

# Banking System Control Structure

## Task 1: Conditional Statements

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

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

Tasks:

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

```
creditscore=int(input("Enter the Credit Course"))
```

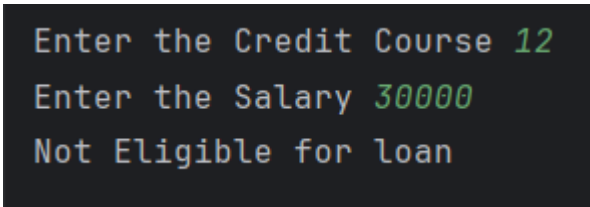
```
Salary=int(input("Enter the Salary"))
```

```
if creditscore>70 and salary>=50000:
```

```
print("Eligible for loan")
```

```
else:
```

```
print("Not Eligible for loan")
```

A screenshot of a terminal window with a dark background. It shows the execution of a Python program. The first prompt is "Enter the Credit Course" followed by the input "12". The second prompt is "Enter the Salary" followed by the input "30000". The final output is "Not Eligible for loan".

```
Enter the Credit Course 12
Enter the Salary 30000
Not Eligible for loan
```

## 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.

```
Balance=float(input("Enter your balance"))
```

```
print("\nATM Menu:")
```

```
print("1. Check Balance")
```

```
print("2. Withdraw")
print("3. Deposit")
choice=int(input("Select any one option"))
match choice:
    case 1:
        print("Available balance is:",Balance)
    case 2:
        print("Enter the amount to be withdrawn")
        amount=int(input())
        if (Balance>amount):
            if(amount%100==0 or amount%500):
                print("amount withdrawn")
            else:
                print("Please provide amount in multiples of 100 or 500")
        else:
            print("Amount greater than available balance")
```

```
Enter your balance 29000
```

```
ATM Menu:
1. Check Balance
2. Withdraw
3. Deposit
Select any one option 1
Available balance is: 29000.0

Process finished with exit code 0
```

### Task 3: Loop Structures

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

Tasks:

1. Create a program that calculates the future balance of a savings account.
2. Use a loop structure (e.g., for loop) to calculate the balance for multiple customers.
3. Prompt the user to enter the initial balance, annual interest rate, and the number of years.
4. Calculate the future balance using the formula:  
$$\text{future\_balance} = \text{initial\_balance} * (1 + \text{annual\_interest\_rate}/100)^{\text{years}}$$
5. Display the future balance for each customer.

```
num = int(input("Enter the number of customers: "))
```

```
for i in range(num):
```

```
    print("\nCustomer", i + 1)
```

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

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

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

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

```
    print("Future balance after", years, "years:", future_balance)
```

```
Enter the number of customers: 2

Customer 1
Enter the initial balance: 5000
Enter the annual interest rate (%): 5
Enter the number of years: 1
Future balance after 1 years: 5250.0

Customer 2
Enter the initial balance: 7000
Enter the annual interest rate (%): 4
Enter the number of years: 1
Future balance after 1 years: 7280.0

Process finished with exit code 0
```

#### Task 4: Looping, Array and Data Validation

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

Tasks:

1. Create a Python program that simulates a bank with multiple customer accounts.
2. Use a loop (e.g., while loop) to repeatedly ask the user for their account number and balance until they enter a valid account number.
3. Validate the account number entered by the user.
4. If the account number is valid, display the account balance. If not, ask the user to try again.

```
account_details = {12435:50000,46321:12000,43598:45000,98435:34000,39572:56000}
```

```
account_number=int(input("Enter the account number: "))
```

```
for i in account_details:
```

```
    if(account_number in account_details):
```

```
        print("The available balance is: ", account_details.get(account_number))
```

```
        break
```

```
    else:
```

```
        print("Enter valid account number ")
```

```
        break
```

```
Enter the account number: 43598
The available balance is: 45000

Process finished with exit code 0
```

```
Enter the account number: 785
Enter valid account number

Process finished with exit code 0
```

## Task 5: Password Validation

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

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

```
password = input("enter the password:")
if len(password) >= 8:
    digit = 0
    upper = 0
    for char in password:
        if char.isdigit():
            digit += 1
        elif char.isupper():
            upper += 1
    if digit >= 1 and upper >= 1:
        print("valid Password")
    else:
        print("Enter password with atleast one digit and atleast one uppercase character ")
else:
    print("enter password with 8 characters or more")
```

```
enter the password:sne
enter password with 8 characters or more

Process finished with exit code 0
```

```
enter the password:bdFjAlo1
valid Password

Process finished with exit code 0
```

### 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.

```
transactions = []

while True:

    print("\nChoose an option:")
    print("1 Deposit")
    print("2 Withdrawal")
    print("3 Exit")

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

    if choice == 1:

        amount = float(input("Enter deposit amount: "))
        transactions.append(('Deposit', amount))
        print("Deposit of ", amount, "added.")

    elif choice==2:

        amount = float(input("Enter withdrawal amount: "))
        transactions.append(('Withdrawal', amount))
        print("Withdrawal of ", amount, "successfull")

    elif choice==3:

        print("\nTransaction History:")

        for transaction in transactions:

            print(transaction[0], " : ", transaction[1])

        break

    else:

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

```
Choose an option:
1 Deposit
2 Withdrawal
3 Exit
Enter your choice: 1
Enter deposit amount: 5000
Deposit of 5000.0 added.

Choose an option:
1 Deposit
2 Withdrawal
3 Exit
Enter your choice: 4
Invalid choice. Please enter a valid option.

Choose an option:
1 Deposit
2 Withdrawal
3 Exit
Enter your choice: 3

Transaction History:
Deposit : 5000.0

Process finished with exit code 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

```
class Customer:
```

```
    def __init__(self, customer_id=None, first_name=None, last_name=None, email=None,
phone=None, address=None):
```

```
        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 customer_id(self):
```

```
        return self._customer_id
```

```
    @customer_id.setter
```

```
    def customer_id(self, custid):
```

```
        self._customer_id = custid
```

```
    @property
```

```
    def first_name(self):
```

```
        return self._first_name
```



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

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

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

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

```
@email.setter
def email(self, em):
    self._email = em
```

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

```
@phone.setter
def phone(self, phn):
    self._phone = phn
```

```
@property
```

```
def address(self):  
    return _self.address
```

```
@address.setter
```

```
def address(self, addr):  
    self._address = addr
```

```
def details(self):  
    print("Customer ID:", self._customer_id)  
    print("First Name:", self._first_name)  
    print("Last Name:", self._last_name)  
    print("Email Address:", self._email)  
    print("Phone Number:", self._phone)  
    print("Address:", self._address)
```

```
cust = Customer("20", "Sne", "BK", "sne@gmail.com", "1234512345", "Coimbatore")  
cust.details()
```

```
Customer ID: 20  
First Name: Sne  
Last Name: BK  
Email Address: sne@gmail.com  
Phone Number: 1234512345  
Address: Coimbatore  
  
Process finished with exit code 0
```

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%

class Accounts:

```
def __init__(self,Account_number,Account_type,Account_balance):
```

```
    self._Account_number=Account_number
```

```
    self._Account_type=Account_type
```

```
    self._Account_balance=Account_balance
```

```
@property
```

```
def Account_number(self):
```

```
    return self._Account_number
```

```
@Account_number.setter
```

```
def Account_number(self,accnum):
```

```
    self._Account_number=accnum
```

```
@property
```

```
def Account_type(self):
```

```
    return self._Account_type
```

```
@Account_type.setter
def Account_type(self, acctype):
    self._Account_type = acctype

@property
def Account_balance(self):
    return self._Account_balance

@Account_balance.setter
def Account_balance(self, accbal):
    self._Account_balance = accbal

def display(self):
    print("Account Number:", self._Account_number)
    print("Account Type:", self._Account_type)
    print("Account Balance:", self._Account_balance)

def deposit(self, amount):
    if amount > 0:
        self._Account_balance += amount
        print("Deposit of ", amount, "completed.")
    else:
        print("Invalid deposit amount. Please enter a positive value.")

def withdraw(self, amount):
    if amount > 0:
        if amount <= self._Account_balance:
            self._Account_balance -= amount
            print("Withdrawal of ", amount, "completed.")
        else:
```

```
        print("Insufficient balance. Withdrawal cannot be processed.")
    else:
        print("Invalid withdrawal amount. Please enter a positive value.")

def calculate_interest(self):
    interest_rate = 4.5 / 100
    interest_amount = self._Account_balance * interest_rate
    print("Interest amount:", interest_amount)

obj=Accounts(224,"savings",50000)
obj.display()
obj.deposit(3000)
obj.withdraw(45000)
obj.calculate_interest()
```

```
Account Number: 224
Account Type: savings
Account Balance: 50000
Deposit of 3000 completed.
Withdrawal of 45000 completed.
Interest amount: 360.0

Process finished with exit code 0
```

#### Task 8: Inheritance and polymorphism

1. Overload the deposit and withdraw methods in Account class as mentioned below.
  - deposit(amount: float): Deposit the specified amount into the account.
  - withdraw(amount: float): Withdraw the specified amount from the account. withdraw amount only if there is sufficient fund else display insufficient balance.
  - deposit(amount: int): Deposit the specified amount into the account.

- `withdraw(amount: int)`: Withdraw the specified amount from the account. withdraw amount only if there is sufficient fund else display insufficient balance.
- `deposit(amount: double)`: Deposit the specified amount into the account.
- `withdraw(amount: double)`: Withdraw the specified amount from the account. withdraw amount only if there is sufficient fund else display insufficient balance.

```
class IntAccount(Accounts):
```

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

```
        if amount > 0:
```

```
            self._Account_balance += amount
```

```
            print("Deposit of ", amount, "completed.")
```

```
        else:
```

```
            print("Invalid deposit amount. Please enter a positive integer value.")
```

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

```
        if amount > 0:
```

```
            if amount <= self._Account_balance:
```

```
                self._Account_balance -= amount
```

```
                print("Withdrawal of ", amount, "completed.")
```

```
            else:
```

```
                print("Insufficient balance. Withdrawal cannot be processed.")
```

```
        else:
```

```
            print("Invalid withdrawal amount. Please enter a positive integer value.")
```

```
class FloatAccount(Accounts):
```

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

```
        if amount > 0:
```

```
            self._Account_balance += amount
```

```
            print("Deposit of ", amount, "completed.")
```

```
        else:
```

```
print("Invalid deposit amount. Please enter a positive float value.")
```

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

```
    if amount > 0:
```

```
        if amount <= self._Account_balance:
```

```
            self._Account_balance -= amount
```

```
            print("Withdrawal of ", amount, "completed.")
```

```
        else:
```

```
            print("Insufficient balance. Withdrawal cannot be processed.")
```

```
    else:
```

```
        print("Invalid withdrawal amount. Please enter a positive float value.")
```

```
int_account = IntAccount("123456", "Savings", 1000)
```

```
int_account.display()
```

```
int_account.deposit(500)
```

```
int_account.withdraw(200)
```

```
int_account.calculate_interest()
```

```
float_account = FloatAccount("789012", "Current", 2000.0)
```

```
float_account.display()
```

```
float_account.deposit(500.50)
```

```
float_account.withdraw(1000.25)
```

```
float_account.calculate_interest()
```

```
Account Number:  225
Account type:    Savings
Available Balance:  14000
Amount has been deposited !!
Available balance =  18000
Insufficient Balance
Intrest Amount =  810.0
```

## 2. Create Subclasses for Specific Account Types

- Create subclasses for specific account types (e.g., `SavingsAccount`, `CurrentAccount`) that inherit from the `Account` class.
  - o **SavingsAccount:** A savings account that includes an additional attribute for interest rate. override the `calculate_interest()` from `Account` class method to calculate interest based on the balance and interest rate.
  - o **CurrentAccount:** A current account that includes an additional attribute `overdraftLimit`. A current account with no interest. Implement the `withdraw()` method to allow overdraft up to a certain limit (configure a constant for the overdraft limit).

```
class SavingsAccount(Accounts):
```

```
    def __init__(self, Account_number=None, Account_balance=None, interest_rate=None):
        super().__init__(Account_number, "Savings", Account_balance)
        self._interest_rate = interest_rate
```

```
    @property
```

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

```
    @interest_rate.setter
```

```
    def interest_rate(self, value):
        self._interest_rate = value
```

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

```
        interest_amount = self.Account_balance * (self.interest_rate / 100)
        print("Interest amount:", interest_amount)
```

```
class CurrentAccount(Accounts):
```

```
    OVERDRAFT_LIMIT = 10000
```



```
def __init__(self, Account_number=None, Account_balance=None,
overdraft_limit=None):
```

```
    super().__init__(Account_number, "Current", Account_balance)
```

```
    self._overdraft_limit = overdraft_limit
```

```
@property
```

```
def overdraft_limit(self):
```

```
    return self._overdraft_limit
```

```
@overdraft_limit.setter
```

```
def overdraft_limit(self, value):
```

```
    self._overdraft_limit = value
```

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

```
    if amount > self.Account_balance + self.overdraft_limit:
```

```
        print("Withdrawal amount exceeds balance and overdraft limit.")
```

```
    else:
```

```
        self.Account_balance -= amount
```

```
        print("Withdrawal of ", amount, "completed.")
```

```
savings_account = SavingsAccount("102", 50000, 7.7)
```

```
savings_account.display()
```

```
savings_account.calculate_interest()
```

```
current_account = CurrentAccount("203", 26000, 10000)
```

```
current_account.display()
```

```
current_account.withdraw(2500)
```

```
Account Number: 102
Account Type: Savings
Account Balance: 50000
Interest amount: 3850.0
Account Number: 203
Account Type: Current
Account Balance: 26000
Withdrawal of 2500 completed.

Process finished with exit code 0
```

1. Create a **Bank** class to represent the banking system. Perform the following operation in main method:
  - Display menu for user to create object for account class by calling parameter constructor. Menu should display options `SavingsAccount` and `CurrentAccount`. user can choose any one option to create account. use switch case for implementation.
  - **deposit(amount: float)**: Deposit the specified amount into the account.
  - **withdraw(amount: float)**: Withdraw the specified amount from the account. For saving account withdraw amount only if there is sufficient fund else display insufficient balance.  
For Current Account withdraw limit can exceed the available balance and should not exceed the overdraft limit.
  - **calculate\_interest()**: Calculate and add interest to the account balance for savings accounts.

class Account:

```
def __init__(self, acc_type, balance=0):
```

```
    self.acc_type = acc_type
```

```
    self.balance = balance
```

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

```
    if amount > 0:
```

```
        self.balance += amount
```

```
        print("Deposit successful. Current balance:", self.balance)
```

```
else:  
    print("Invalid amount for deposit.")
```

```
def withdraw(self, amount):  
    overdraft = 2000  
    if amount > 0:  
        if self.acc_type == "SavingsAccount":  
            if amount <= self.balance:  
                self.balance -= amount  
                print("Withdrawal successful. Current balance:", self.balance)  
            else:  
                print("Insufficient balance.")  
        elif self.acc_type == "CurrentAccount":  
            if amount <= self.balance + overdraft:  
                self.balance -= amount  
                print("Withdrawal successful. Current balance:", self.balance)  
            else:  
                print("overdraft limit exceeded")  
        else:  
            print("Invalid account type.")  
    else:  
        print("Invalid amount for withdrawal.")
```

```
def calculate_interest(self):  
    interest_amount = self.balance * (7.8 / 100)  
    print("Interest amount:", interest_amount)
```

```
class Bank:  
    def create_account(self):  
        print("Choose account type:")
```

```
print("1. SavingsAccount")
print("2. CurrentAccount")
choice = input("Enter choice (1/2): ")
if choice == "1":
    acc_type = "SavingsAccount"
elif choice == "2":
    acc_type = "CurrentAccount"
else:
    print("Invalid choice.")
    return None

balance = float(input("Enter initial balance: "))
return Account(acc_type, balance)

def main(self):
    account = self.create_account()
    if account:
        while True:
            print("\nMenu:")
            print("1. Deposit")
            print("2. Withdraw")
            print("3. Calculate Interest (for SavingsAccount)")
            print("4. Exit")
            choice = input("Enter choice (1/2/3/4): ")

            if choice == "1":
                amount = float(input("Enter deposit amount: "))
                account.deposit(amount)
            elif choice == "2":
                amount = float(input("Enter withdrawal amount: "))
```

```

        account.withdraw(amount)

    elif choice == "3" and account.acc_type == "SavingsAccount":
        account.calculate_interest()

    elif choice == "4":
        print("Exiting program.")
        break

    else:
        print("Invalid choice.")

```

```
bank = Bank()
```

```
bank.main()
```

```

Choose account type:
1. SavingsAccount
2. CurrentAccount
Enter choice (1/2): 1
Enter initial balance: 500

Menu:
1. Deposit
2. Withdraw
3. Calculate Interest (for SavingsAccount)
4. Exit
Enter choice (1/2/3/4): 4
Exiting program.

Process finished with exit code 0

```

### 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.

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

```
class Bankaccount(ABC):
```

```
    def __init__(self, Account_number, Customer_name, Balance):
```

```
        self.Account_number = Account_number
```

```
        self.Customer_name = Customer_name
```

```
        self.Balance = Balance
```

```
    @property
```

```
    def account_number(self):
```

```
        return self.Account_number
```

```
    @account_number.setter
```

```
    def account_number(self, value):
```

```
        self.Account_number = value
```

```
    @property
```

```
    def customer_name(self):
```

```
        return self.Customer_name
```

```
    @customer_name.setter
```

```
    def customer_name(self, value):
```

```
self.Customer_name = value
```

```
@property
```

```
def balance(self):
```

```
    return self.Balance
```

```
@balance.setter
```

```
def balance(self, value):
```

```
    self.Balance = value
```

```
@abstractmethod
```

```
def Deposit(self, amount):
```

```
    pass
```

```
@abstractmethod
```

```
def Withdrawl(self, amount):
```

```
    pass
```

```
@abstractmethod
```

```
def interest(self):
```

```
    pass
```

```
def display(self):
```

```
    print("Account Number:", self.Account_number)
```

```
    print("Account Type:", self.Customer_name)
```

```
    print("Account Balance:", self.Balance)
```

```
class Account(Bankaccount):
```

```
def __init__(self, Account_Number, Customer_name, Balance):  
    super().__init__(Account_Number, Customer_name, Balance)
```

```
def Deposit(self, amount):  
    if amount > 0:  
        self.Balance += amount  
        print("Deposit of ", amount, "completed.")  
    else:  
        print("Invalid deposit amount. Please enter a positive value.")
```

```
def Withdrawl(self, amount):  
  
    if amount > 0:  
        if amount <= self.Balance:  
            self.Balance -= amount  
            print("Withdrawal of ", amount, "completed.")  
        else:  
            print("Insufficient balance. Withdrawal cannot be processed.")  
    else:  
        print("Invalid withdrawal amount. Please enter a positive value.")
```

```
def interest(self):  
    interest_rate = 4.5 / 100  
    interest_amount = self.Balance * interest_rate  
    print("Interest amount:", interest_amount)
```

```
Bank = Account(126, "Sne", 50000)  
Bank.Deposit(34000)  
Bank.Withdrawl(50000)
```



Bank.display()

```
1. Deposit
2. Withdraw
3. Calculate Interest (for SavingsAccount)
4. Exit
Enter choice (1/2/3/4): 2
Enter withdrawal amount: 50000
Withdrawal successful. Current balance: 0.0
```

```
Menu:
1. Deposit
2. Withdraw
3. Calculate Interest (for SavingsAccount)
4. Exit
Enter choice (1/2/3/4): 3
Interest amount: 0.0
```

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 ABC, abstractmethod
```

```
class Bankaccount(ABC):
```

```
    def __init__(self, Account_number, Customer_name, Balance):
        self.Account_number = Account_number
```

```
self.Customer_name = Customer_name  
self.Balance = Balance
```

```
@property
```

```
def account_number(self):  
    return self.Account_number
```

```
@account_number.setter
```

```
def account_number(self, value):  
    self.Account_number = value
```

```
@property
```

```
def customer_name(self):  
    return self.Customer_name
```

```
@customer_name.setter
```

```
def customer_name(self, value):  
    self.Customer_name = value
```

```
@property
```

```
def balance(self):  
    return self.Balance
```

```
@balance.setter
```

```
def balance(self, value):  
    self.Balance = value
```

```
@abstractmethod
```

```
def Deposit(self, amount):  
    pass
```

```
@abstractmethod
```

```
def Withdrawl(self, amount):
```

```
    pass
```

```
def display(self):
```

```
    print("Account Number:", self.Account_number)
```

```
    print("Account Type:", self.Customer_name)
```

```
    print("Account Balance:", self.Balance)
```

```
class SavingsAccount(Bankaccount):
```

```
    def __init__(self,Account_Number,Customer_name,Balance):
```

```
        super().__init__(Account_Number,Customer_name,Balance)
```

```
    def Deposit(self,amount):
```

```
        if amount > 0:
```

```
            self.Balance += amount
```

```
            print("SAVINGS ACCOUNT : Deposit of ", amount, "completed.")
```

```
        else:
```

```
            print("SAVINGS ACCOUNT : Invalid deposit amount. Please enter a positive value.")
```

```
    def Withdrawl(self,amount):
```

```
        if amount > 0:
```

```
            if amount <= self.Balance:
```

```

        self.Balance -= amount

        print("SAVINGS ACCOUNT : Withdrawal of ", amount, "completed.")
    else:
        print("SAVINGS ACCOUNT : Insufficient balance. Withdrawal cannot be
processed.")
    else:
        print("SAVINGS ACCOUNT : Invalid withdrawal amount. Please enter a positive
value.")

def interest(self):
    interest_rate = 4.5 / 100
    interest_amount = self.Balance * interest_rate
    print("SAVINGS ACCOUNT : Interest amount:", interest_amount)

class CurrentAccount(Bankaccount):

    def Deposit(self, amount):
        if amount > 0:
            self.Balance += amount
            print("CURRENT ACCOUNT : Deposit of ", amount, "completed.")
        else:
            print("CURRENT ACCOUNT : Invalid deposit amount. Please enter a positive
value.")

    def Withdrawal(self, amount):
        overdraft_limit = 20000

        if amount > 0:
            if amount <= self.Balance+overdraft_limit:
                self.Balance -= amount
                print("CURRENT ACCOUNT : Withdrawal of ", amount, "completed.")

```

```
        else:
            print("CURRENT ACCOUNT : Insufficient balance. Withdrawal cannot be
processed.")
        else:
            print("CURRENT ACCOUNT : Invalid withdrawal amount. Please enter a positive
value.")
```

```
savings = SavingsAccount(133,"Sne",25000)
```

```
savings.Deposit(3000)
```

```
savings.Withdrawl(13000)
```

```
savings.interest()
```

```
current = CurrentAccount(223,"Barani",45000)
```

```
current.Deposit(15000)
```

```
current.Withdrawl(100000)
```

```
SAVINGS ACCOUNT : Deposit of 3000 completed.
SAVINGS ACCOUNT : Withdrawal of 13000 completed.
SAVINGS ACCOUNT : Interest amount: 675.0
CURRENT ACCOUNT : Deposit of 15000 completed.
CURRENT ACCOUNT : Insufficient balance. Withdrawal cannot be processed.

Process finished with exit code 0
```

1. 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.

class Bank:

def create\_account(self):

choice = int(input("Enter your choice (1/2): "))

if choice == 1:

return SavingsAccount()

elif choice == 2:

return CurrentAccount()

else:

print("Invalid choice.")

return None

def main\_menu(self):

while True:

print("Welcome to the Banking System!")

print("Choose the type of account you want to create:")

print("1. Savings Account")

print("2. Current Account")

account = self.create\_account()

if account:

while True:

print("\nOperations for the account:")

print("1. Deposit")

print("2. Withdraw")

```
print("3. Calculate Interest (Savings Account only)")
print("4. Back to main menu")
operation = int(input("Enter your choice (1/2/3/4): "))
if operation == 1:
    amount = float(input("Enter deposit amount: "))
    account.deposit(amount)
elif operation == 2:
    amount = float(input("Enter withdrawal amount: "))
    account.withdraw(amount)
elif operation == 3:
    if account == SavingsAccount:
        account.calculate_interest()
    else:
        print("Current Account does not support interest calculation.")
elif operation == 4:
    break
else:
    print("Invalid choice.")
if operation == 4:
    break
```

```
class Account:
    def __init__(self):
        self.balance = 0

    def deposit(self, amount):
        self.balance += amount
        print("Deposit of ",amount," successful." )
        print("Available balance: ", self.balance)
```

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

```
class SavingsAccount(Account):  
    def withdraw(self, amount):  
        if self.balance >= amount:  
            self.balance -= amount  
            print("Withdrawal of ",amount," successful." )  
            print("Available balance: ", self.balance)  
        else:  
            print("Insufficient balance.")
```

```
def calculate_interest(self):  
    interest_rate = float(input("Enter interest rate: "))  
    interest = self.balance * interest_rate / 100  
    self.deposit(interest)
```

```
class CurrentAccount(Account):  
    def __init__(self):  
        super().__init__()  
        self.overdraft_limit = 0  
  
    def withdraw(self, amount):  
        if amount <= self.balance + self.overdraft_limit:  
            self.balance -= amount  
            print("Withdrawal of ",amount," successful." )  
            print("Available balance: ", self.balance)  
        else:
```



```
print("Withdrawal amount exceeds available balance and overdraft limit.")
```

```
bank = Bank()
```

```
bank.main_menu()
```

```
Welcome to the Banking System!
Choose the type of account you want to create:
1. Savings Account
2. Current Account
Enter your choice (1/2): 1

Operations for the account:
1. Deposit
2. Withdraw
3. Calculate Interest (Savings Account only)
4. Back to main menu
Enter your choice (1/2/3/4): 2
Enter withdrawal amount: 7000
Insufficient balance.

Operations for the account:
1. Deposit
2. Withdraw
3. Calculate Interest (Savings Account only)
4. Back to main menu
Enter your choice (1/2/3/4): 1
Enter deposit amount: 6500
Deposit of 6500.0 successful.
```

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

import re

class Customer:

def \_\_init\_\_(self, Customer\_id, First\_name, Last\_name, Email\_Address, Phone\_Number, Address):

self.Customer\_id = Customer\_id

self.First\_name = First\_name

self.Last\_name = Last\_name

self.email\_address = Email\_Address

self.phone\_number = Phone\_Number

self.Address = Address

def email\_valid(self, mail):

pattern = r'^[w\.-]+@[a-zA-Z\d\.-]+\.[a-zA-Z]{2,}\$'

if re.match(pattern, mail):

return True

else:

return False

def phone\_valid(self, phone):

if isinstance(phone, int) and len(str(phone))== 10:

return True

else:

return False

```
@property
def customer_id(self):
    return self.Customer_id

@customer_id.setter
def customer_id(self, cusid):
    self.Customer_id = cusid

@property
def first_name(self):
    return self.First_name

@first_name.setter
def first_name(self, fn):
    self.First_name = fn

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

@last_name.setter
def last_name(self, ln):
    self.Last_name = ln

@property
def email_address(self):
    return self.Email_Address

@email_address.setter
```

```
def email_address(self, ea):
    if self.email_valid(ea):
        self.Email_Address = ea
    else:
        raise ValueError("Invalid Email")

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

@phone_number.setter
def phone_number(self, pn):
    if self.phone_valid(pn):
        self.Phone_Number = pn
    else:
        raise ValueError("Invalid Phone Number")

@property
def address(self):
    return self.Address

@address.setter
def address(self, ad):
    self.Address = ad

def display(self):
    print("Customer ID: ", self.Customer_id)
    print("First Name: ", self.First_name)
    print("Last Name: ", self.Last_name)
    print("Email Address: ", self.Email_Address)
```

```
print("Phone Number: ", self.Phone_Number)

print("Address: ", self.Address)

cust = Customer(224,"sne","barani","sne@gmail.com",1234512345,"coimbatore")

cust.display()
```

```
Customer ID: 224
First Name: sne
Last Name: barani
Email Address: sne@gmail.com
Phone Number: 1234512345
Address: coimbatore

Process finished with exit code 0
```

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.

class Account:

```
def __init__(self, account_number=None, account_type=None, account_balance=0):

    self.account_number = account_number

    self.account_type = account_type

    self.account_balance = account_balance
```

@property

```
def account_number(self):
```

```
return self.Account_number
```

```
@account_number.setter
```

```
def account_number(self, value):
```

```
    self.Account_number = value
```

```
@property
```

```
def account_type(self):
```

```
    return self.Account_type
```

```
@account_type.setter
```

```
def account_type(self, value):
```

```
    self.Account_type = value
```

```
@property
```

```
def account_balance(self):
```

```
    return self.Account_balance
```

```
@account_balance.setter
```

```
def account_balance(self, value):
```

```
    self.Account_balance = value
```

```
@property
```

```
def customer(self):
```

```
    return self.Customer
```

```
@customer.setter
```

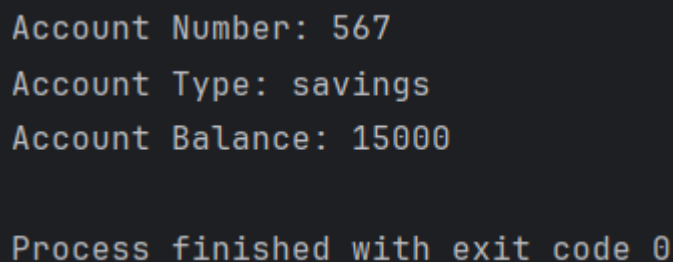
```
def customer(self, value):
```

```
self.Customer = value
```

```
def display(self):  
    print("Account Number:", self.account_number)  
    print("Account Type:", self.account_type)  
    print("Account Balance:", self.account_balance)
```

```
account = Account(567,"savings",15000)
```

```
account.display()
```



```
Account Number: 567  
Account Type: savings  
Account Balance: 15000  
  
Process finished with exit code 0
```

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.

#### 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.



```
class Customer:

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

        self.customer_id = customer_id

        self.first_name = first_name

        self.last_name = last_name

        self.email = email

        self.phone_number = phone_number

        self.address = address
```

```
class Account:

    last_acc_no = 0

    def __init__(self, account_type, initial_balance, customer):

        Account.last_acc_no += 1

        self.account_number = Account.last_acc_no

        self.account_type = account_type

        self.balance = initial_balance

        self.customer = customer
```

```
class SavingsAccount(Account):

    def __init__(self, initial_balance, customer, interest_rate=0.08):

        super().__init__('Savings', initial_balance, customer)

        if initial_balance < 500:

            raise ValueError("Minimum balance for Savings Account must be 500")

        self.interest_rate = interest_rate
```

```
class CurrentAccount(Account):

    def __init__(self, initial_balance, customer, overdraft_limit=10000):

        super().__init__('Current', initial_balance, customer)
```

```
self.overdraft_limit = overdraft_limit
```

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

```
    if amount > self.balance + self.overdraft_limit:
```

```
        print("Withdrawal amount exceeds available balance and overdraft limit.")
```

```
    else:
```

```
        self.balance -= amount
```

```
class ZeroBalanceAccount(Account):
```

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

```
        super().__init__('Zero Balance', 0, customer)
```

```
customer1 = Customer(1, "Sne", "Barani", "sne@gmail.com", "1234512345", "Coimbatore")
```

```
savings_acc = SavingsAccount(1000, customer1.first_name)
```

```
print("Savings Account Number:", savings_acc.account_number)
```

```
print("Savings Account Balance:", savings_acc.balance)
```

```
customer2 = Customer(2, "Sangee", "Barani", "san@gmail.com", "5432154321", "Blr")
```

```
current_acc = CurrentAccount(50000, customer2.first_name)
```

```
print("Current Account Number:", current_acc.account_number)
```

```
print("Current Account Balance:", current_acc.balance)
```

```
current_acc.withdraw(800)
```

```
print("Current Account Balance after withdrawal:", current_acc.balance)
```

```
zero_balance_acc = ZeroBalanceAccount(customer2.first_name)
```

```
print("Zero Balance Account Number:", zero_balance_acc.account_number)
```

```
print("Zero Balance Account Balance:", zero_balance_acc.balance)
```

```
Savings Account Number: 1
Savings Account Balance: 1000
Current Account Number: 2
Current Account Balance: 50000
Current Account Balance after withdrawal: 49200
Zero Balance Account Number: 3
Zero Balance Account Balance: 0

Process finished with exit code 0
```

2. 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):
```

```
        pass
```

```
    @abstractmethod
```

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

```
        pass
```

```
@abstractmethod
```

```
def withdraw(self, account_number, amount):  
    pass
```

```
@abstractmethod
```

```
def transfer(self, from_account_number, to_account_number, amount):  
    pass
```

```
@abstractmethod
```

```
def get_account_details(self, account_number):  
    pass
```

3. 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.

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

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

```
@abstractmethod
```

```
def create_account(self, customer, accNo, accType, balance):  
    pass
```

```
@abstractmethod
```

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

```
@abstractmethod
```

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

```
    pass
```

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

5. 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

6. 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.

7. 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.

8. 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 throw this exception when current account customer try to with draw amount from the current account.
4. NullPointerException handle in main method.

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

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

```
    pass
```

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

```
    pass
```

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

```
    pass
```

```
class Account:
```

```
    def __init__(self, account_type, account_number, balance=0, overdraft_limit=0):
```

```
        self.account_type = account_type
```

```
        self.account_number = account_number
```

```
        self.balance = balance
```

```
        self.overdraft_limit = overdraft_limit
```

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

```
        self.balance += amount
```

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

```
        if self.account_type == "SavingsAccount":
```

```
            if self.balance < amount:
```

```
                raise InsufficientFundException("Insufficient balance in the account.")
```

```
            else:
```

```

        self.balance -= amount

    elif self.account_type == "CurrentAccount":
        if amount > (self.balance + self.overdraft_limit):
            raise OverDraftLimitExceededException("Withdrawal amount exceeds the
overdraft limit.")
        else:
            self.balance -= amount

    def calculate_interest(self, interest_rate):
        if self.account_type == "SavingsAccount":
            interest = self.balance * interest_rate
            self.balance += interest

def main():
    try:
        account_type = input("Enter account type (SavingsAccount/CurrentAccount): ")
        account_number = int(input("Enter account number: "))
        if account_type not in ["SavingsAccount", "CurrentAccount"]:
            raise InvalidAccountException("Invalid account type.")

        if account_type == "SavingsAccount":
            interest_rate = float(input("Enter interest rate for savings account: "))
            account = Account(account_type, account_number)
        elif account_type == "CurrentAccount":
            overdraft_limit = float(input("Enter overdraft limit for current account: "))
            account = Account(account_type, account_number, overdraft_limit=overdraft_limit)

    while True:
        print("\n1. Deposit")
        print("2. Withdraw")
        print("3. Calculate Interest (SavingsAccount)")

```

```

print("4. Exit")

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

if choice == 1:
    amount = float(input("Enter amount to deposit: "))
    account.deposit(amount)
    print("Deposit successful. Current balance:", account.balance)
elif choice == 2:
    amount = float(input("Enter amount to withdraw: "))
    account.withdraw(amount)
    print("Withdrawal successful. Current balance:", account.balance)
elif choice == 3 and account._type == "SavingsAccount":
    account.calculate_interest(interest_rate)
    print("Interest calculated. Current balance:", account.balance)
elif choice == 4:
    break
else:
    print("Invalid choice. Please try again.")

except InsufficientFundException as e:
    print("Error:", e)
except InvalidAccountException as e:
    print("Error:", e)
except OverDraftLimitExceededException as e:
    print("Error:", e)
except ValueError:
    print("Invalid input. Please enter a valid number.")
except Exception as e:
    print("An error occurred:", e)

```



main()

```
Enter account type (SavingsAccount/CurrentAccount): CurrentAccount
Enter account number: 35
Enter overdraft limit for current account: 80000

1. Deposit
2. Withdraw
3. Calculate Interest (SavingsAccount)
4. Exit
Enter your choice: 1
Enter amount to deposit: 4500
Deposit successful. Current balance: 4500.0
```

### Task 13: Collection

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

```
class BankAccount:
    def __init__(self, account_number, customer_name, balance):
        self.account_number = account_number
        self.customer_name = customer_name
        self.balance = balance

    def deposit(self, amount):
        self.balance += amount

    def withdraw(self, amount):
        if amount <= self.balance:
            self.balance -= amount
            print("Balance After withdrawal: ", self.balance)
        else:
            print("Insufficient balance")

    def interest(self):
        intrate = float(input("Enter the interest rate: "))
        intamount = self.balance * (intrate / 100)
        self.balance += intamount
        print("Balance with interest: ", self.balance)

    def display(self):
        print("Account Number:", self.account_number)
```

```

        print("Customer Name:", self.customer_name)
        print("Account Balance:", self.balance)

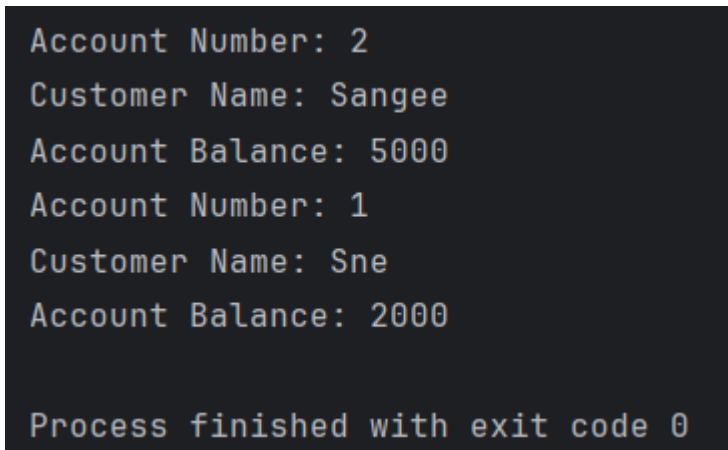
class Bank:
    def __init__(self):
        self.accounts = []

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

    def list_accounts(self):
        self.accounts.sort(key=lambda acc: acc.customer_name)
        for account in self.accounts:
            account.display()

bank = Bank()
acc1 = BankAccount(1, "Sne", 2000)
acc2 = BankAccount(2, "Sangee", 5000)
bank.add_account(acc1)
bank.add_account(acc2)
bank.list_accounts()

```



```

Account Number: 2
Customer Name: Sangee
Account Balance: 5000
Account Number: 1
Customer Name: Sne
Account Balance: 2000

Process finished with exit code 0

```

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.

```

class BankAccount:
    def __init__(self, account_number, customer_name, balance):

```

```

        self.account_number = account_number
        self.customer_name = customer_name
        self.balance = balance

    def deposit(self, amount):
        self.balance += amount

    def withdraw(self, amount):
        if amount <= self.balance:
            self.balance -= amount
            print("Balance After withdrawal: ", self.balance)
        else:
            print("Insufficient balance")

    def interest(self):
        intrate = float(input("Enter the interest rate: "))
        intamount = self.balance * (intrate / 100)
        self.balance += intamount
        print("Balance with interest: ", self.balance)

    def display(self):
        print("Account Number:", self.account_number)
        print("Customer Name:", self.customer_name)
        print("Account Balance:", self.balance)

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

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

    def list_accounts(self):
        print("Using Set")
        sorted_accounts = sorted(self.accounts, key=lambda acc: acc.customer_name)
        for account in sorted_accounts:
            account.display()

bank = Bank()
acc1 = BankAccount(1, "Sne", 2000)
acc2 = BankAccount(2, "Sangee", 5000)
bank.add_account(acc1)
bank.add_account(acc2)

```

```
bank.list_accounts()
```