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)
```

```python
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")

# Example
credit_score = int(input("Enter credit score: "))
annual_income = float(input("Enter annual income: "))

check_loan_eligibility(credit_score, annual_income)
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
              ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
KeyboardInterrupt
PS E:\banking_system> & C:/Users/Lenovo/AppData/Local/Programs/Python/Python312/pytho
n.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()
```

```python
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()
```

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL    PORTS

PS E:\banking_system> & C:/Users/Lenovo/AppData/Local/Programs/Python/Python312/python.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: future_balance = initial_balance * (1 + annual_interest_rate/100)^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()
```

```
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()
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
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()
```

```
accounts= {
  101:5000.00,
  102:90000.00,
  103:400.00
}
Tabnine | Edit | Test | Explain | Document
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()
```

```
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\none 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()
```

```
import re

accounts= {
  101:5000.00,
  102:90000.00,
  103:400.00
}
Tabnine | Edit | Test | Explain | Document
def pwd():
  acc_no = int(input("Enter account number: "))
  if acc_no in accounts:
    print("password must have\natleast 8 characters\none 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")
```

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()
```

```python
transaction = []

Tabnine | Edit | Test | Explain | Document
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}")
```

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}")
```

```python
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")
```

```python
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/Prog
---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
1.show details
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}")
```

```
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}")
```

```python
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")
```

```python
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}")
```

```python
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 (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)
    Tabnine | Edit | Test | Explain | Document
    def show_details(self):
        print("Customer Details : \n")
        self.customer.display_customer_info()
        print("\nAccount Details\n")
        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!")
```

```
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 depositted \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()
```

```
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

  Tabnine | Edit | Test | Explain | Document
  @property
  def name(self):
    return f"{self.first_name}{self.last_name}"

  @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}")
```

```python
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}")
```

```python
class BankAccount(ABC):
    Tabnine | Edit | Test | Explain | Document
    def __init__(self, account_num, account_balance):
        self.account_num = account_num
        self.account_balance = float(account_balance)

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

    Tabnine | Edit | Test | Explain | Document
    @abstractmethod
    def deposit(self,amount):
        pass

    Tabnine | Edit | Test | Explain | Document
    @abstractmethod
    def withdraw(self,amount):
        pass

    Tabnine | Edit | Test | Explain | Document
    @abstractmethod
    def calculate_interest(self):
        pass
```

```python
class Savingsaccount(BankAccount):
    Tabnine | Edit | Test | Explain | Document
    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

    Tabnine | Edit | Test | Explain | Document
    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}")

    Tabnine | Edit | Test | Explain | Document
    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}")

    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(BankAccount):
    Tabnine | Edit | Test | Explain | Document
    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

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

    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")
```

```
PS E:\banking_system> & C:/Users/Lenovo/AppData/Local/Programs/Pyth
---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()
```

```python
import re

#  Customer Class
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}")


```

```python
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

    Tabnine | Edit | Test | Explain | Document
    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()

    Tabnine | Edit | Test | Explain | Document
    @abstractmethod
    def deposit(self, amount): pass

    Tabnine | Edit | Test | Explain | Document
    @abstractmethod
    def withdraw(self, amount): pass

    Tabnine | Edit | Test | Explain | Document
    @abstractmethod
    def calculate_interest(self): pass


#SavingsAccount Class
class Savingsaccount(BankAccount):
    Tabnine | Edit | Test | Explain | Document
    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

    Tabnine | Edit | Test | Explain | Document
    def deposit(self, amount):
        self.account_balance += amount
        print(f"₹{amount:.2f} deposited. New Balance: ₹{self.account_balance:.2f}")

    Tabnine | Edit | Test | Explain | Document
    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!")

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

```python
# Main Bank Application
Tabnine | Edit | Test | Explain | Document
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)
```

```python
        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()
```

```
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()
```

```python
from abc import ABC, abstractmethod


#  Customer Class
class Customer:
    Tabnine | Edit | Test | Explain | Document
    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

    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}")


#  Account Class
class Account:
    last_acc_no = 1000

    Tabnine | Edit | Test | Explain | Document
    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

    Tabnine | Edit | Test | Explain | Document
    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()
```

```python
# ZeroBalanceAccount Class
class ZeroBalanceAccount(Account):
    Tabnine | Edit | Test | Explain | Document
    def __init__(self, customer):
        super().__init__("ZeroBalance", 0.0, customer)


#  Interfaces
class ICustomerServiceProvider(ABC):
    Tabnine | Edit | Test | Explain | Document
    @abstractmethod
    def get_account_balance(self, acc_no): pass

    Tabnine | Edit | Test | Explain | Document
    @abstractmethod
    def deposit(self, acc_no, amount): pass

    Tabnine | Edit | Test | Explain | Document
    @abstractmethod
    def withdraw(self, acc_no, amount): pass

    Tabnine | Edit | Test | Explain | Document
    @abstractmethod
    def transfer(self, from_acc, to_acc, amount): pass

    Tabnine | Edit | Test | Explain | Document
    @abstractmethod
    def get_account_details(self, acc_no): pass


class IBankServiceProvider(ABC):
    Tabnine | Edit | Test | Explain | Document
    @abstractmethod
    def create_account(self, customer, acc_type, balance): pass

    Tabnine | Edit | Test | Explain | Document
    @abstractmethod
    def list_accounts(self): pass

    Tabnine | Edit | Test | Explain | Document
    @abstractmethod
    def calculate_interest(self): pass

```

```python
#  Account Class
class Account:
    last_acc_no = 1000

    Tabnine | Edit | Test | Explain | Document
    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

    Tabnine | Edit | Test | Explain | Document
    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):
    Tabnine | Edit | Test | Explain | Document
    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

    Tabnine | Edit | Test | Explain | Document
    def calculate_interest(self):
        interest = self.acc_balance * (self.interest_rate / 100)
        return interest


# CurrentAccount Class
class CurrentAccount(Account):
    Tabnine | Edit | Test | Explain | Document
    def __init__(self, acc_balance, customer, overdraft_limit=5000):
        super().__init__("Current", acc_balance, customer)
        self.overdraft_limit = overdraft_limit

    Tabnine | Edit | Test | Explain | Document
    def withdraw(self, amount):
        if amount <= self.acc_balance + self.overdraft_limit:
            self.acc_balance -= amount
        else:
            raise Exception("Overdraft limit exceeded")
```

```python
class CustomerServiceProviderImpl(ICustomerServiceProvider):
    Tabnine | Edit | Test | Explain | Document
    def __init__(self):
        self.account_list = []

    Tabnine | Edit | Test | Explain | Document
    def get_account_by_number(self, acc_no):
        for acc in self.account_list:
            if acc.acc_no == acc_no:
                return acc
        return None

    Tabnine | Edit | Test | Explain | Document
    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.")

    Tabnine | Edit | Test | Explain | Document
    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.")

    Tabnine | Edit | Test | Explain | Document
    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

    Tabnine | Edit | Test | Explain | Document
    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)
```

```python
    Tabnine | Edit | Test | Explain | Document
    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):
    Tabnine | Edit | Test | Explain | Document
    def __init__(self, branch_name, branch_address):
        super().__init__()
        self.branch_name = branch_name
        self.branch_address = branch_address

    Tabnine | Edit | Test | Explain | Document
    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}")

    Tabnine | Edit | Test | Explain | Document
    def list_accounts(self):
        for acc in self.account_list:
            acc.display_account_info()

    Tabnine | Edit | Test | Explain | Document
    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}")

```

```
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. OverDraftLimitExcededException thow 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()
```

```python
from abc import ABC, abstractmethod

# Custom Exceptions
class InsufficientFundException(Exception):
    pass

class InvalidAccountException(Exception):
    pass

class OverDraftLimitExceededException(Exception):
    pass

#  Customer Class
class Customer:
    Tabnine | Edit | Test | Explain | Document
    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

    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}")


#  Account Class
class Account:
    last_acc_no = 1000

    Tabnine | Edit | Test | Explain | Document
    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

    Tabnine | Edit | Test | Explain | Document
    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()
```

```python
#  SavingsAccount Class
class SavingsAccount(Account):
    Tabnine | Edit | Test | Explain | Document
    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

    Tabnine | Edit | Test | Explain | Document
    def calculate_interest(self):
        return self.acc_balance * (self.interest_rate / 100)


#  CurrentAccount Class
class CurrentAccount(Account):
    Tabnine | Edit | Test | Explain | Document
    def __init__(self, acc_balance, customer, overdraft_limit=5000):
        super().__init__("Current", acc_balance, customer)
        self.overdraft_limit = overdraft_limit

    Tabnine | Edit | Test | Explain | Document
    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):
    Tabnine | Edit | Test | Explain | Document
    def __init__(self, customer):
        super().__init__("ZeroBalance", 0.0, customer)
```

```python
# Interfaces
class ICustomerServiceProvider(ABC):
    Tabnine | Edit | Test | Explain | Document
    @abstractmethod
    def get_account_balance(self, acc_no): pass

    Tabnine | Edit | Test | Explain | Document
    @abstractmethod
    def deposit(self, acc_no, amount): pass

    Tabnine | Edit | Test | Explain | Document
    @abstractmethod
    def withdraw(self, acc_no, amount): pass

    Tabnine | Edit | Test | Explain | Document
    @abstractmethod
    def transfer(self, from_acc, to_acc, amount): pass

    Tabnine | Edit | Test | Explain | Document
    @abstractmethod
    def get_account_details(self, acc_no): pass


class IBankServiceProvider(ABC):
    Tabnine | Edit | Test | Explain | Document
    @abstractmethod
    def create_account(self, customer, acc_type, balance): pass

    Tabnine | Edit | Test | Explain | Document
    @abstractmethod
    def list_accounts(self): pass

    Tabnine | Edit | Test | Explain | Document
    @abstractmethod
    def calculate_interest(self): pass
```

```python
# Service Implementations
class CustomerServiceProviderImpl(ICustomerServiceProvider):
    Tabnine | Edit | Test | Explain | Document
    def __init__(self):
        self.account_list = []

    Tabnine | Edit | Test | Explain | Document
    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.")

    Tabnine | Edit | Test | Explain | Document
    def get_account_balance(self, acc_no):
        acc = self.get_account_by_number(acc_no)
        return acc.acc_balance

    Tabnine | Edit | Test | Explain | Document
    def deposit(self, acc_no, amount):
        acc = self.get_account_by_number(acc_no)
        acc.acc_balance += amount
        return acc.acc_balance

    Tabnine | Edit | Test | Explain | Document
    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

    Tabnine | Edit | Test | Explain | Document
    def transfer(self, from_acc, to_acc, amount):
        self.withdraw(from_acc, amount)
        self.deposit(to_acc, amount)

    Tabnine | Edit | Test | Explain | Document
    def get_account_details(self, acc_no):
        acc = self.get_account_by_number(acc_no)
        acc.display_account_info()
```

```python
class BankServiceProviderImpl(CustomerServiceProviderImpl, IBankServiceProvider):
    Tabnine | Edit | Test | Explain | Document
    def __init__(self, branch_name, branch_address):
        super().__init__()
        self.branch_name = branch_name
        self.branch_address = branch_address

    Tabnine | Edit | Test | Explain | Document
    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}")

    Tabnine | Edit | Test | Explain | Document
    def list_accounts(self):
        for acc in self.account_list:
            acc.display_account_info()

    Tabnine | Edit | Test | Explain | Document
    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}")
```

```python
# Main App
Tabnine | Edit | Test | Explain | Document
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}")
```

```python
            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()
```

```
====== 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()

```python
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()
```

```python
    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

# Custom Exceptions
class InsufficientFundException(Exception):
    pass

class InvalidAccountException(Exception):
    pass

class OverDraftLimitExceededException(Exception):
    pass

#  Customer Class
class Customer:
    Tabnine | Edit | Test | Explain | Document
    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

    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}")
```

```python
#  Account Class
class Account:
    last_acc_no = 1000

    Tabnine | Edit | Test | Explain | Document
    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

    Tabnine | Edit | Test | Explain | Document
    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):
    Tabnine | Edit | Test | Explain | Document
    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

    Tabnine | Edit | Test | Explain | Document
    def calculate_interest(self):
        return self.acc_balance * (self.interest_rate / 100)


#  CurrentAccount Class
class CurrentAccount(Account):
    Tabnine | Edit | Test | Explain | Document
    def __init__(self, acc_balance, customer, overdraft_limit=5000):
        super().__init__("Current", acc_balance, customer)
        self.overdraft_limit = overdraft_limit

    Tabnine | Edit | Test | Explain | Document
    def withdraw(self, amount):
        if amount <= self.acc_balance + self.overdraft_limit:
            self.acc_balance -= amount
        else:
            raise OverDraftLimitExceededException(" Overdraft limit exceeded.")
```

```python
# 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
```

```python
#  Service Implementations
class CustomerServiceProviderImpl(ICustomerServiceProvider):
    Tabnine | Edit | Test | Explain | Document
    def __init__(self):
        self.account_list = []

    Tabnine | Edit | Test | Explain | Document
    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.")

    Tabnine | Edit | Test | Explain | Document
    def get_account_balance(self, acc_no):
        acc = self.get_account_by_number(acc_no)
        return acc.acc_balance

    Tabnine | Edit | Test | Explain | Document
    def deposit(self, acc_no, amount):
        acc = self.get_account_by_number(acc_no)
        acc.acc_balance += amount
        return acc.acc_balance

    Tabnine | Edit | Test | Explain | Document
    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

    Tabnine | Edit | Test | Explain | Document
    def transfer(self, from_acc, to_acc, amount):
        self.withdraw(from_acc, amount)
        self.deposit(to_acc, amount)

    Tabnine | Edit | Test | Explain | Document
    def get_account_details(self, acc_no):
        acc = self.get_account_by_number(acc_no)
        acc.display_account_info()

```

```python
class BankServiceProviderImpl(CustomerServiceProviderImpl, IBankServiceProvider):
    Tabnine | Edit | Test | Explain | Document
    def __init__(self, branch_name, branch_address):
        super().__init__()
        self.branch_name = branch_name
        self.branch_address = branch_address

    Tabnine | Edit | Test | Explain | Document
    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}")

    Tabnine | Edit | Test | Explain | Document
    def list_accounts(self):
        for acc in self.account_list:
            acc.display_account_info()

    Tabnine | Edit | Test | Explain | Document
    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}")

```

```python
# 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}")
```

```python
            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()
```

```
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}")

```python
#  Account
class Account:
    last_acc_no = 1000

    Tabnine | Edit | Test | Explain | Document
    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

    Tabnine | Edit | Test | Explain | Document
    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()

    Tabnine | Edit | Test | Explain | Document
    def __hash__(self):
        return hash(self.acc_no)

    Tabnine | Edit | Test | Explain | Document
    def __eq__(self, other):
        return isinstance(other, Account) and self.acc_no == other.acc_no
```

```python
class BankServiceProviderImpl(CustomerServiceProviderImpl, IBankServiceProvider):
    Tabnine | Edit | Test | Explain | Document
    def __init__(self, branch_name, branch_address):
        super().__init__()
        self.branch_name = branch_name
        self.branch_address = branch_address

    Tabnine | Edit | Test | Explain | Document
    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}")

    Tabnine | Edit | Test | Explain | Document
    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()

    Tabnine | Edit | Test | Explain | Document
    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}")
```

```
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}")
```

```python
# Account Class
class Account:
    last_acc_no = 1000

    Tabnine | Edit | Test | Explain | Document
    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

    Tabnine | Edit | Test | Explain | Document
    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()
```

```python
class BankServiceProviderImpl(CustomerServiceProviderImpl, IBankServiceProvider):
    Tabnine | Edit | Test | Explain | Document
    def __init__(self, branch_name, branch_address):
        super().__init__()
        self.branch_name = branch_name
        self.branch_address = branch_address

    Tabnine | Edit | Test | Explain | Document
    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}")

    Tabnine | Edit | Test | Explain | Document
    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()

    Tabnine | Edit | Test | Explain | Document
    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}")
```

```
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
```

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}")
```

```python
task14 part1.py > ...
# customer.py

class Customer:
    Tabnine | Edit | Test | Explain | Document
    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

    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}")
```

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()
```

```python
class Account:
    last_acc_no = 1000

    Tabnine | Edit | Test | Explain | Document
    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

    Tabnine | Edit | Test | Explain | Document
    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()
```

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}")
```

```python
entity > transaction.py > ...
# transaction.py

from datetime import datetime

class Transaction:
    Tabnine | Edit | Test | Explain | Document
    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()

    Tabnine | Edit | Test | Explain | Document
    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}")

```

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)
```

```
# account_types.py

from entity.account import Account

class SavingsAccount(Account):
    Tabnine | Edit | Test | Explain | Document
    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

    Tabnine | Edit | Test | Explain | Document
    def calculate_interest(self):
        return self.acc_balance * (self.interest_rate / 100)


class CurrentAccount(Account):
    Tabnine | Edit | Test | Explain | Document
    def __init__(self, acc_balance, customer, overdraft_limit=5000):
        super().__init__("Current", acc_balance, customer)
        self.overdraft_limit = overdraft_limit

    Tabnine | Edit | Test | Explain | Document
    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):
    Tabnine | Edit | Test | Explain | Document
    def __init__(self, customer):
        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
```

```
# icustomer_service_provider.py

from abc import ABC, abstractmethod

class ICustomerServiceProvider(ABC):
    Tabnine | Edit | Test | Explain | Document
    @abstractmethod
    def get_account_balance(self, acc_no): pass

    Tabnine | Edit | Test | Explain | Document
    @abstractmethod
    def deposit(self, acc_no, amount): pass

    Tabnine | Edit | Test | Explain | Document
    @abstractmethod
    def withdraw(self, acc_no, amount): pass

    Tabnine | Edit | Test | Explain | Document
    @abstractmethod
    def transfer(self, from_acc, to_acc, amount): pass

    Tabnine | Edit | Test | Explain | Document
    @abstractmethod
    def get_account_details(self, acc_no): pass
```

6. Create IBankServiceProvider interface/abstract class with following functions: • create_account(Customer customer, long accNo, String accType, float balance): Create a new bank account for the given customer with the initial balance. • listAccounts(): Array of BankAccount: List all accounts in the bank.(List[Account] accountsList) • getAccountDetails(account_number: long): Should return the account and customer details. • calculateInterest(): the calculate_interest() method to calculate interest based on the balance and interest rate.

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
```

```
# ibank_service_provider.py

from abc import ABC, abstractmethod

class IBankServiceProvider(ABC):
    Tabnine | Edit | Test | Explain | Document
    @abstractmethod
    def create_account(self, customer, acc_type, balance): pass

    Tabnine | Edit | Test | Explain | Document
    @abstractmethod
    def list_accounts(self): pass

    Tabnine | Edit | Test | Explain | Document
    @abstractmethod
    def calculate_interest(self): pass
```

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()
```

```python

# customer_service_impl.py

from dao.interfaces.icustomer_service_provider import ICustomerServiceProvider
from exception.exception_module import InvalidAccountException, InsufficientFundException

class CustomerServiceProviderImpl(ICustomerServiceProvider):
    Tabnine | Edit | Test | Explain | Document
    def __init__(self):
        self.account_list = []

    Tabnine | Edit | Test | Explain | Document
    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.")

    Tabnine | Edit | Test | Explain | Document
    def get_account_balance(self, acc_no):
        return self.get_account_by_number(acc_no).acc_balance

    Tabnine | Edit | Test | Explain | Document
    def deposit(self, acc_no, amount):
        acc = self.get_account_by_number(acc_no)
        acc.acc_balance += amount
        return acc.acc_balance

    Tabnine | Edit | Test | Explain | Document
    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

    Tabnine | Edit | Test | Explain | Document
    def transfer(self, from_acc, to_acc, amount):
        self.withdraw(from_acc, amount)
        self.deposit(to_acc, amount)

    Tabnine | Edit | Test | Explain | Document
    def get_account_details(self, acc_no):
        acc = self.get_account_by_number(acc_no)
        acc.display_account_info()
```

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}")
```

```python
# bank_service_impl.py

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):
    Tabnine | Edit | Test | Explain | Document
    def __init__(self, branch_name, branch_address):
        super().__init__()
        self.branch_name = branch_name
        self.branch_address = branch_address

    Tabnine | Edit | Test | Explain | Document
    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}")

    Tabnine | Edit | Test | Explain | Document
    def list_accounts(self):
        for acc in self.account_list:
            acc.display_account_info()

    Tabnine | Edit | Test | Explain | Document
    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}")
```

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
```

```python
from abc import ABC, abstractmethod

class IBankRepository(ABC):
    Tabnine | Edit | Test | Explain | Document
    @abstractmethod
    def create_account(self, customer, acc_type, balance): pass

    Tabnine | Edit | Test | Explain | Document
    @abstractmethod
    def get_account_balance(self, acc_no): pass

    Tabnine | Edit | Test | Explain | Document
    @abstractmethod
    def deposit(self, acc_no, amount): pass

    Tabnine | Edit | Test | Explain | Document
    @abstractmethod
    def withdraw(self, acc_no, amount): pass

    Tabnine | Edit | Test | Explain | Document
    @abstractmethod
    def transfer(self, from_acc, to_acc, amount): pass

    Tabnine | Edit | Test | Explain | Document
    @abstractmethod
    def get_account_details(self, acc_no): pass
```

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")
```

```
class BankRepositoryImpl(IBankRepository):
    Tabnine | Edit | Test | Explain | Document
    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()

    Tabnine | Edit | Test | Explain | Document
    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")

    Tabnine | Edit | Test | Explain | Document
    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()
```

```python
Tabnine | Edit | Test | Explain | Document
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()

Tabnine | Edit | Test | Explain | Document
def transfer(self, from_acc, to_acc, amount):
    self.withdraw(from_acc, amount)
    self.deposit(to_acc, amount)

Tabnine | Edit | Test | Explain | Document
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")
```

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
```

```python
# db_util.py

import sqlite3

class DBUtil:
    Tabnine | Edit | Test | Explain | Document
    @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
```

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()
```

```
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.")
```

```python

            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()
```

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)
```

```
# exception_module.py

class InvalidAccountException(Exception):
    Tabnine | Edit | Test | Explain | Document
    def __init__(self, message="Invalid account number."):
        super().__init__(message)

class InsufficientFundException(Exception):
    Tabnine | Edit | Test | Explain | Document
    def __init__(self, message="Insufficient funds."):
        super().__init__(message)

class OverDraftLimitExceededException(Exception):
    Tabnine | Edit | Test | Explain | Document
    def __init__(self, message="Overdraft limit exceeded."):
        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
```