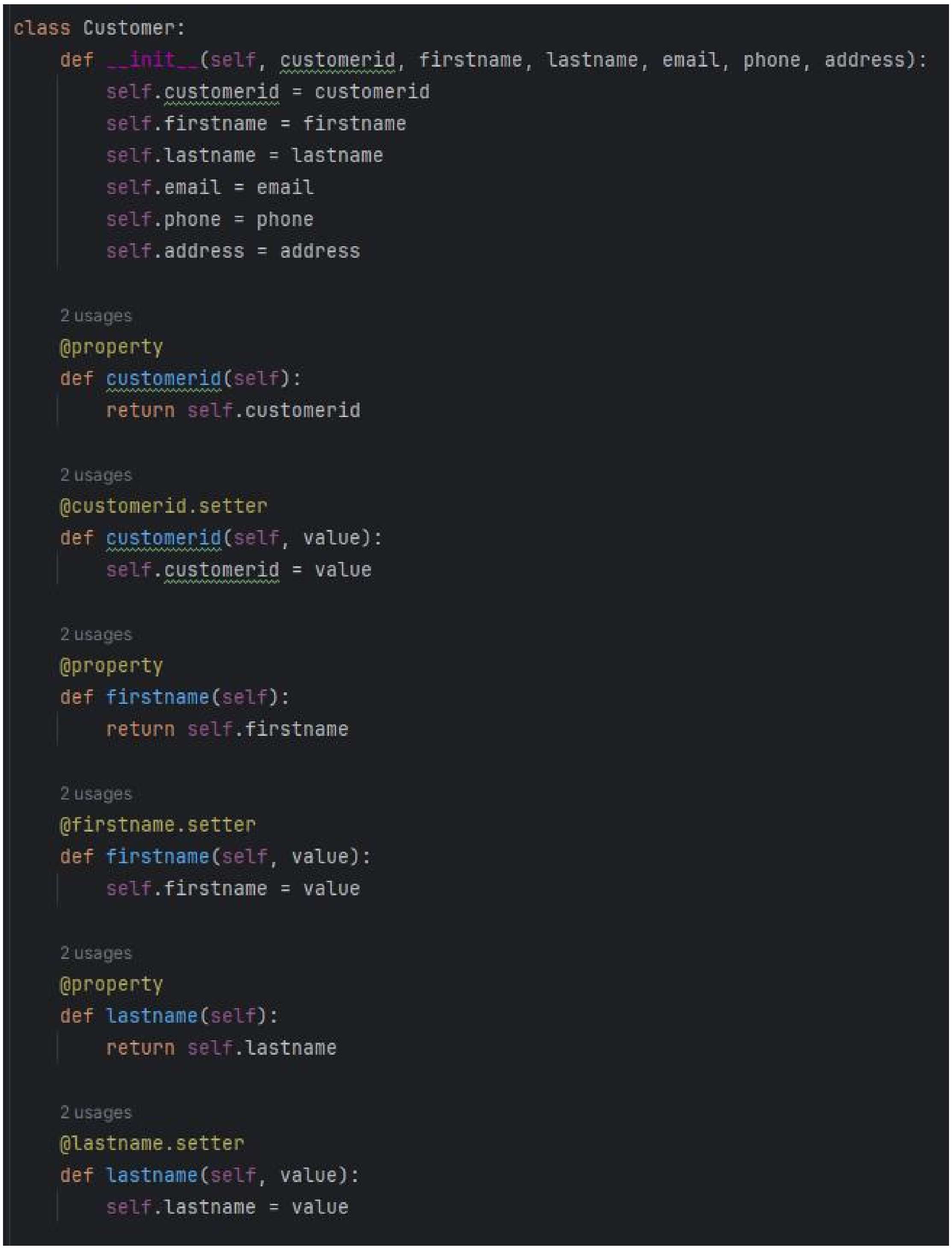
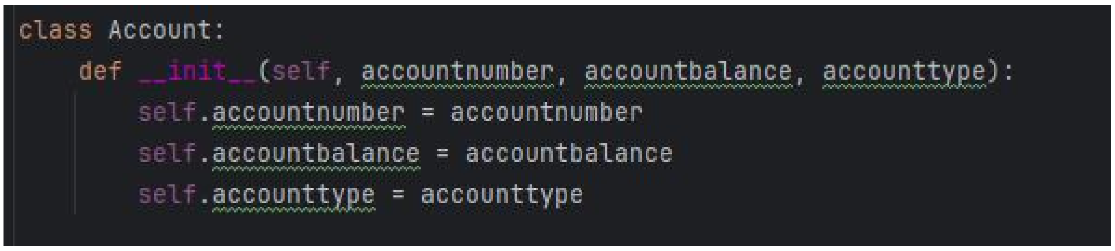
Create a `Customer` class with the following confidential attributes:
• Attributes
o Customer ID
o First Name
o Last Name
o Email Address
o Phone Number
o Address
• Constructor and Methods
o Implement default constructors and overload the constructor with Customer attributes, generate getter and setter, (print all information of attribute) methods for
the attributes.

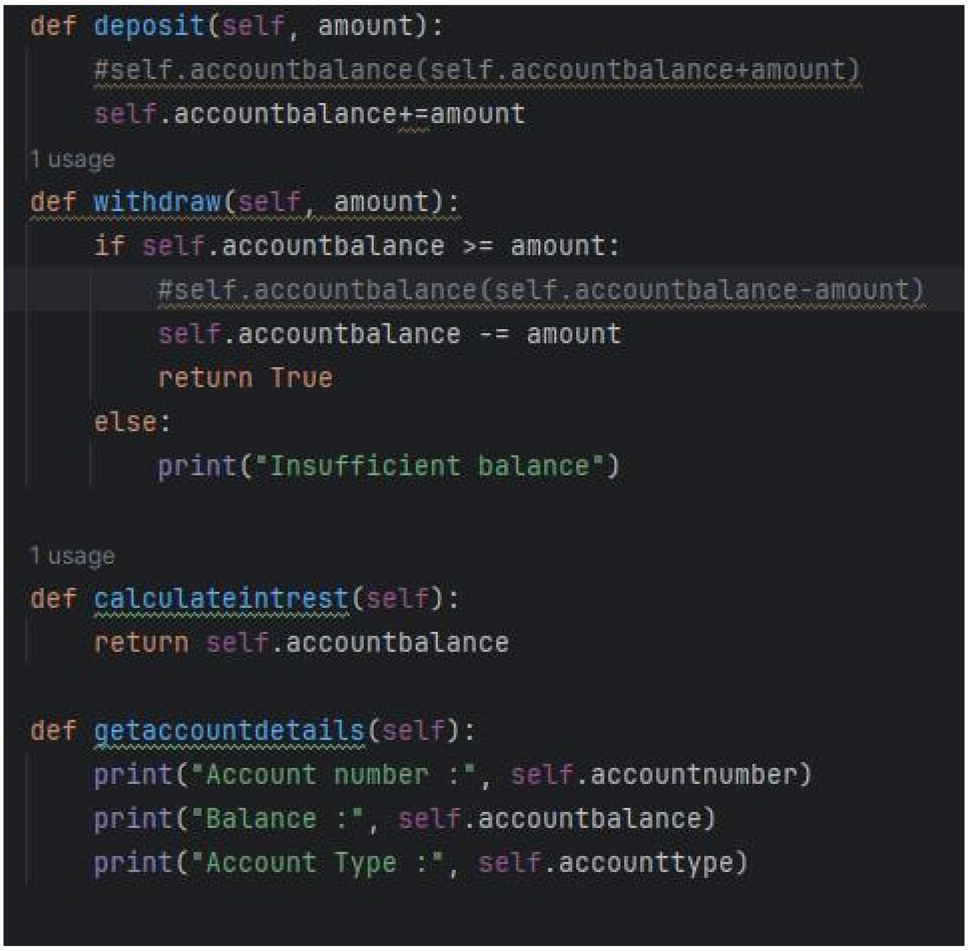

```
class Customer:
    def __init__(self, customerid, firstname, lastname, email, phone, address):
        self.customerid = customerid
        self.firstname = firstname
        self.lastname = lastname
        self.email = email
        self.phone = phone
        self.address = address

    2 usages
    @property
    def customerid(self):
        return self.customerid

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

    2 usages
    @property
    def firstname(self):
        return self.firstname

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

    2 usages
    @property
    def lastname(self):
        return self.lastname

    2 usages
    @lastname.setter
    def lastname(self, value):
        self.lastname = value
```

Create an `Account` class with the following confidential attributes:
• Attributes
o Account Number
o Account Type (e.g., Savings, Current)
o Account Balance



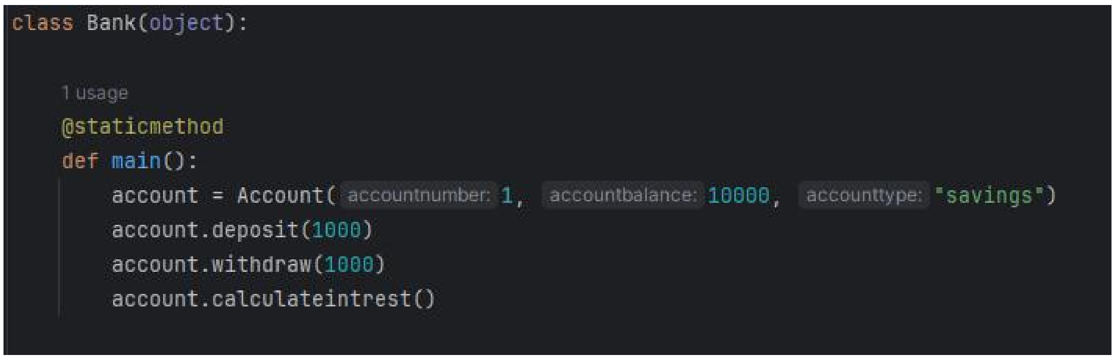
```
class Account:
    def __init__(self, accountnumber, accountbalance, accounttype):
        self.accountnumber = accountnumber
        self.accountbalance = accountbalance
        self.accounttype = accounttype
```

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%



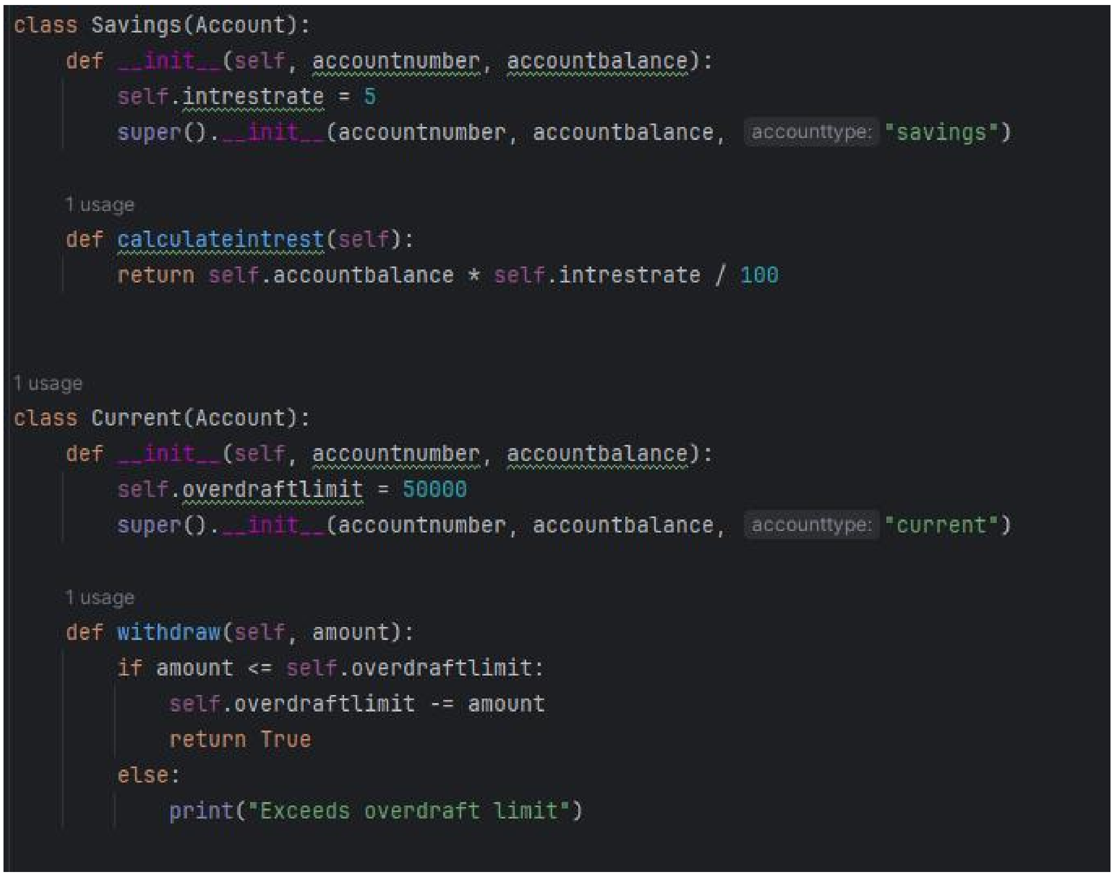
```
def deposit(self, amount):
    #self.accountbalance(self.accountbalance+amount)
    self.accountbalance+=amount

1 usage
def withdraw(self, amount):
    if self.accountbalance >= amount:
        #self.accountbalance(self.accountbalance-amount)
        self.accountbalance -= amount
        return True
    else:
        print("Insufficient balance")

1 usage
def calculateintrest(self):
    return self.accountbalance

def getaccountdetails(self):
    print("Account number :", self.accountnumber)
    print("Balance :", self.accountbalance)
    print("Account Type :", self.accounttype)
```

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.



```
class Bank(object):

    1 usage
    @staticmethod
    def main():
        account = Account( accountnumber: 1,  accountbalance: 10000,  accounttype: "savings")
        account.deposit(1000)
        account.withdraw(1000)
        account.calculateintrest()
```

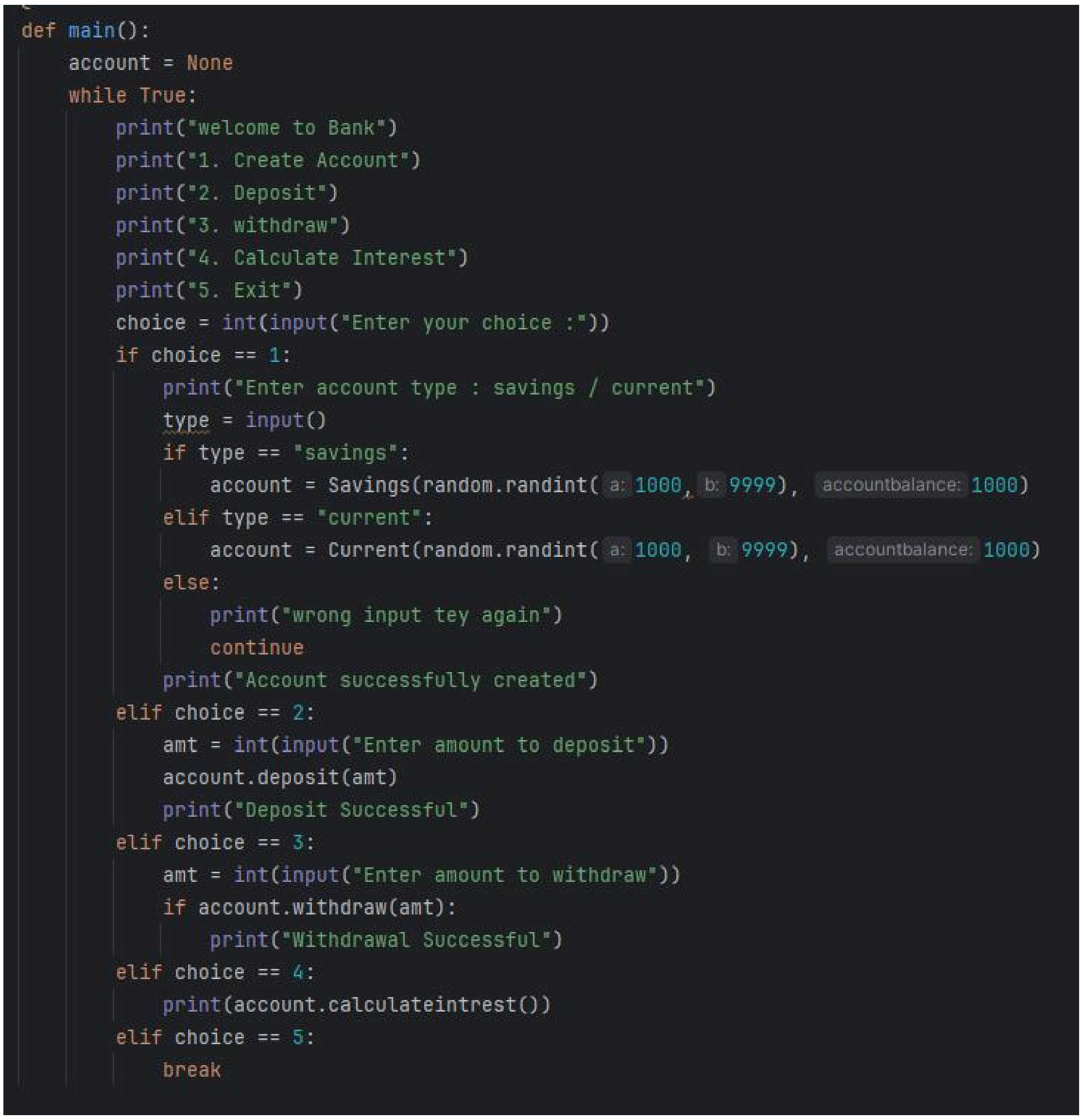
Create Subclasses for Specific Account Types
• Create subclasses for specific account types (e.g., `SavingsAccount`, `CurrentAccount`)
that inherit from the `Account` class.
o **SavingsAccount**: A savings account that includes an additional attribute for interest rate. **override** the calculate_interest()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



```
class Savings(Account):
    def __init__(self, accountnumber, accountbalance):
        self.intrestrate = 5
        super().__init__(accountnumber, accountbalance,  accounttype: "savings")

    1 usage
    def calculateintrest(self):
        return self.accountbalance * self.intrestrate / 100


1 usage
class Current(Account):
    def __init__(self, accountnumber, accountbalance):
        self.overdraftlimit = 50000
        super().__init__(accountnumber, accountbalance,  accounttype: "current")

    1 usage
    def withdraw(self, amount):
        if amount <= self.overdraftlimit:
            self.overdraftlimit -= amount
            return True
        else:
            print("Exceeds overdraft limit")
```

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

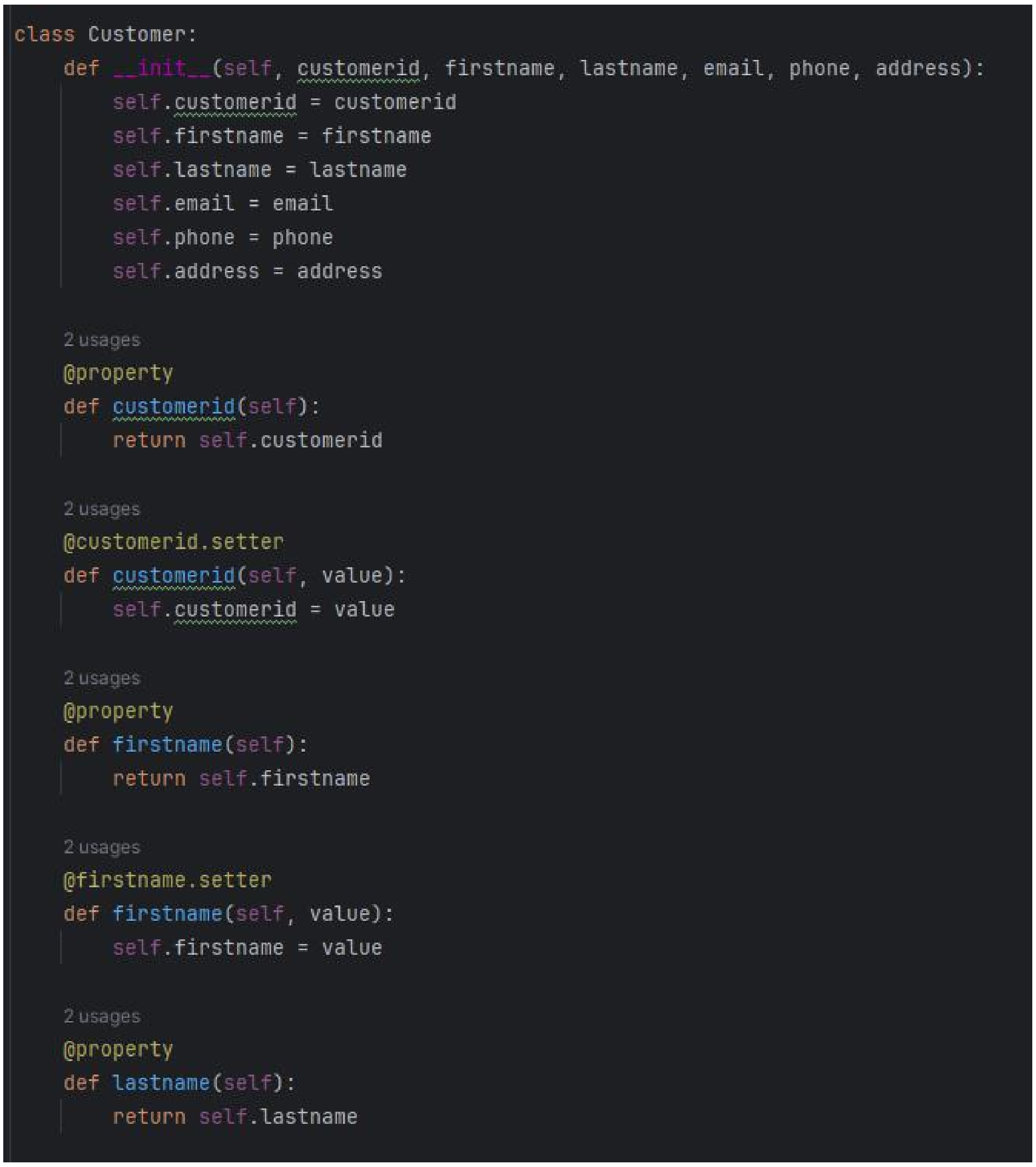
account withdraw amount only if there is sufficient fund else display
insufficient balance.
For Current Account withdraw limit can exceed the available balance and
should not
exceed the overdraft limit.
• **calculate_interest():** Calculate and add interest to the account balance for
savings
accounts.



```python
def main():
    account = None
    while True:
        print("welcome to Bank")
        print("1. Create Account")
        print("2. Deposit")
        print("3. withdraw")
        print("4. Calculate Interest")
        print("5. Exit")
        choice = int(input("Enter your choice :"))
        if choice == 1:
            print("Enter account type : savings / current")
            type = input()
            if type == "savings":
                account = Savings(random.randint(1000, 9999), 1000)
            elif type == "current":
                account = Current(random.randint(1000, 9999), 1000)
            else:
                print("wrong input tey again")
                continue
            print("Account successfully created")
        elif choice == 2:
            amt = int(input("Enter amount to deposit"))
            account.deposit(amt)
            print("Deposit Successful")
        elif choice == 3:
            amt = int(input("Enter amount to withdraw"))
            if account.withdraw(amt):
                print("Withdrawal Successful")
        elif choice == 4:
            print(account.calculateintrest())
        elif choice == 5:
            break
```

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

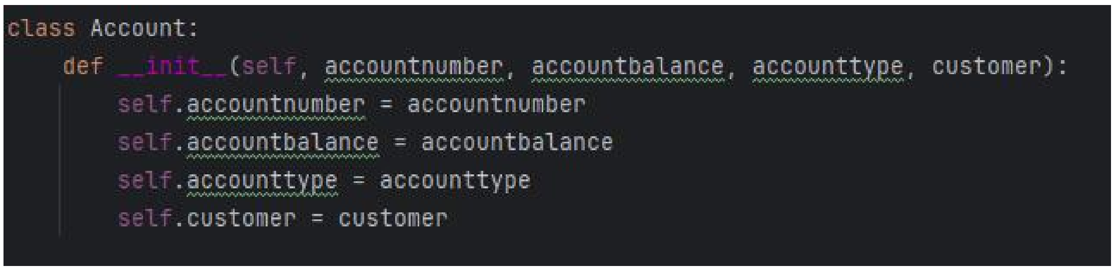
• Methods and Constructor:
o Implement default constructors and overload the constructor with Account attributes, generate getter, setter, print all information of attribute) methods for the attributes



```python
class Customer:
    def __init__(self, customerid, firstname, lastname, email, phone, address):
        self.customerid = customerid
        self.firstname = firstname
        self.lastname = lastname
        self.email = email
        self.phone = phone
        self.address = address

    2 usages
    @property
    def customerid(self):
        return self.customerid

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

    2 usages
    @property
    def firstname(self):
        return self.firstname

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

    2 usages
    @property
    def lastname(self):
        return self.lastname
```

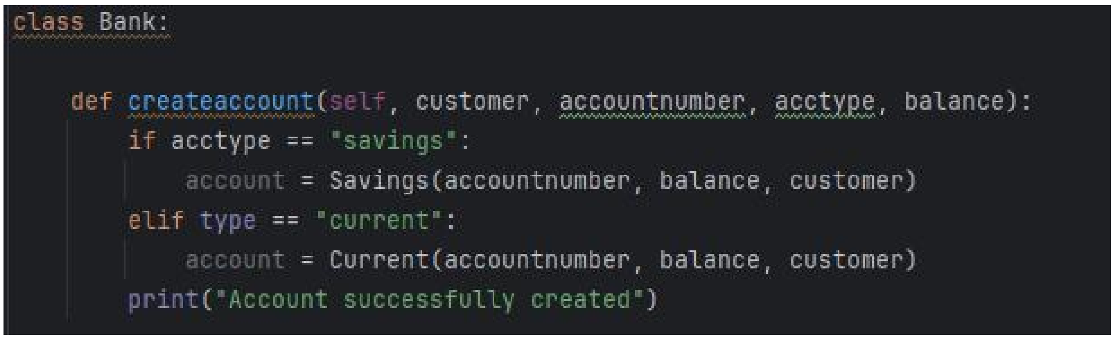
Create an `Account` class with the following attributes:
• Account Number (a unique identifier).
• Account Type (e.g., Savings, Current)
• Account Balance
• Customer (the customer who owns the account)
• Methods and Constructor:
o Implement default constructors and overload the constructor with Account

attributes, generate getter, setter, (print all information of attribute) methods for
the attributes

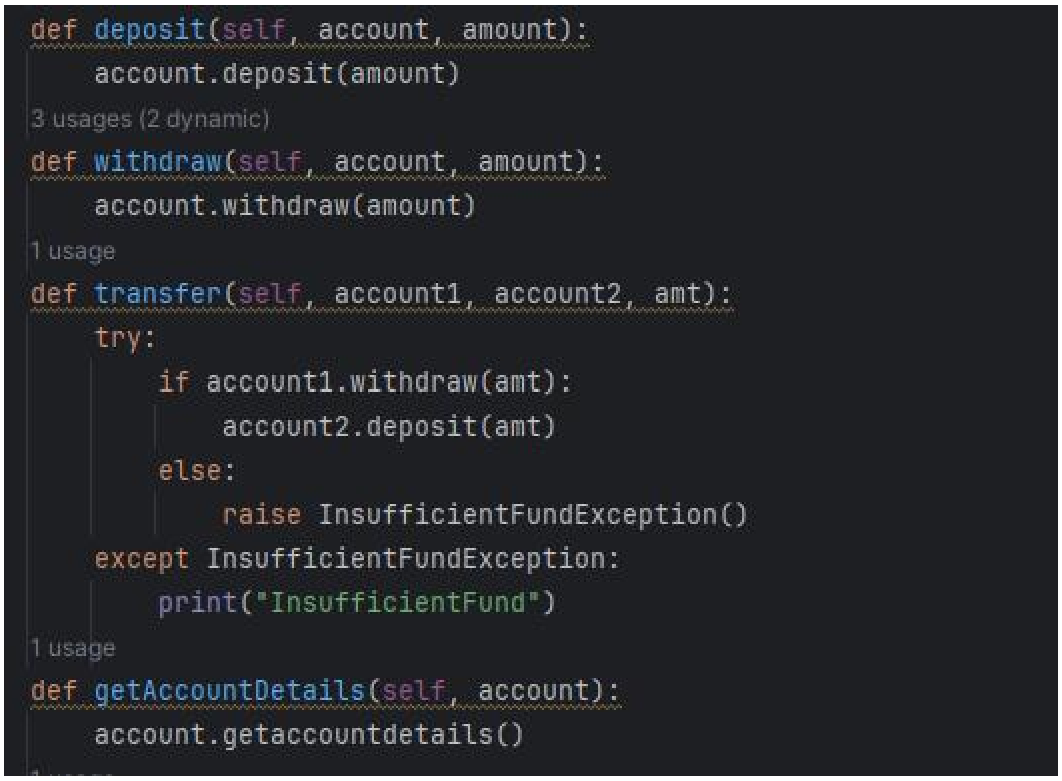


```
class Account:
    def __init__(self, accountnumber, accountbalance, accounttype, customer):
        self.accountnumber = accountnumber
        self.accountbalance = accountbalance
        self.accounttype = accounttype
        self.customer = customer
```

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.

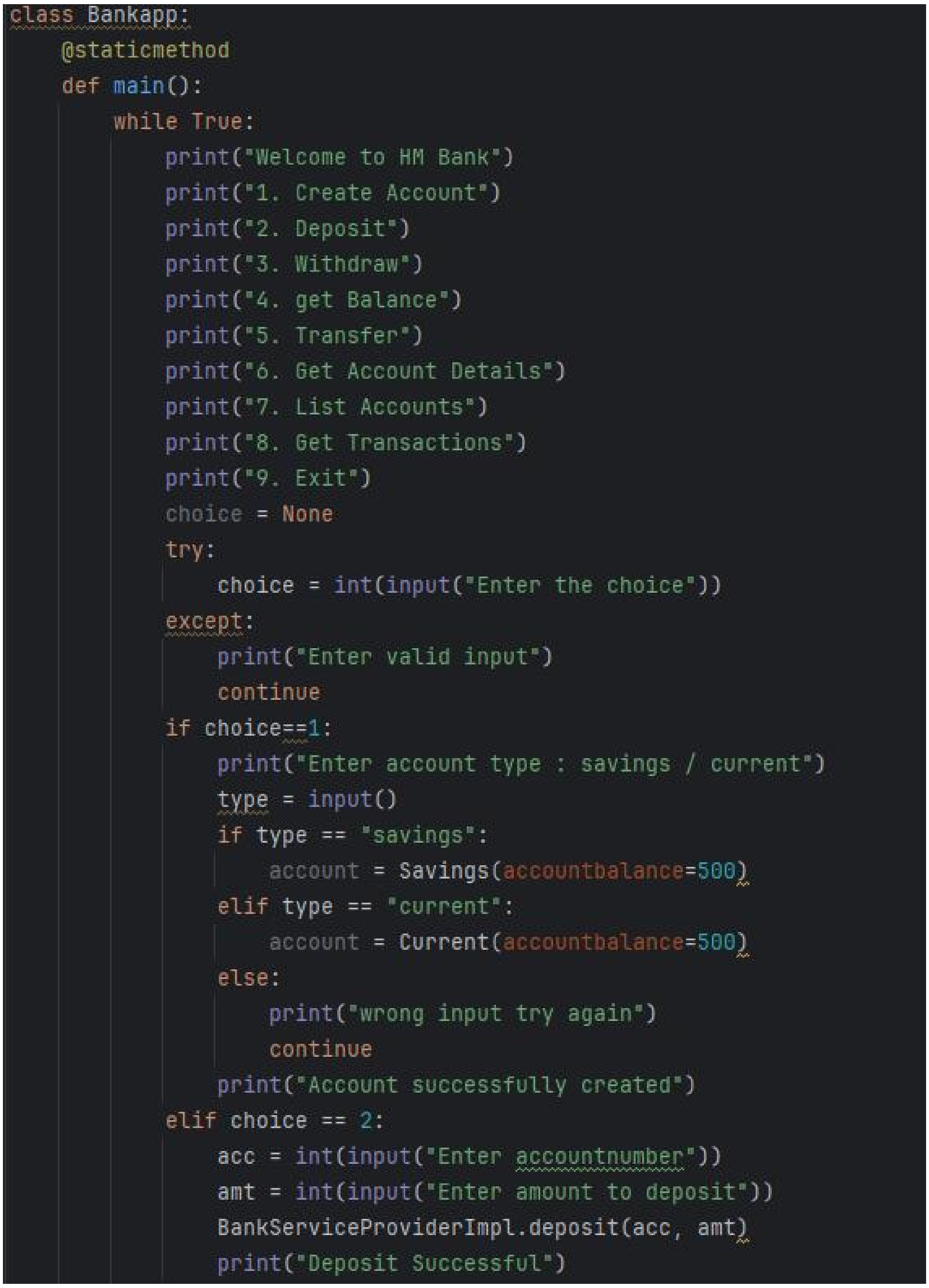


```
class Bank:

    def createaccount(self, customer, accountnumber, acctype, balance):
        if acctype == "savings":
            account = Savings(accountnumber, balance, customer)
        elif type == "current":
            account = Current(accountnumber, balance, customer)
        print("Account successfully created")
```

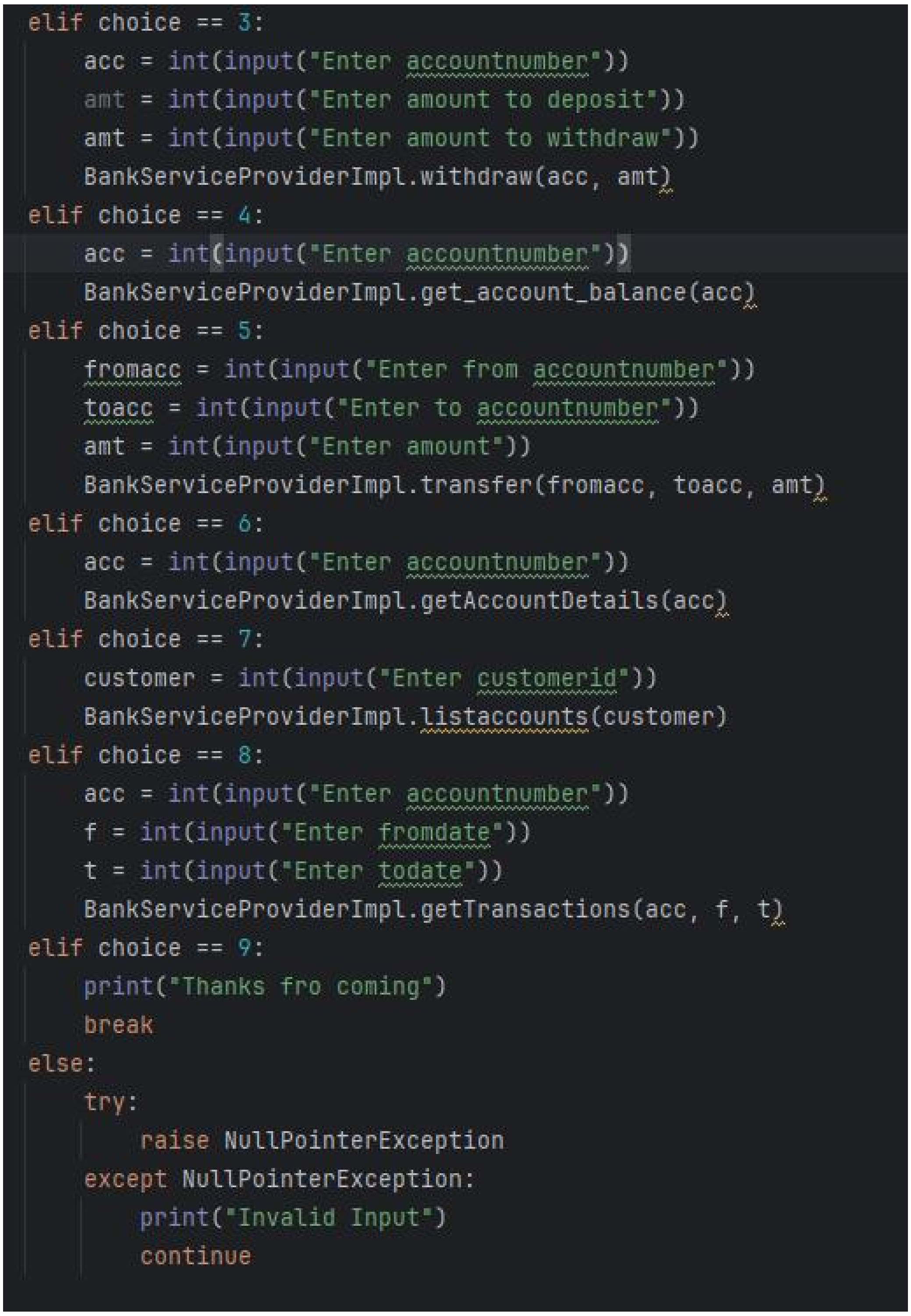


```
def deposit(self, account, amount):
    account.deposit(amount)
3 usages (2 dynamic)
def withdraw(self, account, amount):
    account.withdraw(amount)
1 usage
def transfer(self, account1, account2, amt):
    try:
        if account1.withdraw(amt):
            account2.deposit(amt)
        else:
            raise InsufficientFundException()
    except InsufficientFundException:
        print("InsufficientFund")
1 usage
def getAccountDetails(self, account):
    account.getaccountdetails()
```

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.

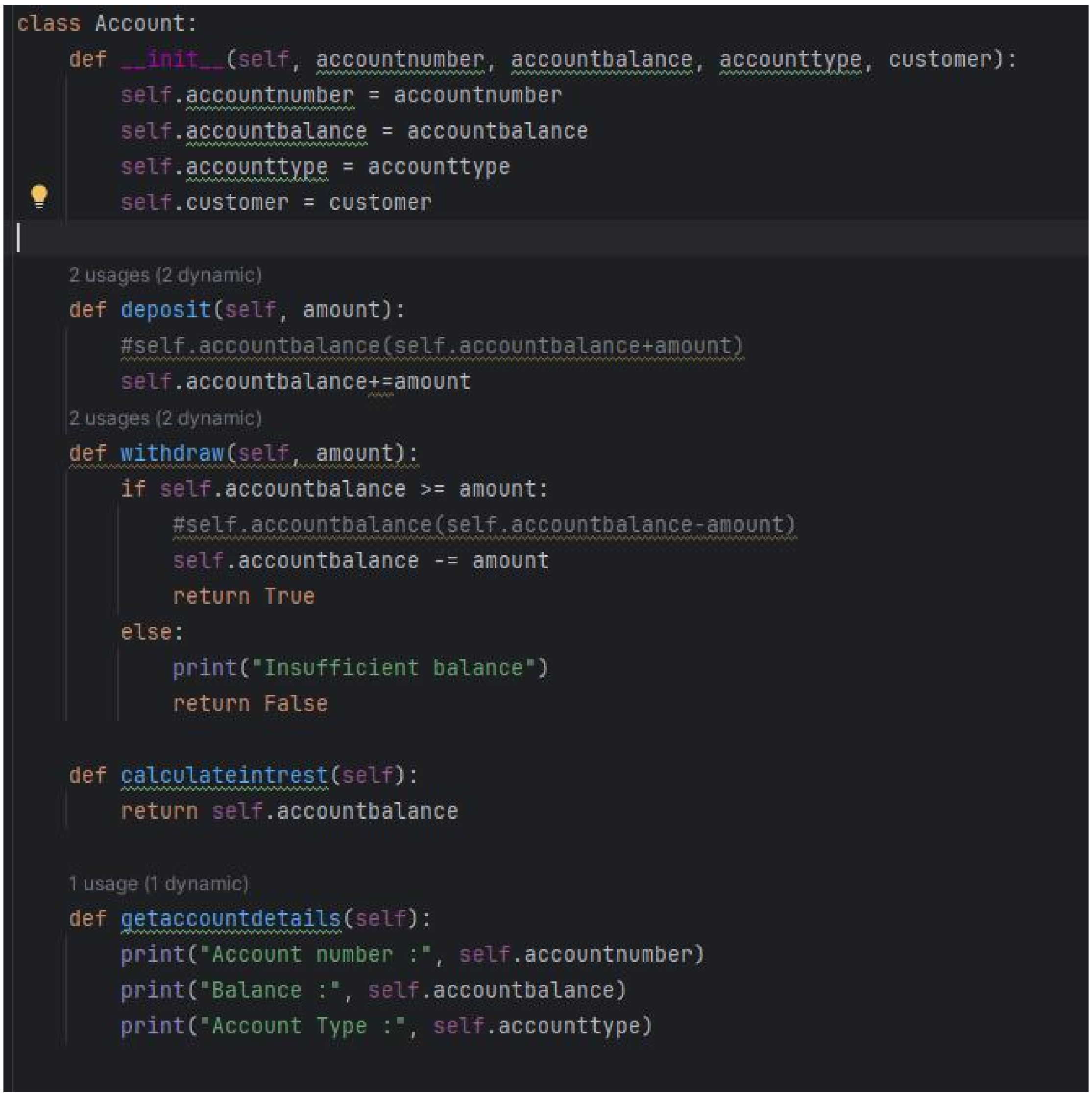


```python
class Bankapp:
    @staticmethod
    def main():
        while True:
            print("Welcome to HM Bank")
            print("1. Create Account")
            print("2. Deposit")
            print("3. Withdraw")
            print("4. get Balance")
            print("5. Transfer")
            print("6. Get Account Details")
            print("7. List Accounts")
            print("8. Get Transactions")
            print("9. Exit")
            choice = None
            try:
                choice = int(input("Enter the choice"))
            except:
                print("Enter valid input")
                continue
            if choice==1:
                print("Enter account type : savings / current")
                type = input()
                if type == "savings":
                    account = Savings(accountbalance=500)
                elif type == "current":
                    account = Current(accountbalance=500)
                else:
                    print("wrong input try again")
                    continue
                print("Account successfully created")
            elif choice == 2:
                acc = int(input("Enter accountnumber"))
                amt = int(input("Enter amount to deposit"))
                BankServiceProviderImpl.deposit(acc, amt)
                print("Deposit Successful")
```



```
elif choice == 3:
    acc = int(input("Enter accountnumber"))
    amt = int(input("Enter amount to deposit"))
    amt = int(input("Enter amount to withdraw"))
    BankServiceProviderImpl.withdraw(acc, amt)
elif choice == 4:
    acc = int(input("Enter accountnumber"))
    BankServiceProviderImpl.get_account_balance(acc)
elif choice == 5:
    fromacc = int(input("Enter from accountnumber"))
    toacc = int(input("Enter to accountnumber"))
    amt = int(input("Enter amount"))
    BankServiceProviderImpl.transfer(fromacc, toacc, amt)
elif choice == 6:
    acc = int(input("Enter accountnumber"))
    BankServiceProviderImpl.getAccountDetails(acc)
elif choice == 7:
    customer = int(input("Enter customerid"))
    BankServiceProviderImpl.listaccounts(customer)
elif choice == 8:
    acc = int(input("Enter accountnumber"))
    f = int(input("Enter fromdate"))
    t = int(input("Enter todate"))
    BankServiceProviderImpl.getTransactions(acc, f, t)
elif choice == 9:
    print("Thanks fro coming")
    break
else:
    try:
        raise NullPointerException
    except NullPointerException:
        print("Invalid Input")
        continue
```

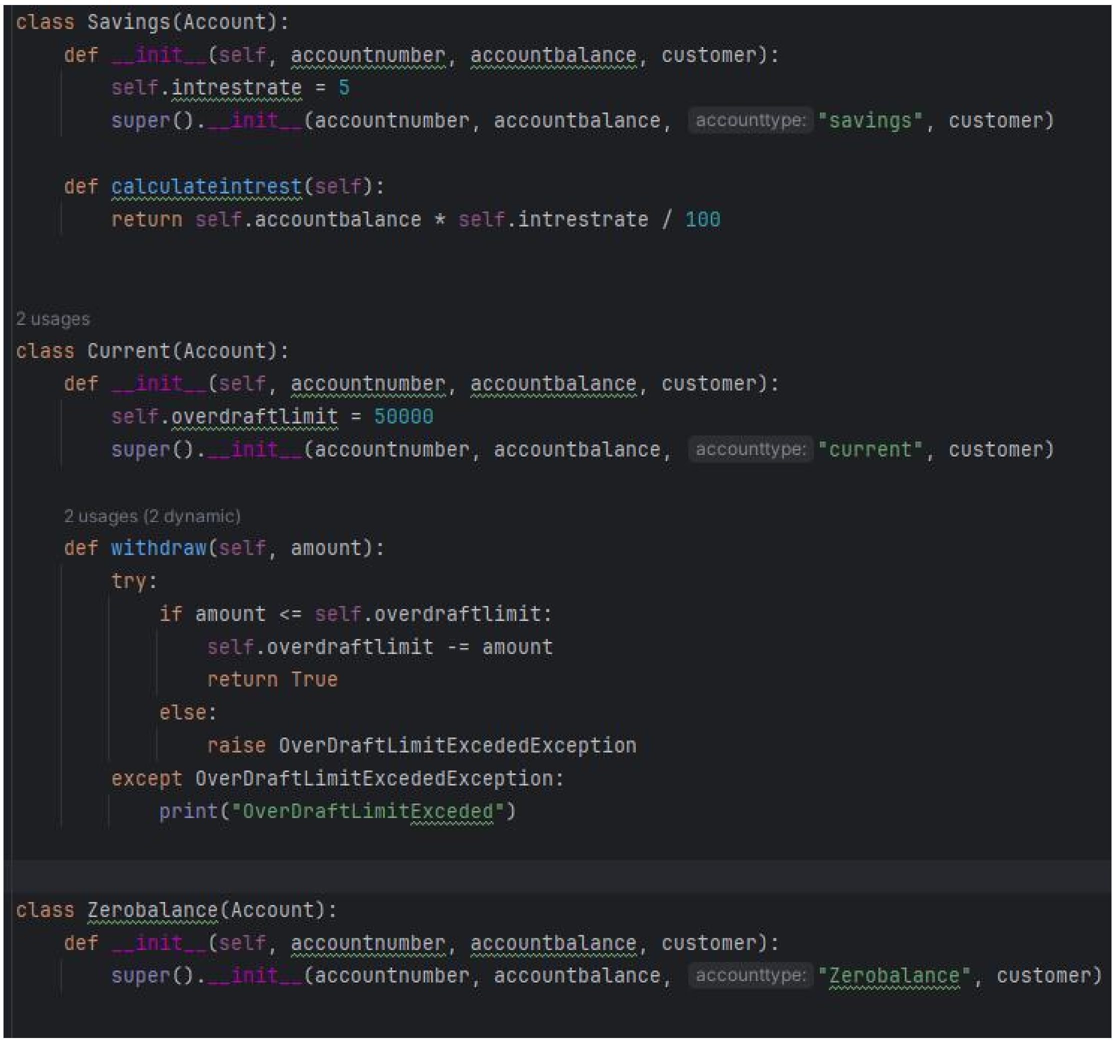
Create an class ‘**Account’** that includes the following attributes. Generate account number using
static variable.
• Account Number (a unique identifier).
• Account Type (e.g., Savings, Current)
• Account Balance
• Customer (the customer who owns the account)
• lastAccNo



```
class Account:
    def __init__(self, accountnumber, accountbalance, accounttype, customer):
        self.accountnumber = accountnumber
        self.accountbalance = accountbalance
        self.accounttype = accounttype
        self.customer = customer

    2 usages (2 dynamic)
    def deposit(self, amount):
        #self.accountbalance(self.accountbalance+amount)
        self.accountbalance+=amount
    2 usages (2 dynamic)
    def withdraw(self, amount):
        if self.accountbalance >= amount:
            #self.accountbalance(self.accountbalance-amount)
            self.accountbalance -= amount
            return True
        else:
            print("Insufficient balance")
            return False

    def calculateintrest(self):
        return self.accountbalance

    1 usage (1 dynamic)
    def getaccountdetails(self):
        print("Account number :", self.accountnumber)
        print("Balance :", self.accountbalance)
        print("Account Type :", self.accounttype)
```

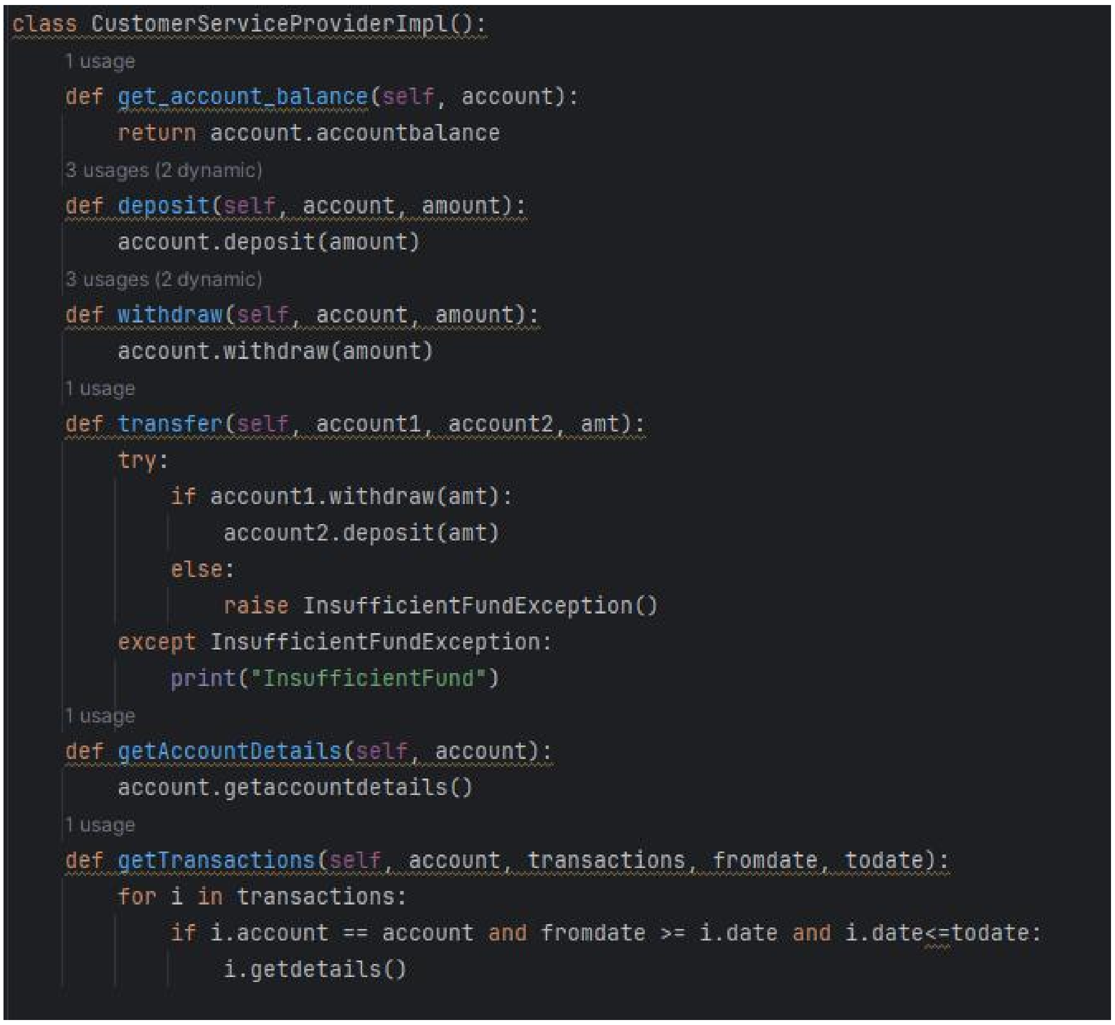
Create three child classes that inherit the Account class and each class must contain below
mentioned attribute:
• **SavingsAccount:** A savings account that includes an additional attribute for interest rate.
Saving account should be created with minimum balance 500.

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



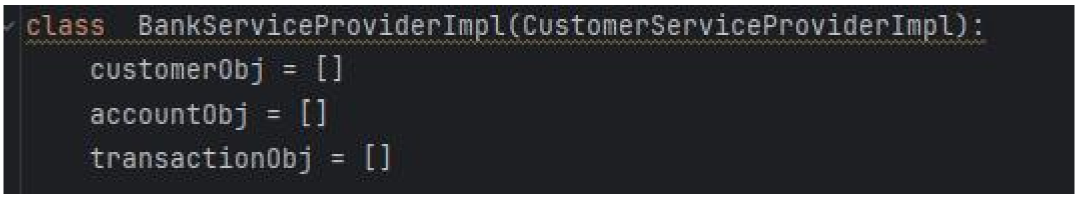
```
class Savings(Account):
    def __init__(self, accountnumber, accountbalance, customer):
        self.intrestrate = 5
        super().__init__(accountnumber, accountbalance, accounttype: "savings", customer)

    def calculateintrest(self):
        return self.accountbalance * self.intrestrate / 100


2 usages
class Current(Account):
    def __init__(self, accountnumber, accountbalance, customer):
        self.overdraftlimit = 50000
        super().__init__(accountnumber, accountbalance, accounttype: "current", customer)

    2 usages (2 dynamic)
    def withdraw(self, amount):
        try:
            if amount <= self.overdraftlimit:
                self.overdraftlimit -= amount
                return True
            else:
                raise OverDraftLimitExcededException
        except OverDraftLimitExcededException:
            print("OverDraftLimitExceded")


class Zerobalance(Account):
    def __init__(self, accountnumber, accountbalance, customer):
        super().__init__(accountnumber, accountbalance, accounttype: "Zerobalance", customer)
```

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



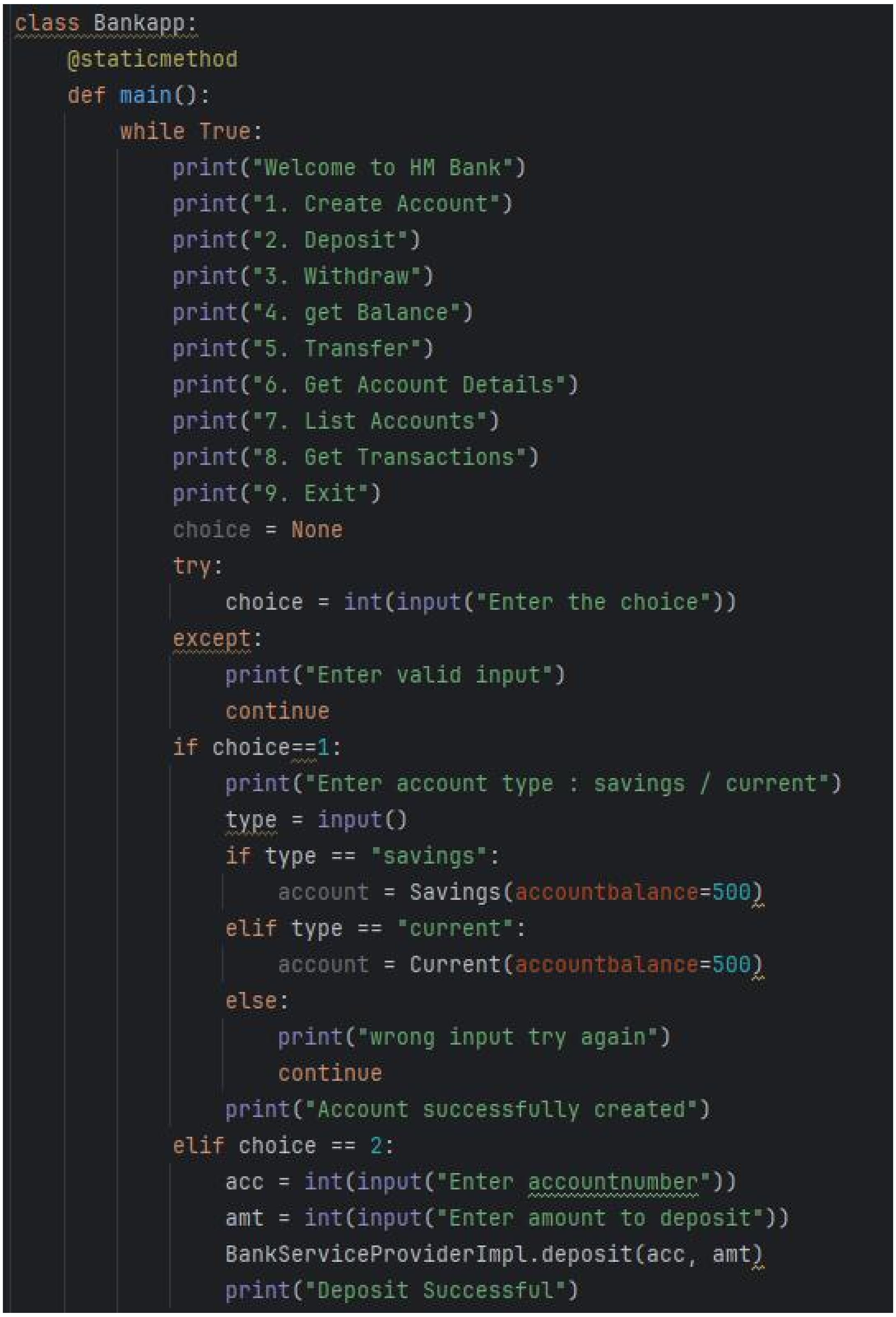
```
class CustomerServiceProviderImpl():
    1 usage
    def get_account_balance(self, account):
        return account.accountbalance
    3 usages (2 dynamic)
    def deposit(self, account, amount):
        account.deposit(amount)
    3 usages (2 dynamic)
    def withdraw(self, account, amount):
        account.withdraw(amount)
    1 usage
    def transfer(self, account1, account2, amt):
        try:
            if account1.withdraw(amt):
                account2.deposit(amt)
            else:
                raise InsufficientFundException()
        except InsufficientFundException:
            print("InsufficientFund")
    1 usage
    def getAccountDetails(self, account):
        account.getaccountdetails()
    1 usage
    def getTransactions(self, account, transactions, fromdate, todate):
        for i in transactions:
            if i.account == account and fromdate >= i.date and i.date<=todate:
                i.getdetails()
```

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

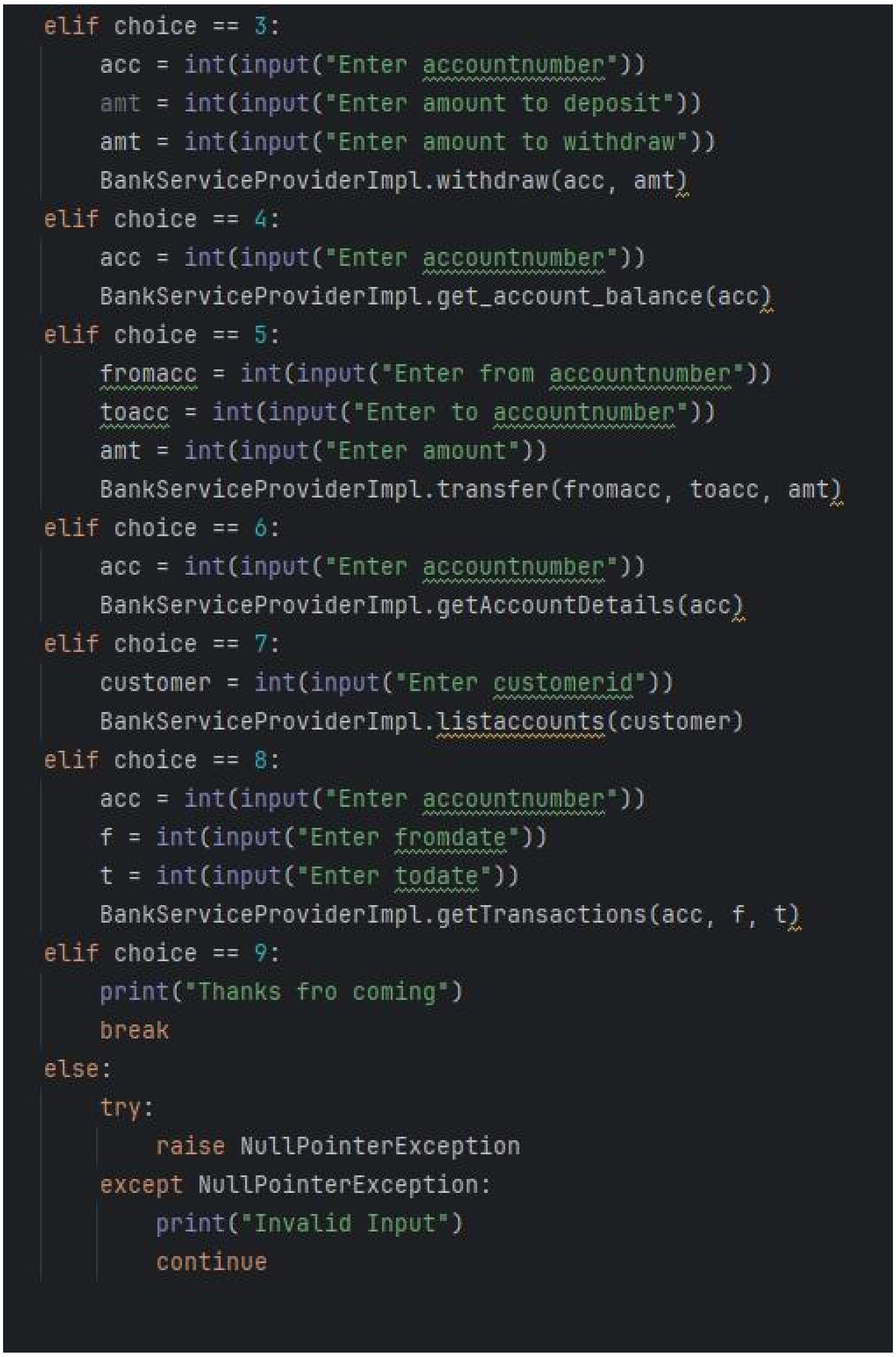


```
class  BankServiceProviderImpl(CustomerServiceProviderImpl):
    customerObj = []
    accountObj = []
    transactionObj = []
```

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.

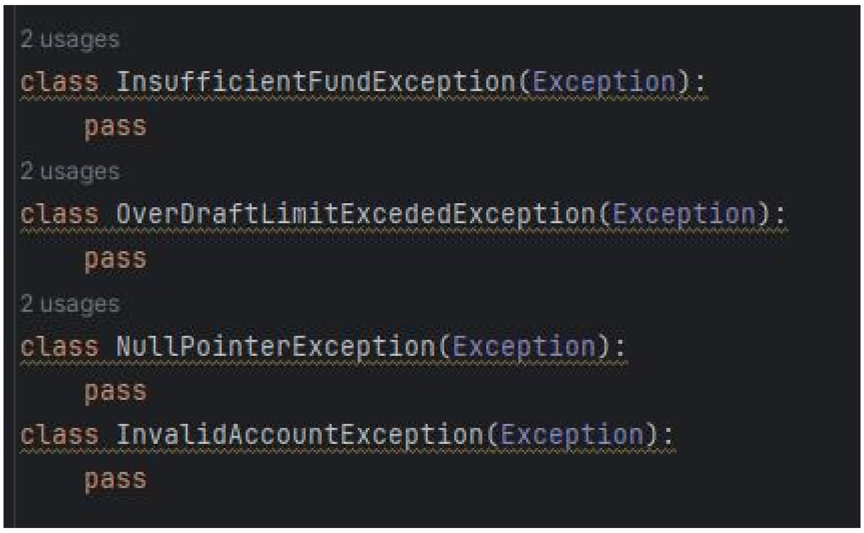


```python
class Bankapp:
    @staticmethod
    def main():
        while True:
            print("Welcome to HM Bank")
            print("1. Create Account")
            print("2. Deposit")
            print("3. Withdraw")
            print("4. get Balance")
            print("5. Transfer")
            print("6. Get Account Details")
            print("7. List Accounts")
            print("8. Get Transactions")
            print("9. Exit")
            choice = None
            try:
                choice = int(input("Enter the choice"))
            except:
                print("Enter valid input")
                continue
            if choice==1:
                print("Enter account type : savings / current")
                type = input()
                if type == "savings":
                    account = Savings(accountbalance=500)
                elif type == "current":
                    account = Current(accountbalance=500)
                else:
                    print("wrong input try again")
                    continue
                print("Account successfully created")
            elif choice == 2:
                acc = int(input("Enter accountnumber"))
                amt = int(input("Enter amount to deposit"))
                BankServiceProviderImpl.deposit(acc, amt)
                print("Deposit Successful")
```



```
elif choice == 3:
    acc = int(input("Enter accountnumber"))
    amt = int(input("Enter amount to deposit"))
    amt = int(input("Enter amount to withdraw"))
    BankServiceProviderImpl.withdraw(acc, amt)
elif choice == 4:
    acc = int(input("Enter accountnumber"))
    BankServiceProviderImpl.get_account_balance(acc)
elif choice == 5:
    fromacc = int(input("Enter from accountnumber"))
    toacc = int(input("Enter to accountnumber"))
    amt = int(input("Enter amount"))
    BankServiceProviderImpl.transfer(fromacc, toacc, amt)
elif choice == 6:
    acc = int(input("Enter accountnumber"))
    BankServiceProviderImpl.getAccountDetails(acc)
elif choice == 7:
    customer = int(input("Enter customerid"))
    BankServiceProviderImpl.listaccounts(customer)
elif choice == 8:
    acc = int(input("Enter accountnumber"))
    f = int(input("Enter fromdate"))
    t = int(input("Enter todate"))
    BankServiceProviderImpl.getTransactions(acc, f, t)
elif choice == 9:
    print("Thanks fro coming")
    break
else:
    try:
        raise NullPointerException
    except NullPointerException:
        print("Invalid Input")
        continue
```

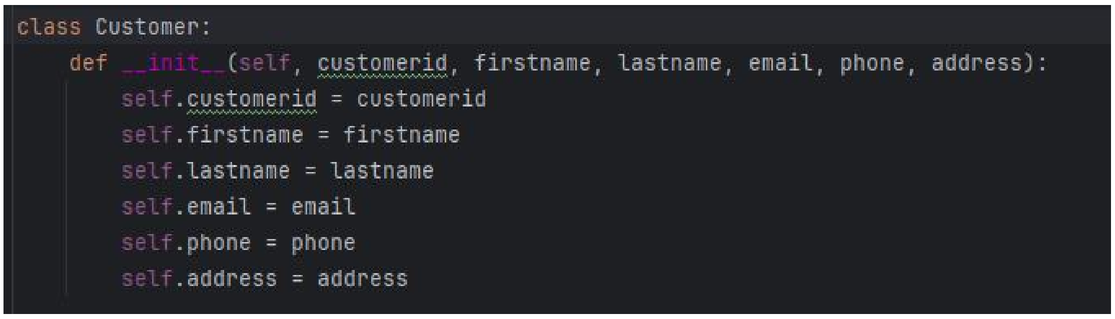
Task 12: Exception Handling
throw the exception whenever needed and Handle in main method,
1. **InsufficientFundException** throw this exception when user try to withdraw amount or transfer
amount to another account and the account runs out of money in the account.
2. **InvalidAccountException** throw 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



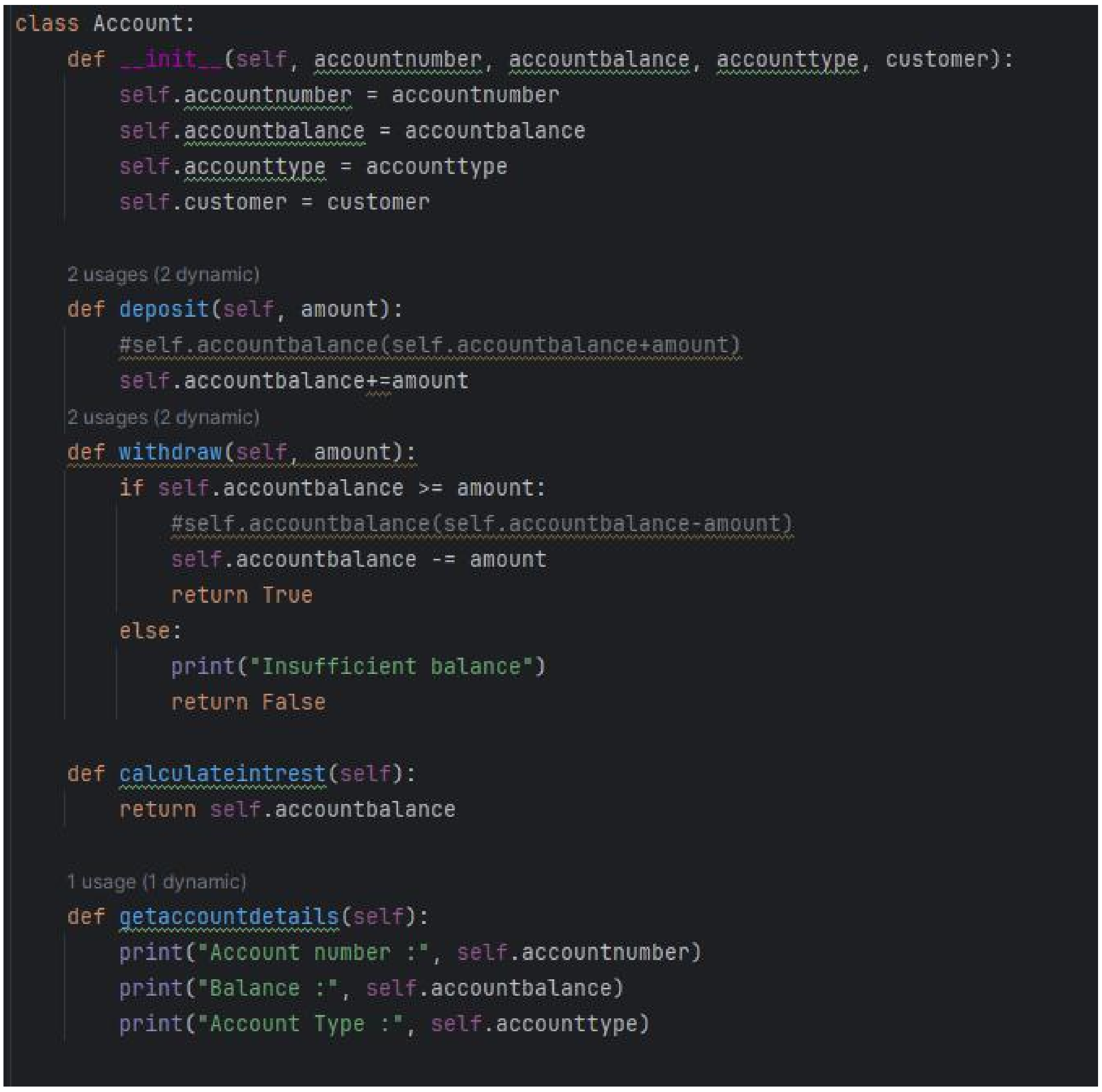
```
2 usages
class InsufficientFundException(Exception):
    pass
2 usages
class OverDraftLimitExcededException(Exception):
    pass
2 usages
class NullPointerException(Exception):
    pass
class InvalidAccountException(Exception):
    pass
```

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

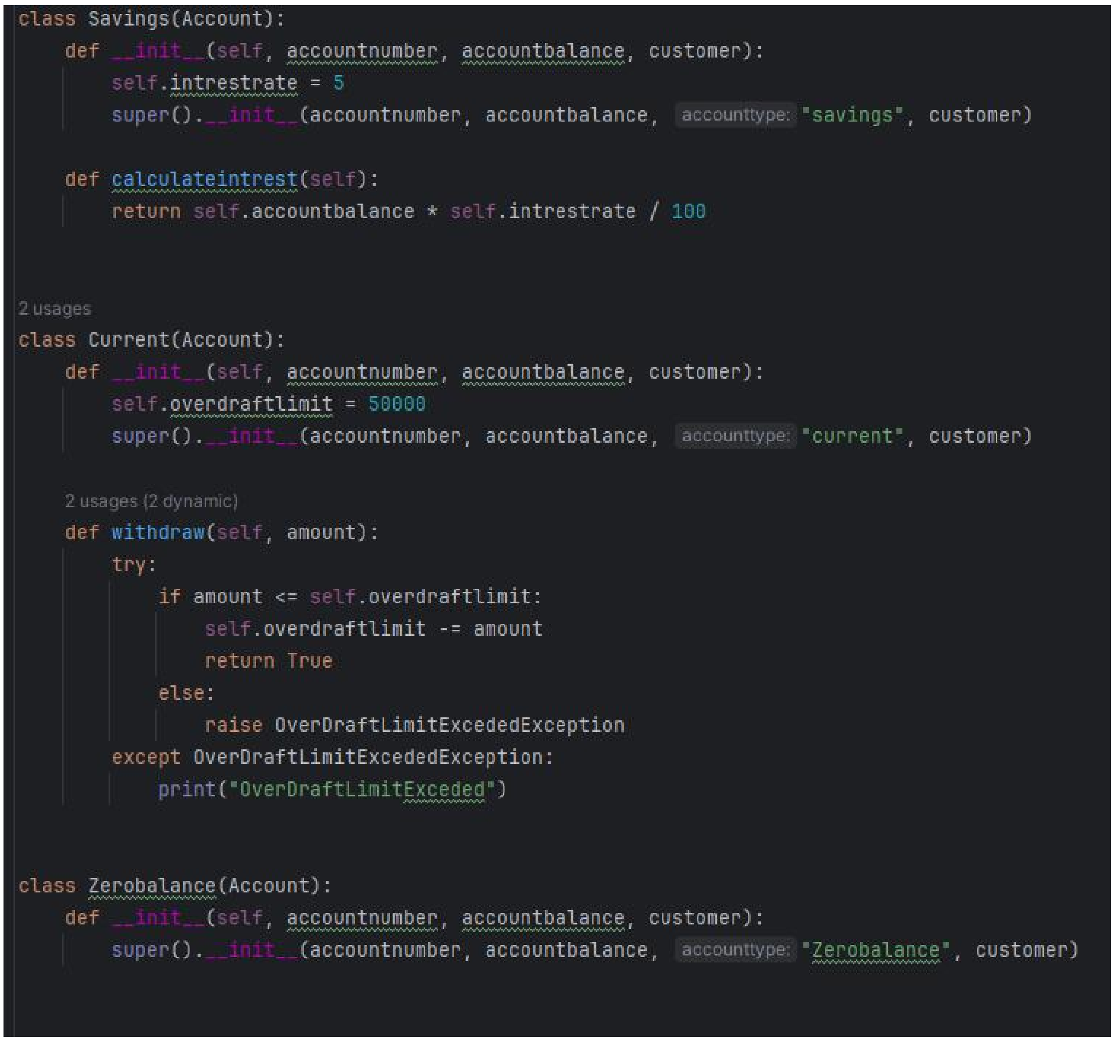
4. Create three child classes that inherit the Account class and each class must contain below
mentioned attribute:
• **SavingsAccount:** A savings account that includes an additional attribute for interest rate.
Saving account should be created with minimum balance 500.
• **CurrentAccount:** A Current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



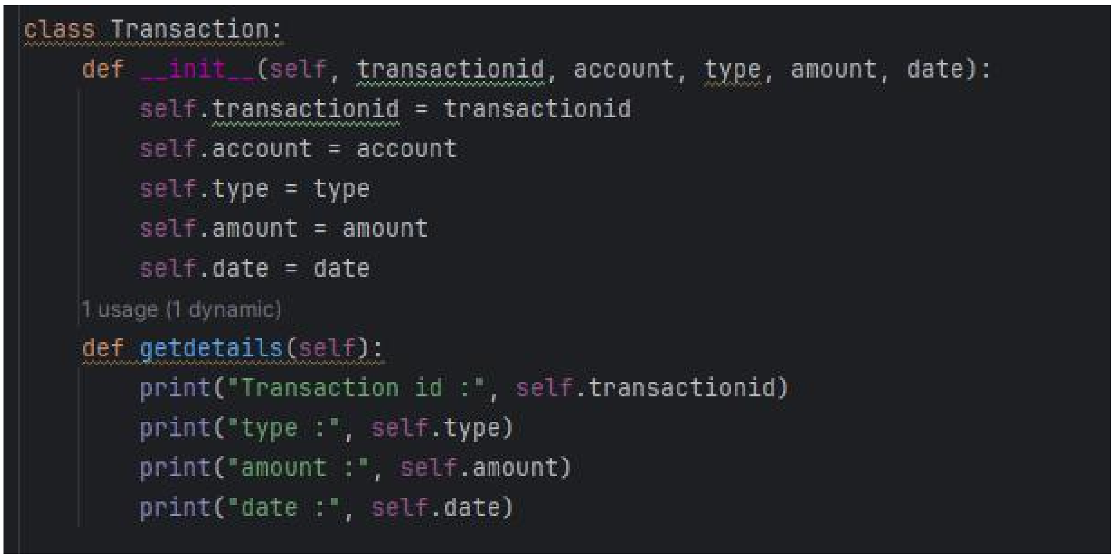
```python
class Customer:
    def __init__(self, customerid, firstname, lastname, email, phone, address):
        self.customerid = customerid
        self.firstname = firstname
        self.lastname = lastname
        self.email = email
        self.phone = phone
        self.address = address
```



```python
class Account:
    def __init__(self, accountnumber, accountbalance, accounttype, customer):
        self.accountnumber = accountnumber
        self.accountbalance = accountbalance
        self.accounttype = accounttype
        self.customer = customer

    2 usages (2 dynamic)
    def deposit(self, amount):
        #self.accountbalance(self.accountbalance+amount)
        self.accountbalance+=amount
    2 usages (2 dynamic)
    def withdraw(self, amount):
        if self.accountbalance >= amount:
            #self.accountbalance(self.accountbalance-amount)
            self.accountbalance -= amount
            return True
        else:
            print("Insufficient balance")
            return False

    def calculateintrest(self):
        return self.accountbalance

    1 usage (1 dynamic)
    def getaccountdetails(self):
        print("Account number :", self.accountnumber)
        print("Balance :", self.accountbalance)
        print("Account Type :", self.accounttype)
```

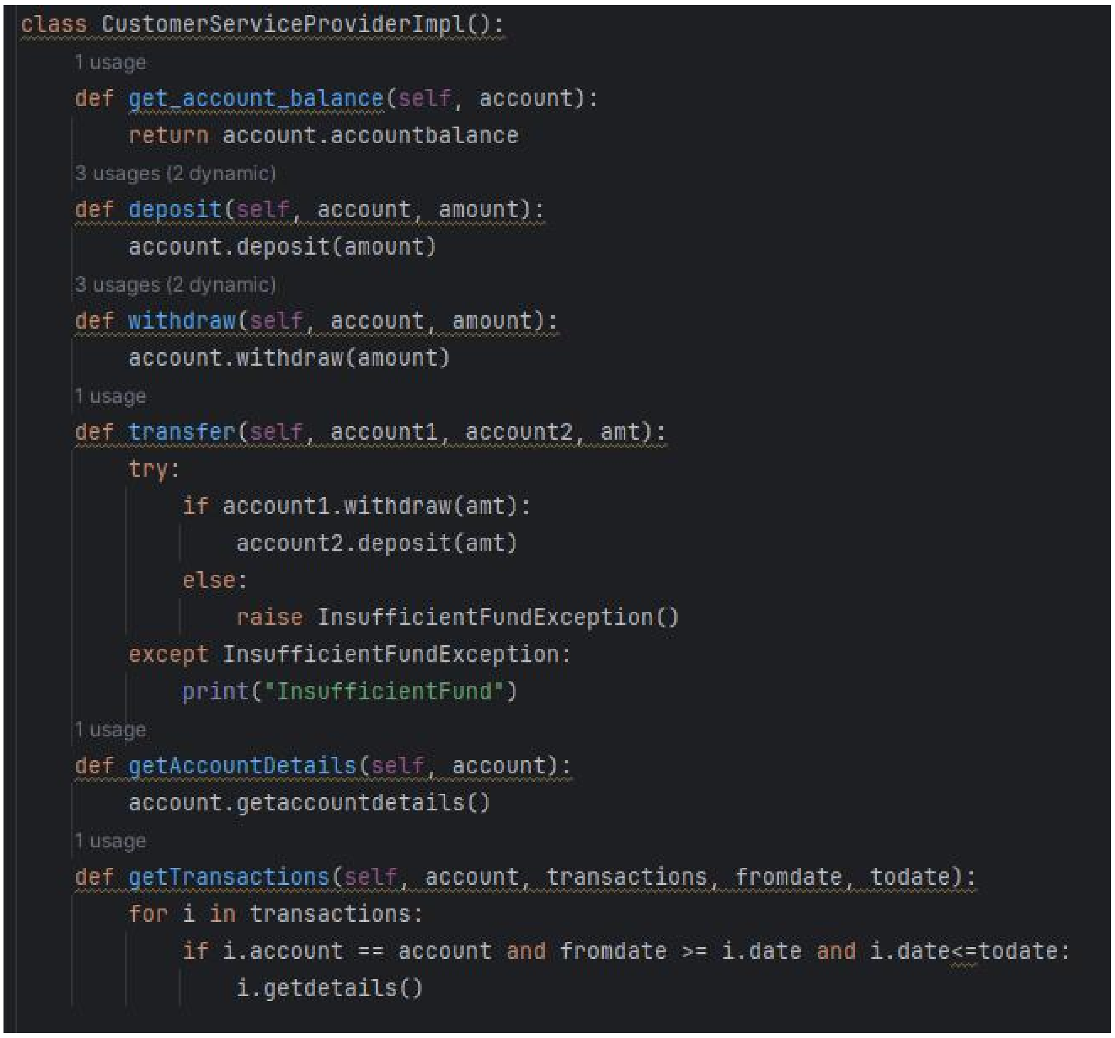


```python
class Savings(Account):
    def __init__(self, accountnumber, accountbalance, customer):
        self.intrestrate = 5
        super().__init__(accountnumber, accountbalance,  accounttype: "savings", customer)

    def calculateintrest(self):
        return self.accountbalance * self.intrestrate / 100


2 usages
class Current(Account):
    def __init__(self, accountnumber, accountbalance, customer):
        self.overdraftlimit = 50000
        super().__init__(accountnumber, accountbalance,  accounttype: "current", customer)

    2 usages (2 dynamic)
    def withdraw(self, amount):
        try:
            if amount <= self.overdraftlimit:
                self.overdraftlimit -= amount
                return True
            else:
                raise OverDraftLimitExcededException
        except OverDraftLimitExcededException:
            print("OverDraftLimitExceded")


class Zerobalance(Account):
    def __init__(self, accountnumber, accountbalance, customer):
        super().__init__(accountnumber, accountbalance,  accounttype: "Zerobalance", customer)
```



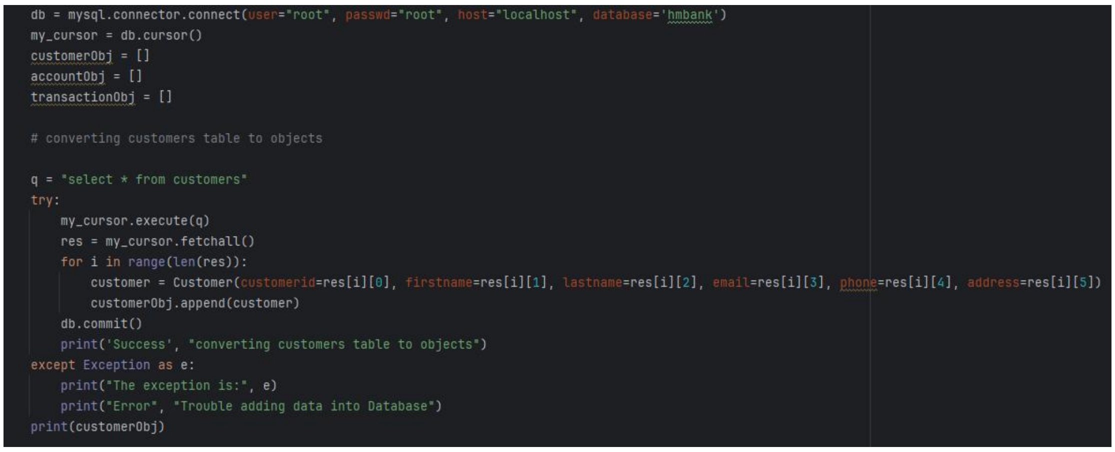
```python
class Transaction:
    def __init__(self, transactionid, account, type, amount, date):
        self.transactionid = transactionid
        self.account = account
        self.type = type
        self.amount = amount
        self.date = date
    1 usage (1 dynamic)
    def getdetails(self):
        print("Transaction id :", self.transactionid)
        print("type :", self.type)
        print("amount :", self.amount)
        print("date :", self.date)
```

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

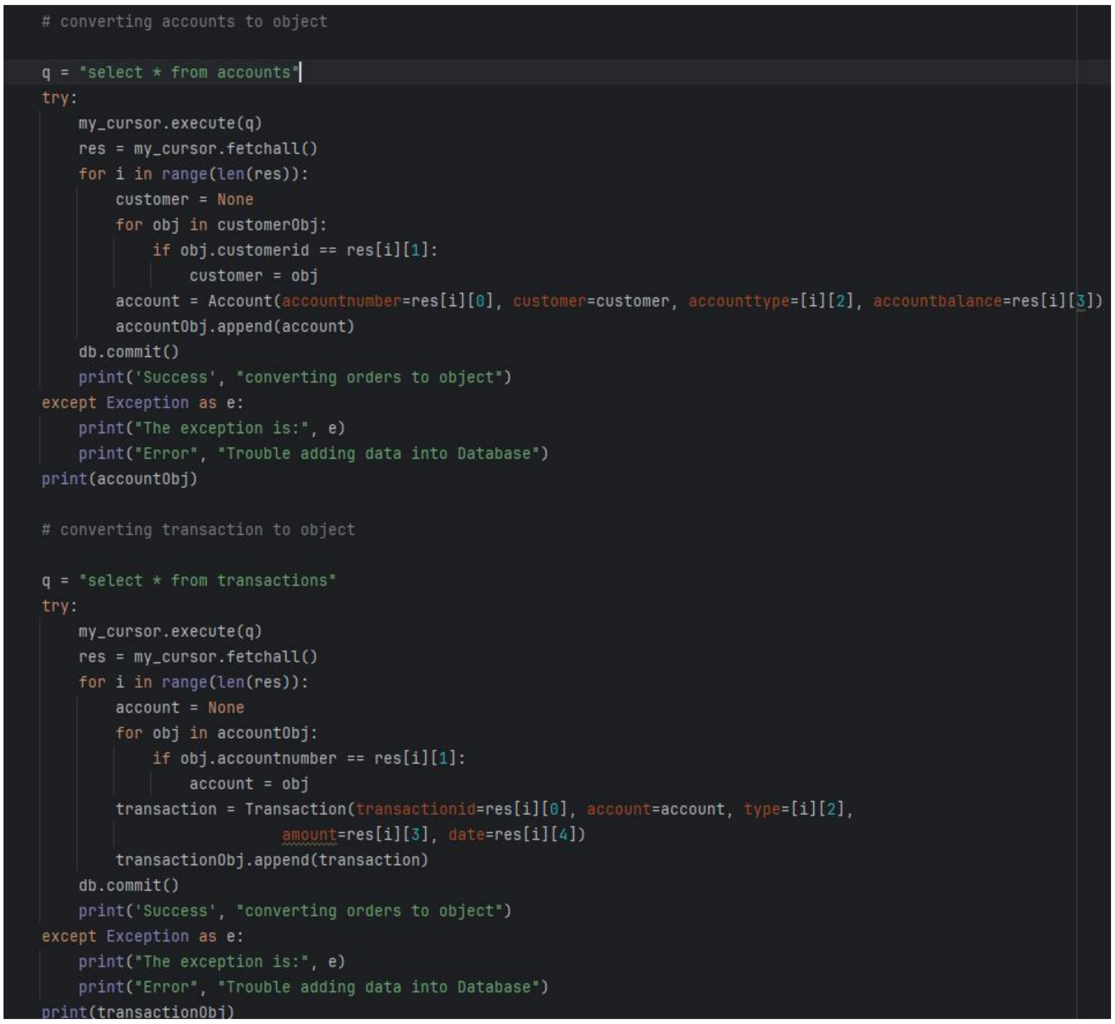


```python
class CustomerServiceProviderImpl():
    1 usage
    def get_account_balance(self, account):
        return account.accountbalance
    3 usages (2 dynamic)
    def deposit(self, account, amount):
        account.deposit(amount)
    3 usages (2 dynamic)
    def withdraw(self, account, amount):
        account.withdraw(amount)
    1 usage
    def transfer(self, account1, account2, amt):
        try:
            if account1.withdraw(amt):
                account2.deposit(amt)
            else:
                raise InsufficientFundException()
        except InsufficientFundException:
            print("InsufficientFund")
    1 usage
    def getAccountDetails(self, account):
        account.getaccountdetails()
    1 usage
    def getTransactions(self, account, transactions, fromdate, todate):
        for i in transactions:
            if i.account == account and fromdate >= i.date and i.date<=todate:
                i.getdetails()
```

Create **BankServiceProviderImpl** class which inherits from **CustomerServiceProviderImpl and**
implements **IBankServiceProvider.**
• Attributes
o accountList: List of **Accounts** to store any account objects.
o transactionList: List of **Transaction** to store transaction objects.
o branchName and branchAddress as String objects

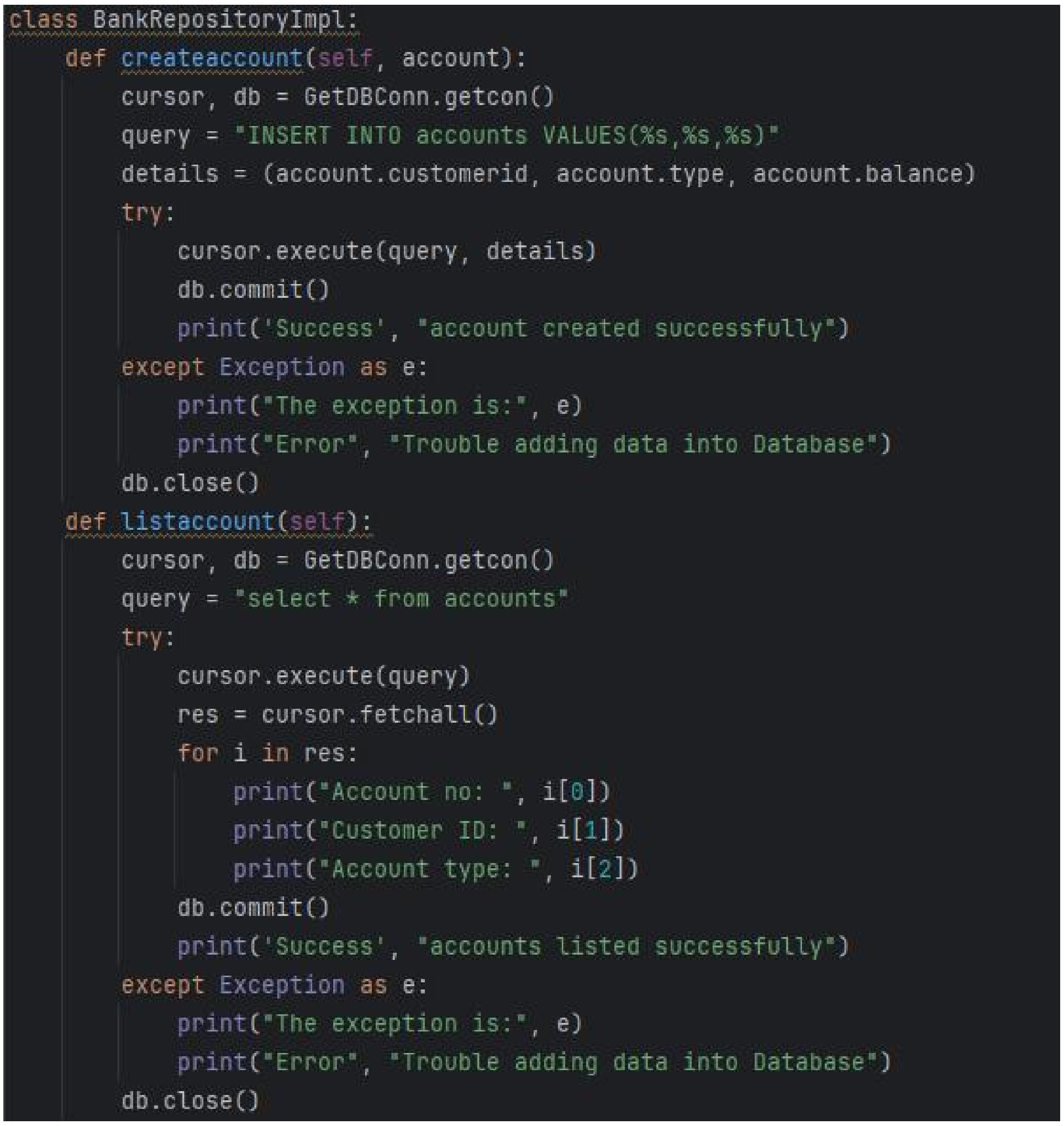


```python
db = mysql.connector.connect(user="root", passwd="root", host="localhost", database='hmbank')
my_cursor = db.cursor()
customerObj = []
accountObj = []
transactionObj = []

# converting customers table to objects

q = "select * from customers"
try:
    my_cursor.execute(q)
    res = my_cursor.fetchall()
    for i in range(len(res)):
        customer = Customer(customerid=res[i][0], firstname=res[i][1], lastname=res[i][2], email=res[i][3], phone=res[i][4], address=res[i][5])
        customerObj.append(customer)
    db.commit()
    print('Success', "converting customers table to objects")
except Exception as e:
    print("The exception is:", e)
    print("Error", "Trouble adding data into Database")
print(customerObj)
```

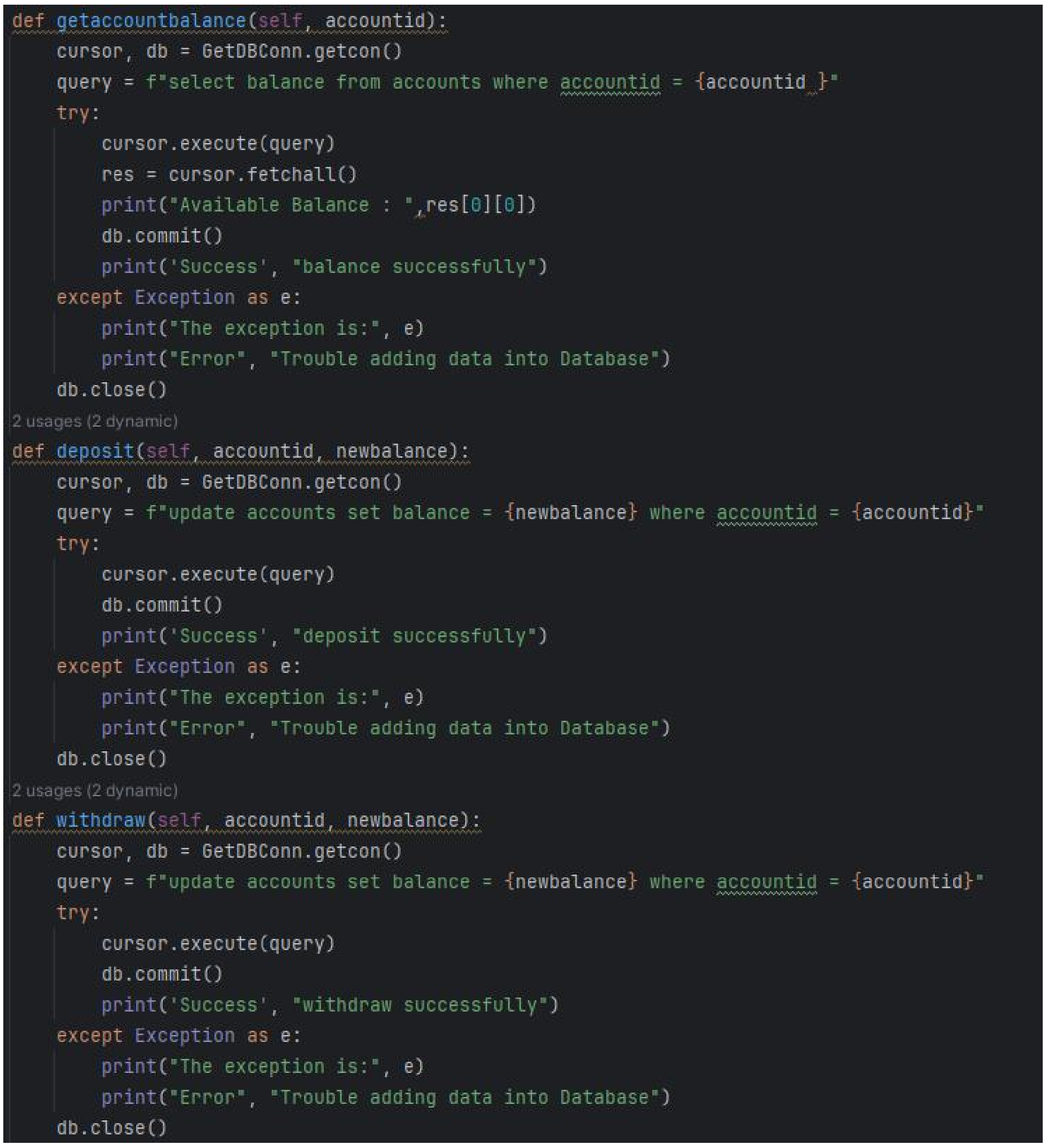


```python
# converting accounts to object

q = "select * from accounts"
try:
    my_cursor.execute(q)
    res = my_cursor.fetchall()
    for i in range(len(res)):
        customer = None
        for obj in customerObj:
            if obj.customerid == res[i][1]:
                customer = obj
        account = Account(accountnumber=res[i][0], customer=customer, accounttype=[i][2], accountbalance=res[i][3])
        accountObj.append(account)
    db.commit()
    print('Success', "converting orders to object")
except Exception as e:
    print("The exception is:", e)
    print("Error", "Trouble adding data into Database")
print(accountObj)

# converting transaction to object

q = "select * from transactions"
try:
    my_cursor.execute(q)
    res = my_cursor.fetchall()
    for i in range(len(res)):
        account = None
        for obj in accountObj:
            if obj.accountnumber == res[i][1]:
                account = obj
        transaction = Transaction(transactionid=res[i][0], account=account, type=[i][2],
                                  amount=res[i][3], date=res[i][4])
        transactionObj.append(transaction)
    db.commit()
    print('Success', "converting orders to object")
except Exception as e:
    print("The exception is:", e)
    print("Error", "Trouble adding data into Database")
print(transactionObj)
```

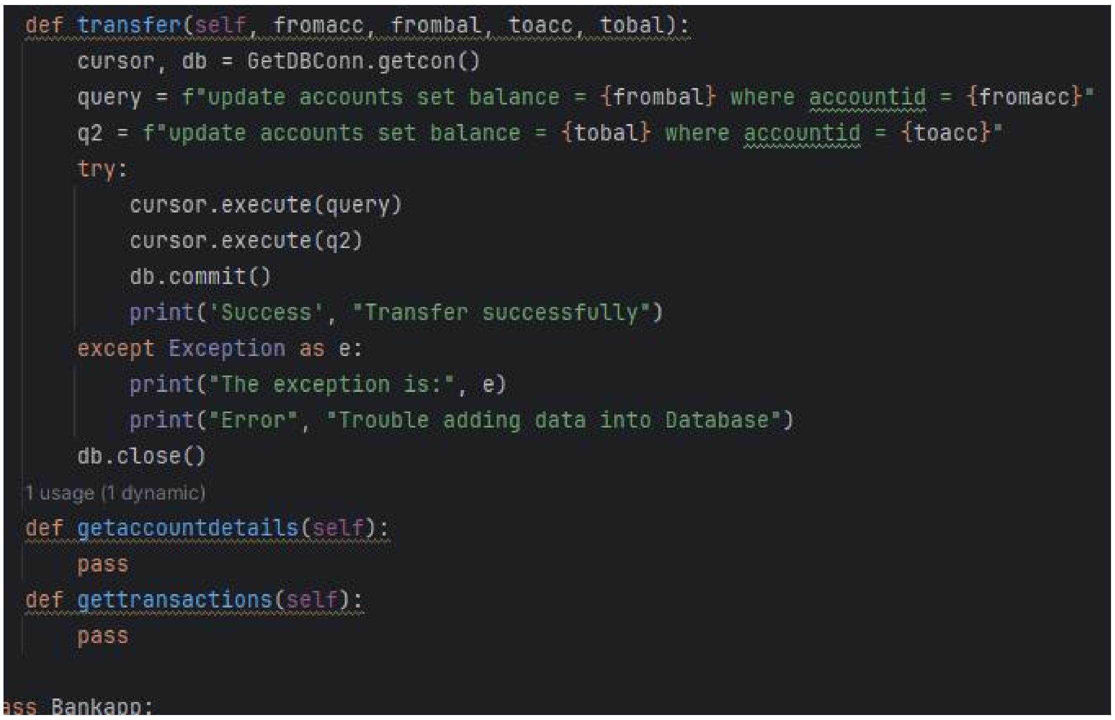
Create **BankRepositoryImpl** class which implement the **IBankRepository** interface/abstract class
and provide implementation of all methods and perform the database operations.



```python
class BankRepositoryImpl:
    def createaccount(self, account):
        cursor, db = GetDBConn.getcon()
        query = "INSERT INTO accounts VALUES(%s,%s,%s)"
        details = (account.customerid, account.type, account.balance)
        try:
            cursor.execute(query, details)
            db.commit()
            print('Success', "account created successfully")
        except Exception as e:
            print("The exception is:", e)
            print("Error", "Trouble adding data into Database")
        db.close()
    def listaccount(self):
        cursor, db = GetDBConn.getcon()
        query = "select * from accounts"
        try:
            cursor.execute(query)
            res = cursor.fetchall()
            for i in res:
                print("Account no: ", i[0])
                print("Customer ID: ", i[1])
                print("Account type: ", i[2])
            db.commit()
            print('Success', "accounts listed successfully")
        except Exception as e:
            print("The exception is:", e)
            print("Error", "Trouble adding data into Database")
        db.close()
```

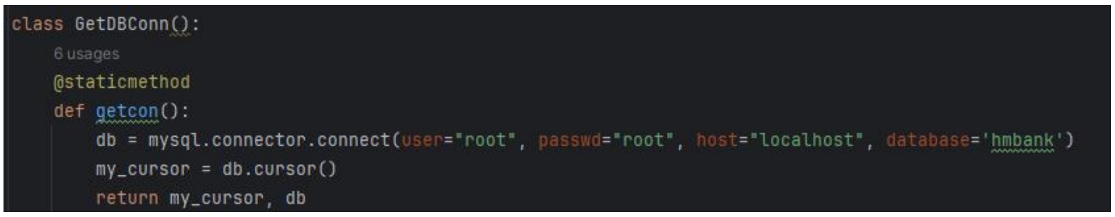


```python
def getaccountbalance(self, accountid):
    cursor, db = GetDBConn.getcon()
    query = f"select balance from accounts where accountid = {accountid }"
    try:
        cursor.execute(query)
        res = cursor.fetchall()
        print("Available Balance : ",res[0][0])
        db.commit()
        print('Success', "balance successfully")
    except Exception as e:
        print("The exception is:", e)
        print("Error", "Trouble adding data into Database")
    db.close()
2 usages (2 dynamic)
def deposit(self, accountid, newbalance):
    cursor, db = GetDBConn.getcon()
    query = f"update accounts set balance = {newbalance} where accountid = {accountid}"
    try:
        cursor.execute(query)
        db.commit()
        print('Success', "deposit successfully")
    except Exception as e:
        print("The exception is:", e)
        print("Error", "Trouble adding data into Database")
    db.close()
2 usages (2 dynamic)
def withdraw(self, accountid, newbalance):
    cursor, db = GetDBConn.getcon()
    query = f"update accounts set balance = {newbalance} where accountid = {accountid}"
    try:
        cursor.execute(query)
        db.commit()
        print('Success', "withdraw successfully")
    except Exception as e:
        print("The exception is:", e)
        print("Error", "Trouble adding data into Database")
    db.close()
```



```
def transfer(self, fromacc, frombal, toacc, tobal):
    cursor, db = GetDBConn.getcon()
    query = f"update accounts set balance = {frombal} where accountid = {fromacc}"
    q2 = f"update accounts set balance = {tobal} where accountid = {toacc}"
    try:
        cursor.execute(query)
        cursor.execute(q2)
        db.commit()
        print('Success', "Transfer successfully")
    except Exception as e:
        print("The exception is:", e)
        print("Error", "Trouble adding data into Database")
    db.close()
1 usage (1 dynamic)
def getaccountdetails(self):
    pass
def gettransactions(self):
    pass

ass Bankapp:
```

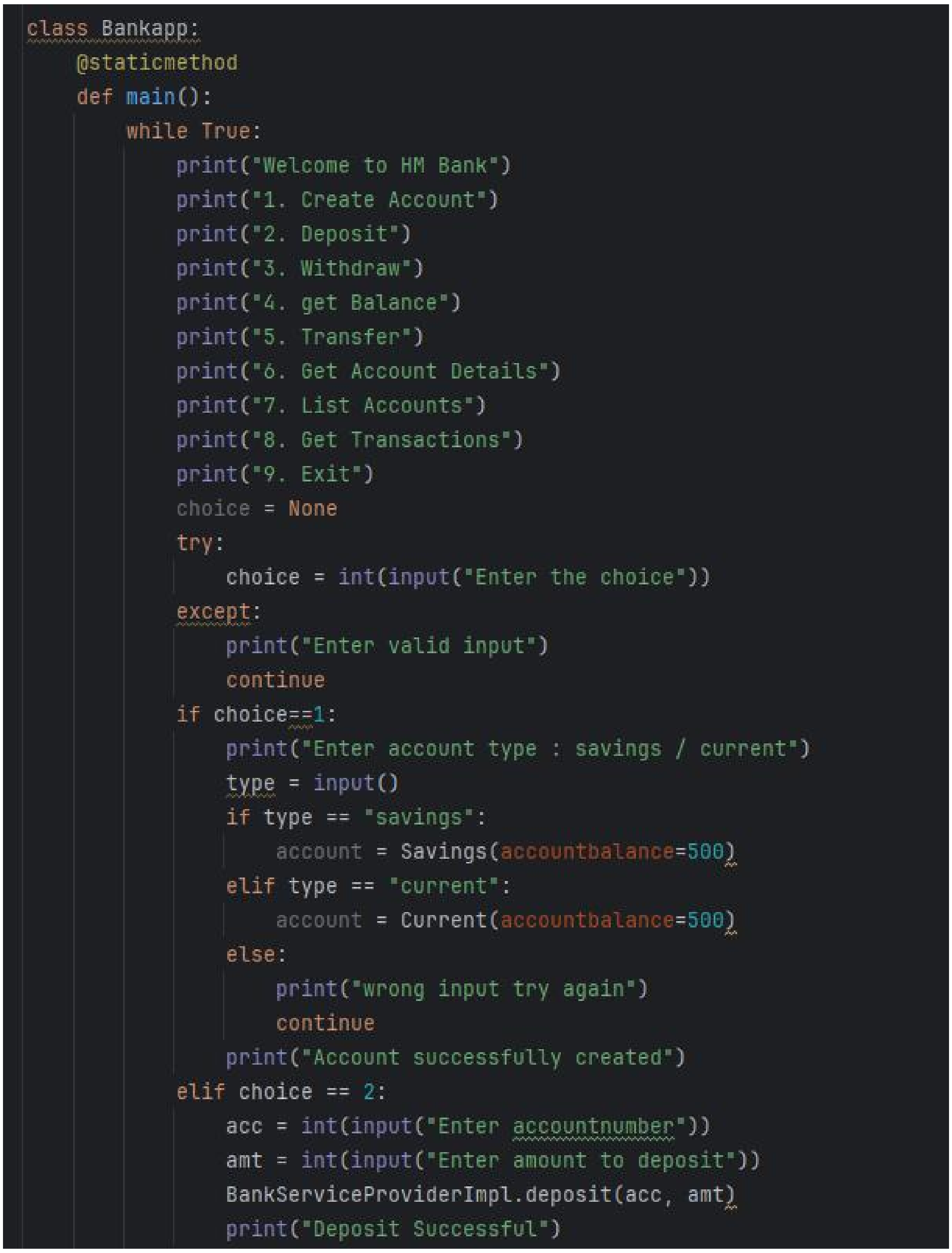
Create **DBUtil** class and add the following method.
• **static getDBConn():Connection** Establish a connection to the database and return
Connection reference



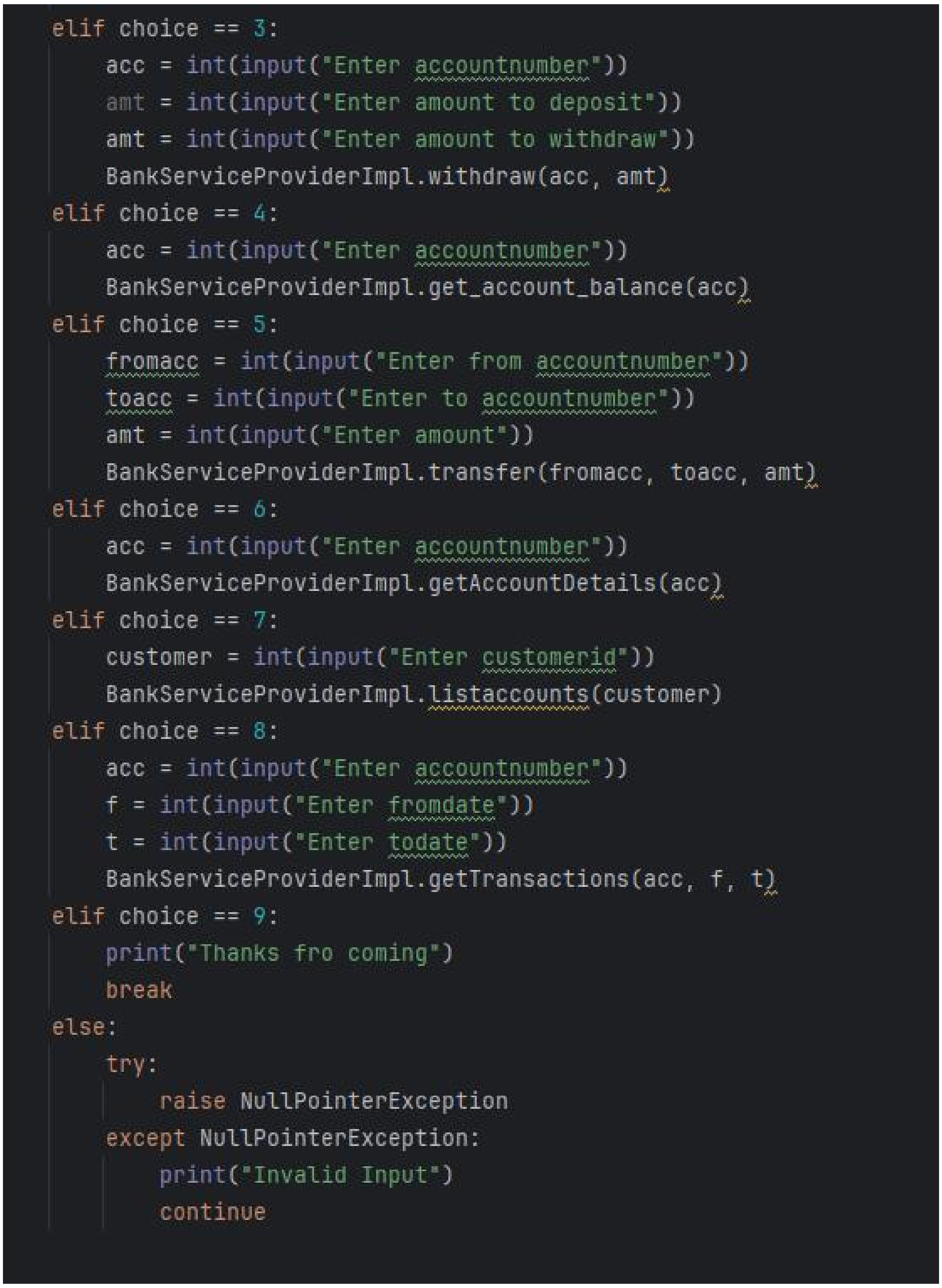
```
class GetDBConn():
    6 usages
    @staticmethod
    def getcon():
        db = mysql.connector.connect(user="root", passwd="root", host="localhost", database='hmbank')
        my_cursor = db.cursor()
        return my_cursor, db
```

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.



```python
class Bankapp:
    @staticmethod
    def main():
        while True:
            print("Welcome to HM Bank")
            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 = None
            try:
                choice = int(input("Enter the choice"))
            except:
                print("Enter valid input")
                continue
            if choice==1:
                print("Enter account type : savings / current")
                type = input()
                if type == "savings":
                    account = Savings(accountbalance=500)
                elif type == "current":
                    account = Current(accountbalance=500)
                else:
                    print("wrong input try again")
                    continue
                print("Account successfully created")
            elif choice == 2:
                acc = int(input("Enter accountnumber"))
                amt = int(input("Enter amount to deposit"))
                BankServiceProviderImpl.deposit(acc, amt)
                print("Deposit Successful")
```



```
elif choice == 3:
    acc = int(input("Enter accountnumber"))
    amt = int(input("Enter amount to deposit"))
    amt = int(input("Enter amount to withdraw"))
    BankServiceProviderImpl.withdraw(acc, amt)
elif choice == 4:
    acc = int(input("Enter accountnumber"))
    BankServiceProviderImpl.get_account_balance(acc)
elif choice == 5:
    fromacc = int(input("Enter from accountnumber"))
    toacc = int(input("Enter to accountnumber"))
    amt = int(input("Enter amount"))
    BankServiceProviderImpl.transfer(fromacc, toacc, amt)
elif choice == 6:
    acc = int(input("Enter accountnumber"))
    BankServiceProviderImpl.getAccountDetails(acc)
elif choice == 7:
    customer = int(input("Enter customerid"))
    BankServiceProviderImpl.listaccounts(customer)
elif choice == 8:
    acc = int(input("Enter accountnumber"))
    f = int(input("Enter fromdate"))
    t = int(input("Enter todate"))
    BankServiceProviderImpl.getTransactions(acc, f, t)
elif choice == 9:
    print("Thanks fro coming")
    break
else:
    try:
        raise NullPointerException
    except NullPointerException:
        print("Invalid Input")
        continue
```