HEXAWARE ASSIGNMENT -1

BANKING SYSTEM

OOPS CONCEPTS

SUBMITTED BY: ANUSHA P

Control Structure

Task 1: Conditional Statements

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

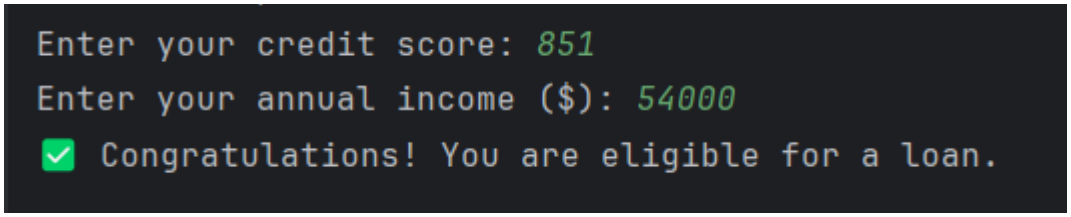
Credit Score must be above 700.

Annual Income must be at least $50,000.

Tasks:

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

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

        if credit_score > 700 and annual_income >= 50000:
            print("Congratulations! You are eligible for a loan.")
        else:
            print("Sorry, you are not eligible for a loan.")

    except ValueError:
        print("⚠Invalid input! Please enter numeric values.")
```



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.

```
while True:
    print("\nBANK SYSTEM MENU")
    print("1. Perform Bank Transactions")
    print("2. Check Loan Eligibility")
    print("3. Calculate Compound Interest")
    print("4. Print Account Details")
    print("5. Exit")

    choice = input("Choose an option: ")

    if choice == "1":
        print("\n--- Bank Transactions ---")
        print("1. Check Balance")
        print("2. Deposit")
        print("3. Withdraw")
        print("4. Calculate Interest (Savings Only)")
        transaction_choice = input("Choose a transaction: ")

        if transaction_choice == "1":
            acc_num = int(input("Enter account number: "))
            bank.check_balance(acc_num)

        elif transaction_choice == "2":
            acc_num = int(input("Enter Account Number: "))
            amount = Decimal(input("Enter deposit amount: "))
            bank.deposit(acc_num, amount)
```

```
    elif transaction_choice == "3":
        acc_num = int(input("Enter Account Number: "))
        amount = Decimal(input("Enter withdrawal amount: "))
        bank.withdraw(acc_num, amount)

    elif transaction_choice == "4":
        acc_num = int(input("Enter Account Number: "))
        bank.calculate_interest(acc_num)

    else:
        print("⚠️ Invalid transaction choice!")
```

```
===== BANK SYSTEM MENU =====
1. Perform Bank Transactions
2. Check Loan Eligibility
3. Calculate Compound Interest
4. Print Account Details
5. Exit
Choose an option: 1

--- Bank Transactions ---
1. Check Balance
2. Deposit
3. Withdraw
4. Calculate Interest (Savings Only)
Choose a transaction: 1
Enter account number: 2
Balance: $25050.00
```

```
Choose a transaction: 2
Enter Account Number: 2
Enter deposit amount: 50
Deposited $50.00. New balance: $25100.00
Account updated successfully!
Deposit successful!
```

```
Choose a transaction: 3
Enter Account Number: 5
Enter withdrawal amount: 50
Enter amount in multiples of 500 or 100!
```

```
Choose a transaction: 4
Enter Account Number: 5
Interest applies only to savings accounts or account not found.
```

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.

```
def calculate_compound_interest(self,initial_balance, annual_interest_rate, years):
    return initial_balance * (1 + annual_interest_rate / 100) ** years

elif choice == "3":
    num_customers = int(input("Enter the number of customers: "))

    for i in range(1, num_customers + 1):
        print(f"\nCustomer {i}:")

        initial_balance = float(input("Enter initial balance: "))
```

annual_interest_rate = float(input("Enter annual interest rate (in %): "))

years = int(input("Enter number of years: "))

future_balance = bank.calculate_compound_interest(initial_balance,annual_interest_rate,years)

print(f"Future balance after {years} years: ₹{future_balance:.2f}")

```
===== BANK SYSTEM MENU =====
1. Perform Bank Transactions
2. Check Loan Eligibility
3. Calculate Compound Interest
4. Print Account Details
5. Exit
Choose an option: 3
Enter the number of customers: 2

Customer 1:
Enter initial balance: 50000
Enter annual interest rate (in %): 10
Enter number of years: 5
Future balance after 5 years: ₹80525.50

Customer 2:
Enter initial balance: 50
Enter annual interest rate (in %): 50
Enter number of years: 3
Future balance after 3 years: ₹168.75
```

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.

```
if transaction_choice == "1":
    while True:
        try:
            acc_num = int(input("Enter account number: "))
            bank.check_balance(acc_num)
            break
        except ValueError as ve:
            print("Enter a valid account number!The account number you have entered is invalid. Please try again.")
        except InvalidAccountException as e:
            print("Enter a valid account number!The account number you have entered is invalid. Please try again.")
```

```
--- Bank Transactions ---
1. Check Balance
2. Deposit
3. Withdraw
4. Calculate Interest (Savings Only)
Choose a transaction: 1
Enter account number: 20
Account does not exist!
Enter a valid account number!The account number you have entered is invalid. Please try again.
Enter account number: 64
Account does not exist!
Enter a valid account number!The account number you have entered is invalid. Please try again.
Enter account number: 2
Balance: $25100.00
```

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.

```
def is_valid_password(self,password,account_number):
    """Validate password based on length, uppercase, and digit conditions."""
    account = self.account_service.get_account(account_number)
    if account:
        if len(password) < 8:
            print("Password must be at least 8 characters long.")
            return False
        if not any(char.isupper() for char in password):
            print("Password must contain at least one uppercase letter.")
            return False
        if not any(char.isdigit() for char in password):
            print("Password must contain at least one digit.")
            return False
        return True
    else:
        print("Account not found!")
        return None
elif choice == "5":
    acc_num =int(input("Enter Account number: "))
    password=input("Enter Password: ")
    res=bank.is_valid_password(password,acc_num)
    if res==True:
        print("Password is valid")
    elif res==False:
        print("Password is invalid")
```

```
Choose an option: 5
Enter Account number: 3
Enter Password: ABc123
Password must be at least 8 characters long.
Password is invalid
```

Task 6: Show transactions

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.

```
def show_transaction_history(self, account_number):
    transactions = self.transaction_dao.get_transactions(account_number)
    if not transactions:
        print("No transactions found.")
    else:
        print(f"\n--- Transaction History for Account {account_number} ---")
        for tran_type, amount, tran_date in transactions:
            print(f"{tran_type.title()} | ₹{amount} | {tran_date} ")
```

```
--- Bank Transactions ---
1. Check Balance
2. Deposit
3. Withdraw
4. Calculate Interest (Savings Only)
5. View Transaction History
Choose a transaction: 5
Enter Account Number: 2

--- Transaction History for Account 2 ---
Withdrawal | ₹500.00 | 2025-03-21 08:22:44
Deposit | ₹100.00 | 2025-04-05 00:42:50
```

Task 7: Class & Object

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

Attributes: Customer ID , First Name , Last Name , Email Address , Phone Number , Address

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

```
class Customer:
    def __init__(self, customer_id=None, first_name=None, last_name=None, email=None, phone=None, address=None):
        """Constructor with default values (Overloaded Constructor)."""
        self.__customer_id = customer_id
        self.__first_name = first_name
        self.__last_name = last_name
        self.__email = email
        self.__phone = phone
        self.__address = address

    # Getters
    def get_customer_id(self):
        return self.__customer_id

    def get_first_name(self):
        return self.__first_name

    def get_last_name(self):
        return self.__last_name

    def get_email(self):
        return self.__email

    def get_phone(self):
        return self.__phone
```

```
    def get_address(self):
        return self.__address

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

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

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

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

    def set_phone(self, phone):
        self.__phone = phone

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

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

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

Constructor and Methods : Implement default constructors and overload the constructor with Account attributes, Generate getter and setter, (print all information of attribute) methods for the attributes.

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%

```
from decimal import Decimal
class Account:
    interest_rate = Decimal(4.5)  # Fixed interest rate

    def __init__(self, account_number=None, account_type=None, balance=0.0):
        self.__account_number = account_number
        self.__account_type = account_type
        self.__balance = balance

    # Getters
    def get_account_number(self):
        return self.__account_number

    def get_account_type(self):
        return self.__account_type

    def get_balance(self):
        return self.__balance

    # Setters
    def set_account_number(self, account_number):
        self.__account_number = account_number

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

    def set_balance(self, balance):
        if balance < 0:
            raise ValueError("Balance cannot be negative!")
        self.__balance = balance

    # Deposit method
    def deposit(self, amount):
        if amount > 0:
```

```
            self.__balance =self.__balance+amount
            print(f"Deposited ${amount:.2f}. New balance: ${self.__balance:.2f}")
        else:
            print("Deposit amount must be greater than zero.")

    # Withdraw method
    def withdraw(self, amount):
        if amount > self.__balance:
            print("Insufficient funds! Withdrawal failed.")
        elif (amount > 0) and (amount % 500 == 0 or amount % 100 == 0):
            self.__balance =self.__balance - amount
            print(f"Withdrawn ${amount:.2f}. Remaining balance: ${self.__balance:.2f}")
        elif (amount<=0):
            print("Withdrawal amount must be greater than zero.")
        else:
            print("Enter amount in multiples of 500 or 100")

    # Calculate interest method
    def calculate_interest(self):
        interest = (self.__balance * Account.interest_rate) / 100
        print(f"Interest earned: ${interest:.2f}")
        return interest
```

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

- create object for account class by calling parameter constructor.
- deposit(amount: float): Deposit the specified amount into the account.
- withdraw(amount: float): Withdraw the specified amount from the account
- calculate_interest(): Calculate and add interest to the account balance for savings accounts.

```
class Bank:
    def __init__(self):
        self.account_service = AccountDaoImpl()
        self.transaction_dao = TransactionDAOImpl()

    def deposit(self, account_number, amount):
        account = self.account_service.get_account(account_number)
        if account:
            account.deposit(amount)
            self.account_service.update_account(account)
            self.transaction_dao.transact(account_number,'deposit',amount)  # Update DB
            print("Deposit successful!")
        else:
            print("Account not found!")

    def withdraw(self, account_number, amount):
        account = self.account_service.get_account(account_number)
        if not account:
            print(f"Debug: Account {account_number} not found!")
            raise InvalidAccountException("Account does not exist!")

        if amount % 500 != 0 and amount % 100 != 0:
            print("Enter amount in multiples of 500 or 100!")
            return

        account.withdraw(amount)
        self.account_service.update_account(account)
        self.transaction_dao.transact(account_number,'withdraw',amount)
        print("Withdrawal successful!")

    def calculate_interest(self, account_number):
        account = self.account_service.get_account(account_number)
        if account and account.get_account_type().lower() == "savings":
            interest = account.calculate_interest()
```

```
            account.deposit(interest)  # Add interest to balance
            self.account_service.update_account(account)  # Update DB
            print("Interest added to account balance.")
        else:
            print("Interest applies only to savings accounts or account not found.")
```

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.

Python doesn't require external type specification and overloading is not needed here

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).

SavingsAccount.py

```
from entity.Account import Account

class SavingsAccount(Account):
    def __init__(self, account_id, balance, interest_rate):
        super().__init__(account_id, "savings", balance)
        self.interest_rate = interest_rate  # in percentage

    def calculate_interest(self):
        interest = self.__balance * (self.interest_rate / 100)
        return interest
```

CurrentAccount.py

```
from entity.Account import Account

class CurrentAccount(Account):
    OVERDRAFT_LIMIT = 1000.00

    def __init__(self, account_id, balance):
        super().__init__(account_id, "current", balance)

    def withdraw(self, amount):
        if amount > 0:
            if self.__balance + self.OVERDRAFT_LIMIT >= amount:
                self.__balance -= amount
            else:
                print("Exceeded overdraft limit!")
        else:
            print("Invalid withdraw amount.")
```

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.

```
while True:
    print("\n===== BANK SYSTEM MENU =====")
    print("1. Create Account")
    print("2. Perform Bank Transactions")
    print("3. Check Loan Eligibility")
    print("4. Calculate Compound Interest")
    print("5. Print Account Details")
    print("6. Create Password ")
    print("7. Exit")

    choice = input("Choose an option: ")

    if choice == "1":
        print("\n--- Create New Account ---")
        print("1. Savings Account")
        print("2. Current Account")
        acc_type = input("Choose account type : ")

        if acc_type in ["1", "2"]:
            print("\n--- Enter Customer Details ---")
            first_name = input("First Name: ")
            last_name = input("Last Name: ")
```

```
        dob = input("Date of Birth (YYYY-MM-DD): ")
        email = input("Email: ")
        phone = input("Phone Number: ")
        address = input("Address: ")

        customer_id = bank.find_or_create_customer(first_name, last_name, dob, email,
phone, address)
        print(f"\nCustomer ID: {customer_id}")

        balance = Decimal(input("Initial Deposit Amount: "))
        account_type = "Savings" if acc_type == "1" else "Current"

        if account_type == "Savings":
            interest_rate = float(input("Enter Interest Rate (%): "))
        else:
            interest_rate = None

        account_number = bank.create_account(customer_id, balance,
account_type,interest_rate)
        print(f"{account_type} Account created with Account Number:
{account_number}")

    else:
        print("Invalid account type selected.")
```

```
--- Create New Account ---
1. Savings Account
2. Current Account
Choose account type : 2

--- Enter Customer Details ---
First Name: Kalpu
Last Name: prabu
Date of Birth (YYYY-MM-DD): 1978-05-31
Email: anu@mail.com
Phone Number: 8975867902
Address: Chennai
Existing customer found.

Customer ID: 11
Initial Deposit Amount: 52000
Current Account created with Account Number: 1002
```

```
Choose a transaction: 3
Enter Account Number: 8
Enter withdrawal amount: 500
Withdrawn 500.00. Remaining balance: -999.65
Overdraft used: 999.65
Remaining overdraft limit: 0.35
Account updated successfully!
Withdrawal successful!
```

```
Choose a transaction: 3
Enter Account Number: 8
Enter withdrawal amount: 500
Exceeded overdraft limit!
```

Task 9: Abstraction

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

Attributes: Account number,Customer name, Balance.

Constructors: oImplement default constructors and overload the constructor with Account attributes, generate getter and setter, print all information of attribute methods for the attributes.

Abstract methods:

- **deposit(amount: float):** Deposit the specified amount into the account.
- **withdraw(amount: float):** Withdraw the specified amount from the account (implement error handling for insufficient funds).
- **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).

```
from abc import abstractmethod
from exception.insufficient_funds_exception import InsufficientFundsException
from decimal import Decimal

class Account:
    # INTEREST_RATE = Decimal(4.5)  # Fixed interest rate
```

```
def __init__(self, account_number=None, account_type=None, balance=0.0):
    self.__account_number = account_number
    self.__account_type = account_type
    self.__balance = balance

# Getters
def get_account_number(self):
    return self.__account_number

def get_account_type(self):
    return self.__account_type

def get_balance(self):
    return self.__balance

# Setters
def set_account_number(self, account_number):
    self.__account_number = account_number

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

def set_balance(self, balance):
    self.__balance = balance

@abstractmethod
def deposit(self, amount):
    pass

@abstractmethod
def withdraw(self, amount):
    pass

@abstractmethod
```

```
    def calculate_interest(self):
        pass

    def print_account_info(self):
        print(f"Account Number: {self.__account_number}")
        print(f"Account Type: {self.__account_type}")
        print(f"Balance: ${self.__balance:.2f}")

    def print_balance(self):
        print(f"Balance: ${self.__balance:.2f}")
```

Already SavingsAccount and CurrentAccount subclasses created

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.

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.

Bank class with the above functionalities is implemented already.

Task 10: Has A Relation / Association

1. Create a `Customer` class with the following

Attributes: Customer ID ,First Name ,Last Name ,Email Address (validate with valid email address) ,Phone Number (Validate 10-digit phone number) ,Address

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

We already have Customer class. So only I have to update the code with validating email and phone number.

```
def set_email(self, email):
    if email is not None and re.match(r'^[\w\.-]+@[\w\.-]+\.\w{2,4}$', email):
        self.__email = email
    else:
        raise ValueError("Invalid email address")

def set_phone(self, phone):
    if phone is not None and re.match(r'^\d{10}$', phone):
        self.__phone = phone
    else:
        raise ValueError("Invalid phone number, must be 10 digits")
```

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

```
    def __init__(self, account_number=None, account_type=None,
balance=0.0,customer=None):
        self.__account_number = account_number
        self.__account_type = account_type
```

```
        self.__balance = balance
        self.__customer = customer

    # Getters
    def get_account_number(self):
        return self.__account_number

    def get_account_type(self):
        return self.__account_type

    def get_balance(self):
        return self.__balance

    def get_customer(self):
        return self.__customer

    # Setters
    def set_account_number(self, account_number):
        self.__account_number = account_number

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

    def set_balance(self, balance):
        self.__balance = balance

    def set_customer(self, customer):
        self.__customer = customer
```

3.Create a Bank Class and must have following requirements:

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.

I have already created the first four methods in the above tasks.

```
def transfer(self, from_account_number, to_account_number, amount):
    if amount <= 0:
        print("Transfer amount must be greater than 0.")
        return False

    from_account = self.account_service.get_account(from_account_number)
    to_account = self.account_service.get_account(to_account_number)

    if not from_account or not to_account:
        print("One or both account numbers are invalid.")
        return False

    if from_account.get_balance() < amount:
        print("Insufficient balance in the source account.")
        return False

    # Perform balance updates in memory
    from_account.set_balance(from_account.get_balance() - amount)
    to_account.set_balance(to_account.get_balance() + amount)

    # Update both accounts in DB
```

```
    self.account_service.update_account(from_account)
    self.account_service.update_account(to_account)

    print(f"₹{amount} transferred from Account {from_account_number} to Account {to_account_number}.")
    return True
```

```
Choose a transaction: 4
Enter Account Number to transfer from: 2
Enter Account Number to transfer to: 1
Enter transfer amount: 50
Account updated successfully!
Account updated successfully!
₹50 transferred from Account 2 to Account 1.
```

```
def get_account_details(self, account_number: int):
    """Fetches account and corresponding customer details."""
    try:
        conn = DBConnUtil.get_connection()
        cursor = conn.cursor()

        sql = """
            SELECT a.account_number, a.account_type, a.balance,
                   c.customer_id, c.first_name, c.last_name, c.dob, c.email, c.phone_number, c.address
            FROM Accounts a
            JOIN Customers c ON a.customer_id = c.customer_id
            WHERE a.account_number = %s
        """
        cursor.execute(sql, (account_number,))
        row = cursor.fetchone()

        if not row:
```

```
            print("Account does not exist!")
            return None

        # Unpack row
        (acc_no, acc_type, balance,
         cust_id, first_name, last_name, dob, email, phone, address) = row

        # Print or return the combined result
        print("Account Details:")
        print(f"  Account Number : {acc_no}")
        print(f"  Account Type   : {acc_type}")
        print(f"  Balance        : ₹{balance}")
        print("Customer Details:")
        print(f"  Customer ID    : {cust_id}")
        print(f"  Name           : {first_name} {last_name}")
        print(f"  DOB            : {dob}")
        print(f"  Email          : {email}")
        print(f"  Phone          : {phone}")
        print(f"  Address        : {address}")
        return {
            "account_number": acc_no,
            "account_type": acc_type,
            "balance": balance,
            "customer_id": cust_id,
            "first_name": first_name,
            "last_name": last_name,
            "dob": dob,
            "email": email,
            "phone": phone,
            "address": address
        }

    except mysql.connector.Error as e:
        print(f"Database error: {e}")
```

```
    return None
finally:
    if cursor: cursor.close()
    if conn: conn.close()
```

```
Choose a transaction: 7
Enter Account Number: 2
Account Details:
  Account Number : 2
  Account Type   : current
  Balance        : ₹26150.00
Customer Details:
  Customer ID    : 2
  Name           : Priya Kumar
  DOB            : 1990-08-22
  Email          : priya.kumar@example.com
  Phone          : 9867543211
  Address        : Mumbai, Maharashtra
```

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

```
def get_next_account_number(self):
    conn = DBConnUtil.get_connection()
    cursor = conn.cursor()
    query = "SELECT MAX(account_number) FROM accounts"
    cursor.execute(query)
    result = cursor.fetchone()
    cursor.close()
    conn.close()
    return max(1001, result[0] + 1 if result[0] is not None else 1001)
```

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.

```
--- Bank Transactions ---
1. Check Balance
2. Deposit
3. Withdraw
4. Transfer
5. Calculate Interest (Savings Only)
6. View Transaction History
7. View Account Details
```

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.

I have already created Customers and Account class and Subclasses savingsaccount and currentaccount.

```
from entity.Account import Account

class ZeroBalanceAccount(Account):
    def __init__(self, account_number):
        super().__init__(account_number, "zero_balance", 0.0)

    def deposit(self, amount):
        if amount > 0:
            self.set_balance(self.get_balance() + amount)
            print(f"Deposited ₹{amount:.2f}. New balance: ₹{self.get_balance():.2f}")
        else:
            print("Deposit amount must be greater than zero.")

    def withdraw(self, amount):
        if amount > 0:
            if self.get_balance() >= amount:
                self.set_balance(self.get_balance() - amount)
                print(f"Withdrawn ₹{amount:.2f}. Remaining balance:
₹{self.get_balance():.2f}")
                return True
            else:
                print("Insufficient balance!")
        else:
            print("Invalid withdraw amount")
        return False
```

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.

```
from abc import ABC, abstractmethod

class ICustomerServiceProvider(ABC):

    @abstractmethod
    def get_account_balance(self, account_number: int) -> float:
        pass

    @abstractmethod
    def deposit(self, account_number: int, amount: float) -> float:
        pass

    @abstractmethod
    def withdraw(self, account_number: int, amount: float) -> float:
        pass

    @abstractmethod
    def transfer(self, from_account_number: int, to_account_number: int, amount: float)
-> None:
        pass

    @abstractmethod
    def get_account_details(self, account_number: int) -> dict:
        pass
```

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.

l**istAccounts()**:Account[] accounts: List all accounts in the bank.
calculateInterest(): the calculate_interest() method to calculate interest based on the balance and interest rate.

```
from abc import ABC, abstractmethod
from entity.Customer import Customer
from typing import List

class IBankServiceProvider(ABC):

    @abstractmethod
    def create_account(self, customer: Customer, accNo: int, accType: str, balance: float)
-> None:
        pass

    @abstractmethod
    def list_accounts(self) -> List:
        pass

    @abstractmethod
    def calculate_interest(self,account_number) -> None:
        pass
```

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

```
from services.customer_service import ICustomerServiceProvider
from dao.account_dao import AccountDAO
from dao.bank_dao import BankDAO
from exception.insufficient_funds_exception import InsufficientFundsException
from entity.Account import Account
from entity.Customer import Customer
from entity.SavingsAccount import SavingsAccount
```

```
from entity.CurrentAccount import CurrentAccount
from util.db_conn_util import DBConnUtil

class CustomerServiceProviderImpl(ICustomerServiceProvider):
    def __init__(self, account_dao: AccountDAO, bank_dao: BankDAO):
        self.account_dao = account_dao
        self.bank_dao = bank_dao

    def get_account_balance(self, account_number: int) -> float:
        account = self.account_dao.get_account(account_number)
        return account.get_balance()

    def deposit(self, account_number: int, amount: float) -> float:
        account = self.account_dao.get_account(account_number)
        account.set_balance( account.get_balance() +amount)
        self.account_dao.update_account(account)
        return account.get_balance()

    def withdraw(self, account_number: int, amount: float) -> float:
        account = self.account_dao.get_account(account_number)
        if account.get_account_type().lower() == "savings" and (account.get_balance() -
amount) < 500:
            raise InsufficientFundsException("Minimum balance requirement not met.")
        if account.get_balance() < amount:
            raise InsufficientFundsException("Insufficient funds.")
        account.set_balance(account.get_balance()- amount)
        self.account_dao.update_account(account)
        return account.get_balance()

    def transfer(self, from_account_number: int, to_account_number: int, amount: float)
-> None:
        self.withdraw(from_account_number, amount)
        self.deposit(to_account_number, amount)
```

```
    def get_account_details(self, account_number: int) :
        account = self.account_dao.get_account_details(account_number)
        return account
```

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

```
from services.customer_service_impl import CustomerServiceProviderImpl
from services.bank_service import IBankServiceProvider
from entity.Account import Account
from typing import List
from dao.bank_dao_impl import BankDAOImpl
from entity.SavingsAccount import SavingsAccount
from entity.CurrentAccount import CurrentAccount
from dao.account_dao_impl import AccountDaoImpl
from entity.Customer import Customer

class BankServiceProviderImpl(CustomerServiceProviderImpl,
IBankServiceProvider):
    def __init__(self, branch_name: str, branch_address: str):
        super().__init__()
        self.accountList: List[Account] = []
        self.branchName: str = branch_name
        self.branchAddress: str = branch_address
        self.dao = BankDAOImpl()
        self.account_service = AccountDaoImpl()

    def create_account(self, customer, balance, account_type, interest_rate=None):
        # customer = self.find_or_create_customer(first_name, last_name, dob, email,
phone, address)
        if customer is None:
            print("Failed to create/find customer. Aborting account creation.")
            return None
```

```
        if customer.get_customer_id() is None:
            inserted_id = self.dao.insert_customer(
                customer.get_first_name(),
                customer.get_last_name(),
                customer.get_dob(),
                customer.get_email(),
                customer.get_phone(),
                customer.get_address()
            )
            if inserted_id:
                customer.set_customer_id(inserted_id)
                print("New customer inserted with ID:", inserted_id)
            else:
                print(" Failed to insert new customer into DB.")
                return None
        account_number = self.dao.get_next_account_number()

        if account_type == "Savings":
            account = SavingsAccount(account_number, account_type, balance,
interest_rate, customer)
        else:
            account = CurrentAccount(account_number, account_type, balance, customer)

        self.dao.insert_account(account_number, customer.get_customer_id(),
account_type, balance)
        return account_number

    def list_accounts(self) -> List[Account]:
        return self.accountList

    def calculate_interest(self,account_number):
        """Calculate and add interest for savings accounts"""
        account = self.account_service.get_account(account_number)
        if account and account.get_account_type().lower() == "savings":
```

```
            interest = account.calculate_interest()
            account.deposit(interest)  # Add interest to balance
            self.account_service.update_account(account)  # Update DB
            print("Interest added to account balance.")
        else:
            print("Interest applies only to savings accounts or account not found.")
```

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.

Already created bank class that implements these methods

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.

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.

```
class InsufficientFundsException(Exception):  8 usages
    def __init__(self, message):
        super().__init__(message)
```

```
class InvalidAccountException(Exception):  12 usages
    def __init__(self, message="Invalid account number!"):
        super().__init__(message)
```

```
class NullPointerException(Exception):  2 usages
    def __init__(self, message="Null value encountered."):
        super().__init__(message)
```

```
class OverDraftLimitExcededException(Exception):  4 usages
    def __init__(self, message="Overdraft limit exceeded for current account."):
        super().__init__(message)
```

Task 13: Collection

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

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<Account> object to sort the accounts based on customer name when listAccounts() method called.

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

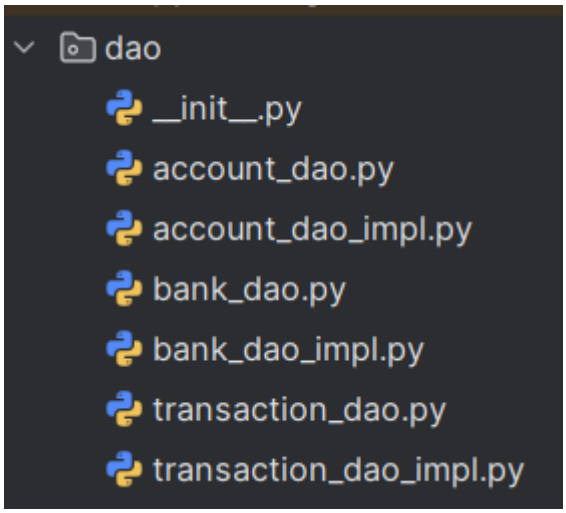
```
from entity.Account import Account

class HMBankList:
    def __init__(self):
        self.accounts = []

    def add_account(self, account: Account):
        self.accounts.append(account)

    def list_accounts(self):
        return sorted(self.accounts, key=lambda acc: acc.customer_name.lower())

class HMBankSet:
    def __init__(self):
        self.accounts = set()

    def add_account(self, account: Account):
        self.accounts.add(account)

    def list_accounts(self):
        return sorted(self.accounts, key=lambda acc: acc.customer_name.lower())

class HMBankMap:
    def __init__(self):
        self.accounts = {}

    def add_account(self, account: Account):
        self.accounts[account.account_number] = account

    def get_account(self, account_number: int) -> Account:
        return self.accounts.get(account_number)

    def list_accounts(self):
        return sorted(self.accounts.values(), key=lambda acc: acc.customer_name.lower())
```

Task 14: Database Connectivity.

For Database access, I have created the dao package and inside the package I have files that contain abstract classes and their implementation in another files.

I have created the db_conn_util file inside the util package for establishing db connectivity

Dao Package :



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.

Bankapp file has been created to perform the operations.

```
BANK SYSTEM MENU
1. Create Account
2. Perform Bank Transactions
3. Check Loan Eligibility
4. Calculate Compound Interest
5. View Account Details
6. Create Password
7. Exit
```

```
Create New Account
1. Savings Account
2. Current Account
```

```
Bank Transactions
1. Check Balance
2. Deposit
3. Withdraw
4. Transfer
5. Calculate Interest (Savings Only)
6. View Transaction History
```