_type)
        print("Transaction Amount:", self.transaction_amount)
        print("Date and Time:", self.date_time)
```

Inheritance concept classes:

CURRENT ACCOUNT CLASS

```
from bean.Accounts import Account


class CurrentAccount(Account):
    def __init__(self, account_number, account_type, account_balance,
customer, overdraft_limit):
        super().__init__(account_type, account_balance, customer)
        self.overdraft_limit = overdraft_limit

    def get_overdraft_limit(self):
        return self.overdraft_limit
```

SAVINGS ACCOUNT CLASS:

```
from bean.Accounts import Account


class SavingsAccount(Account):
    def __init__(self, account_number, account_type, account_balance,
customer, interest_rate):
        if account_balance < 500:
            raise ValueError("Minimum balance for a savings account must be
500")

        super().__init__(account_type, account_balance, customer)
        self.interest_rate = interest_rate
```

```
    def get_interest_rate(self):
        return self.interest_rate
```

ZERO ACCOUNT CLASS:

```
from bean.Accounts import Account


class ZeroBalanceAccount(Account):
    def __init__(self, account_type, customer):
        super().__init__(account_type, 0, customer)
```

Abstract class for various Services:

IBankRepository Abstract class:

```
from abc import ABC, abstractmethod
from datetime import datetime


class IBankRepository(ABC):
    @abstractmethod
    def createAccount(self):
        pass

    @abstractmethod
    def listAccounts(self):
        pass

    @abstractmethod
    def calculateInterest(self):
        pass

    @abstractmethod
    def getAccountBalance(self, account_number):
        pass

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

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

    @abstractmethod
    def transfer(self, from_account_number, to_account_number, amount):
        pass

    @abstractmethod
    def getAccountDetails(self, account_number):
        pass

    @abstractmethod
    def getTransactions(self, account_number, FromDate, ToDate):
        pass
```

IBankServiceProvider Abstract class:

```
class IBankServiceProvider:
    def create_account(self):
        pass

    def listAccounts(self):
        pass

    def getAccountDetails(self, account_number):
        pass

    def calculateInterest(self):
        pass
```

ICustomerServiceProvider Abstract class:

```
from abc import ABC, abstractmethod
from datetime import date


class ICustomerServiceProvider(ABC):
    @abstractmethod
    def get_account_balance(self, account_number):
        pass

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

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

    @abstractmethod
    def transfer(self, from_account_number, to_account_number, amount):
        pass

    @abstractmethod
    def get_account_details(self, account_number):
        pass

    @abstractmethod
    def get_transactions(self, account_number, from_date, to_date):
        pass
```

Database connection class:

DBUtil class:

```
import mysql.connector

class DBUtil:
    @staticmethod
    def getDBConn():
        try:
```

```
            connection = mysql.connector.connect(
                host="localhost",
                user="root",
                password="root",
                database="HMBank",
                port="3306"
            )
            print("Connected to MySQL database")
            return connection
        except mysql.connector.Error as err:
            print("Error:", err)
```

IMPLEMENTATION CLASSES:

BankRepositoryImpl class:

```
import pymysql

from Connection.DBUtil import DBUtil
from Services.IBankRepository import IBankRepository
from bean.Accounts import Account
from models.Customers import Customer
from models.Transactions import Transaction


class BankRepositoryImpl(IBankRepository):
    def __init__(self):
        self.connection = DBUtil.getDBConn()

    def createAccount(self):
        first_name = input("Enter customer's first name: ")
        last_name = input("Enter customer's last name: ")
        email = input("Enter customer's email address: ")
        phone_number = input("Enter customer's phone number: ")
        address = input("Enter customer's address: ")

        customer = Customer(first_name, last_name, email, phone_number,
address)

        accType = input("Enter account type (savings/current/zero_balance):
")
        balance = float(input("Enter initial balance: "))

        # Create a new account object
        new_account = Account(accType, balance, customer)

        # Add the new account to the account list
        db_connection = DBUtil.getDBConn()
        cursor = db_connection.cursor()
        try:
            cursor.execute(
                "INSERT INTO Accounts (account_id, customer_id,
account_type, balance) VALUES (%s, %s, %s, %s)",
                (new_account.get_account_number(),
new_account.get_customer().get_customer_id(),
                 new_account.get_account_type(),
new_account.get_account_balance()))

            db_connection.commit()
```

```
            print("New account added to the database.")

        except pymysql.Error as e:
            print(f"Error occurred: {e}")
            db_connection.rollback()

        finally:
            cursor.close()
            db_connection.close()
        return new_account.get_account_number()

    def listAccounts(self):
        accounts = []
        with self.connection.cursor() as cursor:
            sql = "SELECT * FROM Accounts"
            cursor.execute(sql)
            results = cursor.fetchall()
            for row in results:
                print(row)

    def calculateInterest(self):
        cursor = self.connection.cursor()
        try:


            cursor.execute("SELECT account_id, balance, interest_rate FROM
Accounts")
            accounts = cursor.fetchall()

            for account in accounts:
                account_id, balance, interest_rate = account
                interest_amount = balance * (interest_rate / 100)
                new_balance = balance + interest_amount

                cursor.execute("UPDATE Accounts SET balance = %s WHERE
account_id = %s", (new_balance, account_id))
                self.connection.commit()

            print("Interest calculated and updated successfully.")

        except Exception as e:
            print("Error:", e)

        finally:
            cursor.close()

    def getAccountBalance(self, account_number):
        with self.connection.cursor() as cursor:
            sql = "SELECT balance FROM Accounts WHERE account_id = %s"
            cursor.execute(sql, (account_number,))
            result = cursor.fetchone()
            if result:
                print(result)
            return None

    def deposit(self, account_number, amount):
        current_balance = self.getAccountBalance(account_number)
        if current_balance is not None:
            new_balance = current_balance + amount
            with self.connection.cursor() as cursor:
```

```
                sql = "UPDATE Accounts SET balance = %s WHERE account_id =
%s"
                cursor.execute(sql, (new_balance, account_number))
            self.connection.commit()
            return new_balance
        else:
            return None

    def withdraw(self, account_number, amount):
        current_balance = self.getAccountBalance(account_number)
        if current_balance is not None:
            new_balance = current_balance - amount
            if new_balance >= 0:
                with self.connection.cursor() as cursor:
                    sql = "UPDATE Accounts SET balance = %s WHERE
account_id = %s"
                    cursor.execute(sql, (new_balance, account_number))
                self.connection.commit()
                return new_balance
            else:
                return None
        else:
            return None

    def transfer(self, from_account_number, to_account_number, amount):
        cursor = self.connection.cursor()

        try:

            cursor.execute("SELECT balance FROM Accounts WHERE account_id =
%s", (from_account_number,))
            sender_balance = cursor.fetchone()
            cursor.execute("SELECT balance FROM Accounts WHERE account_id =
%s", (to_account_number,))
            receiver_balance = cursor.fetchone()

            if sender_balance is None or receiver_balance is None:
                print("Invalid account number.")
                return

            sender_balance = sender_balance[0]
            receiver_balance = receiver_balance[0]

            if sender_balance < amount:
                print("Insufficient funds.")
                return

            sender_balance -= amount
            receiver_balance += amount

            cursor.execute("UPDATE Accounts SET balance = %s WHERE
account_id = %s",
                           (sender_balance, from_account_number))


            cursor.execute("UPDATE Accounts SET balance = %s WHERE
account_id = %s",
                           (receiver_balance, to_account_number))


            cursor.execute(
```

```
                "INSERT INTO Transactions (account_id, transaction_type,
amount) VALUES (%s, 'transfer', %s)",
                (from_account_number, -amount))
            cursor.execute(
                "INSERT INTO Transactions (account_id, transaction_type,
amount) VALUES (%s, 'transfer', %s)",
                (to_account_number, amount))

            self.connection.commit()
            print("Transfer successful.")

        except Exception as e:
            self.connection.rollback()
            print("Error:", e)

        finally:
            cursor.close()

    def getAccountDetails(self, account_number):
        with self.connection.cursor() as cursor:
            sql = "SELECT * FROM Accounts WHERE account_id = %s"
            cursor.execute(sql, (account_number,))
            result = cursor.fetchone()
            print(result)

    def getTransactions(self, account_number, from_date, to_date):
        transactions = []
        with self.connection.cursor() as cursor:
            sql = "SELECT * FROM Transactions WHERE account_id = %s AND
transaction_date BETWEEN %s AND %s"
            cursor.execute(sql, (account_number, from_date, to_date))
            results = cursor.fetchall()
            for row in results:
                transaction = Transaction(row['transaction_id'],
row['account_id'], row['transaction_type'],
                                          row['amount'],
row['transaction_date'])
                transactions.append(transaction)
        if results:
            for i in results:
                print(i)
        else:
            print("No transactions found.")

    def __del__(self):
        self.connection.close()
```

BankServiceProviderImpl class:

```
import pymysql

from Connection.DBUtil import DBUtil
from Services.IBankServiceProvider import IBankServiceProvider
from bean.Accounts import Account
from bean.CustomerServiceProviderImpl import CustomerServiceProviderImpl
from models.Customers import Customer
from models.SavingsAccount import SavingsAccount


class BankServiceProviderImpl(CustomerServiceProviderImpl,
```

```
IBankServiceProvider):
    def __init__(self, branchName, branchAddress):
        super().__init__()
        self.branchName = branchName
        self.branchAddress = branchAddress
        self.accountList = []
        self.transactionList = []

    def create_account(self):
        first_name = input("Enter customer's first name: ")
        last_name = input("Enter customer's last name: ")
        email = input("Enter customer's email address: ")
        phone_number = input("Enter customer's phone number: ")
        address = input("Enter customer's address: ")

        customer = Customer(first_name, last_name, email, phone_number,
address)

        accType = input("Enter account type (savings/current/zero_balance):
")
        balance = float(input("Enter initial balance: "))

        new_account = Account(accType, balance, customer)

        self.accountList.append(new_account)
        db_connection = DBUtil.getDBConn()
        cursor = db_connection.cursor()
        try:
            cursor.execute(
                "INSERT INTO Accounts (account_id, customer_id,
account_type, balance) VALUES (%s, %s, %s, %s)",
                (new_account.get_account_number(),
new_account.get_customer().get_customer_id(),
new_account.get_account_type(), new_account.get_account_balance()))

            db_connection.commit()

            print("New account added to the database.")

        except pymysql.Error as e:
            print(f"Error occurred: {e}")
            db_connection.rollback()

        finally:
            cursor.close()
            db_connection.close()
        return new_account.get_account_number()

    def listAccounts(self):
        print("List of Accounts:")
        for account in self.accountList:
            print(f"Account Number: {account.get_account_number()}, Account
Type: {account.get_account_type()}, Balance:
{account.get_account_balance()}")

    def getAccountDetails(self, account_number):
        for account in self.accountList:
            if account.accNo == account_number:
                print("Account Details:")
                print(f"Account Number: {account.get_account_number()}")
                print(f"Account Type: {account.get_account_type()}")
```

```
                print(f"Balance: {account.get_account_balance()}")
                customer = account.get_customer()
                print(f"Customer ID: {customer.get_customer_id()}")
                print(f"Customer Name: {customer.get_first_name()}
{customer.get_last_name()}")
                print(f"Customer Email: {customer.get_email()}")
                print(f"Customer Phone Number:
{customer.get_phone_number()}")
                print(f"Customer Address: {customer.get_address()}")
                return
        print("Account not found.")

    def calculateInterest(self):
        print("Calculating interest for savings accounts:")
        for account in self.accountList:
            if isinstance(account, SavingsAccount):
                interest = account.calculate_interest()
                new_balance = account.get_account_balance() + interest
                account.set_account_balance(new_balance)
                print(f"Interest of {interest} added to account
{account.get_account_number()}")
        print("Interest calculation completed.")
```

CustomerServiceProviderImpl class:

```
import mysql.connector

from Connection.DBUtil import DBUtil
from Services.ICustomerServiceProvider import ICustomerServiceProvider


class CustomerServiceProviderImpl(ICustomerServiceProvider):
    def __init__(self):
        self.db_connection = DBUtil.getDBConn()

    def get_account_balance(self, account_number):
        try:
            cursor = self.db_connection.cursor()
            cursor.execute("SELECT balance FROM Accounts WHERE account_id =
%s", (account_number,))
            result = cursor.fetchone()
            if result:
                return result[0]
            return None
        except mysql.connector.Error as err:
            print("Error:", err)

    def deposit(self, account_number, amount):
        try:
            cursor = self.db_connection.cursor()
            cursor.execute("UPDATE Accounts SET balance = balance + %s
WHERE account_id = %s", (amount, account_number))
            self.db_connection.commit()
            return True
        except mysql.connector.Error as err:
            print("Error:", err)
```

```
            self.db_connection.rollback()
            return False

    def withdraw(self, account_number, amount):
        try:
            current_balance = self.get_account_balance(account_number)
            if current_balance is not None:
                cursor = self.db_connection.cursor()
                cursor.execute("SELECT account_type FROM Accounts WHERE
account_id = %s", (account_number,))
                account_type = cursor.fetchone()[0]

                if account_type == 'Savings':
                    if current_balance >= amount:
                        new_balance = current_balance - amount
                        cursor.execute("UPDATE Accounts SET balance = %s
WHERE account_id = %s",
                                       (new_balance, account_number))
                        self.db_connection.commit()
                        return True
                    else:
                        print("Insufficient balance")
                        return False
                elif account_type == 'Current':
                    cursor.execute("SELECT overdraftLimit FROM Accounts
WHERE account_id = %s", (account_number,))
                    overdraft_limit = cursor.fetchone()[0]
                    if (current_balance + overdraft_limit) >= amount:
                        new_balance = current_balance - amount
                        cursor.execute("UPDATE Accounts SET balance = %s
WHERE account_id = %s",
                                       (new_balance, account_number))
                        self.db_connection.commit()
                        return True
                    else:
                        print("Withdrawal amount exceeds overdraft limit")
                        return False
            return False
        except mysql.connector.Error as err:
            print("Error:", err)
            self.db_connection.rollback()
            return False

    def transfer(self, from_account_number, to_account_number, amount):
        try:
            if self.withdraw(from_account_number, amount):
                self.deposit(to_account_number, amount)
                return True
            return False
        except mysql.connector.Error as err:
            print("Error:", err)
            self.db_connection.rollback()
            return False

    def get_account_details(self, account_number):
        try:
            cursor = self.db_connection.cursor()
            cursor.execute("SELECT * FROM Accounts WHERE account_id = %s",
(account_number,))
            account_details = cursor.fetchone()
            if account_details:
```

```
                return account_details
            return None
        except mysql.connector.Error as err:
            print("Error:", err)

    def get_transactions(self, account_number, FromDate, ToDate):
        try:
            cursor = self.db_connection.cursor()
            cursor.execute("SELECT * FROM Transactions WHERE account_id =
%s AND transaction_date BETWEEN %s AND %s",
                           (account_number, FromDate, ToDate))
            transactions = cursor.fetchall()
            return transactions
        except mysql.connector.Error as err:
            print("Error:", err)
```

BankApp class which implements the system:

```
from bean.BankRepositoryImpl import BankRepositoryImpl



class BankApp:
    def __init__(self):
        self.bank_service_provider = BankRepositoryImpl()

    def display_menu(self):
        print("Welcome to the Banking System")
        print("1. Create Account")
        print("2. Deposit")
        print("3. Withdraw")
        print("4. Get Balance")
        print("5. Transfer")
        print("6. Get Account Details")
        print("7. List Accounts")
        print("8. Get Transactions")
        print("9. Exit")

    def main(self):
        while True:
            self.display_menu()
            choice = input("Enter your choice: ")

            if choice == "1":
                self.create_account_menu()
            elif choice == "2":
                self.deposit()
            elif choice == "3":
                self.withdraw()
            elif choice == "4":
                self.get_balance()
            elif choice == "5":
                self.transfer()
            elif choice == "6":
                self.get_account_details()
            elif choice == "7":
                self.list_accounts()
```

```
            elif choice == "8":
                self.get_transactions()
            elif choice == "9":
                print("Exiting the Banking System. Goodbye!")
                break
            else:
                print("Invalid choice. Please try again.")

    def create_account_menu(self):

        self.bank_service_provider.createAccount()

    def deposit(self):
        account_number = int(input("Enter your account number: "))
        amount = int(input("Enter the amount: "))
        self.bank_service_provider.deposit(account_number, amount)

    def withdraw(self):
        account_number = int(input("Enter your account number: "))
        amount = int(input("Enter the amount: "))
        self.bank_service_provider.withdraw(account_number, amount)

    def get_balance(self):
        account_number = int(input("Enter your account number: "))
        self.bank_service_provider.getAccountBalance(account_number)

    def transfer(self):
        from_account = int(input("Enter the from account number: "))
        to_account = int(input("Enter the from to number: "))
        amount = int(input("Enter the amount: "))
        self.bank_service_provider.transfer(from_account, to_account,
amount)

    def get_account_details(self):
        account_number = int(input("Enter the account number: "))
        self.bank_service_provider.getAccountDetails(account_number)

    def list_accounts(self):
        self.bank_service_provider.listAccounts()

    def get_transactions(self):
        account_number = int(input("Enter the account number: "))
        from_date = input("Enter the from date: ")
        to_date = input("Enter the to date: ")

self.bank_service_provider.getTransactions(account_number,from_date,to_date
)

if __name__ == "__main__":
    # Create BankApp instance and start the main loop
    bank_app = BankApp()
    bank_app.main()
```

Outputs on the working of the system:

```
Connected to MySQL database
Welcome to the Banking System
1. Create Account
2. Deposit
3. Withdraw
4. Get Balance
5. Transfer
6. Get Account Details
7. List Accounts
8. Get Transactions
9. Exit
Enter your choice: 7
(101, 1, 'savings', Decimal('55000.00'))
(102, 2, 'current', Decimal('75000.50'))
(103, 3, 'zero_balance', Decimal('0.00'))
(104, 4, 'savings', Decimal('120000.75'))
(105, 5, 'current', Decimal('25000.25'))
(106, 6, 'savings', Decimal('100000.50'))
(107, 7, 'zero_balance', Decimal('0.00'))
(108, 8, 'current', Decimal('30000.00'))
(109, 9, 'savings', Decimal('80000.25'))
(110, 10, 'current', Decimal('40000.50'))
```

```
Connected to MySQL database
Welcome to the Banking System
1. Create Account
2. Deposit
3. Withdraw
4. Get Balance
5. Transfer
6. Get Account Details
7. List Accounts
8. Get Transactions
9. Exit
Enter your choice: 8
Enter the account number: 101
Enter the from date: 01-01-2024
Enter the to date: 30-01-2024
No transactions found.
```

```
Enter your choice: 6
Enter the account number: 101
(101, 1, 'savings', Decimal('55000.00'))
```

```
Welcome to the Banking System
1. Create Account
2. Deposit
3. Withdraw
4. Get Balance
5. Transfer
6. Get Account Details
7. List Accounts
8. Get Transactions
9. Exit
Enter your choice: 4
Enter your account number: 101
(Decimal('50000.00'),)
```

```
Enter your choice: 5
Enter the from account number: 101
Enter the from to number: 102
Enter the amount: 1000
```

Updated balance after transfer

```
Enter your choice: 4
Enter your account number: 101
(Decimal('49000.00'),)
```