ASSIGNMENT

BANKING SYSTEM

Name: Maddaka Tejaswini

ASSIGNMENT: BANKING SYSTEM

Task 1: Conditional Statements

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

□ Credit Score must be above 700.

□ Annual Income must be at least $50,000.

Tasks:

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

```
credit_score=float(input('Enter the credit score of a customer : '))
annual_income=float(input('Enter the annual income of the customer : '))
if credit_score>700:
    if annual_income>=50000:
        print('Eligible for a loan')
    else:
        print('Ineligible for a lone due to low annual income')
else:
    print('Ineligible for a loan')
output:
```

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.

```
def atm_transaction():
    balance = float(input("Enter your current balance: "))
    print("\nATM Options:")
    print("1. Check Balance")
    print("2. Withdraw")
    print("3. Deposit")

    option = int(input("\nEnter the option number: "))

    if option == 1:
        print(f"\nYour current balance is: {balance}")
    elif option == 2:
        withdraw_amount = float(input("Enter the amount to withdraw: "))

        # Check if withdrawal amount is in multiples of 100 or 500
        if withdraw_amount % 100 != 0 and withdraw_amount % 500 != 0:
            print("Error: The withdrawal amount must be in multiples of 100 or 
500.")
        # Check if the balance is sufficient for withdrawal
        elif withdraw_amount > balance:
            print("Error: Insufficient balance for this withdrawal.")
        else:
            balance -= withdraw_amount
            print(f"Success: You have withdrawn {withdraw_amount}. Your new 
balance is {balance}.")

    elif option == 3:
        deposit_amount = float(input("Enter the amount to deposit: "))
        balance += deposit_amount
        print(f"Success: You have deposited {deposit_amount}.\n Your new 
balance is {balance}.")

    else:
        print("Invalid option selected. Please try again.")

atm_transaction()
```

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.

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

    for i in range(1, num_customers + 1):
        print(f"\nCustomer {i}:")
        initial_balance = float(input("Enter the initial balance: "))
        annual_interest_rate = float(input("Enter the annual interest rate (in
%): "))
        years = int(input("Enter the number of years: "))

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

        print(f"Future balance for Customer {i}: {future_balance:.2f}")

calculate_compound_interest()
```

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.

```
accounts = {
    "12345": 1500.50,
    "67890": 2500.00,
    "54321": 5000.75,
    "98765": 10000.00
}

def check_account_balance():
    while True:

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

```
        if account_number.lower() == 'exit':
            print("Thank you for using the bank system.")
            break

        if account_number in accounts:
            print(f"Account Number: {account_number}")
            print(f"Your balance is: {accounts[account_number]:.2f}\n")
        else:
            print("Invalid account number. Please try again.\n")

check_account_balance()
```

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.

```
def validate_password():

    password = input("Create a password for your bank account: ")

    if len(password) < 8:
        print("Error: Password must be at least 8 characters long.")
        return
    if not any(char.isupper() for char in password):
        print("Error: Password must contain at least one uppercase letter.")
        return
    if not any(char.isdigit() for char in password):
        print("Error: Password must contain at least one digit.")
        return
    print("Password is valid!")

validate_password()
```

Task 6: Password Validation

Create a program that maintains a list of bank transactions (deposits and withdrawals) for a customer.

Use a while loop to allow the user to keep adding transactions until they choose to exit. Display the

transaction history upon exit using looping statements.

```python
def bank_transactions():
    transactions = []
    balance = 0

    while True:
        print("\nSelect a transaction type:")
        print("1. Deposit")
        print("2. Withdraw")
        print("3. Exit and show transaction history")

        choice = input("Enter your choice : ")

        if choice == "1":

            deposit_amount = float(input("Enter deposit amount: "))
            transactions.append(f"Deposited: {deposit_amount}")
            balance += deposit_amount
            print(f"Successfully deposited {deposit_amount}. New balance: {balance}")

        elif choice == "2":

            withdraw_amount = float(input("Enter withdrawal amount: "))

            if withdraw_amount > balance:
                print("Error: Insufficient balance for this withdrawal.")
            else:
                transactions.append(f"Withdrew: {withdraw_amount}")
                balance -= withdraw_amount
                print(f"Successfully withdrew {withdraw_amount}. New balance: {balance}")

        elif choice == "3":

            print("\nTransaction History:")
            for transaction in transactions:
                print(transaction)
            print(f"Final Balance: {balance}")
            break

        else:
            print("Invalid option. Please try again.")

bank_transactions()
```

OOPS, Collections and Exception Handling

Task 7: Class & Object

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

• Attributes

o Customer ID

o First Name

o Last Name

o Email Address

o Phone Number

o Address

• Constructor and Methods

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

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

• Attributes

o Account Number

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

o Account Balance

• Constructor and Methods

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

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

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

-

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

-

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

withdraw amount only if there is sufficient fund else display insufficient

balance.

-

calculate_interest(): method for calculating interest amount for the available

balance. interest rate is fixed to 4.5%

•

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

main method:

o create object for account class by calling parameter constructor.

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

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

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

accounts.

account.py

```
class Account:
    def __init__(self, account_number=None, account_type=None, balance=0.0):
        self.__account_number = account_number
        self.__account_type = account_type
        self.__balance = balance

    # Getters
    def get_account_number(self):
        return self.__account_number

    def get_account_type(self):
        return self.__account_type

    def get_balance(self):
        return self.__balance

    # Setters
    def set_account_number(self, account_number):
        self.__account_number = account_number

    def set_account_type(self, account_type):
        self.__account_type = account_type
    def set_balance(self, balance):
        self.__balance = balance

    def print_account_info(self):
        print(f"Account Number: {self.__account_number}")
        print(f"Account Type: {self.__account_type}")
        print(f"Balance: {self.__balance}")

    def deposit(self, amount):
        if amount > 0:
            self.__balance += amount
            print(f"Deposited amount: {amount}, new balance: {self.__balance}")
        else:
            print("Invalid deposit amount.")

    def withdraw(self, amount):
        if amount > self.__balance:
            print("Insufficient balance.")
        elif amount <= 0:
            print("Invalid withdrawal amount.")
        else:
            self.__balance -= amount
            print(f"Withdrew: {amount}, new balance: {self.__balance}")

    def calculate_interest(self):
        if self.__account_type == 'Savings':
            interest = self.__balance * 0.045
            self.__balance += interest
            print(f"Interest of {interest} added, new balance: {self.__balance}")
        else:
            print("Interest calculation is only available for Savings accounts.")
```

customer.py

```
class Customer:
    def __init__(self, customer_id=None, first_name=None, last_name=None, email=None, phone=None, address=None):
        self.__customer_id = customer_id
        self.__first_name = first_name
        self.__last_name = last_name
        self.__email = email
        self.__phone = phone
        self.__address = address
```

```python
# Getters
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_email(self):
    return self.__email

def get_phone(self):
    return self.__phone

def get_address(self):
    return self.__address

# Setters
def set_customer_id(self, customer_id):
    self.__customer_id = customer_id

def set_first_name(self, first_name):
    self.__first_name = first_name

def set_last_name(self, last_name):
    self.__last_name = last_name

def set_email(self, email):
    self.__email = email

def set_phone(self, phone):
    self.__phone = phone

def set_address(self, address):
    self.__address = address

def print_customer_info(self):
    print(f"Customer ID: {self.__customer_id}")
    print(f"First Name: {self.__first_name}")
    print(f"Last Name: {self.__last_name}")
    print(f"Email: {self.__email}")
    print(f"Phone: {self.__phone}")
    print(f"Address: {self.__address}")
```

bankmain.py

```python
from account import Account
class Bank:
    def main(self):
        account = Account("411456213256", "Savings", 500.0)    #parameterized
constructor
        account.print_account_info()
        account.deposit(150.0)
        account.withdraw(1000.0)
        account.calculate_interest()
        account.print_account_info()


#Bank class main method
if __name__ == "__main__":
    bank = Bank()
    bank.main()
```

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.

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).

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.

account.py

```
class Account:
    def __init__(self, account_number=None, account_type=None, balance=0.0):
        self.__account_number = account_number
        self.__account_type = account_type
        self.__balance = balance

     # Getters
    def get_account_number(self):
        return self.__account_number

    def get_account_type(self):
        return self.__account_type

    def get_balance(self):
        return self.__balance

    # Setters
    def set_account_number(self, account_number):
        self.__account_number = account_number

    def set_account_type(self, account_type):
        self.__account_type = account_type

    def set_balance(self, balance):
        self.__balance = balance

    def print_account_info(self):
        print(f"Account Number: {self.__account_number}")
        print(f"Account Type: {self.__account_type}")
        print(f"Balance: {self.__balance}")

    # Overloaded deposit methods
    def deposit(self, amount):
        if isinstance(amount, (int, float)) and amount > 0:
            self.__balance += amount
            print(f"Deposited amount: {amount}, new balance:
{self.__balance}")
        else:
            print("Invalid deposit amount.")

    # Overloaded withdraw methods
    def withdraw(self, amount):
        if isinstance(amount, (int, float)):
            if amount > self.__balance:
                print("Insufficient balance.")
            elif amount <= 0:
                print("Invalid withdrawal amount.")
            else:
                self.__balance -= amount
                print(f"Withdrew: {amount}, new balance: {self.__balance}")
        else:
            print("Invalid withdrawal amount.")

    def calculate_interest(self):
        if self.__account_type == 'Savings':
            interest = self.__balance * 0.045
            self.__balance += interest
            print(f"Interest of {interest} added, new balance:
{self.__balance}")
        else:
            print("Interest calculation is only available for Savings
accounts.")
```

current_account.py

```
from account import Account
class CurrentAccount(Account):
    OVERDRAFT_LIMIT = 100 # Set a constant overdraft limit

    def __init__(self, account_number, balance):
        super().__init__(account_number, 'Current', balance)

    def withdraw(self, amount):

        if amount > self.get_balance() + self.OVERDRAFT_LIMIT:
            print("Exceeds overdraft limit. Insufficient balance.")
        else:
            new_balance = self.get_balance() - amount
            self.set_balance(new_balance)
            print(f"Withdrew: {amount}, New Balance: {self.get_balance()}")
```

savings_account.py

```
from account import Account
class SavingsAccount(Account):
    def __init__(self, account_number, balance, interest_rate=0.045):
        super().__init__(account_number, 'Savings', balance)
        self.__interest_rate = interest_rate

    def calculate_interest(self):
        interest = self.get_balance() * self.__interest_rate
        self.set_balance(self.get_balance() + interest)
        print(f"Interest of {interest} added, new balance:
{self.get_balance()}")
```

```
bankmain.py
from savings_account import SavingsAccount
from current_account import CurrentAccount
class Bank:
    def main(self):
        print("Welcome to the Banking System!")
        account_type = input("Choose account type (Savings/Current):
").strip().lower()

        if account_type == 'savings':
            account_number = input("Enter account number: ")
            balance = float(input("Enter initial balance: "))
            account = SavingsAccount(account_number, balance)
        elif account_type == 'current':
            account_number = input("Enter account number: ")
            balance = float(input("Enter initial balance: "))
            account = CurrentAccount(account_number, balance)
        else:
            print("Invalid account type.")
            return

        while True:
            print("\nMenu:")
            print("1. Deposit")
            print("2. Withdraw")
            print("3. Calculate Interest (for Savings Account only)")
            print("4. Show Account Info")
            print("5. Exit")
            choice = input("Enter your choice: ")

            if choice == '1':
                amount = float(input("Enter amount to deposit: "))
                account.deposit(amount)
            elif choice == '2':
                amount = float(input("Enter amount to withdraw: "))
                account.withdraw(amount)
            elif choice == '3':
                if isinstance(account, SavingsAccount):
                    account.calculate_interest()
                else:
                    print("Interest calculation is only available for Savings
accounts.")
            elif choice == '4':
                account.print_account_info()
            elif choice == '5':
                print("Exiting the Banking System. ")
                break
            else:
                print("Invalid choice. Please try again.")


# Main execution
if __name__ == "__main__":
    bank = Bank()
    bank.main()
```

Task 9: Abstraction

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

- Attributes:

o Account number.

o Customer name.
o Balance.
- Constructors:
o Implement default constructors and overload the constructor with Account attributes, generate getter and setter, print all information of attribute methods
for the attributes.
- Abstract methods:
o **deposit(amount: float):** Deposit the specified amount into the account.
o **withdraw(amount: float):** Withdraw the specified amount from the account (implement error handling for insufficient funds).
o **calculate_interest():** Abstract method for calculating interest.

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.
o *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.

savings_account.py

```
from bankaccount import BankAccount
```

```
class SavingsAccount(BankAccount):
    def __init__(self, account_number, customer_name, balance,
interest_rate=0.045):
        super().__init__(account_number, customer_name, balance)
        self.__interest_rate = interest_rate

    def deposit(self, amount):
        if amount > 0:
            new_balance = self.get_balance() + amount
            self.set_balance(new_balance)
            print(f"Deposited: {amount}, New Balance: {self.get_balance()}")
        else:
            print("Invalid deposit amount.")

    def withdraw(self, amount):
        if amount > self.get_balance():
            print("Insufficient balance. Can not withdraw the  amount you
entered.")
        else:
            new_balance = self.get_balance() - amount
            self.set_balance(new_balance)
            print(f"Withdrew: {amount}, New Balance: {self.get_balance()}")

    def calculate_interest(self):
        interest = self.get_balance() * self.__interest_rate
        self.set_balance(self.get_balance() + interest)
        print(f"Interest of {interest} added, New Balance:
{self.get_balance()}")
```

current_account.py

```
from bankaccount import BankAccount

class CurrentAccount(BankAccount):
    OVERDRAFT_LIMIT = 1000  #  a constant overdraft limit

    def __init__(self, account_number, customer_name, balance):
        super().__init__(account_number, customer_name, balance)

    def deposit(self, amount):
        if amount > 0:
            new_balance = self.get_balance() + amount
            self.set_balance(new_balance)
            print(f"Deposited: {amount}, New Balance: {self.get_balance()}")
        else:
            print("Invalid deposit amount.")
    def withdraw(self, amount):
        # Check if the withdrawal amount exceeds the total available balance
(including overdraft limit)
        if amount > self.get_balance() + self.OVERDRAFT_LIMIT:
            print("Exceeds overdraft limit. Insufficient balance.")
        else:
            # Deduct the amount from the balance
            new_balance = self.get_balance() - amount
            self.set_balance(new_balance)  # Update the balance
            print(f"Withdrew: {amount}, New Balance: {self.get_balance()}")

    def calculate_interest(self):
        print("Current Account does not earn interest.")
```

bankaccount.py

```
from abc import ABC, abstractmethod

class BankAccount(ABC):
    def __init__(self, account_number=None, customer_name=None, balance=0.0):
        self.__account_number = account_number
        self.__customer_name = customer_name
        self.__balance = balance

    # Getters
    def get_account_number(self):
        return self.__account_number

    def get_customer_name(self):
        return self.__customer_name

    def get_balance(self):
        return self.__balance

    # Setters
    def set_balance(self, amount):
        if amount >= 0:
            self.__balance = amount
        else:
            print("Invalid balance value.")

    def print_account_info(self):
        print(f"Account Number: {self.__account_number}")
        print(f"Customer Name: {self.__customer_name}")
        print(f"Balance: {self.__balance}")

    @abstractmethod
    def deposit(self, amount):
        pass

    @abstractmethod
    def withdraw(self, amount):
        pass

    @abstractmethod
    def calculate_interest(self):
        pass
```

bankmain.py

```
from savings_account import SavingsAccount
from current_account import CurrentAccount

class Bank:
    def main(self):
        print("Welcome to the Banking System!")
        account_type = input("Choose account type (Savings/Current): 
").strip().lower()

        if account_type == 'savings':
            account_number = input("Enter account number: ")
            customer_name = input("Enter customer name: ")
            balance = float(input("Enter initial balance: "))
            account = SavingsAccount(account_number, customer_name, balance)
        elif account_type == 'current':
            account_number = input("Enter account number: ")
            customer_name = input("Enter customer name: ")
            balance = float(input("Enter initial balance: "))
            account = CurrentAccount(account_number, customer_name, balance)
        else:
            print("Invalid account type.")
            return

        while True:
            print("\nMenu:")
            print("1. Deposit")
            print("2. Withdraw")
            print("3. Calculate Interest (for Savings Account only)")
            print("4. Show Account Info")
            print("5. Exit")
            choice = input("Enter your choice: ")

            if choice == '1':
                amount = float(input("Enter amount to deposit: "))
                account.deposit(amount)
            elif choice == '2':
                amount = float(input("Enter amount to withdraw: "))
                account.withdraw(amount)
            elif choice == '3':
                if isinstance(account, SavingsAccount):
                    account.calculate_interest()
                else:
                    print("Interest calculation is only available for Savings 
accounts.")
            elif choice == '4':
                account.print_account_info()
            elif choice == '5':
                print("Exiting the Banking System.")
                break
            else:
                print("Invalid choice. Please try again.")


# Main execution
if __name__ == "__main__":
    bank = Bank()
    bank.main()
```

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:

o 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:

o 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.

Account.py

```python
class Account:
    def __init__(self, account_number, account_type, balance, customer):
        self.__account_number = account_number
        self.__account_type = account_type
        self.__balance = balance
        self.__customer = customer  # Has-a relationship with Customer

    # Getters and Setters
    def get_account_number(self):
        return self.__account_number

    def get_account_type(self):
        return self.__account_type

    def get_balance(self):
        return self.__balance

    def get_customer(self):
        return self.__customer

    def deposit(self, amount):
        self.__balance += amount
        print(f"Deposited: {amount}. New Balance: {self.__balance}")
        return self.__balance

    def withdraw(self, amount):
        if amount <= self.__balance:
            self.__balance -= amount
            print(f"Withdrew: {amount}. New Balance: {self.__balance}")
            return self.__balance
        else:
            print("Insufficient balance.")
            return self.__balance

    def print_account_info(self):
        print(f"Account Number: {self.__account_number}")
        print(f"Account Type: {self.__account_type}")
        print(f"Balance: {self.__balance}")
        self.__customer.print_info()  # Print customer info
```

customer.py

```
class Customer:
    def __init__(self, customer_id, first_name, last_name, email, phone,
address):
        self.__customer_id = customer_id
        self.__first_name = first_name
        self.__last_name = last_name
        self.__email = email
        self.__phone = phone
        self.__address = address

    # Getters and Setters
    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_email(self):
        return self.__email

    def get_phone(self):
        return self.__phone

    def get_address(self):
        return self.__address

    def print_info(self):
        print(f"Customer ID: {self.__customer_id}")
        print(f"First Name: {self.__first_name}")
        print(f"Last Name: {self.__last_name}")
        print(f"Email: {self.__email}")
        print(f"Phone: {self.__phone}")
        print(f"Address: {self.__address}")
```

bank.py

```
from account import Account
class Bank:
    account_counter = 1001  # Static variable for account number generation

    def __init__(self):
        self.__accounts = {}

    def create_account(self, customer, acc_type, balance):
        account_number = Bank.account_counter
        Bank.account_counter += 1  # Increment account number for the next
account
        account = Account(account_number, acc_type, balance, customer)
        self.__accounts[account_number] = account
        print(f"Account created for {customer.get_first_name()}
{customer.get_last_name()} with Account Number: {account_number}")

    def get_account_balance(self, account_number):
        if account_number in self.__accounts:
            return self.__accounts[account_number].get_balance()
        else:
            print("Account not found.")
            return None

    def deposit(self, account_number, amount):
        # Check for account existence immediately
        if account_number not in self.__accounts:
            print("Account not found.")
            return None

        # If account exists, proceed with deposit
        current_balance = self.__accounts[account_number].deposit(amount)
        return current_balance

    def withdraw(self, account_number, amount):
        # Check for account existence immediately
        if account_number not in self.__accounts:
            print("Account not found.")
            return None

        current_balance = self.__accounts[account_number].withdraw(amount)
        return current_balance

    def transfer(self, from_account_number, to_account_number, amount):
        if from_account_number not in self.__accounts:
            print("Sender account not found.")
            return

        if to_account_number not in self.__accounts:
            print("Receiver account not found.")
            return

        if self.__accounts[from_account_number].get_balance() < amount:
            print("Insufficient balance to transfer.")
            return

        # Proceed with withdrawal and deposit if checks pass
        print(f"Transferred {amount} from Account {from_account_number} to
Account {to_account_number}.")
        self.__accounts[from_account_number].withdraw(amount)


    def get_account_details(self, account_number):
        if account_number in self.__accounts:
            account = self.__accounts[account_number]
            account.print_account_info()
        else:
            print("Account not found.")
```

bankapp_main.py

```
from bank import Bank
from customer import Customer

class BankApp:
    def main(self):
        bank = Bank()
        while True:
            print("\nMenu:")
            print("1. Create Account")
            print("2. Deposit")
            print("3. Withdraw")
            print("4. Get Balance")
            print("5. Transfer")
            print("6. Get Account Details")
            print("7. Exit")

            choice = input("Enter your choice: ")

            if choice == '1':
                customer_id = input("Enter Customer ID: ")
                first_name = input("Enter First Name: ")
                last_name = input("Enter Last Name: ")
                email = input("Enter Email Address: ")
                phone = input("Enter Phone Number: ")
                address = input("Enter Address: ")
                customer = Customer(customer_id, first_name, last_name, email,
phone, address)

                acc_type = input("Enter Account Type (Savings/Current): ")
                balance = float(input("Enter Initial Balance: "))

                bank.create_account(customer, acc_type, balance)

            elif choice == '2':
                acc_no = int(input("Enter Account Number: "))
            # Check for account existence immediately before proceeding
                if acc_no not in bank._Bank__accounts:  # Accessing private
member for checking
                    print("Account not found.")
                    continue
                amount = float(input("Enter Deposit Amount: "))
                bank.deposit(acc_no, amount)

            elif choice == '3':
                acc_no = int(input("Enter Account Number: "))
                    # Check for account existence immediately before
proceeding
                if acc_no not in bank._Bank__accounts:  # Accessing private
member for checking
                    print("Account not found.")
                    continue
                amount = float(input("Enter Withdrawal Amount: "))
                bank.withdraw(acc_no, amount)

            elif choice == '4':
                acc_no = int(input("Enter Account Number: "))
                balance = bank.get_account_balance(acc_no)
                if balance is not None:
                    print(f"Current Balance: {balance}")

            elif choice == '5':
                from_acc = int(input("Enter From Account Number: "))

                if from_acc not in bank._Bank__accounts:  # Accessing private
member for checking
                    print("Sender account not found.")
                    continue
                to_acc = int(input("Enter To Account Number: "))
                # Check receiver account existence
                if to_acc not in bank._Bank__accounts:  # Accessing private
member for checking
                    print("Receiver account not found.")
                    continue
                amount = float(input("Enter Transfer Amount: "))
                bank.transfer(from_acc, to_acc, amount)


            elif choice == '6':
                acc_no = int(input("Enter Account Number: "))
                bank.get_account_details(acc_no)

            elif choice == '7':
                print("Exiting the Banking System.")
                break

            else:
                print("Invalid choice. Please try again.")

# Main execution
if __name__ == "__main__":
    app = BankApp()
    app.main()
```

TASK 11,12,13,14

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

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

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.

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.

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

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

Account.py

```
class Account:
    def __init__(self, account_number, account_type, balance):

        self.account_number = account_number
        self.account_type = account_type
        self.balance = balance

    def deposit(self, amount: float):
        self.balance += amount
        print(f"Deposited {amount}. New balance: {self.balance}")

    def deposit_int(self, amount: int):
        self.balance += amount
        print(f"Deposited {amount}. New balance: {self.balance}")

    # Overloaded methods for withdraw
    def withdraw(self, amount: float):
        if self.balance >= amount:
            self.balance -= amount
            print(f"Withdrawn {amount}. New balance: {self.balance}")
        else:
            raise Exception("Insufficient Balance")

    def withdraw_int(self, amount: int):
        if self.balance >= amount:
            self.balance -= amount
            print(f"Withdrawn {amount}. New balance: {self.balance}")
        else:
            raise Exception("Insufficient Balance")

customer.py
class Customer:
    def __init__(self, customer_id,first_name, last_name, dob, email,
phone_number, address):

        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_full_name(self):
        return f"{self.first_name} {self.last_name}"
    def print_info(self):
        print(f"Name: {self.get_full_name()}, Email: {self.email}, Phone: {self.phone_number}, Address: {self.address}")
```

savings_account.py

```
import sys
import os
sys.path.append(os.path.dirname(os.path.dirname(os.path.abspath(__file__))))

from entity.account import Account

class SavingsAccount(Account):
    def __init__(self, account_number, balance, interest_rate=4.5):
        super().__init__(account_number, "Savings", balance)
        self.interest_rate = interest_rate

    def calculate_interest(self):
        return self.balance * (self.interest_rate / 100)
```

current_account.py

```
import sys
import os
sys.path.append(os.path.dirname(os.path.dirname(os.path.abspath(__file__))))

from entity.account import Account

class CurrentAccount(Account):
    OVERDRAFT_LIMIT = 1000  # Assuming $1000 overdraft limit for current accounts

    def __init__(self, account_number, balance):
        super().__init__(account_number, "Current", balance)

    def withdraw(self, amount):
        if self.balance - amount >= -self.OVERDRAFT_LIMIT:
            self.balance -= amount
            print(f"Withdrawn {amount}. New balance: {self.balance}")
        else:
            raise Exception(f"Overdraft limit exceeded! Cannot withdraw {amount}.")
```

bank.py

```
import sys
import os
sys.path.append(os.path.dirname(os.path.dirname(os.path.abspath(__file__))))

from dao.BankServiceProviderImpl import BankServiceProviderImpl
from exception.invalidAcctExcp import InvalidAccountException

class Bank:
    def __init__(self):
        self.service_provider = BankServiceProviderImpl()

    def create_account(self, customer, account_type, initial_balance):
        """
        Creates a new bank account for the given customer.
        """
        account_id = self.service_provider.create_account(customer,
account_type, initial_balance)
        print(f"Account created successfully with Account ID: {account_id}")

    def get_account_balance(self, account_number):
        """
        Retrieves the balance of a specific account.
        """
        try:
            account =
self.service_provider.get_account_details(account_number)
            print(f"Account Balance: {account['balance']}")
        except InvalidAccountException as e:
            print(e)

    def deposit(self, account_number, amount):
        """
        Deposit a specific amount into an account.
        """
        try:
            self.service_provider.deposit(account_number, amount)
            print(f"Deposited {amount} successfully.")
        except InvalidAccountException as e:
            print(e)

    def withdraw(self, account_number, amount):
        """
        Withdraw a specific amount from an account.
        """
        try:
            self.service_provider.withdraw(account_number, amount)
            print(f"Withdrew {amount} successfully.")
        except Exception as e:
            print(e)

    def transfer(self, from_account, to_account, amount):
        """
        Transfer funds from one account to another.
        """
        try:
            # Withdraw from the source account
            self.withdraw(from_account, amount)
            # Deposit into the target account
            self.deposit(to_account, amount)
            print(f"Transferred {amount} from account {from_account} to
account {to_account}.")
        except Exception as e:
            print(e)

    def get_account_details(self, account_number):
        """
        Retrieves and prints details of a specific account.
        """
        try:
            account =
self.service_provider.get_account_details(account_number)
            print(f"Account Details: {account}")
        except InvalidAccountException as e:
            print(e)
```

IBankServiceProvider.py

```
from abc import ABC, abstractmethod

class IBankServiceProvider(ABC):
    @abstractmethod
    def create_account(self, customer, account_number, account_type, balance):
        pass

    @abstractmethod
    def list_accounts(self):
        pass

    @abstractmethod
    def calculate_interest(self, account_number: int):
        pass
```

ICustomerServiceProvider.py

```
from abc import ABC, abstractmethod


class ICustomerServiceProvider(ABC):
    @abstractmethod
    def get_account_balance(self, account_number: int):
        pass

    @abstractmethod
    def deposit(self, account_number: int, amount: float):
        pass
    @abstractmethod
    def withdraw(self, account_number: int, amount: float):
        pass

    @abstractmethod
    def transfer(self, from_account: int, to_account: int, amount: float):
        pass

    @abstractmethod
    def get_account_details(self, account_number: int):
        pass
```

bankServiceProviderImpl.py

```
import sys
import os
sys.path.append(os.path.dirname(os.path.dirname(os.path.abspath(__file__))))


from dao.ICustomerServiceProvider import ICustomerServiceProvider
from dao.IBankServiceProvider import IBankServiceProvider
from dao.CustomerServiceProviderImpl import CustomerServiceProviderImpl
from util.DBConnection import DBConnection
from exception.invalidAcctExcp import InvalidAccountException
from decimal import Decimal
class BankServiceProviderImpl(CustomerServiceProviderImpl, 
IBankServiceProvider):
    connection=None
    def __init__(self, branch_name: str, branch_address: str):
        super().__init__()
        self.accountList = []  # To store the account objects
        self.branchName = branch_name
        self.branchAddress = branch_address
        self.connection=DBConnection.getConnection()

    def create_account(self, customer, account_number, account_type, balance):
        """
        Create a new bank account for the given customer and add it to 
accountList.
        """
        # We use the CustomerServiceProviderImpl to create the account in the 
database
        conn = DBConnection.getConnection()
        cursor = conn.cursor()

        # Insert Customer
        cursor.execute('''
            INSERT INTO dbo.Customers (first_name, last_name, DOB, email, 
phone_number, address)
```

```
            VALUES (?, ?, ?, ?, ?, ?)
        ''', (customer.first_name, customer.last_name, customer.dob, customer.email, customer.phone_number, customer.address))
        customer_id = self.get_customer_id(cursor,customer.first_name)
        account_id = f"ACC{account_number}"

        # Insert Account
        cursor.execute('''
            INSERT INTO Accounts (account_id,customer_id, account_type, balance)
            VALUES (?,?, ?, ?)
        ''', (account_id,customer_id, account_type, balance))
        account_id = self.get_account_id(cursor,customer.customer_id)

        # Add account object to accountList
        self.accountList.append({
            "account_id": account_id,
            "account_type": account_type,
            "balance": balance
        })

        conn.commit()
        conn.close()
        print(f"Account {account_number} created for {customer.first_name} {customer.last_name}")

    def list_accounts(self, customer_id: int):
        """
        List all accounts for a specific customer in the bank.
        """
        conn = DBConnection.getConnection()
        cursor = conn.cursor()

        cursor.execute('''
            SELECT a.account_id, a.account_type, a.balance
            FROM Accounts a
            WHERE a.customer_id = ?
        ''', (customer_id,))
        accounts = cursor.fetchall()

        if not accounts:
            print("No accounts available for this customer.")
        else:
            print(f"Accounts for Customer ID {customer_id}:")
            for account in accounts:
                print(f"Account ID: {account[0]}, Type: {account[1]}, Balance: {account[2]}")
        conn.close()


    def calculate_interest(self, account_number: int):
        """
        Calculate the interest for a specific account (only for savings
accounts).
        """
        conn = DBConnection.getConnection()
        cursor = conn.cursor()
        cursor.execute('SELECT account_type, balance FROM Accounts WHERE
account_id = ?', (account_number,))
        account = cursor.fetchone()

        if account is None:
            raise InvalidAccountException("Account not found.")

        account_type, balance = account
        print(f"Account type: {account_type}, Balance: {balance}")

        if account_type == 'savings':
            interest_rate = Decimal('4.5')  # Assuming a fixed interest rate
            interest = Decimal(balance) * (interest_rate / Decimal('100'))
            print(f"Interest calculated for account {account_number}:
{interest}")
            return interest
        else:
            raise Exception("Interest calculation is only applicable for
savings accounts.")
```

CustomerServiceProviderImpl.py

```
import sys
import os
sys.path.append(os.path.dirname(os.path.dirname(os.path.abspath(__file__))))

from dao.ICustomerServiceProvider import ICustomerServiceProvider
from exception.invalidAcctExcp import InvalidAccountException
from util.DBConnection import DBConnection

class CustomerServiceProviderImpl(ICustomerServiceProvider):
    connection=None
    def __init__(self):
        self.connection=DBConnection.getConnection()



    def get_account_balance(self, account_number: int):
        """
        Retrieve the balance of an account given its account number.
        """
        conn = DBConnection.getConnection()
        cursor = conn.cursor()

        cursor.execute('SELECT balance FROM Accounts WHERE account_id=?', 
(account_number,))
        account = cursor.fetchone()

        if account is None:
            conn.close()
            raise InvalidAccountException("Account not found")

        balance = account[0]
        conn.close()
        print(f"Account ID: {account_number}, Balance: {balance}")
        return balance

    def deposit(self, account_number: int, amount: float):
        """
        Deposit the specified amount into the account.
        """
        conn = DBConnection.getConnection()
        cursor = conn.cursor()

        # Check if the account exists
        cursor.execute('SELECT balance FROM Accounts WHERE account_id = ?', 
(account_number,))
        result = cursor.fetchone()

        if result is None:
            conn.close()
            raise Exception(f"Account with ID {account_number} does not 
exist.")

        # Perform the deposit
        cursor.execute('UPDATE Accounts SET balance = balance + ? WHERE 
account_id = ?', (amount, account_number))
        conn.commit()

        # Fetch the updated balance
        cursor.execute('SELECT balance FROM Accounts WHERE account_id = ?', 
(account_number,))
        new_balance = cursor.fetchone()[0]

        conn.close()

        # Print success message
        print(f"Deposited successfully. New balance: {new_balance}")
        return new_balance


    def withdraw(self, account_number: int, amount: float):
        """
        Withdraw the specified amount from the account.
        """
        conn = DBConnection.getConnection()
        cursor = conn.cursor()

        cursor.execute('SELECT balance FROM Accounts WHERE account_id=?',
(account_number,))
        account = cursor.fetchone()

        if account is None:
            raise InvalidAccountException("Account not found")

        balance = account[0]
        if balance >= amount:
            cursor.execute('UPDATE Accounts SET balance = balance - ? WHERE
account_id=?', (amount, account_number))
            conn.commit()

            cursor.execute('SELECT balance FROM Accounts WHERE account_id=?',
(account_number,))
            new_balance = cursor.fetchone()[0]
            conn.close()
            print(f"Withdrawal successful. Updated balance: {new_balance}")
            return new_balance
        else:
            conn.close()
            raise Exception("Insufficient Balance")

    def transfer(self, from_account: int, to_account: int, amount: float):
        """
        Transfer money from one account to another.
        """
        self.withdraw(from_account, amount)
        self.deposit(to_account, amount)

    def get_account_details(self, account_number: int):
        """
        Retrieve account and customer details for the given account number.
        """
        conn = DBConnection.getConnection()
        cursor = conn.cursor()

        cursor.execute('''
            SELECT c.first_name, c.last_name, c.email, c.phone_number,
a.account_type, a.balance
            FROM Customers c
            JOIN Accounts a ON c.customer_id = a.customer_id
            WHERE a.account_id=?
        ''', (account_number,))
        account_details = cursor.fetchone()

        if account_details is None:
            raise InvalidAccountException("Account not found")

        conn.close()
        print(f"Account Details:\n"
          f"First Name: {account_details[0]}\n"
          f"Last Name: {account_details[1]}\n"
          f"Email: {account_details[2]}\n"
          f"Phone Number: {account_details[3]}\n"
          f"Account Type: {account_details[4]}\n"
          f"Balance: {account_details[5]}")

        return account_details
```

exception.py

```
class InvalidAccountException(Exception):
    def __init__(self, message):
        super().__init__(message)
```

DBConnection.py

```
import pyodbc
from util.PropertyUtil import PropertyUtil

class DBConnection:

    @staticmethod
    def getConnection():
        try:
            connection_string=PropertyUtil.getPropertyString()
            connection=pyodbc.connect(connection_string)
            print("Connected successfully")
            return connection
        except Exception as e:
            print(str(e) + '--Database is not connected--')
            return None
```

PropertyFile.txt

```
driver = {SQL Server}
```

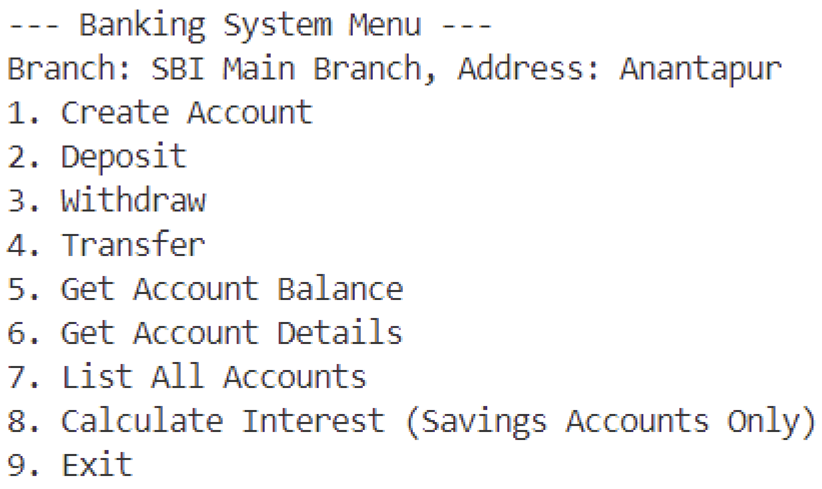
```
server = LAPTOP-Q72Q77L5\SQLEXPRESS
dbname = BankingSystem
trusted_connection = yes
```

PropertyUtil.py

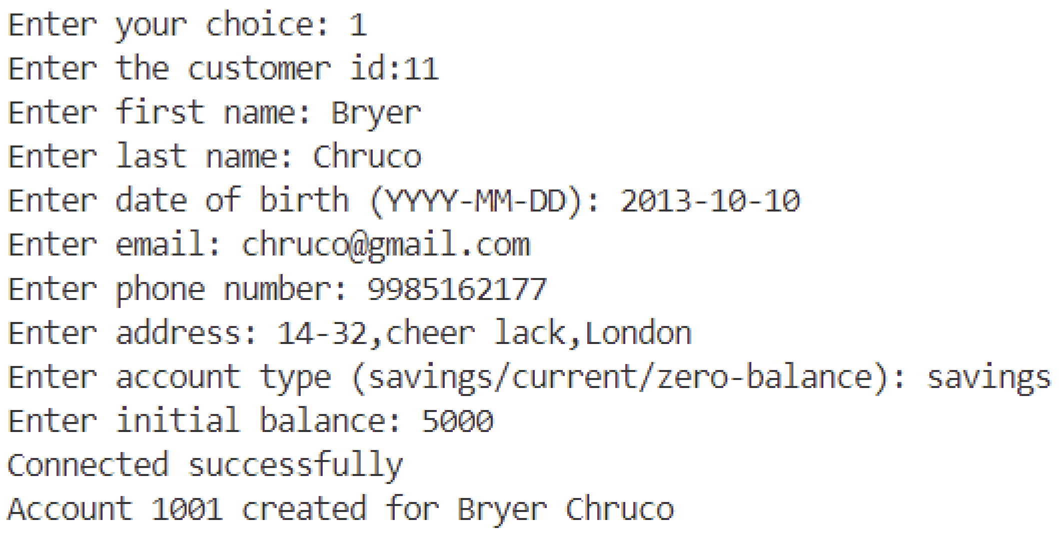
```python
class PropertyUtil:
    @staticmethod
    def getPropertyString(property_file_path=r"C:\Users\Asus\OneDrive\Desktop\BANKING_SYSTEM\util\PropertyFile.txt"):
        try:
            with open(property_file_path, 'r') as file:
                properties = {}
                for line in file:
                    if '=' in line:
                        key, value = line.strip().split('=', 1)  # Split by '=' only on the first occurrence
                        properties[key.strip()] = value.strip()

                # Create the connection string
                connection_string = f"DRIVER={{ODBC Driver 17 for SQL Server}};" \
                                    f"SERVER={properties['server']};" \
                                    f"DATABASE={properties['dbname']};" \
                                    f"Trusted_Connection={properties['trusted_connection']};"
                return connection_string
        except ValueError as ve:
            print('db is missing',ve)
        except Exception as e:
            print(f"Error reading property file: {e}")
            return None
```

BankApp.py

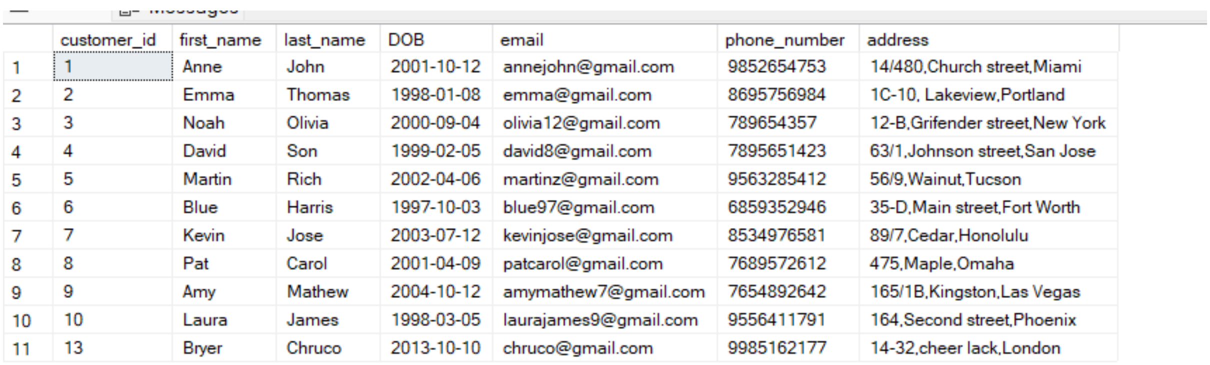
```python
import sys
import os
sys.path.append(os.path.dirname(os.path.dirname(os.path.abspath(__file__))))

from entity.customer import Customer
from entity.bank import Bank
from dao.ICustomerServiceProvider import ICustomerServiceProvider
from dao.IBankServiceProvider import IBankServiceProvider
from dao.CustomerServiceProviderImpl import CustomerServiceProviderImpl
from dao.BankServiceProviderImpl import BankServiceProviderImpl


def main():
    bank_service = BankServiceProviderImpl(branch_name="SBI Main Branch", branch_address="Anantapur")
    while True:
        print("\n--- Banking System Menu ---")
        print(f"Branch: {bank_service.branchName}, Address:
{bank_service.branchAddress}")
        print("1. Create Account")
        print("2. Deposit")
        print("3. Withdraw")
        print("4. Transfer")
        print("5. Get Account Balance")
        print("6. Get Account Details")
        print("7. List All Accounts")
        print("8. Calculate Interest (Savings Accounts Only)")
        print("9. Exit")

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

        if choice == 1:
            customer_id=input('Enter the customer id:')
            first_name = input("Enter first name: ")
            last_name = input("Enter last name: ")
            dob = input("Enter date of birth (YYYY-MM-DD): ")
            email = input("Enter email: ")
            phone_number = input("Enter phone number: ")
            address = input("Enter address: ")
            customer = Customer(customer_id,first_name, last_name, dob, email,
phone_number, address)

            account_type = input("Enter account type (savings/current/zero-
balance): ")
            balance = float(input("Enter initial balance: "))
            account_number = len(bank_service.accountList) + 1001  # Generate
account number
            bank_service.create_account(customer, account_number,
account_type, balance)

        elif choice == 2:
            account_number = int(input("Enter account number: "))
            amount = float(input("Enter amount to deposit: "))
            bank_service.deposit(account_number, amount)

        elif choice == 3:
            account_number = int(input("Enter account number: "))
            amount = float(input("Enter amount to withdraw: "))
            bank_service.withdraw(account_number, amount)

        elif choice == 4:
            from_account = int(input("Enter from account number: "))
            to_account = int(input("Enter to account number: "))
            amount = float(input("Enter amount to transfer: "))
            bank_service.transfer(from_account, to_account, amount)

        elif choice == 5:
            account_number = int(input("Enter account number: "))
            bank_service.get_account_balance(account_number)

        elif choice == 6:
            account_number = int(input("Enter account number: "))
            bank_service.get_account_details(account_number)

        elif choice == 7:
            customer_id = input("Enter Customer ID to list accounts: ")
            bank_service.list_accounts(customer_id)

        elif choice == 8:
            account_number = int(input("Enter account number: "))
            bank_service.calculate_interest(account_number)

        elif choice == 9:
            print("Exiting...")
            break
        else:
            print("Invalid option, please try again.")

if __name__ == "__main__":
    main()
outputs:
```



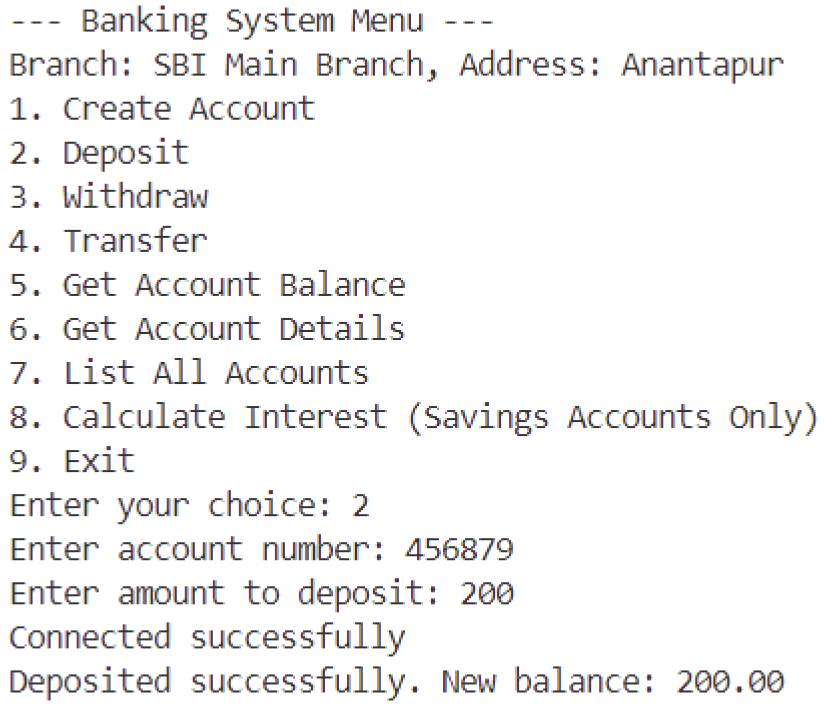
```
--- Banking System Menu ---
Branch: SBI Main Branch, Address: Anantapur
1. Create Account
2. Deposit
3. Withdraw
4. Transfer
5. Get Account Balance
6. Get Account Details
7. List All Accounts
8. Calculate Interest (Savings Accounts Only)
9. Exit
```



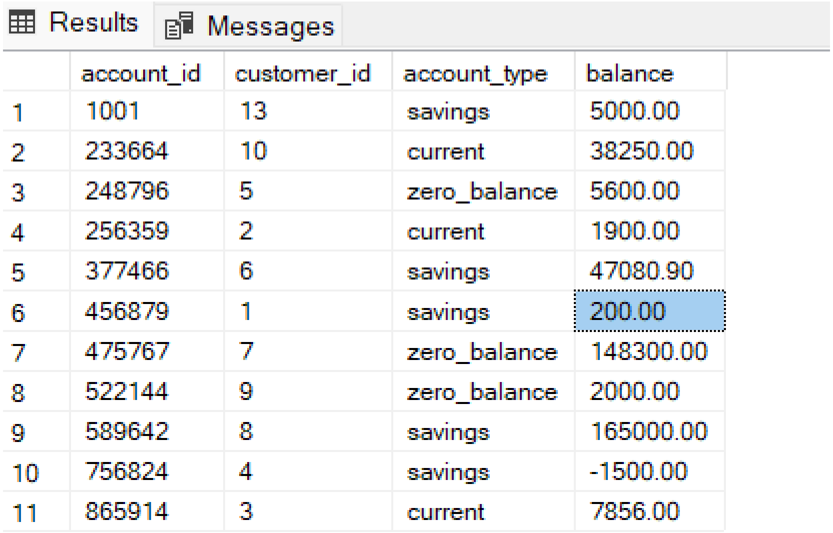
```
Enter your choice: 1
Enter the customer id:11
Enter first name: Bryer
Enter last name: Chruco
Enter date of birth (YYYY-MM-DD): 2013-10-10
Enter email: chruco@gmail.com
Enter phone number: 9985162177
Enter address: 14-32,cheer lack,London
Enter account type (savings/current/zero-balance): savings
Enter initial balance: 5000
Connected successfully
Account 1001 created for Bryer Chruco
```



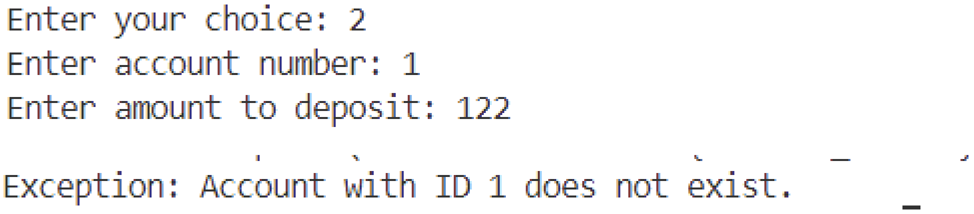
	customer_id	first_name	last_name	DOB	email	phone_number	address
1	1	Anne	John	2001-10-12	annejohn@gmail.com	9852654753	14/480,Church street,Miami
2	2	Emma	Thomas	1998-01-08	emma@gmail.com	8695756984	1C-10, Lakeview,Portland
3	3	Noah	Olivia	2000-09-04	olivia12@gmail.com	789654357	12-B,Grifender street,New York
4	4	David	Son	1999-02-05	david8@gmail.com	7895651423	63/1,Johnson street,San Jose
5	5	Martin	Rich	2002-04-06	martinz@gmail.com	9563285412	56/9,Wainut,Tucson
6	6	Blue	Harris	1997-10-03	blue97@gmail.com	6859352946	35-D,Main street,Fort Worth
7	7	Kevin	Jose	2003-07-12	kevinjose@gmail.com	8534976581	89/7,Cedar,Honolulu
8	8	Pat	Carol	2001-04-09	patcarol@gmail.com	7689572612	475,Maple,Omaha
9	9	Amy	Mathew	2004-10-12	amymathew7@gmail.com	7654892642	165/1B,Kingston,Las Vegas
10	10	Laura	James	1998-03-05	laurajames9@gmail.com	9556411791	164,Second street,Phoenix
11	13	Bryer	Chruco	2013-10-10	chruco@gmail.com	9985162177	14-32,cheer lack,London



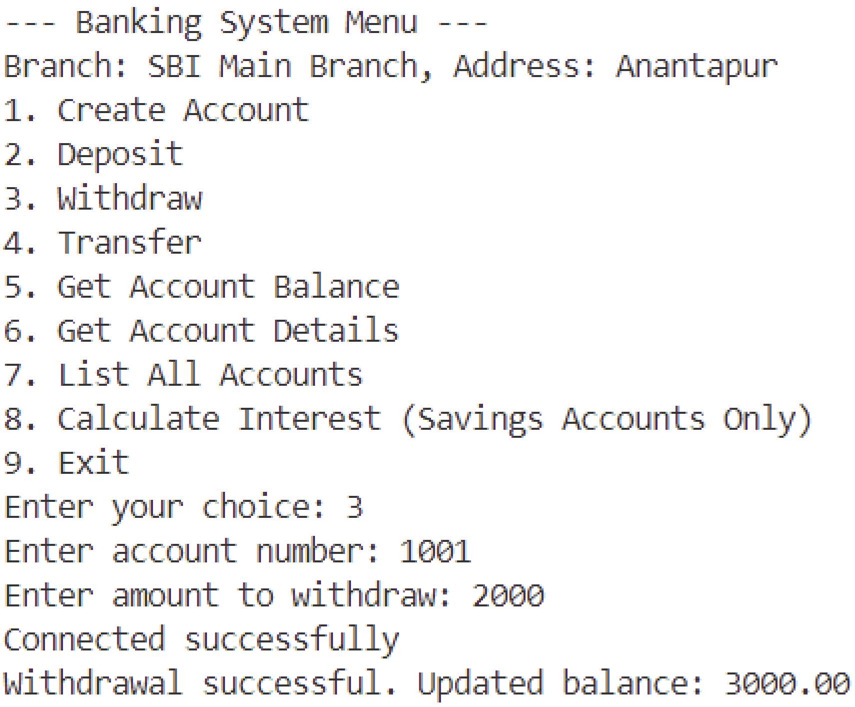
```
--- Banking System Menu ---
Branch: SBI Main Branch, Address: Anantapur
1. Create Account
2. Deposit
3. Withdraw
4. Transfer
5. Get Account Balance
6. Get Account Details
7. List All Accounts
8. Calculate Interest (Savings Accounts Only)
9. Exit
Enter your choice: 2
Enter account number: 456879
Enter amount to deposit: 200
Connected successfully
Deposited successfully. New balance: 200.00
```



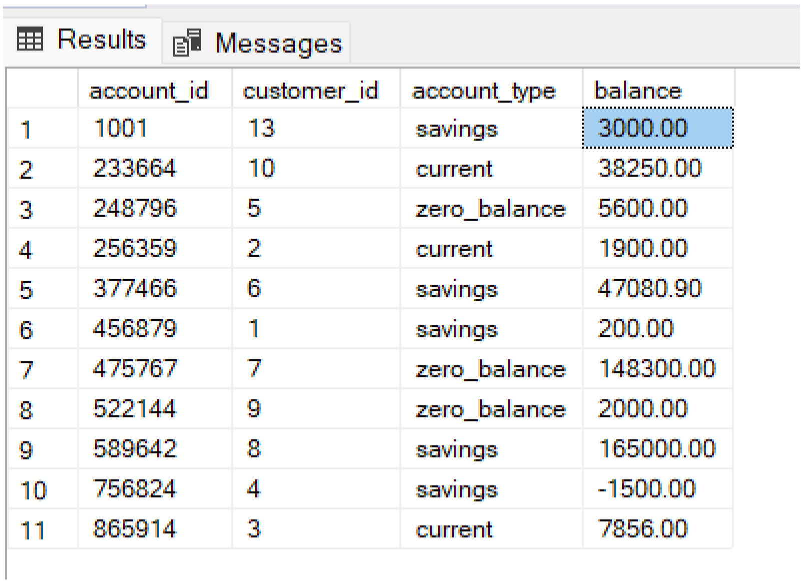
	account_id	customer_id	account_type	balance
1	1001	13	savings	5000.00
2	233664	10	current	38250.00
3	248796	5	zero_balance	5600.00
4	256359	2	current	1900.00
5	377466	6	savings	47080.90
6	456879	1	savings	200.00
7	475767	7	zero_balance	148300.00
8	522144	9	zero_balance	2000.00
9	589642	8	savings	165000.00
10	756824	4	savings	-1500.00
11	865914	3	current	7856.00



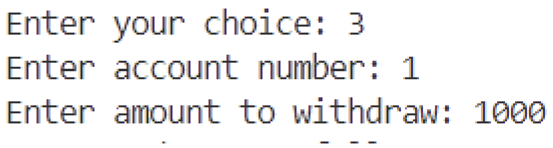
```
Enter your choice: 2
Enter account number: 1
Enter amount to deposit: 122

Exception: Account with ID 1 does not exist.
```



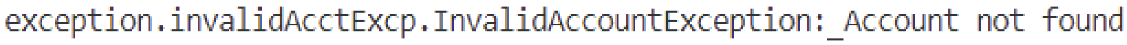
```
--- Banking System Menu ---
Branch: SBI Main Branch, Address: Anantapur
1. Create Account
2. Deposit
3. Withdraw
4. Transfer
5. Get Account Balance
6. Get Account Details
7. List All Accounts
8. Calculate Interest (Savings Accounts Only)
9. Exit
Enter your choice: 3
Enter account number: 1001
Enter amount to withdraw: 2000
Connected successfully
Withdrawal successful. Updated balance: 3000.00
```



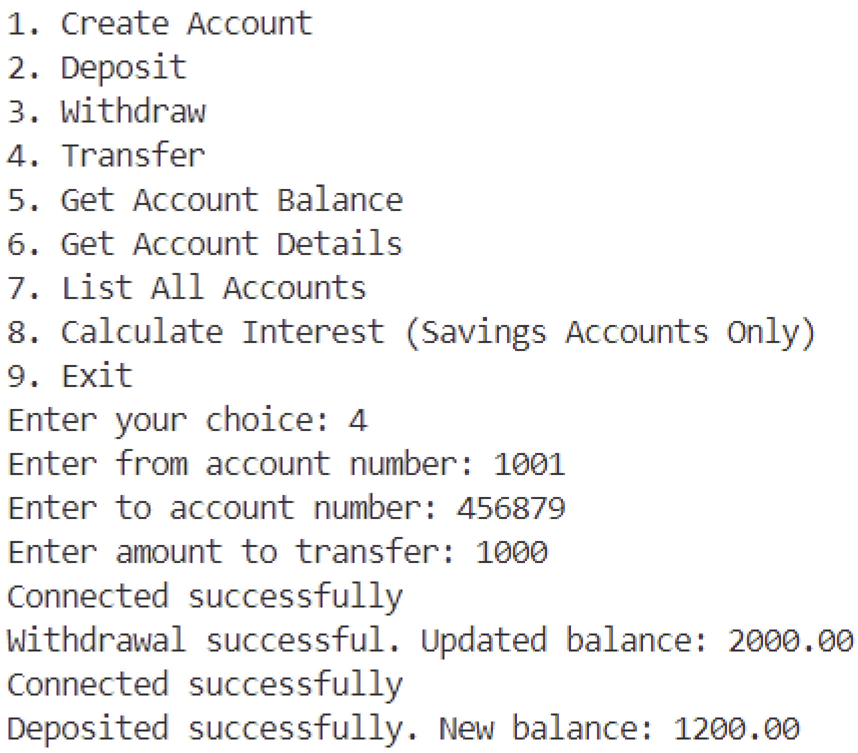
	account_id	customer_id	account_type	balance
1	1001	13	savings	3000.00
2	233664	10	current	38250.00
3	248796	5	zero_balance	5600.00
4	256359	2	current	1900.00
5	377466	6	savings	47080.90
6	456879	1	savings	200.00
7	475767	7	zero_balance	148300.00
8	522144	9	zero_balance	2000.00
9	589642	8	savings	165000.00
10	756824	4	savings	-1500.00
11	865914	3	current	7856.00



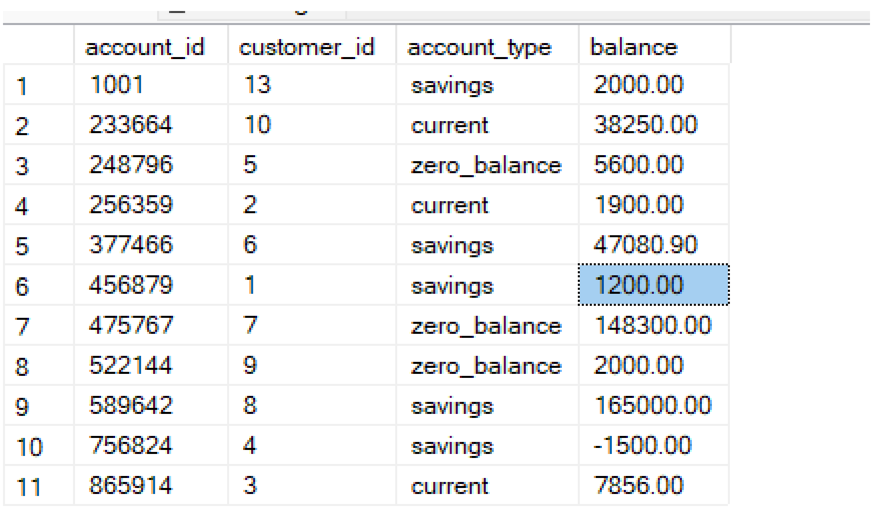
```
Enter your choice: 3
Enter account number: 1
Enter amount to withdraw: 1000
```



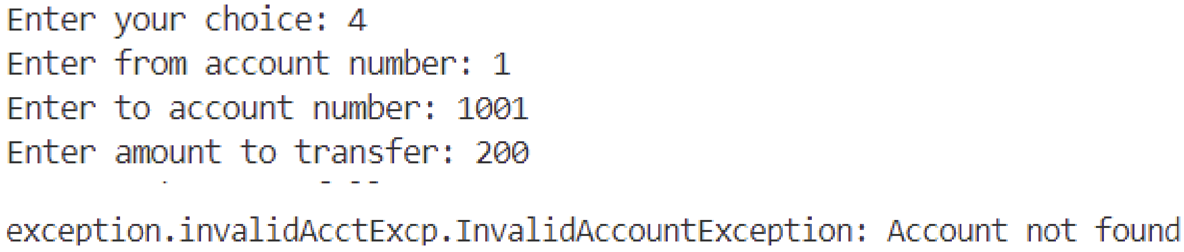
```
exception.invalidAcctExcp.InvalidAccountException:_Account not found
```



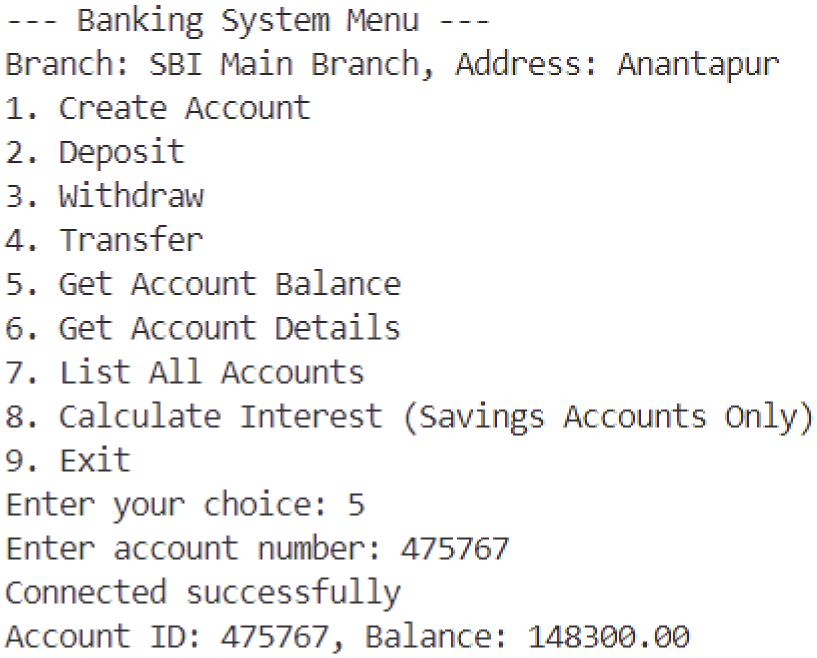
```
1. Create Account
2. Deposit
3. Withdraw
4. Transfer
5. Get Account Balance
6. Get Account Details
7. List All Accounts
8. Calculate Interest (Savings Accounts Only)
9. Exit
Enter your choice: 4
Enter from account number: 1001
Enter to account number: 456879
Enter amount to transfer: 1000
Connected successfully
Withdrawal successful. Updated balance: 2000.00
Connected successfully
Deposited successfully. New balance: 1200.00
```



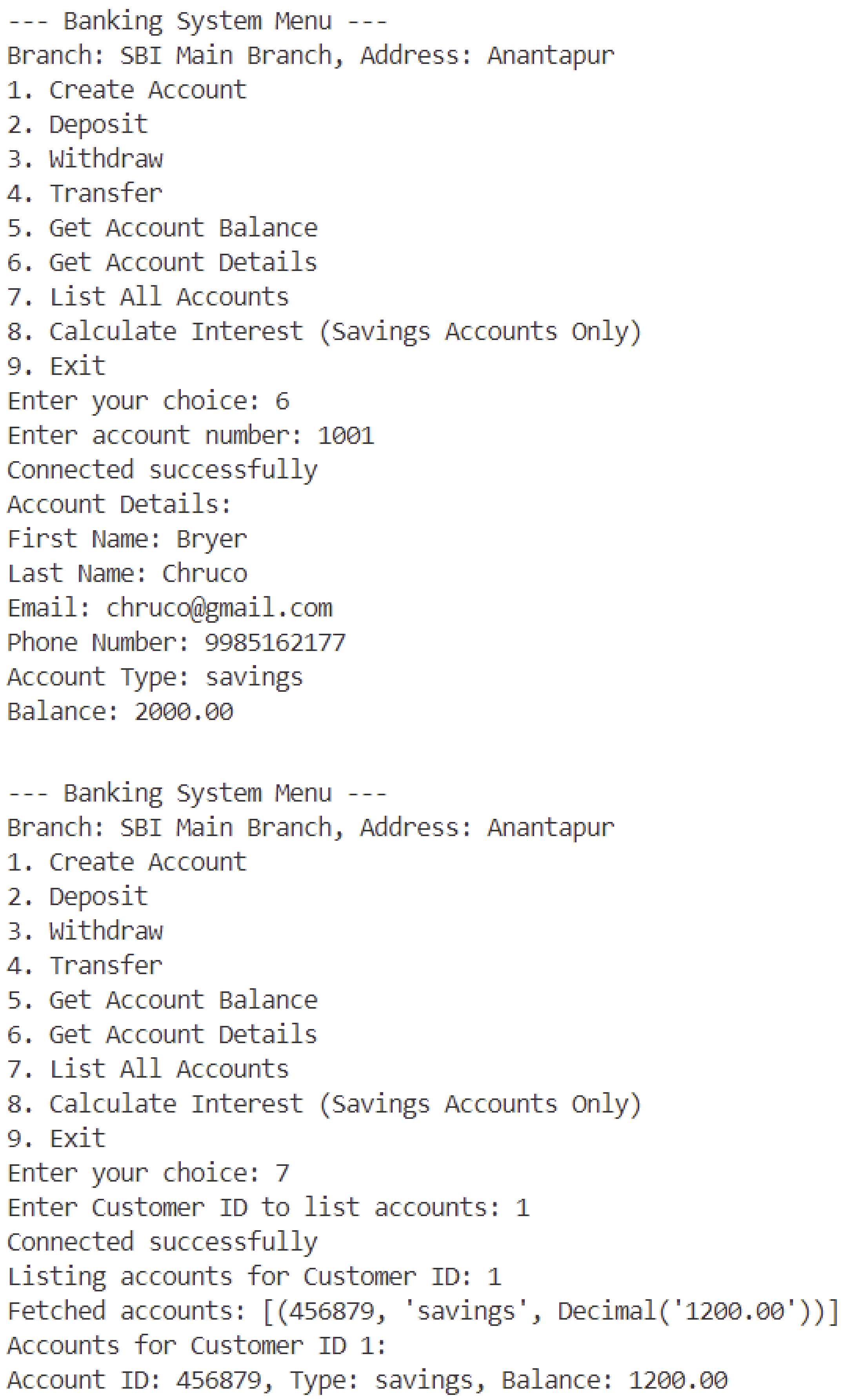
	account_id	customer_id	account_type	balance
1	1001	13	savings	2000.00
2	233664	10	current	38250.00
3	248796	5	zero_balance	5600.00
4	256359	2	current	1900.00
5	377466	6	savings	47080.90
6	456879	1	savings	1200.00
7	475767	7	zero_balance	148300.00
8	522144	9	zero_balance	2000.00
9	589642	8	savings	165000.00
10	756824	4	savings	-1500.00
11	865914	3	current	7856.00



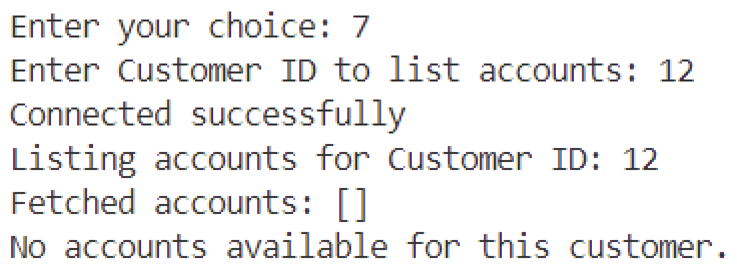
```
Enter your choice: 4
Enter from account number: 1
Enter to account number: 1001
Enter amount to transfer: 200

exception.invalidAcctExcp.InvalidAccountException: Account not found
```



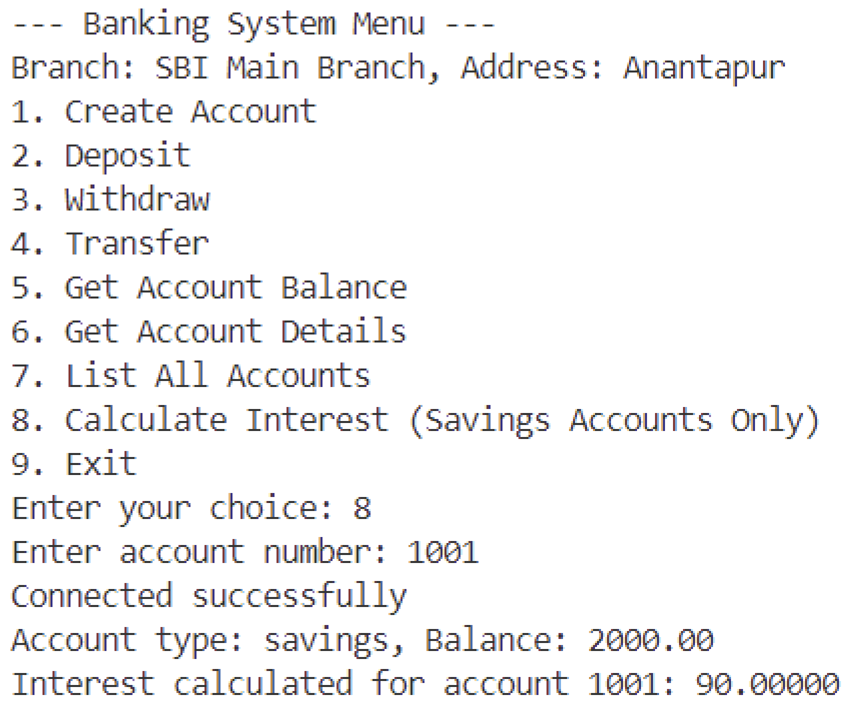
```
--- Banking System Menu ---
Branch: SBI Main Branch, Address: Anantapur
1. Create Account
2. Deposit
3. Withdraw
4. Transfer
5. Get Account Balance
6. Get Account Details
7. List All Accounts
8. Calculate Interest (Savings Accounts Only)
9. Exit
Enter your choice: 5
Enter account number: 475767
Connected successfully
Account ID: 475767, Balance: 148300.00
```



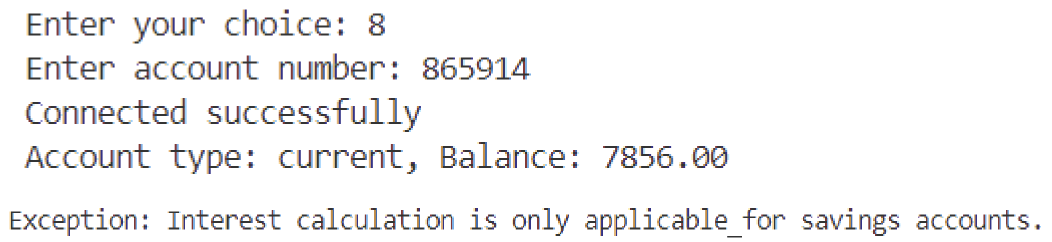
```
--- Banking System Menu ---
Branch: SBI Main Branch, Address: Anantapur
1. Create Account
2. Deposit
3. Withdraw
4. Transfer
5. Get Account Balance
6. Get Account Details
7. List All Accounts
8. Calculate Interest (Savings Accounts Only)
9. Exit
Enter your choice: 6
Enter account number: 1001
Connected successfully
Account Details:
First Name: Bryer
Last Name: Chruco
Email: chruco@gmail.com
Phone Number: 9985162177
Account Type: savings
Balance: 2000.00

--- Banking System Menu ---
Branch: SBI Main Branch, Address: Anantapur
1. Create Account
2. Deposit
3. Withdraw
4. Transfer
5. Get Account Balance
6. Get Account Details
7. List All Accounts
8. Calculate Interest (Savings Accounts Only)
9. Exit
Enter your choice: 7
Enter Customer ID to list accounts: 1
Connected successfully
Listing accounts for Customer ID: 1
Fetched accounts: [(456879, 'savings', Decimal('1200.00'))]
Accounts for Customer ID 1:
Account ID: 456879, Type: savings, Balance: 1200.00
```



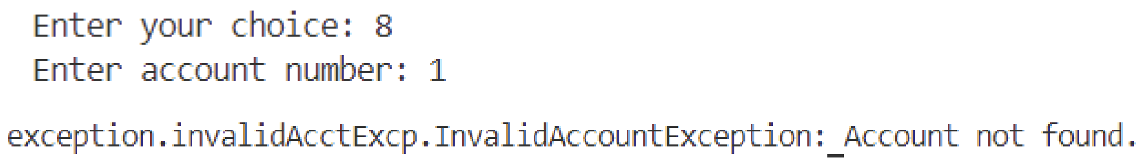
```
Enter your choice: 7
Enter Customer ID to list accounts: 12
Connected successfully
Listing accounts for Customer ID: 12
Fetched accounts: []
No accounts available for this customer.
```



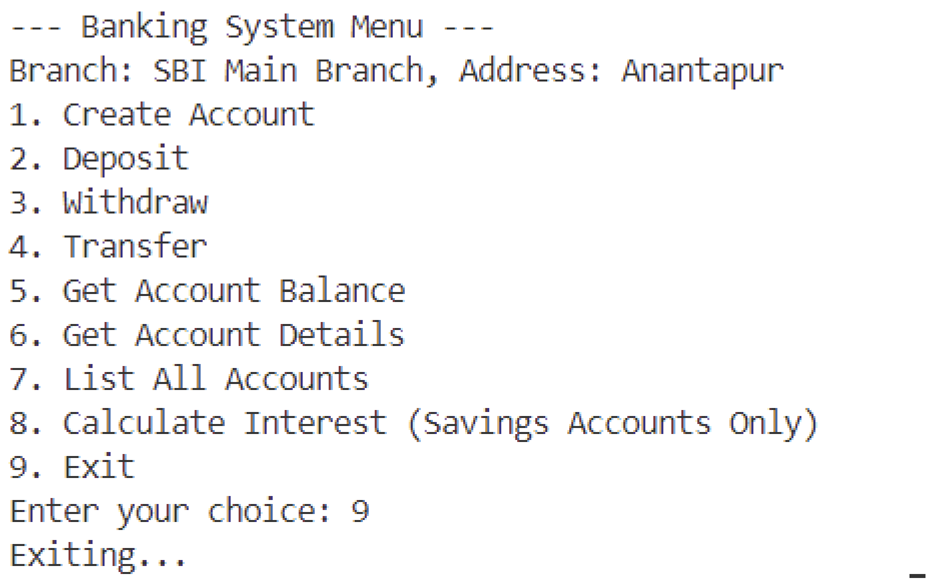
```
--- Banking System Menu ---
Branch: SBI Main Branch, Address: Anantapur
1. Create Account
2. Deposit
3. Withdraw
4. Transfer
5. Get Account Balance
6. Get Account Details
7. List All Accounts
8. Calculate Interest (Savings Accounts Only)
9. Exit
Enter your choice: 8
Enter account number: 1001
Connected successfully
Account type: savings, Balance: 2000.00
Interest calculated for account 1001: 90.00000
```



```
Enter your choice: 8
Enter account number: 865914
Connected successfully
Account type: current, Balance: 7856.00
Exception: Interest calculation is only applicable_for savings accounts.
```



```
Enter your choice: 8
Enter account number: 1
exception.invalidAcctExcp.InvalidAccountException:_Account not found.
```



```
--- Banking System Menu ---
Branch: SBI Main Branch, Address: Anantapur
1. Create Account
2. Deposit
3. Withdraw
4. Transfer
5. Get Account Balance
6. Get Account Details
7. List All Accounts
8. Calculate Interest (Savings Accounts Only)
9. Exit
Enter your choice: 9
Exiting...
```