NAME : GAYATHRI G

TOPIC : Banking System

Task 1: Conditional Statements

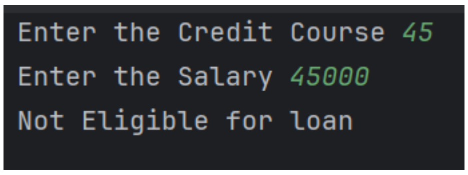
In a bank, you have been given the task is to create a program that checks if a customer is eligible for a loan based on their credit score and income. The eligibility criteria are as follows:

- 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")
```



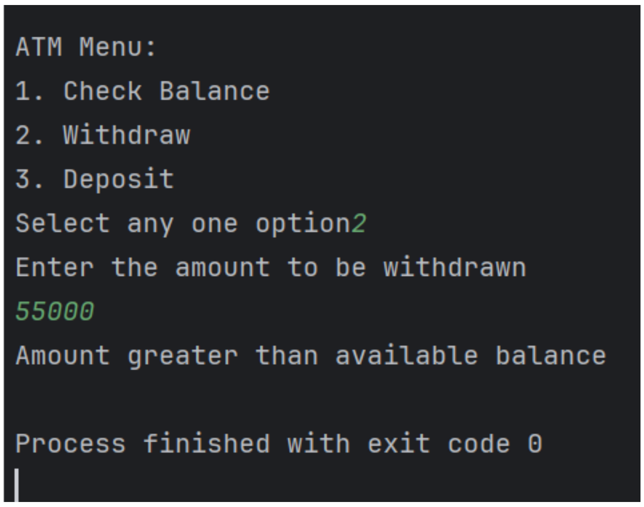
```
Enter the Credit Course 45
Enter the Salary 45000
Not Eligible for loan
```

Task 2: Nested Conditional Statements

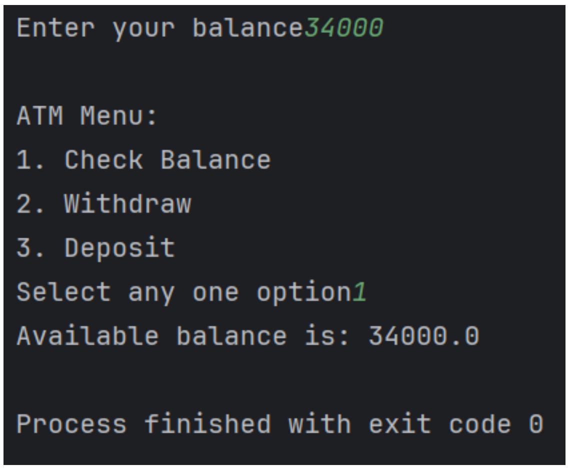
Create a program that simulates an ATM transaction. Display options such as "Check Balance," "Withdraw," "Deposit,". Ask the user to enter their current balance and the amount they want to withdraw or deposit. Implement checks to ensure that the withdrawal amount is not greater than the available balance 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   print("Enter the amount to be withdrawn")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         print("amount withdrawn")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")
```



```
ATM Menu:
1. Check Balance
2. Withdraw
3. Deposit
Select any one option2
Enter the amount to be withdrawn
55000
Amount greater than available balance

Process finished with exit code 0
```



```
Enter your balance34000

ATM Menu:
1. Check Balance
2. Withdraw
3. Deposit
Select any one option1
Available balance is: 34000.0

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.

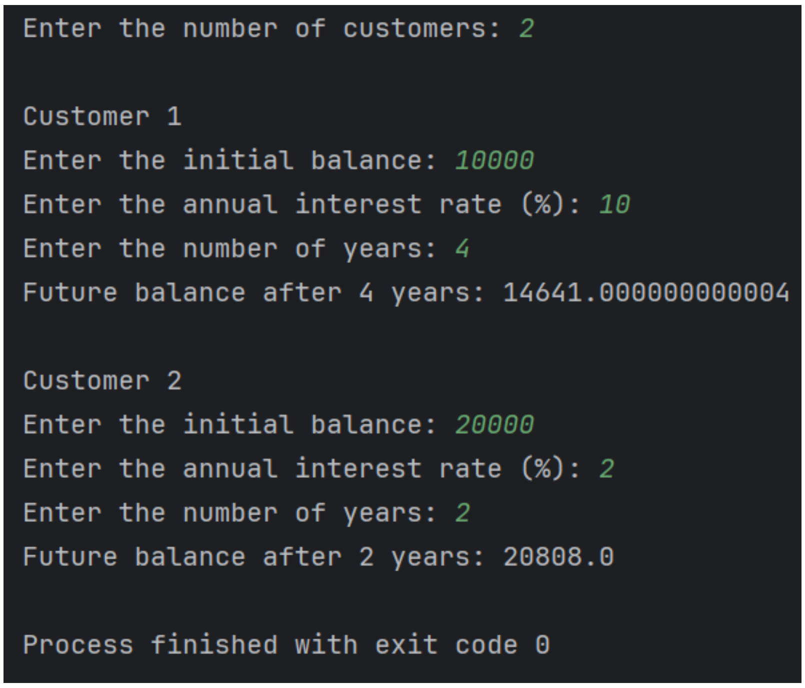
Tasks:

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:

   *future_balance = initial_balance * (1 + annual_interest_rate/100)^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: 10000
Enter the annual interest rate (%): 10
Enter the number of years: 4
Future balance after 4 years: 14641.000000000004

Customer 2
Enter the initial balance: 20000
Enter the annual interest rate (%): 2
Enter the number of years: 2
Future balance after 2 years: 20808.0

Process finished with exit code 0
```

Task 4: Looping, Array and Data Validation

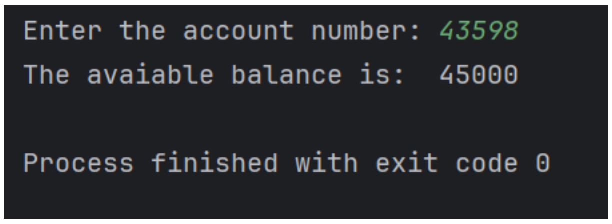
You are tasked with creating a program that allows bank customers to check their account balances. The program should handle multiple customer accounts, and the customer should be able to enter their account number, balance to check the balance.

Tasks:

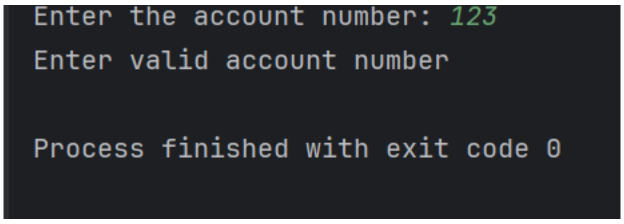
1. Create a Python program that simulates a bank with multiple customer accounts.
2. Use a loop (e.g., while loop) to repeatedly ask the user for their account number and balance until they enter a valid account number.

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 avaiable balance is: ", account_details.get(account_number))
      break
   else:
      print("Enter valid account number ")
      break
```



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

Process finished with exit code 0
```



```
Enter the account number: 123
Enter valid account number

Process finished with exit code 0
```

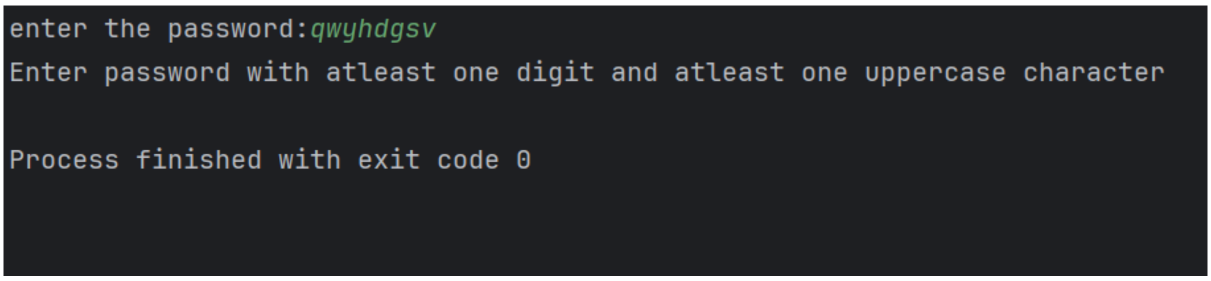
Task 5: Password Validation

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:

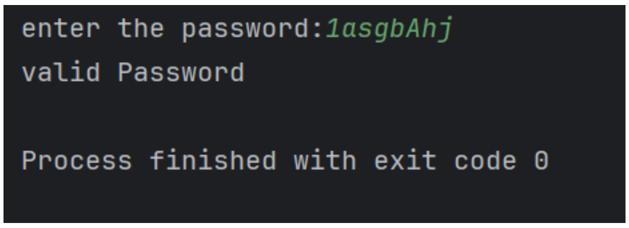
- The password must be at least 8 characters long.
- It must contain at least one uppercase letter.
- It must contain at least one digit.
- Display appropriate messages to indicate whether their password is valid or not.

```
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")
```



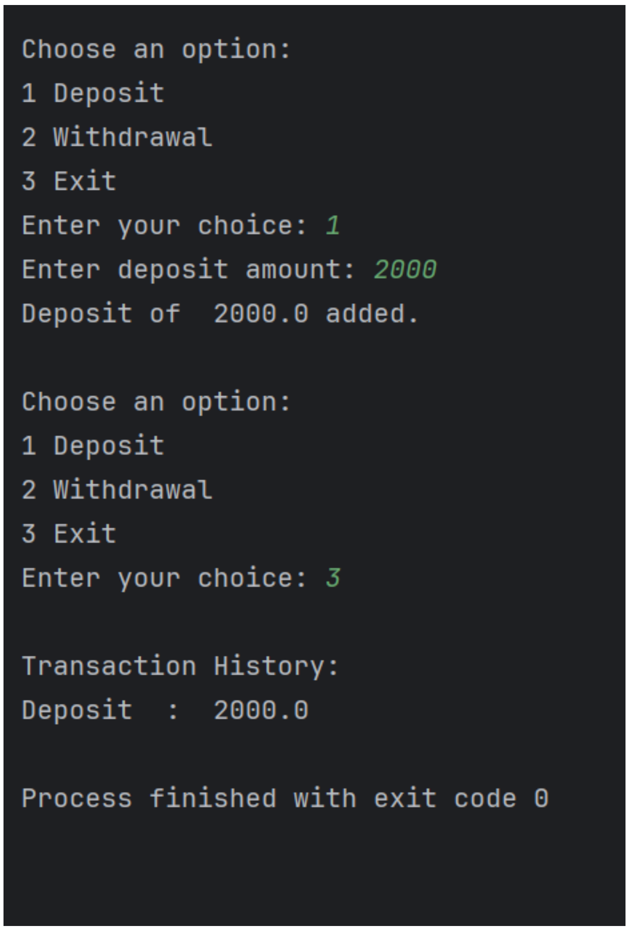
```
enter the password:qwyhdgsv
Enter password with atleast one digit and atleast one uppercase character

Process finished with exit code 0
```



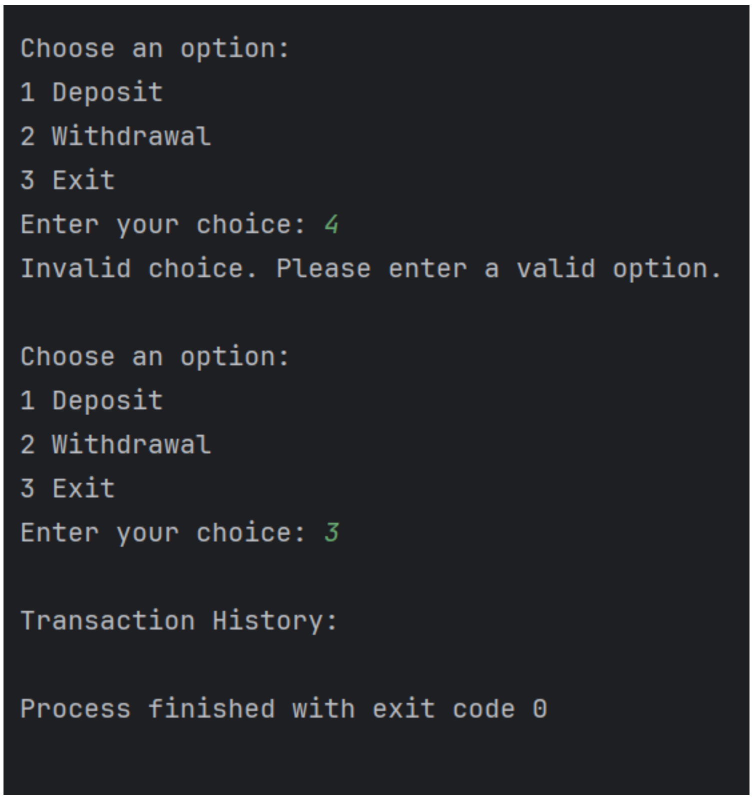
```
enter the password:1asgbAhj
valid Password

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: 2000
Deposit of  2000.0 added.

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

Transaction History:
Deposit  :  2000.0

Process finished with exit code 0
```



```
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:

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

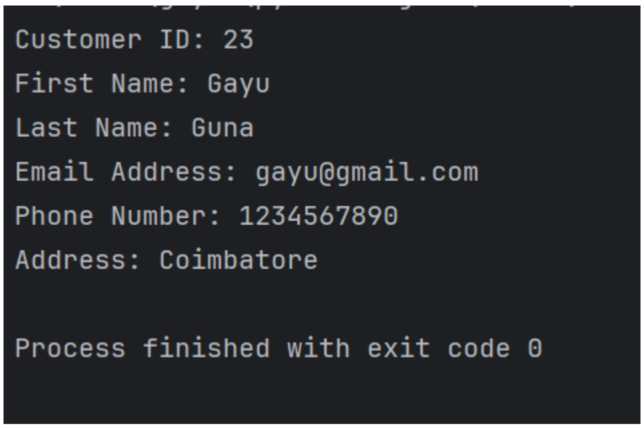
     ○ Customer ID ○ First Name ○ Last Name ○ Email

     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):
       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("23", "Gayu", "Guna", "gayu@gmail.com", "1234567890", "Coimbatore")
cust.details()
```



```
Customer ID: 23
First Name: Gayu
Last Name: Guna
Email Address: gayu@gmail.com
Phone Number: 1234567890
Address: Coimbatore

Process finished with exit code 0
```

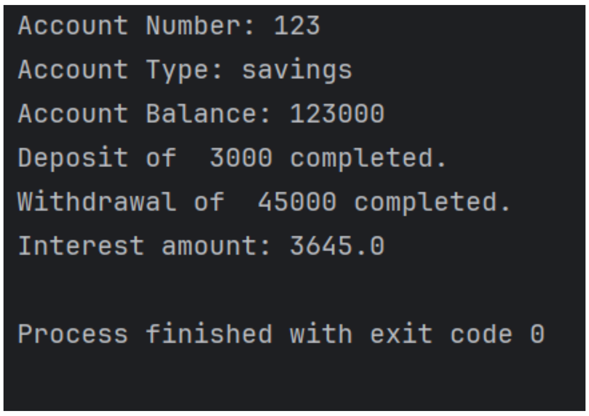
2. Create an `Account` class with the following confidential attributes:
   - **Attributes**
     - Account Number
     - Account Type (e.g., Savings, Current)
     - Account Balance
   - **Constructor and Methods**
     - Implement default constructors and overload the constructor with Account attributes,
     - 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%

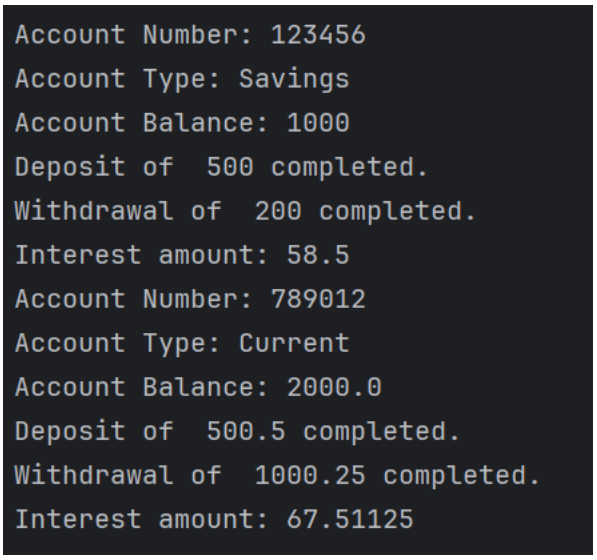
```
class Accounts:
  def __init__(self,Account_number,Account_type,Account_balance):
    self._Account_number=Account_number
    self._Account_type=Account_type
    self._Account_balance=Account_balance
  @property
  def Account_number(self):
    return self._Account_number
  @Account_number.setter
  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(123,"savings",123000)
obj.display()
obj.deposit(3000)
obj.withdraw(45000)
obj.calculate_interest()
```


```
Account Number: 123
Account Type: savings
Account Balance: 123000
Deposit of  3000 completed.
Withdrawal of  45000 completed.
Interest amount: 3645.0

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()
```

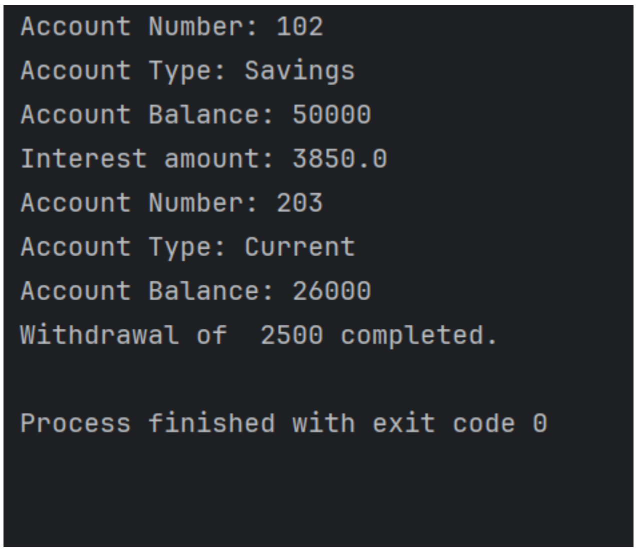


```
Account Number: 123456
Account Type: Savings
Account Balance: 1000
Deposit of  500 completed.
Withdrawal of  200 completed.
Interest amount: 58.5
Account Number: 789012
Account Type: Current
Account Balance: 2000.0
Deposit of  500.5 completed.
Withdrawal of  1000.25 completed.
Interest amount: 67.51125
```

2. Create Subclasses for Specific Account Types
    - Create subclasses for specific account types (e.g., `SavingsAccount`, `CurrentAccount`) that inherit from the `Account` class.
        - **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.
        - **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)
```

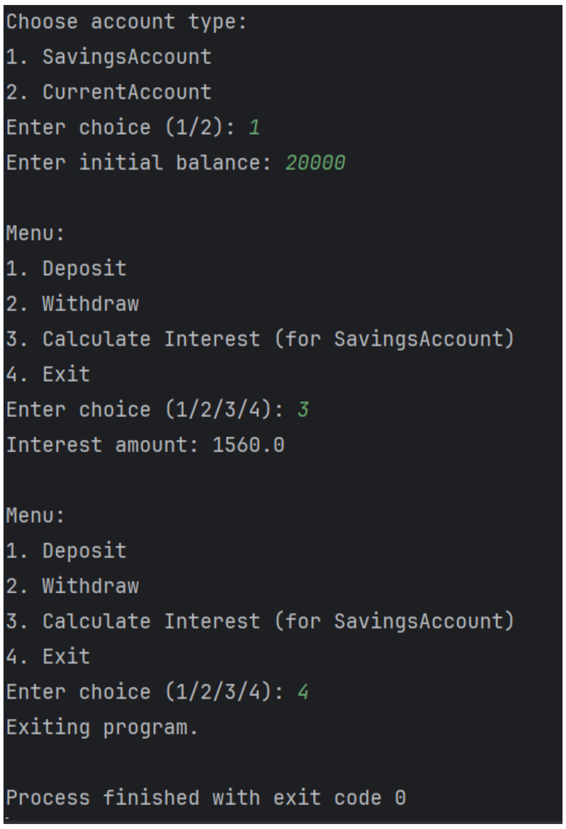


```
Account Number: 102
Account Type: Savings
Account Balance: 50000
Interest amount: 3850.0
Account Number: 203
Account Type: Current
Account Balance: 26000
Withdrawal of  2500 completed.

Process finished with exit code 0
```

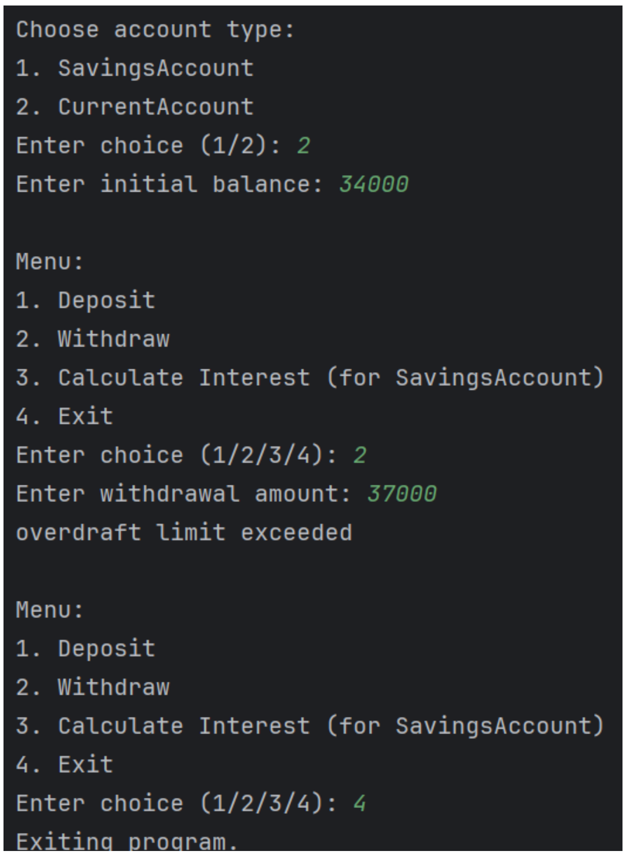
3. Create a **Bank** class to represent the banking system. Perform the following operation in main method:
    - Display menu for user to create object for account class by calling parameter constructor. Menu should display options `SavingsAccount` and `CurrentAccount`. user can choose any one option to create account. use switch case for implementation.
    - **deposit(amount: float):** Deposit the specified amount into the account.
    - **withdraw(amount: float):** Withdraw the specified amount from the account. For saving account withdraw amount only if there is sufficient fund else display insufficient balance.

      For Current Account withdraw limit can exceed the available balance and should not exceed the overdraft limit.
    - **calculate_interest():** Calculate and add interest to the account balance for savings accounts.

```
class 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: 20000

Menu:
1. Deposit
2. Withdraw
3. Calculate Interest (for SavingsAccount)
4. Exit
Enter choice (1/2/3/4): 3
Interest amount: 1560.0

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
```



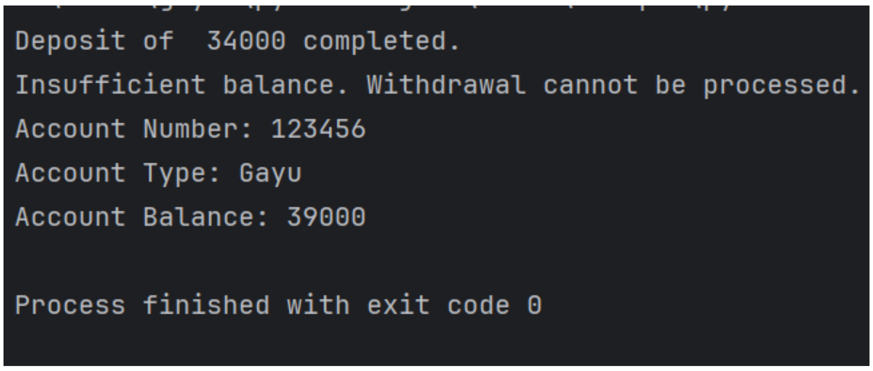
```
Choose account type:
1. SavingsAccount
2. CurrentAccount
Enter choice (1/2): 2
Enter initial balance: 34000

Menu:
1. Deposit
2. Withdraw
3. Calculate Interest (for SavingsAccount)
4. Exit
Enter choice (1/2/3/4): 2
Enter withdrawal amount: 37000
overdraft limit exceeded

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

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:
        - 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:
        - **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.

```
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(123456,"Gayu",5000)
Bank.Deposit(34000)
Bank.Withdrawl(50000)
```

Bank.display()

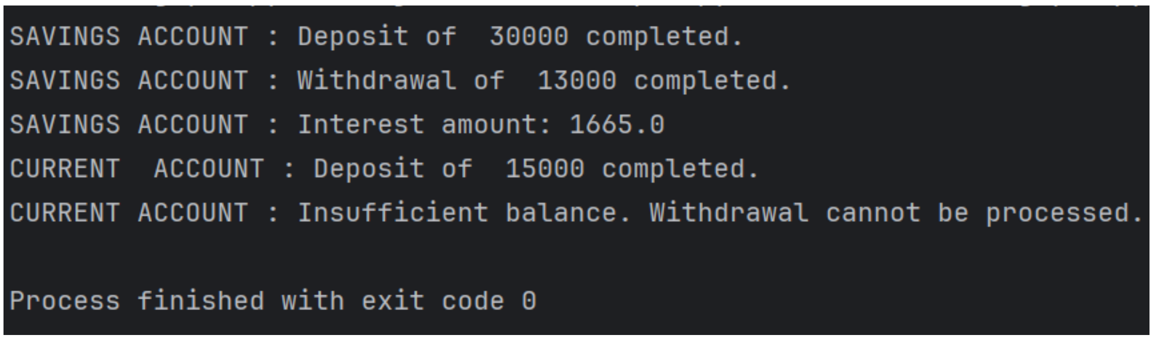


```
Deposit of  34000 completed.
Insufficient balance. Withdrawal cannot be processed.
Account Number: 123456
Account Type: Gayu
Account Balance: 39000

Process finished with exit code 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 Withdrawl(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(123,"Gayu",20000)
savings.Deposit(30000)
savings.Withdrawl(13000)
savings.interest()
```

```
current = CurrentAccount(456,"swathi",45000)
current.Deposit(15000)
current.Withdrawl(100000)
```

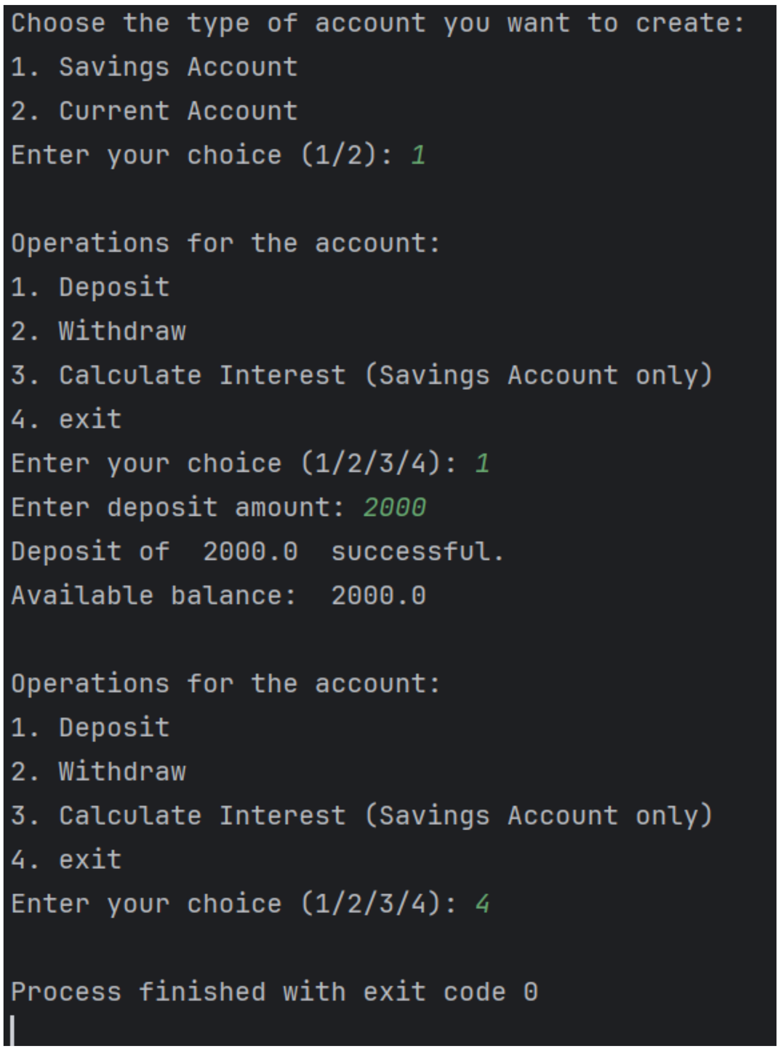


```
SAVINGS ACCOUNT : Deposit of  30000 completed.
SAVINGS ACCOUNT : Withdrawal of  13000 completed.
SAVINGS ACCOUNT : Interest amount: 1665.0
CURRENT  ACCOUNT : Deposit of  15000 completed.
CURRENT ACCOUNT : Insufficient balance. Withdrawal cannot be processed.

Process finished with exit code 0
```

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.

```
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()
```



```
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. exit
Enter your choice (1/2/3/4): 1
Enter deposit amount: 2000
Deposit of  2000.0  successful.
Available balance:  2000.0

Operations for the account:
1. Deposit
2. Withdraw
3. Calculate Interest (Savings Account only)
4. exit
Enter your choice (1/2/3/4): 4

Process finished with exit code 0
```

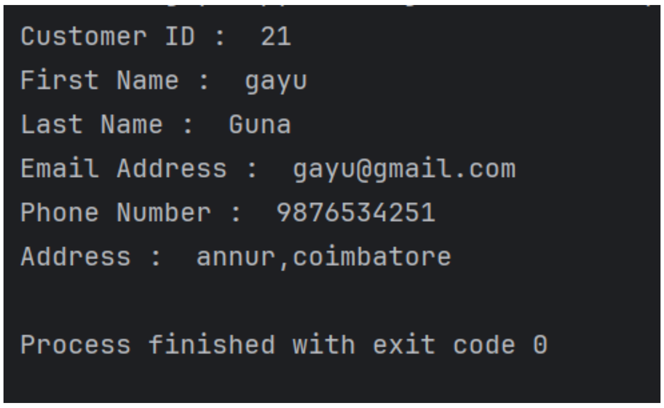
Task 10: Has A Relation / Association

1. Create a `Customer` class with the following attributes:
    - Customer ID
    - First Name
    - Last Name
    - Email Address (validate with valid email address)
    - Phone Number (Validate 10-digit phone number)
    - Address
    - Methods and Constructor:
        - 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
```

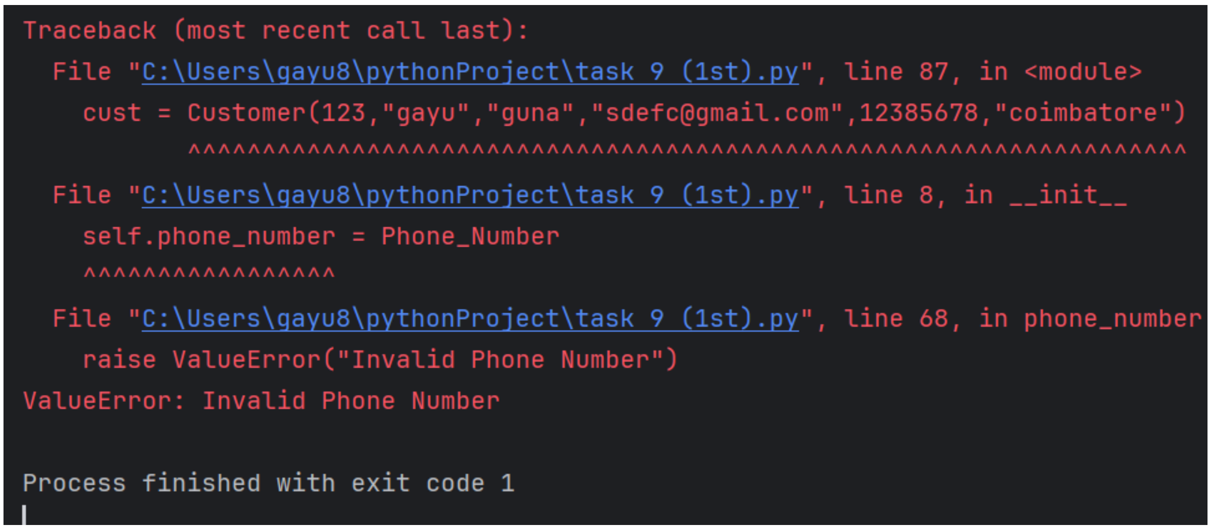
```
@last_name.setter
def last_name(self, ln):
    self.Last_name = ln

@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(123,"gayu","guna","sdefc@gmail.com",12385678,"coimbatore")
cust.display()
```



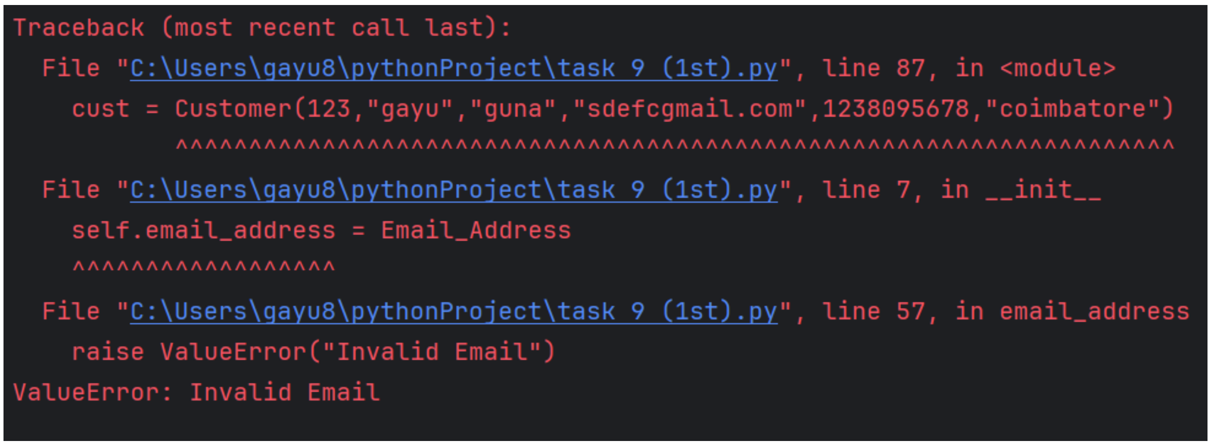
```
Customer ID :  21
First Name :  gayu
Last Name :  Guna
Email Address :  gayu@gmail.com
Phone Number :  9876534251
Address :  annur,coimbatore

Process finished with exit code 0
```

PHONE NUMBER VALIDATION:



```
Traceback (most recent call last):
  File "C:\Users\gayu8\pythonProject\task 9 (1st).py", line 87, in <module>
    cust = Customer(123,"gayu","guna","sdefc@gmail.com",12385678,"coimbatore")
           ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "C:\Users\gayu8\pythonProject\task 9 (1st).py", line 8, in __init__
    self.phone_number = Phone_Number
    ^^^^^^^^^^^^^^^^^
  File "C:\Users\gayu8\pythonProject\task 9 (1st).py", line 68, in phone_number
    raise ValueError("Invalid Phone Number")
ValueError: Invalid Phone Number

Process finished with exit code 1
```

EMAIL VALIDATION:

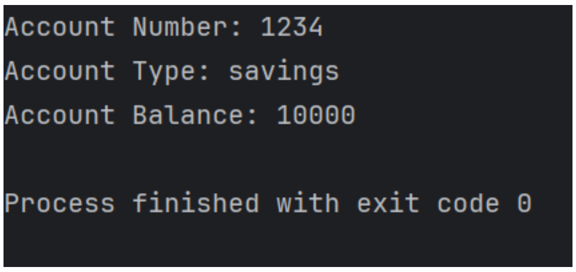


```
Traceback (most recent call last):
  File "C:\Users\gayu8\pythonProject\task 9 (1st).py", line 87, in <module>
    cust = Customer(123,"gayu","guna","sdefcgmail.com",1238095678,"coimbatore")
           ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "C:\Users\gayu8\pythonProject\task 9 (1st).py", line 7, in __init__
    self.email_address = Email_Address
    ^^^^^^^^^^^^^^^^^^
  File "C:\Users\gayu8\pythonProject\task 9 (1st).py", line 57, in email_address
    raise ValueError("Invalid Email")
ValueError: Invalid Email
```

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(1234,"savings",10000)
account.display()
```



```
Account Number: 1234
Account Type: savings
Account Balance: 10000

Process finished with exit code 0
```

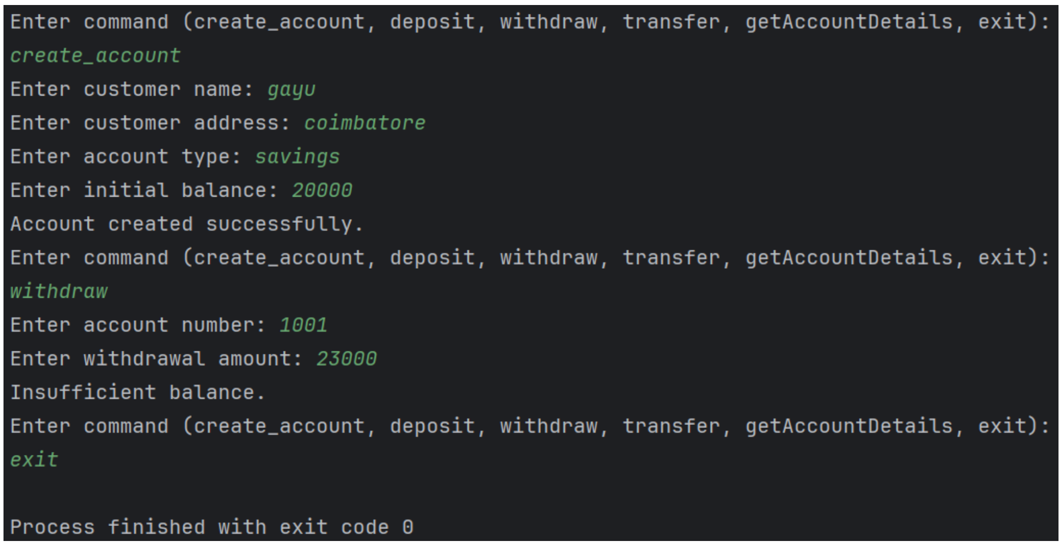
Create a Bank Class and must have following requirements:

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.

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


class Account:
    def __init__(self, customer, account_number, account_type, balance):
        self.customer = customer
        self.account_number = account_number
        self.account_type = account_type
        self.balance = balance

class Bank:
    def __init__(self):
        self.next_account_number = 1001
        self.accounts = {}

    def create_account(self, customer, acc_type, balance):
        acc_no = self.next_account_number
        self.next_account_number += 1
        account = Account(customer, acc_no, acc_type, balance)
        self.accounts[acc_no] = account

    def get_account_balance(self, account_number):
        return self.accounts.get(account_number).balance if account_number in self.accounts else -
1

    def deposit(self, account_number, amount):
        if account_number in self.accounts:
            self.accounts[account_number].balance += amount
            return self.accounts[account_number].balance
        return -1
    def withdraw(self, account_number, amount):
        if account_number in self.accounts:
            if self.accounts[account_number].balance >= amount:
                self.accounts[account_number].balance -= amount
                return self.accounts[account_number].balance
            else:
                return -2  # Insufficient balance
        return -1  # Account not found

    def transfer(self, from_account_number, to_account_number, amount):
        if from_account_number in self.accounts and to_account_number in self.accounts:
            if self.accounts[from_account_number].balance >= amount:
                self.accounts[from_account_number].balance -= amount
                self.accounts[to_account_number].balance += amount
                return True
        return False

    def get_account_details(self, account_number):
        if account_number in self.accounts:
            account = self.accounts[account_number]
            return f"Account Details: \nAccount Number: {account.account_number}\nAccount Type:
{account.account_type}\nBalance: {account.balance}\nCustomer Details: \nName:
{account.customer.name}\nAddress: {account.customer.address}"
        return "Account not found"

class BankApp:
    def __init__(self):
        self.bank = Bank()

    def run(self):
        while True:
            print("Enter command (create_account, deposit, withdraw, transfer, getAccountDetails,
exit): ")
            command = input().strip()
            if command == "exit":
                break
            elif command == "create_account":
                name = input("Enter customer name: ")
                address = input("Enter customer address: ")
                customer = Customer(name, address)
                acc_type = input("Enter account type: ")
                balance = float(input("Enter initial balance: "))
                self.bank.create_account(customer, acc_type, balance)
                print("Account created successfully.")
            elif command == "deposit":
                acc_no = int(input("Enter account number: "))
                amount = float(input("Enter deposit amount: "))
                balance = self.bank.deposit(acc_no, amount)
                if balance != -1:
                    print("Deposit successful. Current balance: ",balance)
                else:
                    print("Account not found.")
            elif command == "withdraw":
                acc_no = int(input("Enter account number: "))
                amount = float(input("Enter withdrawal amount: "))
                balance = self.bank.withdraw(acc_no, amount)
                if balance == -1:
                    print("Account not found.")
                elif balance == -2:
                    print("Insufficient balance.")
                else:
                    print("Withdrawal successful. Current balance: ",balance)
            elif command == "transfer":
                from_acc = int(input("Enter source account number: "))
                to_acc = int(input("Enter destination account number: "))
                amount = float(input("Enter transfer amount: "))
                if self.bank.transfer(from_acc, to_acc, amount):
                    print("Transfer successful.")
                else:
                    print("Transfer failed. Please check account numbers and balance.")
            elif command == "getAccountDetails":
                acc_no = int(input("Enter account number: "))
                details = self.bank.get_account_details(acc_no)
                print(details)
            else:
                print("Invalid command.")


bank_app = BankApp()
bank_app.run()
```



```
Enter command (create_account, deposit, withdraw, transfer, getAccountDetails, exit):
create_account
Enter customer name: gayu
Enter customer address: coimbatore
Enter account type: savings
Enter initial balance: 20000
Account created successfully.
Enter command (create_account, deposit, withdraw, transfer, getAccountDetails, exit):
withdraw
Enter account number: 1001
Enter withdrawal amount: 23000
Insufficient balance.
Enter command (create_account, deposit, withdraw, transfer, getAccountDetails, exit):
exit

Process finished with exit code 0
```

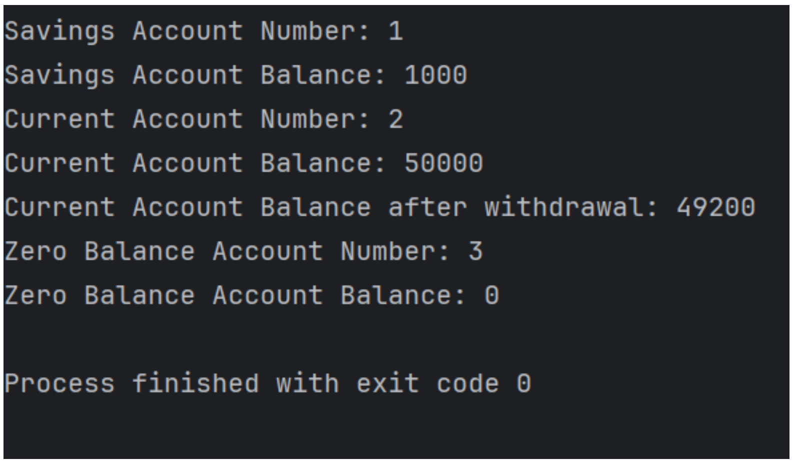
Task 11: Interface/abstract class, and Single Inheritance, static variable

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, "Gayathri", "Guna", "gayu@gmail.com", "1234567890", "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, "swathi", "Guna", "swa@gmail.com", "9001014320", "Salem")
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.

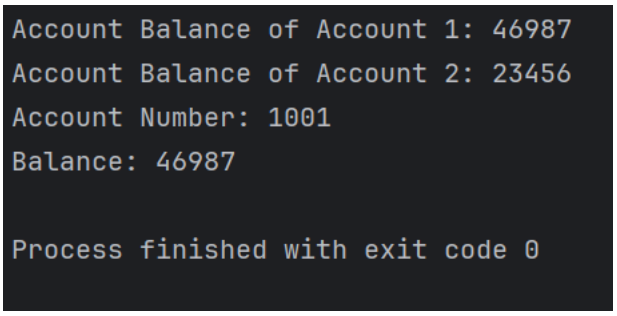
```
from abc import ABC, abstractmethod

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 I**CustomerServiceProvider** provide all implementation methods.

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

    def get_account_balance(self, account_number):
        if account_number in self.accounts:
            return self.accounts[account_number]
        else:
            print("Account not found.")
            return 0

    def deposit(self, account_number: int, amount):
        if account_number in self.accounts:
            self.accounts[account_number] += amount
            print("Deposit successful. Current balance:", self.accounts[account_number])
            return self.accounts[account_number]
        else:
            print("Account not found. Deposit failed.")
            return 0.0

    def withdraw(self, account_number: int, amount):
        if account_number in self.accounts:
            if amount <= self.accounts[account_number]:
                self.accounts[account_number] -= amount
                print("Withdrawal successful. Current balance:", self.accounts[account_number])
                return self.accounts[account_number]
            else:
                print("Insufficient balance for withdrawal.")
                return self.accounts[account_number]
        else:
            print("Account not found. Withdrawal failed.")
            return 0.0

    def transfer(self, from_account_number: int, to_account_number: int, amount: float):
        if from_account_number in self.accounts and to_account_number in self.accounts:
            if amount <= self.accounts[from_account_number]:
                self.accounts[from_account_number] -= amount
                self.accounts[to_account_number] += amount
                print("Transfer successful.")
            else:
                print("Insufficient balance for transfer.")
        else:
            print("One or both accounts not found. Transfer failed.")

    def get_account_details(self, account_number):
        if account_number in self.accounts:
            print("Account Number:", account_number)
            print("Balance:", self.accounts[account_number])

        else:
            print("Account not found.")


customer1 = CustomerServiceProviderImpl()
customer2 = CustomerServiceProviderImpl()
account1_number = 1001
account2_number = 1002
customerserviceprovider1 = CustomerServiceProviderImpl()
customerserviceprovider1.accounts[account1_number] = 46987
customerserviceprovider1.accounts[account2_number] = 23456
print("Account Balance of Account 1:",
customerserviceprovider1.get_account_balance(account1_number))
print("Account Balance of Account 2:",
customerserviceprovider1.get_account_balance(account2_number))
customerserviceprovider1.get_account_details(1001)
```



```
Account Balance of Account 1: 46987
Account Balance of Account 2: 23456
Account Number: 1001
Balance: 46987

Process finished with exit code 0
```

5. Create **BankApp** class and perform following operation:
    - main method to simulate the banking system. Allow the user to interact with the system by entering choice from menu such as "create_account", "deposit", "withdraw",

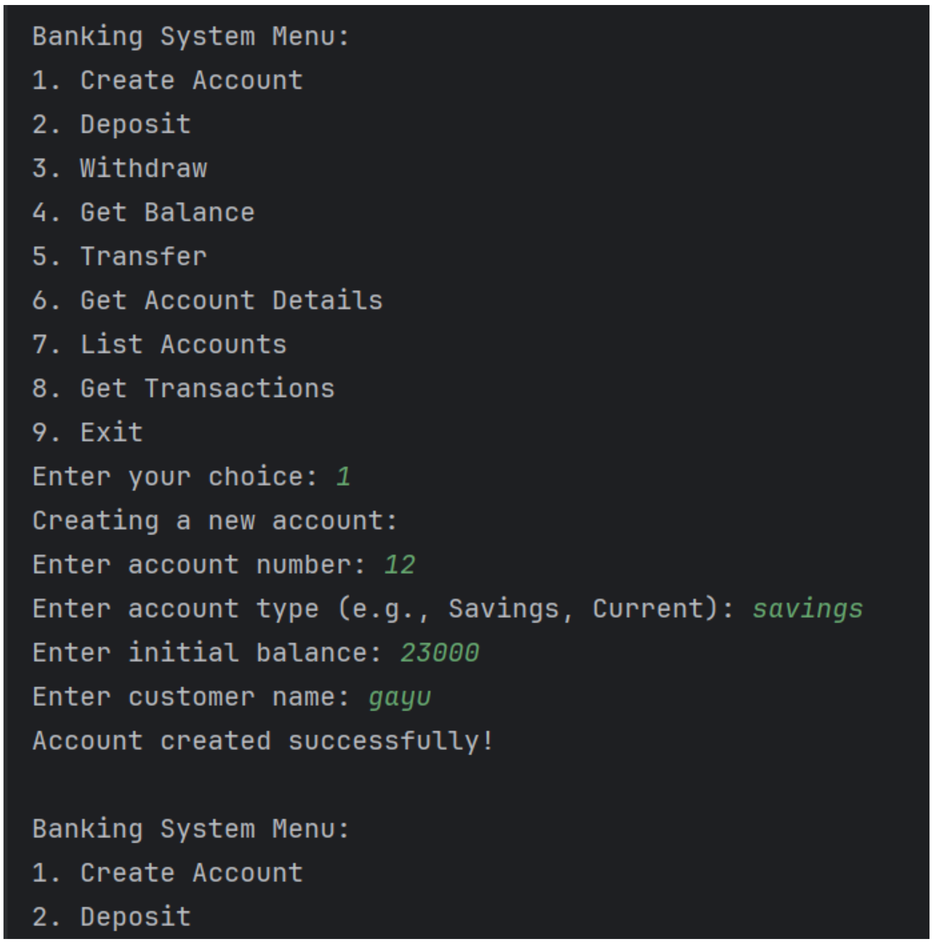
"get_balance", "transfer", "getAccountDetails", "ListAccounts" and "exit."

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

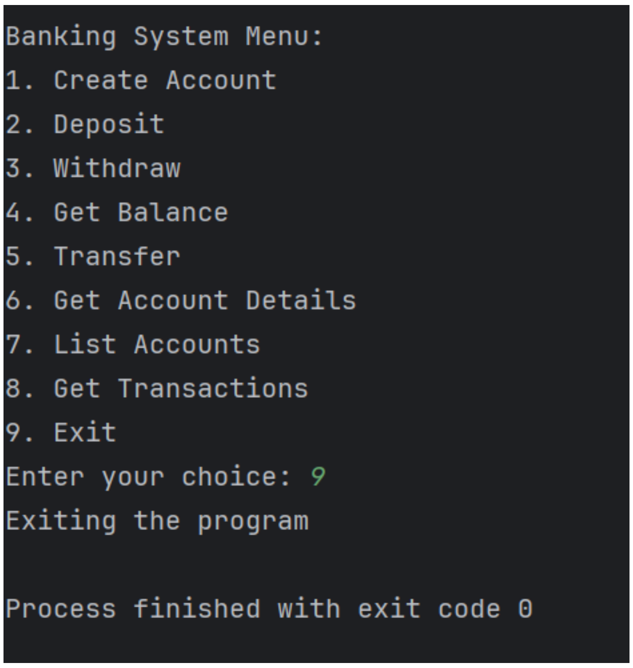
```
class BankApp:
    def __init__(self):
        self.accounts = []
    def create_account(self):
        print("Choose account type:")
        print("1. SavingsAccount")
        print("2. CurrentAccount")
        choice = input("Enter choice (1/2): ")
        if choice == "1":
            acc_type = "SavingsAccount"
        elif choice == "2":
            acc_type = "CurrentAccount"
        else:
            print("Invalid choice.")
            return None

        balance = float(input("Enter initial balance: "))
        account = BankApp()
        self.accounts.append(account)
        print("Account created successfully.")
        return account

    def deposit(self, amount):
        if amount > 0:
            self.balance += amount
            print("Deposit successful. Current balance:", self.balance)
        else:
            print("Invalid amount for deposit.")

    def display_accounts(self):
        print("List of Accounts:")
        for idx, account in enumerate(self.accounts):
            print(f"{idx + 1}. Account Type: {account.acc_type}, Balance:
{account.balance}")

    def get_balance(self):
        self.display_accounts()
```

```python
        idx = int(input("Enter account index: ")) - 1
        if 0 <= idx < len(self.accounts):
            print("Account balance:", self.accounts[idx].balance)
        else:
            print("Invalid account index.")

    def get_account_details(self):
        self.display_accounts()
        idx = int(input("Enter account index: ")) - 1
        if 0 <= idx < len(self.accounts):
            account_details = self.accounts[idx].get_account_details()
            print("Account details:")
            for key, value in account_details.items():
                print(f"{key}: {value}")
        else:
            print("Invalid account index.")

    def transfer(self):
        self.display_accounts()
        from_idx = int(input("Enter sender account index: ")) - 1
        to_idx = int(input("Enter receiver account index: ")) - 1
        if 0 <= from_idx < len(self.accounts) and 0 <= to_idx < len(self.accounts):
            amount = float(input("Enter transfer amount: "))
            if amount > 0:
                if self.accounts[from_idx].balance >= amount:
                    self.accounts[from_idx].balance -= amount
                    self.accounts[to_idx].balance += amount
                    print("Transfer successful.")
                else:
                    print("Insufficient funds.")
            else:
                print("Invalid transfer amount.")
        else:
            print("Invalid account index.")

    def main(self):
        while True:
            print("\nMenu:")
            print("1. Create Account")
            print("2. Deposit")
            print("3. Withdraw")
            print("4. Get Balance")
            print("5. Transfer")
            print("6. Get Account Details")
            print("7. List Accounts")
            print("8. Exit")
            choice = input("Enter choice: ")

            if choice == "1":
                self.create_account()
            elif choice == "2":
                idx = int(input("Enter account index: ")) - 1
                amount = float(input("Enter deposit amount: "))
                if 0 <= idx < len(self.accounts):
                    self.accounts[idx].deposit(amount)
                else:
                    print("Invalid account index.")
            elif choice == "3":
                self.display_accounts()
                idx = int(input("Enter account index: ")) - 1
                amount = float(input("Enter withdrawal amount: "))
                if 0 <= idx < len(self.accounts):
                    self.accounts[idx].withdraw(amount)
                else:
                    print("Invalid account index.")
            elif choice == "4":
                self.get_balance()
            elif choice == "5":
                self.transfer()
            elif choice == "6":
                self.get_account_details()
            elif choice == "7":
                self.display_accounts()
            elif choice == "8":
                print("Exiting program.")
                break
            else:
                print("Invalid choice.")

# Run the main method
if __name__ == "__main__":
    bank_app = BankApp()
```

bank_app.main()



```
Banking System Menu:
1. Create Account
2. Deposit
3. Withdraw
4. Get Balance
5. Transfer
6. Get Account Details
7. List Accounts
8. Get Transactions
9. Exit
Enter your choice: 1
Creating a new account:
Enter account number: 12
Enter account type (e.g., Savings, Current): savings
Enter initial balance: 23000
Enter customer name: gayu
Account created successfully!

Banking System Menu:
1. Create Account
2. Deposit
```



```
Banking System Menu:
1. Create Account
2. Deposit
3. Withdraw
4. Get Balance
5. Transfer
6. Get Account Details
7. List Accounts
8. Get Transactions
9. Exit
Enter your choice: 9
Exiting the program

Process finished with exit code 0
```

Task 12: Exception Handling

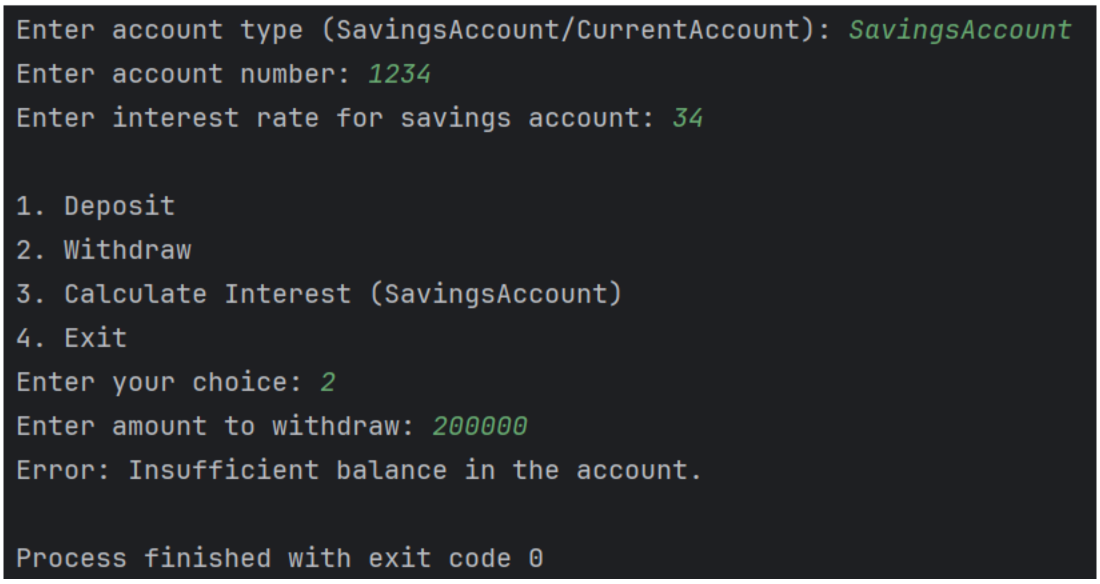
throw the exception whenever needed and Handle in main
method,

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**.**

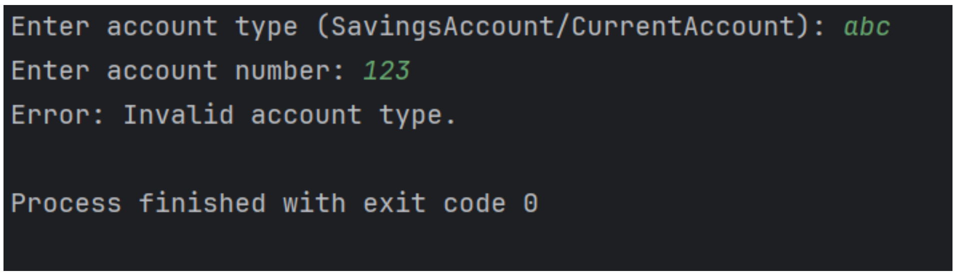
Throw these exceptions from the methods in HMBank class. Make necessary changes to accommodate these exception in the source code. Handle all these exceptions from the main program.

```
class 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:
```

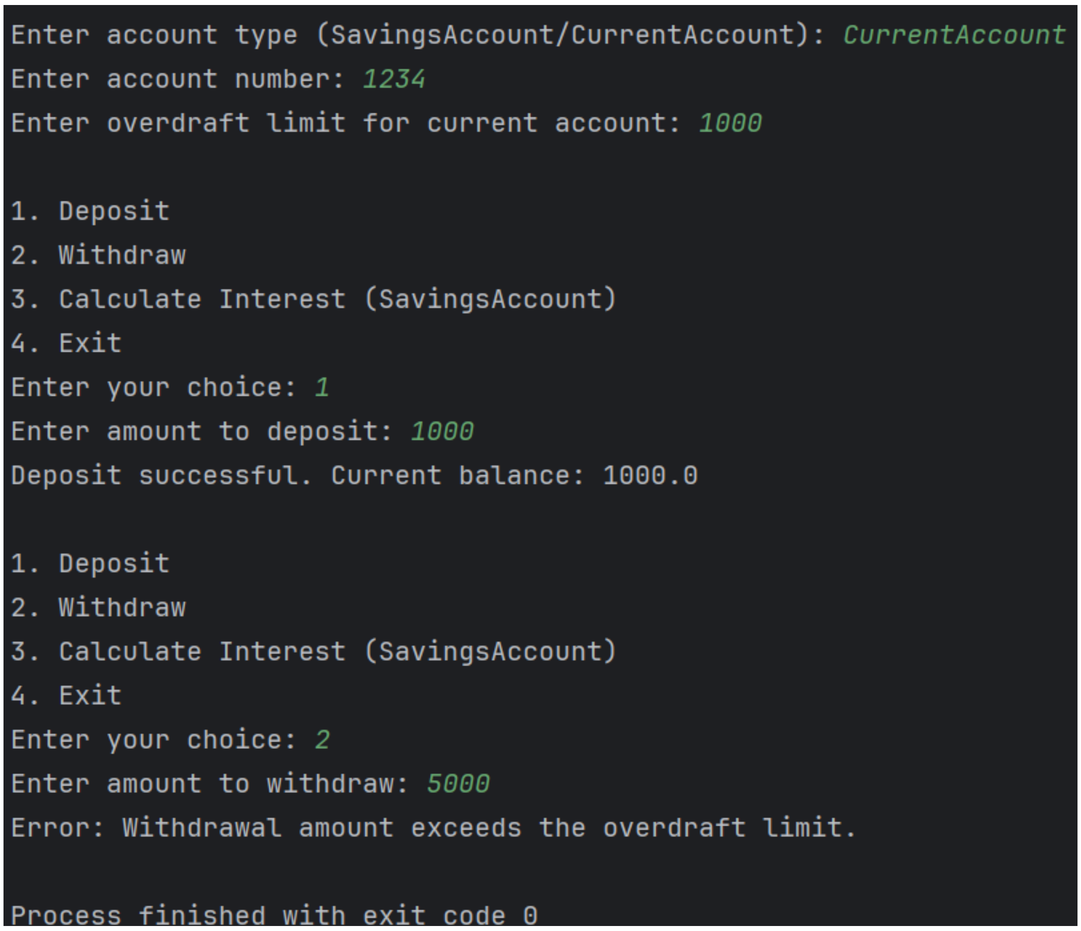
```
            self.balance -= amount

    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): SavingsAccount
Enter account number: 1234
Enter interest rate for savings account: 34

1. Deposit
2. Withdraw
3. Calculate Interest (SavingsAccount)
4. Exit
Enter your choice: 2
Enter amount to withdraw: 200000
Error: Insufficient balance in the account.

Process finished with exit code 0
```



```
Enter account type (SavingsAccount/CurrentAccount): abc
Enter account number: 123
Error: Invalid account type.

Process finished with exit code 0
```



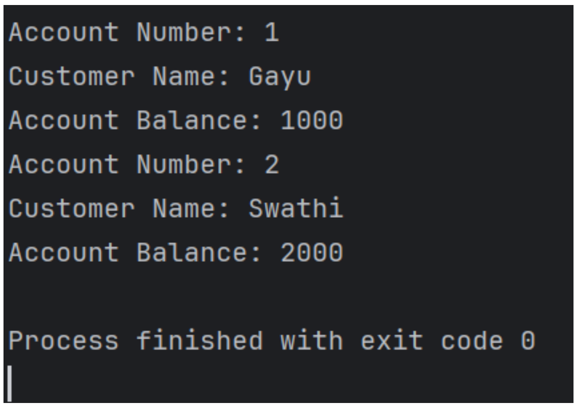
```
Enter account type (SavingsAccount/CurrentAccount): CurrentAccount
Enter account number: 1234
Enter overdraft limit for current account: 1000

1. Deposit
2. Withdraw
3. Calculate Interest (SavingsAccount)
4. Exit
Enter your choice: 1
Enter amount to deposit: 1000
Deposit successful. Current balance: 1000.0

1. Deposit
2. Withdraw
3. Calculate Interest (SavingsAccount)
4. Exit
Enter your choice: 2
Enter amount to withdraw: 5000
Error: Withdrawal amount exceeds the overdraft limit.

Process finished with exit code 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)
```

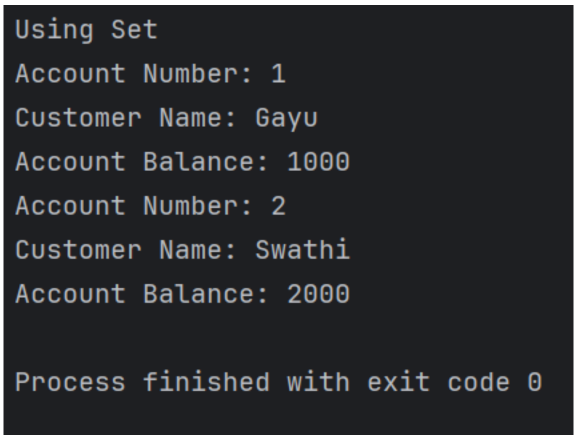
```
        else:
            print("Insufficient balance")

    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, "Gayu", 1000)
acc2 = BankAccount(2, "Swathi", 2000)
bank.add_account(acc1)
bank.add_account(acc2)
bank.list_accounts()
```


```
Account Number: 1
Customer Name: Gayu
Account Balance: 1000
Account Number: 2
Customer Name: Swathi
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, "Gayu", 1000)
acc2 = BankAccount(2, "Swathi", 2000)
bank.add_account(acc1)
bank.add_account(acc2)
bank.list_accounts()
```

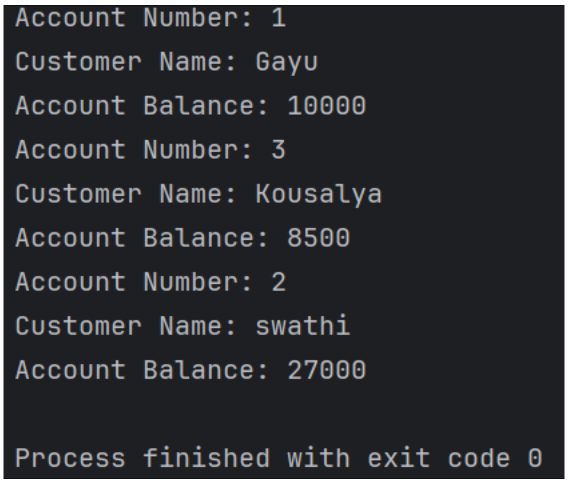


```
Using Set
Account Number: 1
Customer Name: Gayu
Account Balance: 1000
Account Number: 2
Customer Name: Swathi
Account Balance: 2000

Process finished with exit code 0
```

3. From the previous task change the HMBank attribute Accounts to HashMap 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
```

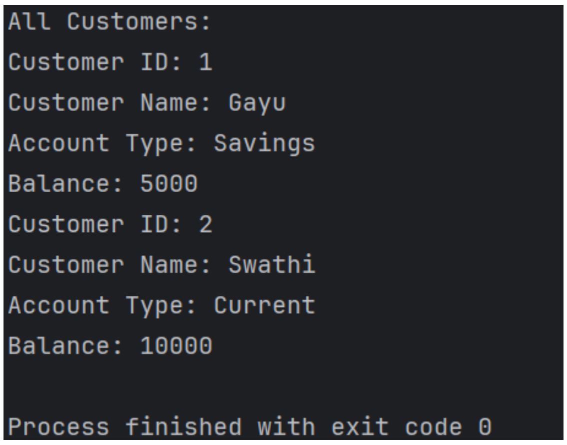
```
    def withdraw(self, amount):
        if amount <= self.balance:
            self.balance -= amount
            print("Balance After withdrawal: ", self.balance)
        else:
            print("Insufficient balance")

    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[account.account_number] = account

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


bank = Bank()
acc1 = BankAccount(1, "Gayu", 10000)
acc2 = BankAccount(2, "swathi", 27000)
acc3 = BankAccount(3, "Kousalya", 8500)
bank.add_account(acc1)
bank.add_account(acc2)
bank.add_account(acc3)
bank.list_accounts()
```


```
Account Number: 1
Customer Name: Gayu
Account Balance: 10000
Account Number: 3
Customer Name: Kousalya
Account Balance: 8500
Account Number: 2
Customer Name: swathi
Account Balance: 27000

Process finished with exit code 0
```

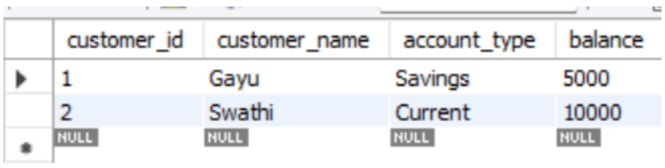
Task 14: Database Connectivity.

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

```
import mysql.connector
class Customer:
  def __init__(self, customer_id, customer_name, account_type, balance):
    self.customer_id = customer_id
    self.customer_name = customer_name
    self.account_type = account_type
    self.balance = balance
  def display(self):
    print("Customer ID:", self.customer_id)
    print("Customer Name:", self.customer_name)
    print("Account Type:", self.account_type)
    print("Balance:", self.balance)


class Database:
  def __init__(self,db_name):
    self.connection = mysql.connector.connect(
      host="localhost",user="root",password="root",port="3306",database=db_name
    )
    self.cursor = self.connection.cursor()
    self.cursor.execute('''CREATE TABLE IF NOT EXISTS customer
              (customer_id int PRIMARY KEY,
              customer_name text,
              account_type text,
                         balance int)''')
        self.connection.commit()

    def add_customer(self, customer):
        query="INSERT INTO customer(customer_id, customer_name, account_type, balance)
VALUES (%s, %s, %s, %s)"
        self.cursor.execute(query,
                         (customer.customer_id, customer.customer_name, customer.account_type,
customer.balance))
        self.connection.commit()

    def display_all_customers(self):
        self.cursor.execute('''SELECT * FROM customer''')
        rows = self.cursor.fetchall()
        for row in rows:
            cust = Customer(row[0], row[1], row[2], row[3])
            cust.display()


    def close(self):
        self.connection.close()

db = Database("customers")

 cust1 = Customer(1, "Gayu", "Savings", 5000)
db.add_customer(cust1)

cust2 = Customer(2, "Swathi", "Current", 10000)
db.add_customer(cust2)

print("All Customers:")
db.display_all_customers()

db.close()
```



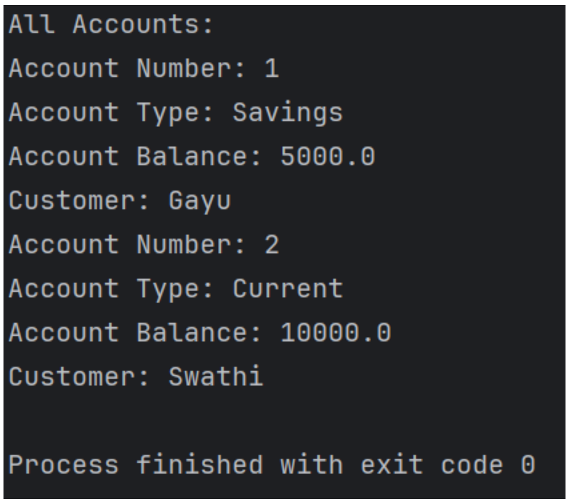
```
All Customers:
Customer ID: 1
Customer Name: Gayu
Account Type: Savings
Balance: 5000
Customer ID: 2
Customer Name: Swathi
Account Type: Current
Balance: 10000

Process finished with exit code 0
```



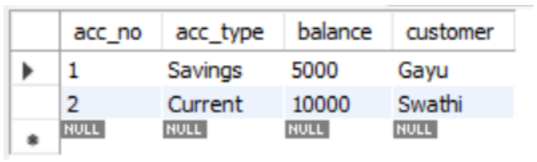
	customer_id	customer_name	account_type	balance
▶	1	Gayu	Savings	5000
	2	Swathi	Current	10000
*	NULL	NULL	NULL	NULL

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

```python
import mysql.connector


class Account:
    lastAccNo = 0

    def __init__(self, acc_type, balance, customer):
        Account.lastAccNo += 1
        self.acc_no = Account.lastAccNo
        self.acc_type = acc_type
        self.balance = balance
        self.customer = customer

    def display(self):
        print("Account Number:", self.acc_no)
        print("Account Type:", self.acc_type)
        print("Account Balance:", self.balance)
        print("Customer:", self.customer)


class Database:
    def __init__(self, db_name):
        self.connection = mysql.connector.connect(
            host="localhost",
            user="root",
            password="root",
            port="3306",
            database=db_name
            )
        self.cursor = self.connection.cursor()
        self.cursor.execute('''CREATE TABLE IF NOT EXISTS accounts
                            (acc_no INTEGER PRIMARY KEY,
                            acc_type TEXT,
                            balance REAL,
                            customer TEXT)''')
        self.connection.commit()

    def add_account(self, account):
        self.cursor.execute('''INSERT INTO accounts(acc_no, acc_type,
balance, customer)VALUES (%s,%s,%s,%s)''',(account.acc_no,
account.acc_type, account.balance, account.customer))
        self.connection.commit()

    def display_all_accounts(self):
        self.cursor.execute('''SELECT * FROM accounts''')
        rows = self.cursor.fetchall()
        for row in rows:
            acc = Account(row[1], row[2], row[3])
            acc.acc_no = row[0]
            acc.display()

    def close(self):
        self.connection.close()



db = Database("customers")


acc1 = Account("Savings", 5000, "Gayu")
db.add_account(acc1)

acc2 = Account("Current", 10000, "Swathi")
db.add_account(acc2)


print("All Accounts:")
db.display_all_accounts()

db.close()
```



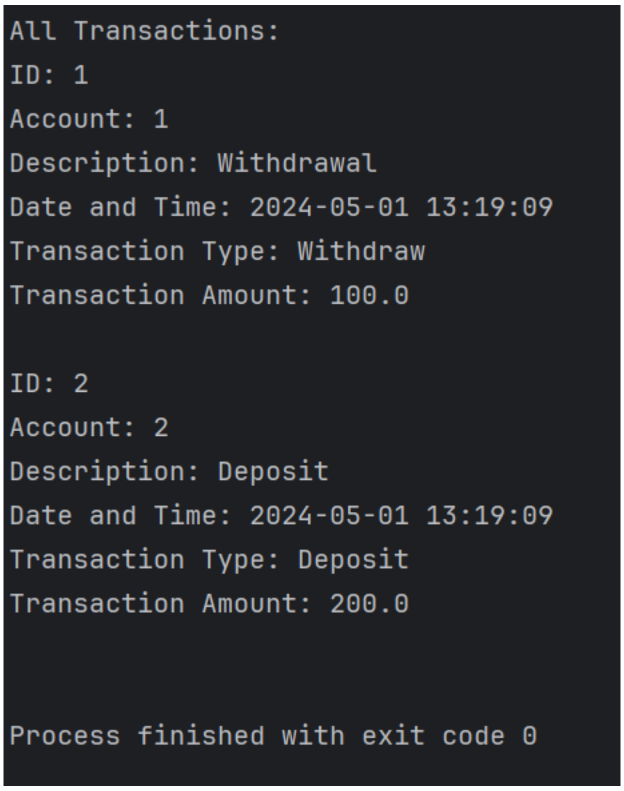
```
All Accounts:
Account Number: 1
Account Type: Savings
Account Balance: 5000.0
Customer: Gayu
Account Number: 2
Account Type: Current
Account Balance: 10000.0
Customer: Swathi

Process finished with exit code 0
```



	acc_no	acc_type	balance	customer
▶	1	Savings	5000	Gayu
	2	Current	10000	Swathi
*	NULL	NULL	NULL	NULL

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

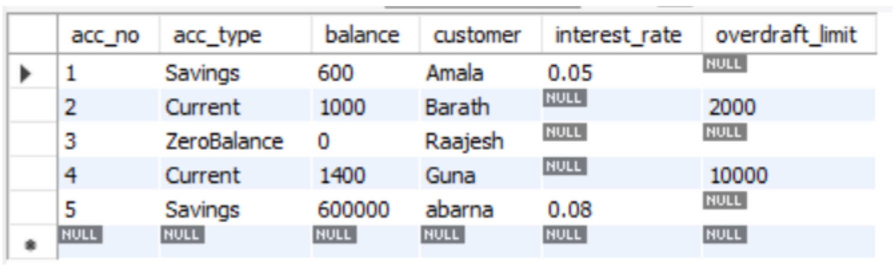
```
import mysql.connector
from datetime import datetime


class Transaction:
    def __init__(self, account, description, transaction_type,
transaction_amount):
        self.account = account
        self.description = description
        self.date_time = datetime.now().strftime("%Y-%m-%d %H:%M:%S")
        self.transaction_type = transaction_type
        self.transaction_amount = transaction_amount


class Database:
    def __init__(self, host, user, password,port, database):
        self.connection = mysql.connector.connect(
            host=host,
            user=user,
            password=password,
            port=port,
            database=database
        )
        self.cursor = self.connection.cursor()
        self.cursor.execute('''CREATE TABLE IF NOT EXISTS transactions
                            (id INT AUTO_INCREMENT PRIMARY KEY,
                            account INT,
                            description VARCHAR(255),
                            date_time DATETIME,
                            transaction_type VARCHAR(50),
                            transaction_amount FLOAT)''')
        self.connection.commit()

    def add_transaction(self, transaction):
        query = '''INSERT INTO transactions(account, description,
date_time, transaction_type, transaction_amount)
                   VALUES (%s, %s, %s, %s, %s)'''
        values = (transaction.account, transaction.description,
transaction.date_time, transaction.transaction_type,
                  transaction.transaction_amount)
        self.cursor.execute(query, values)
        self.connection.commit()

    def display_all_transactions(self):
        self.cursor.execute('''SELECT * FROM transactions''')
        rows = self.cursor.fetchall()
        for row in rows:
            print("ID:", row[0])
            print("Account:", row[1])
            print("Description:", row[2])
            print("Date and Time:", row[3])
            print("Transaction Type:", row[4])
            print("Transaction Amount:", row[5])
            print()

    def close(self):
        self.connection.close()


# Example Usage
db = Database(host="localhost", user="root", password="root",port="3306",
database="customers")

# Adding transactions
transaction1 = Transaction(account=1, description="Withdrawal",
transaction_type="Withdraw", transaction_amount=100)
db.add_transaction(transaction1)

transaction2 = Transaction(account=2, description="Deposit",
transaction_type="Deposit", transaction_amount=200)
db.add_transaction(transaction2)

# Displaying all transactions
print("All Transactions:")
db.display_all_transactions()

db.close()
```



```
All Transactions:
ID: 1
Account: 1
Description: Withdrawal
Date and Time: 2024-05-01 13:19:09
Transaction Type: Withdraw
Transaction Amount: 100.0

ID: 2
Account: 2
Description: Deposit
Date and Time: 2024-05-01 13:19:09
Transaction Type: Deposit
Transaction Amount: 200.0


Process finished with exit code 0
```

4. Create three child classes that inherit the Account class and each class must contain below mentioned attribute:
    - **SavingsAccount:** A savings account that includes an additional attribute for interest rate. Saving account should be created with minimum balance 500.
    - **CurrentAccount:** A Current account that includes an additional attribute for overdraftLimit(credit limit).
    - **ZeroBalanceAccount**: ZeroBalanceAccount can be created with Zero balance.

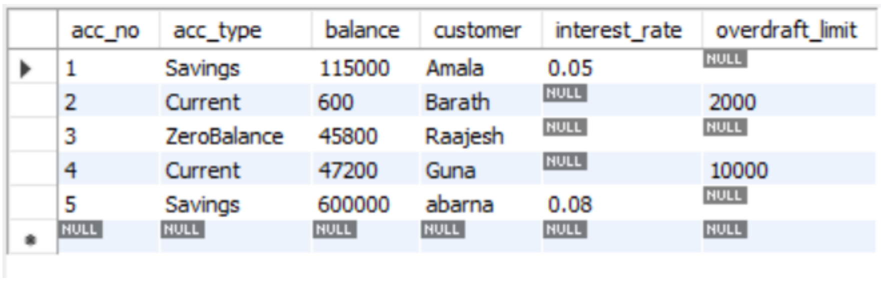
```
import mysql.connector


class Account:


    def __init__(self, acc_type, balance, customer):
        self.acc_type = acc_type
        self.balance = balance
        self.customer = customer


    def display(self):

        print("Account Number:", self.acc_no)
        print("Account Type:", self.acc_type)
        print("Account Balance:", self.balance)
        print("Customer:", self.customer)


class SavingsAccount(Account):
    def __init__(self, balance, customer, interest_rate):
        super().__init__("Savings", balance, customer)
        self.interest_rate = interest_rate
        if balance < 500:
            raise ValueError("Minimum balance for a savings account is 
500")


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


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


class Database:
    def __init__(self, db_name):
        self.connection = mysql.connector.connect(
            host="localhost",
            user="root",
            password="root",
            port="3306",
            database=db_name)
        self.cursor = self.connection.cursor()
        self.cursor.execute('''CREATE TABLE IF NOT EXISTS accounts
                            (acc_no INTEGER PRIMARY KEY AUTO_INCREMENT,
                            acc_type TEXT,
                            balance REAL,
                            customer TEXT,
                            interest_rate REAL,
                            overdraft_limit REAL)''')
        self.connection.commit()

    def add_account(self, account):
        if isinstance(account, SavingsAccount):
            self.cursor.execute('''INSERT INTO accounts( acc_type, balance, 
customer,interest_rate)
                                VALUES ( %s, %s, %s,%s)''',
                                ( account.acc_type, account.balance, 
account.customer,account.interest_rate))
        elif isinstance(account, CurrentAccount):
            self.cursor.execute('''INSERT INTO accounts( acc_type, balance, 
customer,overdraft_limit)
                                VALUES ( %s, %s, %s,%s)''',
                                ( account.acc_type, account.balance, 
account.customer,account.overdraft_limit))
        else:
            self.cursor.execute('''INSERT INTO accounts( acc_type, balance, 
customer)
                                VALUES ( %s, %s, %s)''',
                                ( account.acc_type, account.balance, 
account.customer))
        self.connection.commit()

    def display_all_accounts(self):
        self.cursor.execute('''SELECT * FROM accounts''')
        rows = self.cursor.fetchall()
        for row in rows:
            print(row)
            print(row[1])
            if row[1] == 'Savings':
                acc = SavingsAccount(row[2], row[3], row[4])
            elif row[1] == 'Current':
                acc = CurrentAccount(row[2], row[3], row[5])
            else:
                acc = ZeroBalanceAccount(row[3])
            acc.acc_no = row[0]
            acc.display()

    def close(self):
        self.connection.close()



db = Database("customers")

# Adding accounts
savings_acc = SavingsAccount(balance=600, customer="Amala",
interest_rate=0.05)
db.add_account(savings_acc)

current_acc = CurrentAccount(balance=1000, customer="Barath",
overdraft_limit=2000)
db.add_account(current_acc)

zero_balance_acc = ZeroBalanceAccount(customer="Raajesh")
db.add_account(zero_balance_acc)

current_acc = CurrentAccount(balance=1400, customer="Guna",
overdraft_limit=10000)
db.add_account(current_acc)

savings_acc = SavingsAccount(balance=600000, customer="abarna",
interest_rate=0.08)
db.add_account(savings_acc)


print("All Accounts:")
db.display_all_accounts()

db.close()
```



acc_no	acc_type	balance	customer	interest_rate	overdraft_limit
1	Savings	600	Amala	0.05	NULL
2	Current	1000	Barath	NULL	2000
3	ZeroBalance	0	Raajesh	NULL	NULL
4	Current	1400	Guna	NULL	10000
5	Savings	600000	abarna	0.08	NULL
NULL	NULL	NULL	NULL	NULL	NULL

5. Create **ICustomerServiceProvider** interface/abstract class with following functions:
    - **get_account_balance(account_number: long)**: Retrieve the balance of an account given its account number. should return the current balance of account.
    - **deposit(account_number: long, amount: float)**: Deposit the specified amount into the account. Should return the current balance of account.
    - **withdraw(account_number: long, amount: float)**: Withdraw the specified amount from the account. Should return the current balance of account.
        - A savings account should maintain a minimum balance and checking if the withdrawal violates the minimum balance rule.
        - Current account customers are allowed withdraw overdraftLimit and available account balance. withdraw limit can exceed the available balance and should not exceed the overdraft limit.
    - **transfer(from_account_number: long, to_account_number: int, amount: float)**: Transfer money from one account to another. both account number should be validate from the database use getAccountDetails method.
    - **getAccountDetails(account_number: long):** Should return the account and customer details.
    - **getTransations(account_number: long, FromDate:Date, ToDate: Date):** Should return the list of transaction between two dates.

```
import mysql.connector
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: int, to_account_number,
amount):
        pass

    @abstractmethod
    def get_account_details(self, account_number):
        pass


class CustomerServiceProvider(ICustomerServiceProvider):
    def __init__(self, host, user, password, port, database):
        self.connection = mysql.connector.connect(
            host=host,
            user=user,
            password=password,
            port=port,
            database=database
        )
        self.cursor = self.connection.cursor()

    def display_all_accounts(self):
        self.cursor.execute('''SELECT * FROM accounts''')
        all_accounts = self.cursor.fetchall()
        if all_accounts:
            print("All Accounts Details:")
            for account in all_accounts:
                column_names = [i[0] for i in self.cursor.description]
                account_details = dict(zip(column_names, account))
                print(account_details)
        else:
            print("No accounts found in the database.")

    def get_account_balance(self, account_number):
        self.cursor.execute("SELECT balance FROM accounts WHERE acc_no =
%s", (account_number,))
        balance = self.cursor.fetchone()
        if balance:
            return balance[0]
            print(f"The balance of {account_number} is {balance}")
        else:
            raise ValueError(f"Account with account number {account_number}
not found.")

    def deposit(self, account_number, amount):
        current_balance = self.get_account_balance(account_number)
        new_balance = current_balance + amount
        self.cursor.execute('''UPDATE accounts SET balance = %s WHERE
acc_no = %s''', (new_balance, account_number))
        self.connection.commit()

    def withdraw(self, account_number, amount):
        current_balance = self.get_account_balance(account_number)
        self.cursor.execute('''SELECT acc_type, overdraft_limit FROM
accounts WHERE acc_no = %s''', (account_number,))
        account_info = self.cursor.fetchone()
        if account_info:
            acc_type, overdraft_limit = account_info
            if acc_type == 'Savings':
                if current_balance - amount < 500:
                    raise ValueError("Withdrawal violates minimum balance
rule.")
            elif acc_type == 'Current':
                available_balance = current_balance + overdraft_limit
                if amount > available_balance:
                    raise ValueError("Withdrawal exceeds available balance
and overdraft limit.")
        else:
            raise ValueError(f"Account with account number {account_number}
not found.")

        new_balance = current_balance - amount
        self.cursor.execute('''UPDATE accounts SET balance = %s WHERE
acc_no = %s''', (new_balance, account_number))
        self.connection.commit()

    def transfer(self, from_account_number, to_account_number, amount):
        self.withdraw(from_account_number, amount)
        self.deposit(to_account_number, amount)

    def get_account_details(self, account_number):
        self.cursor.execute('''SELECT * FROM accounts WHERE acc_no = %s''',
(account_number,))
        account_details = self.cursor.fetchone()
        if account_details:
            column_names = [i[0] for i in self.cursor.description]
            return dict(zip(column_names, account_details))
        else:
            raise ValueError(f"Account with account number {account_number}
not found.")

    def close_connection(self):
        self.connection.close()


db = CustomerServiceProvider(host="localhost", user="root",
password="root", port="3306",
                             database="customers")
db.get_account_balance(2)
db.deposit(4, 23000)
db.withdraw(4, 200)
db.get_account_details(4)
db.transfer(2, 4, 200)
db.display_all_accounts()
db.close_connection()
```



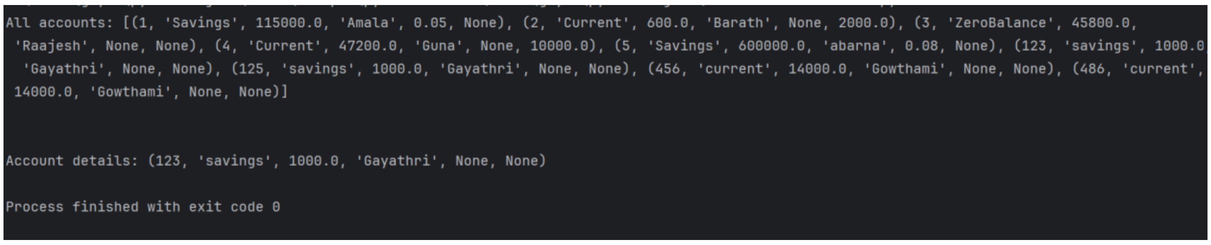
acc_no	acc_type	balance	customer	interest_rate	overdraft_limit
1	Savings	115000	Amala	0.05	NULL
2	Current	600	Barath	NULL	2000
3	ZeroBalance	45800	Raajesh	NULL	NULL
4	Current	47200	Guna	NULL	10000
5	Savings	600000	abarna	0.08	NULL
NULL	NULL	NULL	NULL	NULL	NULL

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

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

```
import mysql.connector
from abc import ABC, abstractmethod


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

    @abstractmethod
    def list_accounts(self):
        pass

    @abstractmethod
    def get_account_details(self, account_number):
        pass



class MySQLBankServiceProvider(IBankServiceProvider):
    def __init__(self, host, user, password, port, database):
        self.connection = mysql.connector.connect(
            host=host,
            user=user,
            password=password,
            port=port,
            database=database
        )
        self.cursor = self.connection.cursor()

    def create_account(self, customer,acc_no,acc_type, balance):
        query = "INSERT INTO accounts (customer,acc_no, acc_type, balance)
VALUES (%s, %s, %s,%s)"
        values = (customer, acc_no, acc_type, balance)
        self.cursor.execute(query, values)
        self.connection.commit()

    def list_accounts(self):
        self.cursor.execute("SELECT * FROM accounts")
        accounts = self.cursor.fetchall()
        return accounts

    def get_account_details(self, account_number):
        query = "SELECT * FROM accounts WHERE acc_no = %s"
        self.cursor.execute(query, (account_number,))
        account_details = self.cursor.fetchone()
        return account_details


    def close_connection(self):
        self.connection.close()



db = MySQLBankServiceProvider(host="localhost", user="root",
password="root", port="3306", database="customers")
```

```
# Create a new account
db.create_account("Gayathri",  125, "savings", 1000.0)
db.create_account("Gowthami",  486, "current", 14000.0)
# List all accounts
accounts = db.list_accounts()
print("All accounts:", accounts)
print()
# Get account details
printing = db.get_account_details(123)
print()
print("Account details:", printing)

db.close_connection()
```



```
All accounts: [(1, 'Savings', 115000.0, 'Amala', 0.05, None), (2, 'Current', 600.0, 'Barath', None, 2000.0), (3, 'ZeroBalance', 45800.0,
 'Raajesh', None, None), (4, 'Current', 47200.0, 'Guna', None, 10000.0), (5, 'Savings', 600000.0, 'abarna', 0.08, None), (123, 'savings', 1000.0
  'Gayathri', None, None), (125, 'savings', 1000.0, 'Gayathri', None, None), (456, 'current', 14000.0, 'Gowthami', None, None), (486, 'current',
 14000.0, 'Gowthami', None, None)]


Account details: (123, 'savings', 1000.0, 'Gayathri', None, None)

Process finished with exit code 0
```

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

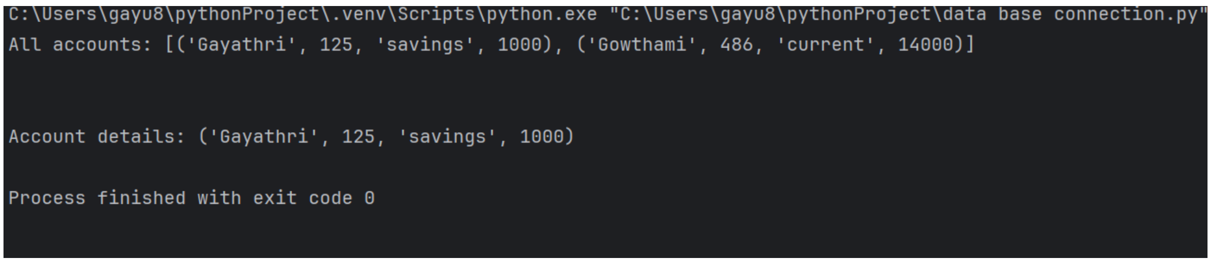
MAIN FILE:

```
8. from sql_query_connection import Queryconnection
from abc import ABC, abstractmethod


class ICustomerServiceProvider(ABC):
    @abstractmethod
    def create_account(self, customer, acc_num, acc_type, balance):
        pass

    @abstractmethod
    def list_accounts(self):
        pass

    @abstractmethod
    def get_account_details(self, account_number):
        pass



class CustomerServiceProvider(ICustomerServiceProvider):
    db = Queryconnection(host="localhost", user="root",
password="root", port="3306", database="customers")
    # Create a new account
    db.create_account("Gayathri",  125, "savings", 1000.0)
    db.create_account("Gowthami",  486, "current", 14000.0)
    # List all accounts
    accounts = db.list_accounts()
    print("All accounts:", accounts)
    print()
    # Get account details
    printing = db.get_account_details(125)
```

```
9.
    print()
    print("Account details:", printing)

    db.close_connection()
```

sql_quer_connection.py:

```
import mysql.connector
class Queryconnection:
    def __init__(self, host, user, password, port, database):
        self.connection = mysql.connector.connect(
            host=host,
            user=user,
            password=password,
            port=port,
            database=database
        )
        self.cursor = self.connection.cursor()

    def create_account(self, customer, acc_num, acc_type, balance):
        query = "INSERT INTO customerserviceprovider (customer,acc_num,
acc_type, balance) VALUES (%s, %s, %s,%s)"
        values = (customer, acc_num, acc_type, balance)
        self.cursor.execute(query, values)
        self.connection.commit()

    def list_accounts(self):
        self.cursor.execute("SELECT * FROM customerserviceprovider")
        accounts = self.cursor.fetchall()
        return accounts

    def get_account_details(self, account_number):
        query = "SELECT * FROM customerserviceprovider WHERE acc_num = %s"
        self.cursor.execute(query, (account_number,))
        account_details = self.cursor.fetchone()
        return account_details

    def close_connection(self):
        self.connection.close()
```



```
C:\Users\gayu8\pythonProject\.venv\Scripts\python.exe "C:\Users\gayu8\pythonProject\data base connection.py"
All accounts: [('Gayathri', 125, 'savings', 1000), ('Gowthami', 486, 'current', 14000)]


Account details: ('Gayathri', 125, 'savings', 1000)

Process finished with exit code 0
```

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

- Attributes ○ accountList: List of **Accounts** to store any account objects. ○ transactionList: List of **Transaction** to store transaction objects. ○ branchName and branchAddress as String objects

```
from typing import List
import CustomerServiceProviderImpl
import IBankServiceProvider


class BankServiceProviderImpl(CustomerServiceProviderImpl, IBankServiceProvider):
    def __init__(self, branch_name: str, branch_address: str):
        super().__init__()
        self.account_list: List[Account] = []
        self.transaction_list: List[Transaction] = []
        self.branch_name = branch_name
        self.branch_address = branch_address

    def create_account(self, customer: Customer, acc_no: int, acc_type: str, balance: float) ->
None:
        account = Account(acc_no, acc_type, balance, customer)
        self.account_list.append(account)

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

    def get_account_details(self, account_number: int) -> Account:
        for account in self.account_list:
            if account.acc_no == account_number:
                return account
        return None

    # Additional methods
    def add_transaction(self, transaction: Transaction) -> None:
        self.transaction_list.append(transaction)

    def list_transactions(self) -> List[Transaction]:
        return self.transaction_list
```

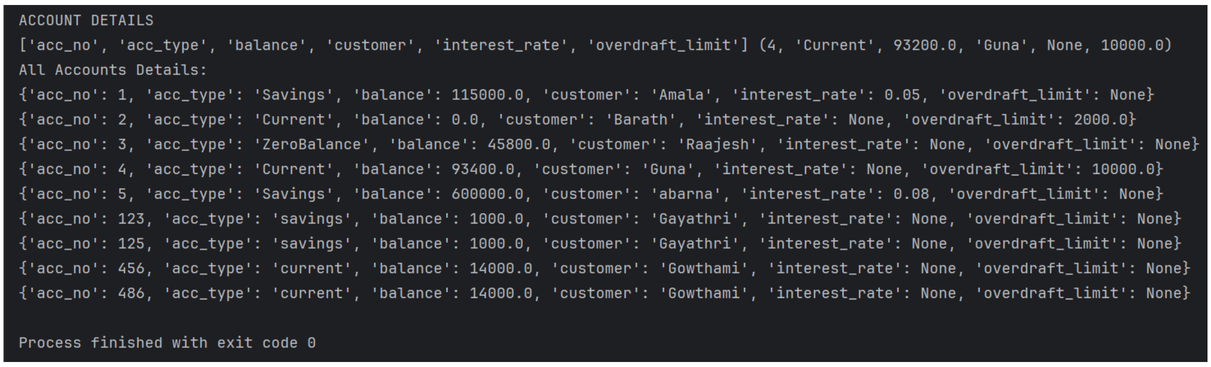
9.Create I**BankRepository** interface/abstract class which include following methods to interact with database.

- **createAccount(customer: Customer, accNo: long, accType: String, balance: float)**: Create a new bank account for the given customer with the initial balance and store in database.
- **listAccounts()**: List<Account> accountsList: List all accounts in the bank from database.
- **calculateInterest():** the calculate_interest() method to calculate interest based on the balance and interest rate.
- **getAccountBalance(account_number: long)**: Retrieve the balance of an account given its account number. should return the current balance of account from database.
- **deposit(account_number: long, amount: float)**: Deposit the specified amount into the account. Should update new balance in database and return the new balance.
- **withdraw(account_number: long, amount: float)**: Withdraw amount should check the balance from account in database and new balance should updated in Database.
    - A savings account should maintain a minimum balance and checking if the withdrawal violates the minimum balance rule.
    - Current account customers are allowed withdraw overdraftLimit and available account balance. withdraw limit can exceed the available balance and should not exceed the overdraft limit.
- **transfer(from_account_number: long, to_account_number: int, amount: float)**: Transfer money from one account to another. check the balance from account in database and new balance should updated in Database.
- **getAccountDetails(account_number: long):** Should return the account and customer details from databse.
- **getTransations(account_number: long, FromDate:Date, ToDate: Date):** Should return the list of transaction between two dates from database.

10.
```
import mysql.connector
from abc import ABC, abstractmethod


class IBankRepository(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: int, to_account_number,
amount):
        pass
    @abstractmethod
    def get_account_details(self, account_number):
        pass
```

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

```
class IBankRepositoryImpl(IBankRepository):
    def __init__(self, host, user, password, port, database):
        self.connection = mysql.connector.connect(
            host=host,
            user=user,
            password=password,
            port=port,
            database=database
        )
        self.cursor = self.connection.cursor()

    def display_all_accounts(self):
        self.cursor.execute('''SELECT * FROM accounts''')
        all_accounts = self.cursor.fetchall()
        if all_accounts:
            print("All Accounts Details:")
            for account in all_accounts:
                column_names = [i[0] for i in
self.cursor.description]
                account_details = dict(zip(column_names, account))
                print(account_details)
        else:
            print("No accounts found in the database.")

    def get_account_balance(self, account_number):
        self.cursor.execute("SELECT balance FROM accounts WHERE
acc_no = %s", (account_number,))
        balance = self.cursor.fetchone()
        if balance:
            return balance[0]
            print(f"The balance of {account_number} is {balance}")
        else:
            raise ValueError(f"Account with account number
{account_number} not found.")

    def deposit(self, account_number, amount):
        current_balance = self.get_account_balance(account_number)
        new_balance = current_balance + amount
        self.cursor.execute('''UPDATE accounts SET balance = %s WHERE
acc_no = %s''', (new_balance, account_number))
        self.connection.commit()

    def withdraw(self, account_number, amount):
        current_balance = self.get_account_balance(account_number)
        self.cursor.execute('''SELECT acc_type, overdraft_limit FROM
accounts WHERE acc_no = %s''', (account_number,))
        account_info = self.cursor.fetchone()
        if account_info:
            acc_type, overdraft_limit = account_info
            if acc_type == 'Savings':
                if current_balance - amount < 500:
                    raise ValueError("Withdrawal violates minimum
balance rule.")
            elif acc_type == 'Current':
                available_balance = current_balance + overdraft_limit
                if amount > available_balance:
                    raise ValueError("Withdrawal exceeds available
balance and overdraft limit.")
        else:
            raise ValueError(f"Account with account number
{account_number} not found.")

        new_balance = current_balance - amount
        self.cursor.execute('''UPDATE accounts SET balance = %s WHERE
acc_no = %s''', (new_balance, account_number))
        self.connection.commit()

    def transfer(self, from_account_number, to_account_number,
amount):
        self.withdraw(from_account_number, amount)
        self.deposit(to_account_number, amount)

    def get_account_details(self, account_number):
        self.cursor.execute('''SELECT * FROM accounts WHERE acc_no =
%s''', (account_number,))
        account_details = self.cursor.fetchone()
        if account_details:
            column_names = [i[0] for i in self.cursor.description]
            print("ACCOUNT DETAILS")
            print(column_names, account_details)
        else:
            raise ValueError(f"Account with account number
{account_number} not found.")

    def close_connection(self):
        self.connection.close()


db = IBankRepositoryImpl(host="localhost", user="root",
password="root", port="3306",
                             database="customers")
db.get_account_balance(2)
db.get_account_details(4)
db.transfer(2, 4, 200)
db.display_all_accounts()
db.close_connection()
```

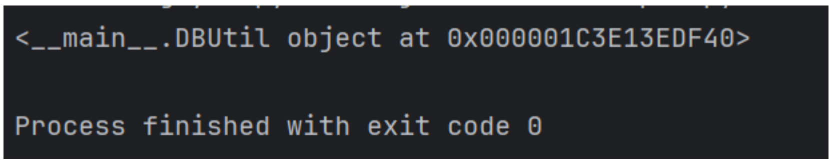


```
ACCOUNT DETAILS
['acc_no', 'acc_type', 'balance', 'customer', 'interest_rate', 'overdraft_limit'] (4, 'Current', 93200.0, 'Guna', None, 10000.0)
All Accounts Details:
{'acc_no': 1, 'acc_type': 'Savings', 'balance': 115000.0, 'customer': 'Amala', 'interest_rate': 0.05, 'overdraft_limit': None}
{'acc_no': 2, 'acc_type': 'Current', 'balance': 0.0, 'customer': 'Barath', 'interest_rate': None, 'overdraft_limit': 2000.0}
{'acc_no': 3, 'acc_type': 'ZeroBalance', 'balance': 45800.0, 'customer': 'Raajesh', 'interest_rate': None, 'overdraft_limit': None}
{'acc_no': 4, 'acc_type': 'Current', 'balance': 93400.0, 'customer': 'Guna', 'interest_rate': None, 'overdraft_limit': 10000.0}
{'acc_no': 5, 'acc_type': 'Savings', 'balance': 600000.0, 'customer': 'abarna', 'interest_rate': 0.08, 'overdraft_limit': None}
{'acc_no': 123, 'acc_type': 'savings', 'balance': 1000.0, 'customer': 'Gayathri', 'interest_rate': None, 'overdraft_limit': None}
{'acc_no': 125, 'acc_type': 'savings', 'balance': 1000.0, 'customer': 'Gayathri', 'interest_rate': None, 'overdraft_limit': None}
{'acc_no': 456, 'acc_type': 'current', 'balance': 14000.0, 'customer': 'Gowthami', 'interest_rate': None, 'overdraft_limit': None}
{'acc_no': 486, 'acc_type': 'current', 'balance': 14000.0, 'customer': 'Gowthami', 'interest_rate': None, 'overdraft_limit': None}

Process finished with exit code 0
```

11.Create **DBUtil** class and add the following method.

- **static getDBConn():Connection** Establish a connection to the database and return Connection reference

```
2.import mysql.connector
class DBUtil:
    def getDBConn(self):
        con=mysql.connector.connect(
            host="localhost",
            user="root",
            password="root",
            port="3306",
            database="customers"
        )
        con.cursor()

obj=DBUtil()
print(obj)
```



```
<__main__.DBUtil object at 0x000001C3E13EDF40>

Process finished with exit code 0
```

12.Create **BankApp** class and perform following operation:

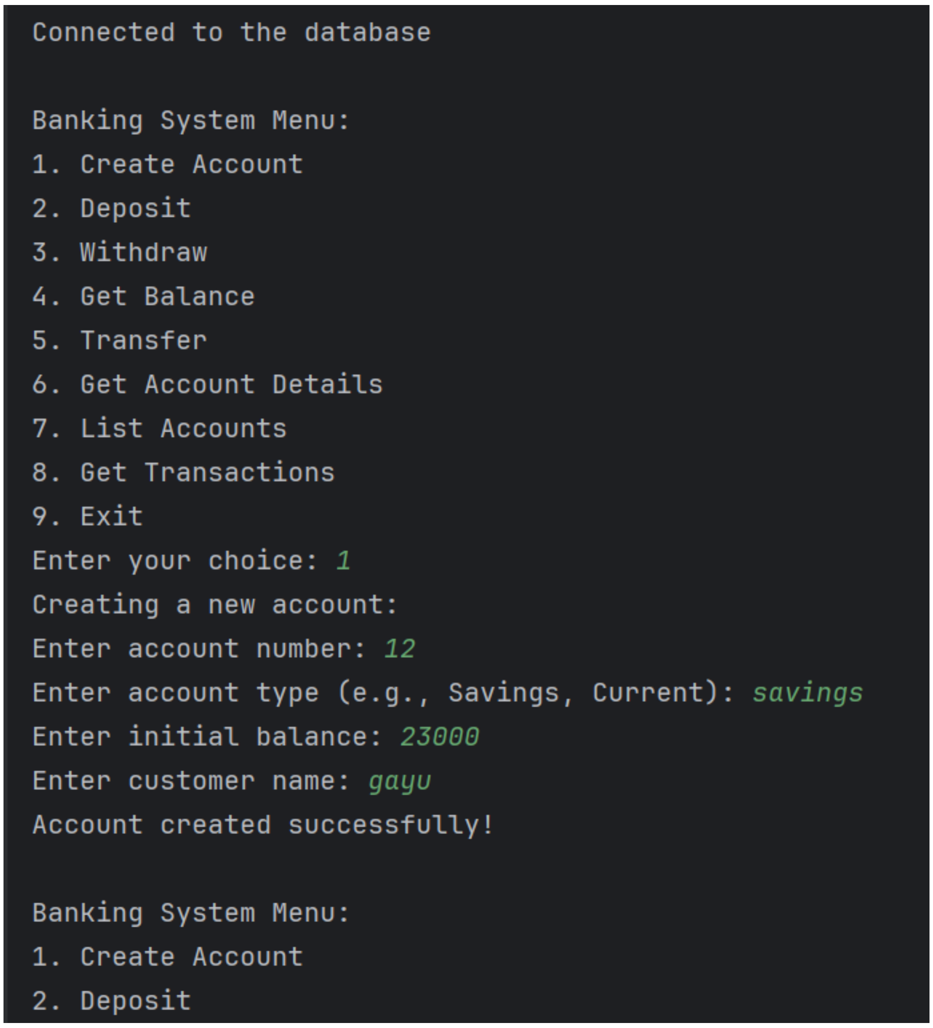
main method to simulate the banking system. Allow the user to interact with the system by entering choice from menu such as "create_account", "deposit", "withdraw", "get_balance", "transfer", "getAccountDetails", "ListAccounts", "getTransactions" and "exit."

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

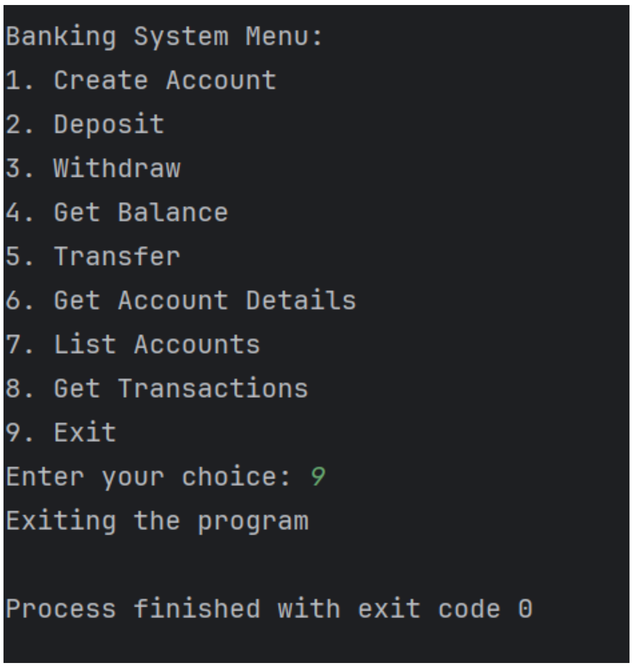
```
import mysql.connector
from mysql.connector import Error
from datetime import datetime

class BankApp:
    def __init__(self):
        self.connection = self.connect_to_database()

    def connect_to_database(self):
        try:
            connection = mysql.connector.connect(
                host="localhost",
                user="root",
                password="root",
                database="customers"
            )
            if connection.is_connected():
                print("Connected to the database")
                return connection
        except Error as e:
            print("Error while connecting to MySQL", e)

    def create_account(self):
        print("Creating a new account:")
        acc_num = input("Enter account number: ")
        acc_type = input("Enter account type (e.g., Savings, Current): ")
        balance = float(input("Enter initial balance: "))
        customer = input("Enter customer name: ")

        cursor = self.connection.cursor()
        try:
            query = "INSERT INTO accounts (acc_no, acc_type,
balance,customer) VALUES (%s, %s, %s,%s)"
            values = (acc_num, acc_type,balance,customer)
            cursor.execute(query, values)
            self.connection.commit()
            print("Account created successfully!")
        except Error as e:
            self.connection.rollback()
            print("Error while creating account:", e)

    def deposit(self):
        print("Depositing money:")
        account_number = input("Enter account number: ")
        amount = float(input("Enter amount to deposit: "))

        cursor = self.connection.cursor()
        try:
            query = "UPDATE accounts SET balance = balance + %s WHERE
acc_no = %s"
            values = (amount, account_number)
            cursor.execute(query, values)
            self.connection.commit()
            print("Deposit successful!")
        except Error as e:
            self.connection.rollback()
            print("Error while depositing money:", e)

    def withdraw(self):
        print("Withdrawing money:")
        account_number = input("Enter account number: ")
        amount = float(input("Enter amount to withdraw: "))

        cursor = self.connection.cursor()
        try:
            query = "UPDATE accounts SET balance = balance - %s WHERE
```

```
acc_no = %s AND balance >= %s"
            values = (amount, account_number, amount)
            cursor.execute(query, values)
            if cursor.rowcount > 0:
                self.connection.commit()
                print("Withdrawal successful!")
            else:
                print("Insufficient funds for withdrawal.")
        except Error as e:
            self.connection.rollback()
            print("Error while withdrawing money:", e)

    def get_balance(self):
        print("Getting account balance:")
        account_number = input("Enter account number: ")

        cursor = self.connection.cursor()
        try:
            query = "SELECT balance FROM accounts WHERE acc_no = %s"
            cursor.execute(query, (account_number,))
            result = cursor.fetchone()
            if result:
                print("Account balance:", result[0])
            else:
                print("Account not found.")
        except Error as e:
            print("Error while getting account balance:", e)

    def transfer(self):
        print("Transferring money:")
        from_account_number = input("Enter sender's account number: ")
        to_account_number = input("Enter receiver's account number: ")
        amount = float(input("Enter amount to transfer: "))

        cursor = self.connection.cursor()
        try:
            query = "UPDATE accounts SET balance = balance - %s WHERE
acc_no = %s AND balance >= %s"
            values = (amount, from_account_number, amount)
            cursor.execute(query, values)
            if cursor.rowcount > 0:
                query = "UPDATE accounts SET balance = balance + %s WHERE
acc_no = %s"
                values = (amount, to_account_number)
                cursor.execute(query, values)
                self.connection.commit()
                print("Transfer successful!")
            else:
                print("Insufficient funds for transfer.")
        except Error as e:
            self.connection.rollback()
            print("Error while transferring money:", e)

    def get_account_details(self):
        print("Getting account details:")
        account_number = input("Enter account number: ")

        cursor = self.connection.cursor()
        try:
            query = "SELECT * FROM accounts WHERE acc_no = %s"
            cursor.execute(query, (account_number,))
```

```python
            result = cursor.fetchone()
            if result:
                print("Account details:")
                print("Account Number:", result[0])
                print("Customer Name:", result[1])
                print("Account Type:", result[2])
                print("Balance:", result[3])
            else:
                print("Account not found.")
        except Error as e:
            print("Error while getting account details:", e)

    def list_accounts(self):
        print("Listing accounts:")

        cursor = self.connection.cursor()
        try:
            query = "SELECT * FROM accounts"
            cursor.execute(query)
            results = cursor.fetchall()
            if results:
                print("Accounts:")
                for result in results:
                    print("Account Number:", result[0])
                    print("Customer Name:", result[1])
                    print("Account Type:", result[2])
                    print("Balance:", result[3])
                    print("-----------------------------")
            else:
                print("No accounts found.")
        except Error as e:
            print("Error while listing accounts:", e)

    def get_transactions(self):
        print("Getting transactions:")
        # Placeholder for transaction retrieval from database

    def display_menu(self):
        print("\nBanking System Menu:")
        print("1. Create Account")
        print("2. Deposit")
        print("3. Withdraw")
        print("4. Get Balance")
        print("5. Transfer")
        print("6. Get Account Details")
        print("7. List Accounts")
        print("8. Get Transactions")
        print("9. Exit")

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

            if choice == "1":
                self.create_account()
            elif choice == "2":
                self.deposit()
            elif choice == "3":
                self.withdraw()
            elif choice == "4":
                self.get_balance()
            elif choice == "5":
                self.transfer()
            elif choice == "6":
                self.get_account_details()
            elif choice == "7":
                self.list_accounts()
            elif choice == "8":
                self.get_transactions()
            elif choice == "9":
                print("Exiting the program")
                if self.connection.is_connected():
                    self.connection.close()
                break
            else:
                print("Invalid choice. Please try again.")


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



```
Connected to the database

Banking System Menu:
1. Create Account
2. Deposit
3. Withdraw
4. Get Balance
5. Transfer
6. Get Account Details
7. List Accounts
8. Get Transactions
9. Exit
Enter your choice: 1
Creating a new account:
Enter account number: 12
Enter account type (e.g., Savings, Current): savings
Enter initial balance: 23000
Enter customer name: gayu
Account created successfully!

Banking System Menu:
1. Create Account
2. Deposit
```



```
Banking System Menu:
1. Create Account
2. Deposit
3. Withdraw
4. Get Balance
5. Transfer
6. Get Account Details
7. List Accounts
8. Get Transactions
9. Exit
Enter your choice: 9
Exiting the program

Process finished with exit code 0
```