NAME: SHIVAM SINGH ASSIGNMENT 3:

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.

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.

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.

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.

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.

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.

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.
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%

- Create a Bank class to represent the banking system. Perform the following operation in main method:
    - create object for account class by calling parameter constructor.
    - deposit(amount: float): Deposit the specified amount into the account.
    - withdraw(amount: float): Withdraw the specified amount from the account.
    - calculate_interest(): Calculate and add interest to the account balance for savings accounts.

Task 8: Inheritance and polymorphism

1. Overload the deposit and withdraw methods in Account class as mentioned below.
    - deposit(amount: float): Deposit the specified amount into the account.
    - withdraw(amount: float): Withdraw the specified amount from the account. withdraw amount only if there is sufficient fund else display insufficient balance.
    - deposit(amount: int): Deposit the specified amount into the account.
    - withdraw(amount: int): Withdraw the specified amount from the account. withdraw amount only if there is sufficient fund else display insufficient balance.
    - deposit(amount: double): Deposit the specified amount into the account.
    - withdraw(amount: double): Withdraw the specified amount from the account. withdraw amount only if there is sufficient fund else display insufficient balance.
2. Create Subclasses for Specific Account Types
    - Create subclasses for specific account types (e.g., `SavingsAccount`, `CurrentAccount`) that inherit from the `Account` class.
        - **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).
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.

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.
2. Create two concrete classes that inherit from **BankAccount**:
    - **SavingsAccount**: A savings account that includes an additional attribute for interest rate. Implement the calculate_interest() method to calculate interest based on the balance and interest rate.
    - **CurrentAccount**: A current account with no interest. Implement the withdraw() method to allow overdraft up to a certain limit (configure a constant for the overdraft limit).
3. Create a Bank class to represent the banking system. Perform the following operation in main method:
    - Display menu for user to create object for account class by calling parameter constructor. Menu should display options `SavingsAccount` and `CurrentAccount`. user can choose any one option to create account. use switch case for implementation. create_account should display sub menu to choose type of accounts.
        - *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.

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.
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.

Create a Bank Class and must have following requirements:

1. Create a Bank class to represent the banking system. It should have the following methods:
    - **create_account(Customer customer, long accNo, String accType, float balance)**: Create a new bank account for the given customer with the initial balance.
    - **get_account_balance(account_number: long)**: Retrieve the balance of an account given its account number. should return the current balance of account.
    - **deposit(account_number: long, amount: float)**: Deposit the specified amount into the account. Should return the current balance of account.
    - **withdraw(account_number: long, amount: float)**: Withdraw the specified amount from the account. Should return the current balance of account.
    - **transfer(from_account_number: long, to_account_number: int, amount: float)**: Transfer money from one account to another.
    - **getAccountDetails(account_number: long):** Should return the account and customer details.
2. Ensure that account numbers are automatically generated when an account is created, starting from 1001 and incrementing for each new account.

3. Create a BankApp class with a main method to simulate the banking system. Allow the user to interact with the system by entering commands such as "create_account", "deposit", "withdraw", "get_balance", "transfer", "getAccountDetails" and "exit." create_account should display sub menu to choose type of accounts and repeat this operation until user exit.

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

1. Create a **'Customer'** class as mentioned above task.
2. Create an class '**Account'** that includes the following attributes. Generate account number using static variable.
   - Account Number (a unique identifier).
   - Account Type (e.g., Savings, Current)
   - Account Balance
   - Customer (the customer who owns the account)
   - lastAccNo
3. Create three child classes that inherit the Account class and each class must contain below mentioned attribute:
   - **SavingsAccount:** A savings account that includes an additional attribute for interest rate. Saving account should be created with minimum balance 500.
   - **CurrentAccount:** A Current account that includes an additional attribute for overdraftLimit(credit limit). withdraw() method to allow overdraft up to a certain limit. withdraw limit can exceed the available balance and should not exceed the overdraft limit.
   - **ZeroBalanceAccount**: ZeroBalanceAccount can be created with Zero balance.
4. Create **ICustomerServiceProvider** interface/abstract class with following functions:
   - **get_account_balance(account_number: long)**: Retrieve the balance of an account given its account number. should return the current balance of account.
   - **deposit(account_number: long, amount: float)**: Deposit the specified amount into the account. Should return the current balance of account.
   - **withdraw(account_number: long, amount: float)**: Withdraw the specified amount from the account. Should return the current balance of account. A savings account should maintain a minimum balance and checking if the withdrawal violates the minimum balance rule.
   - **transfer(from_account_number: long, to_account_number: int, amount: float)**: Transfer money from one account to another.
   - **getAccountDetails(account_number: long):** Should return the account and customer details.
5. Create **IBankServiceProvider** interface/abstract class with following functions:
   - **create_account(Customer customer, long accNo, String accType, float balance)**: Create a new bank account for the given customer with the initial balance.
   - **listAccounts()**:Account[] accounts: List all accounts in the bank.

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

6. Create **CustomerServiceProviderImpl** class which implements I**CustomerServiceProvider** provide all implementation methods.
7. Create **BankServiceProviderImpl** class which inherits from **CustomerServiceProviderImpl and implements IBankServiceProvider**
    - Attributes
        - accountList: Array of **Accounts** to store any account objects.
        - branchName and branchAddress as String objects
8. Create **BankApp** class and perform following operation:
    - main method to simulate the banking system. Allow the user to interact with the system by entering choice from menu such as "create_account", "deposit", "withdraw", "get_balance", "transfer", "getAccountDetails", "ListAccounts" and "exit."
    - create_account should display sub menu to choose type of accounts and repeat this operation until user exit.
9. Place the interface/abstract class in service package and interface/abstract class implementation class, account class in bean package and Bank class in app package.
10. Should display appropriate message when the account number is not found and insufficient fund or any other wrong information provided.

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.

Task 13: Collection

1. From the previous task change the **HMBank** attribute Accounts to List of Accounts and perform the same operation.
2. From the previous task change the **HMBank** attribute Accounts to Set of Accounts and perform the same operation.
    - Avoid adding duplicate Account object to the set.
    - Create Comparator<Account> object to sort the accounts based on customer name when listAccounts() method called.
3. From the previous task change the HMBank attribute Accounts to HashMap of Accounts and perform the same operation.

Task 14: Database Connectivity.

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

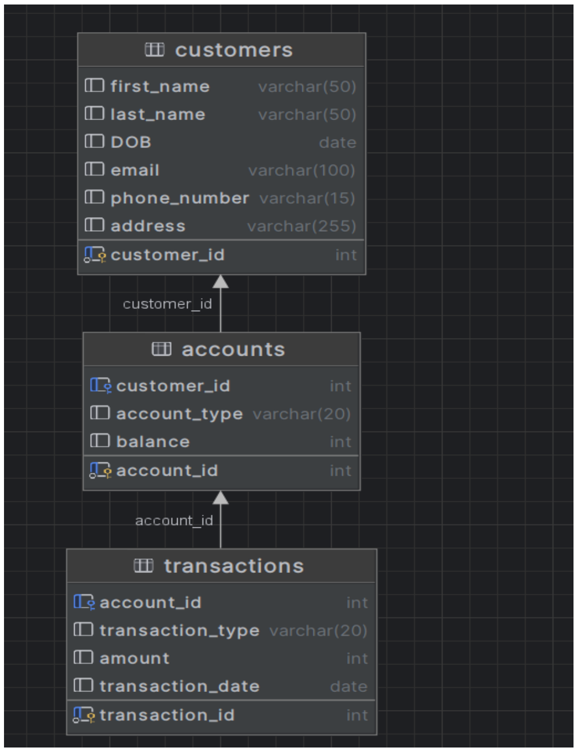
- **getTransations(account_number: long, FromDate:Date, ToDate: Date):** Should return the list of transaction between two dates.

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.
7. Create **CustomerServiceProviderImpl** class which implements I**CustomerServiceProvider** provide all implementation methods. These methods do not interact with database directly.
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
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. Create **BankRepositoryImpl** class which implement the **IBankRepository** interface/abstract class and provide implementation of all methods and perform the database operations.
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**.

Database Structure

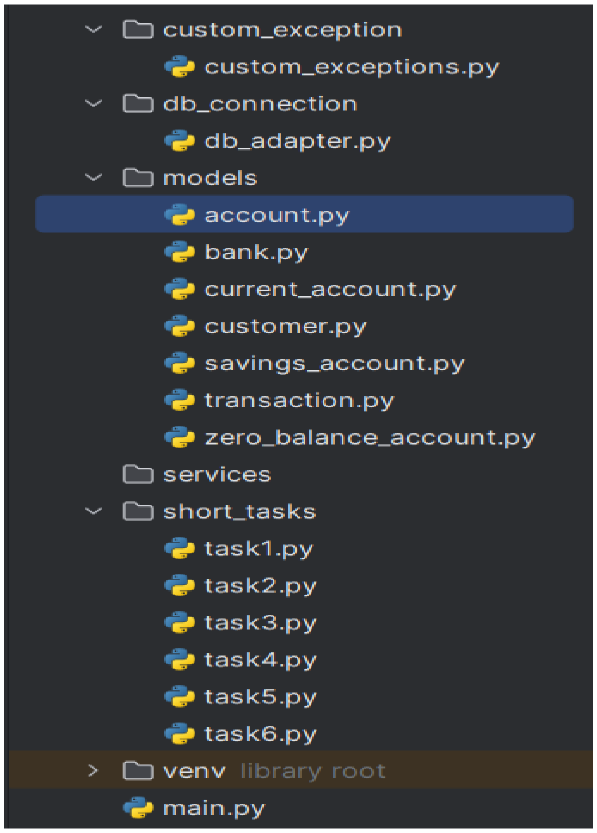


customers	
first_name	varchar(50)
last_name	varchar(50)
DOB	date
email	varchar(100)
phone_number	varchar(15)
address	varchar(255)
customer_id	int

accounts	
customer_id	int
account_type	varchar(20)
balance	int
account_id	int

transactions	
account_id	int
transaction_type	varchar(20)
amount	int
transaction_date	date
transaction_id	int

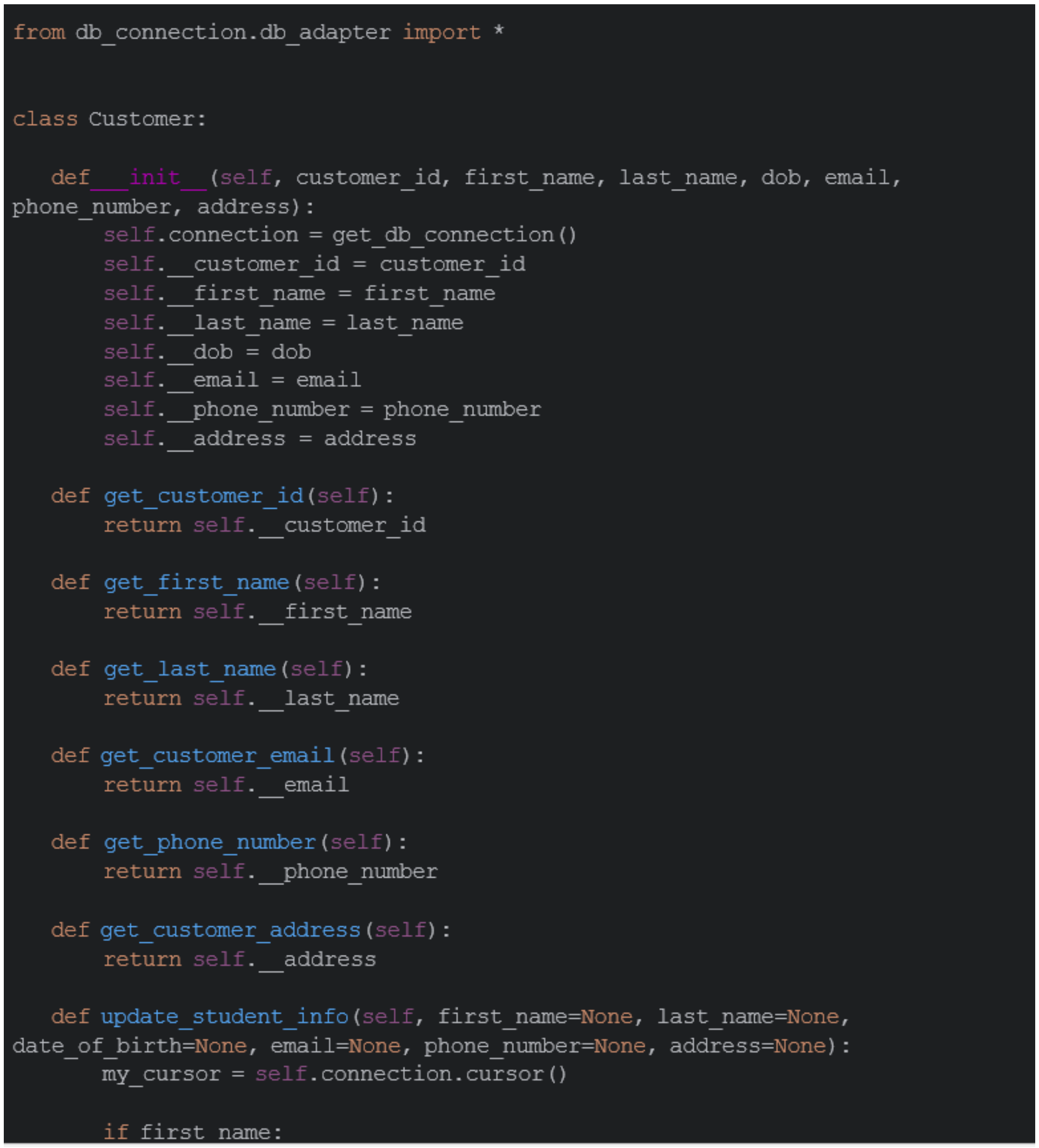
File Structure



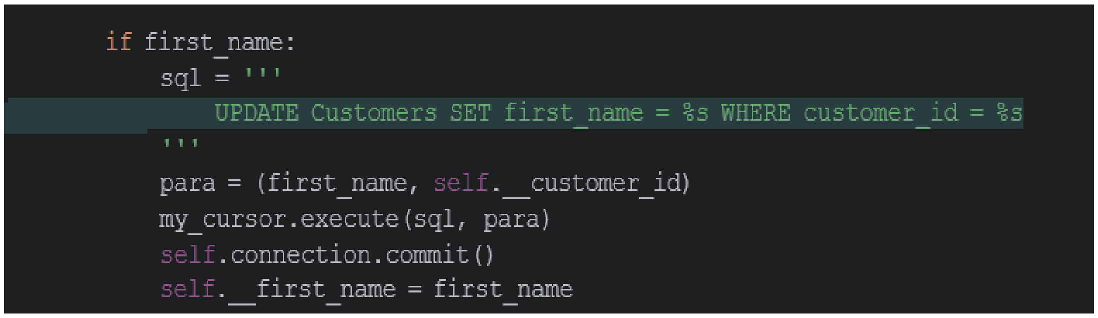
```
⌄ custom_exception
      custom_exceptions.py
⌄ db_connection
      db_adapter.py
⌄ models
      account.py
      bank.py
      current_account.py
      customer.py
      savings_account.py
      transaction.py
      zero_balance_account.py
  services
⌄ short_tasks
      task1.py
      task2.py
      task3.py
      task4.py
      task5.py
      task6.py
> venv  library root
  main.py
```

TASK: 1-13

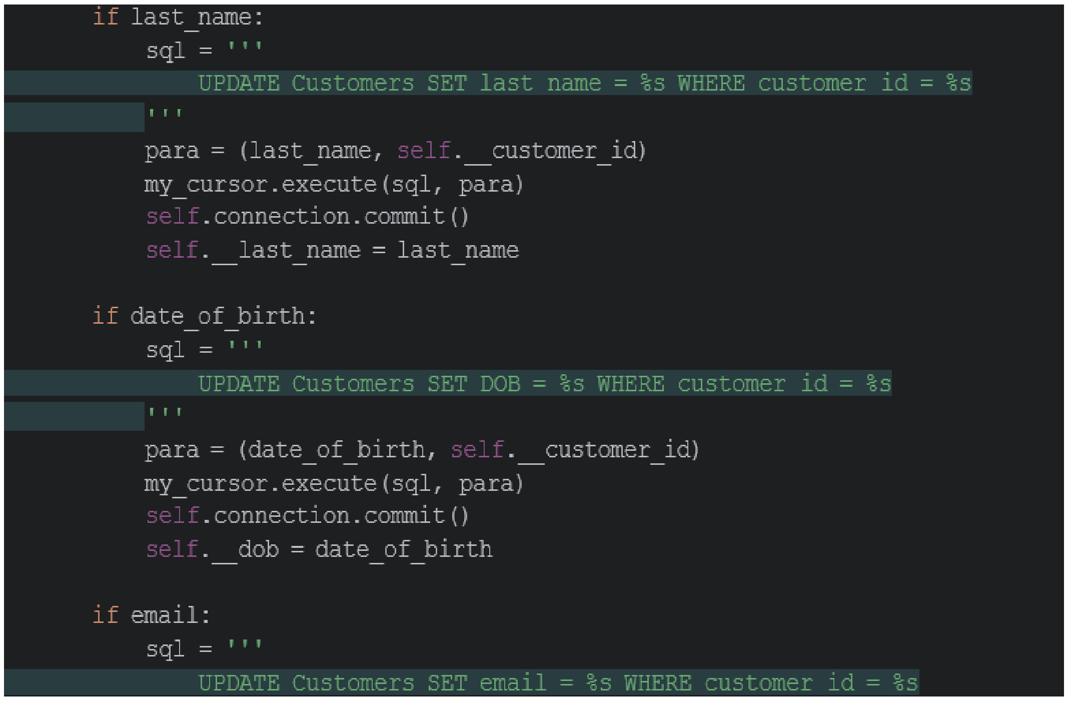
Customer.py



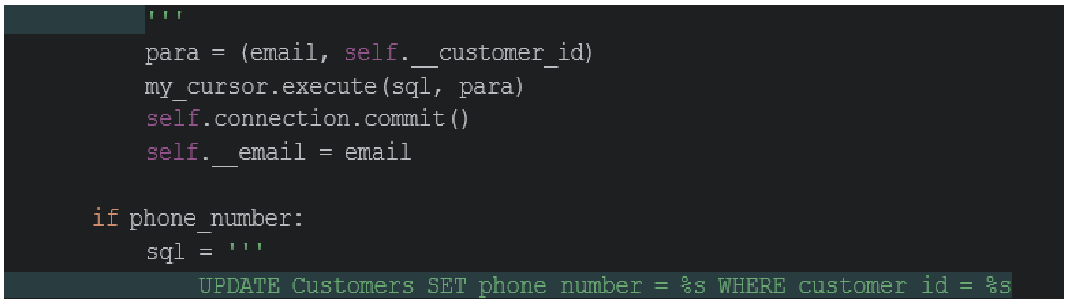
```python
from db_connection.db_adapter import *


class Customer:

    def __init__(self, customer_id, first_name, last_name, dob, email,
phone_number, address):
        self.connection = get_db_connection()
        self.__customer_id = customer_id
        self.__first_name = first_name
        self.__last_name = last_name
        self.__dob = dob
        self.__email = email
        self.__phone_number = phone_number
        self.__address = address

    def get_customer_id(self):
        return self.__customer_id

    def get_first_name(self):
        return self.__first_name

    def get_last_name(self):
        return self.__last_name

    def get_customer_email(self):
        return self.__email

    def get_phone_number(self):
        return self.__phone_number

    def get_customer_address(self):
        return self.__address

    def update_student_info(self, first_name=None, last_name=None,
date_of_birth=None, email=None, phone_number=None, address=None):
        my_cursor = self.connection.cursor()

        if first_name:
```



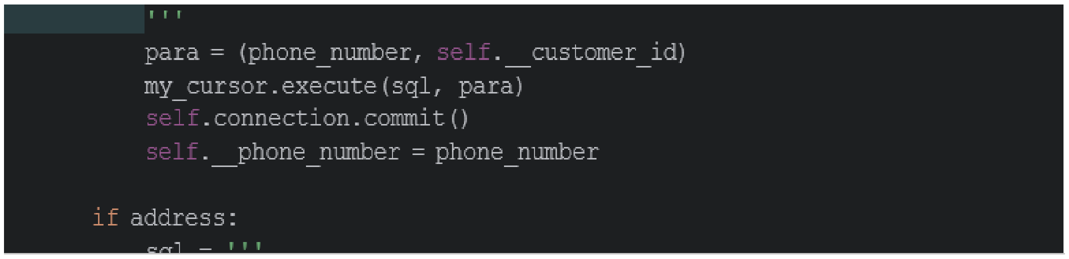
```python
        if first_name:
            sql = '''
                UPDATE Customers SET first_name = %s WHERE customer_id = %s
            '''
            para = (first_name, self.__customer_id)
            my_cursor.execute(sql, para)
            self.connection.commit()
            self.__first_name = first_name
```



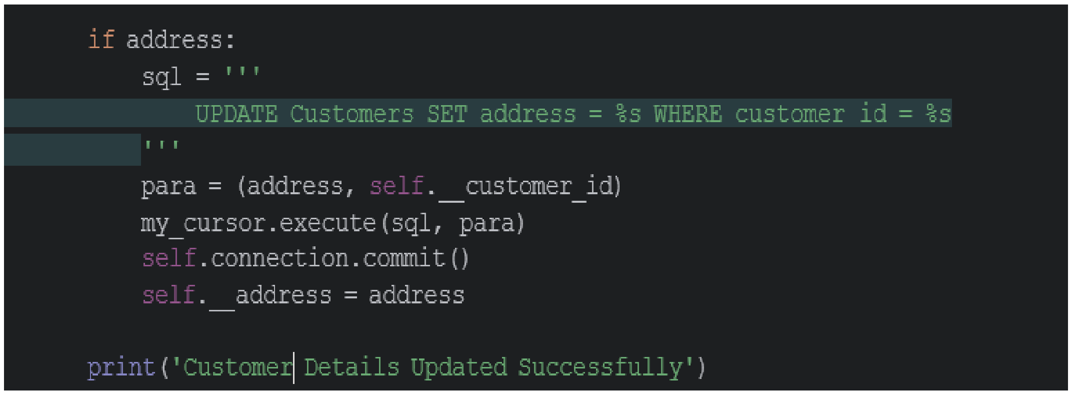
```
        if last_name:
            sql = '''
                UPDATE Customers SET last_name = %s WHERE customer_id = %s
            '''
            para = (last_name, self.__customer_id)
            my_cursor.execute(sql, para)
            self.connection.commit()
            self.__last_name = last_name

        if date_of_birth:
            sql = '''
                UPDATE Customers SET DOB = %s WHERE customer_id = %s
            '''
            para = (date_of_birth, self.__customer_id)
            my_cursor.execute(sql, para)
            self.connection.commit()
            self.__dob = date_of_birth

        if email:
            sql = '''
                UPDATE Customers SET email = %s WHERE customer_id = %s
```



```
            '''
            para = (email, self.__customer_id)
            my_cursor.execute(sql, para)
            self.connection.commit()
            self.__email = email

        if phone_number:
            sql = '''
                UPDATE Customers SET phone_number = %s WHERE customer_id = %s
```

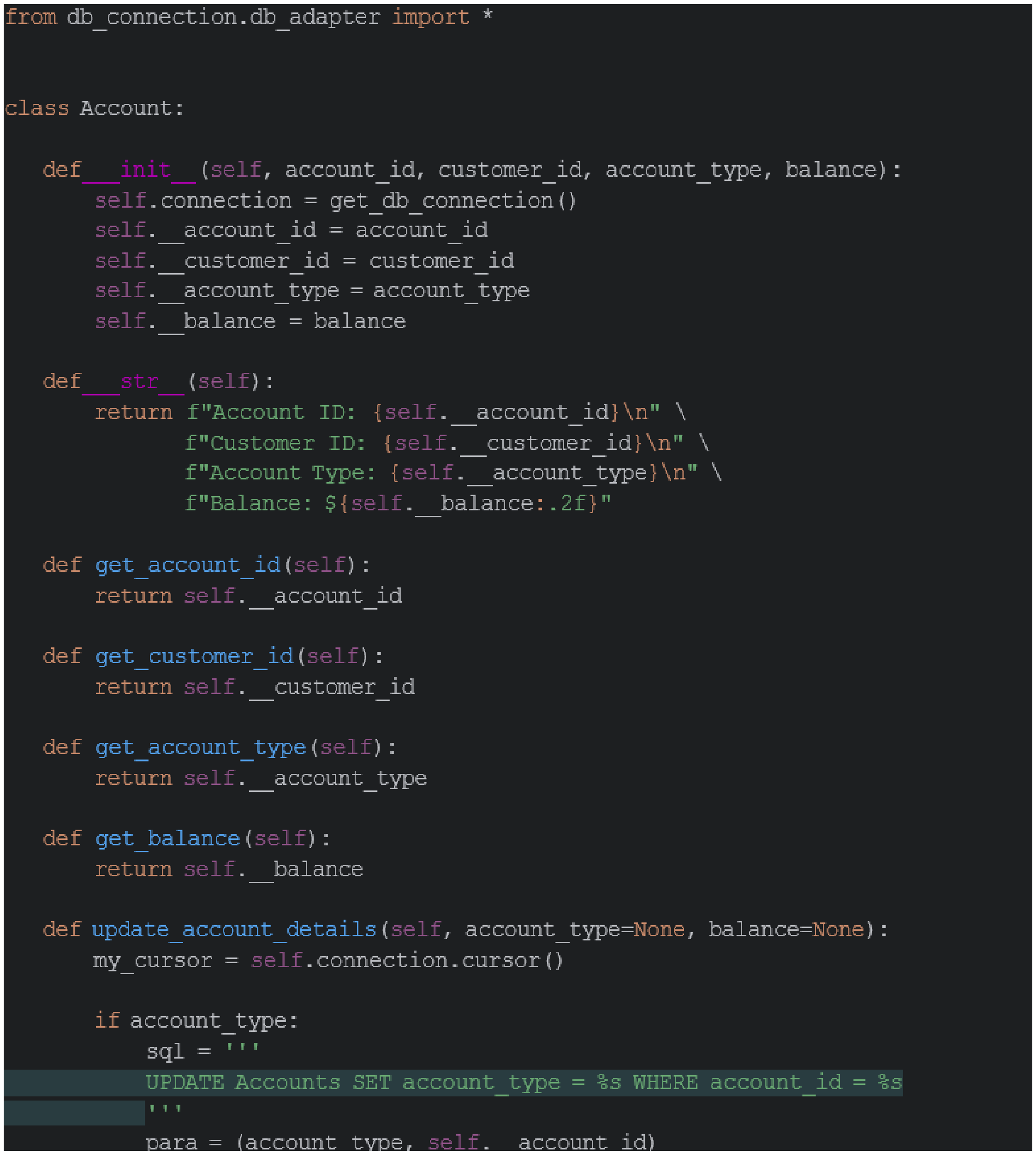


```
            '''
            para = (phone_number, self.__customer_id)
            my_cursor.execute(sql, para)
            self.connection.commit()
            self.__phone_number = phone_number

        if address:
            sql = '''
```

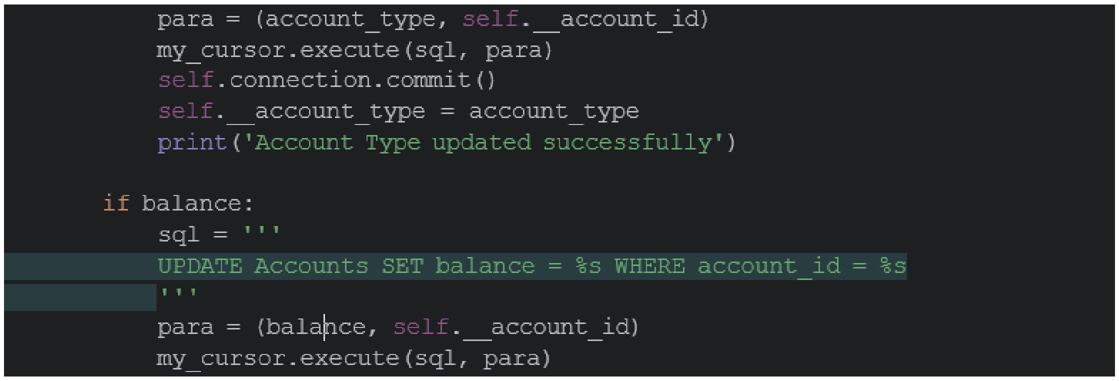


```
        if address:
            sql = '''
                UPDATE Customers SET address = %s WHERE customer_id = %s
            '''
            para = (address, self.__customer_id)
            my_cursor.execute(sql, para)
            self.connection.commit()
            self.__address = address

        print('Customer Details Updated Successfully')
```

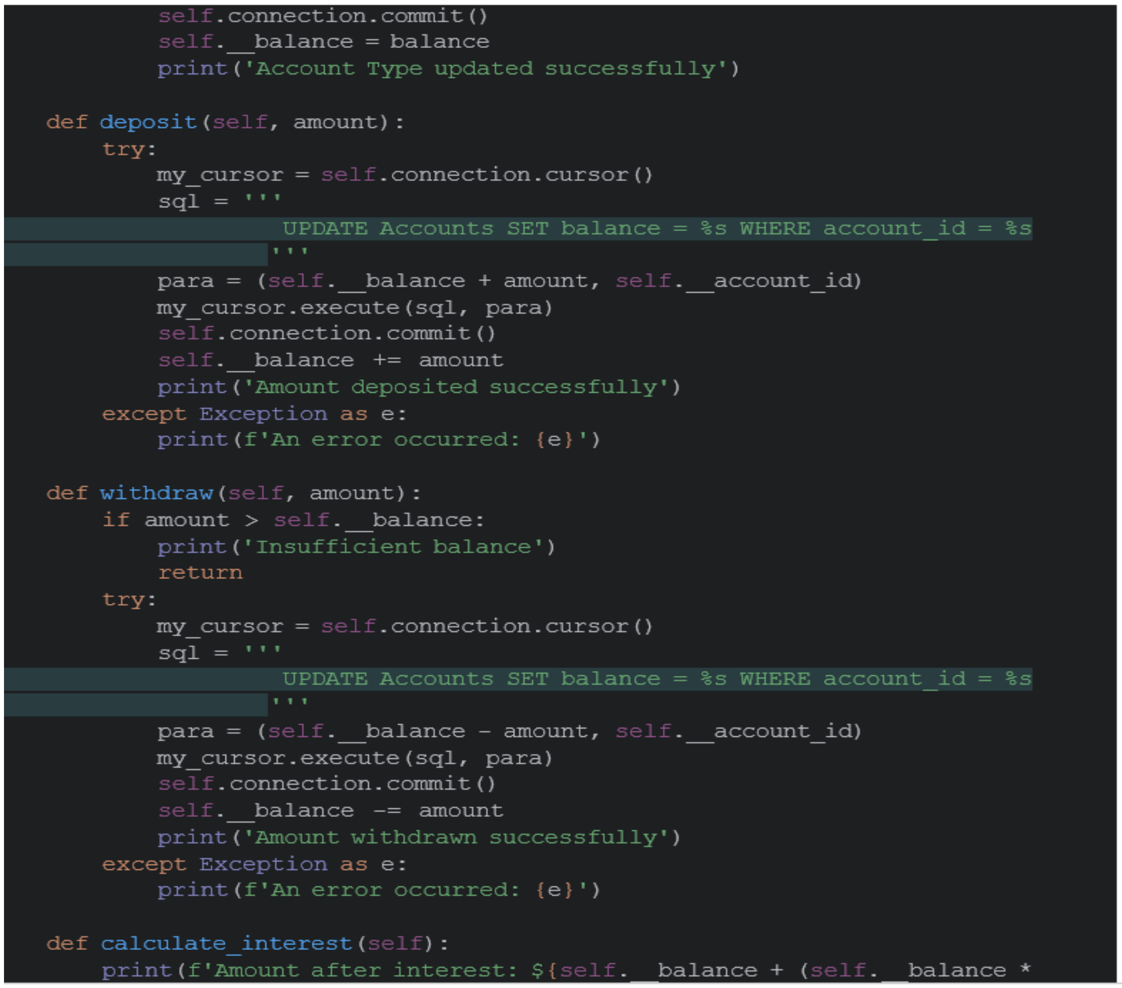
Account.py



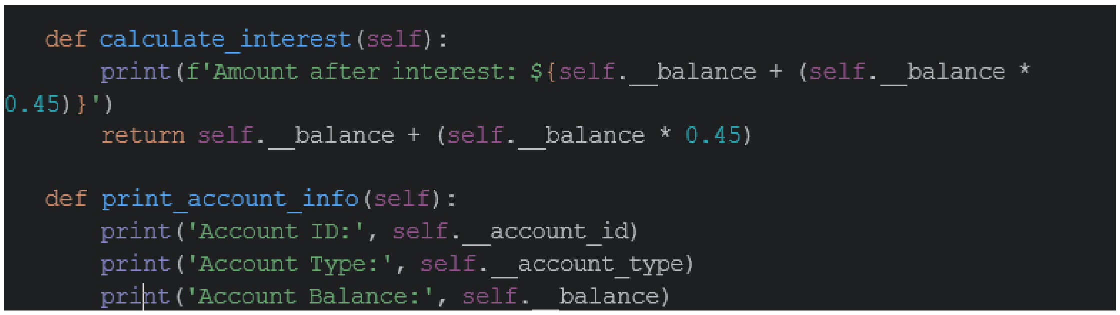
```python
from db_connection.db_adapter import *


class Account:

    def __init__(self, account_id, customer_id, account_type, balance):
        self.connection = get_db_connection()
        self.__account_id = account_id
        self.__customer_id = customer_id
        self.__account_type = account_type
        self.__balance = balance

    def __str__(self):
        return f"Account ID: {self.__account_id}\n" \
               f"Customer ID: {self.__customer_id}\n" \
               f"Account Type: {self.__account_type}\n" \
               f"Balance: ${self.__balance:.2f}"

    def get_account_id(self):
        return self.__account_id

    def get_customer_id(self):
        return self.__customer_id

    def get_account_type(self):
        return self.__account_type

    def get_balance(self):
        return self.__balance

    def update_account_details(self, account_type=None, balance=None):
        my_cursor = self.connection.cursor()

        if account_type:
            sql = '''
            UPDATE Accounts SET account_type = %s WHERE account_id = %s
            '''
            para = (account_type, self.__account_id)
```



```python
            para = (account_type, self.__account_id)
            my_cursor.execute(sql, para)
            self.connection.commit()
            self.__account_type = account_type
            print('Account Type updated successfully')

        if balance:
            sql = '''
            UPDATE Accounts SET balance = %s WHERE account_id = %s
            '''
            para = (balance, self.__account_id)
            my_cursor.execute(sql, para)
```

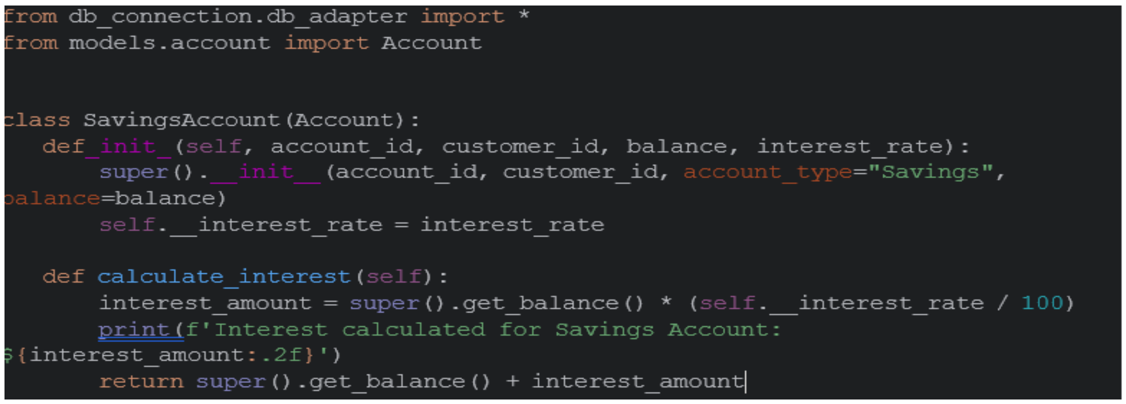


```
            self.connection.commit()
            self.__balance = balance
            print('Account Type updated successfully')

    def deposit(self, amount):
        try:
            my_cursor = self.connection.cursor()
            sql = '''
                      UPDATE Accounts SET balance = %s WHERE account_id = %s
                    '''
            para = (self.__balance + amount, self.__account_id)
            my_cursor.execute(sql, para)
            self.connection.commit()
            self.__balance += amount
            print('Amount deposited successfully')
        except Exception as e:
            print(f'An error occurred: {e}')

    def withdraw(self, amount):
        if amount > self.__balance:
            print('Insufficient balance')
            return
        try:
            my_cursor = self.connection.cursor()
            sql = '''
                      UPDATE Accounts SET balance = %s WHERE account_id = %s
                    '''
            para = (self.__balance - amount, self.__account_id)
            my_cursor.execute(sql, para)
            self.connection.commit()
            self.__balance -= amount
            print('Amount withdrawn successfully')
        except Exception as e:
            print(f'An error occurred: {e}')

    def calculate_interest(self):
        print(f'Amount after interest: ${self.__balance + (self.__balance *
```



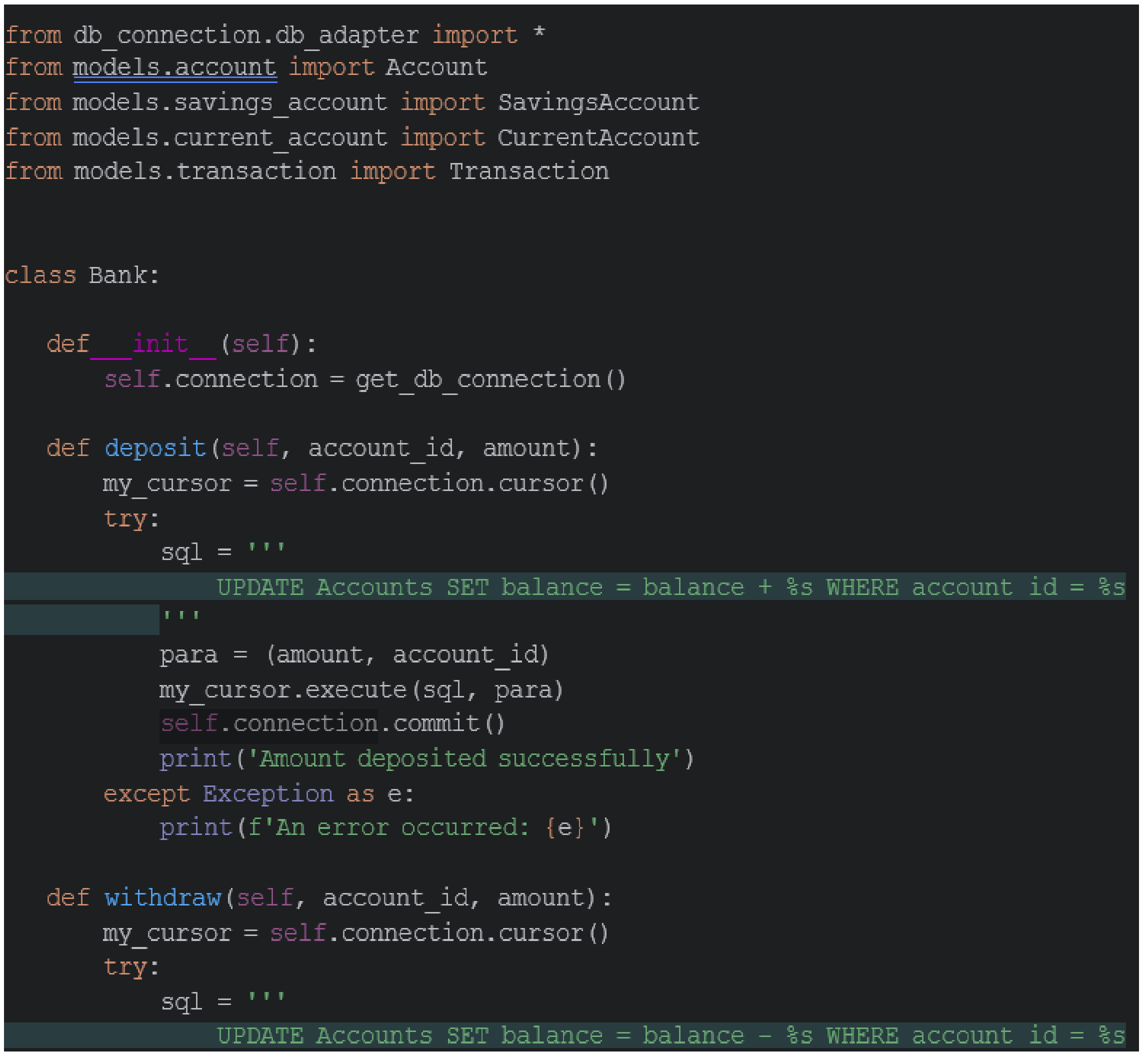
```
    def calculate_interest(self):
        print(f'Amount after interest: ${self.__balance + (self.__balance *
0.45)}')
        return self.__balance + (self.__balance * 0.45)

    def print_account_info(self):
        print('Account ID:', self.__account_id)
        print('Account Type:', self.__account_type)
        print('Account Balance:', self.__balance)
```

Savings_Account.py

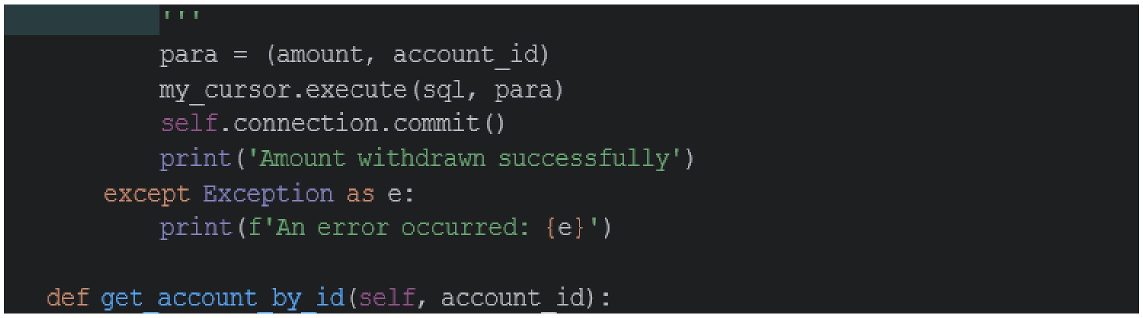


```
from db_connection.db_adapter import *
from models.account import Account


class SavingsAccount(Account):
    def __init__(self, account_id, customer_id, balance, interest_rate):
        super().__init__(account_id, customer_id, account_type="Savings",
balance=balance)
        self.__interest_rate = interest_rate

    def calculate_interest(self):
        interest_amount = super().get_balance() * (self.__interest_rate / 100)
        print(f'Interest calculated for Savings Account:
${interest_amount:.2f}')
        return super().get_balance() + interest_amount
```

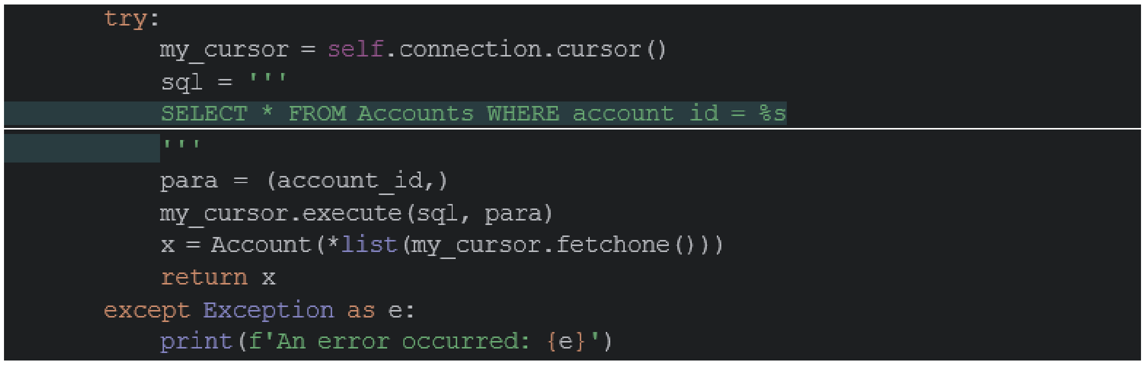
Bank.py



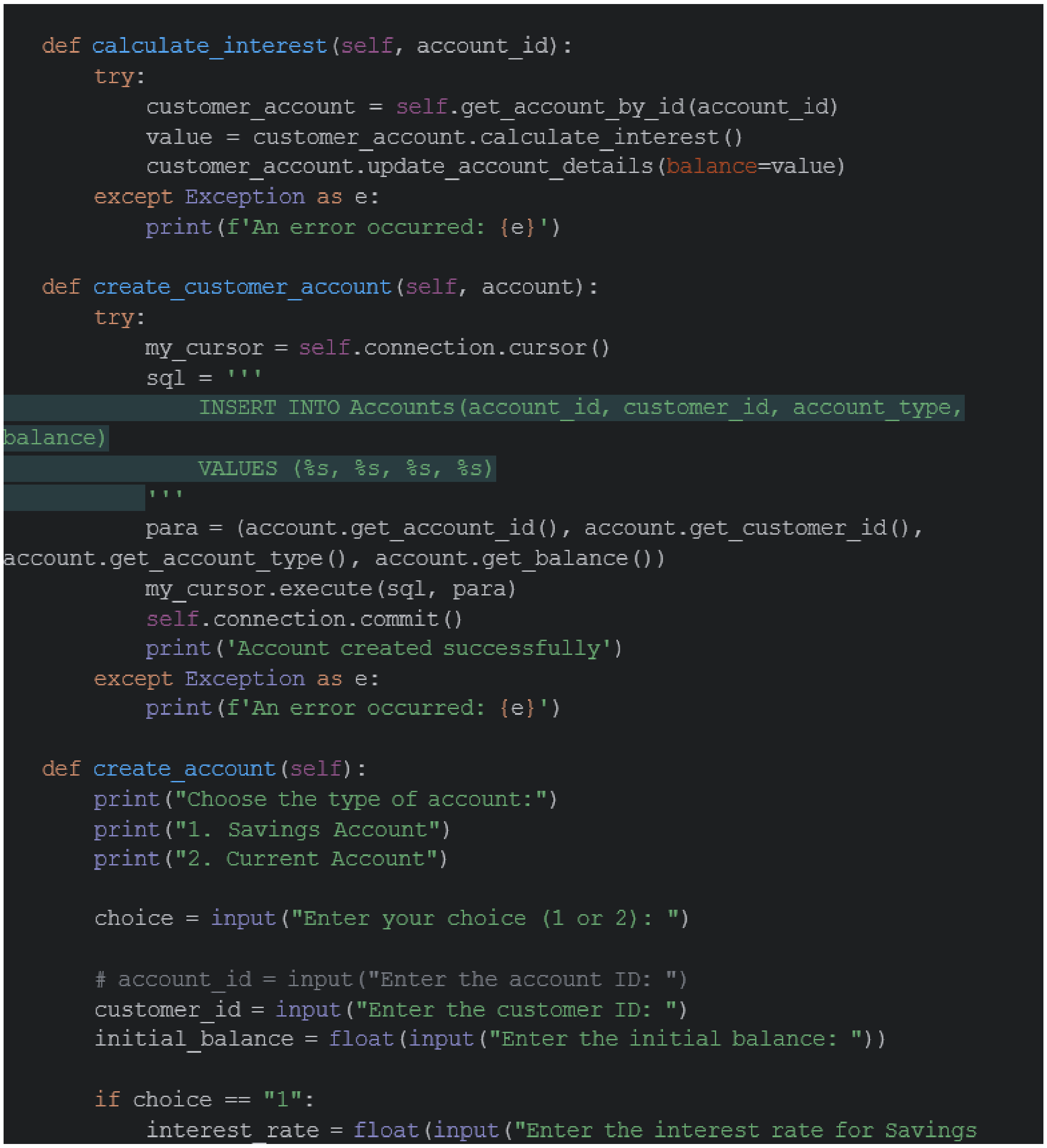
```python
from db_connection.db_adapter import *
from models.account import Account
from models.savings_account import SavingsAccount
from models.current_account import CurrentAccount
from models.transaction import Transaction


class Bank:

    def __init__(self):
        self.connection = get_db_connection()

    def deposit(self, account_id, amount):
        my_cursor = self.connection.cursor()
        try:
            sql = '''
                UPDATE Accounts SET balance = balance + %s WHERE account_id = %s
            '''
            para = (amount, account_id)
            my_cursor.execute(sql, para)
            self.connection.commit()
            print('Amount deposited successfully')
        except Exception as e:
            print(f'An error occurred: {e}')

    def withdraw(self, account_id, amount):
        my_cursor = self.connection.cursor()
        try:
            sql = '''
                UPDATE Accounts SET balance = balance - %s WHERE account_id = %s
```



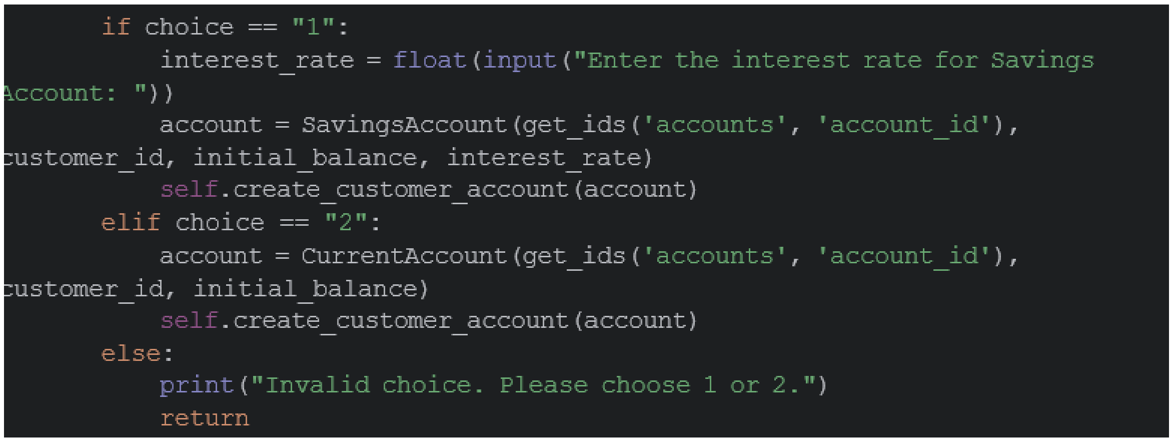
```python
            '''
            para = (amount, account_id)
            my_cursor.execute(sql, para)
            self.connection.commit()
            print('Amount withdrawn successfully')
        except Exception as e:
            print(f'An error occurred: {e}')

    def get_account_by_id(self, account_id):
```



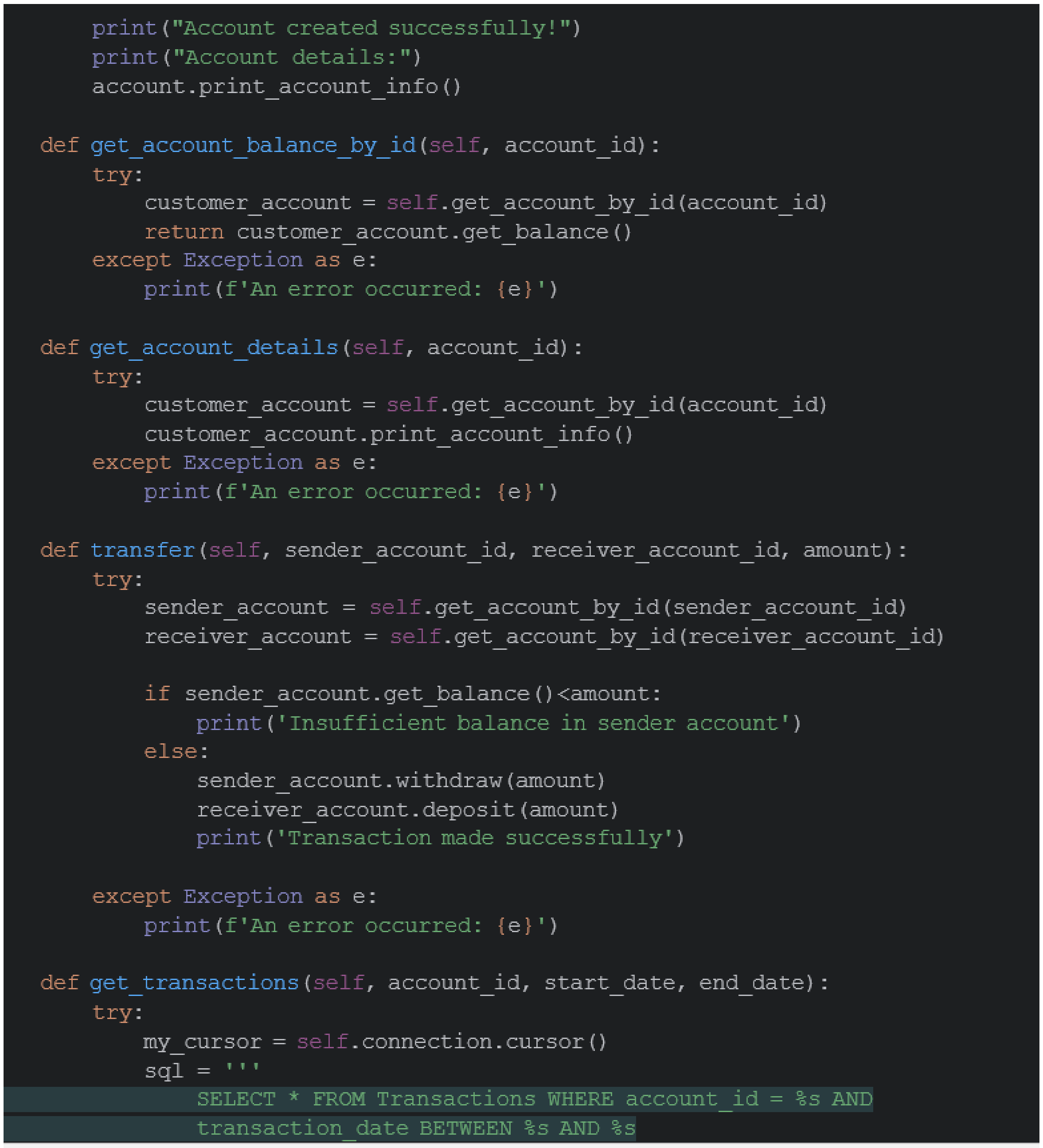
```python
        try:
            my_cursor = self.connection.cursor()
            sql = '''
            SELECT * FROM Accounts WHERE account_id = %s
            '''
            para = (account_id,)
            my_cursor.execute(sql, para)
            x = Account(*list(my_cursor.fetchone()))
            return x
        except Exception as e:
            print(f'An error occurred: {e}')
```



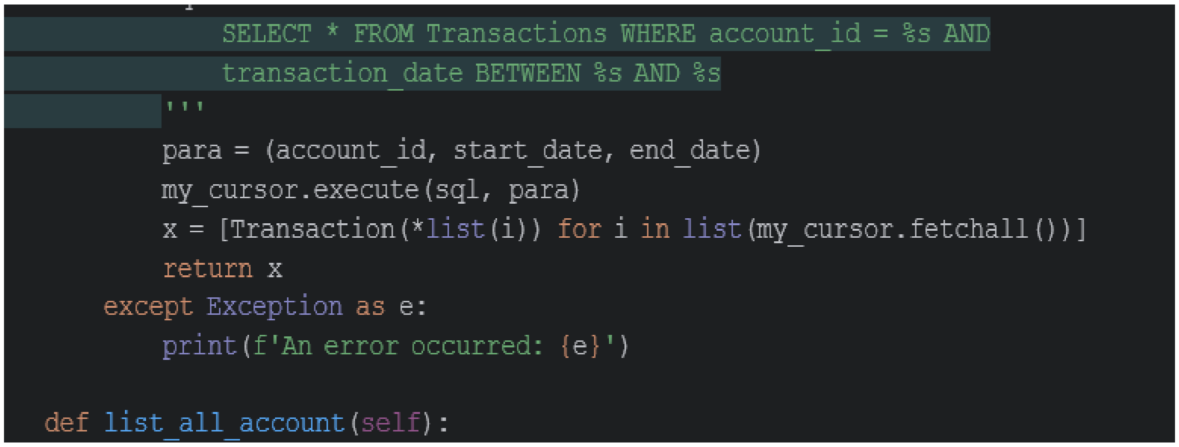
```
    def calculate_interest(self, account_id):
        try:
            customer_account = self.get_account_by_id(account_id)
            value = customer_account.calculate_interest()
            customer_account.update_account_details(balance=value)
        except Exception as e:
            print(f'An error occurred: {e}')

    def create_customer_account(self, account):
        try:
            my_cursor = self.connection.cursor()
            sql = '''
                INSERT INTO Accounts(account_id, customer_id, account_type,
balance)
                VALUES (%s, %s, %s, %s)
            '''
            para = (account.get_account_id(), account.get_customer_id(),
account.get_account_type(), account.get_balance())
            my_cursor.execute(sql, para)
            self.connection.commit()
            print('Account created successfully')
        except Exception as e:
            print(f'An error occurred: {e}')

    def create_account(self):
        print("Choose the type of account:")
        print("1. Savings Account")
        print("2. Current Account")

        choice = input("Enter your choice (1 or 2): ")

        # account_id = input("Enter the account ID: ")
        customer_id = input("Enter the customer ID: ")
        initial_balance = float(input("Enter the initial balance: "))

        if choice == "1":
            interest_rate = float(input("Enter the interest rate for Savings
```



```
        if choice == "1":
            interest_rate = float(input("Enter the interest rate for Savings
Account: "))
            account = SavingsAccount(get_ids('accounts', 'account_id'),
customer_id, initial_balance, interest_rate)
            self.create_customer_account(account)
        elif choice == "2":
            account = CurrentAccount(get_ids('accounts', 'account_id'),
customer_id, initial_balance)
            self.create_customer_account(account)
        else:
            print("Invalid choice. Please choose 1 or 2.")
            return
```

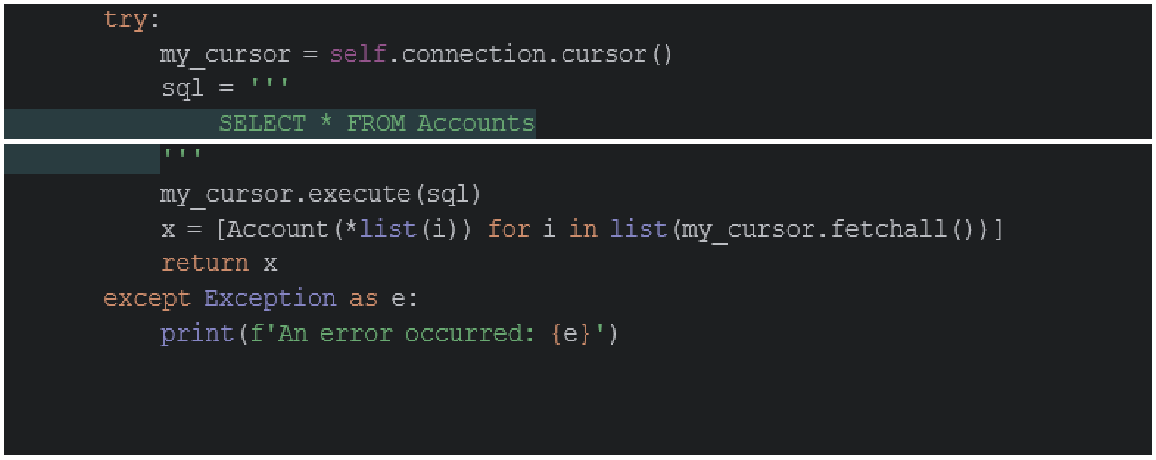


```python
        print("Account created successfully!")
        print("Account details:")
        account.print_account_info()

    def get_account_balance_by_id(self, account_id):
        try:
            customer_account = self.get_account_by_id(account_id)
            return customer_account.get_balance()
        except Exception as e:
            print(f'An error occurred: {e}')

    def get_account_details(self, account_id):
        try:
            customer_account = self.get_account_by_id(account_id)
            customer_account.print_account_info()
        except Exception as e:
            print(f'An error occurred: {e}')

    def transfer(self, sender_account_id, receiver_account_id, amount):
        try:
            sender_account = self.get_account_by_id(sender_account_id)
            receiver_account = self.get_account_by_id(receiver_account_id)

            if sender_account.get_balance()<amount:
                print('Insufficient balance in sender account')
            else:
                sender_account.withdraw(amount)
                receiver_account.deposit(amount)
                print('Transaction made successfully')

        except Exception as e:
            print(f'An error occurred: {e}')

    def get_transactions(self, account_id, start_date, end_date):
        try:
            my_cursor = self.connection.cursor()
            sql = '''
                SELECT * FROM Transactions WHERE account_id = %s AND
                transaction_date BETWEEN %s AND %s
```



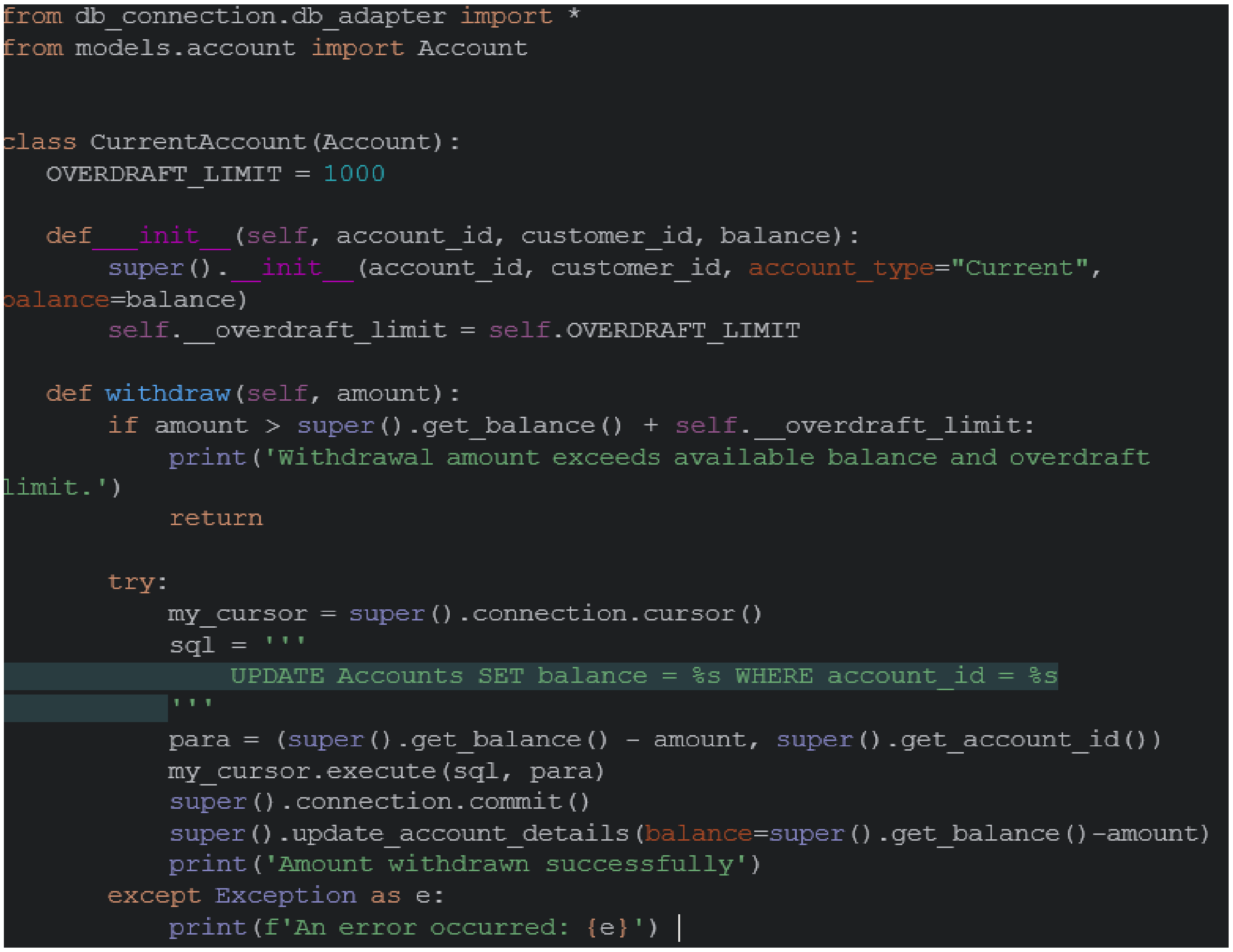
```python
                SELECT * FROM Transactions WHERE account_id = %s AND
                transaction_date BETWEEN %s AND %s
```

```python
            '''
            para = (account_id, start_date, end_date)
            my_cursor.execute(sql, para)
            x = [Transaction(*list(i)) for i in list(my_cursor.fetchall())]
            return x
        except Exception as e:
            print(f'An error occurred: {e}')

    def list_all_account(self):
```



```
        try:
            my_cursor = self.connection.cursor()
            sql = '''
                SELECT * FROM Accounts
            '''
            my_cursor.execute(sql)
            x = [Account(*list(i)) for i in list(my_cursor.fetchall())]
            return x
        except Exception as e:
            print(f'An error occurred: {e}')
```

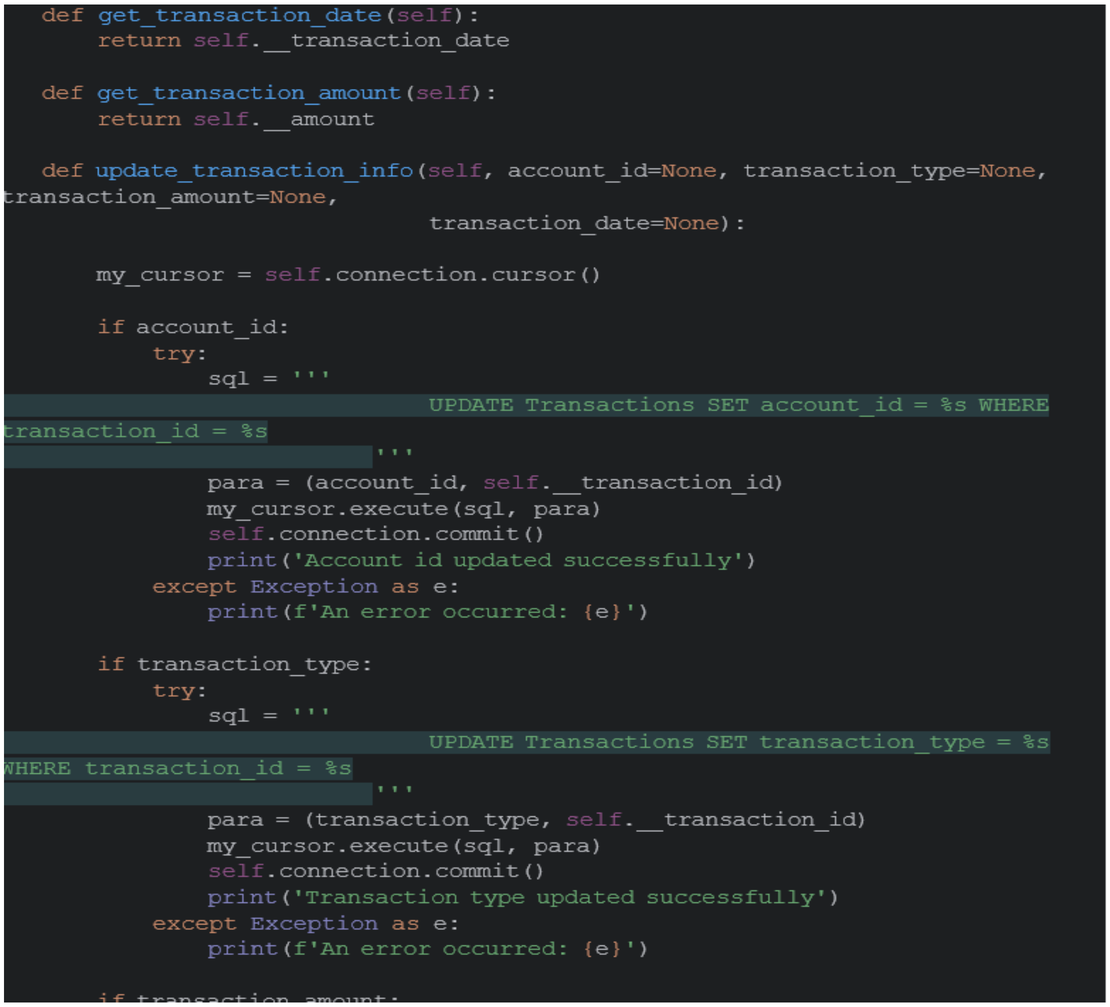
Current_account.py



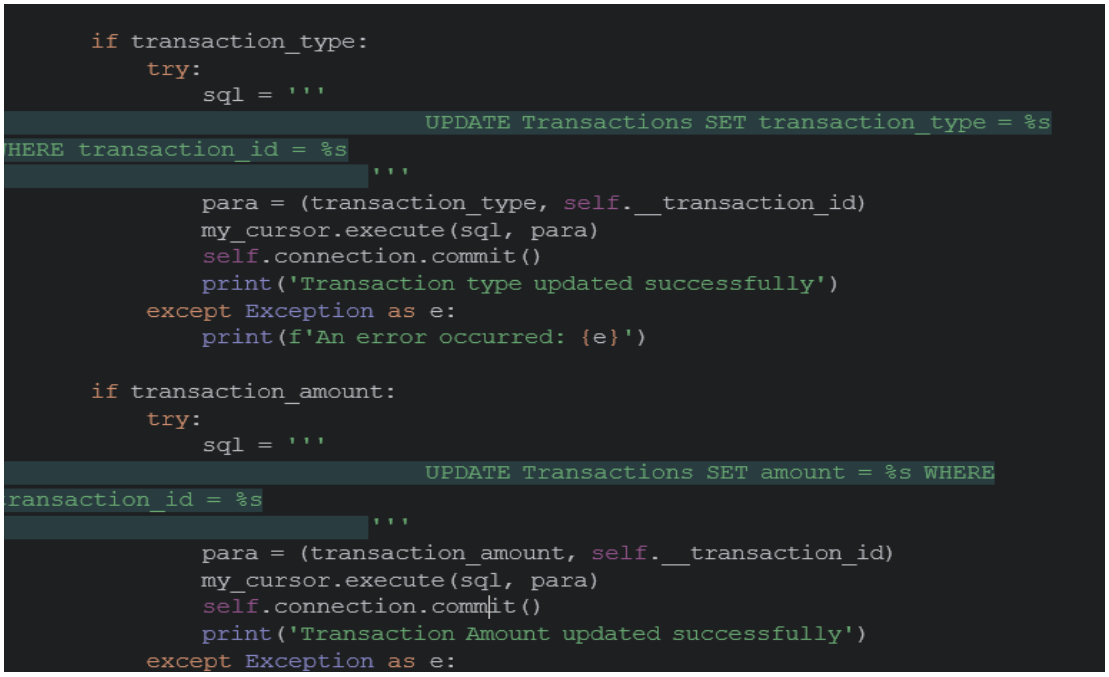
```
from db_connection.db_adapter import *
from models.account import Account


class CurrentAccount(Account):
    OVERDRAFT_LIMIT = 1000

    def __init__(self, account_id, customer_id, balance):
        super().__init__(account_id, customer_id, account_type="Current",
balance=balance)
        self.__overdraft_limit = self.OVERDRAFT_LIMIT

    def withdraw(self, amount):
        if amount > super().get_balance() + self.__overdraft_limit:
            print('Withdrawal amount exceeds available balance and overdraft
limit.')
            return

        try:
            my_cursor = super().connection.cursor()
            sql = '''
                UPDATE Accounts SET balance = %s WHERE account_id = %s
            '''
            para = (super().get_balance() - amount, super().get_account_id())
            my_cursor.execute(sql, para)
            super().connection.commit()
            super().update_account_details(balance=super().get_balance()-amount)
            print('Amount withdrawn successfully')
        except Exception as e:
            print(f'An error occurred: {e}')
```

Transaction.py

```python
from db_connection.db_adapter import *


class Transaction:

    def __init__(self, transaction_id, account_id, transaction_type, amount,
transaction_date):
        self.connection = get_db_connection()
        self.__transaction_id = transaction_id
        self.__account_id = account_id
        self.__transaction_type = transaction_type
        self.__amount = amount
        self.__transaction_date = transaction_date

    def __str__(self):
        return f"Transaction ID: {self.__transaction_id}\n" \
               f"Account ID: {self.__account_id}\n" \
               f"Transaction Type: {self.__transaction_type}\n" \
               f"Amount: ${self.__amount:.2f}\n" \
               f"Transaction Date: {self.__transaction_date}"

    def get_transaction_id(self):
        return self.__transaction_id

    def get_account_id(self):
        return self.__account_id

    def get_transaction_type(self):
        return self.__transaction_type
```



```python
    def get_transaction_date(self):
        return self.__transaction_date

    def get_transaction_amount(self):
        return self.__amount

    def update_transaction_info(self, account_id=None, transaction_type=None,
transaction_amount=None,
                                transaction_date=None):

        my_cursor = self.connection.cursor()

        if account_id:
            try:
                sql = '''
                                UPDATE Transactions SET account_id = %s WHERE
transaction_id = %s
                            '''
                para = (account_id, self.__transaction_id)
                my_cursor.execute(sql, para)
                self.connection.commit()
                print('Account id updated successfully')
            except Exception as e:
                print(f'An error occurred: {e}')

        if transaction_type:
            try:
                sql = '''
                                UPDATE Transactions SET transaction_type = %s
WHERE transaction_id = %s
                            '''
                para = (transaction_type, self.__transaction_id)
                my_cursor.execute(sql, para)
                self.connection.commit()
                print('Transaction type updated successfully')
            except Exception as e:
                print(f'An error occurred: {e}')

        if transaction_amount:
```



```python
        if transaction_type:
            try:
                sql = '''
                                UPDATE Transactions SET transaction_type = %s
HERE transaction_id = %s
                            '''
                para = (transaction_type, self.__transaction_id)
                my_cursor.execute(sql, para)
                self.connection.commit()
                print('Transaction type updated successfully')
            except Exception as e:
                print(f'An error occurred: {e}')

        if transaction_amount:
            try:
                sql = '''
                                UPDATE Transactions SET amount = %s WHERE
ransaction_id = %s
                            '''
                para = (transaction_amount, self.__transaction_id)
                my_cursor.execute(sql, para)
                self.connection.commit()
                print('Transaction Amount updated successfully')
            except Exception as e:
                print(f'An error occurred: {e}')

        if transaction_date:
            try:
                sql = '''
                                UPDATE Transactions SET transaction_date = %s
WHERE transaction_id = %s
                            '''
                para = (transaction_date, self.__transaction_id)
                my_cursor.execute(sql, para)
                self.connection.commit()
                print('Transaction date updated successfully')
            except Exception as e:
                print(f'An error occurred: {e}')
```

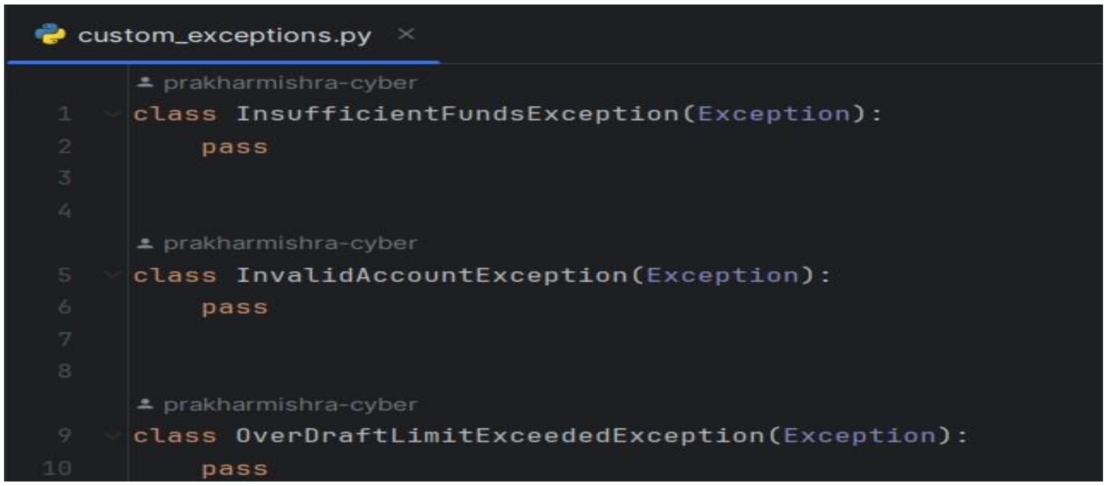
Zero_balance.py

```
from db_connection.db_adapter import *
from models.account import Account


class ZeroBalanceAccount(Account):

    def __init__(self, account_id, customer_id, account_type):
        self.connection = get_db_connection()
        super().__init__(account_id, customer_id, account_type, 0)
```

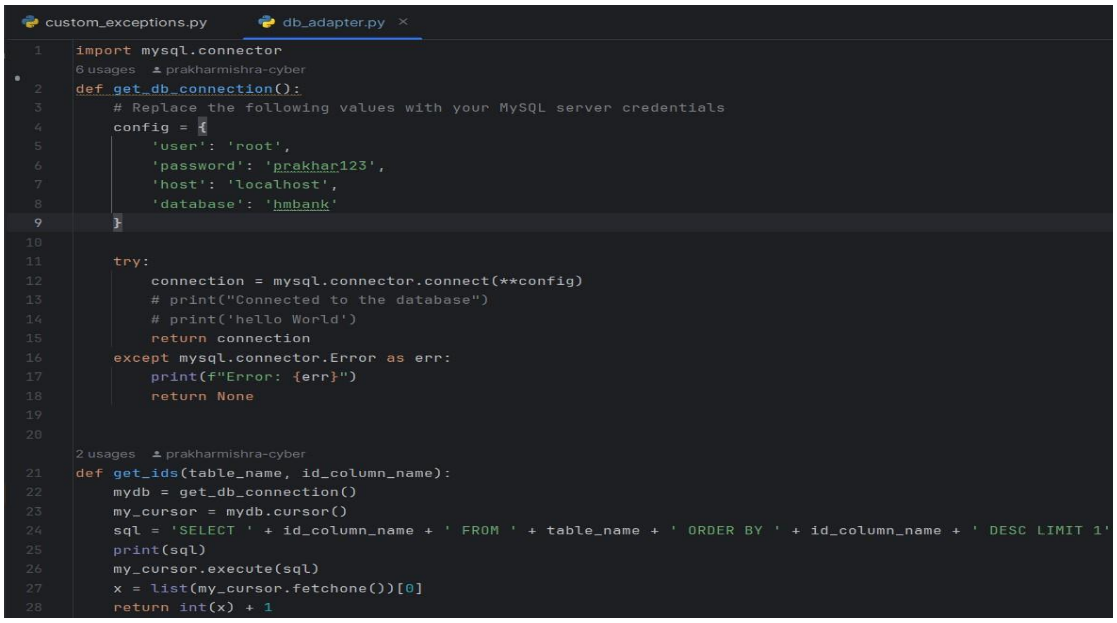
TASK: 14

In this we will make a "Database connector.py" separately as taught in our class, as it will increase our readability as well as reusability of the code at the same time will keep the codes separate and by importing it in the main file our job will be done.

DatabaseConnector.py



```
custom_exceptions.py

prakharmishra-cyber
class InsufficientFundsException(Exception):
    pass


prakharmishra-cyber
class InvalidAccountException(Exception):
    pass


prakharmishra-cyber
class OverDraftLimitExceededException(Exception):
    pass
```



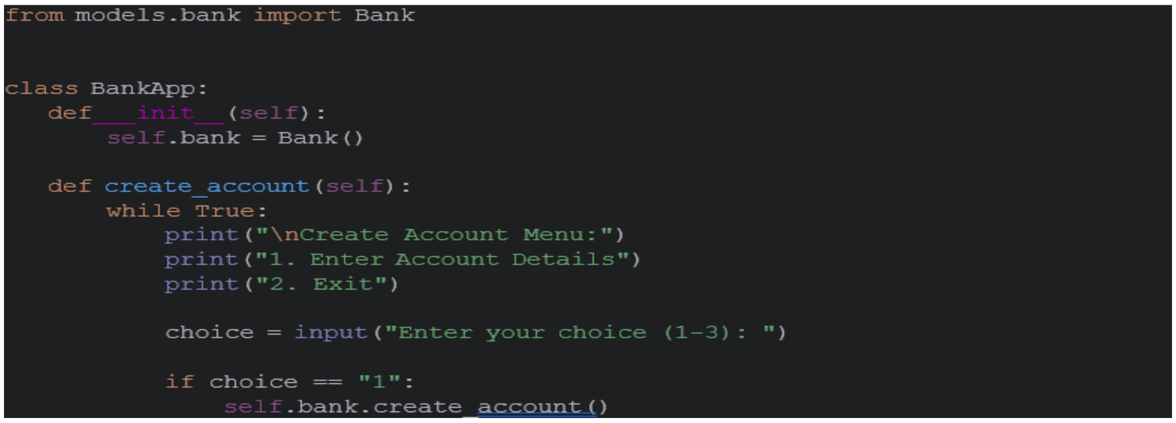
custom_exceptions.py   db_adapter.py

```python
import mysql.connector
6 usages  prakharmishra-cyber
def get_db_connection():
    # Replace the following values with your MySQL server credentials
    config = {
        'user': 'root',
        'password': 'prakhar123',
        'host': 'localhost',
        'database': 'hmbank'
    }

    try:
        connection = mysql.connector.connect(**config)
        # print("Connected to the database")
        # print('hello World')
        return connection
    except mysql.connector.Error as err:
        print(f"Error: {err}")
        return None


2 usages  prakharmishra-cyber
def get_ids(table_name, id_column_name):
    mydb = get_db_connection()
    my_cursor = mydb.cursor()
    sql = 'SELECT ' + id_column_name + ' FROM ' + table_name + ' ORDER BY ' + id_column_name + ' DESC LIMIT 1'
    print(sql)
    my_cursor.execute(sql)
    x = list(my_cursor.fetchone())[0]
    return int(x) + 1
```

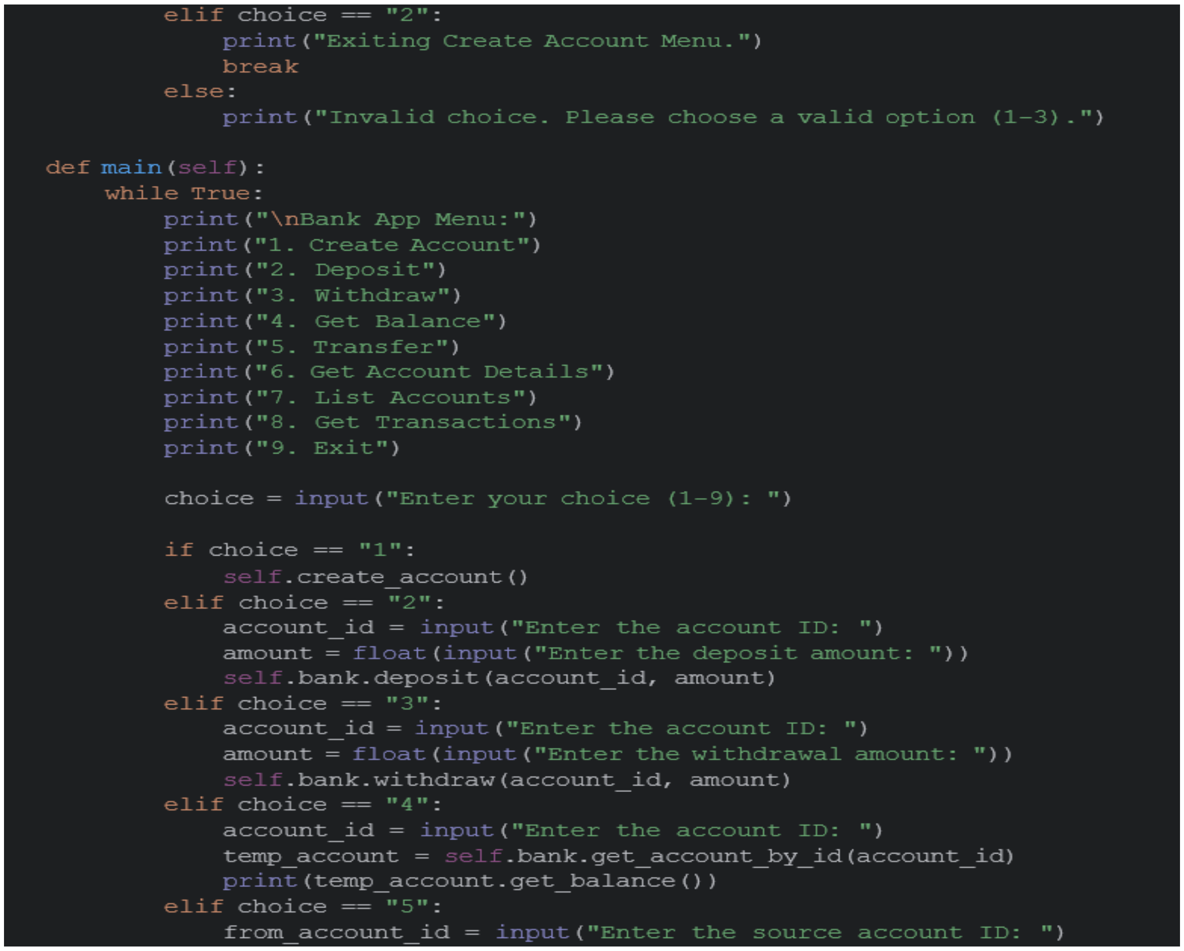
For database connectivity: python -m pip install mysql-connector-python on cmd

And in Pycharm<Settings<Project<Interpreter<pip< mysql-connector-python(install) and then it is ready to "import mysql.connector"

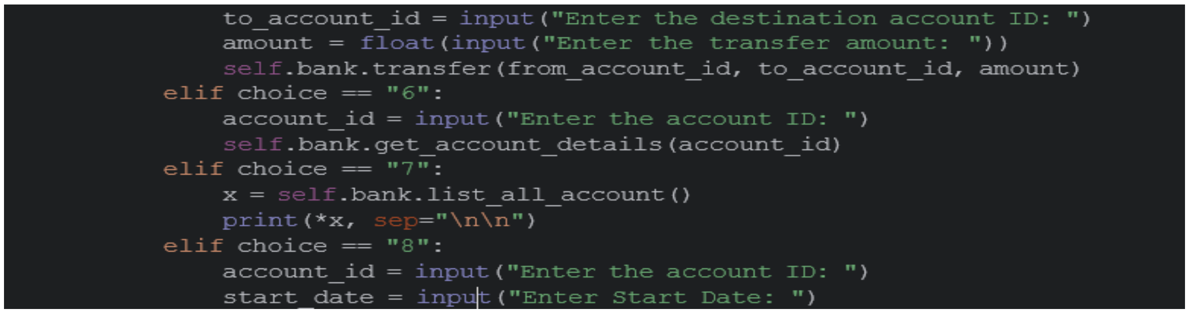
Main



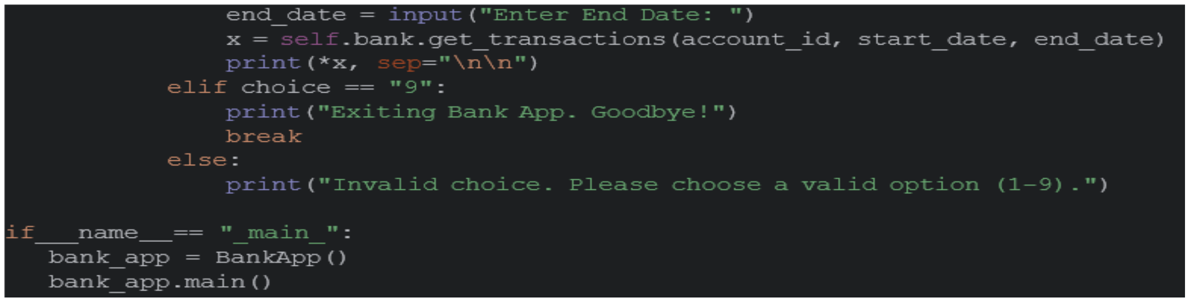
```python
from models.bank import Bank


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

   def create_account(self):
       while True:
           print("\nCreate Account Menu:")
           print("1. Enter Account Details")
           print("2. Exit")

           choice = input("Enter your choice (1-3): ")

           if choice == "1":
               self.bank.create_account()
```



```python
            elif choice == "2":
                print("Exiting Create Account Menu.")
                break
            else:
                print("Invalid choice. Please choose a valid option (1-3).")

    def main(self):
        while True:
            print("\nBank App 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")

            choice = input("Enter your choice (1-9): ")

            if choice == "1":
                self.create_account()
            elif choice == "2":
                account_id = input("Enter the account ID: ")
                amount = float(input("Enter the deposit amount: "))
                self.bank.deposit(account_id, amount)
            elif choice == "3":
                account_id = input("Enter the account ID: ")
                amount = float(input("Enter the withdrawal amount: "))
                self.bank.withdraw(account_id, amount)
            elif choice == "4":
                account_id = input("Enter the account ID: ")
                temp_account = self.bank.get_account_by_id(account_id)
                print(temp_account.get_balance())
            elif choice == "5":
                from_account_id = input("Enter the source account ID: ")
```

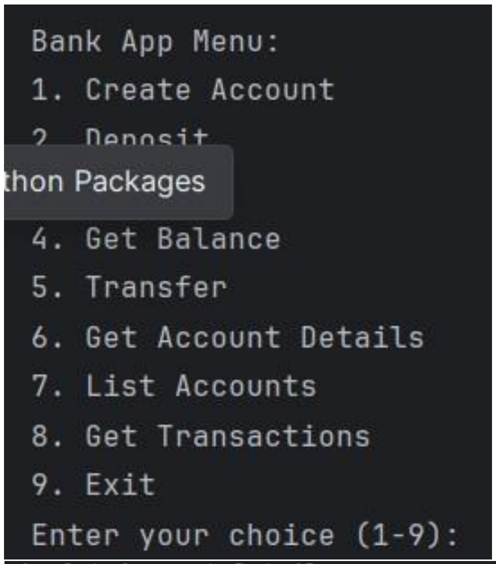


```python
                to_account_id = input("Enter the destination account ID: ")
                amount = float(input("Enter the transfer amount: "))
                self.bank.transfer(from_account_id, to_account_id, amount)
            elif choice == "6":
                account_id = input("Enter the account ID: ")
                self.bank.get_account_details(account_id)
            elif choice == "7":
                x = self.bank.list_all_account()
                print(*x, sep="\n\n")
            elif choice == "8":
                account_id = input("Enter the account ID: ")
                start_date = input("Enter Start Date: ")
```

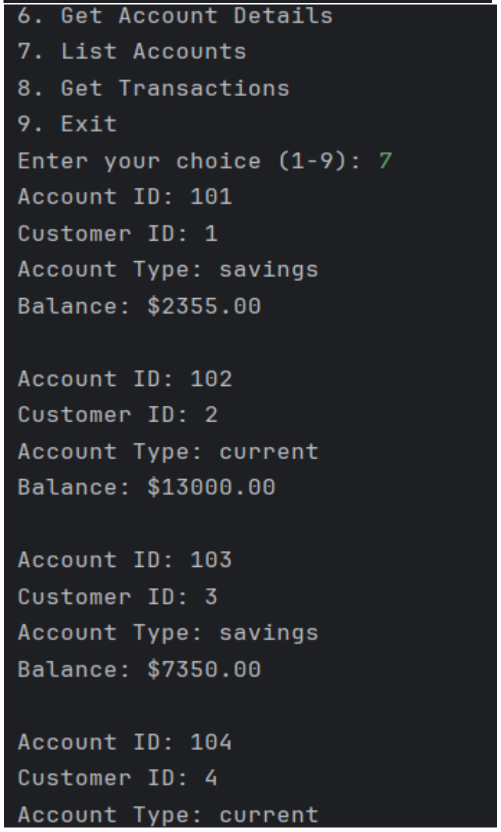


```python
                end_date = input("Enter End Date: ")
                x = self.bank.get_transactions(account_id, start_date, end_date)
                print(*x, sep="\n\n")
            elif choice == "9":
                print("Exiting Bank App. Goodbye!")
                break
            else:
                print("Invalid choice. Please choose a valid option (1-9).")

if __name__ == "__main__":
    bank_app = BankApp()
    bank_app.main()
```

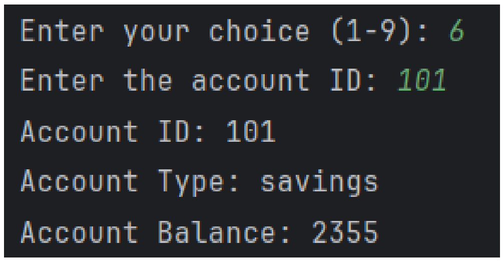
Output-



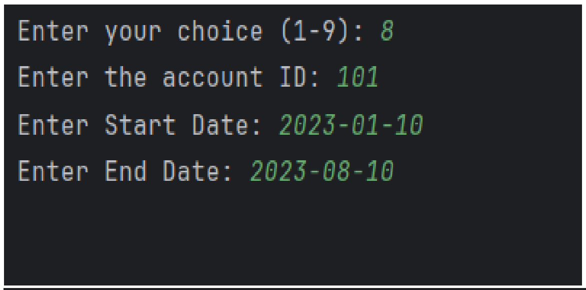
```
Bank App Menu:
1. Create Account
2. Deposit
thon Packages
4. Get Balance
5. Transfer
6. Get Account Details
7. List Accounts
8. Get Transactions
9. Exit
Enter your choice (1-9):
```



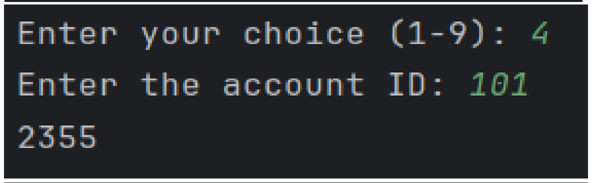
```
6. Get Account Details
7. List Accounts
8. Get Transactions
9. Exit
Enter your choice (1-9): 7
Account ID: 101
Customer ID: 1
Account Type: savings
Balance: $2355.00

Account ID: 102
Customer ID: 2
Account Type: current
Balance: $13000.00

Account ID: 103
Customer ID: 3
Account Type: savings
Balance: $7350.00

Account ID: 104
Customer ID: 4
Account Type: current
```



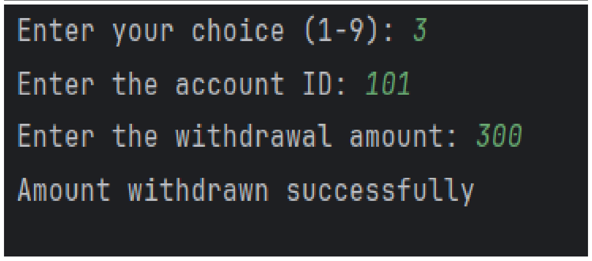
```
Enter your choice (1-9): 6
Enter the account ID: 101
Account ID: 101
Account Type: savings
Account Balance: 2355
```



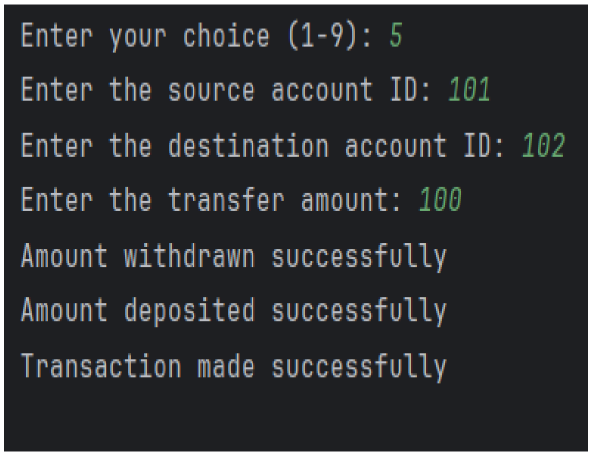
```
Enter your choice (1-9): 8
Enter the account ID: 101
Enter Start Date: 2023-01-10
Enter End Date: 2023-08-10
```



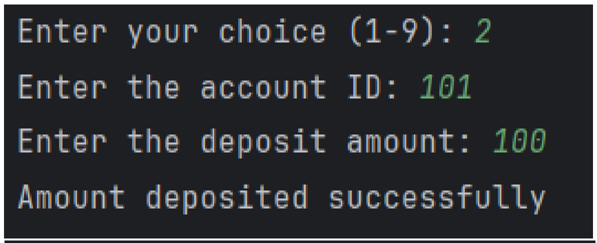
```
Enter your choice (1-9): 4
Enter the account ID: 101
2355
```



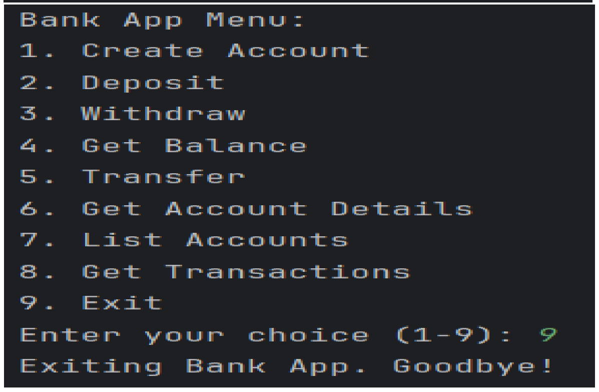
```
Enter your choice (1-9): 3
Enter the account ID: 101
Enter the withdrawal amount: 300
Amount withdrawn successfully
```



```
Enter your choice (1-9): 5
Enter the source account ID: 101
Enter the destination account ID: 102
Enter the transfer amount: 100
Amount withdrawn successfully
Amount deposited successfully
Transaction made successfully
```



```
Enter your choice (1-9): 2
Enter the account ID: 101
Enter the deposit amount: 100
Amount deposited successfully
```



```
Bank App 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 (1-9): 9
Exiting Bank App. Goodbye!
```