Banking System

Control Structure

Task 1: Conditional Statements

In a bank, you have been given the task is to create a program that checks if a customer is eligible for a loan based on their credit score and income. The eligibility criteria are as follows:

- Credit Score must be above 700.
- Annual Income must be at least $50,000.

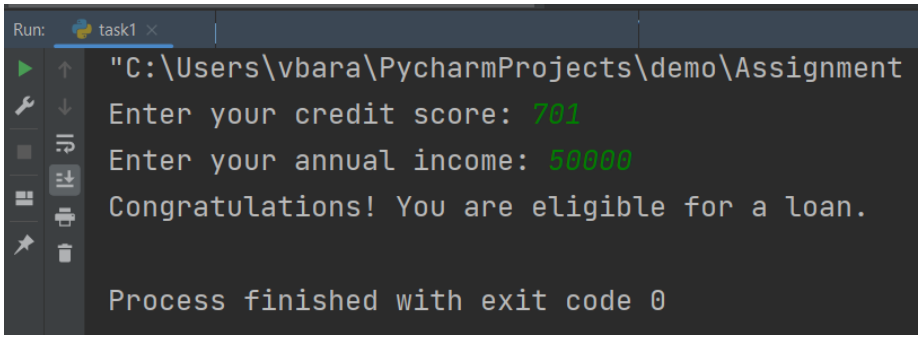
Tasks:

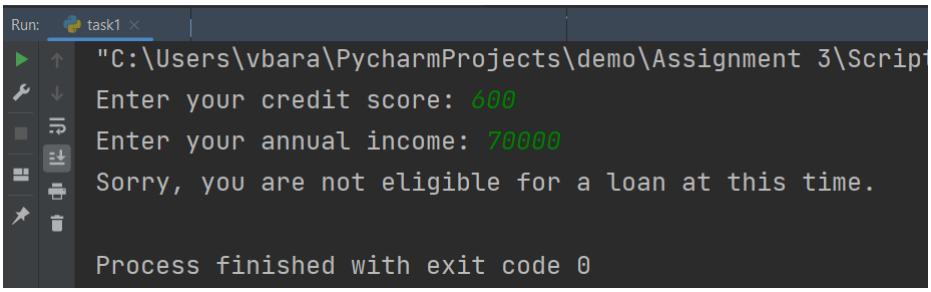
1. Write a program that takes the customer's credit score and annual income as input.
2. Use conditional statements (if-else) to determine if the customer is eligible for a loan.
3. Display an appropriate message based on eligibility.

Source Code:

```
credit_score = int(input("Enter your credit score: "))
annual_income = float(input("Enter your annual income: $"))
if credit_score > 700 and annual_income >= 50000:
    print("Congratulations! You are eligible for a loan.")
else:
    print("Sorry, you are not eligible for a loan at this time.")
```

Output:





Task 2: Nested Conditional Statements

Create a program that simulates an ATM transaction. Display options such as "Check Balance," "Withdraw," "Deposit,". Ask the user to enter their current balance and the amount they want to withdraw or deposit. Implement checks to ensure that the withdrawal amount is not greater than the available balance and that the withdrawal amount is in multiples of 100 or 500. Display appropriate messages for success or failure.

Source Code:

```
while True:    opt = '''Enter the options:
       1----> Check Balance
       2----> Withdraw
       3----> Deposit
       4----> Exit'''
    print(opt)
    option = input()

    if option == '1':
        print("Available Balance:", available_balance)
    elif option == '2':
        print("Enter the amount:")
        amount = int(input())
        if available_balance <= 0 or amount % 100 != 0 or available_balance <
amount:
            print("Invalid amount entered. Please check the amount.")
        else:
            available_balance -= amount
            print("Withdrawal successful. Available Balance:",
available_balance)
    elif option == '3':
        print("Enter the amount to be deposited:")
        amount = int(input())
        available_balance += amount
        print("Deposit successful. Available Balance:", available_balance)
    elif option == '4':
        print("Thanks for banking with us!")
        break
    else:
        print("Invalid option. Please enter a valid option (1-4).")
```

Output:

```
Run: task1 task2
"C:\Users\vbara\PycharmProjects\demo\Assignment 3\
Enter the options:
        1----> Check Balance
        2----> Withdraw
        3----> Deposit
        4----> Exit
1
Available Balance: 15000
Enter the options:
        1----> Check Balance
        2----> Withdraw
        3----> Deposit
        4----> Exit
2
Enter the amount:
1000
```

```
Withdrawal successful. Available Balance: 14000
Enter the options:
        1----> Check Balance
        2----> Withdraw
        3----> Deposit
        4----> Exit
3
Enter the amount to be deposited:
500
Deposit successful. Available Balance: 14500
Enter the options:
        1----> Check Balance
        2----> Withdraw
        3----> Deposit
        4----> Exit
4
Thanks for banking with us!

Process finished with exit code 0
```

Task 3: Loop Structures

You are responsible for calculating compound interest on savings accounts for bank customers. You need to calculate the future balance for each customer's savings account after a certain number of years.

Tasks:

1. Create a program that calculates the future balance of a savings account.

2. Use a loop structure (e.g., for loop) to calculate the balance for multiple customers.

3. Prompt the user to enter the initial balance, annual interest rate, and the number of years.

4. Calculate the future balance using the formula:

future_balance = initial_balance * (1 + annual_interest_rate/100)^years.

5. Display the future balance for each customer.

Source Code:

```
num_customers = int(input("Enter the number of customers: "))
for customer in range(1, num_customers + 1):
    print(f"\nCustomer {customer}:")
    initial_balance = float(input("Enter the initial balance: "))
    annual_interest_rate = float(input("Enter the annual interest rate (in
percentage): "))
    years = int(input("Enter the number of years: "))
    future_balance = initial_balance * (1 + annual_interest_rate/100)**years
    print(f"\nFuture Balance for Customer {customer} after {years} years:
{future_balance:.2f}")
```

Output:

```
Run: task3
"C:\Users\vbara\PycharmProjects\demo\Assignment 3\Scripts\python
Enter the number of customers: 2
Customer 1:
Enter the initial balance: 15000
Enter the annual interest rate (in percentage): 40
Enter the number of years: 3
Future Balance for Customer 1 after 3 years: 41160.00
Customer 2:
Enter the initial balance: 12000
Enter the annual interest rate (in percentage): 12
Enter the number of years: 2
Future Balance for Customer 2 after 2 years: 15052.80

Process finished with exit code 0
```

Task 4: Looping, Array and Data Validation

You are tasked with creating a program that allows bank customers to check their account balances. The program should handle multiple customer accounts, and the customer should be able to enter their account number, balance to check the balance.

Tasks:

1. Create a Python program that simulates a bank with multiple customer accounts.

2. Use a loop (e.g., while loop) to repeatedly ask the user for their account number and

balance until they enter a valid account number.

3. Validate the account number entered by the user.

4. If the account number is valid, display the account balance. If not, ask the user to try again.

Source Code:

```
info={
    'A101':15000,
    'A102':16000,
    'A103':17000,
    'A104':18000
}
while True:
    type=input("Enter the account number")
    if type in info.keys():
        print("You are a valid customer")
        print(f"Your Balance for Account_Id:{type} is {info[type]}")
        break
    else:
        print("Invalid Account Number. Please Try Again....")
```

Output:

```
Run: task4
"C:\Users\vbara\PycharmProjects\demo\Assignment 3\
Enter the account numberA106
Invalid Account Number. Please Try Again....
Enter the account numberA103
You are a valid customer
Your Balance for Account_Id:A103 is 17000

Process finished with exit code 0
```

Task 5: Password Validation

Write a program that prompts the user to create a password for their bank account. Implement if conditions to validate the password according to these rules:

- The password must be at least 8 characters long.
- It must contain at least one uppercase letter.
- It must contain at least one digit.
- Display appropriate messages to indicate whether their password is valid or not.

Source Code:

```
while True:
    password = input("Enter your password: ")
    if len(password) < 8:
        print("Your password must be at least 8 characters long.")
    elif not any(char.isupper() for char in password):
        print("Your password must contain at least one uppercase letter.")
    elif not any(char.isdigit() for char in password):
        print("Your password must contain at least one digit.")
    else:
        print("Password successfully created. Your account is now secure.")
        break
```

Output:

```
Run: task5
"C:\Users\vbara\PycharmProjects\demo\Assignment 3\Scripts\pytho
Enter your password: bala
Your password must be at least 8 characters long.
Enter your password: balakumaran
Your password must contain at least one uppercase letter.
Enter your password: Balakumaran
Your password must contain at least one digit.
Enter your password: Balakumaran_5
Password successfully created. Your account is now secure.

Process finished with exit code 0
```

Task 5:

Create a program that maintains a list of bank transactions (deposits and withdrawals) for a customer.

Use a while loop to allow the user to keep adding transactions until they choose to exit. Display the transaction history upon exit using looping statements.

Source Code:

```
transaction_history = []
opt='''Enter your Choice for Bank Transaction
           1------>Deposit
           2------>Withdrawal
           3------>Exit
        '''
print(opt)
while True:
    choice = input("Enter your choice (1-3): ")
    if choice == '1':
        amount = float(input("Enter the deposit amount: "))
        transaction_history.append(('Deposit', amount))
        print(f"Deposit of {amount:.2f} done.")
    elif choice == '2':
        amount = float(input("Enter the withdrawal amount: "))
        transaction_history.append(('Withdrawal', amount))
        print(f"Withdrawal of {amount:.2f} from your account.")
    elif choice == '3':
        print("Transaction History:")
        for transaction_type, amount in transaction_history:
            print(f"{transaction_type}: {amount:.2f}")
        print("Thank you for Banking with us!")
        break
    else:
        print("Invalid choice. Please enter a number between 1 and 3.")
```

Output:

```
Run: task6
"C:\Users\vbara\PycharmProjects\demo\Assignment 3\Scripts\
Enter your Choice for BanK Transaction
            1------>Deposit
            2------>Withdrawal
            3------>Exit

Enter your choice (1-3): 1
Enter the deposit amount: 1000
Deposit of 1000.00 done.
Enter your choice (1-3): 2
Enter the withdrawal amount: 500
Withdrawal of 500.00 from your account.
Enter your choice (1-3): 2
Enter the withdrawal amount: 100
Withdrawal of 100.00 from your account.
Enter your choice (1-3): 200
Invalid choice. Please enter a number between 1 and 3.
Enter your choice (1-3): 3
```

```
Transaction History:
Deposit: 1000.00
Withdrawal: 500.00
Withdrawal: 100.00
Thank you for Banking with us!

Process finished with exit code 0
```

Task 7: Class & Object

1. Create a `Customer` class with the following confidential attributes:

• Attributes

o Customer ID o First Name o Last Name o Email Address o Phone Number o Address

• Constructor and Methods

o Implement default constructors and overload the constructor with Customer attributes, generate getter and setter, (print all information of attribute) methods for the attributes.

2. Create an `Account` class with the following confidential attributes:

• Attributes

o Account Number o Account Type (e.g., Savings, Current) o Account Balance

• Constructor and Methods

o Implement default constructors and overload the constructor with Account attributes,

o Generate getter and setter, (print all information of attribute) methods for the attributes.

o Add methods to the `Account` class to allow deposits and withdrawals.

- deposit(amount: float): Deposit the specified amount into the account.

- withdraw(amount: float): Withdraw the specified amount from the account. withdraw amount only if there is sufficient fund else display insufficient balance.

- calculate_interest(): method for calculating interest amount for the available balance. interest rate is fixed to 4.5%

• Create a Bank class to represent the banking system. Perform the following operation in main method:

o create object for account class by calling parameter constructor.

o deposit(amount: float): Deposit the specified amount into the account.

o withdraw(amount: float): Withdraw the specified amount from the account.

o calculate_interest(): Calculate and add interest to the account balance for savings accounts.

Source Code:

```
class Customer:
    def __init__(self, CustomerID=None, First_Name=None, Last_Name=None,
Email=None, Phone_NO=None, Address=None):
        self.CustomerID = CustomerID
        self.First_Name = First_Name
        self.Last_Name = Last_Name
        self.Email = Email
        self.Phone_NO = Phone_NO
        self.Address = Address

    @property
    def CustomerID(self):
        return self._CustomerID
```

```python
    @CustomerID.setter
    def CustomerID(self, value):
        self._CustomerID = value

    @property
    def First_Name(self):
        return self._First_Name

    @First_Name.setter
    def First_Name(self, value):
        self._First_Name = value

    @property
    def Last_Name(self):
        return self._Last_Name

    @Last_Name.setter
    def Last_Name(self, value):
        self._Last_Name = value

    @property
    def Email(self):
        return self._Email

    @Email.setter
    def Email(self, value):
        self._Email = value

    @property
    def Phone_NO(self):
        return self._Phone_NO

    @Phone_NO.setter
    def Phone_NO(self, value):
        self._Phone_NO = value

    @property
    def Address(self):
        return self._Address
```

```
    @Address.setter
    def Address(self, value):
        self._Address = value

    def print_info(self):
        print("Customer ID: ", self.CustomerID)
        print("First Name: ", self.First_Name)
        print("Last Name: ", self.Last_Name)
        print("Email: ", self.Email)
        print("Phone Number: ", self.Phone_NO)
        print("Address: ", self.Address)

class Account:
    def __init__(self, Account_NO=None, Account_Type=None,
Account_Balance=None):
        self._Account_NO = Account_NO
        self._Account_Type = Account_Type
        self._Account_Balance = Account_Balance

    @property
    def Account_NO(self):
        return self._Account_NO

    @Account_NO.setter
    def Account_NO(self, value):
        self._Account_NO = value

    @property
    def Account_Type(self):
        return self._Account_Type

    @Account_Type.setter
    def Account_Type(self, value):
        self._Account_Type = value

    @property
    def Account_Balance(self):
        return self._Account_Balance

    @Account_Balance.setter
    def Account_Balance(self, value):
```

```
        self._Account_Balance = value

    def Deposit(self, Amount):
        if Amount > 0:
            self._Account_Balance += Amount
            print(f"Amount of {Amount} has been deposited")
        else:
            print("Enter a valid amount")

    def Withdraw(self, Amount):
        if Amount <= self._Account_Balance:
            self._Account_Balance -= Amount
            print(f"Withdrawal of amount {Amount} has been done")
        else:
            print("Insufficient balance in your account")

    def calculate_interest(self):
        result = self._Account_Balance * 0.045
        self._Account_Balance+=result
        print("Calculated interest rate for this account: ",result)


class Bank:
    def main(self):
        c1 = Customer('1', 'bala', 'kumaran', 'balankkumaran55@gmail.com',
'6382474871', 'Pondicherry')
        c1.print_info()
        a1 = Account('A101', 'Savings', 15000)
        a1.Deposit(1000)
        a1.Withdraw(600)
        a1.calculate_interest()

b1 = Bank()
b1.main()
```

Output:

```
Run: task7
"C:\Users\vbara\PycharmProjects\demo\Assignment 3\Scripts\pytho
Customer ID:  1
First Name:  bala
Last Name:  kumaran
Email:  balankkumaran55@gmail.com
Phone Number:  6382474871
Address:  Pondicherry
Amount of 1000 has been deposited
Withdrawal of amount 600 has been done
Calculated interest rate for this account:  693.0
```

Task 8: Inheritance and polymorphism

1. Overload the deposit and withdraw methods in Account class as mentioned below.

• deposit(amount: float): Deposit the specified amount into the account.

• withdraw(amount: float): Withdraw the specified amount from the account. withdraw amount only if there is sufficient fund else display insufficient balance.

• deposit(amount: int): Deposit the specified amount into the account.

• withdraw(amount: int): Withdraw the specified amount from the account. withdraw amount only if there is sufficient fund else display insufficient balance.

• deposit(amount: double): Deposit the specified amount into the account.

• withdraw(amount: double): Withdraw the specified amount from the account. withdraw amount only if there is sufficient fund else display insufficient balance.

Source Code:

```
class Account:
    def __init__(self, Account_NO=None, Account_Type=None,
Account_Balance=None):
        self._Account_NO = Account_NO
        self._Account_Type = Account_Type
        self._Account_Balance = Account_Balance

    @property
    def Account_NO(self):
        return self._Account_NO

    @Account_NO.setter
```

```
    def Account_NO(self, value):
        self._Account_NO = value

    @property
    def Account_Type(self):
        return self._Account_Type

    @Account_Type.setter
    def Account_Type(self, value):
        self._Account_Type = value

    @property
    def Account_Balance(self):
        return self._Account_Balance

    @Account_Balance.setter
    def Account_Balance(self, value):
        self._Account_Balance = value

    def Deposit(self, Amount:int):
        if Amount > 0:
            self._Account_Balance += Amount
            print(f"Amount of {Amount} has been deposited")
        else:
            print("Enter a valid amount")

    def Withdraw(self, Amount:int):
        if Amount <= self._Account_Balance:
            self._Account_Balance -= Amount
            print(f"Withdrawal of amount {Amount} has been done")
        else:
            print("Insufficient balance in your account")
    def Deposit(self, Amount:float):
        if Amount > 0:
            self._Account_Balance += Amount
            print(f"Amount of {Amount} has been deposited")
        else:
            print("Enter a valid amount")

    def Withdraw(self, Amount:float):
        if Amount <= self._Account_Balance:
```

```python
            self._Account_Balance -= Amount
            print(f"Withdrawal of amount {Amount} has been done")
        else:
            print("Insufficient balance in your account")

    def calculate_interest(self):
        result = self._Account_Balance * 0.045
        self._Account_Balance+=result
        print("Calculated interest rate for this account: ",result)


class Bank:
    def main(self):
        c1 = Customer('1', 'bala', 'kumaran', 'balankkumaran55@gmail.com',
'6382474871', 'Pondicherry')
        c1.print_info()
        a1 = Account('A101', 'Savings', 15000)
        a1.Deposit(1000)
        a1.Deposit(1000.50)
        a1.Withdraw(600.0)
        a1.calculate_interest()

b1 = Bank()
b1.main()
```

Output:

```
Run: task7
"C:\Users\vbara\PycharmProjects\demo\Assignment 3\Scripts\python.exe
Customer ID:  1
First Name:  bala
Last Name:  kumaran
Email:  balankkumaran55@gmail.com
Phone Number:  6382474871
Address:  Pondicherry
Amount of 1000 has been deposited
Amount of 1000.5 has been deposited
Withdrawal of amount 600.0 has been done
Calculated interest rate for this account:  738.0224999999999
```

2. Create Subclasses for Specific Account Types

• Create subclasses for specific account types (e.g., `SavingsAccount`, `CurrentAccount`) that inherit from the `Account` class.

o SavingsAccount: A savings account that includes an additional attribute for interest rate. override the calculate_interest() from Account class method to calculate interest based on the balance and interest rate.

o CurrentAccount: A current account that includes an additional attribute overdraftLimit. A current account with no interest. Implement the withdraw() method to allow overdraft up to a certain limit (configure a constant for the overdraft limit).

```
class SavingsAccount(Account):
    def __init__(self, Account_NO=None, Account_Type=None,
Account_Balance=None, Interest_Rate=None):
        super().__init__(Account_NO, Account_Type, Account_Balance)
        self.Interest_Rate = Interest_Rate
    def calculate_interest(self):
        if self.Interest_Rate:
            result = self._Account_Balance * self.Interest_Rate
            self._Account_Balance += result
            print("Calculated interest rate for this account:", result)
        else:
            print("No interest rate specified for SavingsAccount")


class CurrentAccount(Account):
    OVERDRAFT_LIMIT = 1000
    def __init__(self, Account_NO=None, Account_Type=None,
Account_Balance=None, overdraftLimit=None):
        super().__init__(Account_NO, Account_Type, Account_Balance)
        self.overdraftLimit = self.OVERDRAFT_LIMIT

    def withdraw(self, amount):
        if amount <= self._Account_Balance + self.overdraftLimit:
            self._Account_Balance -= amount
            print(f"Withdrawal of amount {amount} has been done")
        else:
            print("Insufficient balance in your account and overdraft limit
exceeded")
```

3. Create a Bank class to represent the banking system. Perform the following operation in main method:

• Display menu for user to create object for account class by calling parameter constructor. Menu should display options `SavingsAccount` and `CurrentAccount`. user can choose any one option to create account. use switch case for implementation.

• deposit(amount: float): Deposit the specified amount into the account.

• withdraw(amount: float): Withdraw the specified amount from the account. For saving account withdraw amount only if there is sufficient fund else display insufficient balance. For Current Account withdraw limit can exceed the available balance and should not exceed the overdraft limit.

• calculate_interest(): Calculate and add interest to the account balance for savings accounts.

```
class Bank:
    def main(self):
        while True:
            print("1. Create Savings Account")
            print("2. Create Current Account")
            print("3. Exit")

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

            if choice == 1:
                sa = SavingsAccount()
                sa.Account_NO = input("Enter account number: ")
                sa.Account_Type = 'Savings'
                sa.Account_Balance = float(input("Enter initial balance: "))
                sa.Interest_Rate = float(input("Enter interest rate: "))
                sa.Deposit(float(input("Enter the amount to be deposited: ")))
                sa.calculate_interest()

            elif choice == 2:
                ca = CurrentAccount()
                ca.Account_NO = input("Enter account number: ")
                ca.Account_Type = 'Current'
                ca.Account_Balance = float(input("Enter initial balance: "))
                ca.withdraw(float(input("Enter withdrawal amount: ")))
```

```
            elif choice == 3:
                print("Exiting...")
                break
            else:
                print("Invalid choice. Please try again.")

b1 = Bank()
b1.main()
```

Output:

```
Run: task7
"C:\Users\vbara\PycharmProjects\demo\Assignment 3\Scripts\python.exe"
1. Create Savings Account
2. Create Current Account
3. Exit
Enter your choice: 1
Enter account number: A102
Enter initial balance: 5000
Enter interest rate: 3.2
Enter the amount to be deposited: 500
Amount of 500.0 has been deposited
Calculated interest rate for this account: 17600.0
1. Create Savings Account
2. Create Current Account
3. Exit
Enter your choice: 2
Enter account number: A103
Enter initial balance: 6000
Enter withdrawal amount: 10000
Insufficient balance in your account and overdraft limit exceeded
```

```
1. Create Savings Account
2. Create Current Account
3. Exit
Enter your choice: 3
Exiting...

Process finished with exit code 0
```

Task 10: Has A Relation / Association

1. Create a `Customer` class with the following attributes:

• Customer ID

• First Name

• Last Name

• Email Address (validate with valid email address)

• Phone Number (Validate 10-digit phone number)

• Address

• Methods and Constructor:

o Implement default constructors and overload the constructor with Account attributes, generate getter, setter, print all information of attribute) methods for the attributes.

```
import re

class Customer:
    def __init__(self, customer_id, first_name, last_name, email, phone_number,
address):
        self.customer_id = customer_id
        self.first_name = first_name
        self.last_name = last_name
        self.email = email
        self.phone_number = phone_number
        self.address = address

    def validate_email(self, email):
        pattern = re.compile(r"[^@]+@[^@]+\.[^@]+")
        return bool(pattern.match(email))

    def validate_phone_number(self, phone_number):
        return len(str(phone_number)) == 10 and str(phone_number).isdigit()

    @property
    def customer_id(self):
        return self._customer_id

    @customer_id.setter
    def customer_id(self, value):
        self._customer_id = value

    @property
```

```python
    def first_name(self):
        return self._first_name

    @first_name.setter
    def first_name(self, value):
        self._first_name = value

    @property
    def last_name(self):
        return self._last_name

    @last_name.setter
    def last_name(self, value):
        self._last_name = value

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

    @email.setter
    def email(self, value):
        if self.validate_email(value):
            self._email = value
        else:
            raise ValueError("Invalid email format")

    @property
    def phone_number(self):
        return self._phone_number

    @phone_number.setter
    def phone_number(self, value):
        if self.validate_phone_number(value):
            self._phone_number = value
        else:
            raise ValueError("Invalid phone number format")

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

```
    @address.setter
    def address(self, value):
        self._address = value

    def print_info(self):
        print(f"Customer ID: {self.customer_id}")
        print(f"First Name: {self.first_name}")
        print(f"Last Name: {self.last_name}")
        print(f"Email: {self.email}")
        print(f"Phone Number: {self.phone_number}")
        print(f"Address: {self.address}")
```

2. Create an `Account` class with the following attributes:

• Account Number (a unique identifier).

• Account Type (e.g., Savings, Current)

• Account Balance

• Customer (the customer who owns the account)

• Methods and Constructor:

o Implement default constructors and overload the constructor with Account attributes, generate getter, setter, (print all information of attribute) methods for the attributes.

```
class Account:
    account_counter = 1000

    def __init__(self, account_type, balance, customer):
        self.account_number = Account.account_counter
        Account.account_counter += 1
        self.account_type = account_type
        self.balance = balance
        self.customer = customer

    @property
    def account_number(self):
        return self._account_number

    @account_number.setter
    def account_number(self, value):
        self._account_number = value

    @property
```

```python
    def account_type(self):
        return self._account_type

    @account_type.setter
    def account_type(self, value):
        self._account_type = value

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

    @balance.setter
    def balance(self, value):
        self._balance = value

    @property
    def customer(self):
        return self._customer

    @customer.setter
    def customer(self, value):
        self._customer = value

    def print_info(self):
        print(f"Account Number: {self.account_number}")
        print(f"Account Type: {self.account_type}")
        print(f"Balance: {self.balance}")
        print("Customer Information:")
        self.customer.print_info()
```

Create a Bank Class and must have following requirements:

1. Create a Bank class to represent the banking system. It should have the following methods:

• create_account(Customer customer, long accNo, String accType, float balance): Create a new bank account for the given customer with the initial balance.

• get_account_balance(account_number: long): Retrieve the balance of an account given

its account number. should return the current balance of account.

• deposit(account_number: long, amount: float): Deposit the specified amount into the account. Should return the current balance of account.

• withdraw(account_number: long, amount: float): Withdraw the specified amount from the account. Should return the current balance of account.

• transfer(from_account_number: long, to_account_number: int, amount: float): Transfer money from one account to another.

• getAccountDetails(account_number: long): Should return the account and customer details.

2. Ensure that account numbers are automatically generated when an account is created, starting from 1001 and incrementing for each new account.

```
class Bank:
    accounts = []

    def create_account(self, customer, account_type, balance):
        account = Account(account_type, balance, customer)
        Bank.accounts.append(account)
        print(f"Account created successfully. Account Number:
{account.account_number}")

    def get_account_balance(self, account_number):
        account = None
        for acc in Bank.accounts:
            if acc.account_number == account_number:
                account = acc
                break
        if account:
            return account.balance
        else:
            return None

    def deposit(self, account_number, amount):
        account = None
        for acc in Bank.accounts:
            if acc.account_number == account_number:
                account = acc
                break
        if account:
            account.balance += amount
            return account.balance
        else:
            return None

    def withdraw(self, account_number, amount):
        account = None
        for acc in Bank.accounts:
            if acc.account_number == account_number:
```

```
                account = acc
                break
        if account and account.balance >= amount:
            account.balance -= amount
            return account.balance
        else:
            return None

    def transfer(self, from_account_number, to_account_number, amount):
        from_account = None
        for acc in Bank.accounts:
            if acc.account_number == from_account_number:
                from_account = acc
                break

        to_account = None
        for acc in Bank.accounts:
            if acc.account_number == to_account_number:
                to_account = acc
                break

        if from_account and to_account and from_account.balance >= amount:
            from_account.balance -= amount
            to_account.balance += amount
            return True
        else:
            return False

    def get_account_details(self, account_number):
        account = None
        for acc in Bank.accounts:
            if acc.account_number == account_number:
                account = acc
                break
        if account:
            account.print_info()
        else:
            print("Account not found.")
```

3. Create a BankApp class with a main method to simulate the banking system. Allow the user to interact with the system by entering commands such as "create_account", "deposit", "withdraw", "get_balance", "transfer", "getAccountDetails" and "exit." create_account should display sub menu to choose type of accounts and repeat this operation until user exit

Source Code:

```
class BankApp:
    def main(self):
        bank = Bank()
        while True:
            print("1. Create Account")
            print("2. Get Account Balance")
            print("3. Deposit")
            print("4. Withdraw")
            print("5. Transfer")
            print("6. Get Account Details")
            print("7. Exit")

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

            if choice == 1:
                print("1. Savings Account")
                print("2. Current Account")
                account_type_choice = int(input("Choose the account type: "))
                customer_id = input("Enter Customer ID: ")
                first_name = input("Enter First Name: ")
                last_name = input("Enter Last Name: ")
                email = input("Enter Email: ")
                phone_number = input("Enter Phone Number: ")
                address = input("Enter Address: ")

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

                balance = float(input("Enter Initial Balance: "))
                if account_type_choice == 1:
                    account_type = "Savings"
                else:
                    account_type = "Current"

                bank.create_account(customer, account_type, balance)
```

```
            elif choice == 2:
                account_number = int(input("Enter Account Number: "))
                balance = bank.get_account_balance(account_number)
                if balance is not None:
                    print(f"Account Balance: {balance}")
                else:
                    print("Account not found.")

            elif choice == 3:
                account_number = int(input("Enter Account Number: "))
                amount = float(input("Enter Deposit Amount: "))
                new_balance = bank.deposit(account_number, amount)
                if new_balance is not None:
                    print(f"Deposit successful. New Balance: {new_balance}")
                else:
                    print("Account not found or invalid amount.")

            elif choice == 4:
                account_number = int(input("Enter Account Number: "))
                amount = float(input("Enter Withdrawal Amount: "))
                new_balance = bank.withdraw(account_number, amount)
                if new_balance is not None:
                    print(f"Withdrawal successful. New Balance: {new_balance}")
                else:
                    print("Account not found or insufficient balance.")

            elif choice == 5:
                from_account_number = int(input("Enter From Account Number: "))
                to_account_number = int(input("Enter To Account Number: "))
                amount = float(input("Enter Transfer Amount: "))
                success = bank.transfer(from_account_number, to_account_number,
amount)
                if success:
                    print("Transfer successful.")
                else:
                    print("Transfer failed. Check account details or
insufficient balance.")

            elif choice == 6:
                account_number = int(input("Enter Account Number: "))
                bank.get_account_details(account_number)
```

```
            elif choice == 7:
                print("Exiting...")
                break

            else:
                print("Invalid choice. Please try again.")

bank_app = BankApp()
bank_app.main()
```

Output:

Creating an account for the user named Balakumaran P of type savings and providing the initial balance.

```
Run: task9
"C:\Users\vbara\PycharmProjects\demo\Assignment 3\Scripts\python.exe"
1. Create Account
2. Get Account Balance
3. Deposit
4. Withdraw
5. Transfer
6. Get Account Details
7. Exit
Enter your choice: 1
1. Savings Account
2. Current Account
Choose the account type: 1
Enter Customer ID: 1
Enter First Name: Balakumaran
Enter Last Name: P
Enter Email: balabkkumaran55@gmail.com
Enter Phone Number: 6382474871
Enter Address: Pondicherry
Enter Initial Balance: 15000
Account created successfully. Account Number: 1000
```

Trying to get account balance for invalid customer who's account has not been created.

```
1. Create Account
2. Get Account Balance
3. Deposit
4. Withdraw
5. Transfer
6. Get Account Details
7. Exit
Enter your choice: 2
Enter Account Number: 1
Account not found.
```

Getting balance amount for the account ID 1000 (Account Holder : Balakumaran P)

```
1. Create Account
2. Get Account Balance
3. Deposit
4. Withdraw
5. Transfer
6. Get Account Details
7. Exit
Enter your choice: 2
Enter Account Number: 1000
Account Balance: 15000.0
```

Trying to deposit a amount of 200 to the account number 1000.

```
1. Create Account
2. Get Account Balance
3. Deposit
4. Withdraw
5. Transfer
6. Get Account Details
7. Exit
Enter your choice: 3
Enter Account Number: 1000
Enter Deposit Amount: 200
Deposit successful. New Balance: 15200.0
```

Triying to withdraw amount of 700 from the account number 500. As there is account created with an account number 500 its printing Account not found .

```
1. Create Account
2. Get Account Balance
3. Deposit
4. Withdraw
5. Transfer
6. Get Account Details
7. Exit
Enter your choice: 4
Enter Account Number: 500
Enter Withdrawal Amount: 700
Account not found or insufficient balance.
```

Trying to withdraw an amount of 1000 from the account number 1000:

```
1. Create Account
2. Get Account Balance
3. Deposit
4. Withdraw
5. Transfer
6. Get Account Details
7. Exit
Enter your choice: 4
Enter Account Number: 1000
Enter Withdrawal Amount: 1000
Withdrawal successful. New Balance: 14200.0
```

Creating another account with account number 1001 for the holder Rahul Sharma:

```
1. Create Account
2. Get Account Balance
3. Deposit
4. Withdraw
5. Transfer
6. Get Account Details
7. Exit
Enter your choice: 1
1. Savings Account
2. Current Account
Choose the account type: 1
Enter Customer ID: 1
Enter First Name: Rahul
Enter Last Name: Sharma
Enter Email: rahulrock@gmail.com
Enter Phone Number: 9585335667
Enter Address: Pondicherry
Enter Initial Balance: 12000
Account created successfully. Account Number: 1001
```

Transfering an amount of 700 from the account number 1000 to the account number 1001.

```
1. Create Account
2. Get Account Balance
3. Deposit
4. Withdraw
5. Transfer
6. Get Account Details
7. Exit
Enter your choice: 5
Enter From Account Number: 1000
Enter To Account Number: 1001
Enter Transfer Amount: 700
Transfer successful.
```

Getting Account details of a particular customer:

```
1. Create Account
2. Get Account Balance
3. Deposit
4. Withdraw
5. Transfer
6. Get Account Details
7. Exit
Enter your choice: 6
Enter Account Number: 1001
Account Number: 1001
Account Type: Savings
Balance: 12700.0
Customer Information:
Customer ID: 1
First Name: Rahul
Last Name: Sharma
Email: rahulrock@gmail.com
Phone Number: 9585335667
Address: Pondicherry
```

Task 11: Interface/abstract class, and Single Inheritance, static variable

1. Create a 'Customer' class as mentioned above task.

```
class Customer:
    def __init__(self, customer_id, name, email):
        self.customer_id = customer_id
        self.name = name
        self.email = email
```

2. Create an class 'Account' that includes the following attributes. Generate account number using static variable.

• Account Number (a unique identifier). • Account Type (e.g., Savings, Current)

• Account Balance • Customer (the customer who owns the account) • lastAccNo

```
class Account():
    last_acc_no = 0

    def __init__(self, account_type, customer, balance):
        Account.last_acc_no += 1
        self.account_number = Account.last_acc_no
        self.account_type = account_type
        self.customer = customer
        self.balance = balance
```

3. Create three child classes that inherit the Account class and each class must contain below mentioned attribute:

• **SavingsAccount**: A savings account that includes an additional attribute for interest rate. Saving account should be created with minimum balance 500.

• **CurrentAccount**: A Current account that includes an additional attribute for overdraftLimit(credit limit). withdraw() method to allow overdraft up to a certain limit.withdraw limit can exceed the available balance and should not exceed the overdraft limit.

• **ZeroBalanceAccount**: ZeroBalanceAccount can be created with Zero balance.

```
class SavingsAccount(Account):
    def __init__(self, customer, balance=500, interest_rate=0.02):
        super().__init__("Savings", customer, balance)
        self.interest_rate = interest_rate
    def calculate_interest(self):
        if self.interest_rate:
            result = self.balance * self.interest_rate
            self.balance += result
            print("Calculated interest rate for this account:", result)
        else:
            print("No interest rate specified for SavingsAccount")
```

```python
class CurrentAccount(Account):
    def __init__(self, customer, balance, overdraft_limit):
        super().__init__("Current", customer, balance)
        self.overdraft_limit = overdraft_limit

    def withdraw(self, amount):
        if amount > self.balance + self.overdraft_limit:
            print("Withdrawal exceeds available balance and overdraft limit.")
        else:
            self.balance -= amount
            print(f"Withdrawal successful. Current balance: {self.balance}")

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

4. Create **ICustomerServiceProvider** interface/abstract class with following functions:

• get_account_balance(account_number: long): Retrieve the balance of an account givenits account number. should return the current balance of account.

• deposit(account_number: long, amount: float): Deposit the specified amount into the account. Should return the current balance of account.

• withdraw(account_number: long, amount: float): Withdraw the specified amount from the account. Should return the current balance of account. A savings account should maintain a minimum balance and checking if the withdrawal violates the minimum

balance rule.

• transfer(from_account_number: long, to_account_number: int, amount: float): Transfer money from one account to another.

• getAccountDetails(account_number: long): Should return the account and customer details.

```python
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):
```

5. Create **IBankServiceProvider** interface/abstract class with following functions:

• create_account(Customer customer, long accNo, String accType, float balance): Create a new bank account for the given customer with the initial balance.

• listAccounts():Account[] accounts: List all accounts in the bank.

• calculateInterest(): the calculate_interest() method to calculate interest based on the balance and interest rate.

```
class IBankServiceProvider(ABC):
    @abstractmethod
    def create_account(self, customer, acc_type, balance):
        pass

    @abstractmethod
    def list_accounts(self):
        pass

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

6. Create CustomerServiceProviderImpl class which implements ICustomerServiceProvider provide all implementation methods.

```
class CustomerServiceProviderImpl(ICustomerServiceProvider):
    accounts = []

    def create_account(self, customer, acc_type, balance):
        if acc_type == "Savings":
            account = SavingsAccount(customer, balance)
        elif acc_type == "Current":
            account = CurrentAccount(customer, balance, overdraft_limit=1000)
        else:
            account = ZeroBalanceAccount(customer)
```

```
        CustomerServiceProviderImpl.accounts.append(account)
        return account

    def list_accounts(self):
        return CustomerServiceProviderImpl.accounts

    def get_account_balance(self, account_number):
        for account in CustomerServiceProviderImpl.accounts:
            if account.account_number == account_number:
                return account.balance
        return None

    def deposit(self, account_number, amount):
        for account in CustomerServiceProviderImpl.accounts:
            if account.account_number == account_number:
                account.balance += amount
                return account.balance
        return None

    def withdraw(self, account_number, amount):
        for account in CustomerServiceProviderImpl.accounts:
    if account.account_number == account_number:
        if account.balance < 0:
            print("Withdrawal violates minimum balance rule.")
            return account.balance
        account.balance -= amount
        return account.balance
return None


    def transfer(self, from_account_number, to_account_number, amount):
        from_account = None
        for acc in CustomerServiceProviderImpl.accounts:
            if acc.account_number == from_account_number:
                from_account = acc
                break

        to_account = None
        for acc in CustomerServiceProviderImpl.accounts:
            if acc.account_number == to_account_number:
```

```
                to_account = acc
                break

        if from_account and to_account and from_account.balance >= amount:
            from_account.balance -= amount
            to_account.balance += amount
            return True
        else:
            return False

    def get_account_details(self, account_number):
        for account in CustomerServiceProviderImpl.accounts:
            if account.account_number == account_number:
                return {"account": account, "customer": account.customer}
        return None
```

7. Create **BankServiceProviderImpl** class which inherits from **CustomerServiceProviderImpl** and implements **IBankServiceProvider**

• Attributes

o accountList: Array of Accounts to store any account objects.

o branchName and branchAddress as String objects

```
class
BankSercicerProviderImpl(CustomerServiceProviderImpl,IBankServiceProvider):

    account_list=[]
    branchName=''
    branchAddress=''

    def create_account(self, customer, acc_type, balance):
        account = super().create_account(customer, acc_type, balance)
        self.account_list.append(account)
        return account
    def list_accounts(self):
        return BankSercicerProviderImpl.account_list

    def calculate_interest(self):
        for account in BankSercicerProviderImpl.account_list:
            if hasattr(account, 'interest_rate'):
                interest = account.balance * account.interest_rate
                account.balance += interest
```

```
                if isinstance(account, SavingsAccount):
                    print(
                        f"Interest of {interest} added to Savings Account
{account.account_number}. New balance: {account.balance}")
```

8. Create BankApp class and perform following operation:

• main method to simulate the banking system. Allow the user to interact with the system by entering choice from menu such as "create_account", "deposit", "withdraw","get_balance", "transfer", "getAccountDetails", "ListAccounts" and "exit."

• create_account should display sub menu to choose type of accounts and repeat this operation until user exit.

```
class BankApp:
    def __init__(self):
        self.customer_service_provider = BankSercicеProviderImpl()

    def display_menu(self):
        print("\nBanking System Menu:")
        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. Exit")

    def create_account_submenu(self):
        print("\nSelect Account Type:")
        print("1. Savings Account")
        print("2. Current Account")
        print("3. Zero Balance Account")

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

            if choice == "1":
                self.create_account_submenu()
                acc_type_choice = input("Enter your choice for account type: ")
                if acc_type_choice in ["1", "2", "3"]:
                    name = input("Enter customer name: ")
```

```
                    email = input("Enter customer email: ")
                    customer = Customer(customer_id=None, name=name,
email=email)

                    if acc_type_choice == "1":
                        self.customer_service_provider.create_account(customer,
"Savings", balance=500)
                    elif acc_type_choice == "2":
                        self.customer_service_provider.create_account(customer,
"Current", balance=1000)
                    elif acc_type_choice == "3":
                        self.customer_service_provider.create_account(customer,
"ZeroBalance", balance=0)
                else:
                    print("Invalid choice. Please enter a valid option.")

            elif choice == "2":
                account_number = int(input("Enter Account Number: "))
                amount = float(input("Enter Deposit Amount: "))
                self.customer_service_provider.deposit(account_number, amount)

            elif choice == "3":
                account_number = int(input("Enter Account Number: "))
                amount = float(input("Enter Withdrawal Amount: "))
                self.customer_service_provider.withdraw(account_number, amount)

            elif choice == "4":
                account_number = int(input("Enter Account Number: "))
                balance =
self.customer_service_provider.get_account_balance(account_number)
                if balance is not None:
                    print(f"Current Balance: {balance}")
                else:
                    print("Account not found.")

            elif choice == "5":
                from_account_number = int(input("Enter From Account Number: "))
                to_account_number = int(input("Enter To Account Number: "))
                amount = float(input("Enter Transfer Amount: "))
                success =
self.customer_service_provider.transfer(from_account_number, to_account_number,
```

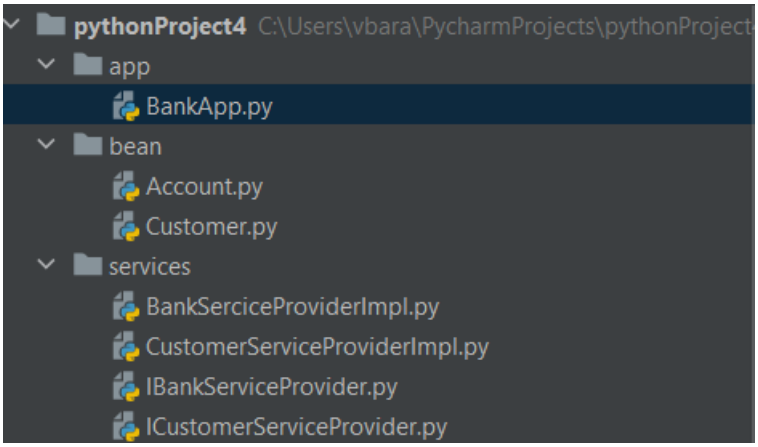
```
amount)
                if success:
                    print("Transfer successful.")
                else:
                    print("Transfer failed. Please check account details and
balance.")

            elif choice == "6":
                account_number = int(input("Enter Account Number: "))
                account_details =
self.customer_service_provider.get_account_details(account_number)
                if account_details is not None:
                    print("Account Details:")
                    print(f"Account Number:
{account_details['account'].account_number}")
                    print(f"Customer Name: {account_details['customer'].name}")
                    print(f"Balance: {account_details['account'].balance}")

                else
                    print("Account not found.")

            elif choice == "7":
                accounts = self.customer_service_provider.list_accounts()
                if accounts:
                    print("List of Accounts:")
                    for account in accounts:
                        print(f"Account Number: {account.account_number}, Type:
{account.account_type}")
                else:
                    print("No accounts found.")

            elif choice == "8":
                print("Exiting the Banking System. Goodbye!")
                break

            else:
                print("Invalid choice. Please enter a valid option.")


if __name__ == "__main__":
```

```
    bank_app = BankApp()
    bank_app.main()
```

9. Place the interface/abstract class in service package and interface/abstract class implementation

class, account class in bean package and Bank class in app package.

I organized the project by creating folders like 'app', 'bean', and 'service'. I then moved files into their respective folders and updated import statements.



Implementation:

Creating a new account for the customer Balakumaran in Savings type:

```
Run: BankApp
Banking System Menu:
1. Create Account
2. Deposit
3. Withdraw
4. Get Balance
5. Transfer
6. Get Account Details
7. List Accounts
8. Exit
Enter your choice: 1

Select Account Type:
1. Savings Account
2. Current Account
3. Zero Balance Account
Enter your choice for account type: 1
Enter customer name: Balakumaran
Enter customer email: balabkkumaran55@gmail.com
Account created successfully. Account Number: 1 Account Type: Savings
```

Depositing an amount of 200 to the account number 1:

```
Banking System Menu:
1. Create Account
2. Deposit
3. Withdraw
4. Get Balance
5. Transfer
6. Get Account Details
7. List Accounts
8. Exit
Enter your choice: 2
Enter Account Number: 1
Enter Deposit Amount: 500
```

Getting the current balance for the account number 1:

```
Banking System Menu:
1. Create Account
2. Deposit
3. Withdraw
4. Get Balance
5. Transfer
6. Get Account Details
7. List Accounts
8. Exit
Enter your choice: 4
Enter Account Number: 1
Current Balance: 900.0
```

Transferring an amount of 200 from account 1 to account 2:

```
Banking System Menu:
1. Create Account
2. Deposit
3. Withdraw
4. Get Balance
5. Transfer
6. Get Account Details
7. List Accounts
8. Exit
Enter your choice: 5
Enter From Account Number: 1
Enter To Account Number: 2
Enter Transfer Amount: 200
Transfer successful.
```

Getting the details of account with their customer details and available balance:

```
Banking System Menu:
1. Create Account
2. Deposit
3. Withdraw
4. Get Balance
5. Transfer
6. Get Account Details
7. List Accounts
8. Exit
Enter your choice: 6
Enter Account Number: 2
Account Details:
Account Number: 2
Customer Name: Kavin kumar
Balance: 700.0
```

Listing the accounts that were created:

```
Banking System Menu:
1. Create Account
2. Deposit
3. Withdraw
4. Get Balance
5. Transfer
6. Get Account Details
7. List Accounts
8. Exit
Enter your choice: 7
List of Accounts:
Account Number: 1, Type: Savings
Account Number: 2, Type: Savings
```

Task 12: Exception Handling

throw the exception whenever needed and Handle in main method,

1. **InsufficientFundException** throw this exception when user try to withdraw amount or transfer amount to another account and the account runs out of money in the account.

2. **InvalidAccountException** throw this exception when user entered the invalid account number when tries to transfer amount, get account details classes

Source Code:

```
class InsufficientFundException(Exception):
    def __init__(self):
        super().__init__("You have Insufficient fund in your account.")

class InvalidAccountException(Exception):
    def __init__(self):
        super().__init__("User entered the invalid account number when tries to
transfer amount")

class OverDraftLimitExcededException(Exception):
    def __init__(self):
        super().__init__("Over Draft limit exceeded..")
```

In the withdraw method of the CustomerServiceProviderImpl class, when the user attempts to withdraw an amount that is less than their actual available balance, I have designed a custom exception named InsufficientFundException to be thrown when the user tries to withdraw an insufficient amount

```
def withdraw(self, account_number, amount):
    for account in CustomerServiceProviderImpl.accounts:
        if account.account_number == account_number:
            if account.balance < amount:
                print("Withdrawal violates minimum balance rule.")
                raise InsufficientFundException
                return account.balance
            account.balance -= amount
            return account.balance
    return None
```

Output:

The available balance for the account number is 1:

```
Banking System Menu:
1. Create Account
2. Deposit
3. Withdraw
4. Get Balance
5. Transfer
6. Get Account Details
7. List Accounts
8. Exit
Enter your choice: 4
Enter Account Number: 1
Current Balance: 500
```

I am trying to withdraw an amount of 1000, which exceeds my available balance, causing an error, and then handling the exception:

```
Banking System Menu:
1. Create Account
2. Deposit
3. Withdraw
4. Get Balance
5. Transfer
6. Get Account Details
7. List Accounts
8. Exit
Enter your choice: 3
Enter Account Number: 1
Enter Withdrawal Amount: 1000
Withdrawal violates minimum balance rule.
You have Insufficient fund in your account.
```

In the `transfer` method of the `CustomerServiceProviderImpl` class, if a customer attempts to make a transfer for an amount greater than their available balance, it will raise the `InsufficientFundException`. Additionally, if the account to which the customer tries to send or receive the amount is invalid, it will raise the `InvalidAccountException`.

```python
def transfer(self, from_account_number, to_account_number, amount):
    from_account = None
    for acc in CustomerServiceProviderImpl.accounts:
        if acc.account_number == from_account_number:
            from_account = acc
            break
    to_account = None
    for acc in CustomerServiceProviderImpl.accounts:
        if acc.account_number == to_account_number:
            to_account = acc
            break
    if from_account.balance<amount:
        raise InsufficientFundException

    if from_account and to_account and from_account.balance >= amount:
        from_account.balance -= amount
        to_account.balance += amount
        return True
    else:
        raise InvalidAccountException
```

```python
elif choice == "5":
    try:
        from_account_number = int(input("Enter From Account Number: "))
        to_account_number = int(input("Enter To Account Number: "))
        amount = float(input("Enter Transfer Amount: "))
        success = self.customer_service_provider.transfer(from_account_number,
to_account_number, amount)
        if success:
            print("Transfer successful.")
        else:
            print("Transfer failed. Please check account details and balance.")
    except InvalidAccountException as e:
        print(e)
    except InsufficientFundException as e:
        print(e)
```

Output:

Attempting to transfer an amount from an account that does not exist:

```
Banking System Menu:
1. Create Account
2. Deposit
3. Withdraw
4. Get Balance
5. Transfer
6. Get Account Details
7. List Accounts
8. Exit
Enter your choice: 5
Enter From Account Number: 1
Enter To Account Number: 5
Enter Transfer Amount: 500
User entered the invalid account number when tries to transfer amount
```

Attempting to transfer an amount of 1000 from account 1, which exceeds its available balance:

```
Enter your choice: 5
Enter From Account Number: 1
Enter To Account Number: 2
Enter Transfer Amount: 1000
You have Insufficient fund in your account.
```

Task 14: Database Connectivity.

1. Create a '**Customer**' class as mentioned above task.

```
class Customer:
    def __init__(self, customer_id, first_name, last_name, email, phone_number,
address):
        self.customer_id = customer_id
        self.first_name = first_name
        self.last_name = last_name
        self.email = email
        self.phone_number = phone_number
        self.address = addres
```

2. Create an class '**Account**' that includes the following attributes. Generate account number using static variable.

- Account Number (a unique identifier).
- Account Type (e.g., Savings, Current)
- Account Balance
- Customer (the customer who owns the account)
- lastAccNo

```
class Account:
    account_counter = 1000

    def __init__(self, account_type, balance, customer):
        self.account_number = Account.account_counter
        Account.account_counter += 1
        self.account_type = account_type
        self.balance = balance
        self.customer = customer
```

3. Create a class '**TRANSACTION**' that include following attributes

- Account
- Description
- Date and Time
- TransactionType(Withdraw, Deposit, Transfer)
- TransactionAmount

```
class Transaction:

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

4. Create three child classes that inherit the Account class and each class must contain below mentioned attribute:

• **SavingsAccount**: A savings account that includes an additional attribute for interest rate. Saving account should be created with minimum balance 500.

• **CurrentAccount**: A Current account that includes an additional attribute for overdraftLimit(credit limit).

• **ZeroBalanceAccount**: ZeroBalanceAccount can be created with Zero balance.

```
class SavingsAccount(Account):
    def __init__(self, customer, balance=500, interest_rate=0.02):
        super().__init__("Savings", customer, balance)
        self.interest_rate = interest_rate

class CurrentAccount(Account):
    def __init__(self,customer,balance,overdrafted_limit):
        super().__init__(self,customer)
        self.balance=balance
        self.overdrafted_limit=overdrafted_limit

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

5. Create **ICustomerServiceProvider** interface/abstract class with following functions:

• **get_account_balance**(account_number: long): Retrieve the balance of an account given its account number. should return the current balance of account.

• **deposit**(account_number: long, amount: float): Deposit the specified amount into the account. Should return the current balance of account.

• **withdraw**(account_number: long, amount: float): Withdraw the specified amount from the account. Should return the current balance of account.

o A savings account should maintain a minimum balance and checking if the withdrawal violates the minimum balance rule.

o Current account customers are allowed withdraw overdraftLimit and available account balance. withdraw limit can exceed the available balance and should not exceed the overdraft limit.

• **transfer**(from_account_number: long, to_account_number: int, amount: float):

Transfer money from one account to another. both account number should be validate from the database use getAccountDetails method.

• **getAccountDetails**(account_number: long): Should return the account and customer details.

• **getTransations**(account_number: long, FromDate:Date, ToDate: Date): Should return the list of transaction between two dates.

```python
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
```

6. **Create IBankServiceProvider interface/abstract class with following functions:**

• **create_account**(Customer customer, long accNo, String accType, float balance): Create a new bank account for the given customer with the initial balance.

• listAccounts(): Array of BankAccount: List all accounts in the bank.(List[Account]accountsList)

• **getAccountDetails**(account_number: long): Should return the account and customer details.

• **calculateInterest**(): the calculate_interest() method to calculate interest based on the balance and interest rate.

```python
class IBankServiceProvider(ABC):

    @abstractmethod
    def create_account(self,customer,account_number,account_type,balance):
        pass

    @abstractmethod
    def lisAccounts(self):
```

```
        pass

    @abstractmethod
    def getAccountDetails(self,account_number):
        pass

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

7. Create **CustomerServiceProviderImpl** class which implements **ICustomerServiceProvider** provide all implementation methods. These methods do not interact with database directly.

```
class CustomerServiceProviderImpl(ICustomerServiceProvider):
    def __init__(self):
        self.accounts={}
        self.transactions=[]

    def get_account_balance(self, account_number):
        if account_number in self.accounts:
            return self.accounts[account_number].balance
        else:
            return None

    def deposit(self, account_number, amount):
        if account_number in self.accounts:
            self.accounts[account_number].balance+=amount

transaction=Transaction(self.accounts[account_number],'Deposit',datetime.now(),
amount)
            self.transactions.append(transaction)
            print(f"Deposited an amount of {amount}")
            return self.accounts[account_number].balance
        return None

    def withdraw(self, account_number, amount):
        if account_number in self.accounts:
            account=self.accounts[account_number]

            if isinstance(account,SavingsAccount) and account.balance-
amount<500:
                print("Insufficient fund in your account")
```

```
            elif isinstance(account,CurrentAccount) and
account.balance+account.overdrafted_limit<amount:
                print("Insufficient fund in your account")
            else:
                account.balance-=amount

transaction=Transaction(account,'Withdraw',datetime.now(),amount)
                self.transactions.append(transaction)
                return account.balance
        return None

    def transfer(self, from_account_number, to_account_number, amount):
        from_balance = self.get_account_balance(from_account_number)
        if from_balance is not None and from_balance >= amount:
            self.withdraw(from_account_number, amount)
            to_balance = self.deposit(to_account_number, amount)

transaction=Transaction(self.accounts[from_account_number],'Tranfer- Sending
Amount',datetime.now(),amount)
            self.transactions.append(transaction)
            transaction1=Transaction(self.accounts[to_account_number],
'Tranfer- Receive Amount', datetime.now(),
                                     amount)
            self.transactions.append(transaction1)
            if to_balance is not None:
                return to_balance
        print("Error: Insufficient funds for transfer.")
        return None

    def get_account_details(self, account_number):
        if account_number in self.accounts:
            return self.accounts[account_number]
        else:
            return None

    def get_transactions(self, account_number, from_date, to_date):
        if account_number in self.accounts:
            account_transactions = [transaction for transaction in
self.transactions
                                    if transaction.account.account_number ==
account_number
```

```
                                        and from_date <= transaction.date_time <=
to_date]
            return account_transactions
        else:
            return None
```

8. Create **BankServiceProviderImpl** class which inherits from **CustomerServiceProviderImpl** and implements **IBankServiceProvider**.

• Attributes

o **accountList**: List of Accounts to store any account objects. o **transactionList**: List of Transaction to store transaction objects.

o **branchName** and **branchAddress** as String objects

```
class BankServiceProviderImpl(CustomerServiceProviderImpl,
IBankServiceProvider):

    def create_account(self, customer, acc_no, acc_type, balance):
        account = None
        if acc_type == "Savings":
            account = SavingsAccount(customer, balance)
        elif acc_type == "Current":
            account = CurrentAccount(customer, balance, overdrafted_limit=1000)
        elif acc_type == "ZeroBalance":
            account = ZeroBalanceAccount(customer)
        else:
            print("Invalid account type.")
        if account:
            self.accounts[acc_no] = account
            return account
    def list_accounts(self):
        return list(self.accounts.values())

    def get_account_details(self, account_number):
        return self.get_account_details(account_number)

    def calculate_interest(self):
        for account in self.accounts.values():
            if isinstance(account, SavingsAccount):
                interest = account.balance * account.interest_rate
                print(f"Account {account.account_number}: Interest calculated -
{interest}")
```

9. Create **IBankRepository** interface/abstract class which include following methods to interact with database.

• **createAccount**(customer: Customer, accNo: long, accType: String, balance: float):

Create a new bank account for the given customer with the initial balance and store in database.

• **listAccounts**(): List<Account> accountsList: List all accounts in the bank from database.

• **calculateInterest**(): the calculate_interest() method to calculate interest based on the balance and interest rate.

• **getAccountBalance**(account_number: long): Retrieve the balance of an account given its account number. should return the current balance of account from database.

• **deposit**(account_number: long, amount: float): Deposit the specified amount into the account. Should update new balance in database and return the new balance.

• **withdraw**(account_number: long, amount: float): Withdraw amount should check the balance from account in database and new balance should updated in Database.

o A savings account should maintain a minimum balance and checking if the withdrawal violates the minimum balance rule.

o Current account customers are allowed withdraw overdraftLimit and available account balance. withdraw limit can exceed the available balance and should not exceed the overdraft limit.

• **transfer**(from_account_number: long, to_account_number: int, amount: float):

Transfer money from one account to another. check the balance from account in

database and new balance should updated in Database.

• **getAccountDetails**(account_number: long): Should return the account and customer

details from databse.

• **getTransations**(account_number: long, FromDate:Date, ToDate: Date): Should return

the list of transaction between two dates from database.

```
class IBankRepository(ABC):


    @abstractmethod
    def createAccount(self,con,customer,account_number,account_type,balance):
        pass
    @abstractmethod
    def listAccount(self,con):
        pass
    @abstractmethod
    def calculateInterest(self, con, user_interest_rate):
        pass
    @abstractmethod
    def getAccountBalance(self,con,account_number):
        pass
    @abstractmethod
```

```
    def deposit(self,con,account_number,amount):
        pass
    @abstractmethod
    def withdraw(self,con,account_number,amount):
        pass
    @abstractmethod
    def transfer(self,con,from_account_number,to_account_number,amount):
        pass
    @abstractmethod
    def getAccountdetails(self,con,account_number):
        pass
    @abstractmethod
    def getTransactions(self,con,account_number,from_date,to_date):
        pass
```

10. Create **BankRepositoryImpl** class which implement the **IBankRepository** interface/abstract class and provide implementation of all methods and perform the database operations.

```
class BankRepositoryImpl(IBankRepository):
    def __init__(self):
       self.counter = 1000
       self.account_list = []
       self.counter2=15

    def generateAccountNumber(self,con):
       cur=con.cursor()
       cur.execute("SELECT MAX(account_id) from Accounts")
       max_account_id=cur.fetchone()[0]
       max_account_id+=1
       return str(max_account_id)
    def generateCustomerId(self,con):
       cur = con.cursor()
       cur.execute("SELECT MAX(customer_id) FROM Customers")
       max_customer_id = cur.fetchone()[0]
       max_customer_id += 1
    return str(max_customer_id)
    def listAccount(self, con):
        cur = con.cursor()
        cur.execute("SELECT * FROM Accounts")
        res = cur.fetchall()
        return res
```

```
    def createAccount(self, con, customer, account_number, account_type,
balance):
        account = None
        account_number = self.generateAccountNumber()

        if account_type == "Savings":
            account = SavingsAccount(customer, balance)
        elif account_type == "Current":
            account = CurrentAccount(customer, balance, overdrafted_limit=1000)
        elif account_type == "ZeroBalance":
            account = ZeroBalanceAccount(customer)
        else:
            print("Invalid account type.")

        if account:
            try:
                customer_id = customer[0]
                cur = con.cursor()
                cur.execute(
                    "INSERT INTO Accounts (account_id, customer_id,
account_type, balance) VALUES (%s, %s, %s, %s)",
                    (account_number, customer_id, account_type, balance))
                con.commit()
                print(f"{account_type} Account for Customer ID {customer_id}
created successfully...")
                self.account_list.append(account)
            except Exception as e:
                print(f"Error in createAccount: {e}")

    def createCustomer(self,con, customer_id, first_name,
last_name,DOB,email,phone_no,address ):
    customer_id=self.generateCustomerID()
    cur=con.cursor()
    cur.execute("INSERT INTO Customers VALUES (%s, %s, %s, %s, %s, %s, %s)",
                (customer_id,first_name,last_name,DOB,email,phone_no,address))
    con.commit()
    print(f"Welcome {first_name}")
    return Customer(customer_id, first_name, last_name, email, phone_no,
address)

    def calculateInterest(self, con, user_interest_rate):
        try:
            cur = con.cursor()
            cur.execute("SELECT account_id, balance, account_type FROM Accounts
WHERE account_type = 'Savings'")
            savings_accounts = cur.fetchall()

            for account_id, balance, account_type in savings_accounts:
                if account_type == 'Savings':
```

```
                    account = SavingsAccount(None)
                    account.account_number = account_id
                    account.balance =float(balance)
                    interest_rate = float(user_interest_rate)
                    interest = account.balance * interest_rate
                    new_balance = account.balance + interest

                    cur.execute("UPDATE Accounts SET balance = %s WHERE
account_id = %s", (new_balance, account_id))
                    con.commit()
                    print(f"Account {account_id}: Interest calculated -
{interest}. New balance - {new_balance}")
        except Exception as e:
            print(f"Error in calculateInterest: {e}")

    def getAccountBalance(self, con, account_number):
        cur = con.cursor()
        cur.execute("SELECT balance FROM Accounts WHERE account_id = %s",
(account_number,))
        res = cur.fetchone()
        return res

    def deposit(self, con, account_number, amount):
        try:

            cur=con.cursor()
            transaction_type='Deposit'
            cur.execute("INSERT INTO Transactions
(account_id,transaction_type,amount,transaction_date) VALUES (%s, %s, %s,
%s)",(account_number,transaction_type,amount,datetime.now()))
            con.commit()
            print(f"Deposit of amount {amount} done successfully...")

            cur2=con.cursor()
            cur2.execute("UPDATE Accounts SET balance = balance + %s WHERE
account_id = %s", (amount, account_number))
            con.commit()

        except Exception as e:
            con.rollback()
            print(e)
```

```
    def withdraw(self, con, account_number, amount):
        try:
            cur=con.cursor()
            transaction_type='Withdrawal'
            cur.execute(
                "INSERT INTO Transactions (account_id, transaction_type,
amount, transaction_date) VALUES (%s, %s, %s, %s)",
                (account_number, transaction_type, amount, datetime.now()))
            print(f"Withdrawal of amount {amount} from account {account_number}
done successfully")
            con.commit()
            cur2=con.cursor()
            cur2.execute("UPDATE Accounts SET balance = balance - %s WHERE
account_id = %s",(amount,account_number))
            con.commit()
        except Exception as e:
            print(e)

    def transfer(self, con, from_account_number, to_account_number, amount):
        try:
            cur = con.cursor()
            cur.execute("SELECT balance FROM Accounts WHERE account_id = %s",
(from_account_number,))
            from_balance = cur.fetchone()
            if from_balance[0] > amount:
                with con.cursor() as cur:
                    # Deduct amount from the sender's account
                    cur.execute("UPDATE Accounts SET balance = balance - %s
WHERE account_id = %s",
                                (amount, from_account_number))

                    # Add amount to the receiver's account
                    cur.execute("UPDATE Accounts SET balance = balance + %s
WHERE account_id = %s",
                                (amount, to_account_number))
                    print(f"Amount Transfered Successfully from Account
{from_account_number} to {to_account_number}")
                con.commit()
        except Exception as e:
            print(e)
```

```
    def getAccountdetails(self, con, account_number):
        cur=con.cursor()
        cur.execute("SELECT * FROM Accounts WHERE account_id =
%s",(account_number,))
        details=cur.fetchone()
        print("=" * 20)
        print("Account Details: ")
        print("="*20)
        print(f"Account ID: {details[0]}")
        print(f"Customer ID: {details[1]}")
        print(f"Account Type: {details[2]}")
        print(f"Balance: {details[3]}")


    def getTransactions(self, con, account_number, from_date, to_date):
        try:
            with con.cursor() as cur:
                cur.execute("SELECT * FROM Transactions WHERE account_id = %s
AND transaction_date BETWEEN %s AND %s",
                            (account_number, from_date, to_date))
                res = cur.fetchall()
            print(f"Transaction Details for {account_number}")
            for transaction in res:
                transaction_id, account_id, transaction_type, amount,
transaction_date= transaction
                print(f"Transaction ID :{transaction_id}, Account ID
:{account_id}, Type : {transaction_type}, Date: {transaction_date}")
        except Exception as e:
            print(f"Error: {e}")
```

11. Create DBUtil class and add the following method.

• static getDBConn():Connection Establish a connection to the database and return

Connection reference

```
class DBUtil:
    def __init__(self):
        self.con = mysql.connector.connect(host='localhost', user='root',
passwd='root', database='HMBank', port='3306')
        if self.con:
            print("Connection successful")
```

```
    def __del__(self):
        if hasattr(self, 'con') and self.con.is_connected():
            self.con.close()
            print("Connection closed")
```

12. Create BankApp class and perform following operation:

• main method to simulate the banking system. Allow the user to interact with the system by entering choice from menu such as "create_account", "deposit", "withdraw", "get_balance", "transfer", "getAccountDetails", "ListAccounts", "getTransactions" and "exit."

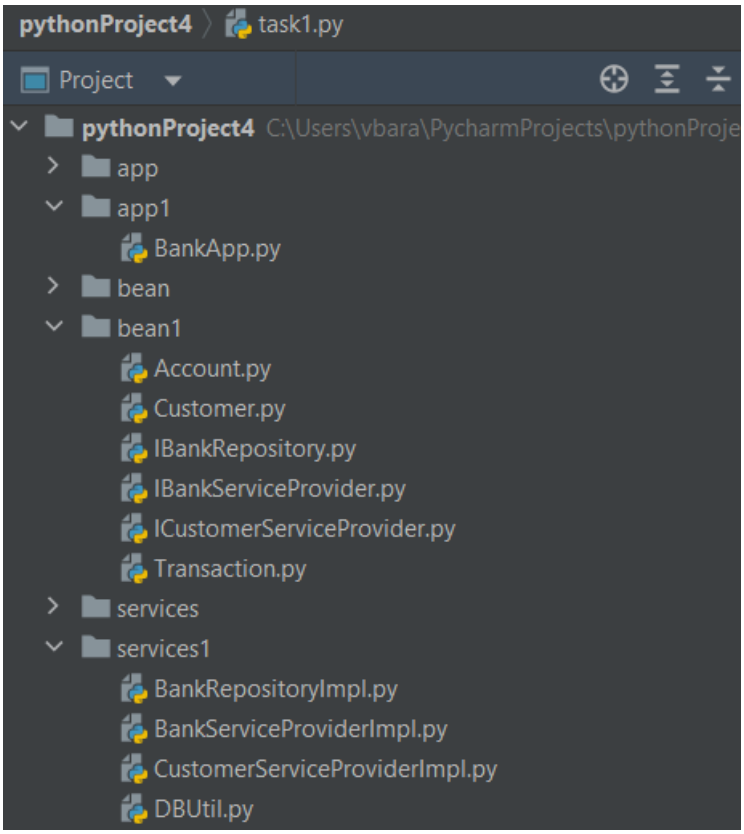
• create_account should display sub menu to choose type of accounts and repeat this operation until user exit.

```
class BankApp():
    def main(self):
        obj = DBUtil()
        bank = BankRepositoryImpl()
        print("Banking System Menu: ")

        print('''
                 1------>Create Account
                 2------>Deposit
                 3------>Withdraw
                 4------>Get Balance
                 5------>Transfer
                 6------>Get Account Details
                 7------>List Accounts
                 8------>Get Transactions
                 9------>Exit
            ''')
        while True:
            choice=int(input("Enter your choice: "))
            if choice == 1:
                print("Enter the customer details: ")
                first_name = input("Enter your first name: ")
                last_name = input("Enter your last name: ")
                dob = input("Enter your Date of Birth: ")
                email = input("Enter your email")
                phone_no = int(input("Enter the phone number: "))
```

```
                address = input("Enter your address: ")

                customer = bank.createCustomer(obj.con, first_name, last_name,
dob, email, phone_no, address)

                account_type = input("Enter the type of account you are willing
to create: ")
                balance = int(input("Enter your initial balance: "))
                bank.createAccount(obj.con, customer, account_type, balance)

            if choice == 2:
                account_number=int(input("Enter your account number: "))
                amount=int(input("Enter the amount: "))
                bank.deposit(obj.con,account_number,amount)

            if choice == 3:
                account_number = int(input("Enter your account number: "))
                amount = int(input("Enter the amount: "))
                bank.withdraw(obj.con,account_number,amount)

            if choice == 4:
                account_number=int(input("Enter your account number: "))
                balance=bank.getAccountBalance(obj.con,account_number)
                print("Your Available balance: ",float(balance[0]))

            if choice == 5:
                from_account=int(input("Enter the from account number: "))
                to_account = int(input("Enter the from to number: "))
                amount=int(input("Enter the amount: "))
                bank.transfer(obj.con,from_account,to_account,amount)

            if choice == 6:
                account_number=int(input("Enter the account number: "))
                bank.getAccountdetails(obj.con,account_number)

            if choice == 7:
                account_list=bank.listAccount(obj.con)
                for account in account_list:
                    print(" ",account)

            if choice == 8:
```

```
                account_number=int(input("Enter the account number: "))
                from_date=input("Enter the from date: ")
                to_date = input("Enter the to date: ")
                bank.getTransactions(obj.con,account_number,from_date,to_date)

            if choice == 9:
                print("Thank you for banking with us.....")
                break;



b=BankApp()
b.main()
```

13. Place the interface/abstract class in service package and interface/abstract class implementation class, account class in bean package and Bank class in app package.



14. Should throw appropriate exception as mentioned in above task along with handle SQLException

Implementation:

To create an account for the customer Keerthivasan K of the Savings type with an initial balance of 10000, the system prompts the user to enter the customer details. After entering the details, the system allows the user to create an account of the specified type with its initial balance.

```
Run: BankApp (1)
"C:\Users\vbara\PycharmProjects\demo\Assignment 3\Scripts\python.exe" C:/Users/vbara
Connection successful
Banking System Menu:

                    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: 1
Enter the customer details:
Enter your first name: Keerthivasan
Enter your last name: K
Enter your Date of Birth: 2001-07-07
Enter your emailkeerthi@gmail.com
Enter the phone number: 763554279
Enter your address: Karaikal
Welcome Keerthivasan
Enter the type of account you are willing to create: Savings
Enter your initial balance: 10000
Savings Account for Customer ID 26 created successfully with Account Number 116...
```

The customer details have been recorded in the database:

Result Grid | Filter Rows: | Edit: | Export/Import: | Wrap Cell Content:

customer_id	first_name	last_name	DOB	email	phone_number	address
17	Gokul	G	2002-02-02	gokul@gmail.com	3287832	Thindi
18	Chandru	M	2000-09-13	chandru@gmail.com	32783822	Goa
19	Mukesh	K	2002-09-02	mukesh@gmail.com	97662536	Ladakh
20	Mouli	J	2000-12-01	Mouli@gmail.com	878367252	Delhi
21	Suresh	M	2003-09-08	sur@gmail.com	2328671	Pondy
22	Karan	K	2000-09-07	karan@gmail.com	86356211	Erode
23	Rubavathi	P	1998-09-24	ruba@gmail.com	7094382646	Alankuppam
24	Sathish	P	2002-09-08	sathish@gmail.com	987635245	Pondicherry
25	Abina	T	2001-06-22	abinatovari@gmail.com	8765342677	Kerala
26	Keerthiva...	K	2001-07-07	keerthi@gmail.com	763554279	Karaikal
NULL	NULL	NULL	NULL	NULL	NULL	NULL

The account for the customer Keerthivasan K (Customer ID = 26) of type Savings with an initial balance of 10000 has been created.

Result Grid | Filter Rows: | Edit:

account_id	customer_id	account_type	balance
107	7	savings	8000.00
108	8	current	11000.00
109	9	savings	9500.00
110	10	current	7000.00
111	5	curent	4000.00
112	1	Savings	1291467.97
113	1	Savings	1291467.97
114	1	Savings	1291467.97
115	25	Savings	14000.00
116	26	Savings	10000.00
NULL	NULL	NULL	NULL

Depositing an amount of 1500 to the account number 116 has been completed:

```
Enter your choice: 2
Enter your account number: 116
Enter the amount: 1500
Deposit of amount 1500 done successfully...
```

Upon depositing an amount of 1500 into account number 116, the initial balance has been updated from 10000 to 11500:

account_id	customer_id	account_type	balance
107	7	savings	8000.00
108	8	current	11000.00
109	9	savings	9500.00
110	10	current	7000.00
111	5	curent	4000.00
112	1	Savings	1291467.97
113	1	Savings	1291467.97
114	1	Savings	1291467.97
115	25	Savings	14000.00
116	26	Savings	11500.00
NULL	NULL	NULL	NULL

The deposit of 1500 to account number 116 has been recorded in the database with the current date:

transaction_id	account_id	transaction_type	amount	transaction_date
1126	101	Withdrawal	2000.00	2024-02-02
1127	101	Withdrawal	2000.00	2024-02-02
1128	101	Withdrawal	2000.00	2024-02-02
1129	101	Withdrawal	2000.00	2024-02-02
1130	101	Withdrawal	2000.00	2024-02-02
1131	101	Deposit	1000.00	2024-02-03
1132	102	Withdrawal	300.00	2024-02-03
1135	115	Deposit	5000.00	2024-02-03
1136	115	Withdrawal	1000.00	2024-02-03
1137	116	Deposit	1500.00	2024-02-03
NULL	NULL	NULL	NULL	NULL

Withdrawing an amount of 4000 from account number 116 has been initiated:

```
Enter your choice: 3
Enter your account number: 116
Enter the amount: 4000
Withdrawal of amount 4000 from account 116 done successfully
```

As an amount of 4000 is being withdrawn from account number 116, the withdrawn amount has been deducted from its account, and the initial balance is updated accordingly:

account_id	customer_id	account_type	balance
107	7	savings	8000.00
108	8	current	11000.00
109	9	savings	9500.00
110	10	current	7000.00
111	5	curent	4000.00
112	1	Savings	1291467.97
113	1	Savings	1291467.97
114	1	Savings	1291467.97
115	25	Savings	14000.00
116	26	Savings	7500.00
NULL	NULL	NULL	NULL

The withdrawal of 4000 from account number 116 has been recorded in the database with the current date:

transaction_id	account_id	transaction_type	amount	transaction_date
1127	101	Withdrawal	2000.00	2024-02-02
1128	101	Withdrawal	2000.00	2024-02-02
1129	101	Withdrawal	2000.00	2024-02-02
1130	101	Withdrawal	2000.00	2024-02-02
1131	101	Deposit	1000.00	2024-02-03
1132	102	Withdrawal	300.00	2024-02-03
1135	115	Deposit	5000.00	2024-02-03
1136	115	Withdrawal	1000.00	2024-02-03
1137	116	Deposit	1500.00	2024-02-03
1138	116	Withdrawal	4000.00	2024-02-03
NULL	NULL	NULL	NULL	NULL

Fetching the balance for account number 116:

```
Enter your choice: 4
Enter your account number: 116
Your Available balance:  7500.0
```

account_id	customer_id	account_type	balance
107	7	savings	8000.00
108	8	current	11000.00
109	9	savings	9500.00
110	10	current	7000.00
111	5	curent	4000.00
112	1	Savings	1291467.97
113	1	Savings	1291467.97
114	1	Savings	1291467.97
115	25	Savings	14000.00
116	26	Savings	7500.00
NULL	NULL	NULL	NULL

Initiating a transfer of 2000 from account number 116 to account number 115:

```
Enter your choice: 5
Enter the from account number: 116
Enter the from to number: 115
Enter the amount: 2000
Amount Transfered Successfully from Account 116 to 115
```

The amount is being transferred from account number 116, and the corresponding amount is deducted from its balance:

Before transfer:

114	1	Savings	1291467.97
115	25	Savings	14000.00
116	26	Savings	7500.00
NULL	NULL	NULL	NULL

After transfer:

account_id	customer_id	account_type	balance
107	7	savings	8000.00
108	8	current	11000.00
109	9	savings	9500.00
110	10	current	7000.00
111	5	curent	4000.00
112	1	Savings	1291467.97
113	1	Savings	1291467.97
114	1	Savings	1291467.97
115	25	Savings	16000.00
116	26	Savings	5500.00
NULL	NULL	NULL	NULL

The amount is being transferred to account number 115, and the corresponding amount is added to its balance:

Before transfer:

114	1	Savings	1291467.97
115	25	Savings	14000.00

After transfer:

Result Grid | Filter Rows: | Edit:

account_id	customer_id	account_type	balance
107	7	savings	8000.00
108	8	current	11000.00
109	9	savings	9500.00
110	10	current	7000.00
111	5	curent	4000.00
112	1	Savings	1291467.97
113	1	Savings	1291467.97
114	1	Savings	1291467.97
115	25	Savings	16000.00
116	26	Savings	5500.00
NULL	NULL	NULL	NULL

Getting account details for the account number 115:

```
Enter your choice: 6
Enter the account number: 115
====================
Account Details:
====================
Account ID: 115
Customer ID: 25
Account Type: Savings
Balance: 16000.00
```

Listing all the accounts from the database:

```
Enter your choice: 7
  (101, 1, 'savings', Decimal('-14000.00'))
  (102, 2, 'current', Decimal('11700.00'))
  (103, 3, 'savings', Decimal('3500.00'))
  (104, 4, 'current', Decimal('12000.00'))
  (105, 5, 'savings', Decimal('14999.00'))
  (106, 6, 'current', Decimal('6000.00'))
  (107, 7, 'savings', Decimal('8000.00'))
  (108, 8, 'current', Decimal('11000.00'))
  (109, 9, 'savings', Decimal('9500.00'))
  (110, 10, 'current', Decimal('7000.00'))
  (111, 5, 'curent', Decimal('4000.00'))
  (112, 1, 'Savings', Decimal('1291467.97'))
  (113, 1, 'Savings', Decimal('1291467.97'))
  (114, 1, 'Savings', Decimal('1291467.97'))
  (115, 25, 'Savings', Decimal('16000.00'))
  (116, 26, 'Savings', Decimal('5500.00'))
```

Result Grid | Filter Rows: | Edit:

account_id	customer_id	account_type	balance
101	1	savings	-14000.00
102	2	current	11700.00
103	3	savings	3500.00
104	4	current	12000.00
105	5	savings	14999.00
106	6	current	6000.00
107	7	savings	8000.00
108	8	current	11000.00
109	9	savings	9500.00
110	10	current	7000.00
111	5	curent	4000.00
112	1	Savings	1291467.97
113	1	Savings	1291467.97
114	1	Savings	1291467.97
115	25	Savings	16000.00
116	26	Savings	5500.00
NULL	NULL	NULL	NULL

Fetching the transaction details for account 115 from the date 2024-02-02 to 2024-02-03:

```
Enter your choice: 8
Enter the account number: 115
Enter the from date: 2024-02-02
Enter the to date: 2024-02-03
Transaction Details for 115
Transaction ID :1135, Account ID :115, Type : Deposit, Date: 2024-02-03
Transaction ID :1136, Account ID :115, Type : Withdrawal, Date: 2024-02-03
```

Result Grid | Filter Rows: | Edit: | Export/Import:

transaction_id	account_id	transaction_type	amount	transaction_date
1120	101	Withdrawal	2000.00	2024-02-02
1121	101	Withdrawal	2000.00	2024-02-02
1122	101	Withdrawal	2000.00	2024-02-02
1123	101	Withdrawal	2000.00	2024-02-02
1124	101	Withdrawal	2000.00	2024-02-02
1125	101	Withdrawal	2000.00	2024-02-02
1126	101	Withdrawal	2000.00	2024-02-02
1127	101	Withdrawal	2000.00	2024-02-02
1128	101	Withdrawal	2000.00	2024-02-02
1129	101	Withdrawal	2000.00	2024-02-02
1130	101	Withdrawal	2000.00	2024-02-02
1131	101	Deposit	1000.00	2024-02-03
1132	102	Withdrawal	300.00	2024-02-03
1135	115	Deposit	5000.00	2024-02-03
1136	115	Withdrawal	1000.00	2024-02-03
1137	116	Deposit	1500.00	2024-02-03
1138	116	Withdrawal	4000.00	2024-02-03
NULL	NULL	NULL	NULL	NULL

Choosing option 9 to exit the loop and complete the banking process.

```
Enter your choice: 9
Thank you for banking with us.....
Connection closed

Process finished with exit code 0
```