Assignment:-3

By Shreyasi Reja

Banking System

Control Structure

Task 1: Conditional Statements

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

Credit Score must be above 700.

Annual Income must be at least $50,000.

Tasks:

1. Write a program that takes the customer's credit score and annual income as input.

2. Use conditional statements (if-else) to determine if the customer is eligible for a loan.

3. Display an appropriate message based on eligibility.

```python
def check_loan_eligibility(credit_score, annual_income):
    credit_score_threshold = 700
    annual_income_threshold = 50000

    if credit_score > credit_score_threshold and annual_income >= annual_income_threshold:
        return "Congratulations! You are eligible for a loan."
    else:
        return "Sorry, you are not eligible for a loan."

customer_credit_score = int(input("Enter your credit score: "))
customer_annual_income = float(input("Enter your annual income: $"))

result = check_loan_eligibility(customer_credit_score, customer_annual_income)
print(result)
```

Output:-

```python
def check_loan_eligibility(credit_score, annual_income):
    credit_score_threshold = 700
    annual_income_threshold = 50000

    if credit_score > credit_score_threshold and annual_income >= annual_income_threshold:
        return "Congratulations! You are eligible for a loan."
    else:
        return "Sorry, you are not eligible for a loan."

customer_credit_score = int(input("Enter your credit score: "))
customer_annual_income = float(input("Enter your annual income: $"))
```

```
"C:\Users\SHREYASI\OneDrive\Desktop\Hexaware technical traning code\pythonProject\.venv\Scripts\python.exe" "C:\Users\SHREYASI\OneDrive\Desktop\Hexaware t
Enter your credit score: 100
Enter your annual income: $300
Sorry, you are not eligible for a loan.

Process finished with exit code 0
```

Task 2: Nested Conditional Statements

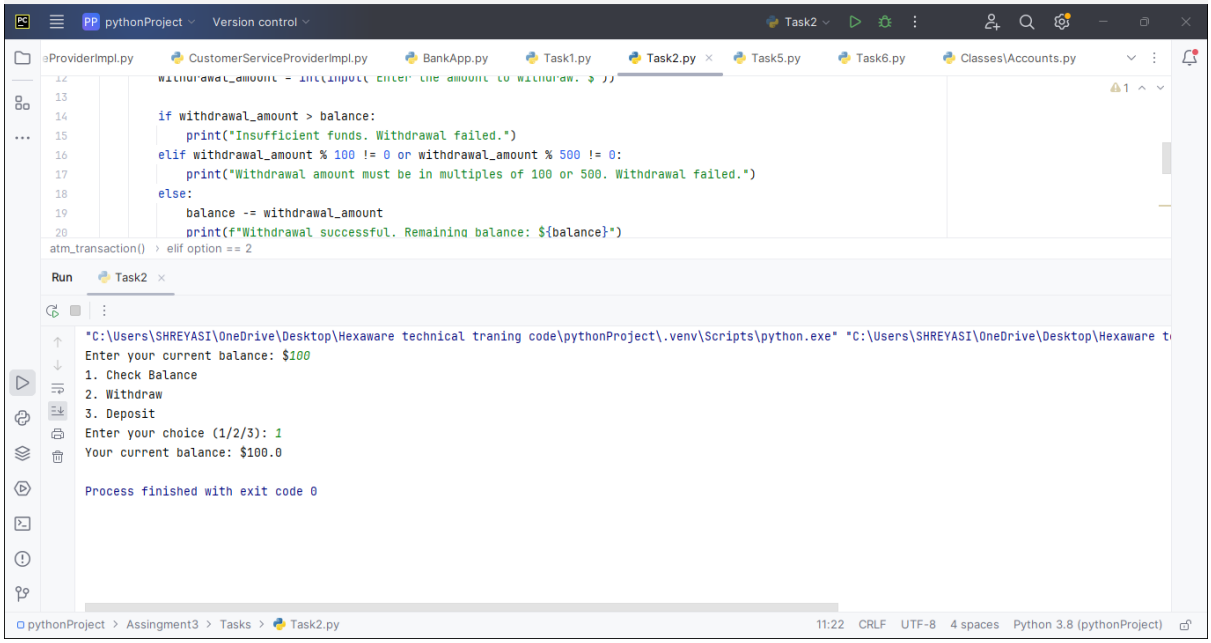
Create a program that simulates an ATM transaction. Display options such as "Check Balance," "Withdraw," "Deposit,". Ask the user to enter their current balance and the amount they want to withdraw or deposit. Implement checks to ensure that the withdrawal amount is not greater than the available balance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

amount is in multiples of 100 or 500. Display appropriate messages for success or failure.

```python
def atm_transaction(balance):
    print("1. Check Balance")
    print("2. Withdraw")
    print("3. Deposit")

    option = int(input("Enter your choice (1/2/3): "))

    if option == 1:
        print(f"Your current balance: ${balance}")

    elif option == 2:
        withdrawal_amount = int(input("Enter the amount to withdraw: $"))

        if withdrawal_amount > balance:
            print("Insufficient funds. Withdrawal failed.")
        elif withdrawal_amount % 100 != 0 or withdrawal_amount % 500 != 0:
            print("Withdrawal amount must be in multiples of 100 or 500. Withdrawal failed.")
        else:
            balance -= withdrawal_amount
            print(f"Withdrawal successful. Remaining balance: ${balance}")

    elif option == 3:
        deposit_amount = int(input("Enter the amount to deposit: $"))
        balance += deposit_amount
```

```python
        if withdrawal_amount > balance:
            print("Insufficient funds. Withdrawal failed.")
        elif withdrawal_amount % 100 != 0 or withdrawal_amount % 500 != 0:
            print("Withdrawal amount must be in multiples of 100 or 500. Withdrawal failed.")
        else:
            balance -= withdrawal_amount
            print(f"Withdrawal successful. Remaining balance: ${balance}")

    elif option == 3:
        deposit_amount = int(input("Enter the amount to deposit: $"))
        balance += deposit_amount
        print(f"Deposit successful. Updated balance: ${balance}")

    else:
        print("Invalid option. Please choose a valid option (1/2/3).")

initial_balance = float(input("Enter your current balance: $"))

atm_transaction(initial_balance)
```

Output



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.

```python
def calculate_future_balance(initial_balance, annual_interest_rate, years):
    future_balance = initial_balance * (1 + annual_interest_rate/100) ** years
    return future_balance

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

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

    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 = calculate_future_balance(initial_balance, annual_interest_rate, years)
    print(f"Future balance after {years} years: ${future_balance:.2f}")
```

Output

```
"C:\Users\SHREYASI\OneDrive\Desktop\Hexaware technical traning code\pythonProject\.venv\Scripts\python.exe" "C:\Users\SHREYASI\OneDrive\Desktop\Hexaware t
Enter the number of customers: 3

Customer 1:
Enter the initial balance: $100
Enter the annual interest rate (%): 4
Enter the number of years: 3
Future balance after 3 years: $112.49

Customer 2:
Enter the initial balance: $700
Enter the annual interest rate (%): 6
Enter the number of years: 8
Future balance after 8 years: $1115.69

Customer 3:
Enter the initial balance: $500
Enter the annual interest rate (%): 8
Enter the number of years: 5
Future balance after 5 years: $734.66

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.

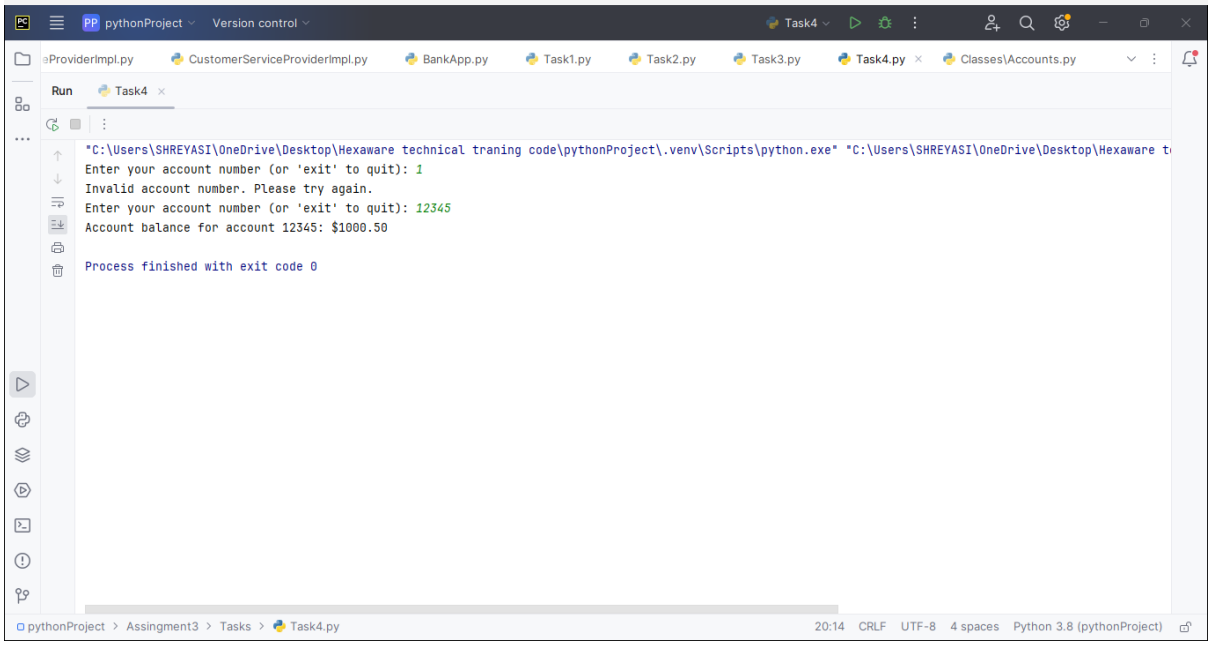
3. Validate the account number entered by the user.

4. If the account number is valid, display the account balance. If not, ask the user to try again.

```python
customer_accounts = {
    '12345': 1000.50,
    '56789': 5000.25,
    '98765': 750.80
}

1 usage
def check_account_balance(account_number):

    if account_number in customer_accounts:
        return customer_accounts[account_number]
    else:
        return None

while True:

    account_number = input("Enter your account number (or 'exit' to quit): ")

    if account_number.lower() == 'exit':
        print("Exiting the program.")
        break

    balance = check_account_balance(account_number)
    if balance is not None:
        print(f"Account balance for account {account_number}: ${balance:.2f}")
        break
    else:
```

```python
def check_account_balance(account_number):

    if account_number in customer_accounts:
        return customer_accounts[account_number]
    else:
        return None

while True:

    account_number = input("Enter your account number (or 'exit' to quit): ")

    if account_number.lower() == 'exit':
        print("Exiting the program.")
        break

    balance = check_account_balance(account_number)
    if balance is not None:
        print(f"Account balance for account {account_number}: ${balance:.2f}")
        break
    else:
        print("Invalid account number. Please try again.")
```

Output



Task 5: Password Validation

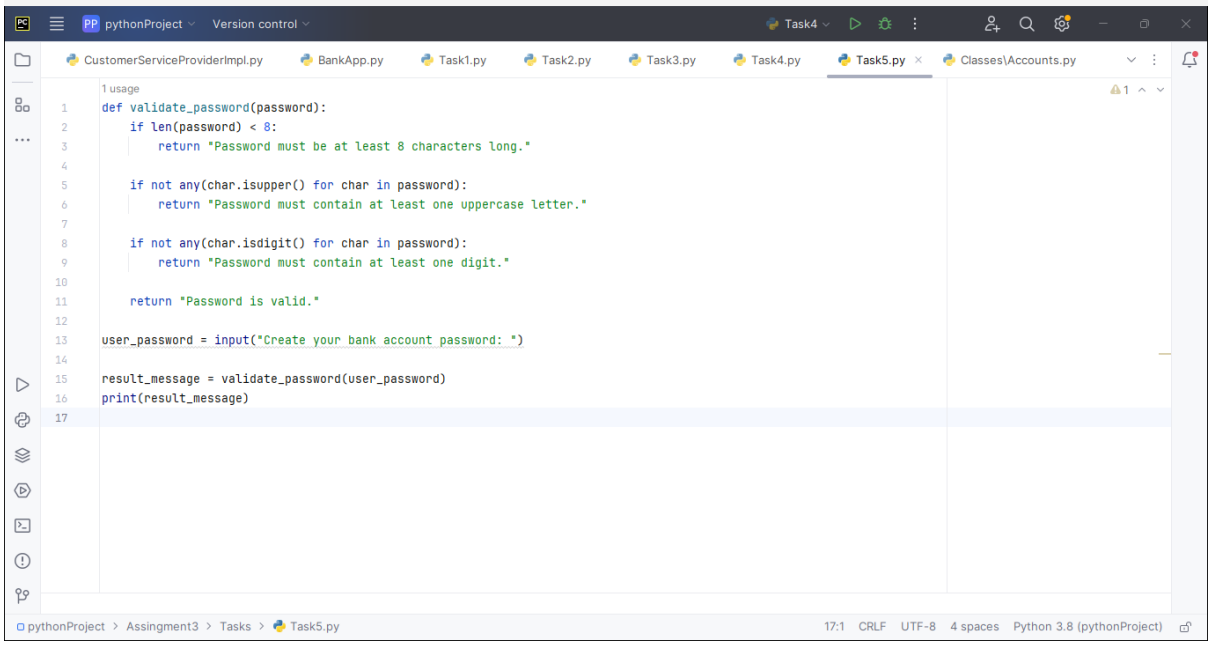
Write a program that prompts the user to create a password for their bank account. Implement if conditions to validate the password according to these rules:

The password must be at least 8 characters long.

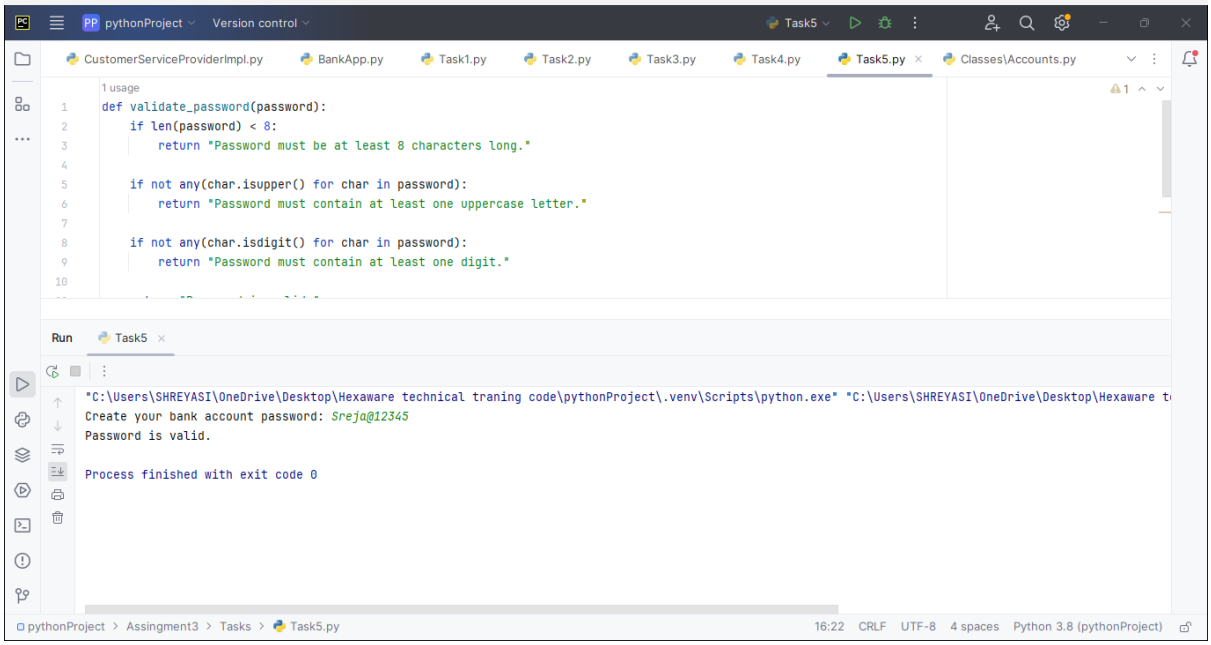
It must contain at least one uppercase letter.

It must contain at least one digit.

Display appropriate messages to indicate whether their password is valid or not.



Output



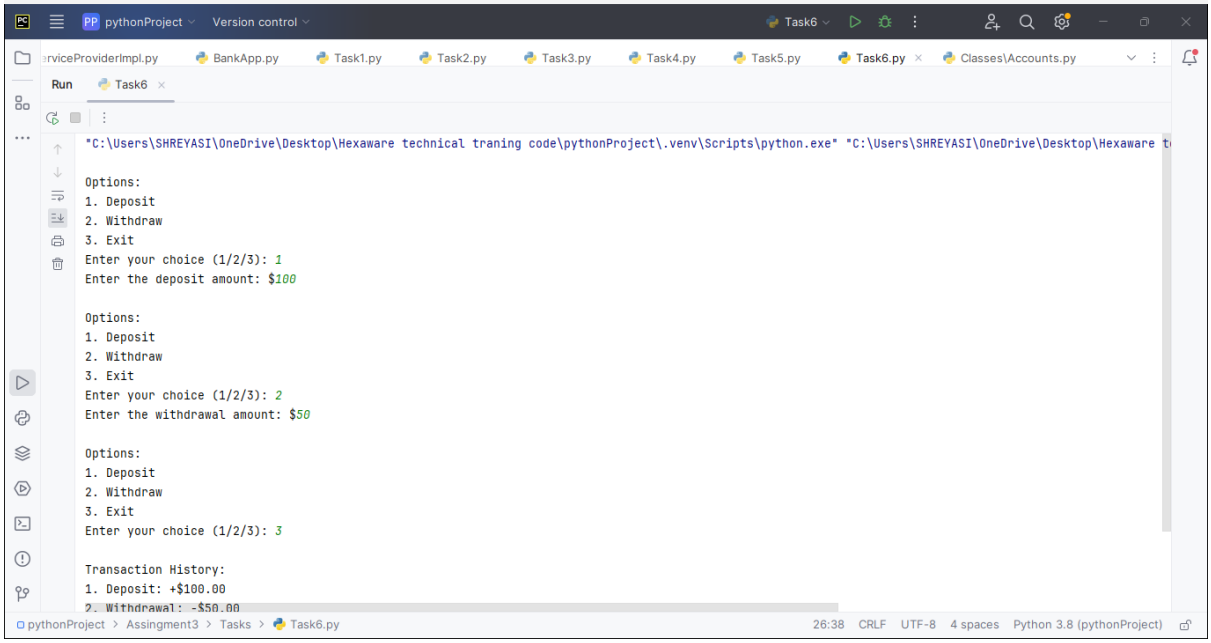
Task 6: Password Validation

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.

```python
transaction_history = []

def display_transaction_history():
    print("\nTransaction History:")
    for i, transaction in enumerate(transaction_history, start=1):
        print(f"{i}. {transaction}")

while True:
    print("\nOptions:")
    print("1. Deposit")
    print("2. Withdraw")
    print("3. Exit")

    choice = input("Enter your choice (1/2/3): ")

    if choice == '1':
        deposit_amount = float(input("Enter the deposit amount: $"))
        transaction_history.append(f"Deposit: +${deposit_amount:.2f}")

    elif choice == '2':
        withdrawal_amount = float(input("Enter the withdrawal amount: $"))
        transaction_history.append(f"Withdrawal: -${withdrawal_amount:.2f}")

    elif choice == '3':
        display_transaction_history()
        print("Exiting the program.")
```

```python
    choice = input("Enter your choice (1/2/3): ")

    if choice == '1':
        deposit_amount = float(input("Enter the deposit amount: $"))
        transaction_history.append(f"Deposit: +${deposit_amount:.2f}")

    elif choice == '2':
        withdrawal_amount = float(input("Enter the withdrawal amount: $"))
        transaction_history.append(f"Withdrawal: -${withdrawal_amount:.2f}")

    elif choice == '3':
        display_transaction_history()
        print("Exiting the program.")
        break

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

Output



OOPS, Collections and Exception Handling

Task 7: Class & Object

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

Attributes

o Customer ID

o First Name

o Last Name

o Email Address

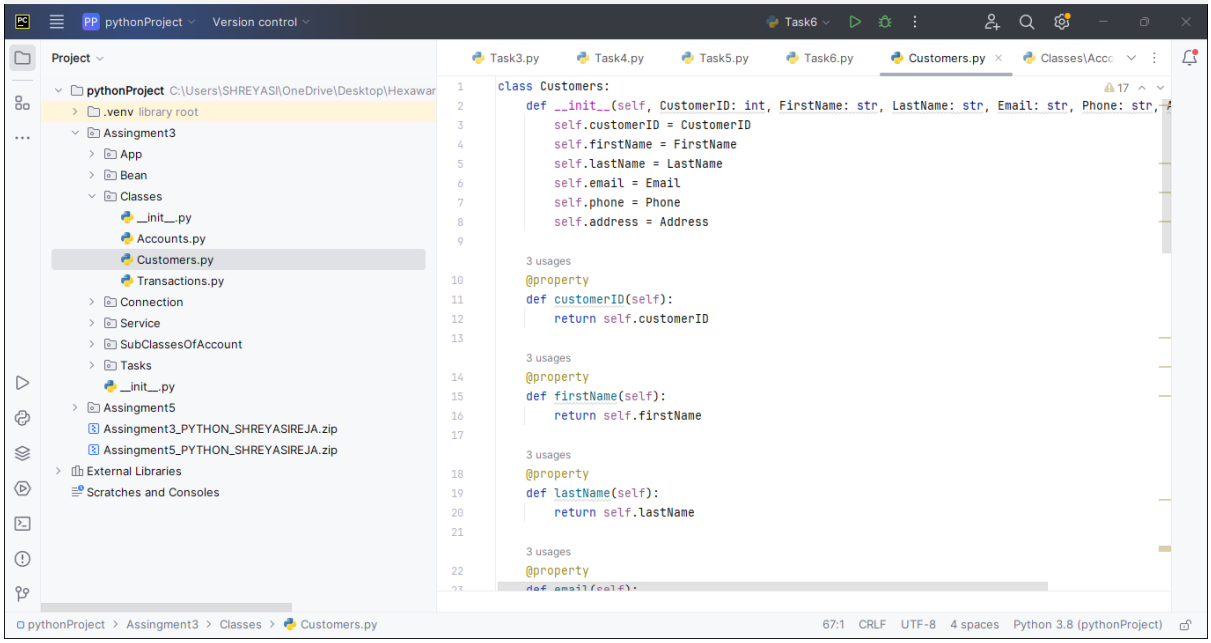
o Phone Number

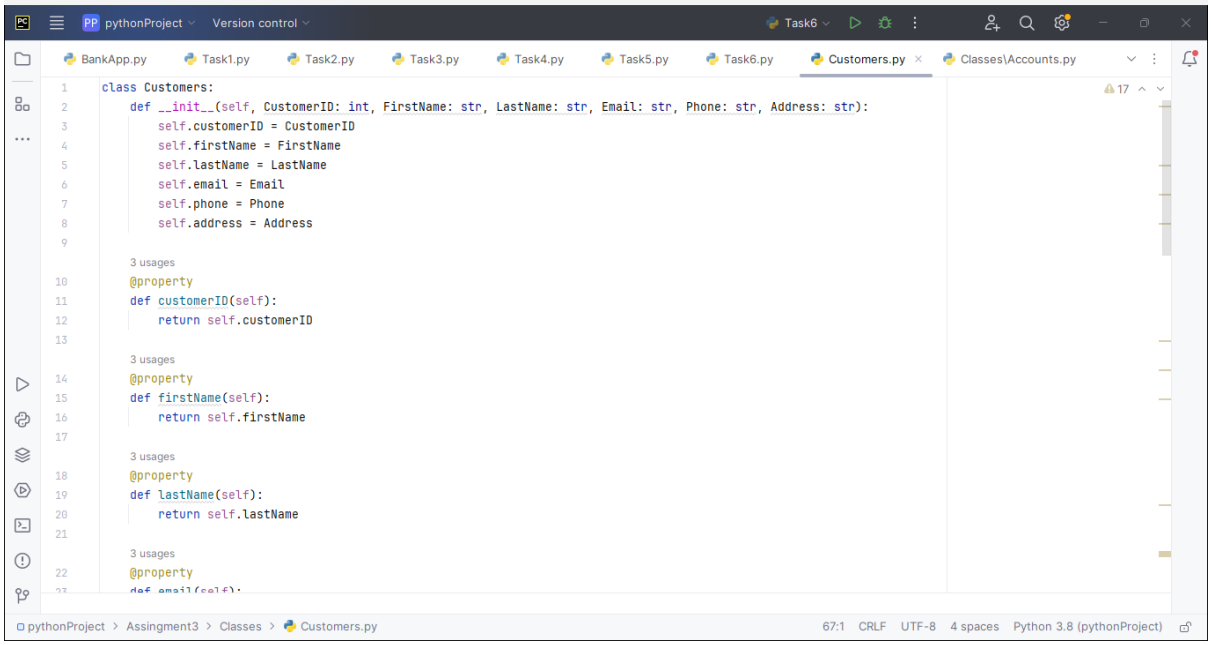
o Address

Constructor and Methods

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

CUSTOMER:-


pythonProject
Version control
Task6
Project
pythonProject C:\Users\SHREYASI\OneDrive\Desktop\Hexawar
.venv library root
Assingment3
App
Bean
Classes
__init__.py
Accounts.py
Customers.py
Transactions.py
Connection
Service
SubClassesOfAccount
Tasks
__init__.py
Assingment5
Assingment3_PYTHON_SHREYASIREJA.zip
Assingment5_PYTHON_SHREYASIREJA.zip
External Libraries
Scratches and Consoles
Task3.py
Task4.py
Task5.py
Task6.py
Customers.py
Classes\Acco
class Customers:
def __init__(self, CustomerID: int, FirstName: str, LastName: str, Email: str, Phone: str,
self.customerID = CustomerID
self.firstName = FirstName
self.lastName = LastName
self.email = Email
self.phone = Phone
self.address = Address
3 usages
@property
def customerID(self):
return self.customerID
3 usages
@property
def firstName(self):
return self.firstName
3 usages
@property
def lastName(self):
return self.lastName
3 usages
@property
pythonProject > Assingment3 > Classes > Customers.py
67:1 CRLF UTF-8 4 spaces Python 3.8 (pythonProject)


pythonProject
Version control
Task6
BankApp.py
Task1.py
Task2.py
Task3.py
Task4.py
Task5.py
Task6.py
Customers.py
Classes\Accounts.py
class Customers:
def __init__(self, CustomerID: int, FirstName: str, LastName: str, Email: str, Phone: str, Address: str):
self.customerID = CustomerID
self.firstName = FirstName
self.lastName = LastName
self.email = Email
self.phone = Phone
self.address = Address
3 usages
@property
def customerID(self):
return self.customerID
3 usages
@property
def firstName(self):
return self.firstName
3 usages
@property
def lastName(self):
return self.lastName
3 usages
@property
pythonProject > Assingment3 > Classes > Customers.py
67:1 CRLF UTF-8 4 spaces Python 3.8 (pythonProject)

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

    3 usages
    @property
    def phone(self):
        return self.phone

    3 usages
    @property
    def address(self):
        return self.address

    2 usages
    @customerID.setter
    def customerID(self, customerID):
        self.customerID = customerID

    2 usages
    @firstName.setter
    def firstName(self, firstName):
        self.firstName = firstName

    2 usages
    @lastName.setter
```

```python
    @lastName.setter
    def lastName(self, lastName):
        self.lastName = lastName

    2 usages
    @email.setter
    def email(self, email):
        self.email = email

    2 usages
    @phone.setter
    def phone(self, phone):
        self.phone = phone

    2 usages
    @address.setter
    def address(self, address):
        self.address = address

    def getCustomerDetails(self):
        print("Customer Details: ")
        print(f"ID: {self.customerID}")
        print(f"Name: {self.firstName} {self.lastName}")
        print(f"Email: {self.email}")
        print(f"Phone: {self.phone}")
        print(f"Address: {self.address}")
```

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

Attributes

o Account Number

o Account Type (e.g., Savings, Current)

o Account Balance

Constructor and Methods

o Implement default constructors and overload the constructor with Account attributes,

o Generate getter and setter, (print all information of attribute) methods for the attributes.

o Add methods to the `Account` class to allow deposits and withdrawals.

- deposit(amount: float): Deposit the specified amount into the account.

- withdraw(amount: float): Withdraw the specified amount from the account. withdraw amount only if there is sufficient fund else display insufficient balance.

- calculate_interest(): method for calculating interest amount for the available balance. interest rate is fixed to 4.5%

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

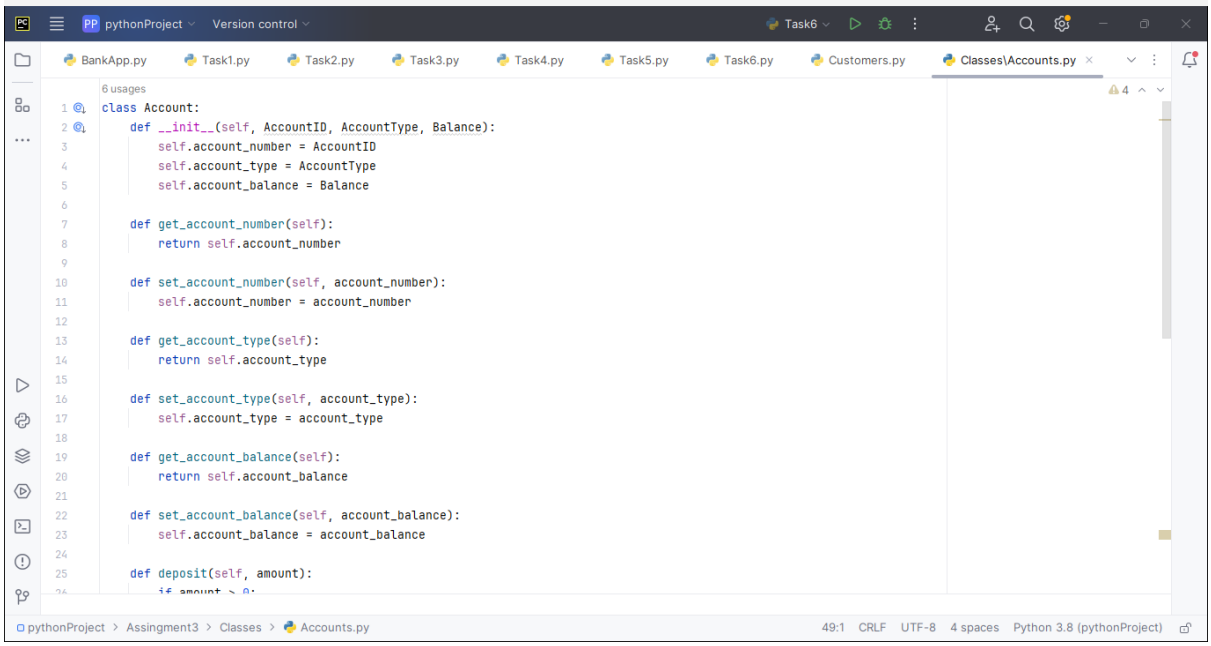
o create object for account class by calling parameter constructor.

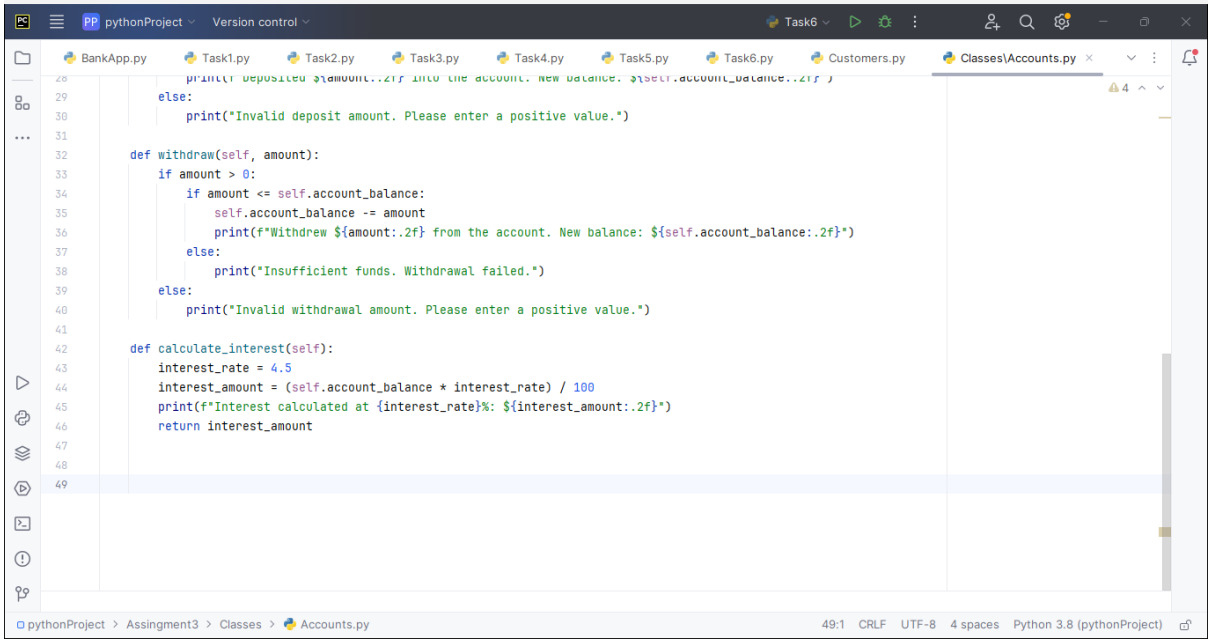
o deposit(amount: float): Deposit the specified amount into the account.

o withdraw(amount: float): Withdraw the specified amount from the account.

o calculate_interest(): Calculate and add interest to the account balance for savings accounts.

ACCOUNT:-





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.

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.

ANS: USE INHERITANCE IN TASK 7 ACCOUNTS CLASS.

Task 9: Abstraction

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

Attributes:

o Account number.

o Customer name.

o Balance.

Constructors:

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

Abstract methods:

o deposit(amount: float): Deposit the specified amount into the account.

o withdraw(amount: float): Withdraw the specified amount from the account (implement error handling for insufficient funds).

o calculate_interest(): Abstract method for calculating interest.

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

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.

ANS:-USE ABSTRACTION IN TASK 7 ACCOUNTS CLASS.

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

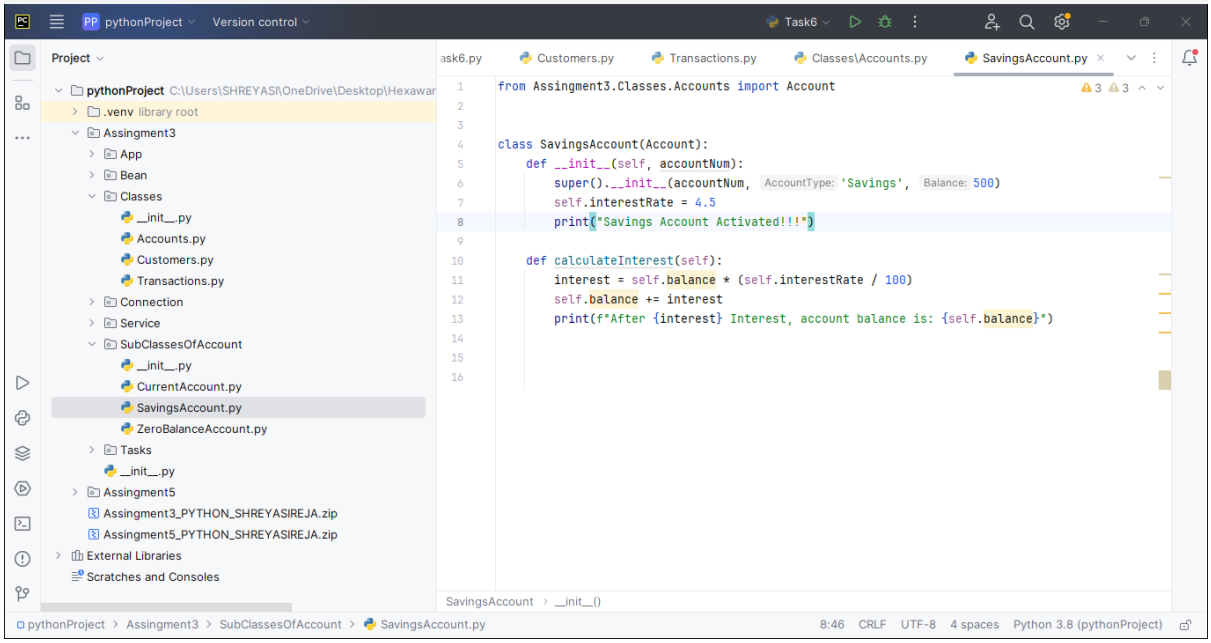
Customer (the customer who owns the account)

LastAccNo

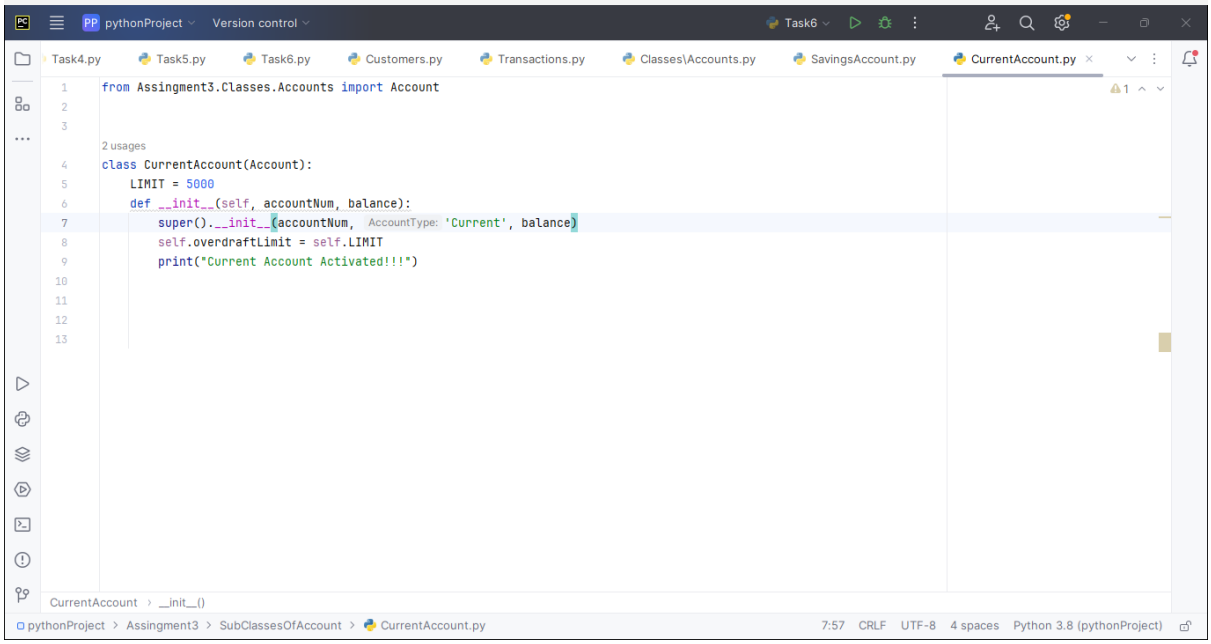
ANS:-TASK DONE IN TASK 7 ACCOUNTS CLASS AND CUSTOMER CLASS.

3. Create three child classes that inherit the Account class and each class must contain below mentioned attribute:

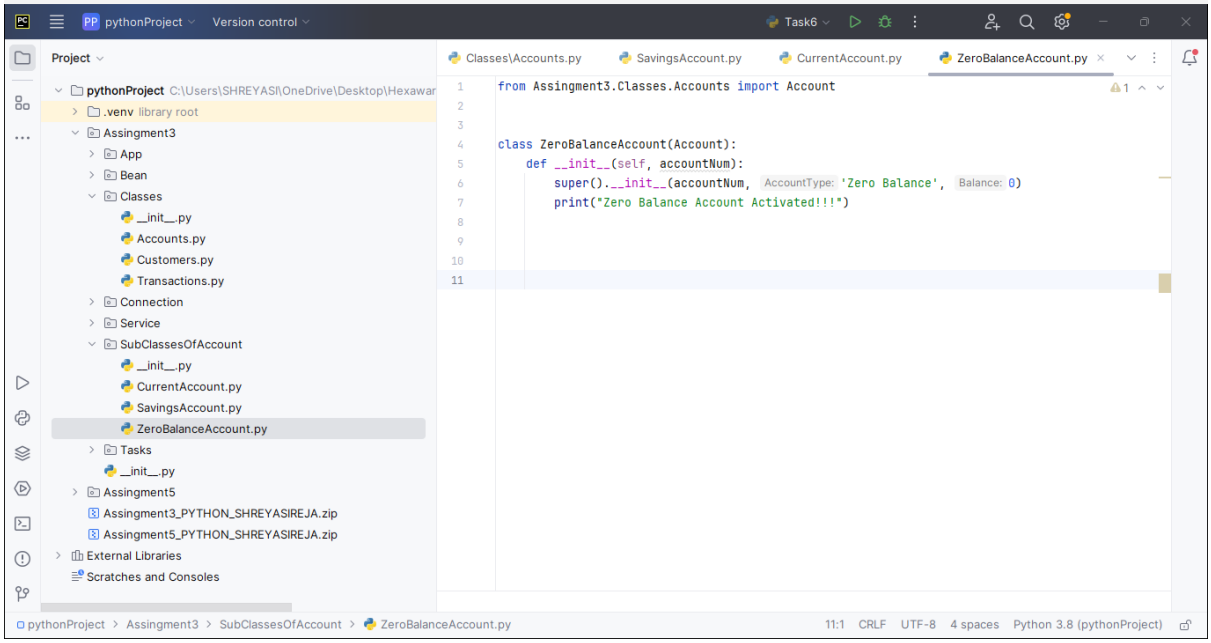
SavingsAccount: A savings account that includes an additional attribute for interest rate. Saving account should be created with minimum balance 500.



CurrentAccount: A Current account that includes an additional attribute for overdraftLimit(credit limit). withdraw() method to allow overdraft up to a certain limit. withdraw limit can exceed the available balance and should not exceed the overdraft limit.



ZeroBalanceAccount: ZeroBalanceAccount can be created with Zero balance.



4. Create ICustomerServiceProvider interface/abstract class with following functions:

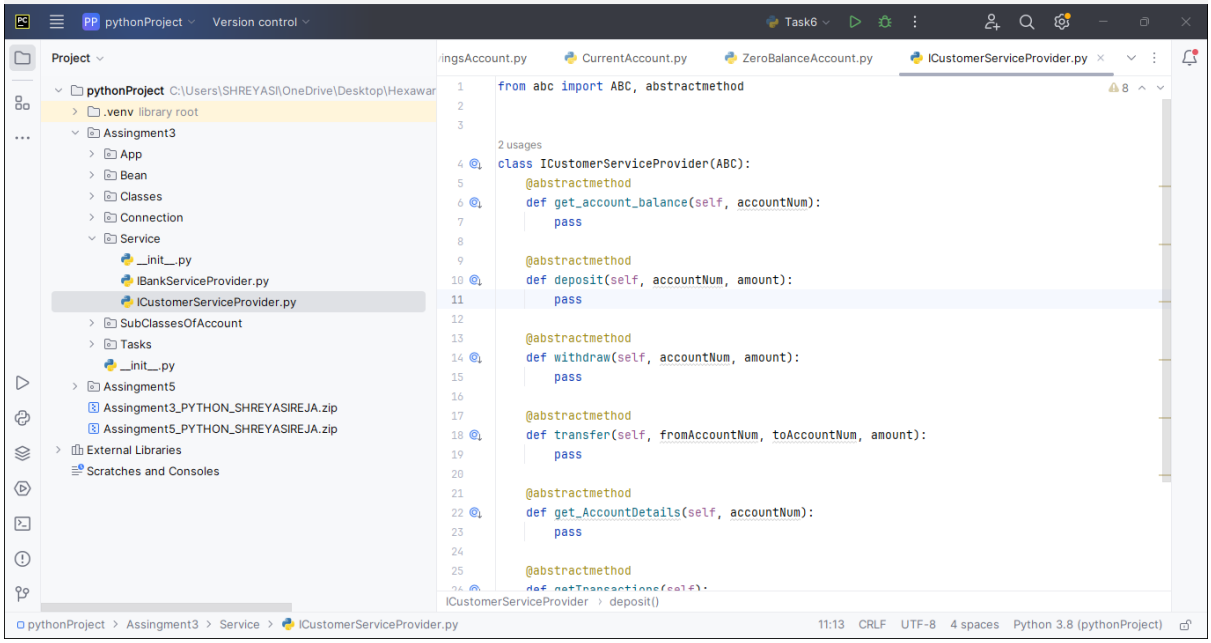
get_account_balance(account_number: long): Retrieve the balance of an account given its account number. should return the current balance of account.

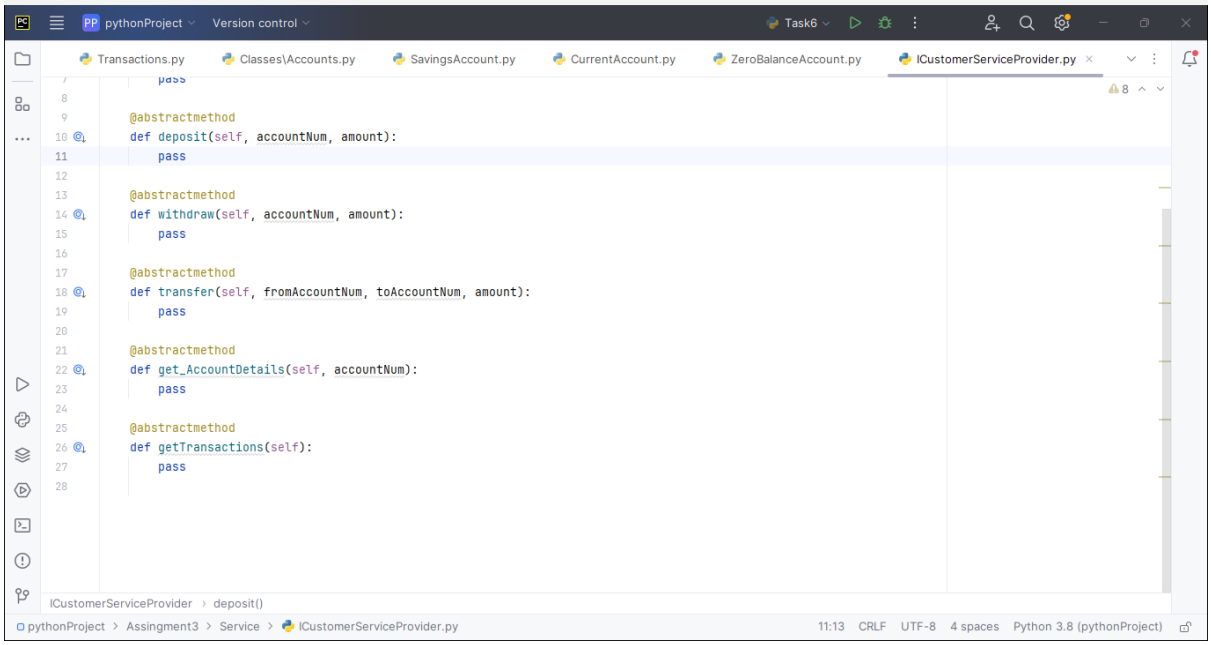
deposit(account_number: long, amount: float): Deposit the specified amount into the account. Should return the current balance of account.

withdraw(account_number: long, amount: float): Withdraw the specified amount from the account. Should return the current balance of account. A savings account should maintain a minimum balance and checking if the withdrawal violates the minimum balance rule.

transfer(from_account_number: long, to_account_number: int, amount: float): Transfer money from one account to another.

getAccountDetails(account_number: long): Should return the account and customer details.





5. Create IBankServiceProvider interface/abstract class with following functions:

create_account(Customer customer, long accNo, String accType, float balance): Create a new bank account for the given customer with the initial balance.

listAccounts():Account[] accounts: List all accounts in the bank.

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

```python
from abc import ABC, abstractmethod


class IBankServiceProvider(ABC):
    @abstractmethod
    def create_account(self):
        pass

    @abstractmethod
    def listAccounts(self):
        pass

    @abstractmethod
    def get_AccountDetails(self, accountNum):
        pass

    @abstractmethod
    def calculateInterest(self, inRate):
        pass
```

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

```python
from Assingment3.Service.ICustomerServiceProvider import ICustomerServiceProvider
from Assingment3.SubClassesOfAccount.CurrentAccount import CurrentAccount


class CustomerServiceProviderImpl(ICustomerServiceProvider):
    def __init__(self, dbUtil):
        self.dbUtil = dbUtil

    def get_account_balance(self, accountNum):
        query = "Select balance from accounts where accountID = %s"
        value = (accountNum,)
        result = self.dbUtil.fetchOne(query, value)
        return result[0]

    def deposit(self, accountNum, amount):
        if amount <= 0:
            print("Invalid Amount")
        else:
            if self.get_AccountDetails(accountNum) is not None:
                balance = self.get_account_balance(accountNum)
                newBal = float(balance) + amount
                query = "Update accounts set balance=%s where accountID = %s"
                values = (newBal, accountNum)
```

pythonProject Version control Task6

:count.py CurrentAccount.py ZeroBalanceAccount.py ICustomerServiceProvider.py IBankServiceProvider.py CustomerServiceProviderImpl.py

```python
    2 usages
    def deposit(self, accountNum, amount):
        if amount <= 0:
            print("Invalid Amount")
        else:
            if self.get_AccountDetails(accountNum) is not None:
                balance = self.get_account_balance(accountNum)
                newBal = float(balance) + amount
                query = "Update accounts set balance=%s where accountID = %s"
                values = (newBal, accountNum)
                self.dbUtil.executeQuery(query, values)
                print(f"New Balance is: {newBal}")
            else:
                raise Exception("InvalidAccountIDException")

    2 usages
    def withdraw(self, accountNum, amount):
        currentBal = self.get_account_balance(accountNum)
        accountType = self.getAccountType(accountNum)

        if self.get_AccountDetails(accountNum) is not None:
            if amount <= 0:
                return "Invalid Amount"
            else:
                if accountType == 'savings':
                    if amount > currentBal:
```

CustomerServiceProviderImpl › get_account_balance()

pythonProject › Assingment3 › Bean › CustomerServiceProviderImpl.py 9:47 CRLF UTF-8 4 spaces Python 3.8 (pythonProject)

pythonProject Version control Task6

:count.py CurrentAccount.py ZeroBalanceAccount.py ICustomerServiceProvider.py IBankServiceProvider.py CustomerServiceProviderImpl.py

```python
            else:
                if accountType == 'savings':
                    if amount > currentBal:
                        raise Exception("InsufficientFundException")
                    else:
                        if float(currentBal) - amount < 500.00:
                            raise Exception("MinimumBalanceLimitException")
                        else:
                            query = "Update accounts set balance=balance-%s where accountID = %s"
                            values = (amount, accountNum)
                            self.dbUtil.executeQuery(query, values)
                            result = self.get_account_balance(accountNum)
                            return result
                elif accountType == 'current':
                    if amount > currentBal and amount - currentBal > CurrentAccount.LIMIT:
                        raise Exception("OverdraftLimitExceededException")
                    else:
                        query = "Update accounts set balance=balance-%s where accountID = %s"
                        values = (amount, accountNum)
                        self.dbUtil.executeQuery(query, values)
                        result = self.get_account_balance(accountNum)
                        return result
                else:
                    raise Exception("ZeroBalanceAccountException")
        else:
            raise Exception("InvalidAccountIDException")
```

CustomerServiceProviderImpl › get_account_balance()

pythonProject › Assingment3 › Bean › CustomerServiceProviderImpl.py 9:47 CRLF UTF-8 4 spaces Python 3.8 (pythonProject)

```python
                else:
                    raise Exception("ZeroBalanceAccountException")
            else:
                raise Exception("InvalidAccountIDException")

    1 usage
    def transfer(self, fromAccountNum, toAccountNum, amount):
        if self.get_AccountDetails(fromAccountNum) and self.get_AccountDetails(toAccountNum):
            try:
                self.withdraw(fromAccountNum, amount)
                self.deposit(toAccountNum, amount)
            except Exception as e:
                return f"Transfer Failed!!! Error: {e}"

    6 usages
    def get_AccountDetails(self, accountNum):
        query = "Select * from accounts join customers on accounts.customerID = customers.customerID where accountID=%s"
        value = (accountNum,)
        result = self.dbUtil.fetchall(query, value)
        return result

    1 usage
    def getTransactions(self):
        accountNum = input("Enter your accountID: ")
        fromDate = input("From: ")
        toDate = input("To: ")
```

```python
    def get_AccountDetails(self, accountNum):
        query = "Select * from accounts join customers on accounts.customerID = customers.customerID where accountID=%s"
        value = (accountNum,)
        result = self.dbUtil.fetchall(query, value)
        return result

    1 usage
    def getTransactions(self):
        accountNum = input("Enter your accountID: ")
        fromDate = input("From: ")
        toDate = input("To: ")
        if self.get_AccountDetails(accountNum) is not None:
            query = "Select * from transactions where transaction_date between %s and %s and accountID = %s"
            values = (fromDate, toDate, accountNum)
            result = self.dbUtil.fetchall(query, values)
            return result
        else:
            raise Exception("InvalidAccountIDException")

    2 usages
    def getAccountType(self, accountNum):
        value = (accountNum,)
        result = self.dbUtil.fetchOne("Select account_type from accounts where accountID = %s", value)
        return result[0]
```

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

Attributes

o accountList: Array of Accounts to store any account objects.

o branchName and branchAddress as String objects

```python
from Assingment3.Service.IBankServiceProvider import IBankServiceProvider


class BankServiceProviderImpl(IBankServiceProvider):
    def __init__(self, dbUtil):
        self.dbUtil = dbUtil

    def create_account(self):
        customerID = input("Please enter your customerID: ")
        if self.existingCustomer(customerID) is None:
            raise Exception("CustomerNotRegisteredException")
        else:
            print("Please fill up the account details: ")
            account = {
                'accountID': self.generateUniqueAccountID(),
                'customerID': customerID,
            }
            print("What type of account do you want to create?")
            print("1. Savings Account")
            print("2. Current Account")
            print("3. Zero Balance Account")
            ch = int(input("Enter your choice: "))

            if ch == 1:
```

```python
            if ch == 1:
                    account['account_type'] = 'savings'
                    account['balance'] = float(input("Enter the amount: "))
                    return self.AddAccountIntoDatabase(account)

            elif ch == 2:
                    account['account_type'] = 'current'
                    account['balance'] = float(input("Enter the amount: "))
                    return self.AddAccountIntoDatabase(account)

            elif ch == 3:
                    account['account_type'] = 'zero balance'
                    account['balance'] = 0.0
                    return self.AddAccountIntoDatabase(account)
            else:
                print("Invalid choice. Please enter a valid choice.")

    def listAccounts(self):
        query = "SELECT accountID, customerID, account_type, balance FROM accounts"
        result = self.dbUtil.fetchall(query, values=None)
        print(result)

    def get_AccountDetails(self, accountNum):
        query = "Select * from accounts where accountID=(%s)"
        value = (accountNum,)
```

```python
    def listAccounts(self):
        query = "SELECT accountID, customerID, account_type, balance FROM accounts"
        result = self.dbUtil.fetchall(query, values=None)
        print(result)

    def get_AccountDetails(self, accountNum):
        query = "Select * from accounts where accountID=(%s)"
        value = (accountNum,)
        return self.dbUtil.fetchall(query, value)

    def calculateInterest(self, inRate=4.5):
        query = "select balance * %s as inRate from accounts"
        return self.dbUtil.fetchall(query, inRate)

    1 usage
    def get_no_of_accounts(self):
        query = "Select count(*) from accounts"
        result = self.dbUtil.fetchOne(query)
        return result[0]

    1 usage
    def generateUniqueAccountID(self):
        concat = ('B0', str(self.get_no_of_accounts()+ 1))
        return "".join(concat)

    1 usage
```

```python
        query = "Select count(*) from accounts"
        result = self.dbUtil.fetchOne(query)
        return result[0]

    1 usage
    def generateUniqueAccountID(self):
        concat = ('B0', str(self.get_no_of_accounts()+ 1))
        return "".join(concat)

    1 usage
    def existingCustomer(self, customerID):
        query = "Select * from Customers where customerID = %s"
        value = (customerID,)
        return self.dbUtil.fetchOne(query, value)

    3 usages
    def AddAccountIntoDatabase(self, accounts):
        query = "insert into accounts values(%s, %s, %s, %s)"
        values = (accounts['accountID'], accounts['customerID'], accounts['account_type'], accounts['balance'])
        return self.dbUtil.executeQuery(query, values)
```

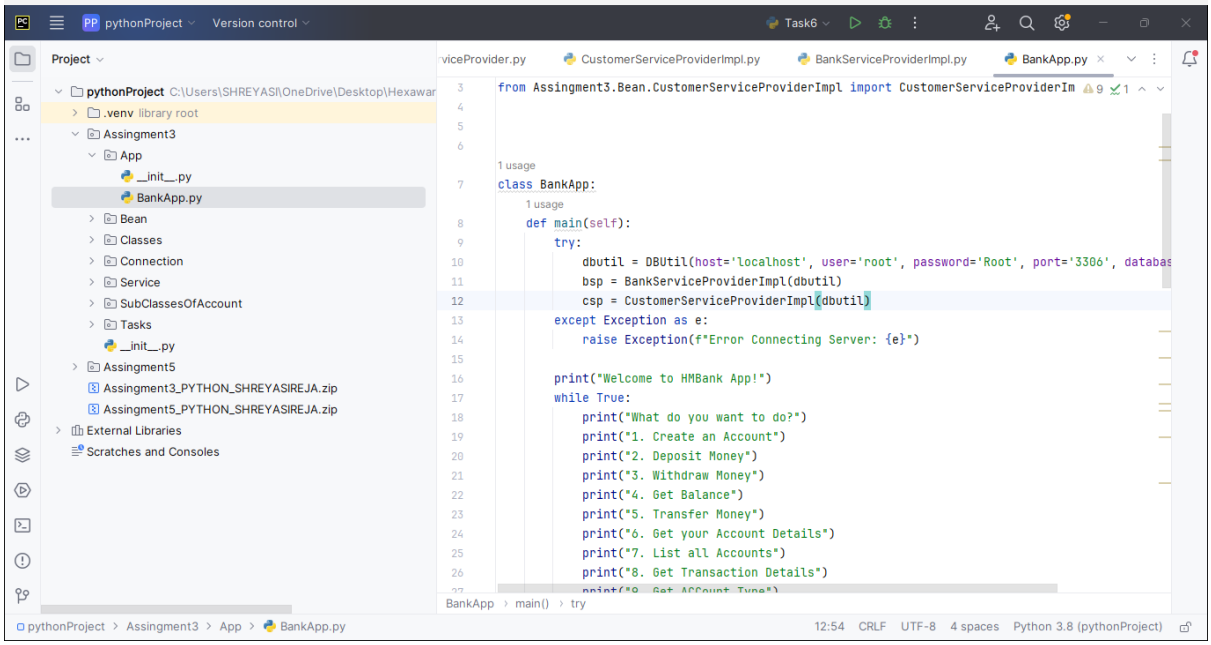
8. Create BankApp class and perform following operation:

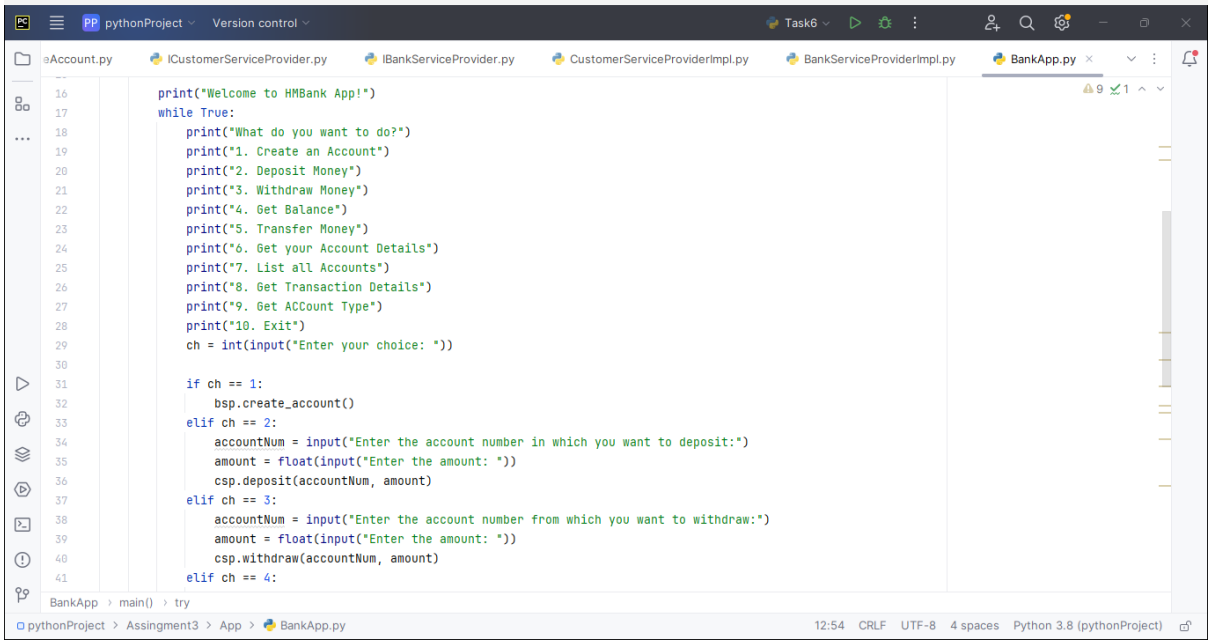
main method to simulate the banking system. Allow the user to interact with the system by entering choice from menu such as "create_account", "deposit", "withdraw", "get_balance", "transfer", "getAccountDetails", "ListAccounts" and "exit."

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

9. Place the interface/abstract class in service package and interface/abstract class implementation class, account class in bean package and Bank class in app package.

BANK APP:-

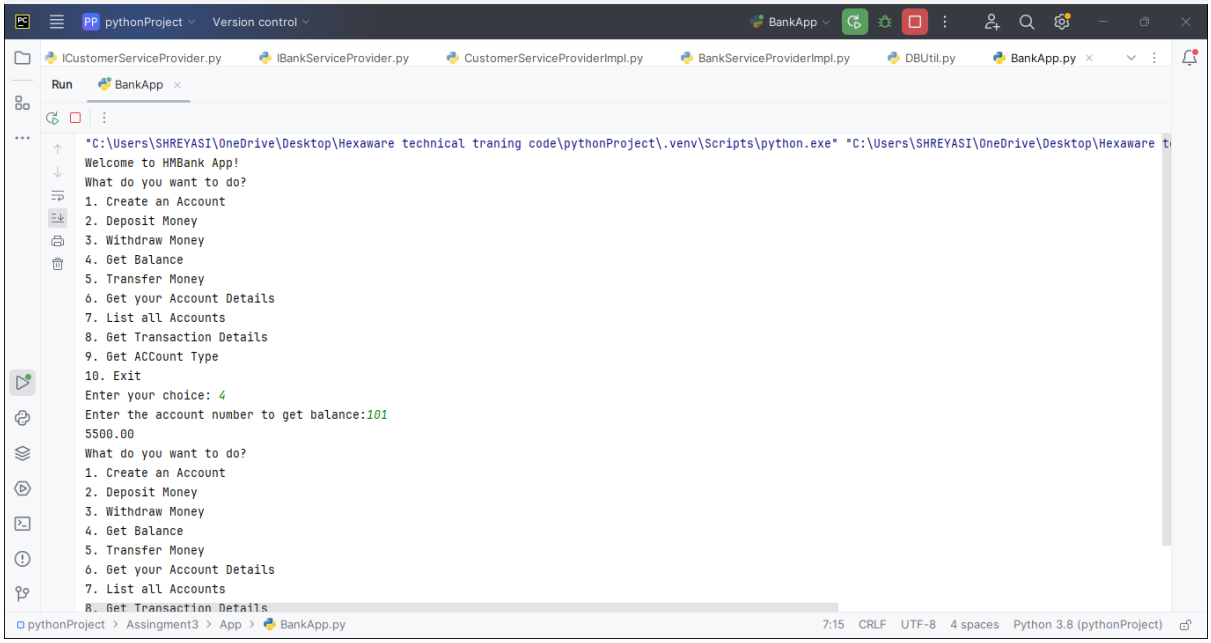




```python
            elif ch == 4:
                accountNum = input("Enter the account number to get balance:")
                print(csp.get_account_balance(accountNum))
            elif ch == 5:
                fromAcc = input("AccountID of the sender: ")
                toAcc = input("AccountID of the receiver: ")
                amount = float(input("Enter the amount: "))
                csp.transfer(fromAcc, toAcc, amount)
            elif ch == 6:
                accountNum = input("Enter the account number to get details:")
                print(csp.get_AccountDetails(accountNum))
            elif ch == 7:
                bsp.listAccounts()
            elif ch == 8:
                print(csp.getTransactions())
            elif ch == 9:
                accountNum = input("Enter your accountID: ")
                print(csp.getAccountType(accountNum))
            elif ch == 10:
                dbutil.closeConnection()
                break


if __name__ == '__main__':
    obj = BankApp()
    obj.main()
```

```python
            toAcc = input("AccountID of the receiver: ")
            amount = float(input("Enter the amount: "))
            csp.transfer(fromAcc, toAcc, amount)
        elif ch == 6:
            accountNum = input("Enter the account number to get details:")
            print(csp.get_AccountDetails(accountNum))
        elif ch == 7:
            bsp.listAccounts()
        elif ch == 8:
            print(csp.getTransactions())
        elif ch == 9:
            accountNum = input("Enter your accountID: ")
            print(csp.getAccountType(accountNum))
        elif ch == 10:
            dbutil.closeConnection()
            break


if __name__ == '__main__':
    obj = BankApp()
    obj.main()
```

OUTPUT:-



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.

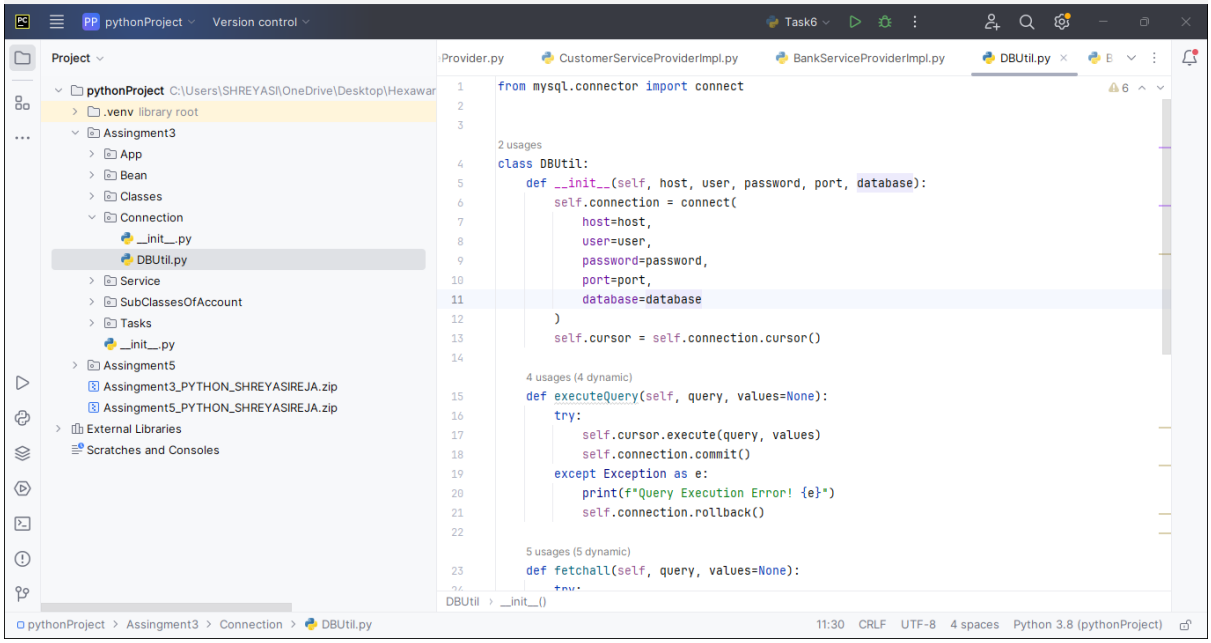
Throw these exceptions from the methods in HMBank class. Make necessary changes to accommodate these exception in the source code. Handle all these exceptions from the main program.

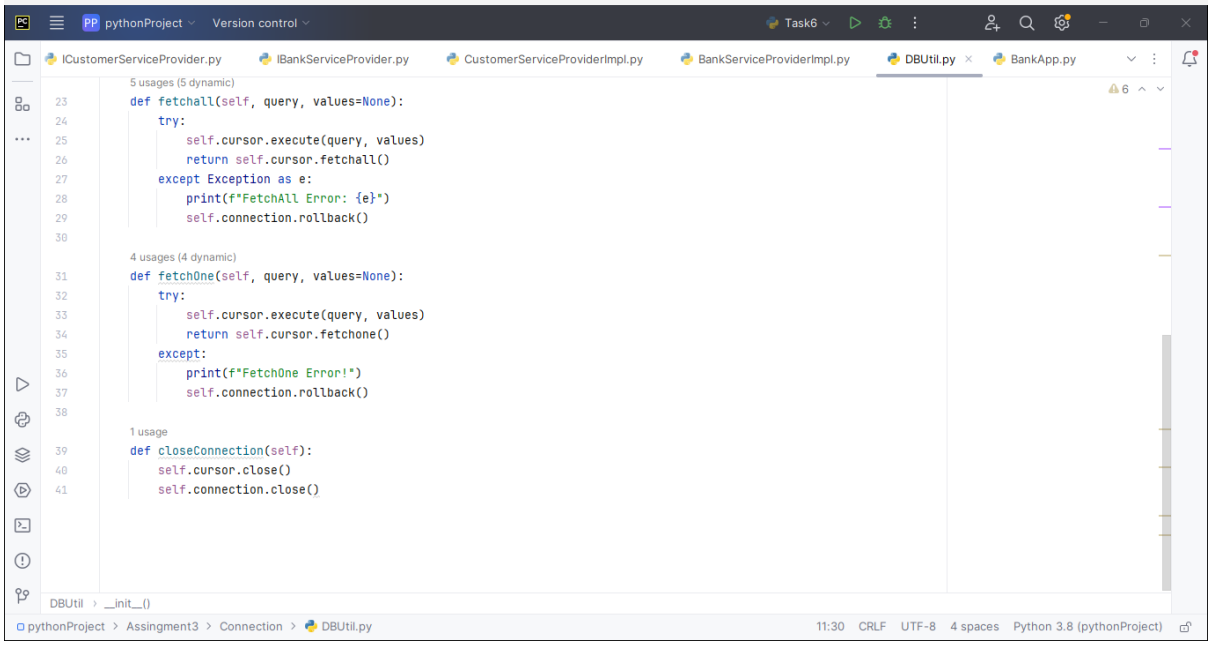
ANS:-USE EXCEPTION HANDLING IN TASK 11 SUBTASK 8 BANK APP.

Task 14: Database Connectivity.

11. Create DBUtil class and add the following method.

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





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.

13. Place the interface/abstract class in service package and interface/abstract class implementation class, account class in bean package and Bank class in app package.

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

ANS:-USE IN TASK 11 SUB TASK 8 BANK APP.