

ASSIGNMENT -03

SQL & OOPS BANKING SYSTEM

(OOPS)

- MOHAMMED IBRAHIM SHERIFF U

BATCH 4

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.

Code:

```
def check_loan_eligibility(credit_score, annual_income):  
    if credit_score > 700 and annual_income >= 50000:  
        print(" Eligible for Loan")  
    else:  
        print(" Not eligible for Loan")  
  
credit_score = int(input("Enter credit score: "))  
annual_income = float(input("Enter annual income: "))  
check_loan_eligibility(credit_score, annual_income)
```

```
3 def check_loan_eligibility(credit_score, annual_income):
4     if credit_score > 700 and annual_income >= 50000:
5         print(" Eligible for Loan")
6     else:
7         print(" Not eligible for Loan")
8
9 # Example
10 credit_score = int(input("Enter credit score: "))
11 annual_income = float(input("Enter annual income: "))
12
13 check_loan_eligibility(credit_score, annual_income)
14
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

~~~~~

KeyboardInterrupt  
PS E:\banking\_system> & C:/Users/Lenovo/AppData/Local/Programs/Python/Python312/python.exe e:/banking\_system/task1.py  
Enter credit score: 750  
Enter annual income: 200000  
Eligible for Loan  
PS E:\banking\_system> |

**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.

**Code:**

```
def atm_sim():
    balance = float(input("Enter current balance: "))
    print("\noption 1: check balance\noption 2: withdrawal\noption 3: deposit")
    choice = int(input("Enter 1 / 2 / 3 : "))
    if (choice == 1):
        print("your current balance is : ", balance)
    elif (choice == 2):
        withdrawal = int(input("Enter amount to withdraw: "))
        if (withdrawal > balance):
            print("Insufficient funds")
        else:
            balance = balance - withdrawal
            print(f"amount of {withdrawal} has been withdrawn, new balance = {balance}")
    elif (choice == 3):
        deposit = int(input("Enter the amount you want to deposit: "))
```

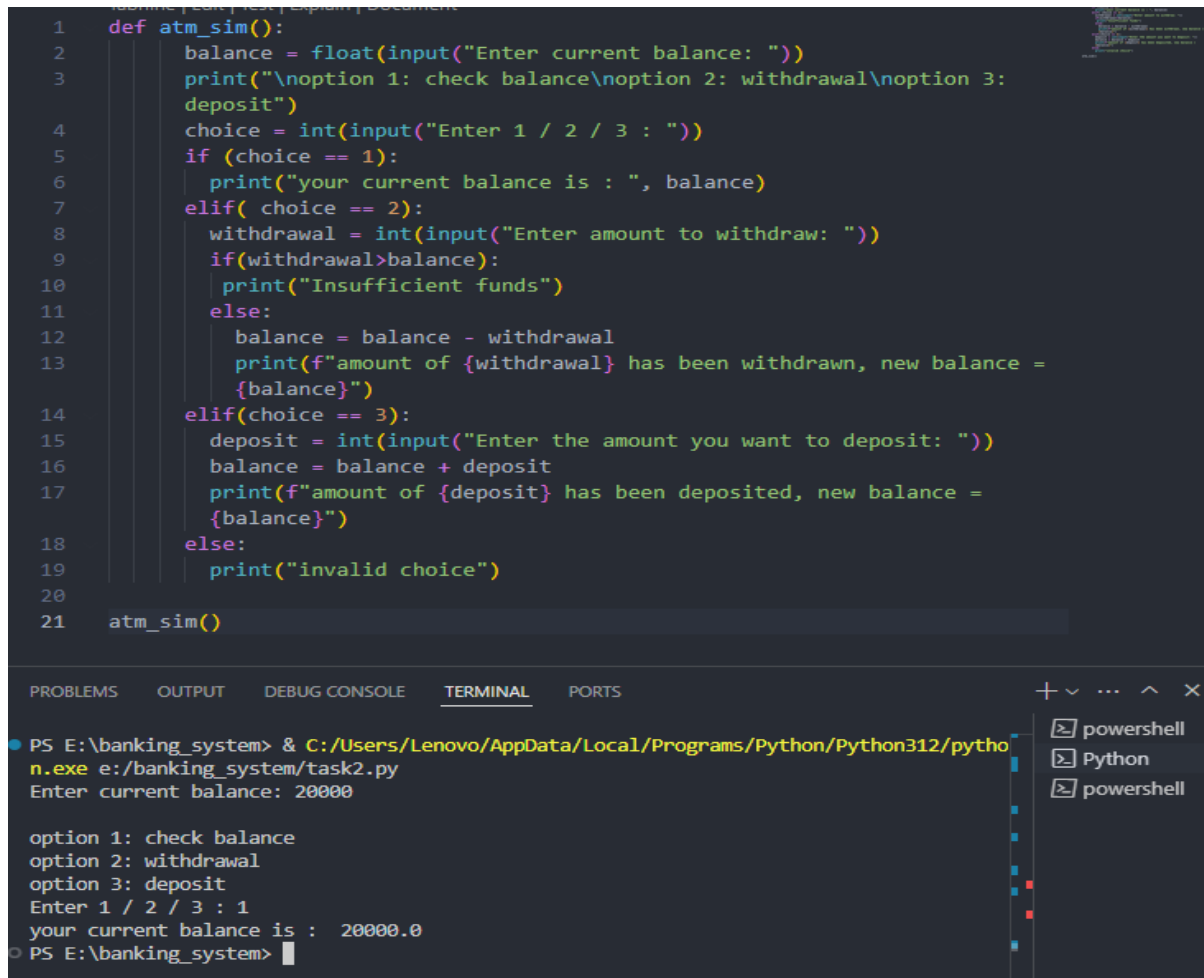
```
balance = balance + deposit
```

```
print(f'amount of {deposit} has been deposited, new balance = {balance}')
```

```
else:
```

```
print("invalid choice")
```

```
atm_sim()
```



```
1 def atm_sim():
2     balance = float(input("Enter current balance: "))
3     print("\noption 1: check balance\noption 2: withdrawal\noption 3:
4     deposit")
5     choice = int(input("Enter 1 / 2 / 3 : "))
6     if (choice == 1):
7         print("your current balance is : ", balance)
8     elif (choice == 2):
9         withdrawal = int(input("Enter amount to withdraw: "))
10        if (withdrawal > balance):
11            print("Insufficient funds")
12        else:
13            balance = balance - withdrawal
14            print(f"amount of {withdrawal} has been withdrawn, new balance =
15            {balance}")
16        elif (choice == 3):
17            deposit = int(input("Enter the amount you want to deposit: "))
18            balance = balance + deposit
19            print(f"amount of {deposit} has been deposited, new balance =
20            {balance}")
21        else:
22            print("invalid choice")
23
24    atm_sim()
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

PS E:\banking\_system> & C:/Users/Lenovo/AppData/Local/Programs/Python/Python312/pytho  
n.exe e:/banking\_system/task2.py  
Enter current balance: 20000  
  
option 1: check balance  
option 2: withdrawal  
option 3: deposit  
Enter 1 / 2 / 3 : 1  
your current balance is : 20000.0  
PS E:\banking\_system>

**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:  $\text{future\_balance} = \text{initial\_balance} * (1 + \text{annual\_interest\_rate}/100)^{\text{years}}$ . 5. Display the future balance for each customer.

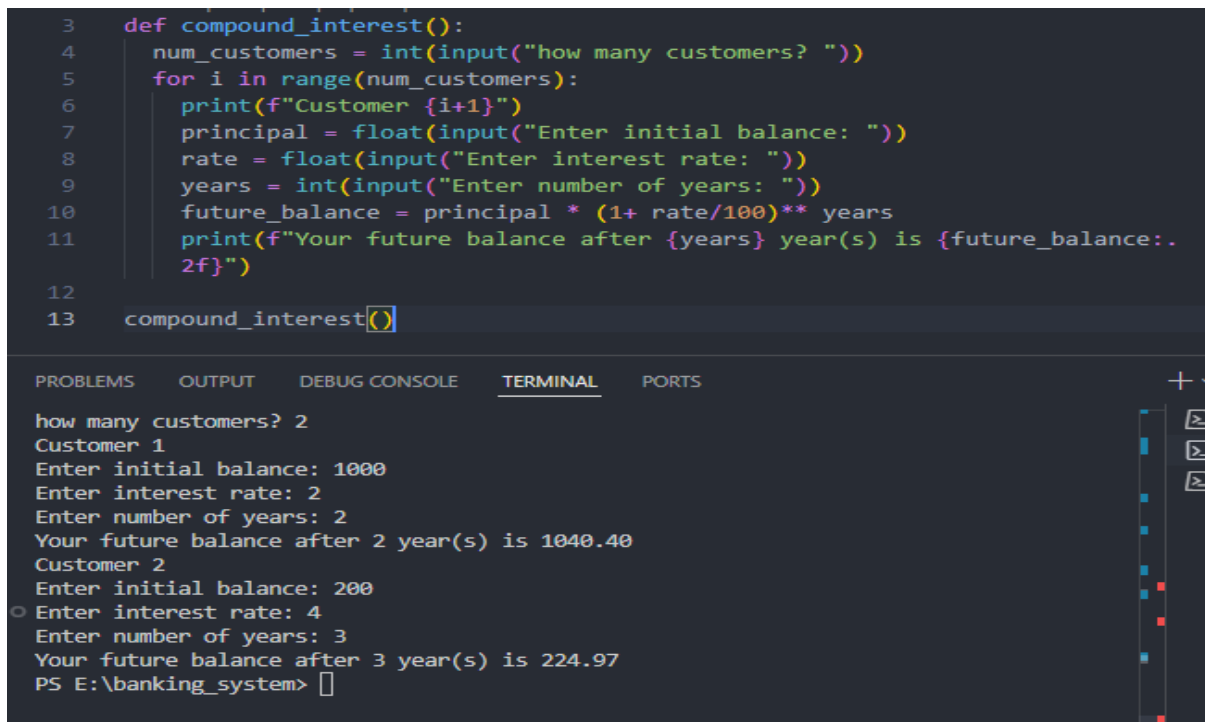
**Code:**

```
def compound_interest():
```

```

num_customers = int(input("how many customers? "))
for i in range(num_customers):
    print(f"Customer {i+1}")
    principal = float(input("Enter initial balance: "))
    rate = float(input("Enter interest rate: "))
    years = int(input("Enter number of years: "))
    future_balance = principal * (1 + rate/100)** years
    print(f"Your future balance after {years} year(s) is {future_balance:.2f}")
compound_interest()

```



The screenshot shows a Python IDE with a dark theme. The top pane displays the `compound_interest()` function definition, which is identical to the code block above. The bottom pane shows the `TERMINAL` output, where the function is called. The terminal shows the following interaction:

```

how many customers? 2
Customer 1
Enter initial balance: 1000
Enter interest rate: 2
Enter number of years: 2
Your future balance after 2 year(s) is 1040.40
Customer 2
Enter initial balance: 200
Enter interest rate: 4
Enter number of years: 3
Your future balance after 3 year(s) is 224.97
PS E:\banking_system>

```

**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.

**Code:**

```

accounts= {
    101:5000.00, 102:90000.00, 103:400.00
}

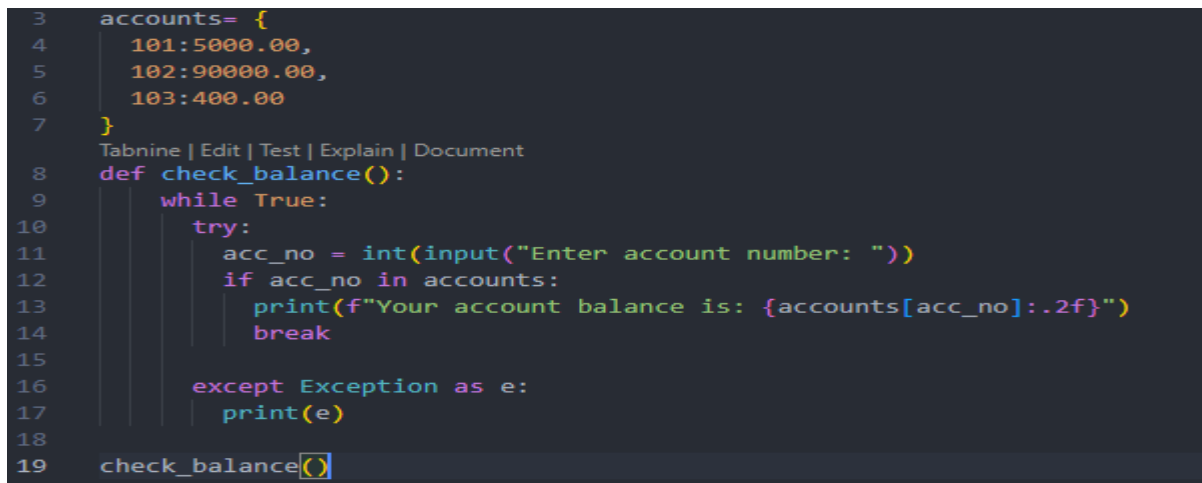
```

```

}

def check_balance():
    while True:
        try:
            acc_no = int(input("Enter account number: "))
            if acc_no in accounts:
                print(f"Your account balance is: {accounts[acc_no]:.2f}")
                break
        except Exception as e:
            print(e)
    check_balance()

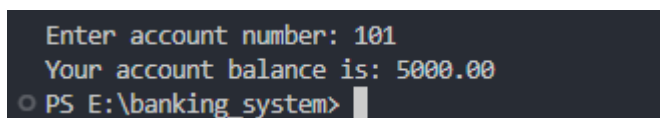
```



```

3     accounts= {
4         101:5000.00,
5         102:90000.00,
6         103:400.00
7     }
8     def check_balance():
9         while True:
10            try:
11                acc_no = int(input("Enter account number: "))
12                if acc_no in accounts:
13                    print(f"Your account balance is: {accounts[acc_no]:.2f}")
14                    break
15            except Exception as e:
16                print(e)
17
18    check_balance()

```



```

Enter account number: 101
Your account balance is: 5000.00
PS E:\banking_system>

```

**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.

**Code:**

```

import re

accounts= {
    101:5000.00, 102:90000.00, 103:400.00

```

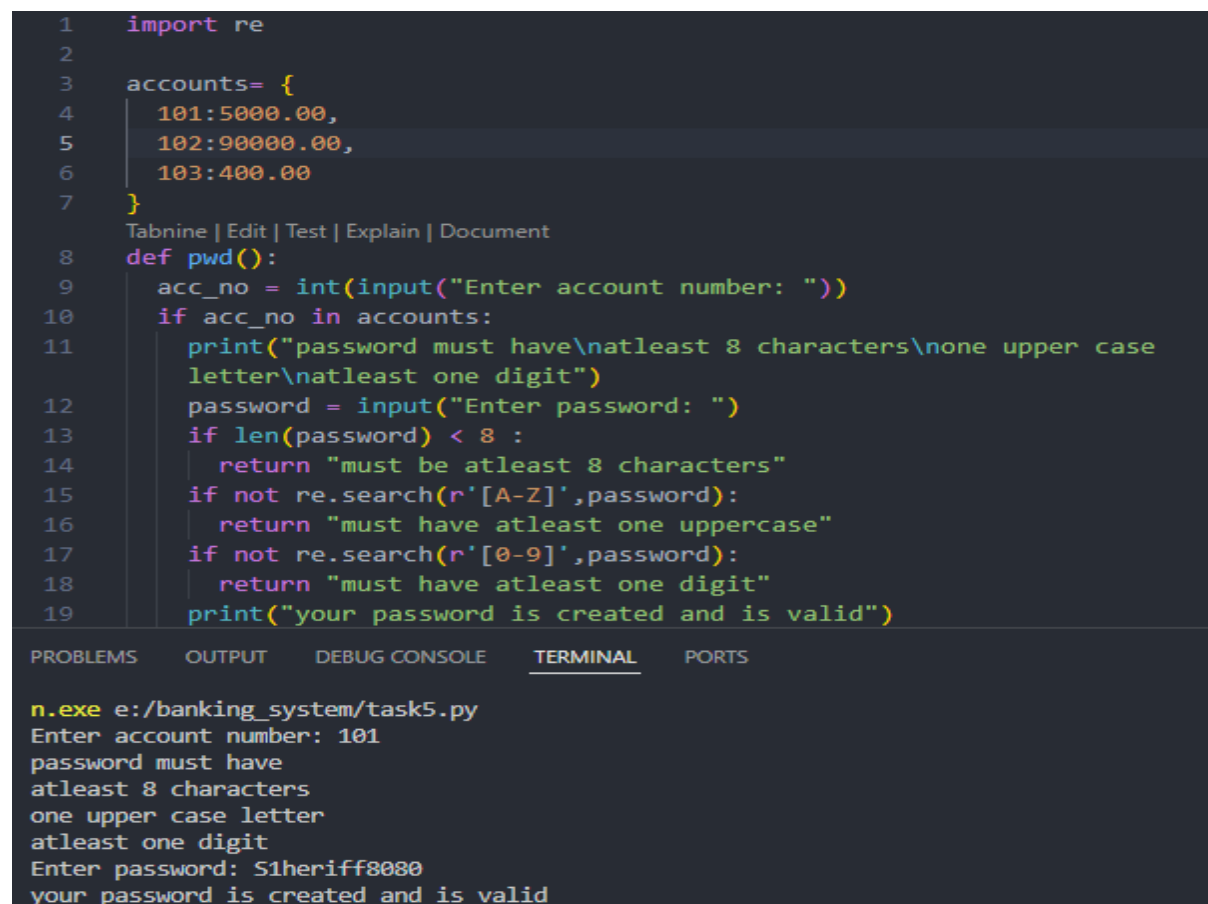
```

}

def pwd():
    acc_no = int(input("Enter account number: "))
    if acc_no in accounts:
        print("password must have\natleast 8 characters\nnone upper case letter\natleast one digit")
        password = input("Enter password: ")
        if len(password) < 8 :
            return "must be atleast 8 characters"
        if not re.search(r'[A-Z]',password):
            return "must have atleast one uppercase"
        if not re.search(r'[0-9]',password):
            return "must have atleast one digit"
        print("your password is created and is valid")

pwd()

```



```

1  import re
2
3  accounts= {
4      101:5000.00,
5      102:90000.00,
6      103:400.00
7  }
8
9  def pwd():
10     acc_no = int(input("Enter account number: "))
11     if acc_no in accounts:
12         print("password must have\natleast 8 characters\nnone upper case
13         letter\natleast one digit")
14         password = input("Enter password: ")
15         if len(password) < 8 :
16             return "must be atleast 8 characters"
17         if not re.search(r'[A-Z]',password):
18             return "must have atleast one uppercase"
19         if not re.search(r'[0-9]',password):
20             return "must have atleast one digit"
21         print("your password is created and is valid")

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```

n.exe e:/banking_system/task5.py
Enter account number: 101
password must have
atleast 8 characters
one upper case letter
atleast one digit
Enter password: S1heriff8080
your password is created and is valid

```

**Task 6: Password Validation** 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.

**Code:**

```
transaction = []

def transaction_history():

    balance = float(input("Enter balance amount: "))

    while True:

        print("1. Deposit\n2. Withdraw\n3. Exit")

        option = int(input("Choose option: "))

        if option == 1:

            deposit = int(input("Enter the amount to deposit: "))

            balance = balance + deposit

            print(f"Amount of {deposit} has been deposited and your new balance is {balance}")

            transaction.append(f"Deposited amount {deposit:.2f}")

        elif option == 2:

            withdraw = int(input("Enter the amount to withdraw : "))

            if balance > withdraw:

                balance = balance - withdraw

                print(f"An amount of {withdraw} has been withdrawn and your new balance is {balance}")

                transaction.append(f"Withdrawn amount {withdraw:.2f}")

            else:

                print("Insufficient funds")

        elif option == 3:

            break

    print("Transaction Details ")

    for i in transaction:

        print("_",i)

transaction_history()
```

```
30 transaction = []
31
32 Tabnine | Edit | Test | Explain | Document
33 def transaction_history():
34     balance = float(input("Enter balance amount: "))
35     while True:
36         print("1. Deposit\n2. Withdraw\n3. Exit")
37         option = int(input("Choose option: "))
38         if option == 1:
39             deposit = int(input("Enter the amount to deposit: "))
40             balance = balance + deposit
41             print(f"Amount of {deposit} has been deposited and your new balance is {balance}")
42             transaction.append(f"Deposited amount {deposit:.2f}")
43         elif option == 2:
44             withdraw = int(input("Enter the amount to withdraw : "))
45             if balance > withdraw:
46                 balance = balance - withdraw
47                 print(f"An amount of {withdraw} has been withdrawn and your new balance is {balance}")
48
49 PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
n.exe e:/banking_system/task6.py
Enter balance amount: 1000
1. Deposit
2. Withdraw
3. Exit
Choose option: 1
Enter the amount to deposit: 200
Amount of 200 has been deposited and your new balance is 1200.0
```

## OOPS, Collections and Exception Handling

### 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 account

Code:



### *#customer class*

class Customer:

def

\_\_init\_\_(self,customer\_id=0,first\_name="",last\_name="",email="",phone="",address=""):

self.customer\_id = customer\_id

self.first\_name = first\_name

self.last\_name = last\_name

self.email = email

self.phone = phone

self.address = address

@property

def email(self):

return self.\_email

@email.setter

def email(self,email):

if "@" in email:

self.\_email = email

else:

raise ValueError("Invalid email format")

@property

def phone(self):

return self.\_phone

@phone.setter

def phone(self,phone):

if phone.isdigit() and len(phone) == 10:

self.\_phone = phone

else:

raise ValueError("Phone number must be of 10 digits")

def display\_customer\_info(self):

print(f'Customer id: {self.customer\_id}')

print(f'Name : {self.first\_name} {self.last\_name}')

print(f'Email: {self.email}')

print(f'Phone: {self.phone}')

```
print(f'Address: {self.address}')
```

### *#account class*

```
class Accounts:
```

```
    def __init__(self,account_num=0,account_type="",account_balance=0):
```

```
        self.account_num = account_num
```

```
        self.account_type = account_type
```

```
        self.account_balance = account_balance
```

```
    #get
```

```
    def get_account_num(self): return self.account_num
```

```
    def get_account_type(self): return self.account_type
```

```
    def get_account_balance(self): return self.account_balance
```

```
    #set
```

```
    def set_account_num(self,account_num): self.account_num = account_num
```

```
    def set_account_type(self,account_type): self.account_type = account_type
```

```
    def set_account_balance(self,account_balance): self.account_balance = account_balance
```

```
    #method
```

```
    def display_account_info(self):
```

```
        print(f'Account number is : {self.account_num}')
```

```
        print(f'Account type is : {self.account_type}')
```

```
        print(f'Account balance is : {self.account_balance:.2f}')
```

```
    def deposit(self,amount :float):
```

```
        self.account_balance+=amount
```

```
        print(f'The amount of {amount:.2f} has been deposited and the new balance is  
{self.account_balance:.2f}')
```

```
    def withdraw(self,amount:float):
```

```
        if self.account_balance > amount:
```

```
            self.account_balance = self.account_balance - amount
```

```
            print(f'The amount of {amount:.2f} has been withdrawn and your new balance is  
{self.account_balance:.2f}')
```

```
        else:
```

```
            print("Insufficient funds!")
```

```
    def calculate_interest(self):
```

```
rate = 4.5

if(self.account_balance > 0):

    interest = self.account_balance * rate

    print(f"The interest for your balance of {self.account_balance} is {interest}")
```

### **#bank class**

```
class Bank:
```

```
    def __init__(self):

        self.customer = None

        self.account = None

    def create_customer_accounts(self):

        print("---Customer Registration---")

        customer_id = int(input("Enter customer id: "))

        first_name = input("Enter first name : ")

        last_name = input("Enter last name: ")

        email = input("Enter email: ")

        phone = input("Enter phone number : ")

        address = input("Enter the address : ")

        try:

            self.customer = Customer(customer_id,first_name,last_name,email,phone,address)

        except Exception as e:

            print(e)

        print("---Account creation---")

        account_num = int(input("Enter account number : "))

        account_type = input("Enter account type : ")

        account_balance = int(input("Enter available balance : "))

        try:

            self.account = Accounts(account_num,account_type,account_balance)

        except Exception as E:

            print(E)

    def show_details(self):

        print("Customer Details : \n")

        self.customer.display_customer_info()
```

```

self.account.display_account_info()

def perform_transactions(self):
    while True:
        choice = int(input("Enter choice :\n1.show details\n2.deposit\n3.withdraw\n4.Calculate
Interest\n5.exit \nOption : "))

        if choice == 1:
            self.show_details()

        elif choice == 2:
            amount = float(input("Enter the amount you want to deposit : "))
            self.account.deposit(amount)

        elif choice == 3:
            amount = float(input("Enter the amount you want to withdraw : "))
            self.account.withdraw(amount)

        elif choice == 4:
            self.account.calculate_interest()

        elif choice == 5:
            print("Thank you!")
            break

        else:
            print("invalid choice")

```

```

class Customer:
    Tabnine | Edit | Test | Explain | Document
    def __init__(self, customer_id=0, first_name="", last_name="", email="", phone="", address=""):
        self.customer_id = customer_id
        self.first_name = first_name
        self.last_name = last_name
        self.email = email
        self.phone = phone
        self.address = address

    @property
    def email(self):
        return self._email
    Tabnine | Edit | Test | Explain | Document
    @email.setter
    def email(self, email):
        if "@" in email:
            self._email = email
        else:
            raise ValueError("Invalid email format")

    @property
    def phone(self):
        return self._phone
    Tabnine | Edit | Test | Explain | Document
    @phone.setter
    def phone(self, phone):
        if phone.isdigit() and len(phone) == 10:
            self._phone = phone
        else:
            raise ValueError("Phone number must be of 10 digits")

    Tabnine | Edit | Test | Explain | Document
    def display_customer_info(self):
        print(f"Customer id: {self.customer_id}")
        print(f"Name : {self.first_name} {self.last_name}")
        print(f"Email: {self.email}")
        print(f"Phone: {self.phone}")
        print(f"Address: {self.address}")

```

```

class Bank:
    Tabnine | Edit | Test | Explain | Document
    def __init__(self):
        self.customer = None
        self.account = None

    Tabnine | Edit | Test | Explain | Document
    def create_customer_accounts(self):
        print("---Customer Registration---")
        customer_id = int(input("Enter customer id: "))
        first_name = input("Enter first name: ")
        last_name = input("Enter last name: ")
        email = input("Enter email: ")
        phone = input("Enter phone number: ")
        address = input("Enter the address: ")
        try:
            self.customer = Customer(customer_id, first_name, last_name, email, phone, address)
        except Exception as e:
            print(e)
        print("---Account creation---")
        account_num = int(input("Enter account number: "))
        account_type = input("Enter account type: ")
        account_balance = int(input("Enter available balance: "))
        try:
            self.account = Accounts(account_num, account_type, account_balance)
        except Exception as E:
            print(E)

    Tabnine | Edit | Test | Explain | Document
    def show_details(self):
        print("Customer Details : \n")
        self.customer.display_customer_info()
        self.account.display_account_info()

    Tabnine | Edit | Test | Explain | Document
    def perform_transactions(self):
        while True:
            choice = int(input("Enter choice : \n1.show details\n2.deposit\n3.withdraw\n4.Calculate Interest\n5.exit \nOption : "))
            if choice == 1:
                self.show_details()
            elif choice == 2:
                amount = float(input("Enter the amount you want to deposit: "))
                self.account.deposit(amount)
            elif choice == 3:
                amount = float(input("Enter the amount you want to withdraw: "))
                self.account.withdraw(amount)
            elif choice == 4:
                self.account.calculate_interest()
            elif choice == 5:
                print("Thank you!")
                break
            else:
                print("invalid choice")

```

```

class Accounts:
    Tabnine | Edit | Test | Explain | Document
    def __init__(self, account_num=0, account_type="", account_balance=0):
        self.account_num = account_num
        self.account_type = account_type
        self.account_balance = account_balance

    #get
    Tabnine | Edit | Test | Explain | Document
    def get_account_num(self): return self.account_num
    Tabnine | Edit | Test | Explain | Document
    def get_account_type(self): return self.account_type
    Tabnine | Edit | Test | Explain | Document
    def get_account_balance(self): return self.account_balance

    #set
    Tabnine | Edit | Test | Explain | Document
    def set_account_num(self, account_num): self.account_num = account_num
    Tabnine | Edit | Test | Explain | Document
    def set_account_type(self, account_type): self.account_type = account_type
    Tabnine | Edit | Test | Explain | Document
    def set_account_balance(self, account_balance): self.account_balance = account_balance

    #method
    Tabnine | Edit | Test | Explain | Document
    def display_account_info(self):
        print(f"Account number is : {self.account_num}")
        print(f"Account type is : {self.account_type}")
        print(f"Account balance is : {self.account_balance:.2f}")

    Tabnine | Edit | Test | Explain | Document
    def deposit(self, amount : float):
        self.account_balance += amount
        print(f"The amount of {amount:.2f} has been deposited and the new balance is {self.account_balance:.2f}")

    Tabnine | Edit | Test | Explain | Document
    def withdraw(self, amount: float):
        if self.account_balance > amount:
            self.account_balance = self.account_balance - amount
            print(f"The amount of {amount:.2f} has been withdrawn and your new balance is {self.account_balance:.2f}")
        else:
            print("Insufficient funds!")

    Tabnine | Edit | Test | Explain | Document
    def calculate_interest(self):
        rate = 4.5
        if self.account_balance > 0:
            interest = self.account_balance * rate
            print(f"The interest for your balance of {self.account_balance} is {interest}")

```

```

PS E:\banking_system> & C:/Users/Lenovo/AppData/Local/Programs/Python/Python310/Python.exe
---Customer Registration---
Enter customer id: 1
Enter first name : mohammed
Enter last name: Sheriff
Enter email: sheriff@gmail.com
Enter phone number : 1234567890
Enter the address : west 13
---Account creation---
Enter account number : 101
Enter account type : savings
Enter available balance : 10000
Enter choice :
1.show details
2.deposit
3.withdraw
4.Calculate Interest
5.exit
Option : 1
Customer Details :

Customer id: 1
Name : mohammed Sheriff
Email: sheriff@gmail.com
Phone: 1234567890
Address: west 13
Account number is : 101
Account type is : savings
Account balance is : 10000.00
Enter choice :
1.show details
2.deposit
3.withdraw
4.Calculate Interest
5.exit
1.show details
2.deposit
3.withdraw
4.Calculate Interest
5.exit
Option : 5
Thank you!
PS E:\banking_system>

```

## 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.
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).

**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.

**Code:**

**#class customer**

class Customer:

def

\_\_init\_\_(self,customer\_id=0,first\_name="",last\_name="",email="",phone="",address=""):

self.customer\_id = customer\_id

self.first\_name = first\_name

self.last\_name = last\_name

self.email = email

self.phone = phone

self.address = address

@property

def email(self):

return self.\_email

@email.setter

def email(self,email):

if "@" in email:

self.\_email = email

else:

raise ValueError("Invalid email format")

@property

def phone(self):

return self.\_phone

@phone.setter

def phone(self,phone):

if phone.isdigit() and len(phone) == 10:

```

        self._phone = phone
    else:
        raise ValueError("Phone number must be of 10 digits")
def display_customer_info(self):
    print(f"Customer id: {self.customer_id}")
    print(f"Name : {self.first_name} {self.last_name}")
    print(f"Email: {self.email}")
    print(f"Phone: {self.phone}")
    print(f"Address: {self.address}")

```

### **#class Accounts**

```
class Accounts:
```

```

    def __init__(self,account_num=0,account_type="",account_balance=0):
        self.account_num = account_num
        self.account_type = account_type
        self.account_balance = account_balance
    #get
    def get_account_num(self): return self.account_num
    def get_account_type(self): return self.account_type
    def get_account_balance(self): return self.account_balance
    #set
    def set_account_num(self,account_num): self.account_num = account_num
    def set_account_type(self,account_type): self.account_type = account_type
    def set_account_balance(self,account_balance): self.account_balance = account_balance
    #method
    def display_account_info(self):
        print(f"Account number is : {self.account_num}")
        print(f"Account type is : {self.account_type}")
        print(f"Account balance is : {self.account_balance:.2f}")
    def deposit(self,amount :float):
        self.account_balance+=amount
        print(f"The amount of {amount:.2f} has been deposited and the new balance is {self.account_balance:.2f}")

```



```

def withdraw(self,amount:float):
    if self.account_balance > amount:
        self.account_balance = self.account_balance - amount
        print(f"The amount of {amount:.2f} has been withdrawn and your new balance is {self.account_balance:.2f}")
    else:
        print("Insufficient funds!")
def calculate_interest(self):
    rate = 4.5
    if(self.account_balance > 0):
        interest = self.account_balance * rate
        print(f"The interest for your balance of {self.account_balance} is {interest}")

```

```

8  class Customer:
    Tabnine | Edit | Test | Explain | Document
9  def __init__(self,customer_id=0,first_name="",last_name="",email="",phone="",address=""):
10     self.customer_id = customer_id
11     self.first_name = first_name
12     self.last_name = last_name
13     self.email = email
14     self.phone = phone
15     self.address = address
16
17     @property
18     def email(self):
19         return self._email
    Tabnine | Edit | Test | Explain | Document
20     @email.setter
21     def email(self,email):
22         if "@" in email:
23             self._email = email
24         else:
25             raise ValueError("Invalid email format")
26
27
28
29     @property
30     def phone(self):
31         return self._phone
    Tabnine | Edit | Test | Explain | Document
32     @phone.setter
33     def phone(self,phone):
34         if phone.isdigit() and len(phone) == 10:
35             self._phone = phone
36         else:
37             raise ValueError("Phone number must be of 10 digits")
38
39     Tabnine | Edit | Test | Explain | Document
40     def display_customer_info(self):
41         print(f"Customer id: {self.customer_id}")
42         print(f"Name : {self.first_name} {self.last_name}")
43         print(f>Email: {self.email}")
44         print(f"Phone: {self.phone}")
45         print(f"Address: {self.address}")

```

```

class Savingsaccount(Accounts):
    Tabnine | Edit | Test | Explain | Document
    def __init__(self, account_num, account_balance, interest_rate = 4.5):
        super().__init__(account_num, "Savings", account_balance)
        self.interest_rate = interest_rate

    Tabnine | Edit | Test | Explain | Document
    def calculate_interest(self):
        interest = self.account_balance * (self.interest_rate/100)
        print(f"The interest at {self.interest_rate}% is {interest:.2f}")

class Currentaccount(Accounts):
    Tabnine | Edit | Test | Explain | Document
    def __init__(self, account_num=0, account_balance=0, overdraft_limit = 5000.00):
        super().__init__(account_num, "Current", account_balance)
        self.overdraft_limit = overdraft_limit

    Tabnine | Edit | Test | Explain | Document
    def withdraw(self, amount:float):
        if amount <= self.account_balance + self.overdraft_limit:
            self.account_balance -= amount
            print(f"An amount of {amount} has been withdrawn\n New balance : {self.account_balance:.2f}")
        else:
            print(f"exceeded overdraft limit, \nMaximum withdrawal limit is {self.account_balance + self.overdraft_limit:.2f}")

    Tabnine | Edit | Test | Explain | Document
    def calculate_interest(self):
        print("Current account do not earn interest\n")

```

```

55 class Accounts:
    Tabnine | Edit | Test | Explain | Document
56     def __init__(self, account_num=0, account_type="", account_balance=0):
57         self.account_num = account_num
58         self.account_type = account_type
59         self.account_balance = account_balance
60
61     #get
    Tabnine | Edit | Test | Explain | Document
62     def get_account_num(self): return self.account_num
    Tabnine | Edit | Test | Explain | Document
63     def get_account_type(self): return self.account_type
    Tabnine | Edit | Test | Explain | Document
64     def get_account_balance(self): return self.account_balance
65
66     #set
    Tabnine | Edit | Test | Explain | Document
67     def set_account_num(self, account_num): self.account_num = account_num
    Tabnine | Edit | Test | Explain | Document
68     def set_account_type(self, account_type): self.account_type = account_type
    Tabnine | Edit | Test | Explain | Document
69     def set_account_balance(self, account_balance): self.account_balance = account_balance
70
71     #method
    Tabnine | Edit | Test | Explain | Document
72     def display_account_info(self):
73         print(f"Account number is : {self.account_num}")
74         print(f"Account type is : {self.account_type}")
75         print(f"Account balance is : {self.account_balance:.2f}")
76
    Tabnine | Edit | Test | Explain | Document
77     def deposit(self, amount :float):
78         self.account_balance += amount
79         print(f"The amount of {amount:.2f} has been deposited and the new balance is {self.account_balance:.2f}")
80
    Tabnine | Edit | Test | Explain | Document
81     def withdraw(self, amount:float):
82         if self.account_balance > amount:
83             self.account_balance = self.account_balance - amount
84             print(f"The amount of {amount:.2f} has been withdrawn and your new balance is {self.account_balance:.2f}")
85         else:
86             print("Insufficient funds!")
87
88
    Tabnine | Edit | Test | Explain | Document
89     def calculate_interest(self):
90         rate = 4.5
91         if(self.account_balance > 0):
92             interest = self.account_balance * rate
93             print(f"The interest for your balance of {self.account_balance} is {interest}")
94

```

```

133 class Bank:
134     Tabnine | Edit | Test | Explain | Document
135     def __init__(self):
136         self.customer = None
137         self.account = None
138     Tabnine | Edit | Test | Explain | Document
139     def create_customer_accounts(self):
140         print("---Customer Registration---")
141         customer_id = int(input("Enter customer id: "))
142         first_name = input("Enter first name: ")
143         last_name = input("Enter last name: ")
144         email = input("Enter email: ")
145         phone = input("Enter phone number: ")
146         address = input("Enter the address: ")
147         try:
148             self.customer = Customer(customer_id, first_name, last_name, email, phone, address)
149         except Exception as e:
150             print(e)
151         print("---Account creation---")
152         account_num = int(input("Enter account number: "))
153         account_type = input("Enter account type (Savings/Current): ").strip().lower()
154         account_balance = int(input("Enter available balance: "))
155         try:
156             if account_type == "savings":
157                 self.account = Savingsaccount(account_num, account_balance)
158             elif account_type == "current":
159                 self.account = Currentaccount(account_num, account_balance)
160         except Exception as E:
161             print(E)
162     Tabnine | Edit | Test | Explain | Document
163     def show_details(self):
164         print("Customer Details : \n")
165         self.customer.display_customer_info()
166         print("\nAccount Details\n")
167         self.account.display_account_info()
168     Tabnine | Edit | Test | Explain | Document
169     def perform_transactions(self):
170         while True:
171             choice = int(input("Enter choice : \n1.show details\n2.deposit\n3.withdraw\n4.Calculate Interest\n5.exit \nOption : "))
172             if choice == 1:
173                 self.show_details()
174             elif choice == 2:
175                 amount = float(input("Enter the amount you want to deposit: "))
176                 self.account.deposit(amount)
177             elif choice == 3:
178                 amount = float(input("Enter the amount you want to withdraw: "))
179                 self.account.withdraw(amount)
180             elif choice == 4:
181                 self.account.calculate_interest()
182             elif choice == 5:
183                 print("Thank you!")

```

```

PS E:\banking_system> & C:/Users/Lenovo/AppData/Local/Programs/Python/Python312/python.exe e:/banking_system/task8.py
---Customer Registration---
Enter customer id: 1
Enter first name: Mohammed
Enter last name: Sheriff
Enter email: SHeriff@gmail.com
Enter phone number: 1234567890
Enter the address: 12west
---Account creation---
Enter account number: 1
Enter account type (Savings/Current): Current
Enter available balance: 20000
Enter choice:
1.show details
2.deposit
3.withdraw
4.Calculate Interest
5.exit
Option: 4
Current account do not earn interest

```

## Task 9: Abstraction

1. Create an abstract class **BankAccount** that represents a generic bank account. It should include the following attributes and methods:

- **Attributes:** o Account number. o Customer name. o Balance.
- **Constructors:** o Implement default constructors and overload the constructor with Account attributes, generate getter and setter, print all information of attribute methods for the attributes.
- **Abstract methods:** o **deposit(amount: float):** Deposit the specified amount into the account. o **withdraw(amount: float):** Withdraw the specified amount from the account (implement error handling for insufficient funds). o **calculate\_interest():** Abstract method for calculating interest.

2. Create two concrete classes that inherit from **BankAccount**:

- **SavingsAccount:** A savings account that includes an additional attribute for interest rate. Implement the **calculate\_interest()** method to calculate interest based on the balance and interest rate.
- **CurrentAccount:** 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).

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. **create\_account** should display sub menu to choose type of accounts. o **Hint:** `Account acc = new SavingsAccount();` or `Account acc = new CurrentAccount();`
- **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.

### Code:

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

```
#class Customer
```

```
class Customer:
```

```
    def
```

```
    __init__(self, customer_id=0, first_name="", last_name="", email="", phone="", address=""):
```

```
        self.customer_id = customer_id
```

```
        self.first_name = first_name
```

```
        self.last_name = last_name
```

```
        self.email = email
```

```
        self.phone = phone
```

```
        self.address = address
```

```
    @property
```

```
    def name(self):
```

```

        return f'{self.first_name} {self.last_name}'

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

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

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

@phone.setter
def phone(self,phone):
    if phone.isdigit() and len(phone) == 10:
        self._phone = phone
    else:
        raise ValueError("Phone number must be of 10 digits")

def display_customer_info(self):
    print(f'Customer id: {self.customer_id}')
    print(f'Name : {self.first_name} {self.last_name}')
    print(f'Email: {self.email}')
    print(f'Phone: {self.phone}')
    print(f'Address: {self.address}')

```

### ***#account class***

```

class Accounts:
    def __init__(self,account_num=0,account_type="",account_balance=0):
        self.account_num = account_num
        self.account_type = account_type
        self.account_balance = account_balance

```

```

#get

def get_account_num(self): return self.account_num
def get_account_type(self): return self.account_type
def get_account_balance(self): return self.account_balance

#set

def set_account_num(self,account_num): self.account_num = account_num
def set_account_type(self,account_type): self.account_type = account_type
def set_account_balance(self,account_balance): self.account_balance = account_balance

#method

def display_account_info(self):
    print(f'Account number is : {self.account_num}')
    print(f'Account type is : {self.account_type}')
    print(f'Account balance is : {self.account_balance:.2f}')
def deposit(self,amount :float):
    self.account_balance+=amount
    print(f'The amount of {amount:.2f} has been deposited and the new balance is {self.account_balance:.2f}')
def withdraw(self,amount:float):
    if self.account_balance > amount:
        self.account_balance = self.account_balance - amount
        print(f'The amount of {amount:.2f} has been withdrawn and your new balance is {self.account_balance:.2f}')
    else:
        print("Insufficient funds!")
def calculate_interest(self):
    rate = 4.5
    if(self.account_balance > 0):
        interest = self.account_balance * rate
        print(f'The interest for your balance of {self.account_balance} is {interest}')

```

**#abstract class BankAccount**

```

class BankAccount(ABC):
    def __init__(self, account_num, account_balance):
        self.account_num = account_num
        self.account_balance = float(account_balance)
    def display_account_info(self):
        print(f'Account number is : {self.account_num}')
        print(f'Account balance is : {self.account_balance:.2f}')
    @abstractmethod
    def deposit(self, amount):
        pass
    @abstractmethod
    def withdraw(self, amount):
        pass
    @abstractmethod
    def calculate_interest(self):
        pass

class Savingsaccount(BankAccount):
    def __init__(self, account_num, account_balance, interest_rate = 4.5):
        super().__init__(account_num, account_balance)
        self.account_type = "Savings"
        self.interest_rate = interest_rate
    def deposit(self, amount):
        self.account_balance += amount
        print(f'An amount of {amount:.2f} has been deposited\nThe new balance is {self.account_balance:.2f}')
    def withdraw(self, amount):
        if amount < self.account_balance :
            self.account_balance -= amount
            print(f'An amount of {amount} has been withdrawn \n The new balance is {self.account_balance}')
    def calculate_interest(self):
        interest = self.account_balance * (self.interest_rate/100)
        print(f'The interest at {self.interest_rate}% is {interest:.2f}')

```

```

class Currentaccount(BankAccount):
    def __init__(self, account_num=0, account_balance=0, overdraft_limit = 5000.00):
        super().__init__(account_num, account_balance)
        self.account_type = "Current"
        self.overdraft_limit = overdraft_limit
    def deposit(self, amount):
        self.account_balance += amount
        print(f"The amount of {amount:.2f} has been deposited \n the new balance is {self.account_balance:.2f}")
    def withdraw(self, amount:float):
        if amount <= self.account_balance + self.overdraft_limit:
            self.account_balance -= amount
            print(f"An amount of {amount} has been withdrawn\n New balance : {self.account_balance:.2f}")
        else:
            print(f"exceeded overdraft limit, \nMaximum withdrawal limit is {self.account_balance + self.overdraft_limit:.2f}")
    def calculate_interest(self):
        print("Current account do not earn interest\n")

```

### ***#bank class***

```

class Bank:
    def __init__(self):
        self.customer = None
        self.account = None
    def create_customer_accounts(self):
        print("---Customer Registration---")
        customer_id = int(input("Enter customer id: "))
        first_name = input("Enter first name : ")
        last_name = input("Enter last name: ")
        email = input("Enter email: ")
        phone = input("Enter phone number : ")
        address = input("Enter the address : ")

```



```

try:
    self.customer = Customer(customer_id,first_name,last_name,email,phone,address)
except Exception as e:
    print(e)
print("---Account creation---")
account_num = int(input("Enter account number : "))
account_type = input("Enter account type (Savings/Current): ").strip().lower()
account_balance = int(input("Enter available balance : "))
try:
    if account_type == "savings":
        self.account = Savingsaccount(account_num,account_balance)
    elif account_type == "current":
        self.account = Currentaccount(account_num,account_balance)
except Exception as E:
    print(E)
def show_details(self):
    print("Customer Details : \n")
    self.customer.display_customer_info()
    print("\nAccount Details\n")
    self.account.display_account_info()
def perform_transactions(self):
    while True:
        choice = int(input("Enter choice :\n1.show details\n2.deposit\n3.withdraw\n4.Calculate Interest\n5.exit \nOption : "))
        if choice == 1:
            self.show_details()
        elif choice == 2:
            amount = float(input("Enter the amount you want to deposit : "))
            self.account.deposit(amount)
        elif choice == 3:
            amount = float(input("Enter the amount you want to withdraw : "))
            self.account.withdraw(amount)

```

```

elif choice == 4:

    self.account.calculate_interest()

elif choice == 5:

    print("Thank you!")

    break

else:

    print("invalid choice")

if __name__ == "__main__":

    bank = Bank()

    bank.create_customer_accounts()

    bank.perform_transactions()

```

```

12  class Customer:
    Tabnine | Edit | Test | Explain | Document
13  def __init__(self, customer_id=0, first_name="", last_name="", email="", phone="", address=""):
14      self.customer_id = customer_id
15      self.first_name = first_name
16      self.last_name = last_name
17      self.email = email
18      self.phone = phone
19      self.address = address
20
    Tabnine | Edit | Test | Explain | Document
21  @property
22  def name(self):
23      return f"{self.first_name}{self.last_name}"
24
25  @property
26  def email(self):
27      return self._email
    Tabnine | Edit | Test | Explain | Document
28  @email.setter
29  def email(self, email):
30      if "@" in email:
31          self._email = email
32      else:
33          raise ValueError("Invalid email format")
34
35  @property
36  def phone(self):
37      return self._phone
    Tabnine | Edit | Test | Explain | Document
38  @phone.setter
39  def phone(self, phone):
40      if phone.isdigit() and len(phone) == 10:
41          self._phone = phone
42      else:
43          raise ValueError("Phone number must be of 10 digits")
44
    Tabnine | Edit | Test | Explain | Document
45  def display_customer_info(self):
46      print(f"Customer id: {self.customer_id}")
47      print(f"Name : {self.first_name} {self.last_name}")
48      print(f"Email: {self.email}")
49      print(f"Phone: {self.phone}")
50      print(f"Address: {self.address}")

```

```

61 class Accounts:
    Tabnine | Edit | Test | Explain | Document
62     def __init__(self,account_num=0,account_type="",account_balance=0):
63         self.account_num = account_num
64         self.account_type = account_type
65         self.account_balance = account_balance
66
67     #get
    Tabnine | Edit | Test | Explain | Document
68     def get_account_num(self): return self.account_num
    Tabnine | Edit | Test | Explain | Document
69     def get_account_type(self): return self.account_type
    Tabnine | Edit | Test | Explain | Document
70     def get_account_balance(self): return self.account_balance
71
    #set
    Tabnine | Edit | Test | Explain | Document
72     def set_account_num(self,account_num): self.account_num = account_num
    Tabnine | Edit | Test | Explain | Document
73     def set_account_type(self,account_type): self.account_type = account_type
    Tabnine | Edit | Test | Explain | Document
74     def set_account_balance(self,account_balance): self.account_balance = account_balance
75
76     #method
    Tabnine | Edit | Test | Explain | Document
77     def display_account_info(self):
78         print(f"Account number is : {self.account_num}")
79         print(f"Account type is : {self.account_type}")
80         print(f"Account balance is : {self.account_balance:.2f}")
81
    Tabnine | Edit | Test | Explain | Document
82     def deposit(self,amount :float):
83         self.account_balance+=amount
84         print(f"The amount of {amount:.2f} has been deposited and the new balance is {self.account_balance:.2f}")
85
    Tabnine | Edit | Test | Explain | Document
86     def withdraw(self,amount:float):
87         if self.account_balance > amount:
88             self.account_balance = self.account_balance - amount
89             print(f"The amount of {amount:.2f} has been withdrawn and your new balance is {self.account_balance:.2f}")
90         else:
91             print("Insufficient funds!")
92
    Tabnine | Edit | Test | Explain | Document
93     def calculate_interest(self):
94         rate = 4.5
95         if(self.account_balance > 0):
96             interest = self.account_balance * rate
97             print(f"The interest for your balance of {self.account_balance} is {interest}")

```

```

110 class BankAccount(ABC):
    Tabnine | Edit | Test | Explain | Document
111     def __init__(self, account_num, account_balance):
112         self.account_num = account_num
113         self.account_balance = float(account_balance)
114
    Tabnine | Edit | Test | Explain | Document
115     def display_account_info(self):
116         print(f"Account number is : {self.account_num}")
117         print(f"Account balance is : {self.account_balance:.2f}")
118
    Tabnine | Edit | Test | Explain | Document
119     @abstractmethod
120     def deposit(self, amount):
121         pass
122
    Tabnine | Edit | Test | Explain | Document
123     @abstractmethod
124     def withdraw(self, amount):
125         pass
126
    Tabnine | Edit | Test | Explain | Document
127     @abstractmethod
128     def calculate_interest(self):
129         pass

```

```

136 class Savingsaccount(BankAccount):
    Tabnine | Edit | Test | Explain | Document
137     def __init__(self, account_num, account_balance, interest_rate = 4.5):
138         super().__init__(account_num, account_balance)
139         self.account_type = "Savings"
140         self.interest_rate = interest_rate
141
    Tabnine | Edit | Test | Explain | Document
142     def deposit(self, amount):
143         self.account_balance += amount
144         print(f"An amount of {amount:.2f} has been deposited\nThe new balance is {self.account_balance:.2f}")
145
    Tabnine | Edit | Test | Explain | Document
146     def withdraw(self, amount):
147         if amount < self.account_balance :
148             self.account_balance -= amount
149             print(f"An amount of {amount} has been withdrawn \n The new balance is {self.account_balance}")
150
    Tabnine | Edit | Test | Explain | Document
151     def calculate_interest(self):
152         interest = self.account_balance * (self.interest_rate/100)
153         print(f"The interest at {self.interest_rate}% is {interest:.2f}")
154
155
156
157 class Currentaccount(BankAccount):
    Tabnine | Edit | Test | Explain | Document
158     def __init__(self, account_num=0, account_balance=0, overdraft_limit = 5000.00):
159         super().__init__(account_num, account_balance)
160         self.account_type = "Current"
161         self.overdraft_limit = overdraft_limit
162
    Tabnine | Edit | Test | Explain | Document
163     def deposit(self, amount):
164         self.account_balance += amount
165         print(f"The amount of {amount:.2f} has been deposited \n the new balance is {self.account_balance:.2f}")
166
    Tabnine | Edit | Test | Explain | Document
167     def withdraw(self, amount:float):
168         if amount <= self.account_balance + self.overdraft_limit:
169             self.account_balance -= amount
170             print(f"An amount of {amount} has been withdrawn\n New balance : {self.account_balance:.2f}")
171         else:
172             print(f"exceeded overdraft limit, \nMaximum withdrawal limit is {self.account_balance + self.overdraft_limit:.2f}")
173
    Tabnine | Edit | Test | Explain | Document
174     def calculate_interest(self):
175         print("Current account do not earn interest\n")

```

```

PS E:\banking_system> & C:/Users/Lenovo/AppData/Local/Programs/Python/Python39-32/Python.exe
---Customer Registration---
Enter customer id: 1
Enter first name : Mohammed
Enter last name: Sheriff
Enter email: S@email.com
Enter phone number : 1232332323
Enter the address : se12
---Account creation---
Enter account number : 101
Enter account type (Savings/Current): Savings
Enter available balance : 500
Enter choice :
1.show details
2.deposit
3.withdraw
4.Calculate Interest
5.exit
Option : 3
Enter the amount you want to withdraw : 200
An amount of 200.0 has been withdrawn
The new balance is 300.0
Enter choice :
1.show details
2.deposit
3.withdraw
4.Calculate Interest
5.exit
Option : 1
Customer Details :

Customer id: 1
Name : Mohammed Sheriff
Email: S@email.com
Phone: 1232332323
Address: se12

Account Details

Account number is : 101
Account balance is : 300.00
Enter choice :
1.show details
2.deposit
3.withdraw
4.Calculate Interest
5.exit
Option : 5

```

### Task 10: Has A Relation / Association

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. 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. 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.**

**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**

**Code:**

```
import re
```

```
# Customer Class
```

```
class Customer:
```

```
    def __init__(self, customer_id=0, first_name="", last_name="", email="", phone="", address=""):
```

```
        self.customer_id = customer_id
```

```
        self.first_name = first_name
```

```
        self.last_name = last_name
```

```
        self.email = email
```

```
        self.phone = phone
```

```
        self.address = address
```

```
    @property
```

```
    def email(self):
```

```
        return self._email
```

```
    @email.setter
```

```
    def email(self, email):
```

```
        if "@" in email:
```

```
            self._email = email
```

```
        else:
```

```
            raise ValueError("Invalid email format")
```

```
    @property
```

```

def phone(self):
    return self._phone

@phone.setter
def phone(self, phone):
    if phone.isdigit() and len(phone) == 10:
        self._phone = phone
    else:
        raise ValueError("Phone number must be of 10 digits")

def display_customer_info(self):
    print(f"Customer ID: {self.customer_id}")
    print(f"Name: {self.first_name} {self.last_name}")
    print(f>Email: {self.email}")
    print(f>Phone: {self.phone}")
    print(f>Address: {self.address}")

```

### **# *BankAccount Abstract Class***

```

from abc import ABC, abstractmethod

class BankAccount(ABC):
    def __init__(self, account_num, account_balance, customer):
        self.account_num = account_num
        self.account_balance = float(account_balance)
        self.customer = customer # Has-A Relationship

    def display_account_info(self):
        print(f"\n--- Account Info ---")
        print(f"Account Number: {self.account_num}")
        print(f"Account Balance: ₹{self.account_balance:.2f}")
        self.customer.display_customer_info()

    @abstractmethod
    def deposit(self, amount): pass

    @abstractmethod
    def withdraw(self, amount): pass

    @abstractmethod
    def calculate_interest(self): pass

```

### ***#SavingsAccount Class***

```
class Savingsaccount(BankAccount):  
    def __init__(self, account_num, account_balance, customer, interest_rate=4.5):  
        super().__init__(account_num, account_balance, customer)  
        self.account_type = "Savings"  
        self.interest_rate = interest_rate  
    def deposit(self, amount):  
        self.account_balance += amount  
        print(f"₹{amount:.2f} deposited. New Balance: ₹{self.account_balance:.2f}")  
    def withdraw(self, amount):  
        if amount <= self.account_balance:  
            self.account_balance -= amount  
            print(f"₹{amount:.2f} withdrawn. New Balance: ₹{self.account_balance:.2f}")  
        else:  
            print("Insufficient funds!")  
    def calculate_interest(self):  
        interest = self.account_balance * (self.interest_rate / 100)  
        print(f"Interest earned at {self.interest_rate}%: ₹{interest:.2f}")
```

### ***# CurrentAccount Class***

```
class Currentaccount(BankAccount):  
    def __init__(self, account_num, account_balance, customer, overdraft_limit=5000.0):  
        super().__init__(account_num, account_balance, customer)  
        self.account_type = "Current"  
        self.overdraft_limit = overdraft_limit  
    def deposit(self, amount):  
        self.account_balance += amount  
        print(f"₹{amount:.2f} deposited. New Balance: ₹{self.account_balance:.2f}")  
    def withdraw(self, amount):  
        if amount <= self.account_balance + self.overdraft_limit:  
            self.account_balance -= amount  
            print(f"₹{amount:.2f} withdrawn. New Balance: ₹{self.account_balance:.2f}")
```



```
        else:
            print("Exceeded overdraft limit!")
def calculate_interest(self):
    print("Current accounts do not earn interest.")
```

### **# *Bank Class***

```
class Bank:
    def __init__(self):
        self.accounts = []
        self.next_account_num = 1001
    def create_account(self, customer, acc_type, balance):
        acc_num = self.next_account_num
        self.next_account_num += 1
        if acc_type.lower() == "savings":
            account = Savingsaccount(acc_num, balance, customer)
        elif acc_type.lower() == "current":
            account = Currentaccount(acc_num, balance, customer)
        else:
            print("Invalid account type.")
            return
        self.accounts.append(account)
        print(f"\n Account created successfully. Account Number: {acc_num}")
    def get_account_by_number(self, account_num):
        for acc in self.accounts:
            if acc.account_num == account_num:
                return acc
        return None
    def get_account_balance(self, account_num):
        acc = self.get_account_by_number(account_num)
        if acc:
            return acc.account_balance
        print("Account not found.")
    def deposit(self, account_num, amount):
```

```

    acc = self.get_account_by_number(account_num)
    if acc:
        acc.deposit(amount)
    else:
        print("Account not found.")
def withdraw(self, account_num, amount):
    acc = self.get_account_by_number(account_num)
    if acc:
        acc.withdraw(amount)
    else:
        print("Account not found.")
def transfer(self, from_acc, to_acc, amount):
    sender = self.get_account_by_number(from_acc)
    receiver = self.get_account_by_number(to_acc)
    if sender and receiver:
        if sender.account_balance + (sender.overdraft_limit if isinstance(sender,
Currentaccount) else 0) >= amount:
            sender.withdraw(amount)
            receiver.deposit(amount)
            print(" Transfer successful.")
        else:
            print(" Transfer failed due to insufficient funds.")
    else:
        print(" One or both account numbers are invalid.")
def get_account_details(self, account_num):
    acc = self.get_account_by_number(account_num)
    if acc:
        acc.display_account_info()
    else:
        print("Account not found.")

```

### ***# Main Bank Application***

```
def main():
```

```
bank = Bank()
while True:
    print("\n===== HMBank Menu =====")
    print("1. Create Account")
    print("2. Deposit")
    print("3. Withdraw")
    print("4. Transfer")
    print("5. Get Balance")
    print("6. Get Account Details")
    print("7. Exit")
    choice = input("Enter your choice: ")
    if choice == "1":
        print("--- Enter Customer Details ---")
        try:
            cust_id = int(input("Customer ID: "))
            fname = input("First Name: ")
            lname = input("Last Name: ")
            email = input("Email: ")
            phone = input("Phone (10 digits): ")
            address = input("Address: ")
            customer = Customer(cust_id, fname, lname, email, phone, address)
            acc_type = input("Account Type (Savings/Current): ")
            balance = float(input("Initial Balance: "))
            bank.create_account(customer, acc_type, balance)
        except Exception as e:
            print(f"Error: {e}")
    elif choice == "2":
        acc_no = int(input("Enter Account Number: "))
        amount = float(input("Amount to deposit: "))
        bank.deposit(acc_no, amount)
    elif choice == "3":
        acc_no = int(input("Enter Account Number: "))
        amount = float(input("Amount to withdraw: "))
```

```
        bank.withdraw(acc_no, amount)
elif choice == "4":
    from_acc = int(input("From Account Number: "))
    to_acc = int(input("To Account Number: "))
    amount = float(input("Transfer Amount: "))
    bank.transfer(from_acc, to_acc, amount)
elif choice == "5":
    acc_no = int(input("Enter Account Number: "))
    balance = bank.get_account_balance(acc_no)
    if balance is not None:
        print(f"Current Balance: ₹ {balance:.2f}")
elif choice == "6":
    acc_no = int(input("Enter Account Number: "))
    bank.get_account_details(acc_no)
elif choice == "7":
    print("Thank you for using HMBank. Goodbye!")
    break
else:
    print("Invalid choice. Please try again.")

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

```

1  import re
2
3  # Customer Class
4  class Customer:
5      Tabnine | Edit | Test | Explain | Document
6      def __init__(self, customer_id=0, first_name="", last_name="", email="", phone="", address=""):
7          self.customer_id = customer_id
8          self.first_name = first_name
9          self.last_name = last_name
10         self.email = email
11         self.phone = phone
12         self.address = address
13
14         @property
15         def email(self):
16             return self._email
17
18         Tabnine | Edit | Test | Explain | Document
19         @email.setter
20         def email(self, email):
21             if "@" in email:
22                 self._email = email
23             else:
24                 raise ValueError("Invalid email format")
25
26         @property
27         def phone(self):
28             return self._phone
29
30         Tabnine | Edit | Test | Explain | Document
31         @phone.setter
32         def phone(self, phone):
33             if phone.isdigit() and len(phone) == 10:
34                 self._phone = phone
35             else:
36                 raise ValueError("Phone number must be of 10 digits")
37
38         Tabnine | Edit | Test | Explain | Document
39         def display_customer_info(self):
40             print(f"Customer ID: {self.customer_id}")
41             print(f"Name: {self.first_name} {self.last_name}")
42             print(f>Email: {self.email}")
43             print(f"Phone: {self.phone}")
44             print(f"Address: {self.address}")

```

```

46 class BankAccount(ABC):
47     def __init__(self, account_num, account_balance, customer):
48         self.account_num = account_num
49         self.account_balance = float(account_balance)
50         self.customer = customer # Has-A Relationship
51
52     def display_account_info(self):
53         print(f"\n--- Account Info ---")
54         print(f"Account Number: {self.account_num}")
55         print(f"Account Balance: ₹{self.account_balance:.2f}")
56         self.customer.display_customer_info()
57
58     @abstractmethod
59     def deposit(self, amount): pass
60
61     @abstractmethod
62     def withdraw(self, amount): pass
63
64     @abstractmethod
65     def calculate_interest(self): pass
66
67
68 #SavingsAccount Class
69 class Savingsaccount(BankAccount):
70     def __init__(self, account_num, account_balance, customer, interest_rate=4.5):
71         super().__init__(account_num, account_balance, customer)
72         self.account_type = "Savings"
73         self.interest_rate = interest_rate
74
75     def deposit(self, amount):
76         self.account_balance += amount
77         print(f"₹{amount:.2f} deposited. New Balance: ₹{self.account_balance:.2f}")
78
79     def withdraw(self, amount):
80         if amount <= self.account_balance:
81             self.account_balance -= amount
82             print(f"₹{amount:.2f} withdrawn. New Balance: ₹{self.account_balance:.2f}")
83         else:
84             print("Insufficient funds!")
85
86     def calculate_interest(self):
87         interest = self.account_balance * (self.interest_rate / 100)
88         print(f"Interest earned at {self.interest_rate}%: ₹{interest:.2f}")

```

```

181 # Main Bank Application
182 Tabnine | Edit | Test | Explain | Document
183 def main():
184     bank = Bank()
185
186     while True:
187         print("\n===== HMBank Menu =====")
188         print("1. Create Account")
189         print("2. Deposit")
190         print("3. Withdraw")
191         print("4. Transfer")
192         print("5. Get Balance")
193         print("6. Get Account Details")
194         print("7. Exit")
195         choice = input("Enter your choice: ")
196
197         if choice == "1":
198             print("--- Enter Customer Details ---")
199             try:
200                 cust_id = int(input("Customer ID: "))
201                 fname = input("First Name: ")
202                 lname = input("Last Name: ")
203                 email = input("Email: ")
204                 phone = input("Phone (10 digits): ")
205                 address = input("Address: ")
206                 customer = Customer(cust_id, fname, lname, email, phone, address)
207
208                 acc_type = input("Account Type (Savings/Current): ")
209                 balance = float(input("Initial Balance: "))
210
211                 bank.create_account(customer, acc_type, balance)
212
213             except Exception as e:
214                 print(f"Error: {e}")
215
216         elif choice == "2":
217             acc_no = int(input("Enter Account Number: "))
218             amount = float(input("Amount to deposit: "))
219             bank.deposit(acc_no, amount)
220
221         elif choice == "3":
222             acc_no = int(input("Enter Account Number: "))
223             amount = float(input("Amount to withdraw: "))
224             bank.withdraw(acc_no, amount)
225
226         elif choice == "4":
227             from_acc = int(input("From Account Number: "))
228             to_acc = int(input("To Account Number: "))
229             amount = float(input("Transfer Amount: "))
230             bank.transfer(from_acc, to_acc, amount)

```

```

231         elif choice == "5":
232             acc_no = int(input("Enter Account Number: "))
233             balance = bank.get_account_balance(acc_no)
234             if balance is not None:
235                 print(f"Current Balance: ₹{balance:.2f}")
236
237         elif choice == "6":
238             acc_no = int(input("Enter Account Number: "))
239             bank.get_account_details(acc_no)
240
241         elif choice == "7":
242             print("Thank you for using HMBank. Goodbye!")
243             break
244
245         else:
246             print("Invalid choice. Please try again.")
247
248 if __name__ == "__main__":
249     main()

```

```

PS E:\banking_system> & C:/Users/Lenovo/AppData/Local/Programs/Python/Python312/python.exe e:/banking_system/task10.py

===== HMBank Menu =====
1. Create Account
2. Deposit
3. Withdraw
4. Transfer
5. Get Balance
6. Get Account Details
7. Exit
Enter your choice: 1
--- Enter Customer Details ---
Customer ID: 1
First Name: Mohammed
Last Name: Sheriff
Email: s@gmail.com
Phone (10 digits): 2121212121
Address: 12west
Account Type (Savings/Current): Savings
Initial Balance: 5000

Account created successfully. Account Number: 1001

===== HMBank Menu =====
1. Create Account
2. Deposit
3. Withdraw
4. Transfer
5. Get Balance
6. Get Account Details
7. Exit
Enter your choice: 5
Enter Account Number: 1001
Enter your choice: 5
Enter Account Number: 1001
Current Balance: ₹5000.00

===== HMBank Menu =====
1. Create Account
2. Deposit
3. Withdraw
4. Transfer
5. Get Balance
6. Get Account Details
7. Exit
Enter your choice: 7
Thank you for using HMBank. Goodbye!
PS E:\banking_system>

```

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

1. Create a 'Customer' class as mentioned above task.
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
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.
4. 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. 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.

**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

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

**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

**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.

**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.**

**10. Should display appropriate message when the account number is not found and insufficient fund or any other wrong information provided.**

**Code:**

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

**# Customer Class**

```
class Customer:
```

```
    def __init__(self, customer_id, first_name, last_name, email, phone, address):
```

```
        self.customer_id = customer_id
```

```
        self.first_name = first_name
```

```
        self.last_name = last_name
```

```
        self.email = email
```

```
        self.phone = phone
```

```
        self.address = address
```

```
def display_customer_info(self):  
    print(f"Customer ID: {self.customer_id}")  
    print(f"Name: {self.first_name} {self.last_name}")  
    print(f"Email: {self.email}")  
    print(f"Phone: {self.phone}")  
    print(f"Address: {self.address}")
```

### **# *Account Class***

```
class Account:  
    last_acc_no = 1000  
    def __init__(self, acc_type, acc_balance, customer):  
        Account.last_acc_no += 1  
        self.acc_no = Account.last_acc_no  
        self.acc_type = acc_type  
        self.acc_balance = acc_balance  
        self.customer = customer  
    def display_account_info(self):  
        print(f"\nAccount Number: {self.acc_no}")  
        print(f"Account Type: {self.acc_type}")  
        print(f"Account Balance: ₹{self.acc_balance:.2f}")  
        self.customer.display_customer_info()
```

### **#*SavingsAccount Class***

```
class SavingsAccount(Account):  
    def __init__(self, acc_balance, customer, interest_rate=4.5):  
        if acc_balance < 500:  
            raise ValueError("Minimum balance for SavingsAccount is ₹500.")  
        super().__init__("Savings", acc_balance, customer)  
        self.interest_rate = interest_rate  
  
    def calculate_interest(self):  
        interest = self.acc_balance * (self.interest_rate / 100)  
        return interest
```

### **# *CurrentAccount Class***

```
class CurrentAccount(Account):  
    def __init__(self, acc_balance, customer, overdraft_limit=5000):  
        super().__init__("Current", acc_balance, customer)  
        self.overdraft_limit = overdraft_limit  
    def withdraw(self, amount):  
        if amount <= self.acc_balance + self.overdraft_limit:  
            self.acc_balance -= amount  
        else:  
            raise Exception("Overdraft limit exceeded")
```

### **# *ZeroBalanceAccount Class***

```
class ZeroBalanceAccount(Account):  
    def __init__(self, customer):  
        super().__init__("ZeroBalance", 0.0, customer)
```

### **# *Interfaces***

```
class ICustomerServiceProvider(ABC):  
    @abstractmethod  
    def get_account_balance(self, acc_no): pass  
    @abstractmethod  
    def deposit(self, acc_no, amount): pass  
    @abstractmethod  
    def withdraw(self, acc_no, amount): pass  
    @abstractmethod  
    def transfer(self, from_acc, to_acc, amount): pass  
    @abstractmethod  
    def get_account_details(self, acc_no): pass  
  
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
```

### **# *Service Implementations***

```
class CustomerServiceProviderImpl(ICustomerServiceProvider):
```

```
    def __init__(self):
```

```
        self.account_list = []
```

```
    def get_account_by_number(self, acc_no):
```

```
        for acc in self.account_list:
```

```
            if acc.acc_no == acc_no:
```

```
                return acc
```

```
        return None
```

```
    def get_account_balance(self, acc_no):
```

```
        acc = self.get_account_by_number(acc_no)
```

```
        if acc:
```

```
            return acc.acc_balance
```

```
        else:
```

```
            raise Exception("Account not found.")
```

```
    def deposit(self, acc_no, amount):
```

```
        acc = self.get_account_by_number(acc_no)
```

```
        if acc:
```

```
            acc.acc_balance += amount
```

```
            return acc.acc_balance
```

```
        else:
```

```
            raise Exception("Account not found.")
```

```
    def withdraw(self, acc_no, amount):
```

```
        acc = self.get_account_by_number(acc_no)
```

```
        if isinstance(acc, SavingsAccount) and acc.acc_balance - amount < 500:
```

```
            raise Exception("Minimum balance of ₹500 required.")
```

```
        elif isinstance(acc, CurrentAccount):
```

```

        acc.withdraw(amount)
    elif acc and acc.acc_balance >= amount:
        acc.acc_balance -= amount
    else:
        raise Exception("Insufficient balance or account not found.")
    return acc.acc_balance

def transfer(self, from_acc, to_acc, amount):
    sender = self.get_account_by_number(from_acc)
    receiver = self.get_account_by_number(to_acc)
    if not sender or not receiver:
        raise Exception("Invalid account numbers")
    self.withdraw(from_acc, amount)
    self.deposit(to_acc, amount)

def get_account_details(self, acc_no):
    acc = self.get_account_by_number(acc_no)
    if acc:
        acc.display_account_info()
    else:
        raise Exception("Account not found.")

class BankServiceProviderImpl(CustomerServiceProviderImpl, IBankServiceProvider):
    def __init__(self, branch_name, branch_address):
        super().__init__()
        self.branch_name = branch_name
        self.branch_address = branch_address

    def create_account(self, customer, acc_type, balance=0.0):
        if acc_type.lower() == "savings":
            acc = SavingsAccount(balance, customer)
        elif acc_type.lower() == "current":
            acc = CurrentAccount(balance, customer)
        elif acc_type.lower() == "zerobalance":
            acc = ZeroBalanceAccount(customer)
        else:
            raise Exception("Invalid account type.")

```

```

        self.account_list.append(acc)

        print(f' Account created successfully with Account Number: {acc.acc_no}')

    def list_accounts(self):
        for acc in self.account_list:
            acc.display_account_info()

    def calculate_interest(self):
        for acc in self.account_list:
            if isinstance(acc, SavingsAccount):
                interest = acc.calculate_interest()
                print(f'Account {acc.acc_no} earned interest ₹{interest:.2f}')

```

### **# Main Application-**

```

def main():
    bank = BankServiceProviderImpl("HexaBank", "Main Branch")

    while True:
        print("\n===== HMBank Menu =====")
        print("1. Create Account")
        print("2. Deposit")
        print("3. Withdraw")
        print("4. Transfer")
        print("5. Get Balance")
        print("6. Get Account Details")
        print("7. List All Accounts")
        print("8. Calculate Interest")
        print("9. Exit")
        choice = input("Enter your choice: ")
        try:
            if choice == "1":
                print("--- Customer Details ---")
                cid = int(input("Customer ID: "))
                fname = input("First Name: ")
                lname = input("Last Name: ")
                email = input("Email: ")

```

```
phone = input("Phone (10 digits): ")
addr = input("Address: ")
customer = Customer(cid, fname, lname, email, phone, addr)
acc_type = input("Account Type (Savings/Current/ZeroBalance): ").lower()
bal = 0.0 if acc_type == "zerobalance" else float(input("Initial Balance: "))
bank.create_account(customer, acc_type, bal)
elif choice == "2":
    acc_no = int(input("Account Number: "))
    amt = float(input("Amount to Deposit: "))
    new_bal = bank.deposit(acc_no, amt)
    print(f"New Balance: ₹{new_bal:.2f}")
elif choice == "3":
    acc_no = int(input("Account Number: "))
    amt = float(input("Amount to Withdraw: "))
    new_bal = bank.withdraw(acc_no, amt)
    print(f"New Balance: ₹{new_bal:.2f}")
elif choice == "4":
    from_acc = int(input("From Account: "))
    to_acc = int(input("To Account: "))
    amt = float(input("Amount to Transfer: "))
    bank.transfer(from_acc, to_acc, amt)
    print("Transfer Successful.")
elif choice == "5":
    acc_no = int(input("Account Number: "))
    bal = bank.get_account_balance(acc_no)
    print(f"Current Balance: ₹{bal:.2f}")
elif choice == "6":
    acc_no = int(input("Account Number: "))
    bank.get_account_details(acc_no)
elif choice == "7":
    bank.list_accounts()
elif choice == "8":
    bank.calculate_interest()
```

```

elif choice == "9":
    print("Thank you for using HMBank!")
    break
else:
    print(" Invalid choice.")
except Exception as e:
    print(f" Error: {e}")
if __name__ == "__main__":
    main()

```

```

1  from abc import ABC, abstractmethod
2
3
4  # Customer Class
5  class Customer:
6      Tabnine | Edit | Test | Explain | Document
7      def __init__(self, customer_id, first_name, last_name, email, phone, address):
8          self.customer_id = customer_id
9          self.first_name = first_name
10         self.last_name = last_name
11         self.email = email
12         self.phone = phone
13         self.address = address
14
15     Tabnine | Edit | Test | Explain | Document
16     def display_customer_info(self):
17         print(f"Customer ID: {self.customer_id}")
18         print(f"Name: {self.first_name} {self.last_name}")
19         print(f"Email: {self.email}")
20         print(f"Phone: {self.phone}")
21         print(f"Address: {self.address}")
22
23     # Account Class
24     class Account:
25         last_acc_no = 1000
26
27         Tabnine | Edit | Test | Explain | Document
28         def __init__(self, acc_type, acc_balance, customer):
29             Account.last_acc_no += 1
30             self.acc_no = Account.last_acc_no
31             self.acc_type = acc_type
32             self.acc_balance = acc_balance
33             self.customer = customer
34
35         Tabnine | Edit | Test | Explain | Document
36         def display_account_info(self):
37             print(f"\nAccount Number: {self.acc_no}")
38             print(f"Account Type: {self.acc_type}")
39             print(f"Account Balance: ₹{self.acc_balance:.2f}")
40             self.customer.display_customer_info()

```



```

66 # ZeroBalanceAccount Class
67 class ZeroBalanceAccount(Account):
68     Tabnine | Edit | Test | Explain | Document
69     def __init__(self, customer):
70         super().__init__("ZeroBalance", 0.0, customer)
71
72 # Interfaces
73 class ICustomerServiceProvider(ABC):
74     Tabnine | Edit | Test | Explain | Document
75     @abstractmethod
76     def get_account_balance(self, acc_no): pass
77
78     Tabnine | Edit | Test | Explain | Document
79     @abstractmethod
80     def deposit(self, acc_no, amount): pass
81
82     Tabnine | Edit | Test | Explain | Document
83     @abstractmethod
84     def withdraw(self, acc_no, amount): pass
85
86     Tabnine | Edit | Test | Explain | Document
87     @abstractmethod
88     def transfer(self, from_acc, to_acc, amount): pass
89
90     Tabnine | Edit | Test | Explain | Document
91     @abstractmethod
92     def get_account_details(self, acc_no): pass
93
94 class IBankServiceProvider(ABC):
95     Tabnine | Edit | Test | Explain | Document
96     @abstractmethod
97     def create_account(self, customer, acc_type, balance): pass
98
99     Tabnine | Edit | Test | Explain | Document
100    @abstractmethod
101    def list_accounts(self): pass
102
103    Tabnine | Edit | Test | Explain | Document
104    @abstractmethod
105    def calculate_interest(self): pass

```

```

22 # Account Class
23 class Account:
24     last_acc_no = 1000
25
26     Tabnine | Edit | Test | Explain | Document
27     def __init__(self, acc_type, acc_balance, customer):
28         Account.last_acc_no += 1
29         self.acc_no = Account.last_acc_no
30         self.acc_type = acc_type
31         self.acc_balance = acc_balance
32         self.customer = customer
33
34     Tabnine | Edit | Test | Explain | Document
35     def display_account_info(self):
36         print(f"\nAccount Number: {self.acc_no}")
37         print(f"Account Type: {self.acc_type}")
38         print(f"Account Balance: ₹{self.acc_balance:.2f}")
39         self.customer.display_customer_info()
40
41 # SavingsAccount Class
42 class SavingsAccount(Account):
43     Tabnine | Edit | Test | Explain | Document
44     def __init__(self, acc_balance, customer, interest_rate=4.5):
45         if acc_balance < 500:
46             raise ValueError("Minimum balance for SavingsAccount is ₹500.")
47         super().__init__("Savings", acc_balance, customer)
48         self.interest_rate = interest_rate
49
50     Tabnine | Edit | Test | Explain | Document
51     def calculate_interest(self):
52         interest = self.acc_balance * (self.interest_rate / 100)
53         return interest
54
55 # CurrentAccount Class
56 class CurrentAccount(Account):
57     Tabnine | Edit | Test | Explain | Document
58     def __init__(self, acc_balance, customer, overdraft_limit=5000):
59         super().__init__("Current", acc_balance, customer)
60         self.overdraft_limit = overdraft_limit
61
62     Tabnine | Edit | Test | Explain | Document
63     def withdraw(self, amount):
64         if amount <= self.acc_balance + self.overdraft_limit:
65             self.acc_balance -= amount
66         else:
67             raise Exception("Overdraft limit exceeded")

```

```

102 class CustomerServiceProviderImpl(ICustomerServiceProvider):
103     Tabnine | Edit | Test | Explain | Document
104     def __init__(self):
105         self.account_list = []
106
107     Tabnine | Edit | Test | Explain | Document
108     def get_account_by_number(self, acc_no):
109         for acc in self.account_list:
110             if acc.acc_no == acc_no:
111                 return acc
112         return None
113
114     Tabnine | Edit | Test | Explain | Document
115     def get_account_balance(self, acc_no):
116         acc = self.get_account_by_number(acc_no)
117         if acc:
118             return acc.acc_balance
119         else:
120             raise Exception("Account not found.")
121
122     Tabnine | Edit | Test | Explain | Document
123     def deposit(self, acc_no, amount):
124         acc = self.get_account_by_number(acc_no)
125         if acc:
126             acc.acc_balance += amount
127             return acc.acc_balance
128         else:
129             raise Exception("Account not found.")
130
131     Tabnine | Edit | Test | Explain | Document
132     def withdraw(self, acc_no, amount):
133         acc = self.get_account_by_number(acc_no)
134         if isinstance(acc, SavingsAccount) and acc.acc_balance - amount < 500:
135             raise Exception("Minimum balance of ₹500 required.")
136         elif isinstance(acc, CurrentAccount):
137             acc.withdraw(amount)
138         elif acc and acc.acc_balance >= amount:
139             acc.acc_balance -= amount
140         else:
141             raise Exception("Insufficient balance or account not found.")
142         return acc.acc_balance
143
144     Tabnine | Edit | Test | Explain | Document
145     def transfer(self, from_acc, to_acc, amount):
146         sender = self.get_account_by_number(from_acc)
147         receiver = self.get_account_by_number(to_acc)
148         if not sender or not receiver:
149             raise Exception("Invalid account numbers")
150         self.withdraw(from_acc, amount)
151         self.deposit(to_acc, amount)

```

```

147     Tabnine | Edit | Test | Explain | Document
148     def get_account_details(self, acc_no):
149         acc = self.get_account_by_number(acc_no)
150         if acc:
151             acc.display_account_info()
152         else:
153             raise Exception("Account not found.")
154
155     class BankServiceProviderImpl(CustomerServiceProviderImpl, IBankServiceProvider):
156         Tabnine | Edit | Test | Explain | Document
157         def __init__(self, branch_name, branch_address):
158             super().__init__()
159             self.branch_name = branch_name
160             self.branch_address = branch_address
161
162         Tabnine | Edit | Test | Explain | Document
163         def create_account(self, customer, acc_type, balance=0.0):
164             if acc_type.lower() == "savings":
165                 acc = SavingsAccount(balance, customer)
166             elif acc_type.lower() == "current":
167                 acc = CurrentAccount(balance, customer)
168             elif acc_type.lower() == "zerobalance":
169                 acc = ZeroBalanceAccount(customer)
170             else:
171                 raise Exception("Invalid account type.")
172             self.account_list.append(acc)
173             print(f" Account created successfully with Account Number: {acc.acc_no}")
174
175         Tabnine | Edit | Test | Explain | Document
176         def list_accounts(self):
177             for acc in self.account_list:
178                 acc.display_account_info()
179
180         Tabnine | Edit | Test | Explain | Document
181         def calculate_interest(self):
182             for acc in self.account_list:
183                 if isinstance(acc, SavingsAccount):
184                     interest = acc.calculate_interest()
185                     print(f"Account {acc.acc_no} earned interest ₹{interest:.2f}")

```

● PS E:\banking\_system> & C:/Users/Lenovo/AppData/Local/Programs,

===== HMBank Menu =====

1. Create Account
2. Deposit
3. Withdraw
4. Transfer
5. Get Balance
6. Get Account Details
7. List All Accounts
8. Calculate Interest
9. Exit

Enter your choice: 1

--- Customer Details ---

Customer ID: 1

First Name: Mohammed

Last Name: Sheriff

Email: Sheriff@email.com

Phone (10 digits): 1234567890

Address: west 12

Account Type (Savings/Current/ZeroBalance): ZeroBalance

Account created successfully with Account Number: 1001

**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.

**3. OverDraftLimitExceededException** throw this exception when current account customer try to with draw amount from the current account.

**4. NullPointerException** handle in main method. Throw these exceptions from the methods in HMBank class. Make necessary changes to accommodate these exception in the source code. Handle all these exceptions from the main program

**Code:**

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

**#Exceptions below**

```
class InsufficientFundException(Exception):
```

```
    pass
```

```
class InvalidAccountException(Exception):
```

```
    pass
```

```
class OverDraftLimitExceededException(Exception):
```

```
    pass
```

**# Customer Class file**

```
class Customer:
```

```
    def __init__(self, customer_id, first_name, last_name, email, phone, address):
```

```
        self.customer_id = customer_id
```

```
        self.first_name = first_name
```

```
        self.last_name = last_name
```

```
        self.email = email
```

```
        self.phone = phone
```

```
        self.address = address
```

```
    def display_customer_info(self):
```

```
        print(f"Customer ID: {self.customer_id}")
```

```
        print(f"Name: {self.first_name} {self.last_name}")
```

```
print(f'Email: {self.email}')
print(f'Phone: {self.phone}')
print(f'Address: {self.address}')
```

### **# *Account Class***

```
class Account:
    last_acc_no = 1000
    def __init__(self, acc_type, acc_balance, customer):
        Account.last_acc_no += 1
        self.acc_no = Account.last_acc_no
        self.acc_type = acc_type
        self.acc_balance = acc_balance
        self.customer = customer
    def display_account_info(self):
        print(f'\nAccount Number: {self.acc_no}')
        print(f'Account Type: {self.acc_type}')
        print(f'Account Balance: ₹{self.acc_balance:.2f}')
        self.customer.display_customer_info()
```

### **# *SavingsAccount Class***

```
class SavingsAccount(Account):
    def __init__(self, acc_balance, customer, interest_rate=4.5):
        if acc_balance < 500:
            raise InsufficientFundException("Minimum balance ₹500 required for savings account.")
        super().__init__("Savings", acc_balance, customer)
        self.interest_rate = interest_rate

    def calculate_interest(self):
        return self.acc_balance * (self.interest_rate / 100)
```

### **# *CurrentAccount Class***

```
class CurrentAccount(Account):
```

```
def __init__(self, acc_balance, customer, overdraft_limit=5000):
    super().__init__("Current", acc_balance, customer)
    self.overdraft_limit = overdraft_limit
def withdraw(self, amount):
    if amount <= self.acc_balance + self.overdraft_limit:
        self.acc_balance -= amount
    else:
        raise OverDraftLimitExceededException(" Overdraft limit exceeded.")
```

### # *ZeroBalanceAccount Class*

```
class ZeroBalanceAccount(Account):
    def __init__(self, customer):
        super().__init__("ZeroBalance", 0.0, customer)
```

### # *Interfaces*

```
class ICustomerServiceProvider(ABC):
    @abstractmethod
    def get_account_balance(self, acc_no): pass
    @abstractmethod
    def deposit(self, acc_no, amount): pass
    @abstractmethod
    def withdraw(self, acc_no, amount): pass
    @abstractmethod
    def transfer(self, from_acc, to_acc, amount): pass

    @abstractmethod
    def get_account_details(self, acc_no): pass
```

```
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
```

### **# Service Implementations**

```
class CustomerServiceProviderImpl(ICustomerServiceProvider):
```

```
    def __init__(self):
```

```
        self.account_list = []
```

```
    def get_account_by_number(self, acc_no):
```

```
        for acc in self.account_list:
```

```
            if acc.acc_no == acc_no:
```

```
                return acc
```

```
        raise InvalidAccountException(f"Account {acc_no} not found.")
```

```
    def get_account_balance(self, acc_no):
```

```
        acc = self.get_account_by_number(acc_no)
```

```
        return acc.acc_balance
```

```
    def deposit(self, acc_no, amount):
```

```
        acc = self.get_account_by_number(acc_no)
```

```
        acc.acc_balance += amount
```

```
        return acc.acc_balance
```

```
    def withdraw(self, acc_no, amount):
```

```
        acc = self.get_account_by_number(acc_no)
```

```
        if isinstance(acc, SavingsAccount):
```

```
            if acc.acc_balance - amount < 500:
```

```
                raise InsufficientFundException(" Withdrawal would violate minimum balance of ₹500.")
```

```
            acc.acc_balance -= amount
```

```
        elif isinstance(acc, CurrentAccount):
```

```
            acc.withdraw(amount)
```

```
        elif acc.acc_balance >= amount:
```

```
            acc.acc_balance -= amount
```

```
        else:
```

```
            raise InsufficientFundException(" Insufficient funds.")
```

```

        return acc.acc_balance

    def transfer(self, from_acc, to_acc, amount):
        self.withdraw(from_acc, amount)
        self.deposit(to_acc, amount)

    def get_account_details(self, acc_no):
        acc = self.get_account_by_number(acc_no)
        acc.display_account_info()

```

```

class BankServiceProviderImpl(CustomerServiceProviderImpl, IBankServiceProvider):

```

```

    def __init__(self, branch_name, branch_address):
        super().__init__()
        self.branch_name = branch_name
        self.branch_address = branch_address

    def create_account(self, customer, acc_type, balance=0.0):
        if acc_type == "savings":
            acc = SavingsAccount(balance, customer)
        elif acc_type == "current":
            acc = CurrentAccount(balance, customer)
        elif acc_type == "zerobalance":
            acc = ZeroBalanceAccount(customer)
        else:
            raise ValueError("Invalid account type.")
        self.account_list.append(acc)
        print(f"Account created. Account Number: {acc.acc_no}")

    def list_accounts(self):
        for acc in self.account_list:
            acc.display_account_info()

    def calculate_interest(self):
        for acc in self.account_list:
            if isinstance(acc, SavingsAccount):
                interest = acc.calculate_interest()
                print(f"Account {acc.acc_no} earned interest ₹{interest:.2f}")

```



## # *Main App*

```
def main():
```

```
    bank = BankServiceProviderImpl("HexaBank", "Main Branch")
```

```
    while True:
```

```
        print("\n===== HMBank Menu =====")
```

```
        print("1. Create Account")
```

```
        print("2. Deposit")
```

```
        print("3. Withdraw")
```

```
        print("4. Transfer")
```

```
        print("5. Get Balance")
```

```
        print("6. Get Account Details")
```

```
        print("7. List Accounts")
```

```
        print("8. Calculate Interest")
```

```
        print("9. Exit")
```

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

```
        try:
```

```
            if choice == "1":
```

```
                cid = int(input("Customer ID: "))
```

```
                fname = input("First Name: ")
```

```
                lname = input("Last Name: ")
```

```
                email = input("Email: ")
```

```
                phone = input("Phone: ")
```

```
                addr = input("Address: ")
```

```
                customer = Customer(cid, fname, lname, email, phone, addr)
```

```
                acc_type = input("Account Type (savings/current/zerobalance): ").lower()
```

```
                bal = 0.0 if acc_type == "zerobalance" else float(input("Initial Balance: "))
```

```
                bank.create_account(customer, acc_type, bal)
```

```
            elif choice == "2":
```

```
                acc = int(input("Account Number: "))
```

```
                amt = float(input("Amount to deposit: "))
```

```
                print(f"New Balance: ₹{bank.deposit(acc, amt):.2f}")
```

```
            elif choice == "3":
```

```
                acc = int(input("Account Number: "))
```

```

        amt = float(input("Amount to withdraw: "))
        print(f'New Balance: ₹{bank.withdraw(acc, amt):.2f}')
    elif choice == "4":
        from_acc = int(input("From Account: "))
        to_acc = int(input("To Account: "))
        amt = float(input("Amount: "))
        bank.transfer(from_acc, to_acc, amt)
        print("Transfer successful.")
    elif choice == "5":
        acc = int(input("Account Number: "))
        print(f'Balance: ₹{bank.get_account_balance(acc):.2f}')
    elif choice == "6":
        acc = int(input("Account Number: "))
        bank.get_account_details(acc)
    elif choice == "7":
        bank.list_accounts()
    elif choice == "8":
        bank.calculate_interest()
    elif choice == "9":
        print("Thanks for using HMBank!")
        break
    else:
        print(" Invalid choice.")

except (InvalidAccountException, InsufficientFundException,
OverDraftLimitExceededException, ValueError) as e:
    print(f' {e}')
except Exception as e:
    print(f' Unexpected error: {e}')

if __name__ == "__main__":
    main()

```

```

1  from abc import ABC, abstractmethod
2
3  # Custom Exceptions
4  class InsufficientFundException(Exception):
5      pass
6
7  class InvalidAccountException(Exception):
8      pass
9
10 class OverDraftLimitExceededException(Exception):
11     pass
12
13 # Customer Class
14 class Customer:
15     Tabnine | Edit | Test | Explain | Document
16     def __init__(self, customer_id, first_name, last_name, email, phone, address):
17         self.customer_id = customer_id
18         self.first_name = first_name
19         self.last_name = last_name
20         self.email = email
21         self.phone = phone
22         self.address = address
23
24     Tabnine | Edit | Test | Explain | Document
25     def display_customer_info(self):
26         print(f"Customer ID: {self.customer_id}")
27         print(f>Name: {self.first_name} {self.last_name}")
28         print(f>Email: {self.email}")
29         print(f>Phone: {self.phone}")
30         print(f>Address: {self.address}")
31
32 # Account Class
33 class Account:
34     last_acc_no = 1000
35
36     Tabnine | Edit | Test | Explain | Document
37     def __init__(self, acc_type, acc_balance, customer):
38         Account.last_acc_no += 1
39         self.acc_no = Account.last_acc_no
40         self.acc_type = acc_type
41         self.acc_balance = acc_balance
42         self.customer = customer
43
44     Tabnine | Edit | Test | Explain | Document
45     def display_account_info(self):
46         print(f"\nAccount Number: {self.acc_no}")
47         print(f>Account Type: {self.acc_type}")
48         print(f>Account Balance: ₹{self.acc_balance:.2f}")
49         self.customer.display_customer_info()

```

```

49 # SavingsAccount Class
50 class SavingsAccount(Account):
51     Tabnine | Edit | Test | Explain | Document
52     def __init__(self, acc_balance, customer, interest_rate=4.5):
53         if acc_balance < 500:
54             raise InsufficientFundException("Minimum balance ₹500 required for savings account.")
55         super().__init__("Savings", acc_balance, customer)
56         self.interest_rate = interest_rate
57
58     Tabnine | Edit | Test | Explain | Document
59     def calculate_interest(self):
60         return self.acc_balance * (self.interest_rate / 100)
61
62 # CurrentAccount Class
63 class CurrentAccount(Account):
64     Tabnine | Edit | Test | Explain | Document
65     def __init__(self, acc_balance, customer, overdraft_limit=5000):
66         super().__init__("Current", acc_balance, customer)
67         self.overdraft_limit = overdraft_limit
68
69     Tabnine | Edit | Test | Explain | Document
70     def withdraw(self, amount):
71         if amount <= self.acc_balance + self.overdraft_limit:
72             self.acc_balance -= amount
73         else:
74             raise OverDraftLimitExceededException(" Overdraft limit exceeded.")
75
76 # ZeroBalanceAccount Class
77 class ZeroBalanceAccount(Account):
78     Tabnine | Edit | Test | Explain | Document
79     def __init__(self, customer):
80         super().__init__("ZeroBalance", 0.0, customer)

```

```

80 # Interfaces
81 class ICustomerServiceProvider(ABC):
82     Tabnine | Edit | Test | Explain | Document
83     @abstractmethod
84     def get_account_balance(self, acc_no): pass
85
86     Tabnine | Edit | Test | Explain | Document
87     @abstractmethod
88     def deposit(self, acc_no, amount): pass
89
90     Tabnine | Edit | Test | Explain | Document
91     @abstractmethod
92     def withdraw(self, acc_no, amount): pass
93
94     Tabnine | Edit | Test | Explain | Document
95     @abstractmethod
96     def transfer(self, from_acc, to_acc, amount): pass
97
98     Tabnine | Edit | Test | Explain | Document
99     @abstractmethod
100     def get_account_details(self, acc_no): pass
101
102 class IBankServiceProvider(ABC):
103     Tabnine | Edit | Test | Explain | Document
104     @abstractmethod
105     def create_account(self, customer, acc_type, balance): pass
106
107     Tabnine | Edit | Test | Explain | Document
108     @abstractmethod
109     def list_accounts(self): pass
110
111     Tabnine | Edit | Test | Explain | Document
112     @abstractmethod
113     def calculate_interest(self): pass

```

```

109 # Service Implementations
110 class CustomerServiceProviderImpl(ICustomerServiceProvider):
111     Tabnine | Edit | Test | Explain | Document
112     def __init__(self):
113         self.account_list = []
114
115     Tabnine | Edit | Test | Explain | Document
116     def get_account_by_number(self, acc_no):
117         for acc in self.account_list:
118             if acc.acc_no == acc_no:
119                 return acc
120         raise InvalidAccountException(f"Account {acc_no} not found.")
121
122     Tabnine | Edit | Test | Explain | Document
123     def get_account_balance(self, acc_no):
124         acc = self.get_account_by_number(acc_no)
125         return acc.acc_balance
126
127     Tabnine | Edit | Test | Explain | Document
128     def deposit(self, acc_no, amount):
129         acc = self.get_account_by_number(acc_no)
130         acc.acc_balance += amount
131         return acc.acc_balance
132
133     Tabnine | Edit | Test | Explain | Document
134     def withdraw(self, acc_no, amount):
135         acc = self.get_account_by_number(acc_no)
136         if isinstance(acc, SavingsAccount):
137             if acc.acc_balance - amount < 500:
138                 raise InsufficientFundException(" Withdrawal would violate minimum balance of ₹500.")
139             acc.acc_balance -= amount
140         elif isinstance(acc, CurrentAccount):
141             acc.withdraw(amount)
142         elif acc.acc_balance >= amount:
143             acc.acc_balance -= amount
144         else:
145             raise InsufficientFundException(" Insufficient funds.")
146         return acc.acc_balance
147
148     Tabnine | Edit | Test | Explain | Document
149     def transfer(self, from_acc, to_acc, amount):
150         self.withdraw(from_acc, amount)
151         self.deposit(to_acc, amount)
152
153     Tabnine | Edit | Test | Explain | Document
154     def get_account_details(self, acc_no):
155         acc = self.get_account_by_number(acc_no)
156         acc.display_account_info()

```

```

152 ~ class BankServiceProviderImpl(CustomerServiceProviderImpl, IBankServiceProvider):
    Tabnine | Edit | Test | Explain | Document
153 ~     def __init__(self, branch_name, branch_address):
154         super().__init__()
155         self.branch_name = branch_name
156         self.branch_address = branch_address
157
    Tabnine | Edit | Test | Explain | Document
158 ~     def create_account(self, customer, acc_type, balance=0.0):
159 ~         if acc_type == "savings":
160             acc = SavingsAccount(balance, customer)
161 ~         elif acc_type == "current":
162             acc = CurrentAccount(balance, customer)
163 ~         elif acc_type == "zerobalance":
164             acc = ZeroBalanceAccount(customer)
165 ~         else:
166             raise ValueError("Invalid account type.")
167         self.account_list.append(acc)
168         print(f" Account created. Account Number: {acc.acc_no}")
169
    Tabnine | Edit | Test | Explain | Document
170 ~     def list_accounts(self):
171 ~         for acc in self.account_list:
172             acc.display_account_info()
173
    Tabnine | Edit | Test | Explain | Document
174 ~     def calculate_interest(self):
175 ~         for acc in self.account_list:
176 ~             if isinstance(acc, SavingsAccount):
177                 interest = acc.calculate_interest()
178                 print(f"Account {acc.acc_no} earned interest ₹{interest:.2f}")
179

```

```

181 # Main App
    Tabnine | Edit | Test | Explain | Document
182 def main():
183     bank = BankServiceProviderImpl("HexaBank", "Main Branch")
184
185     while True:
186         print("\n===== HMBank Menu =====")
187         print("1. Create Account")
188         print("2. Deposit")
189         print("3. Withdraw")
190         print("4. Transfer")
191         print("5. Get Balance")
192         print("6. Get Account Details")
193         print("7. List Accounts")
194         print("8. Calculate Interest")
195         print("9. Exit")
196
197         choice = input("Enter your choice: ")
198
199         try:
200             if choice == "1":
201                 cid = int(input("Customer ID: "))
202                 fname = input("First Name: ")
203                 lname = input("Last Name: ")
204                 email = input("Email: ")
205                 phone = input("Phone: ")
206                 addr = input("Address: ")
207                 customer = Customer(cid, fname, lname, email, phone, addr)
208
209                 acc_type = input("Account Type (savings/current/zerobalance): ").lower()
210                 bal = 0.0 if acc_type == "zerobalance" else float(input("Initial Balance: "))
211                 bank.create_account(customer, acc_type, bal)
212
213             elif choice == "2":
214                 acc = int(input("Account Number: "))
215                 amt = float(input("Amount to deposit: "))
216                 print(f"New Balance: ₹{bank.deposit(acc, amt):.2f}")
217
218             elif choice == "3":
219                 acc = int(input("Account Number: "))
220                 amt = float(input("Amount to withdraw: "))
221                 print(f"New Balance: ₹{bank.withdraw(acc, amt):.2f}")

```

```

223         elif choice == "4":
224             from_acc = int(input("From Account: "))
225             to_acc = int(input("To Account: "))
226             amt = float(input("Amount: "))
227             bank.transfer(from_acc, to_acc, amt)
228             print("Transfer successful.")
229
230         elif choice == "5":
231             acc = int(input("Account Number: "))
232             print(f"Balance: ₹{bank.get_account_balance(acc):.2f}")
233
234         elif choice == "6":
235             acc = int(input("Account Number: "))
236             bank.get_account_details(acc)
237
238         elif choice == "7":
239             bank.list_accounts()
240
241         elif choice == "8":
242             bank.calculate_interest()
243
244         elif choice == "9":
245             print("Thanks for using HMBank!")
246             break
247
248         else:
249             print(" Invalid choice.")
250
251     except (InvalidAccountException, InsufficientFundException, OverDraftLimitExceededException, ValueError) as e:
252         print(f" {e}")
253     except Exception as e:
254         print(f" Unexpected error: {e}")
255
256
257 if __name__ == "__main__":
258     main()
259

```

===== HMBank Menu =====

```

1. Create Account
2. Deposit
3. Withdraw
4. Transfer
5. Get Balance
6. Get Account Details
7. List Accounts
8. Calculate Interest
9. Exit
Enter your choice: 1
Customer ID: 1
First Name: Mohammed
Last Name: Ibrahim
Email: Sheriff@gmail.com
Phone: 1234567890
Address: west 13
Account Type (savings/current/zerobalance): Current
Initial Balance: 60000
Account created. Account Number: 1001

```

===== HMBank Menu =====

```

1. Create Account
2. Deposit
3. Withdraw
4. Transfer
5. Get Balance
6. Get Account Details
7. List Accounts
8. Calculate Interest
9. Exit
Enter your choice: 3
Account Number: 1001
Amount to withdraw: 70000
Overdraft limit exceeded.

```

### Task 13: Collection

**1. From the previous task change the HMBank attribute Accounts to List of Accounts and perform the same operation.**

**Code:**

```
class BankServiceProviderImpl:
    def __init__(self):
        self.account_list = [] # list of Account objects
    def create_account(self, customer, acc_type, balance):
        if acc_type.lower() == "savings":
            acc = SavingsAccount(balance, customer)
        elif acc_type.lower() == "current":
            acc = CurrentAccount(balance, customer)
        elif acc_type.lower() == "zerobalance":
            acc = ZeroBalanceAccount(customer)
        else:
            raise ValueError("Invalid account type")
        self.account_list.append(acc)
        print(f" Account created. Account Number: {acc.acc_no}")
    def get_account_by_number(self, acc_no):
        for acc in self.account_list:
            if acc.acc_no == acc_no:
                return acc
        raise InvalidAccountException("Account not found.")
    def deposit(self, acc_no, amount):
        acc = self.get_account_by_number(acc_no)
        acc.acc_balance += amount
        return acc.acc_balance
    def withdraw(self, acc_no, amount):
        acc = self.get_account_by_number(acc_no)
        if isinstance(acc, SavingsAccount):
            if acc.acc_balance - amount < 500:
                raise InsufficientFundException("Minimum ₹500 balance required.")
            acc.acc_balance -= amount
```

```

        elif isinstance(acc, CurrentAccount):
            acc.withdraw(amount)
        elif acc.acc_balance >= amount:
            acc.acc_balance -= amount
        else:
            raise InsufficientFundException("Insufficient funds.")
        return acc.acc_balance

def transfer(self, from_acc, to_acc, amount):
    self.withdraw(from_acc, amount)
    self.deposit(to_acc, amount)

def get_account_details(self, acc_no):
    acc = self.get_account_by_number(acc_no)
    acc.display_account_info()

def list_accounts(self):
    for acc in self.account_list:
        acc.display_account_info()

def calculate_interest(self):
    for acc in self.account_list:
        if isinstance(acc, SavingsAccount):
            interest = acc.calculate_interest()
            print(f"Account {acc.acc_no} earned interest ₹ {interest:.2f}")

```

```

from abc import ABC, abstractmethod

```

### **#Exceptions**

```

class InsufficientFundException(Exception):
    pass

class InvalidAccountException(Exception):
    pass

class OverDraftLimitExceededException(Exception):
    pass

```

### **# Customer Class**



```
class Customer:
```

```
    def __init__(self, customer_id, first_name, last_name, email, phone, address):
```

```
        self.customer_id = customer_id
```

```
        self.first_name = first_name
```

```
        self.last_name = last_name
```

```
        self.email = email
```

```
        self.phone = phone
```

```
        self.address = address
```

```
    def display_customer_info(self):
```

```
        print(f"Customer ID: {self.customer_id}")
```

```
        print(f"Name: {self.first_name} {self.last_name}")
```

```
        print(f"Email: {self.email}")
```

```
        print(f"Phone: {self.phone}")
```

```
        print(f"Address: {self.address}")
```

## **# Account Class**

```
class Account:
```

```
    last_acc_no = 1000
```

```
    def __init__(self, acc_type, acc_balance, customer):
```

```
        Account.last_acc_no += 1
```

```
        self.acc_no = Account.last_acc_no
```

```
        self.acc_type = acc_type
```

```
        self.acc_balance = acc_balance
```

```
        self.customer = customer
```

```
    def display_account_info(self):
```

```
        print(f"\nAccount Number: {self.acc_no}")
```

```
        print(f"Account Type: {self.acc_type}")
```

```
        print(f"Account Balance: ₹{self.acc_balance:.2f}")
```

```
        self.customer.display_customer_info()
```

## **# SavingsAccount Class**

```
class SavingsAccount(Account):
```

```
    def __init__(self, acc_balance, customer, interest_rate=4.5):
```

```

        if acc_balance < 500:
            raise InsufficientFundException("Minimum balance ₹500 required for savings account.")
        super().__init__("Savings", acc_balance, customer)
        self.interest_rate = interest_rate
    def calculate_interest(self):
        return self.acc_balance * (self.interest_rate / 100)

```

### # CurrentAccount Class

```

class CurrentAccount(Account):
    def __init__(self, acc_balance, customer, overdraft_limit=5000):
        super().__init__("Current", acc_balance, customer)
        self.overdraft_limit = overdraft_limit

    def withdraw(self, amount):
        if amount <= self.acc_balance + self.overdraft_limit:
            self.acc_balance -= amount
        else:
            raise OverDraftLimitExceededException(" Overdraft limit exceeded.")

```

### # ZeroBalanceAccount Class

```

class ZeroBalanceAccount(Account):
    def __init__(self, customer):
        super().__init__("ZeroBalance", 0.0, customer)

```

### # Interfaces

```

class ICustomerServiceProvider(ABC):
    @abstractmethod
    def get_account_balance(self, acc_no): pass
    @abstractmethod
    def deposit(self, acc_no, amount): pass
    @abstractmethod
    def withdraw(self, acc_no, amount): pass

```

```

    @abstractmethod
    def transfer(self, from_acc, to_acc, amount): pass

    @abstractmethod
    def get_account_details(self, acc_no): pass
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


# Service Implementations
class CustomerServiceProviderImpl(ICustomerServiceProvider):

    def __init__(self):
        self.account_list = []

    def get_account_by_number(self, acc_no):
        for acc in self.account_list:
            if acc.acc_no == acc_no:
                return acc

        raise InvalidAccountException(f"Account {acc_no} not found.")

    def get_account_balance(self, acc_no):
        acc = self.get_account_by_number(acc_no)
        return acc.acc_balance

    def deposit(self, acc_no, amount):
        acc = self.get_account_by_number(acc_no)
        acc.acc_balance += amount
        return acc.acc_balance

    def withdraw(self, acc_no, amount):
        acc = self.get_account_by_number(acc_no)
        if isinstance(acc, SavingsAccount):
            if acc.acc_balance - amount < 500:

```

```

        raise InsufficientFundException(" Withdrawal would violate minimum balance of
₹500.")

        acc.acc_balance -= amount
    elif isinstance(acc, CurrentAccount):
        acc.withdraw(amount)
    elif acc.acc_balance >= amount:
        acc.acc_balance -= amount
    else:
        raise InsufficientFundException(" Insufficient funds.")
    return acc.acc_balance

def transfer(self, from_acc, to_acc, amount):
    self.withdraw(from_acc, amount)
    self.deposit(to_acc, amount)

def get_account_details(self, acc_no):
    acc = self.get_account_by_number(acc_no)
    acc.display_account_info()

class BankServiceProviderImpl(CustomerServiceProviderImpl, IBankServiceProvider):
    def __init__(self, branch_name, branch_address):
        super().__init__()
        self.branch_name = branch_name
        self.branch_address = branch_address

    def create_account(self, customer, acc_type, balance=0.0):
        if acc_type == "savings":
            acc = SavingsAccount(balance, customer)
        elif acc_type == "current":
            acc = CurrentAccount(balance, customer)
        elif acc_type == "zerobalance":
            acc = ZeroBalanceAccount(customer)
        else:
            raise ValueError("Invalid account type.")
        self.account_list.append(acc)
        print(f" Account created. Account Number: {acc.acc_no}")

```

```

def list_accounts(self):
    for acc in self.account_list:
        acc.display_account_info()
def calculate_interest(self):
    for acc in self.account_list:
        if isinstance(acc, SavingsAccount):
            interest = acc.calculate_interest()
            print(f'Account {acc.acc_no} earned interest ₹{interest:.2f}')

```

### # *Main App*

```

def main():
    bank = BankServiceProviderImpl("HexaBank", "Main Branch")
    while True:
        print("\n===== HMBank Menu =====")
        print("1. Create Account")
        print("2. Deposit")
        print("3. Withdraw")
        print("4. Transfer")
        print("5. Get Balance")
        print("6. Get Account Details")
        print("7. List Accounts")
        print("8. Calculate Interest")
        print("9. Exit")
        choice = input("Enter your choice: ")
        try:
            if choice == "1":
                cid = int(input("Customer ID: "))
                fname = input("First Name: ")
                lname = input("Last Name: ")
                email = input("Email: ")
                phone = input("Phone: ")
                addr = input("Address: ")
                customer = Customer(cid, fname, lname, email, phone, addr)

```

```
acc_type = input("Account Type (savings/current/zerobalance): ").lower()
bal = 0.0 if acc_type == "zerobalance" else float(input("Initial Balance: "))
bank.create_account(customer, acc_type, bal)
elif choice == "2":
    acc = int(input("Account Number: "))
    amt = float(input("Amount to deposit: "))
    print(f'New Balance: ₹{bank.deposit(acc, amt):.2f}')
elif choice == "3":
    acc = int(input("Account Number: "))
    amt = float(input("Amount to withdraw: "))
    print(f'New Balance: ₹{bank.withdraw(acc, amt):.2f}')
elif choice == "4":
    from_acc = int(input("From Account: "))
    to_acc = int(input("To Account: "))
    amt = float(input("Amount: "))
    bank.transfer(from_acc, to_acc, amt)
    print("Transfer successful.")
elif choice == "5":
    acc = int(input("Account Number: "))
    print(f'Balance: ₹{bank.get_account_balance(acc):.2f}')
elif choice == "6":
    acc = int(input("Account Number: "))
    bank.get_account_details(acc)
elif choice == "7":
    bank.list_accounts()
elif choice == "8":
    bank.calculate_interest()
elif choice == "9":
    print("Thanks for using HMBank!")
    break
else:
    print(" Invalid choice.")
```

except (InvalidAccountException, InsufficientFundException, OverDraftLimitExceededException, ValueError) as e:

print(f" {e}")

except Exception as e:

print(f" Unexpected error: {e}")

if \_\_name\_\_ == "\_\_main\_\_":

main()

```
1 class BankServiceProviderImpl:
2     def __init__(self):
3         self.account_list = [] # List of Account objects
4
5     def create_account(self, customer, acc_type, balance):
6         if acc_type.lower() == "savings":
7             acc = SavingsAccount(balance, customer)
8         elif acc_type.lower() == "current":
9             acc = CurrentAccount(balance, customer)
10        elif acc_type.lower() == "zerobalance":
11            acc = ZeroBalanceAccount(customer)
12        else:
13            raise ValueError("Invalid account type")
14
15        self.account_list.append(acc)
16        print(f" Account created. Account Number: {acc.acc_no}")
17
18    def get_account_by_number(self, acc_no):
19        for acc in self.account_list:
20            if acc.acc_no == acc_no:
21                return acc
22        raise InvalidAccountException("Account not found.")
23
24    def deposit(self, acc_no, amount):
25        acc = self.get_account_by_number(acc_no)
26        acc.acc_balance += amount
27        return acc.acc_balance
28
29    def withdraw(self, acc_no, amount):
30        acc = self.get_account_by_number(acc_no)
31        if isinstance(acc, SavingsAccount):
32            if acc.acc_balance - amount < 500:
33                raise InsufficientFundException("Minimum ₹500 balance required.")
34            acc.acc_balance -= amount
35        elif isinstance(acc, CurrentAccount):
36            acc.withdraw(amount)
37        elif acc.acc_balance >= amount:
38            acc.acc_balance -= amount
39        else:
40            raise InsufficientFundException("Insufficient funds.")
41        return acc.acc_balance
42
43    def transfer(self, from_acc, to_acc, amount):
44        self.withdraw(from_acc, amount)
45        self.deposit(to_acc, amount)
46
47    def get_account_details(self, acc_no):
48        acc = self.get_account_by_number(acc_no)
49        acc.display_account_info()
```

```

50
51     def list_accounts(self):
52         for acc in self.account_list:
53             acc.display_account_info()
54
55     def calculate_interest(self):
56         for acc in self.account_list:
57             if isinstance(acc, SavingsAccount):
58                 interest = acc.calculate_interest()
59                 print(f"Account {acc.acc_no} earned interest ₹{interest:.2f}")
60
61 from abc import ABC, abstractmethod
62
63 # Custom Exceptions
64 class InsufficientFundException(Exception):
65     pass
66
67 class InvalidAccountException(Exception):
68     pass
69
70 class OverDraftLimitExceededException(Exception):
71     pass
72
73 # Customer Class
74 class Customer:
75     Tabnine | Edit | Test | Explain | Document
76     def __init__(self, customer_id, first_name, last_name, email, phone, address):
77         self.customer_id = customer_id
78         self.first_name = first_name
79         self.last_name = last_name
80         self.email = email
81         self.phone = phone
82         self.address = address
83
84     Tabnine | Edit | Test | Explain | Document
85     def display_customer_info(self):
86         print(f"Customer ID: {self.customer_id}")
87         print(f"Name: {self.first_name} {self.last_name}")
88         print(f"Email: {self.email}")
89         print(f"Phone: {self.phone}")
90         print(f"Address: {self.address}")
91

```

```

91 # Account Class
92 class Account:
93     last_acc_no = 1000
94
95     Tabnine | Edit | Test | Explain | Document
96     def __init__(self, acc_type, acc_balance, customer):
97         Account.last_acc_no += 1
98         self.acc_no = Account.last_acc_no
99         self.acc_type = acc_type
100         self.acc_balance = acc_balance
101         self.customer = customer
102
103     Tabnine | Edit | Test | Explain | Document
104     def display_account_info(self):
105         print(f"\nAccount Number: {self.acc_no}")
106         print(f"Account Type: {self.acc_type}")
107         print(f"Account Balance: ₹{self.acc_balance:.2f}")
108         self.customer.display_customer_info()
109
110 # SavingsAccount Class
111 class SavingsAccount(Account):
112     Tabnine | Edit | Test | Explain | Document
113     def __init__(self, acc_balance, customer, interest_rate=4.5):
114         if acc_balance < 500:
115             raise InsufficientFundException("Minimum balance ₹500 required for savings account.")
116         super().__init__("Savings", acc_balance, customer)
117         self.interest_rate = interest_rate
118
119     Tabnine | Edit | Test | Explain | Document
120     def calculate_interest(self):
121         return self.acc_balance * (self.interest_rate / 100)
122
123 # CurrentAccount Class
124 class CurrentAccount(Account):
125     Tabnine | Edit | Test | Explain | Document
126     def __init__(self, acc_balance, customer, overdraft_limit=5000):
127         super().__init__("Current", acc_balance, customer)
128         self.overdraft_limit = overdraft_limit
129
130     Tabnine | Edit | Test | Explain | Document
131     def withdraw(self, amount):
132         if amount <= self.acc_balance + self.overdraft_limit:
133             self.acc_balance -= amount
134         else:
135             raise OverDraftLimitExceededException("Overdraft limit exceeded.")

```



```

134 # ZeroBalanceAccount Class
135 class ZeroBalanceAccount(Account):
136     def __init__(self, customer):
137         super().__init__("ZeroBalance", 0.0, customer)
138
139
140 # Interfaces
141 class ICustomerServiceProvider(ABC):
142     @abstractmethod
143     def get_account_balance(self, acc_no): pass
144
145     @abstractmethod
146     def deposit(self, acc_no, amount): pass
147
148     @abstractmethod
149     def withdraw(self, acc_no, amount): pass
150
151     @abstractmethod
152     def transfer(self, from_acc, to_acc, amount): pass
153
154     @abstractmethod
155     def get_account_details(self, acc_no): pass
156
157
158 class IBankServiceProvider(ABC):
159     @abstractmethod
160     def create_account(self, customer, acc_type, balance): pass
161
162     @abstractmethod
163     def list_accounts(self): pass
164
165     @abstractmethod
166     def calculate_interest(self): pass
167

```

```

169 # Service Implementations
170 class CustomerServiceProviderImpl(ICustomerServiceProvider):
171     Tabnine | Edit | Test | Explain | Document
172     def __init__(self):
173         self.account_list = []
174
175     Tabnine | Edit | Test | Explain | Document
176     def get_account_by_number(self, acc_no):
177         for acc in self.account_list:
178             if acc.acc_no == acc_no:
179                 return acc
180             raise InvalidAccountException(f"Account {acc_no} not found.")
181
182     Tabnine | Edit | Test | Explain | Document
183     def get_account_balance(self, acc_no):
184         acc = self.get_account_by_number(acc_no)
185         return acc.acc_balance
186
187     Tabnine | Edit | Test | Explain | Document
188     def deposit(self, acc_no, amount):
189         acc = self.get_account_by_number(acc_no)
190         acc.acc_balance += amount
191         return acc.acc_balance
192
193     Tabnine | Edit | Test | Explain | Document
194     def withdraw(self, acc_no, amount):
195         acc = self.get_account_by_number(acc_no)
196         if isinstance(acc, SavingsAccount):
197             if acc.acc_balance - amount < 500:
198                 raise InsufficientFundException(" Withdrawal would violate minimum balance of ₹500.")
199             acc.acc_balance -= amount
200         elif isinstance(acc, CurrentAccount):
201             acc.withdraw(amount)
202         elif acc.acc_balance >= amount:
203             acc.acc_balance -= amount
204         else:
205             raise InsufficientFundException(" Insufficient funds.")
206         return acc.acc_balance
207
208     Tabnine | Edit | Test | Explain | Document
209     def transfer(self, from_acc, to_acc, amount):
210         self.withdraw(from_acc, amount)
211         self.deposit(to_acc, amount)
212
213     Tabnine | Edit | Test | Explain | Document
214     def get_account_details(self, acc_no):
215         acc = self.get_account_by_number(acc_no)
216         acc.display_account_info()
217
218
219

```

```

211
212 class BankServiceProviderImpl(CustomerServiceProviderImpl, IBankServiceProvider):
213     Tabnine | Edit | Test | Explain | Document
214     def __init__(self, branch_name, branch_address):
215         super().__init__()
216         self.branch_name = branch_name
217         self.branch_address = branch_address
218
219     Tabnine | Edit | Test | Explain | Document
220     def create_account(self, customer, acc_type, balance=0.0):
221         if acc_type == "savings":
222             acc = SavingsAccount(balance, customer)
223         elif acc_type == "current":
224             acc = CurrentAccount(balance, customer)
225         elif acc_type == "zerobalance":
226             acc = ZeroBalanceAccount(customer)
227         else:
228             raise ValueError("Invalid account type.")
229         self.account_list.append(acc)
230         print(f" Account created. Account Number: {acc.acc_no}")
231
232     Tabnine | Edit | Test | Explain | Document
233     def list_accounts(self):
234         for acc in self.account_list:
235             acc.display_account_info()
236
237     Tabnine | Edit | Test | Explain | Document
238     def calculate_interest(self):
239         for acc in self.account_list:
240             if isinstance(acc, SavingsAccount):
241                 interest = acc.calculate_interest()
242                 print(f"Account {acc.acc_no} earned interest ₹{interest:.2f}")
243
244

```

```

241 # Main App
242 Tabnine | Edit | Test | Explain | Document
243 def main():
244     bank = BankServiceProviderImpl("HexaBank", "Main Branch")
245
246     while True:
247         print("\n===== HMBank Menu =====")
248         print("1. Create Account")
249         print("2. Deposit")
250         print("3. Withdraw")
251         print("4. Transfer")
252         print("5. Get Balance")
253         print("6. Get Account Details")
254         print("7. List Accounts")
255         print("8. Calculate Interest")
256         print("9. Exit")
257
258         choice = input("Enter your choice: ")
259
260         try:
261             if choice == "1":
262                 cid = int(input("Customer ID: "))
263                 fname = input("First Name: ")
264                 lname = input("Last Name: ")
265                 email = input("Email: ")
266                 phone = input("Phone: ")
267                 addr = input("Address: ")
268                 customer = Customer(cid, fname, lname, email, phone, addr)
269
270                 acc_type = input("Account Type (savings/current/zerobalance): ").lower()
271                 bal = 0.0 if acc_type == "zerobalance" else float(input("Initial Balance: "))
272                 bank.create_account(customer, acc_type, bal)
273
274             elif choice == "2":
275                 acc = int(input("Account Number: "))
276                 amt = float(input("Amount to deposit: "))
277                 print(f"New Balance: ₹{bank.deposit(acc, amt):.2f}")
278
279             elif choice == "3":
280                 acc = int(input("Account Number: "))
281                 amt = float(input("Amount to withdraw: "))
282                 print(f"New Balance: ₹{bank.withdraw(acc, amt):.2f}")

```

```

283             elif choice == "4":
284                 from_acc = int(input("From Account: "))
285                 to_acc = int(input("To Account: "))
286                 amt = float(input("Amount: "))
287                 bank.transfer(from_acc, to_acc, amt)
288                 print("Transfer successful.")
289
290             elif choice == "5":
291                 acc = int(input("Account Number: "))
292                 print(f"Balance: ₹{bank.get_account_balance(acc):.2f}")
293
294             elif choice == "6":
295                 acc = int(input("Account Number: "))
296                 bank.get_account_details(acc)
297
298             elif choice == "7":
299                 bank.list_accounts()
300
301             elif choice == "8":
302                 bank.calculate_interest()
303
304             elif choice == "9":
305                 print("Thanks for using HMBank!")
306                 break
307
308             else:
309                 print("Invalid choice.")
310
311         except (InvalidAccountException, InsufficientFundException, OverDraftLimitExceededException, ValueError) as e:
312             print(f" {e}")
313         except Exception as e:
314             print(f" Unexpected error: {e}")
315
316
317 if __name__ == "__main__":
318     main()

```

```

PS E:\banking_system> & C:/Users/Lenovo/AppData/Local/Programs/Python/P
===== HMBank Menu =====
1. Create Account
2. Deposit
3. Withdraw
4. Transfer
5. Get Balance
6. Get Account Details
7. List Accounts
8. Calculate Interest
9. Exit
Enter your choice: 1
Customer ID: 1
First Name: Mohammed
Last Name: Sheriff
Email: Sheriff@email.com
Phone: 1234567890
Address: west 13
Account Type (savings/current/zerobalance): Savings
Initial Balance: 20000
Account created. Account Number: 1001

===== HMBank Menu =====
1. Create Account
2. Deposit
3. Withdraw
4. Transfer
5. Get Balance
6. Get Account Details
7. List Accounts
8. Calculate Interest
9. Exit
Enter your choice: 7

Account Number: 1001
Account Type: Savings
Account Balance: ₹20000.00
Customer ID: 1
Name: Mohammed Sheriff
Email: Sheriff@email.com
Phone: 1234567890
Address: west 13

```

**2. From the previous task change the HMBank attribute Accounts to Set of Accounts and perform the same operation. • Avoid adding duplicate Account object to the set. • Create Comparator object to sort the accounts based on customer name when listAccounts() method called.**

**Code:**

**# Account**

**class Account:**

**last\_acc\_no = 1000**

**def \_\_init\_\_(self, acc\_type, acc\_balance, customer):**

**Account.last\_acc\_no += 1**

**self.acc\_no = Account.last\_acc\_no**

**self.acc\_type = acc\_type**

**self.acc\_balance = acc\_balance**

**self.customer = customer**

**def display\_account\_info(self):**

**print(f"\nAccount Number: {self.acc\_no}")**

**print(f'Account Type: {self.acc\_type}')**

**print(f'Account Balance: ₹{self.acc\_balance:.2f}')**

```
        self.customer.display_customer_info()

    def __hash__(self):
        return hash(self.acc_no)

    def __eq__(self, other):
        return isinstance(other, Account) and self.acc_no == other.acc_no
```

### **#Bankservice provider class**

```
class BankServiceProviderImpl(CustomerServiceProviderImpl, IBankServiceProvider):

    def __init__(self, branch_name, branch_address):
        super().__init__()
        self.branch_name = branch_name
        self.branch_address = branch_address

    def create_account(self, customer, acc_type, balance=0.0):
        if acc_type == "savings":
            acc = SavingsAccount(balance, customer)
        elif acc_type == "current":
            acc = CurrentAccount(balance, customer)
        elif acc_type == "zerobalance":
            acc = ZeroBalanceAccount(customer)
        else:
            raise ValueError("Invalid account type.")
        if acc in self.account_set:
            print(" Account already exists (by acc_no). Not added again.")
        else:
            self.account_set.add(acc)
            print(f" Account created. Account Number: {acc.acc_no}")

    def list_accounts(self):
        sorted_accounts = sorted(self.account_set, key=lambda acc:
acc.customer.first_name.lower())
        for acc in sorted_accounts:
            acc.display_account_info()

    def calculate_interest(self):
        for acc in self.account_set:
```

```
if isinstance(acc, SavingsAccount):
```

```
    interest = acc.calculate_interest()
```

```
    print(f'Account {acc.acc_no} earned interest ₹{interest:.2f}')
```

```
27 # Account
28 class Account:
29     last_acc_no = 1000
30
31     Tabnine | Edit | Test | Explain | Document
32     def __init__(self, acc_type, acc_balance, customer):
33         Account.last_acc_no += 1
34         self.acc_no = Account.last_acc_no
35         self.acc_type = acc_type
36         self.acc_balance = acc_balance
37         self.customer = customer
38
39     Tabnine | Edit | Test | Explain | Document
40     def display_account_info(self):
41         print(f"\nAccount Number: {self.acc_no}")
42         print(f"Account Type: {self.acc_type}")
43         print(f"Account Balance: ₹{self.acc_balance:.2f}")
44         self.customer.display_customer_info()
45
46     Tabnine | Edit | Test | Explain | Document
47     def __hash__(self):
48         return hash(self.acc_no)
49
50     Tabnine | Edit | Test | Explain | Document
51     def __eq__(self, other):
52         return isinstance(other, Account) and self.acc_no == other.acc_no
```

```
153 class BankServiceProviderImpl(CustomerServiceProviderImpl, IBankServiceProvider):
154     Tabnine | Edit | Test | Explain | Document
155     def __init__(self, branch_name, branch_address):
156         super().__init__()
157         self.branch_name = branch_name
158         self.branch_address = branch_address
159
160     Tabnine | Edit | Test | Explain | Document
161     def create_account(self, customer, acc_type, balance=0.0):
162         if acc_type == "savings":
163             acc = SavingsAccount(balance, customer)
164         elif acc_type == "current":
165             acc = CurrentAccount(balance, customer)
166         elif acc_type == "zerobalance":
167             acc = ZeroBalanceAccount(customer)
168         else:
169             raise ValueError("Invalid account type.")
170         if acc in self.account_set:
171             print("Account already exists (by acc_no). Not added again.")
172         else:
173             self.account_set.add(acc)
174             print(f"✅ Account created. Account Number: {acc.acc_no}")
175
176     Tabnine | Edit | Test | Explain | Document
177     def list_accounts(self):
178         sorted_accounts = sorted(self.account_set, key=lambda acc: acc.customer.first_name.lower())
179         for acc in sorted_accounts:
180             acc.display_account_info()
181
182     Tabnine | Edit | Test | Explain | Document
183     def calculate_interest(self):
184         for acc in self.account_set:
185             if isinstance(acc, SavingsAccount):
186                 interest = acc.calculate_interest()
187                 print(f'Account {acc.acc_no} earned interest ₹{interest:.2f}')
```

```

PS E:\banking_system> & C:/Users/Lenovo/AppData/Local/Programs/Python/P
===== HMBank Menu =====
1. Create Account
2. Deposit
3. Withdraw
4. Transfer
5. Get Balance
6. Get Account Details
7. List Accounts
8. Calculate Interest
9. Exit
Enter your choice: 1
Customer ID: 1
First Name: Mohammed
Last Name: Sheriff
Email: Sheriff@email.com
Phone: 1234567890
Address: west 13
Account Type (savings/current/zerobalance): Savings
Initial Balance: 20000
Account created. Account Number: 1001

===== HMBank Menu =====
1. Create Account
2. Deposit
3. Withdraw
4. Transfer
5. Get Balance
6. Get Account Details
7. List Accounts
8. Calculate Interest
9. Exit
Enter your choice: 7

Account Number: 1001
Account Type: Savings
Account Balance: ₹20000.00
Customer ID: 1
Name: Mohammed Sheriff
Email: Sheriff@email.com
Phone: 1234567890
Address: west 13

```

**3. From the previous task change the HMBank attribute Accounts to HashMap of Accounts and perform the same operation**

**Code:**

**# Account Class**

class Account:

last\_acc\_no = 1000

def \_\_init\_\_(self, acc\_type, acc\_balance, customer):

Account.last\_acc\_no += 1

self.acc\_no = Account.last\_acc\_no

self.acc\_type = acc\_type

self.acc\_balance = acc\_balance

self.customer = customer

def display\_account\_info(self):

print(f"\nAccount Number: {self.acc\_no}")

print(f"Account Type: {self.acc\_type}")

```
print(f'Account Balance: ₹{self.acc_balance:.2f}')
self.customer.display_customer_info()
```

### **#Bank service provider class**

```
class BankServiceProviderImpl(CustomerServiceProviderImpl, IBankServiceProvider):
```

```
    def __init__(self, branch_name, branch_address):
```

```
        super().__init__()
```

```
        self.branch_name = branch_name
```

```
        self.branch_address = branch_address
```

```
    def create_account(self, customer, acc_type, balance=0.0):
```

```
        if acc_type == "savings":
```

```
            acc = SavingsAccount(balance, customer)
```

```
        elif acc_type == "current":
```

```
            acc = CurrentAccount(balance, customer)
```

```
        elif acc_type == "zerobalance":
```

```
            acc = ZeroBalanceAccount(customer)
```

```
        else:
```

```
            raise ValueError("Invalid account type.")
```

```
        if acc.acc_no in self.account_map:
```

```
            print(" Account number already exists.")
```

```
        else:
```

```
            self.account_map[acc.acc_no] = acc
```

```
            print(f' Account created. Account Number: {acc.acc_no}')
```

```
    def list_accounts(self):
```

```
        sorted_accounts = sorted(self.account_map.values(), key=lambda acc:
acc.customer.first_name.lower())
```

```
        for acc in sorted_accounts:
```

```
            acc.display_account_info()
```

```
    def calculate_interest(self):
```

```
        for acc in self.account_map.values():
```

```
            if isinstance(acc, SavingsAccount):
```

```
                interest = acc.calculate_interest()
```

```
                print(f'Account {acc.acc_no} earned interest ₹{interest:.2f}')
```



```

24
25 # Account Class
26 class Account:
27     last_acc_no = 1000
28
29     Tabnine | Edit | Test | Explain | Document
30     def __init__(self, acc_type, acc_balance, customer):
31         Account.last_acc_no += 1
32         self.acc_no = Account.last_acc_no
33         self.acc_type = acc_type
34         self.acc_balance = acc_balance
35         self.customer = customer
36
37     Tabnine | Edit | Test | Explain | Document
38     def display_account_info(self):
39         print(f"\nAccount Number: {self.acc_no}")
40         print(f"Account Type: {self.acc_type}")
41         print(f"Account Balance: ₹{self.acc_balance:.2f}")
42         self.customer.display_customer_info()

```

```

139 class BankServiceProviderImpl(CustomerServiceProviderImpl, IBankServiceProvider):
140     Tabnine | Edit | Test | Explain | Document
141     def __init__(self, branch_name, branch_address):
142         super().__init__()
143         self.branch_name = branch_name
144         self.branch_address = branch_address
145
146     Tabnine | Edit | Test | Explain | Document
147     def create_account(self, customer, acc_type, balance=0.0):
148         if acc_type == "savings":
149             acc = SavingsAccount(balance, customer)
150         elif acc_type == "current":
151             acc = CurrentAccount(balance, customer)
152         elif acc_type == "zerobalance":
153             acc = ZeroBalanceAccount(customer)
154         else:
155             raise ValueError("Invalid account type.")
156
157         if acc.acc_no in self.account_map:
158             print("Account number already exists.")
159         else:
160             self.account_map[acc.acc_no] = acc
161             print(f"Account created. Account Number: {acc.acc_no}")
162
163     Tabnine | Edit | Test | Explain | Document
164     def list_accounts(self):
165         sorted_accounts = sorted(self.account_map.values(), key=lambda acc: acc.customer.first_name.lower())
166         for acc in sorted_accounts:
167             acc.display_account_info()
168
169     Tabnine | Edit | Test | Explain | Document
170     def calculate_interest(self):
171         for acc in self.account_map.values():
172             if isinstance(acc, SavingsAccount):
173                 interest = acc.calculate_interest()
174                 print(f"Account {acc.acc_no} earned interest ₹{interest:.2f}")

```

```

PS E:\banking_system> & C:/Users/Lenovo/AppData/Local/Programs/Python/Python39-6/Python.exe E:\banking_system\main.py
===== HMBank Menu =====
1. Create Account
2. Deposit
3. Withdraw
4. Transfer
5. Get Balance
6. Get Account Details
7. List Accounts
8. Calculate Interest
9. Exit
Enter your choice: 1
Customer ID: 1
First Name: Mohammed
Last Name: Sheriff
Email: Sheriff@email.com
Phone: 1234567890
Address: west 13
Account Type (savings/current/zerobalance): Savings
Initial Balance: 20000
Account created. Account Number: 1001

===== HMBank Menu =====
1. Create Account
2. Deposit
3. Withdraw
4. Transfer
5. Get Balance
6. Get Account Details
7. List Accounts
8. Calculate Interest
9. Exit
Enter your choice: 7

Account Number: 1001
Account Type: Savings
Account Balance: ₹20000.00
Customer ID: 1
Name: Mohammed Sheriff
Email: Sheriff@email.com
Phone: 1234567890
Address: west 13

```

#### Task 14: Database Connectivity.

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

Code:

# customer.py

class Customer:

def \_\_init\_\_(self, customer\_id, first\_name, last\_name, email, phone, address):

self.customer\_id = customer\_id

self.first\_name = first\_name

self.last\_name = last\_name

self.email = email

self.phone = phone

self.address = address

def display\_customer\_info(self):

print(f"Customer ID : {self.customer\_id}")

print(f"Name : {self.first\_name} {self.last\_name}")

```
print(f'Email      : {self.email}')
print(f'Phone      : {self.phone}')
print(f'Address    : {self.address}')
```

A screenshot of a code editor with a dark background. The file name at the top is 'task14 part1.py'. The code defines a 'Customer' class. The '\_\_init\_\_' method takes six arguments: 'self', 'customer\_id', 'first\_name', 'last\_name', 'email', 'phone', and 'address'. It assigns each argument to a corresponding attribute on 'self'. The 'display\_customer\_info' method prints out the customer's details in a formatted string. The editor has a sidebar on the left and a top bar with options like 'Tabnine', 'Edit', 'Test', 'Explain', and 'Document'.

```
1  # customer.py
2
3  class Customer:
4      def __init__(self, customer_id, first_name, last_name, email, phone, address):
5          self.customer_id = customer_id
6          self.first_name = first_name
7          self.last_name = last_name
8          self.email = email
9          self.phone = phone
10         self.address = address
11
12     def display_customer_info(self):
13         print(f"Customer ID      : {self.customer_id}")
14         print(f"Name              : {self.first_name} {self.last_name}")
15         print(f"Email                : {self.email}")
16         print(f"Phone                 : {self.phone}")
17         print(f"Address               : {self.address}")
18
```

**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**

**Code:**

class Account:

    last\_acc\_no = 1000 *# static variable shared across all accounts*

    def \_\_init\_\_(self, acc\_type, acc\_balance, customer: Customer):

        Account.last\_acc\_no += 1

        self.acc\_no = Account.last\_acc\_no

        self.acc\_type = acc\_type

        self.acc\_balance = acc\_balance

        self.customer = customer

    def display\_account\_info(self):

        print(f"\nAccount Number : {self.acc\_no}")

        print(f'Account Type : {self.acc\_type}')

        print(f'Account Balance : ₹{self.acc\_balance:.2f}')

        self.customer.display\_customer\_info()

```

5  class Account:
6      last_acc_no = 1000
7
8      Tabnine | Edit | Test | Explain | Document
9      def __init__(self, acc_type, acc_balance, customer: Customer):
10         Account.last_acc_no += 1
11         self.acc_no = Account.last_acc_no
12         self.acc_type = acc_type
13         self.acc_balance = acc_balance
14         self.customer = customer
15
16     Tabnine | Edit | Test | Explain | Document
17     def display_account_info(self):
18         print(f"\nAccount Number : {self.acc_no}")
19         print(f"Account Type : {self.acc_type}")
20         print(f"Account Balance : ₹{self.acc_balance:.2f}")
21         self.customer.display_customer_info()

```

**3. Create a class 'TRANSACTION' that include following attributes • Account • Description • Date and Time • TransactionType(Withdraw, Deposit, Transfer) • TransactionAmount**

**Code:**

**#file of transaction.py**

from datetime import datetime

class Transaction:

def \_\_init\_\_(self, transaction\_id, acc\_no, description, transaction\_type, transaction\_amount, transaction\_date=None):

self.transaction\_id = transaction\_id

self.acc\_no = acc\_no

self.description = description

self.transaction\_type = transaction\_type # e.g., deposit, withdraw

self.transaction\_amount = transaction\_amount

self.transaction\_date = transaction\_date or datetime.now().isoformat()

def display\_transaction(self):

print(f"Transaction ID : {self.transaction\_id}")

print(f"Account No : {self.acc\_no}")

print(f"Type : {self.transaction\_type}")

print(f"Description : {self.description}")

print(f"Amount : ₹{self.transaction\_amount:.2f}")

print(f>Date : {self.transaction\_date}")

```

entity > transaction.py > ...
1  # transaction.py
2
3  from datetime import datetime
4
5  class Transaction:
6      Tabnine | Edit | Test | Explain | Document
7      def __init__(self, transaction_id, acc_no, description, transaction_type, transaction_amount, transaction_date=None):
8          self.transaction_id = transaction_id
9          self.acc_no = acc_no
10         self.description = description
11         self.transaction_type = transaction_type # e.g., deposit, withdraw
12         self.transaction_amount = transaction_amount
13         self.transaction_date = transaction_date or datetime.now().isoformat()
14
15         Tabnine | Edit | Test | Explain | Document
16         def display_transaction(self):
17             print(f"Transaction ID : {self.transaction_id}")
18             print(f"Account No : {self.acc_no}")
19             print(f"Type : {self.transaction_type}")
20             print(f>Description : {self.description}")
21             print(f"Amount : ₹{self.transaction_amount:.2f}")
22             print(f>Date : {self.transaction_date}")

```

**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.

**Code:**

**# account\_types.py file**

from entity.account import Account

class SavingsAccount(Account):

def \_\_init\_\_(self, acc\_balance, customer, interest\_rate=4.5):

if acc\_balance < 500:

raise ValueError("Minimum ₹500 required for Savings Account")

super().\_\_init\_\_("Savings", acc\_balance, customer)

self.interest\_rate = interest\_rate

def calculate\_interest(self):

return self.acc\_balance \* (self.interest\_rate / 100)

class CurrentAccount(Account):

def \_\_init\_\_(self, acc\_balance, customer, overdraft\_limit=5000):

super().\_\_init\_\_("Current", acc\_balance, customer)

self.overdraft\_limit = overdraft\_limit

```

def withdraw(self, amount):
    if amount <= self.acc_balance + self.overdraft_limit:
        self.acc_balance -= amount
    else:
        raise Exception("Overdraft limit exceeded")

class ZeroBalanceAccount(Account):
    def __init__(self, customer):
        super().__init__("ZeroBalance", 0.0, customer)

```

```

1  # account_types.py
2
3  from entity.account import Account
4
5  class SavingsAccount(Account):
6      Tabnine | Edit | Test | Explain | Document
7      def __init__(self, acc_balance, customer, interest_rate=4.5):
8          if acc_balance < 500:
9              raise ValueError("Minimum ₹500 required for Savings Account")
10             super().__init__("Savings", acc_balance, customer)
11             self.interest_rate = interest_rate
12
13     Tabnine | Edit | Test | Explain | Document
14     def calculate_interest(self):
15         return self.acc_balance * (self.interest_rate / 100)
16
17     class CurrentAccount(Account):
18         Tabnine | Edit | Test | Explain | Document
19         def __init__(self, acc_balance, customer, overdraft_limit=5000):
20             super().__init__("Current", acc_balance, customer)
21             self.overdraft_limit = overdraft_limit
22
23         Tabnine | Edit | Test | Explain | Document
24         def withdraw(self, amount):
25             if amount <= self.acc_balance + self.overdraft_limit:
26                 self.acc_balance -= amount
27             else:
28                 raise Exception("Overdraft limit exceeded")
29
30     class ZeroBalanceAccount(Account):
31         Tabnine | Edit | Test | Explain | Document
32         def __init__(self, customer):
33             super().__init__("ZeroBalance", 0.0, customer)

```

**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.

**Code:**

**# icustomer\_service\_provider.py**

from abc import ABC, abstractmethod

class ICustomerServiceProvider(ABC):

    @abstractmethod

    def get\_account\_balance(self, acc\_no): pass

    @abstractmethod

    def deposit(self, acc\_no, amount): pass

    @abstractmethod

    def withdraw(self, acc\_no, amount): pass

    @abstractmethod

    def transfer(self, from\_acc, to\_acc, amount): pass

    @abstractmethod

    def get\_account\_details(self, acc\_no): pass

```
1  # icustomer_service_provider.py
2
3  from abc import ABC, abstractmethod
4
5  class ICustomerServiceProvider(ABC):
6      Tabnine | Edit | Test | Explain | Document
7      @abstractmethod
8      def get_account_balance(self, acc_no): pass
9
10     Tabnine | Edit | Test | Explain | Document
11     @abstractmethod
12     def deposit(self, acc_no, amount): pass
13
14     Tabnine | Edit | Test | Explain | Document
15     @abstractmethod
16     def withdraw(self, acc_no, amount): pass
17
18     Tabnine | Edit | Test | Explain | Document
19     @abstractmethod
20     def transfer(self, from_acc, to_acc, amount): pass
21
22     Tabnine | Edit | Test | Explain | Document
23     @abstractmethod
24     def get_account_details(self, acc_no): pass
25
```

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.

Code:

# **ibank\_service\_provider.py** file

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

```
class IBankServiceProvider(ABC):
```

```
    @abstractmethod
```

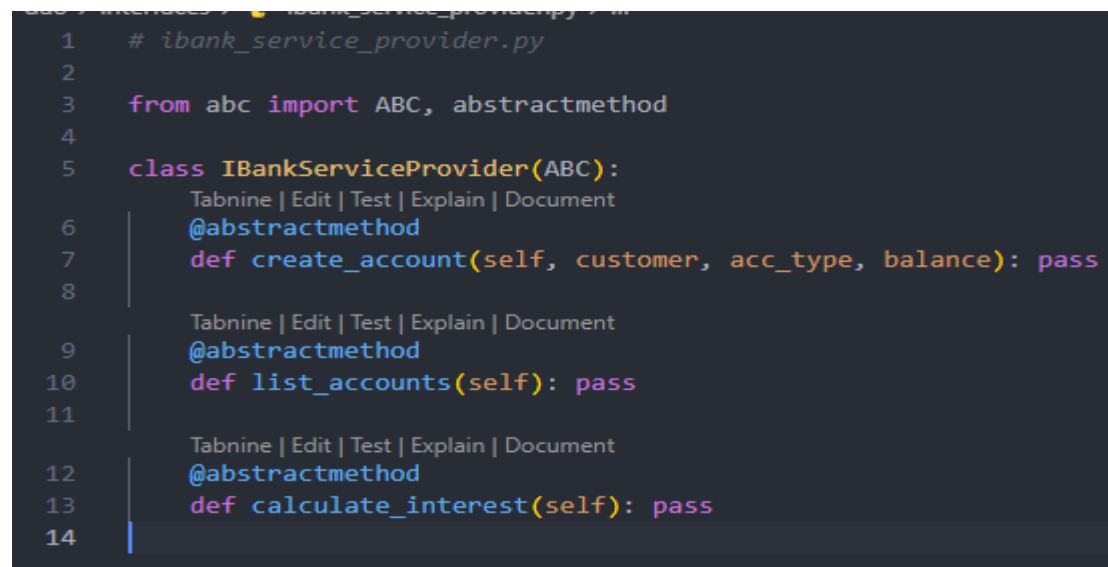
```
    def create_account(self, customer, acc_type, balance): pass
```

```
    @abstractmethod
```

```
    def list_accounts(self): pass
```

```
    @abstractmethod
```

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



```
1  # ibank_service_provider.py
2
3  from abc import ABC, abstractmethod
4
5  class IBankServiceProvider(ABC):
6      @abstractmethod
7      def create_account(self, customer, acc_type, balance): pass
8
9      @abstractmethod
10     def list_accounts(self): pass
11
12     @abstractmethod
13     def calculate_interest(self): pass
14
```

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

Code:

# **customer\_service\_impl.py** file

```
from dao.interfaces.icustomer_service_provider import ICustomerServiceProvider
```

```
from exception.exception_module import InvalidAccountException,
InsufficientFundException
```



```
class CustomerServiceProviderImpl(ICustomerServiceProvider):
    def __init__(self):
        self.account_list = []
    def get_account_by_number(self, acc_no):
        for acc in self.account_list:
            if acc.acc_no == acc_no:
                return acc
        raise InvalidAccountException("Account not found.")
    def get_account_balance(self, acc_no):
        return self.get_account_by_number(acc_no).acc_balance
    def deposit(self, acc_no, amount):
        acc = self.get_account_by_number(acc_no)
        acc.acc_balance += amount
        return acc.acc_balance
    def withdraw(self, acc_no, amount):
        acc = self.get_account_by_number(acc_no)
        if acc.acc_type == "Savings" and acc.acc_balance - amount < 500:
            raise InsufficientFundException("Minimum ₹500 must be maintained.")
        elif acc.acc_balance >= amount:
            acc.acc_balance -= amount
        else:
            raise InsufficientFundException("Insufficient funds.")
        return acc.acc_balance
    def transfer(self, from_acc, to_acc, amount):
        self.withdraw(from_acc, amount)
        self.deposit(to_acc, amount)
    def get_account_details(self, acc_no):
        acc = self.get_account_by_number(acc_no)
        acc.display_account_info()
```

```

1
2 # customer_service_impl.py
3
4 from dao.interfaces.icustomer_service_provider import ICustomerServiceProvider
5 from exception.exception_module import InvalidAccountException, InsufficientFundException
6
7 class CustomerServiceProviderImpl(ICustomerServiceProvider):
8     Tabnine | Edit | Test | Explain | Document
9     def __init__(self):
10         self.account_list = []
11
12     Tabnine | Edit | Test | Explain | Document
13     def get_account_by_number(self, acc_no):
14         for acc in self.account_list:
15             if acc.acc_no == acc_no:
16                 return acc
17             raise InvalidAccountException("Account not found.")
18
19     Tabnine | Edit | Test | Explain | Document
20     def get_account_balance(self, acc_no):
21         return self.get_account_by_number(acc_no).acc_balance
22
23     Tabnine | Edit | Test | Explain | Document
24     def deposit(self, acc_no, amount):
25         acc = self.get_account_by_number(acc_no)
26         acc.acc_balance += amount
27         return acc.acc_balance
28
29     Tabnine | Edit | Test | Explain | Document
30     def withdraw(self, acc_no, amount):
31         acc = self.get_account_by_number(acc_no)
32         if acc.acc_type == "Savings" and acc.acc_balance - amount < 500:
33             raise InsufficientFundException("Minimum ₹500 must be maintained.")
34         elif acc.acc_balance >= amount:
35             acc.acc_balance -= amount
36         else:
37             raise InsufficientFundException("Insufficient funds.")
38         return acc.acc_balance
39
40     Tabnine | Edit | Test | Explain | Document
41     def transfer(self, from_acc, to_acc, amount):
42         self.withdraw(from_acc, amount)
43         self.deposit(to_acc, amount)
44
45     Tabnine | Edit | Test | Explain | Document
46     def get_account_details(self, acc_no):
47         acc = self.get_account_by_number(acc_no)
48         acc.display_account_info()
49
50

```

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

**Code:**

**# bank\_service\_impl.py file**

```

from dao.implementations.customer_service_impl import CustomerServiceProviderImpl
from dao.interfaces.ibank_service_provider import IBankServiceProvider
from entity.account_types import SavingsAccount, CurrentAccount, ZeroBalanceAccount
class BankServiceProviderImpl(CustomerServiceProviderImpl, IBankServiceProvider):
    def __init__(self, branch_name, branch_address):
        super().__init__()
        self.branch_name = branch_name
        self.branch_address = branch_address

```

```

def create_account(self, customer, acc_type, balance=0.0):
    if acc_type == "savings":
        acc = SavingsAccount(balance, customer)
    elif acc_type == "current":
        acc = CurrentAccount(balance, customer)
    elif acc_type == "zerobalance":
        acc = ZeroBalanceAccount(customer)
    else:
        raise ValueError("Invalid account type")
    self.account_list.append(acc)
    print(f"Account created. Account Number: {acc.acc_no}")

def list_accounts(self):
    for acc in self.account_list:
        acc.display_account_info()

def calculate_interest(self):
    for acc in self.account_list:
        if acc.acc_type == "Savings":
            interest = acc.calculate_interest()
            print(f"Account {acc.acc_no} interest: ₹{interest:.2f}")

```

```

1  # bank_service_impl.py
2
3  from dao.implementations.customer_service_impl import CustomerServiceProviderImpl
4  from dao.interfaces.ibank_service_provider import IBankServiceProvider
5  from entity.account_types import SavingsAccount, CurrentAccount, ZeroBalanceAccount
6
7  class BankServiceProviderImpl(CustomerServiceProviderImpl, IBankServiceProvider):
8      Tabnine | Edit | Test | Explain | Document
9      def __init__(self, branch_name, branch_address):
10         super().__init__()
11         self.branch_name = branch_name
12         self.branch_address = branch_address
13
14     Tabnine | Edit | Test | Explain | Document
15     def create_account(self, customer, acc_type, balance=0.0):
16         if acc_type == "savings":
17             acc = SavingsAccount(balance, customer)
18         elif acc_type == "current":
19             acc = CurrentAccount(balance, customer)
20         elif acc_type == "zerobalance":
21             acc = ZeroBalanceAccount(customer)
22         else:
23             raise ValueError("Invalid account type")
24         self.account_list.append(acc)
25         print(f"Account created. Account Number: {acc.acc_no}")
26
27     Tabnine | Edit | Test | Explain | Document
28     def list_accounts(self):
29         for acc in self.account_list:
30             acc.display_account_info()
31
32     Tabnine | Edit | Test | Explain | Document
33     def calculate_interest(self):
34         for acc in self.account_list:
35             if acc.acc_type == "Savings":
36                 interest = acc.calculate_interest()
37                 print(f"Account {acc.acc_no} interest: ₹{interest:.2f}")
38
39

```

**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 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.

**Code:**

**# ibank\_repository.py file**

```
from abc import ABC, abstractmethod

class IBankRepository(ABC):

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

    @abstractmethod
    def get_account_balance(self, acc_no): pass

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

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

    @abstractmethod
    def transfer(self, from_acc, to_acc, amount): pass

    @abstractmethod
    def get_account_details(self, acc_no): pass
```

```

2
3 from abc import ABC, abstractmethod
4
5 class IBankRepository(ABC):
6     Tabnine | Edit | Test | Explain | Document
7     @abstractmethod
8     def create_account(self, customer, acc_type, balance): pass
9
10    Tabnine | Edit | Test | Explain | Document
11    @abstractmethod
12    def get_account_balance(self, acc_no): pass
13
14    Tabnine | Edit | Test | Explain | Document
15    @abstractmethod
16    def deposit(self, acc_no, amount): pass
17
18    Tabnine | Edit | Test | Explain | Document
19    @abstractmethod
20    def withdraw(self, acc_no, amount): pass
21
22    Tabnine | Edit | Test | Explain | Document
23    @abstractmethod
24    def transfer(self, from_acc, to_acc, amount): pass
25
26    Tabnine | Edit | Test | Explain | Document
27    @abstractmethod
28    def get_account_details(self, acc_no): pass
29

```

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

**Code:**

**# bank\_repository\_impl.py file**

```

import sqlite3

from datetime import datetime

from util.db_util import DBUtil

from dao.interfaces.ibank_repository import IBankRepository

from entity.customer import Customer

from exception.exception_module import InvalidAccountException,
InsufficientFundException

class BankRepositoryImpl(IBankRepository):

    def create_account(self, customer, acc_type, balance):

        conn = DBUtil.get_connection()

        cur = conn.cursor()

        try:

            cur.execute("INSERT INTO customers VALUES (?, ?, ?, ?, ?, ?)",

                        (customer.customer_id, customer.first_name, customer.last_name,

                         customer.email, customer.phone, customer.address))

```

```

cur.execute("SELECT MAX(acc_no) FROM accounts")
result = cur.fetchone()
acc_no = (result[0] or 1000) + 1
cur.execute("INSERT INTO accounts VALUES (?, ?, ?, ?)",
            (acc_no, customer.customer_id, acc_type, balance))
conn.commit()
print(f'Account created. Account Number: {acc_no}')
except Exception as e:
    conn.rollback()
    print(f'Error: {e}')
finally:
    conn.close()

def get_account_balance(self, acc_no):
    conn = DBUtil.get_connection()
    cur = conn.cursor()
    cur.execute("SELECT acc_balance FROM accounts WHERE acc_no = ?", (acc_no,))
    result = cur.fetchone()
    conn.close()
    if result:
        return result[0]
    else:
        raise InvalidAccountException("Account not found")

def deposit(self, acc_no, amount):
    conn = DBUtil.get_connection()
    cur = conn.cursor()
    cur.execute("UPDATE accounts SET acc_balance = acc_balance + ? WHERE acc_no =
?", (amount, acc_no))
    if cur.rowcount == 0:
        raise InvalidAccountException("Invalid account number")
    cur.execute("""INSERT INTO transactions (acc_no, description, transaction_type,
transaction_amount, transaction_date)
                VALUES (?, 'Deposit', 'deposit', ?, ?)""",
                (acc_no, amount, datetime.now().isoformat()))

```

```

        conn.commit()

        conn.close()

    def withdraw(self, acc_no, amount):

        conn = DBUtil.get_connection()

        cur = conn.cursor()

        cur.execute("SELECT acc_balance FROM accounts WHERE acc_no = ?", (acc_no,))

        result = cur.fetchone()

        if not result:

            raise InvalidAccountException("Account not found")

        if result[0] < amount:

            raise InsufficientFundException("Insufficient funds")

        cur.execute("UPDATE accounts SET acc_balance = acc_balance - ? WHERE acc_no =
?", (amount, acc_no))

        cur.execute("""INSERT INTO transactions (acc_no, description, transaction_type,
transaction_amount, transaction_date)

            VALUES (?, 'Withdrawal', 'withdraw', ?, ?)""",

            (acc_no, amount, datetime.now().isoformat()))

        conn.commit()

        conn.close()

    def transfer(self, from_acc, to_acc, amount):

        self.withdraw(from_acc, amount)

        self.deposit(to_acc, amount)

    def get_account_details(self, acc_no):

        conn = DBUtil.get_connection()

        cur = conn.cursor()

        cur.execute("""

            SELECT c.*, a.acc_no, a.acc_type, a.acc_balance

            FROM customers c JOIN accounts a ON c.customer_id = a.customer_id

            WHERE a.acc_no = ?

            """, (acc_no,))

        row = cur.fetchone()

        conn.close()

        if row:

```

```

print(f"\n--- Account Details ---")

print(f"Customer ID   : {row[0]}")

print(f"Name           : {row[1]} {row[2]}")

print(f"Email            : {row[3]}")

print(f"Phone            : {row[4]}")

print(f"Address           : {row[5]}")

print(f"Account No       : {row[6]}")

print(f"Account Type     : {row[7]}")

print(f"Account Balance: ₹{row[8]:.2f}")

```

else:

```

raise InvalidAccountException("Account not found")

```

```

10 class BankRepositoryImpl(IBankRepository):
    Tabnine | Edit | Test | Explain | Document
11     def create_account(self, customer, acc_type, balance):
12         conn = DBUtil.get_connection()
13         cur = conn.cursor()
14         try:
15             cur.execute("INSERT INTO customers VALUES (?, ?, ?, ?, ?, ?)",
16                         (customer.customer_id, customer.first_name, customer.last_name,
17                          customer.email, customer.phone, customer.address))
18             cur.execute("SELECT MAX(acc_no) FROM accounts")
19             result = cur.fetchone()
20             acc_no = (result[0] or 1000) + 1
21             cur.execute("INSERT INTO accounts VALUES (?, ?, ?, ?)",
22                         (acc_no, customer.customer_id, acc_type, balance))
23             conn.commit()
24             print(f" Account created. Account Number: {acc_no}")
25         except Exception as e:
26             conn.rollback()
27             print(f" Error: {e}")
28         finally:
29             conn.close()
30
31     Tabnine | Edit | Test | Explain | Document
32     def get_account_balance(self, acc_no):
33         conn = DBUtil.get_connection()
34         cur = conn.cursor()
35         cur.execute("SELECT acc_balance FROM accounts WHERE acc_no = ?", (acc_no,))
36         result = cur.fetchone()
37         conn.close()
38         if result:
39             return result[0]
40         else:
41             raise InvalidAccountException("Account not found")
42
43     Tabnine | Edit | Test | Explain | Document
44     def deposit(self, acc_no, amount):
45         conn = DBUtil.get_connection()
46         cur = conn.cursor()
47         cur.execute("UPDATE accounts SET acc_balance = acc_balance + ? WHERE acc_no = ?", (amount, acc_no))
48         if cur.rowcount == 0:
49             raise InvalidAccountException("Invalid account number")
50         cur.execute("""INSERT INTO transactions (acc_no, description, transaction_type, transaction_amount, transaction_date)
51                     VALUES (?, 'Deposit', 'deposit', ?, ?)""",
52                     (acc_no, amount, datetime.now().isoformat()))
53         conn.commit()
54         conn.close()

```



```

Tabnine | Edit | Test | Explain | Document
54 def withdraw(self, acc_no, amount):
55     conn = DBUtil.get_connection()
56     cur = conn.cursor()
57     cur.execute("SELECT acc_balance FROM accounts WHERE acc_no = ?", (acc_no,))
58     result = cur.fetchone()
59     if not result:
60         raise InvalidAccountException("Account not found")
61     if result[0] < amount:
62         raise InsufficientFundException("Insufficient funds")
63     cur.execute("UPDATE accounts SET acc_balance = acc_balance - ? WHERE acc_no = ?", (amount, acc_no))
64     cur.execute("""INSERT INTO transactions (acc_no, description, transaction_type, transaction_amount, transaction_date)
65                 VALUES (?, 'Withdrawal', 'withdraw', ?, ?)""",
66                 (acc_no, amount, datetime.now().isoformat()))
67     conn.commit()
68     conn.close()
69
Tabnine | Edit | Test | Explain | Document
70 def transfer(self, from_acc, to_acc, amount):
71     self.withdraw(from_acc, amount)
72     self.deposit(to_acc, amount)
73
Tabnine | Edit | Test | Explain | Document
74 def get_account_details(self, acc_no):
75     conn = DBUtil.get_connection()
76     cur = conn.cursor()
77     cur.execute("""
78         SELECT c.*, a.acc_no, a.acc_type, a.acc_balance
79         FROM customers c JOIN accounts a ON c.customer_id = a.customer_id
80         WHERE a.acc_no = ?
81         """, (acc_no,))
82     row = cur.fetchone()
83     conn.close()
84     if row:
85         print(f"\n--- Account Details ---")
86         print(f"Customer ID : {row[0]}")
87         print(f"Name : {row[1]} {row[2]}")
88         print(f"Email : {row[3]}")
89         print(f"Phone : {row[4]}")
90         print(f"Address : {row[5]}")
91         print(f"Account No : {row[6]}")
92         print(f"Account Type : {row[7]}")
93         print(f"Account Balance: ₹{row[8]:.2f}")
94     else:
95         raise InvalidAccountException("Account not found")

```

## 11. Create DBUtil class and add the following method. • static getDBConn():Connection Establish a connection to the database and return Connection reference

**Code:**

# db\_util.py file

```
import sqlite3
```

```
class DBUtil:
```

```
    @staticmethod
```

```
    def get_connection():
```

```
        conn = sqlite3.connect("HMBank.db")
```

```
        cur = conn.cursor()
```

```
        cur.execute("""
```

```
            create table if not exists customers (
```

```
                customer_id integer primary key,
```

```
                first_name text,
```

```
                last_name text,
```

```

        email text,
        phone text,
        address text
    )
    """
cur.execute("""
    create table if not exists accounts (
        acc_no integer primary key,
        customer_id int,
        acc_type text,
        acc_balance real,
        foreign key (customer_id) references customers(customer_id)
    )
    """)
cur.execute("""
    create table if not exists transactions (
        transaction_id integer primary key autoincrement,
        acc_no int,
        description text,
        transaction_type text,
        transaction_amount real,
        transaction_date text,
        foreign key (acc_no) references accounts(acc_no)
    )
    """)
conn.commit()
return conn

```

```

1  # db_util.py
2
3  import sqlite3
4
5  class DBUtil:
6      Tabnine | Edit | Test | Explain | Document
7      @staticmethod
8      def get_connection():
9          conn = sqlite3.connect("HMBank.db")
10         cur = conn.cursor()
11         cur.execute("""
12             create table if not exists customers (
13                 customer_id integer primary key,
14                 first_name text,
15                 last_name text,
16                 email text,
17                 phone text,
18                 address text
19             )
20         """)
21         cur.execute("""
22             create table if not exists accounts (
23                 acc_no integer primary key,
24                 customer_id int,
25                 acc_type text,
26                 acc_balance real,
27                 foreign key (customer_id) references customers(customer_id)
28             )
29         """)
30         cur.execute("""
31             create table if not exists transactions (
32                 transaction_id integer primary key autoincrement,
33                 acc_no int,
34                 description text,
35                 transaction_type text,
36                 transaction_amount real,
37                 transaction_date text,
38                 foreign key (acc_no) references accounts(acc_no)
39             )
40         """)
41         conn.commit()
42         return conn

```

**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.

**Code:**

**# bank\_app.py file**

```
from entity.customer import Customer
```

```
from dao.implementations.bank_repository_impl import BankRepositoryImpl
```

```
def main():
```

```
    bank = BankRepositoryImpl()
```

```
    while True:
```

```
        print("\n===== HMBank Menu =====")
```

```
        print("1. Create Account")
```

```
        print("2. Deposit")
```

```
print("3. Withdraw")
print("4. Transfer")
print("5. Get Balance")
print("6. Get Account Details")
print("7. Exit")
choice = input("Enter your choice: ")
try:
    if choice == "1":
        cid = int(input("Customer ID: "))
        fname = input("First Name: ")
        lname = input("Last Name: ")
        email = input("Email: ")
        phone = input("Phone: ")
        address = input("Address: ")
        acc_type = input("Account Type (savings/current/zerobalance): ").lower()
        balance = 0.0 if acc_type == "zerobalance" else float(input("Initial Balance: "))
        customer = Customer(cid, fname, lname, email, phone, address)
        bank.create_account(customer, acc_type, balance)
    elif choice == "2":
        acc_no = int(input("Account Number: "))
        amt = float(input("Deposit Amount: "))
        bank.deposit(acc_no, amt)
        print(" Deposit successful.")
    elif choice == "3":
        acc_no = int(input("Account Number: "))
        amt = float(input("Withdraw Amount: "))
        bank.withdraw(acc_no, amt)
        print(" Withdrawal successful.")
    elif choice == "4":
        from_acc = int(input("From Account Number: "))
        to_acc = int(input("To Account Number: "))
        amt = float(input("Transfer Amount: "))
        bank.transfer(from_acc, to_acc, amt)
```

```

        print(" Transfer successful.")

    elif choice == "5":

        acc_no = int(input("Account Number: "))

        balance = bank.get_account_balance(acc_no)

        print(f" Current Balance: ₹{balance:.2f}")

    elif choice == "6":

        acc_no = int(input("Account Number: "))

        bank.get_account_details(acc_no)

    elif choice == "7":

        print(" Thank you for using HMBank!")

        break

    else:

        print(" Invalid option. Try again.")

except Exception as e:

    print(f" Error: {e}")

if __name__ == "__main__":

    main()

```

```

6  def main():
7      bank = BankRepositoryImpl()
8
9      while True:
10         print("\n===== HMBank Menu =====")
11         print("1. Create Account")
12         print("2. Deposit")
13         print("3. Withdraw")
14         print("4. Transfer")
15         print("5. Get Balance")
16         print("6. Get Account Details")
17         print("7. Exit")
18
19         choice = input("Enter your choice: ")
20
21         try:
22             if choice == "1":
23                 cid = int(input("Customer ID: "))
24                 fname = input("First Name: ")
25                 lname = input("Last Name: ")
26                 email = input("Email: ")
27                 phone = input("Phone: ")
28                 address = input("Address: ")
29                 acc_type = input("Account Type (savings/current/zerobalance): ").lower()
30                 balance = 0.0 if acc_type == "zerobalance" else float(input("Initial Balance: "))
31                 customer = Customer(cid, fname, lname, email, phone, address)
32                 bank.create_account(customer, acc_type, balance)
33
34             elif choice == "2":
35                 acc_no = int(input("Account Number: "))
36                 amt = float(input("Deposit Amount: "))
37                 bank.deposit(acc_no, amt)
38                 print(" Deposit successful.")
39
40             elif choice == "3":
41                 acc_no = int(input("Account Number: "))
42                 amt = float(input("Withdraw Amount: "))
43                 bank.withdraw(acc_no, amt)
44                 print(" Withdrawal successful.")
45
46             elif choice == "4":
47                 from_acc = int(input("From Account Number: "))
48                 to_acc = int(input("To Account Number: "))
49                 amt = float(input("Transfer Amount: "))
50                 bank.transfer(from_acc, to_acc, amt)
51                 print(" Transfer successful.")

```

```

52
53         elif choice == "5":
54             acc_no = int(input("Account Number: "))
55             balance = bank.get_account_balance(acc_no)
56             print(f" Current Balance: ₹{balance:.2f}")
57
58         elif choice == "6":
59             acc_no = int(input("Account Number: "))
60             bank.get_account_details(acc_no)
61
62         elif choice == "7":
63             print(" Thank you for using HMBank!")
64             break
65
66         else:
67             print(" Invalid option. Try again.")
68
69     except Exception as e:
70         print(f" Error: {e}")
71
72 if __name__ == "__main__":
73     main()
74

```

**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.**

HMBank/

```

├── entity/
│   ├── customer.py
│   ├── account.py
│   ├── account_types.py
│   └── transaction.py
├── dao/
│   ├── interfaces/
│   │   ├── icustomer_service_provider.py
│   │   ├── ibank_service_provider.py
│   │   └── ibank_repository.py
│   └── implementations/
│       ├── customer_service_impl.py
│       ├── bank_service_impl.py
│       └── bank_repository_impl.py
├── util/
│   └── db_util.py
├── exception/
│   └── exception_module.py
├── main/
│   └── bank_app.py

```

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

**# exception\_module.py file**

```
class InvalidAccountException(Exception):  
    def __init__(self, message="Invalid account number."):  
        super().__init__(message)  
  
class InsufficientFundException(Exception):  
    def __init__(self, message="Insufficient funds."):  
        super().__init__(message)  
  
class OverDraftLimitExceededException(Exception):  
    def __init__(self, message="Overdraft limit exceeded."):  
        super().__init__(message)
```

```
1  # exception_module.py  
2  
3  class InvalidAccountException(Exception):  
4      Tabnine | Edit | Test | Explain | Document  
5      def __init__(self, message="Invalid account number."):  
6          super().__init__(message)  
7  
8  class InsufficientFundException(Exception):  
9      Tabnine | Edit | Test | Explain | Document  
10     def __init__(self, message="Insufficient funds."):  
11         super().__init__(message)  
12  
13 class OverDraftLimitExceededException(Exception):  
14     Tabnine | Edit | Test | Explain | Document  
15     def __init__(self, message="Overdraft limit exceeded."):  
16         super().__init__(message)
```

```
PS E:\banking_system> python -m main.bank_app  
>>
```

```
===== HMBank Menu =====  
1. Create Account  
2. Deposit  
3. Withdraw  
4. Transfer  
5. Get Balance  
6. Get Account Details  
7. Exit  
Enter your choice: 3  
Account Number: 1001  
Withdraw Amount: 1000000  
Error: Insufficient funds
```

```
PS E:\banking_system> python -m main.bank_app
>>

===== HMBank Menu =====
1. Create Account
2. Deposit
3. Withdraw
4. Transfer
5. Get Balance
6. Get Account Details
7. Exit
Enter your choice: 3
Account Number: 1002
Withdraw Amount: 3000
Error: Account not found

===== HMBank Menu =====
1. Create Account
2. Deposit
3. Withdraw
4. Transfer
5. Get Balance
6. Get Account Details
7. Exit
Enter your choice: 1
Customer ID: 2
First Name: ibrahim
Last Name: sheriff
Email: re@email.com
Phone: 1234567980
Address: 12west
Account Type (savings/current/zerobalance): savings
Initial Balance: 30000
Account created. Account Number: 1002

===== HMBank Menu =====
1. Create Account
2. Deposit
3. Withdraw
4. Transfer
5. Get Balance
6. Get Account Details
7. Exit
Enter your choice: 4
From Account Number: 1001
To Account Number: 1002
Transfer Amount: 5000
Error: Insufficient funds
```