**Banking System**

**Control Structure**

***Task 1: Conditional Statements***

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

*• Credit Score must be above 700.*

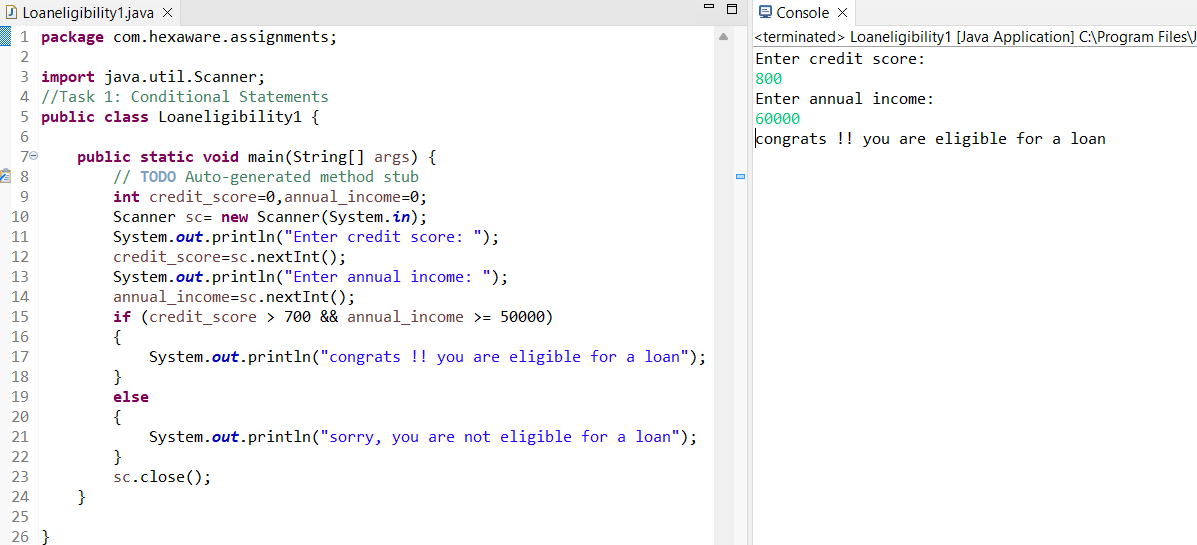
*• Annual Income must be at least $50,000.*

***Tasks:***

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

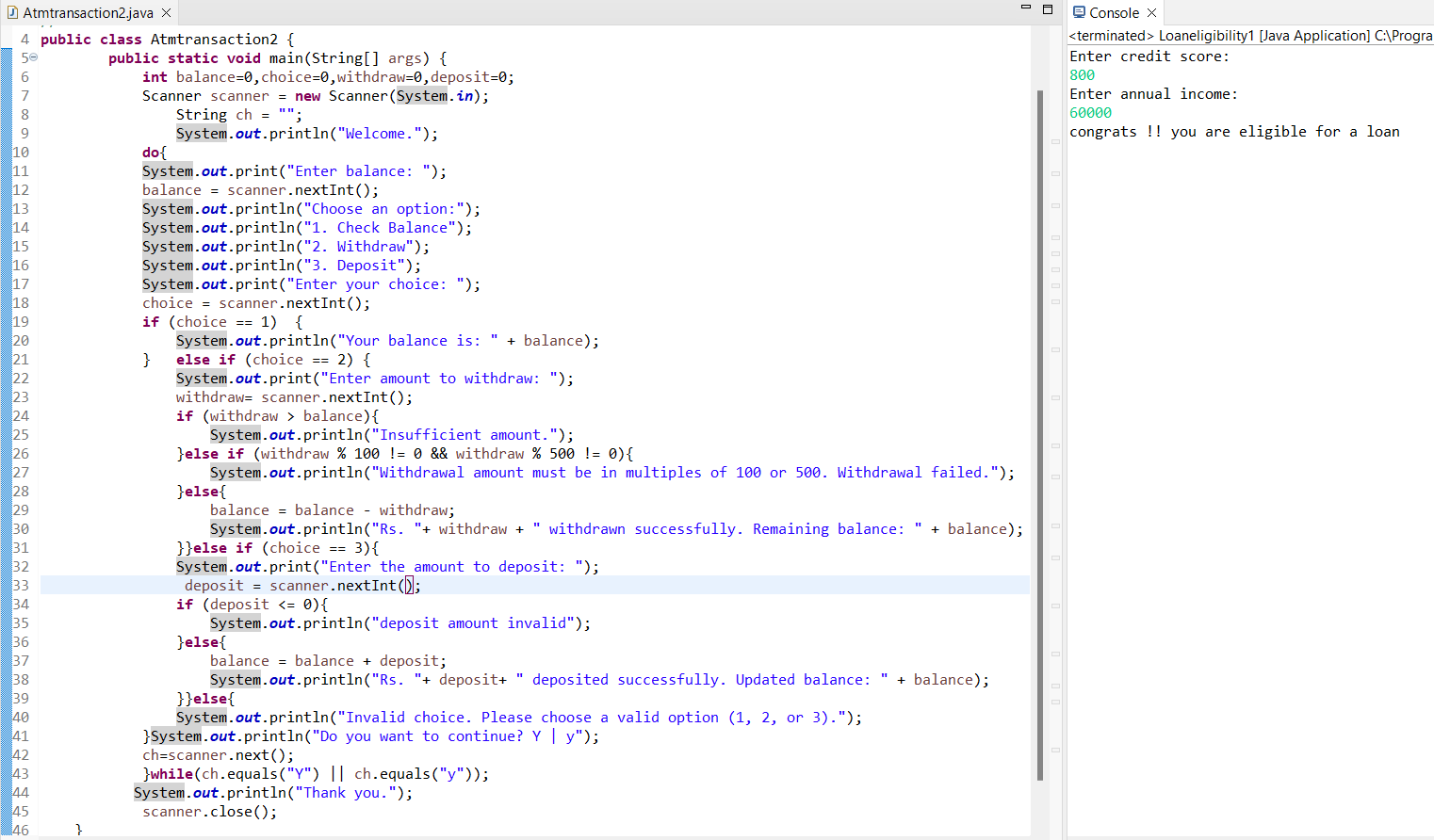
*2. Use conditional statements (if-else) to determine if the customer is eligible for a loan.*

*3. Display an appropriate message based on eligibility.*



***Task 2: Nested Conditional Statements***

*Create a program that simulates an ATM transaction. Display options such as "Check Balance,"  "Withdraw," "Deposit,". Ask the user to enter their current balance and the amount they want to  withdraw or deposit. Implement checks to ensure that the withdrawal amount is not greater than the  available balance and that the withdrawal amount is in multiples of 100 or 500. Display appropriate  messages for success or failure.*



***Task 3: Loop Structures***

*You are responsible for calculating compound interest on savings accounts for bank customers. You  need to calculate the future balance for each customer's savings account after a certain number of years.*

***Tasks:***

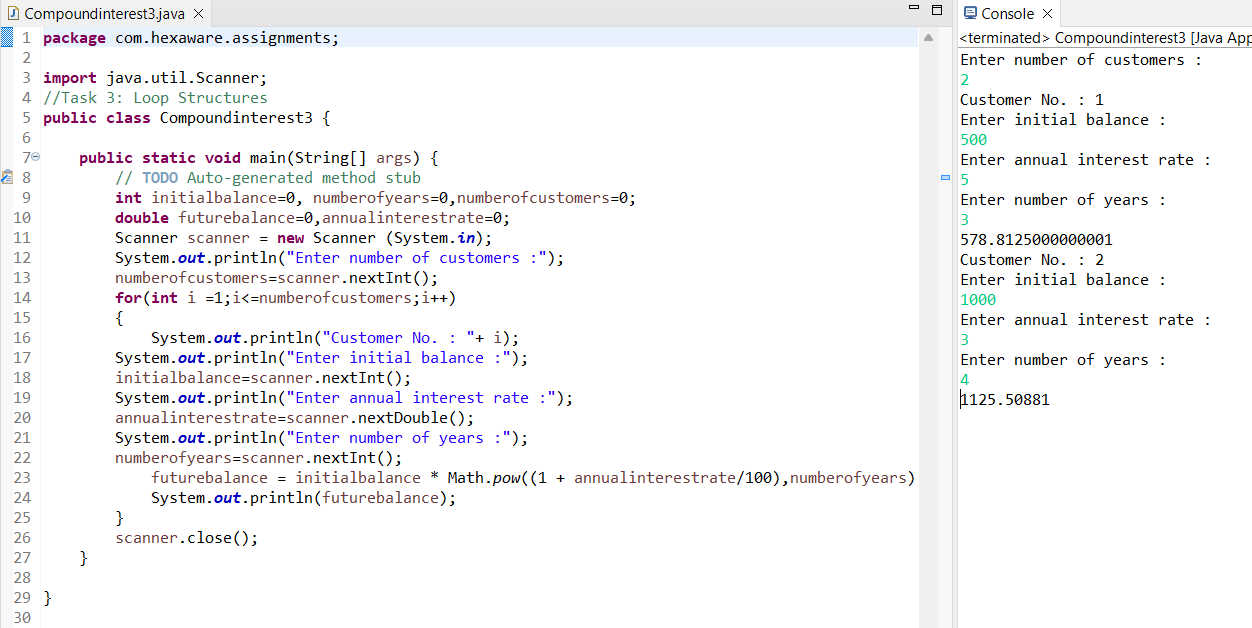
*1. Create a program that calculates the future balance of a savings account.*

*2. Use a loop structure (e.g., for loop) to calculate the balance for multiple customers.*

*3. Prompt the user to enter the initial balance, annual interest rate, and the number of years. 4. Calculate the future balance using the formula:*

*future\_balance = initial\_balance \* (1 + annual\_interest\_rate/100)^years.*

*5. Display the future balance for each customer.*



***Task 4: Looping, Array and Data Validation***

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

***Tasks:***

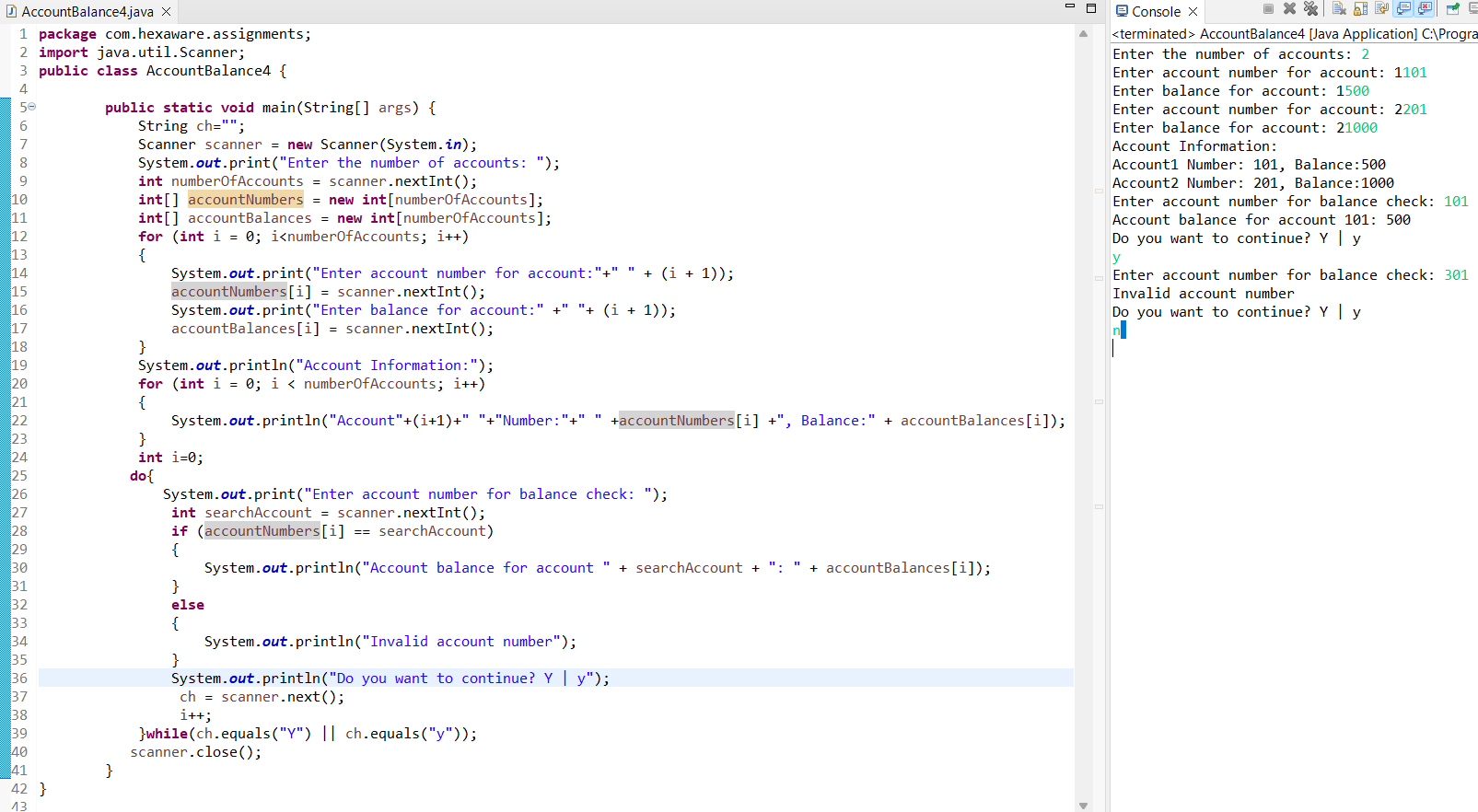
*1. Create a Python program that simulates a bank with multiple customer accounts.*

*2. Use a loop (e.g., while loop) to repeatedly ask the user for their account number and*

*balance until they enter a valid account number.*

*3. Validate the account number entered by the user.*

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



***Task 5: Password Validation***

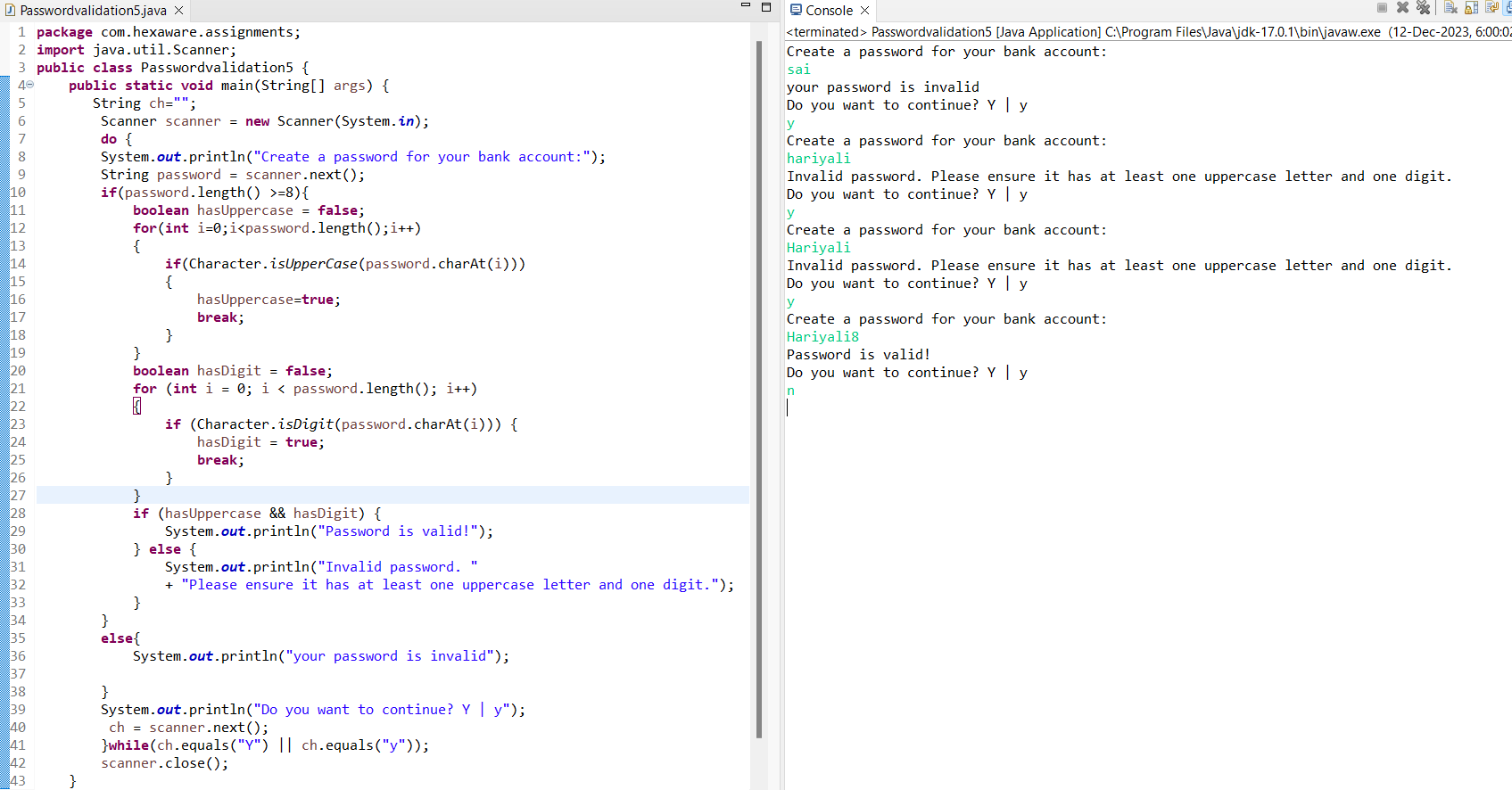
*Write a program that prompts the user to create a password for their bank account. Implement if  conditions to validate the password according to these rules:*

*• The password must be at least 8 characters long.*

*• It must contain at least one uppercase letter.*

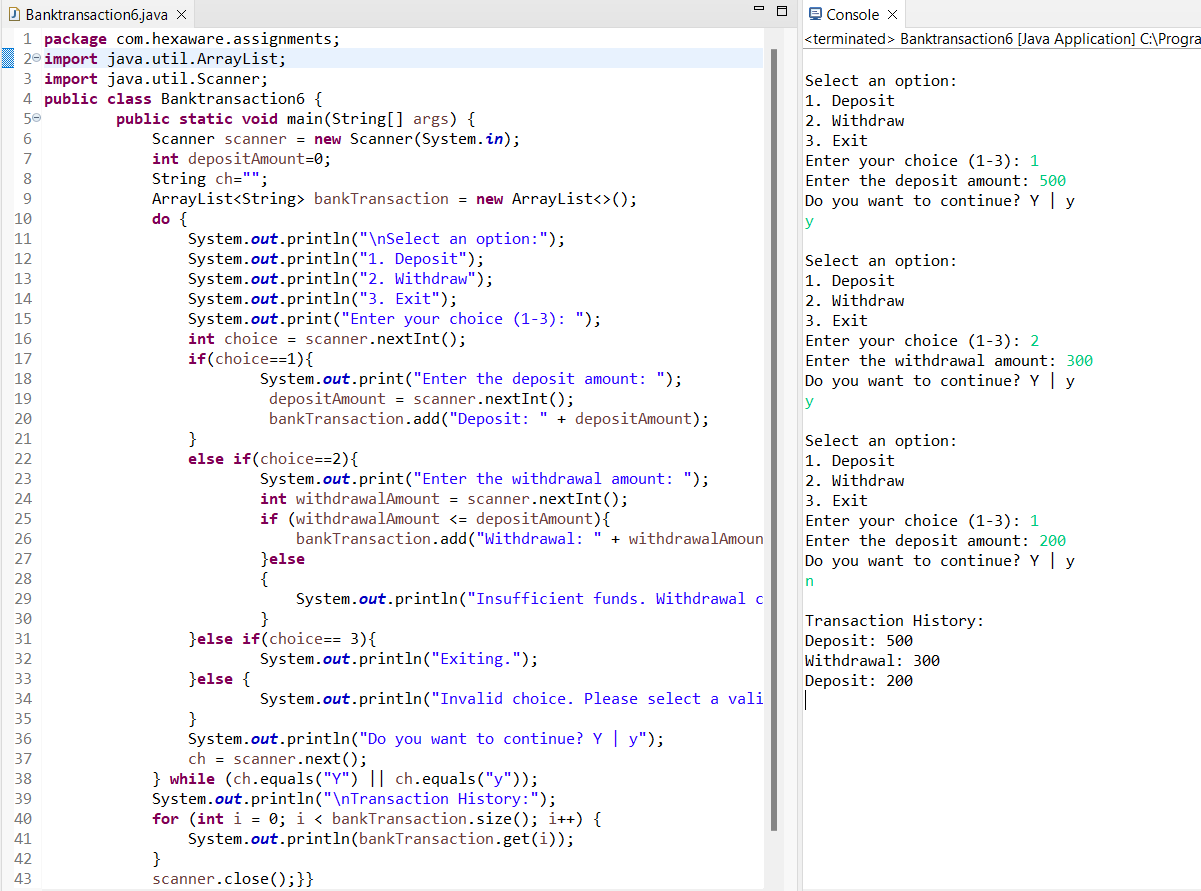
*• It must contain at least one digit.*

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



***Task 6: Transaction History***

*Create a program that maintains a list of bank transactions (deposits and withdrawals) for a customer.  Use a while loop to allow the user to keep adding transactions until they choose to exit. Display the transaction history upon exit using looping statements*.



***OOPS, Collections and Exception Handling***

***Task 7: Class & Object***

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

*• Attributes*

*o Customer ID*

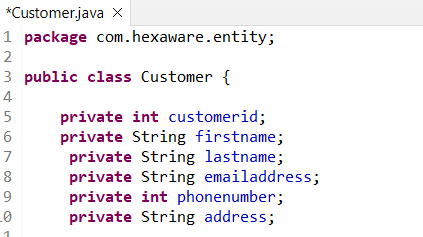
*o First Name*

*o Last Name*

*o Email Address*

*o Phone Number*

*o Address*

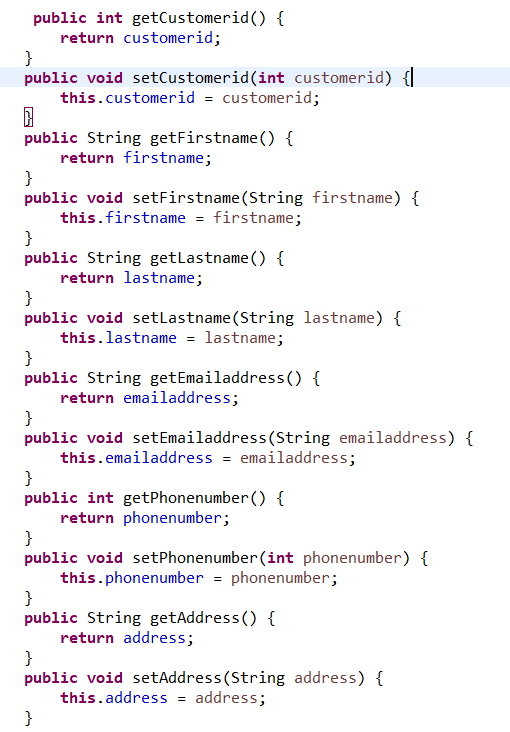


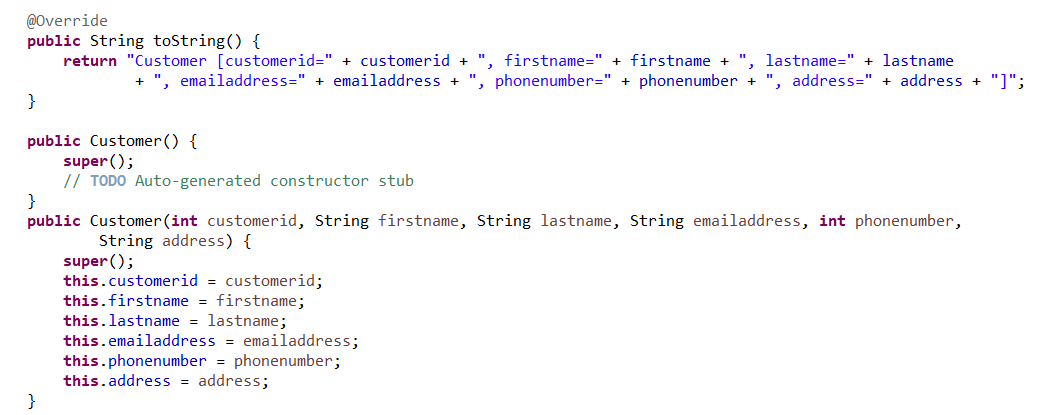
*• Constructor and Methods*

*o Implement default constructors and overload the constructor with Customer*

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

*the attributes.*





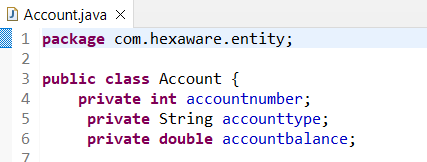
*2. Create an `Account` class with the following confidential attributes:*

*• 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.*

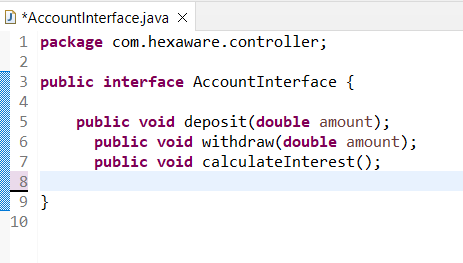
*- deposit(amount: float): Deposit the specified amount into the account.*

*- 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:*

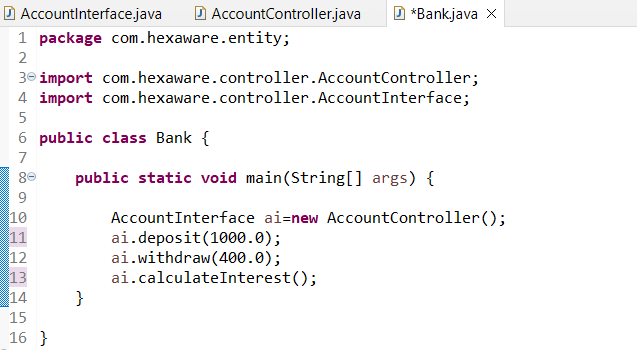
*o create object for account class by calling parameter constructor.*

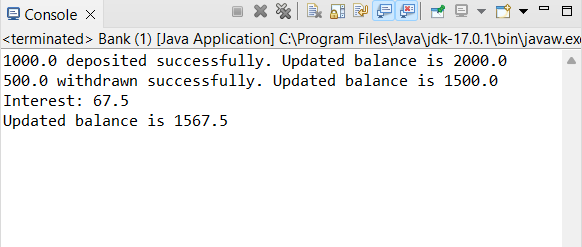
*o deposit(amount: float): Deposit the specified amount into the account.*

*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*





***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.*

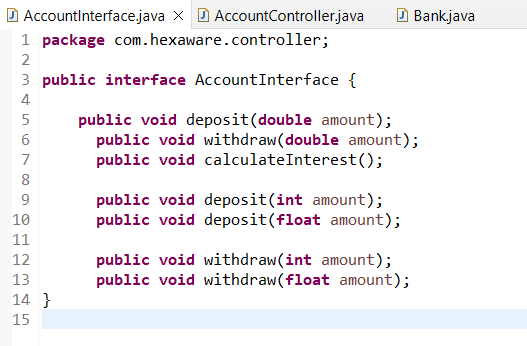
*• deposit(amount: int): Deposit the specified amount into the account.*

*• 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*

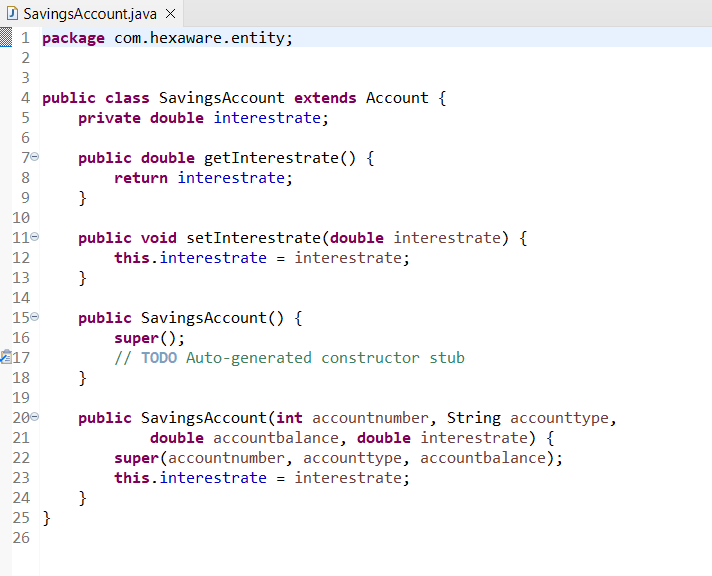
*calculate interest based on the balance and interest rate.*

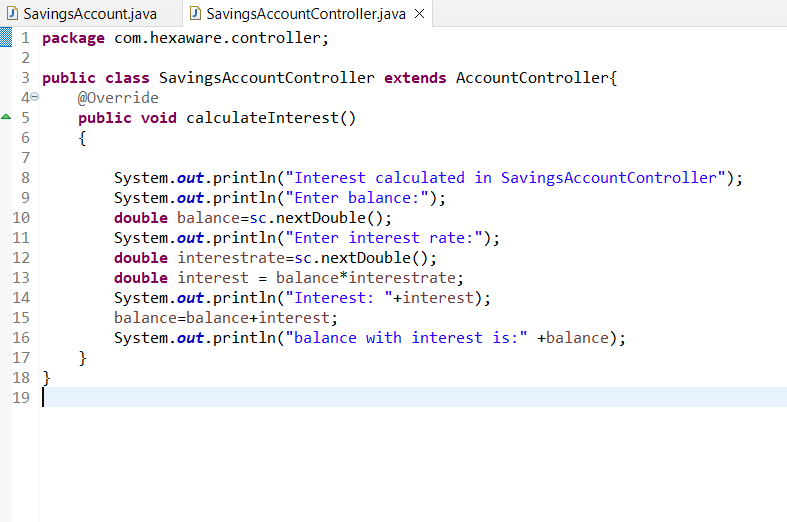
*o CurrentAccount: A current account that includes an additional attribute*

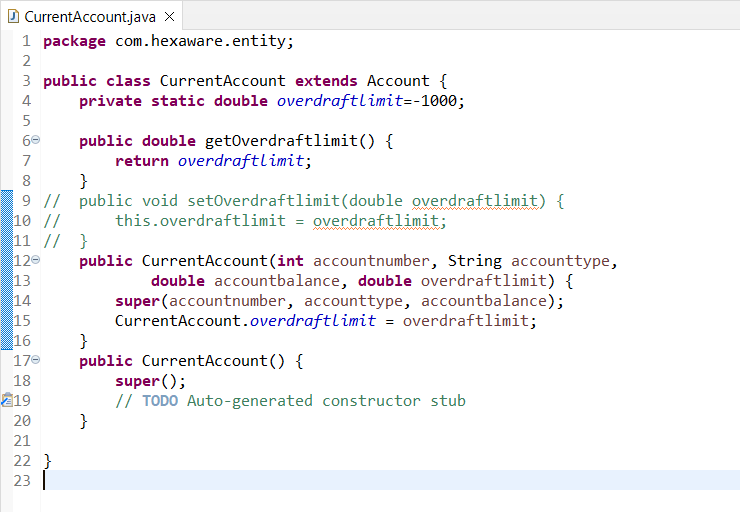
*overdraftLimit. A current account with no interest. Implement the withdraw()*

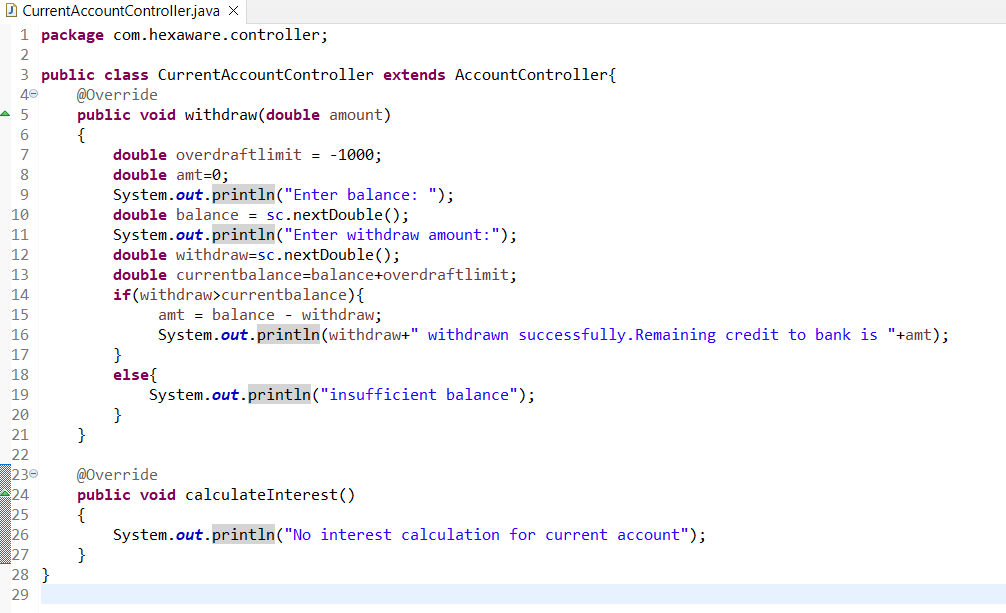
*method to allow overdraft up to a certain limit (configure a constant for the*

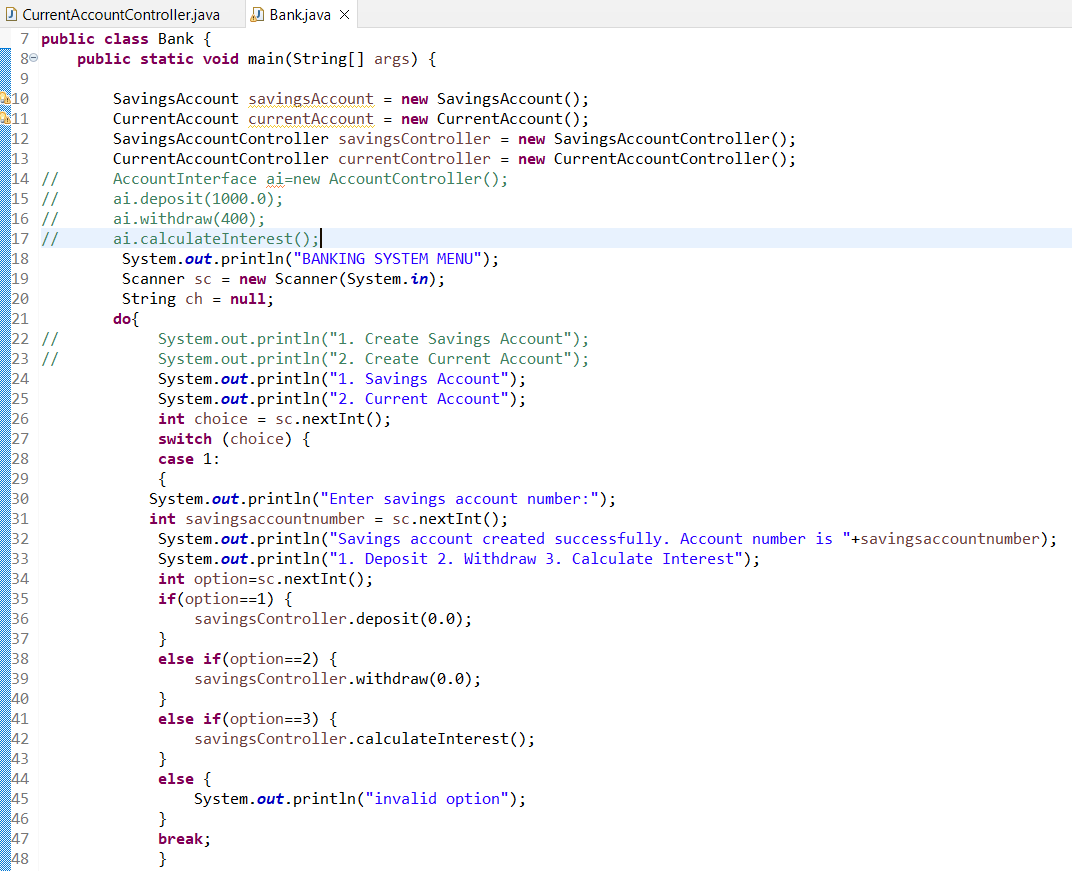
*overdraft limit).*

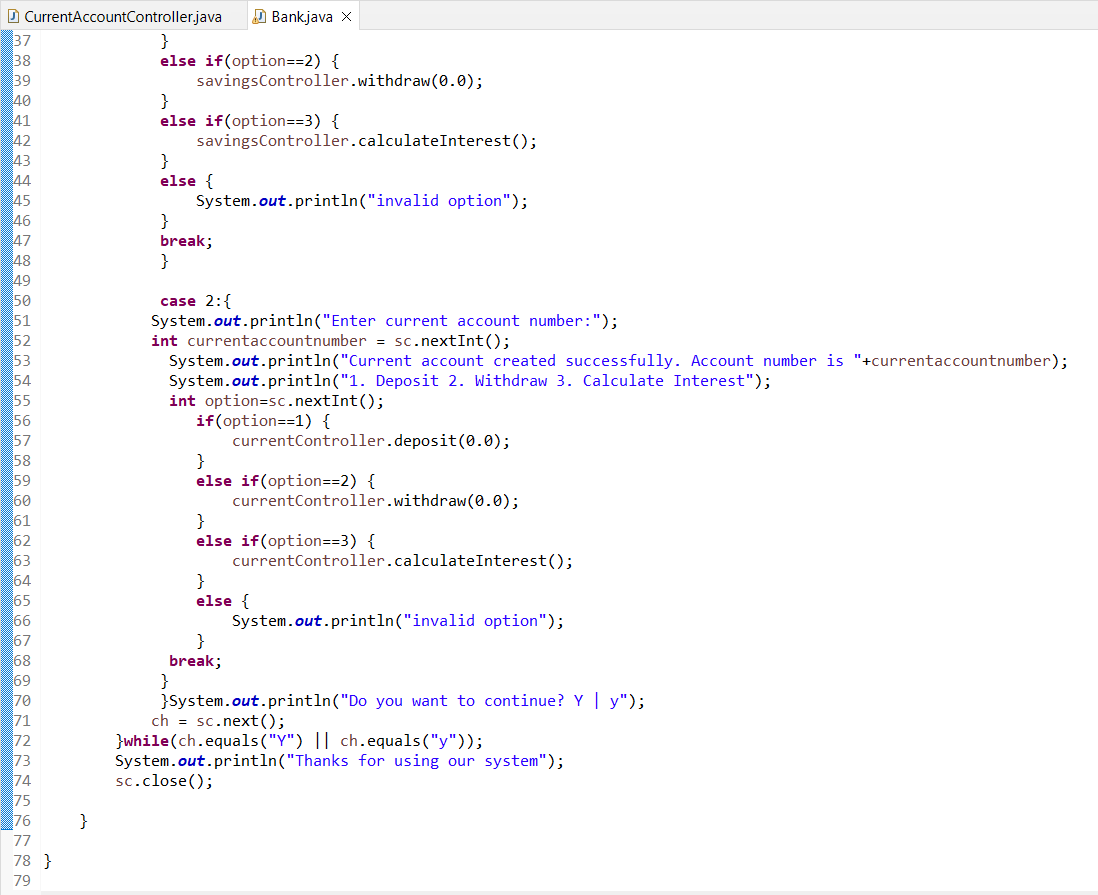


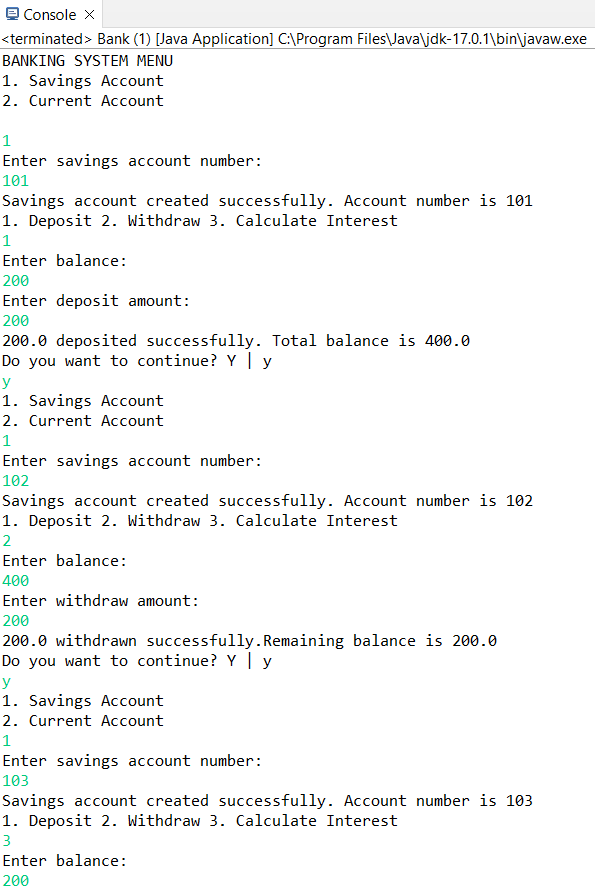


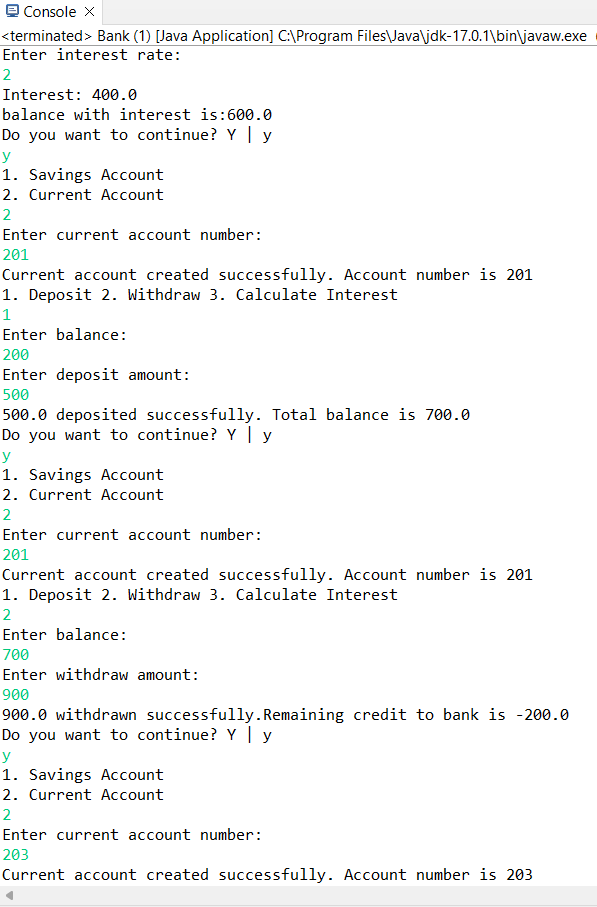


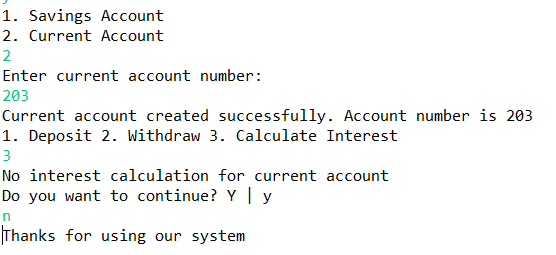












***Task 9: Abstraction***

*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.*

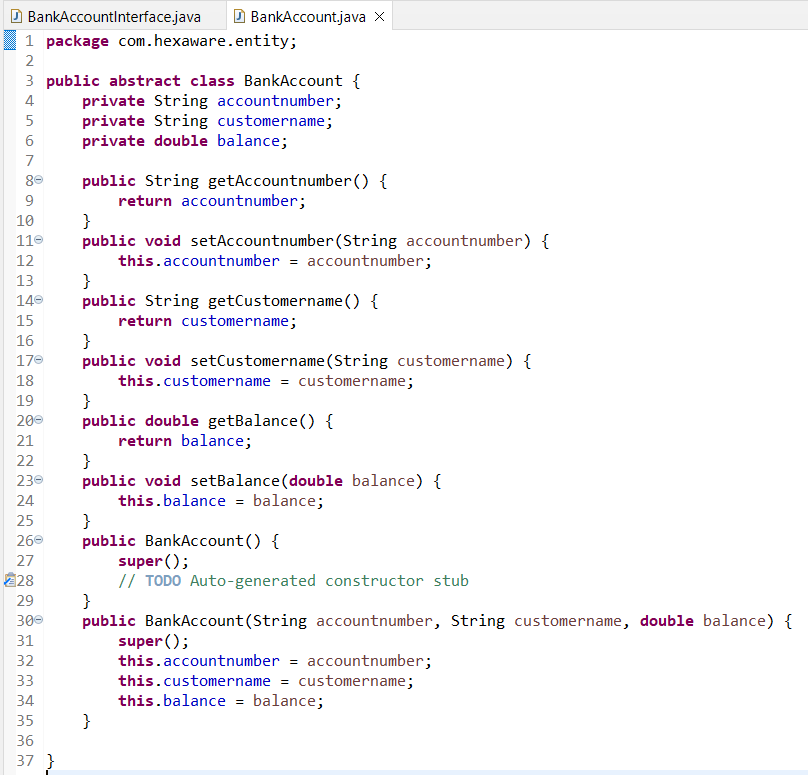
*o Balance.*

*• 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.*



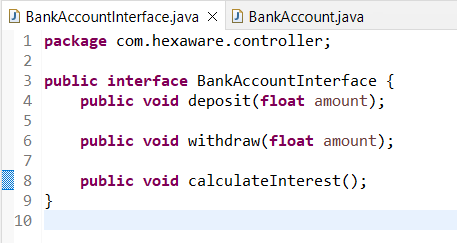
*• Abstract methods:*

*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.*



*Create two concrete classes that inherit from BankAccount:*

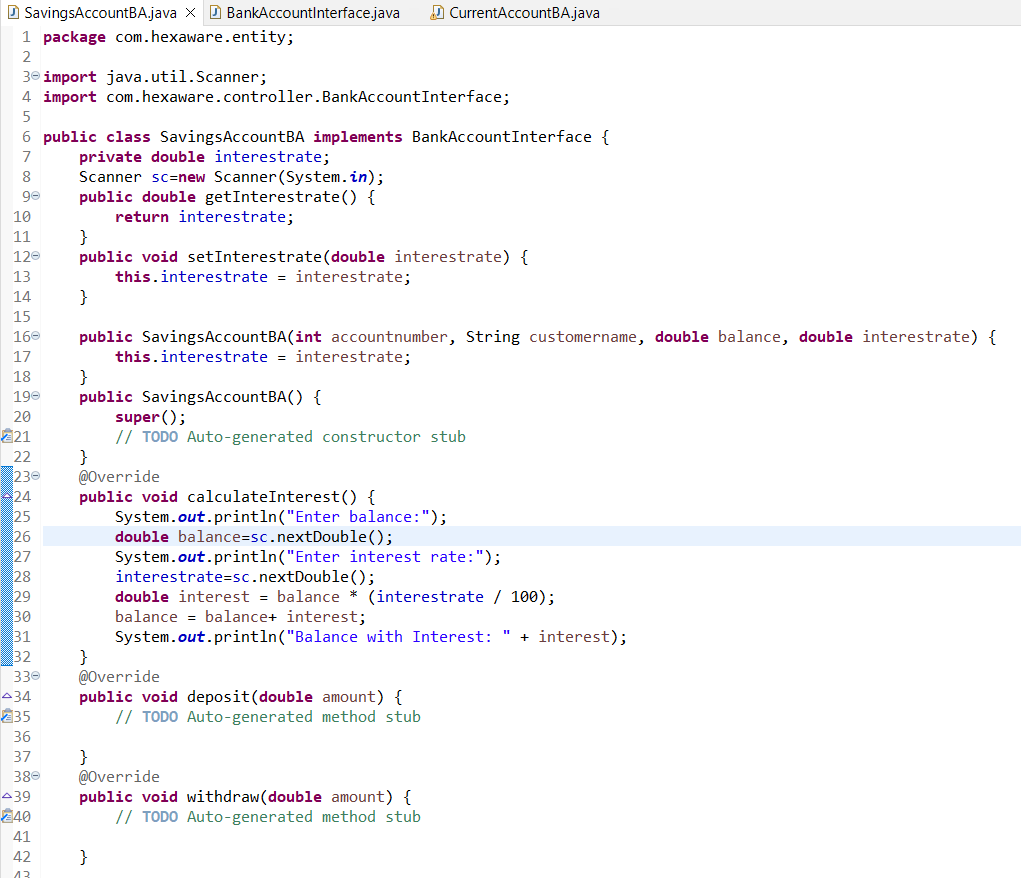
*• SavingsAccount: A savings account that includes an additional attribute for interest rate.*

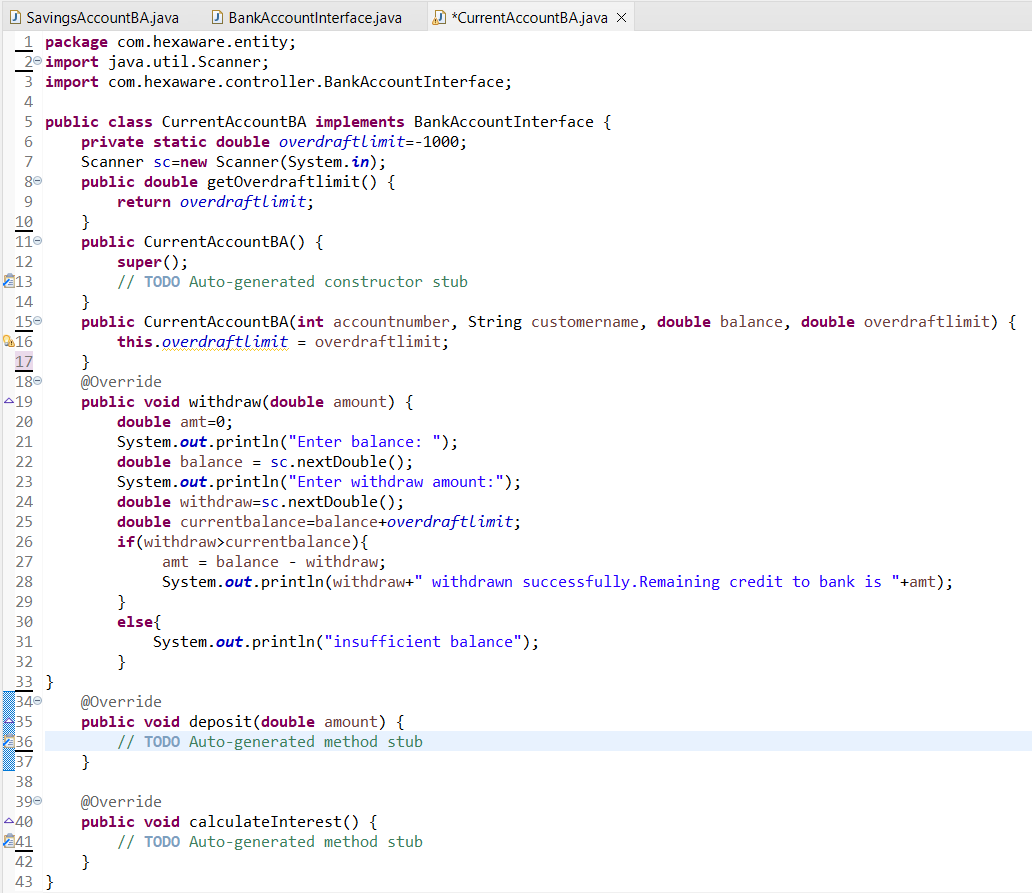
*Implement the calculate\_interest() method to calculate interest based on the balance*

*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.*

*Output : (SAME AS TASK 8)*

***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.*

*Output : (SAME AS TASK 7)*

*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.*

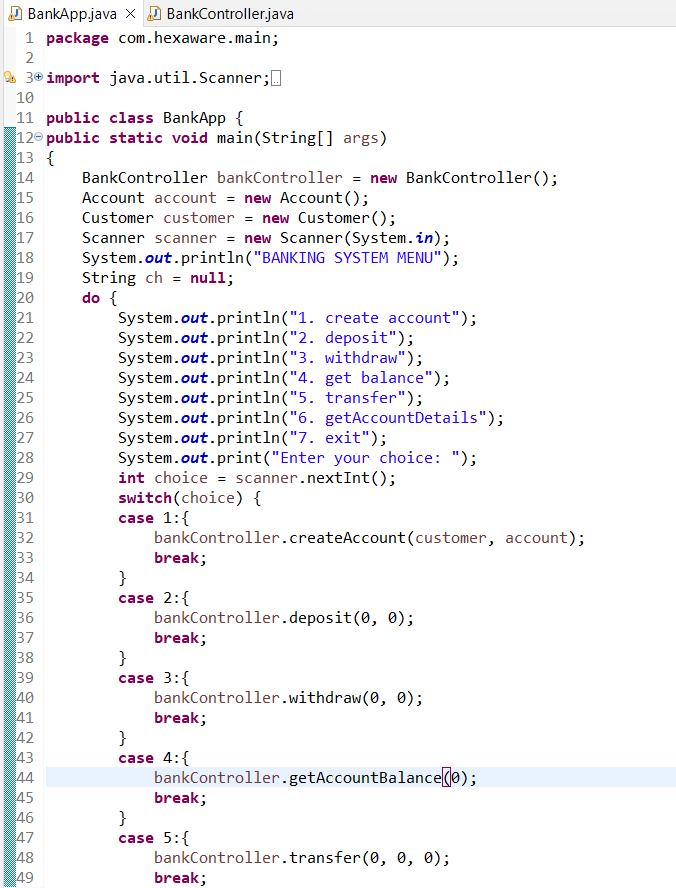
*• getAccountDetails(account\_number: long): Should return the account and customer*

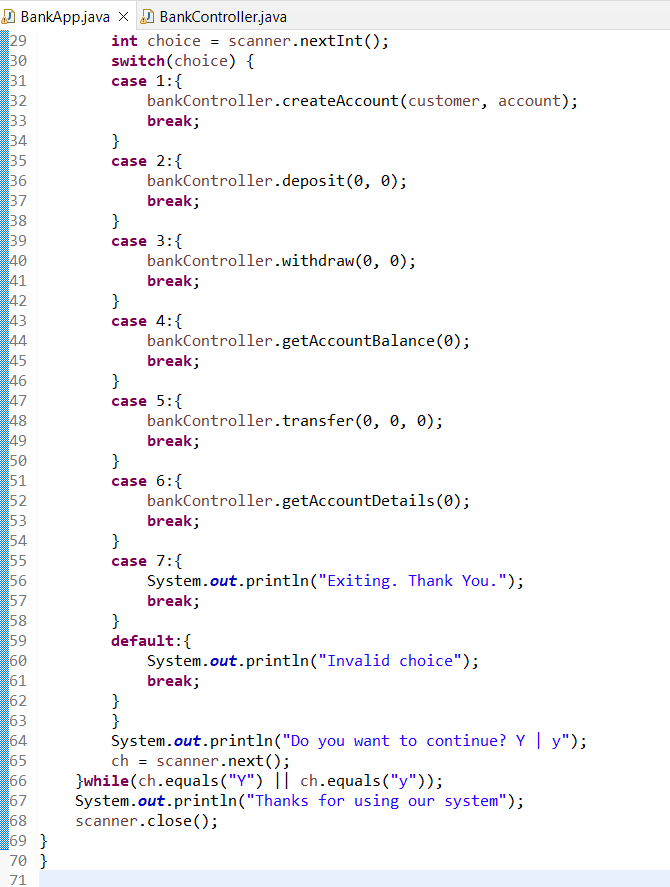
*details.3. Create a BankApp class with a main method to simulate the banking system. Allow the user to*

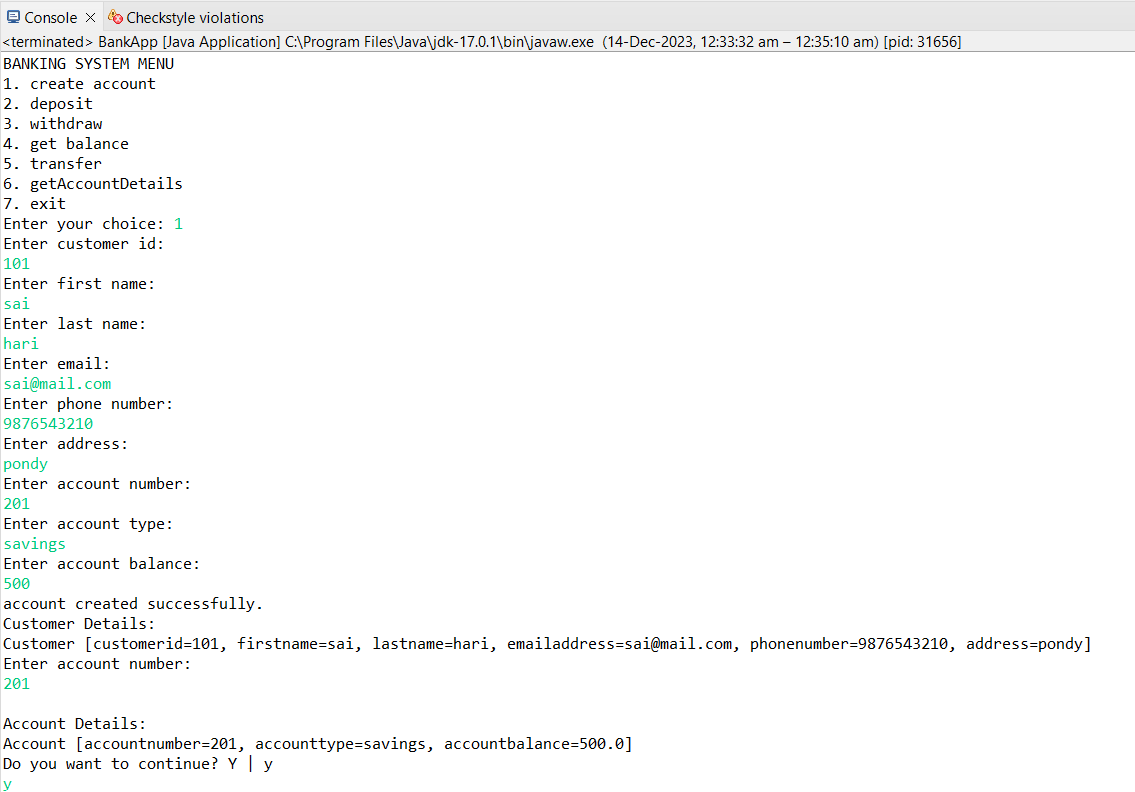
*interact with the system by entering commands such as "create\_account", "deposit",*

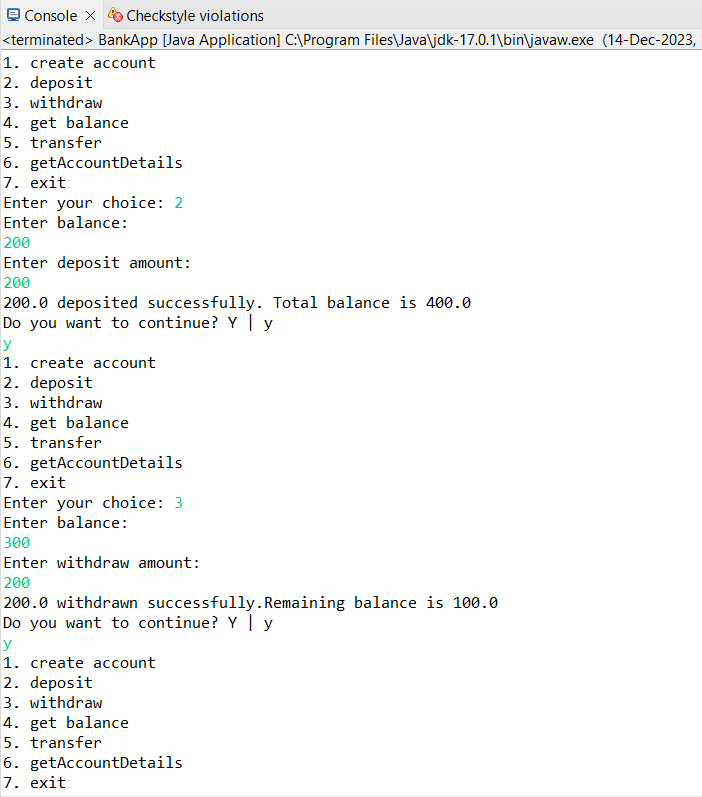
*"withdraw", "get\_balance", "transfer", "getAccountDetails" and "exit." create\_account should*

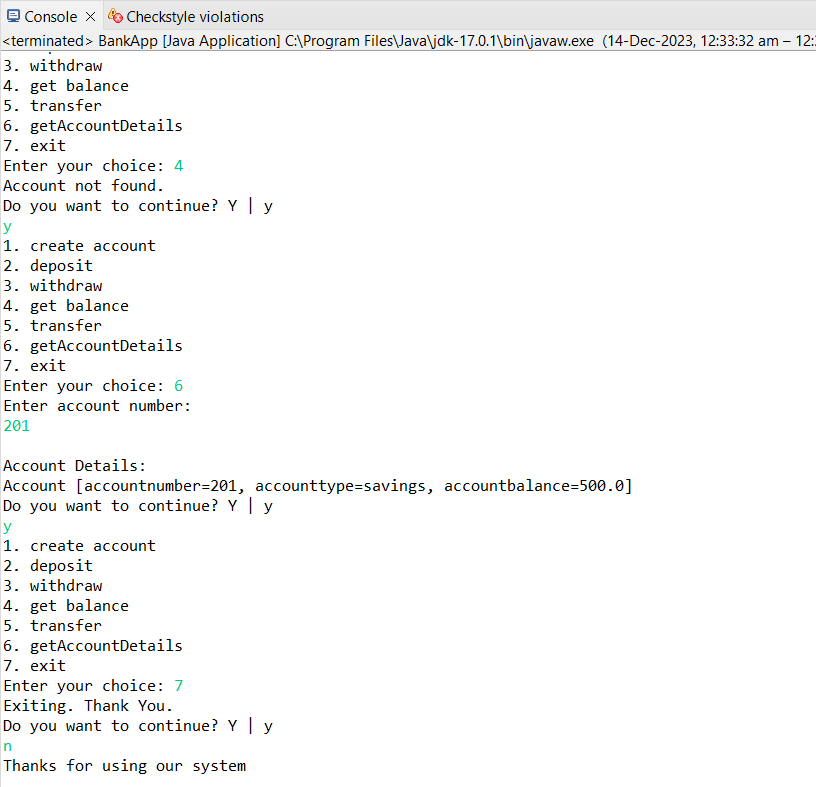
*display sub menu to choose type of accounts and repeat this operation until user exit*.











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

1. *Create a ‘Customer’ class as mentioned above task.*

*OUTPUT: Same as task 7*

*2. Create an class ‘Account’ that includes the following attributes. Generate account number using*

*static variable.*

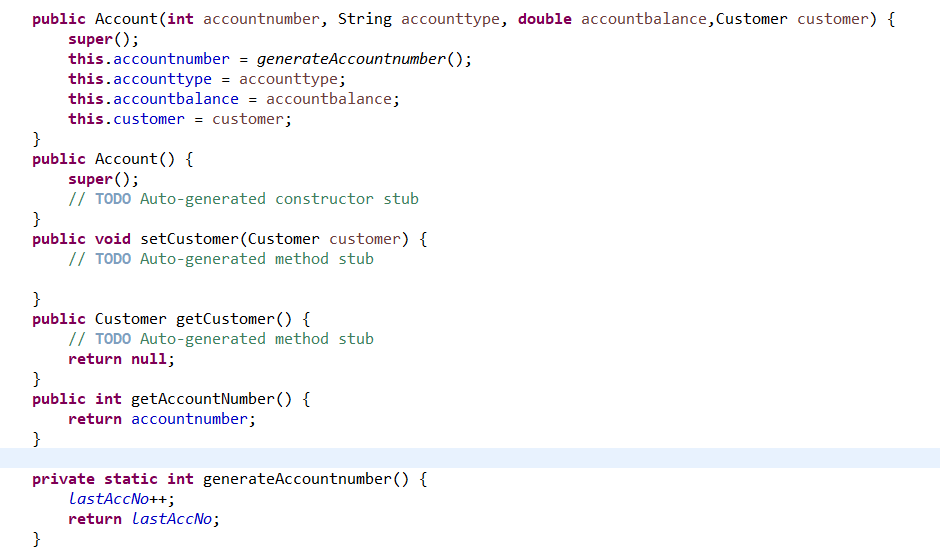
*• Account Number (a unique identifier).*

*• Account Type (e.g., Savings, Current)*

*• Account Balance*

*• Customer (the customer who owns the account)*

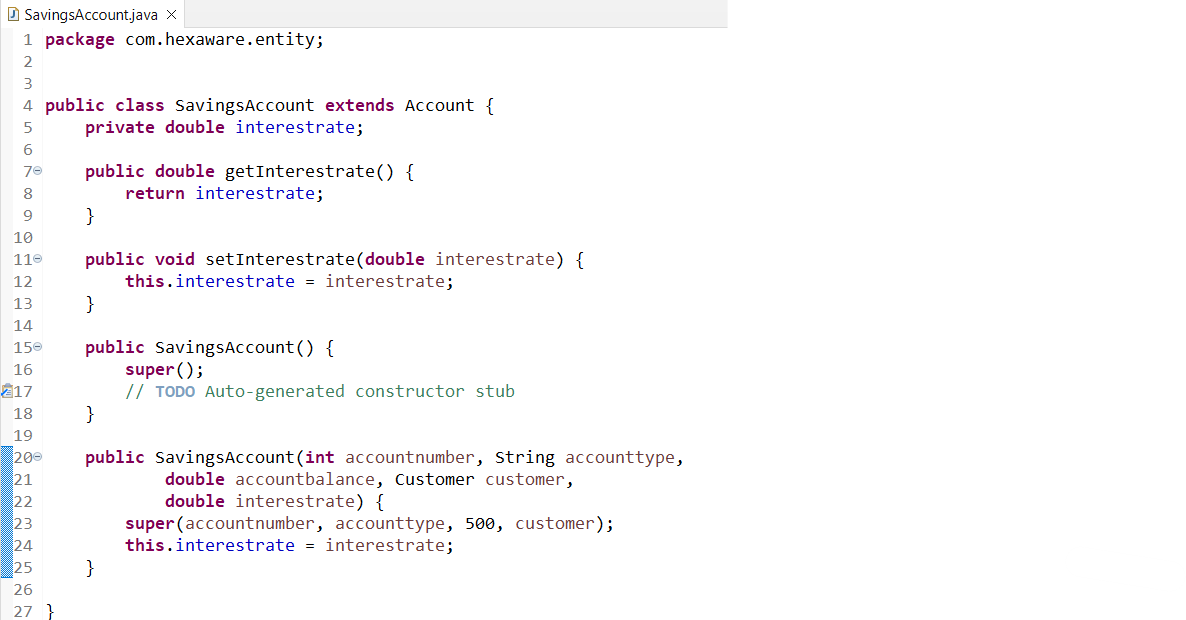
*• lastAccNo*



*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*

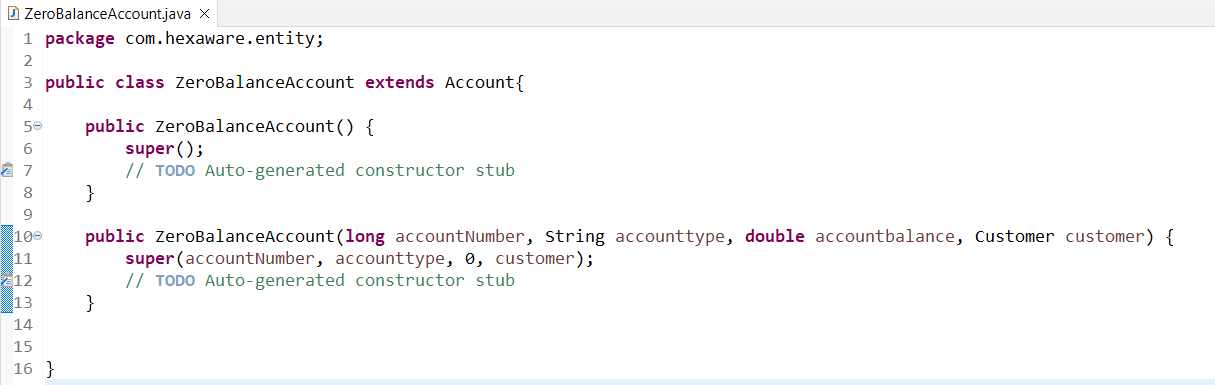
*overdraftLimit(credit limit). withdraw() method to allow overdraft up to a certain limit.*

*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*

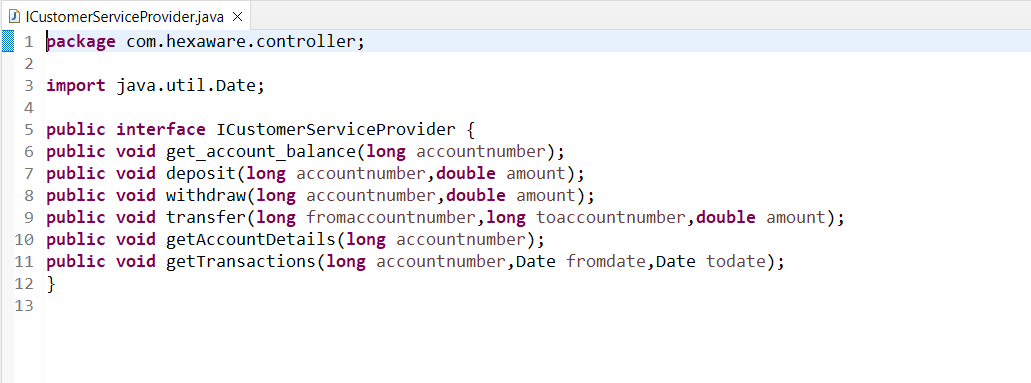
*balance rule.*

*• 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.*

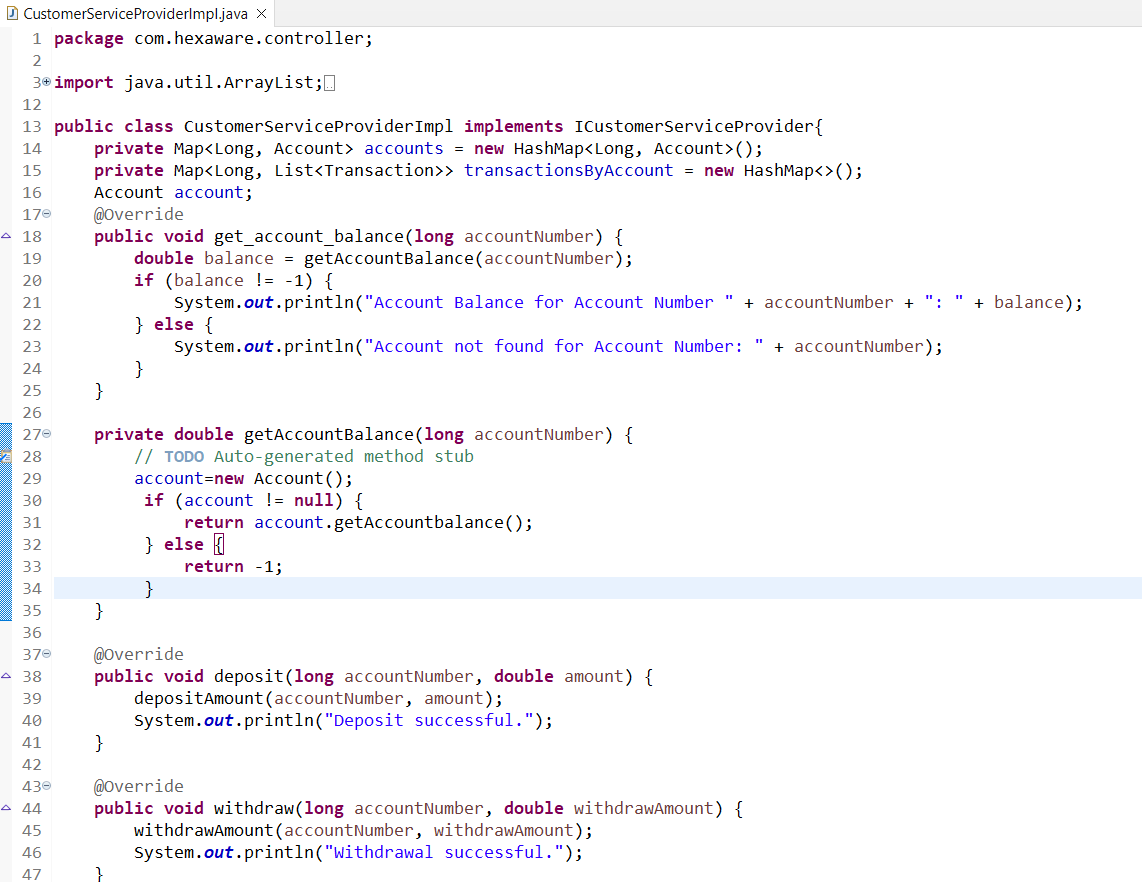
*• calculateInterest(): the calculate\_interest() method to calculate interest based on the*

*balance and interest rate.*



*6. Create CustomerServiceProviderImpl class which implements ICustomerServiceProvider*

*provide all implementation methods.*



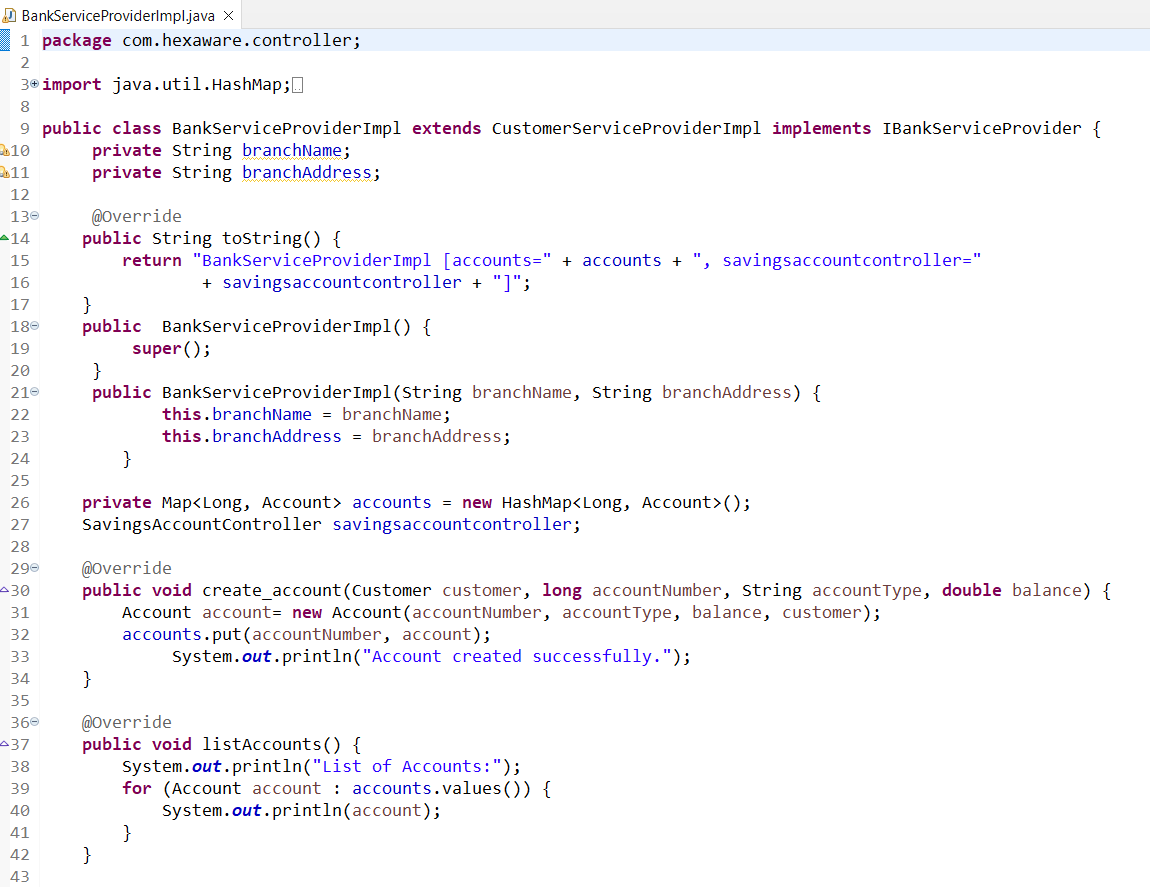
*7. Create BankServiceProviderImpl class which inherits from CustomerServiceProviderImpl and*

*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*

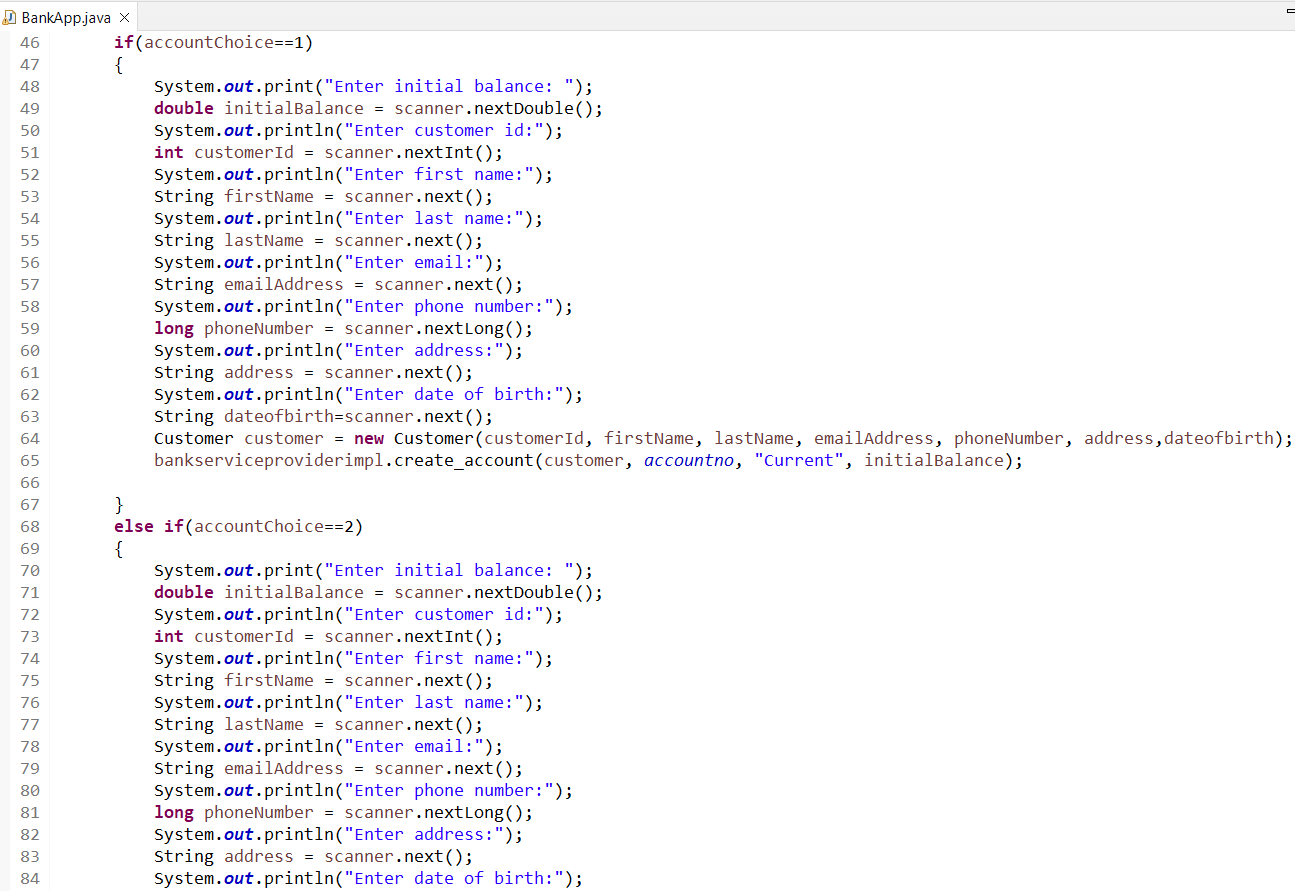
*by entering choice from menu such as "create\_account", "deposit", "withdraw",*

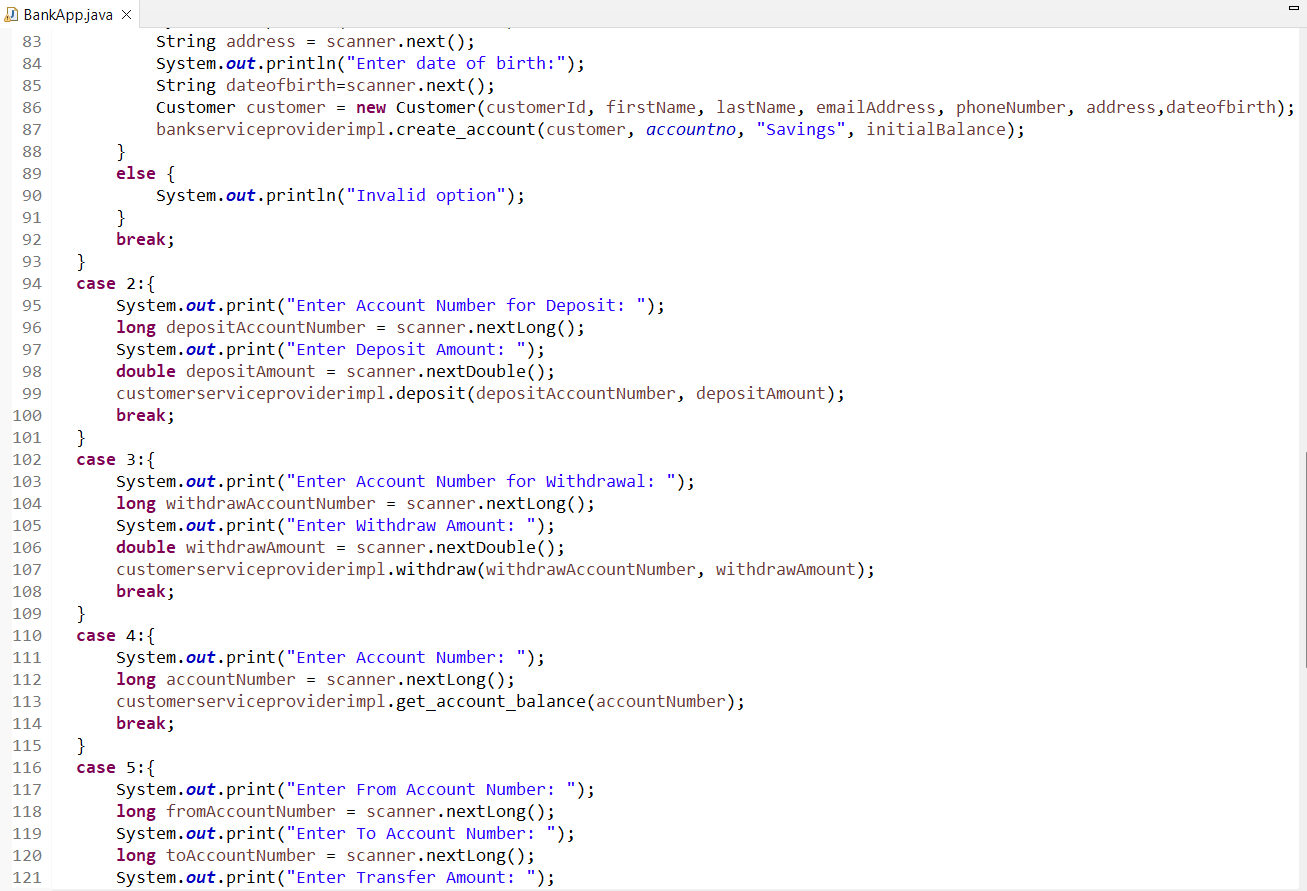
*"get\_balance", "transfer", "getAccountDetails", "ListAccounts" and "exit."*

*• create\_account should display sub menu to choose type of accounts and repeat this*

*operation until user exit.*

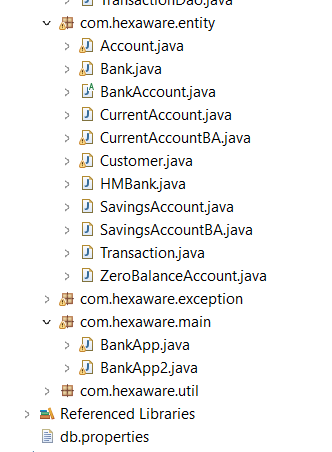
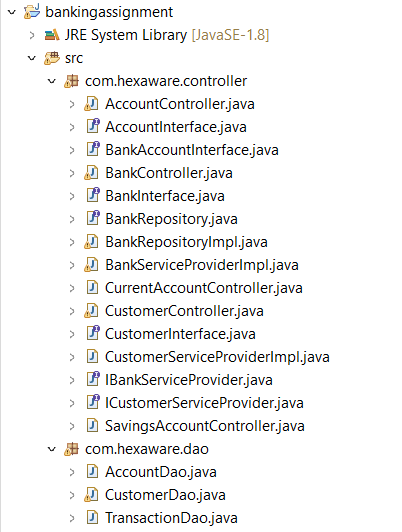






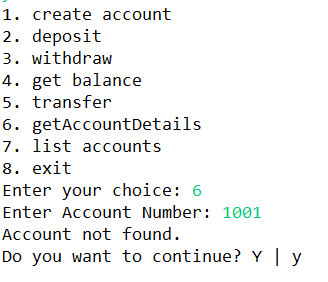
*9. Place the interface/abstract class in service package and interface/abstract class implementation*

*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.*

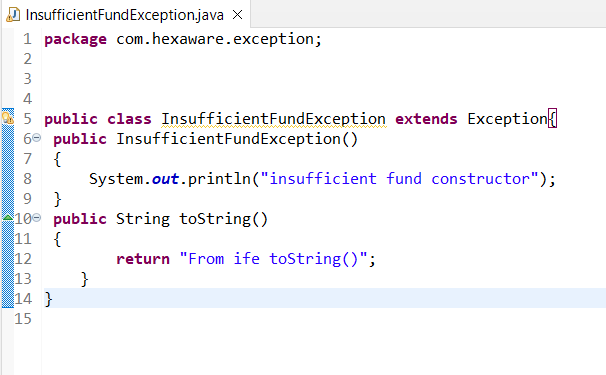


***Task 12: Exception Handling***

*throw the exception whenever needed and Handle in main method,*

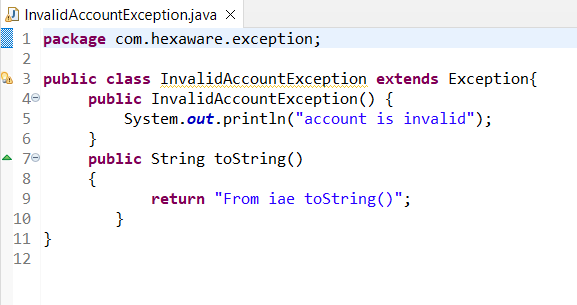
*1. InsufficientFundException throw this exception when user try to withdraw amount or transfer*

*amount to another account and the account runs out of money in the account.*



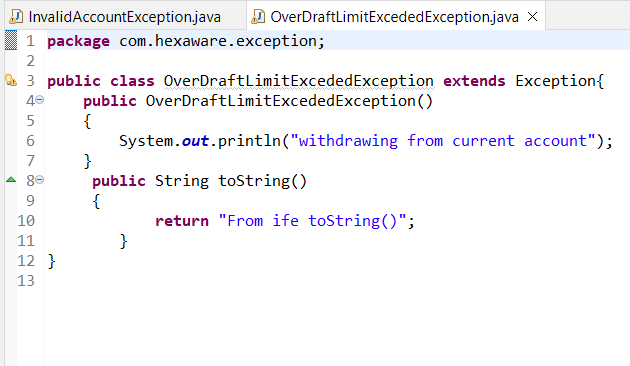
*2. InvalidAccountException throw this exception when user entered the invalid account number*

*when tries to transfer amount, get account details classes.*

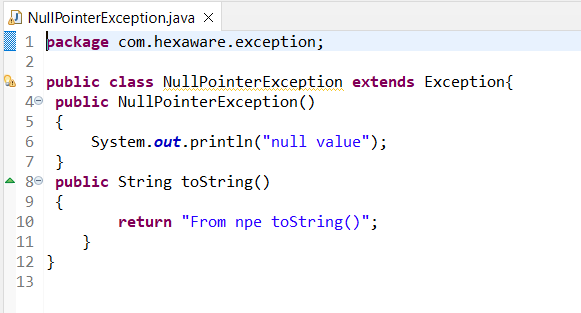


*3. OverDraftLimitExcededException thow this exception when current account customer try to*

*with draw amount from the current account.*



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



***Task 13: Collection***

*1. From the previous task change the HMBank attribute Accounts to List of Accounts and perform*

*the same operation.*

*2. From the previous task change the HMBank attribute Accounts to Set of Accounts and perform*

*the same operation.*

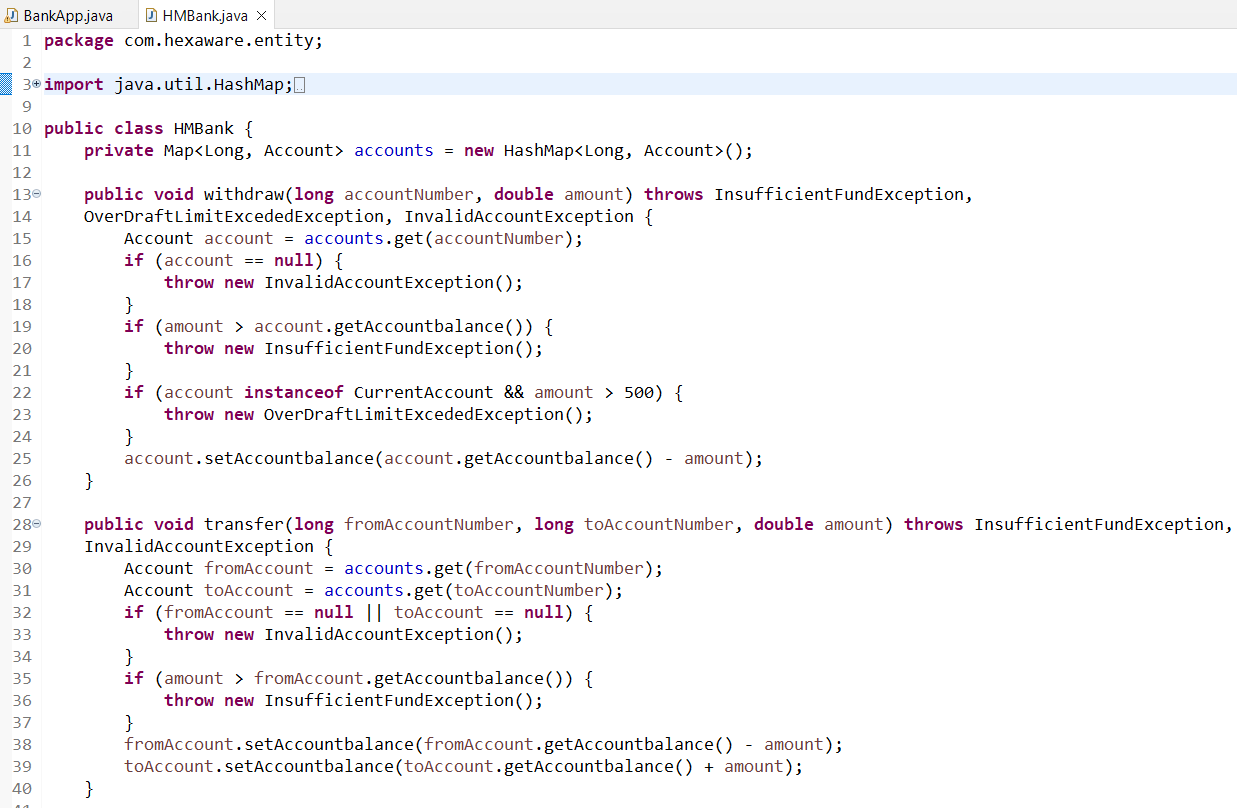
*• Avoid adding duplicate Account object to the set.*

*• Create Comparator<Account> object to sort the accounts based on customer name*

*when listAccounts() method called.*

*3. From the previous task change the HMBank attribute Accounts to HashMap of Accounts and*

*perform the same operation.*





***Task 14: Database Connectivity****.*

*1. Create a ‘Customer’ class as mentioned above task.*

*2. Create an class ‘Account’ that includes the following attributes. Generate account number using*

*static variable.*

*• Account Number (a unique identifier).*

*• Account Type (e.g., Savings, Current)*

*• Account Balance*

*• Customer (the customer who owns the account)*

*• lastAccNo*

*3. Create a class ‘TRANSACTION’ that include following attributes*

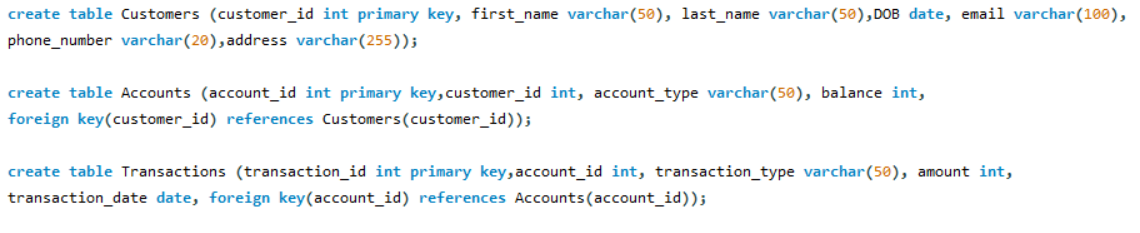
*• Account*

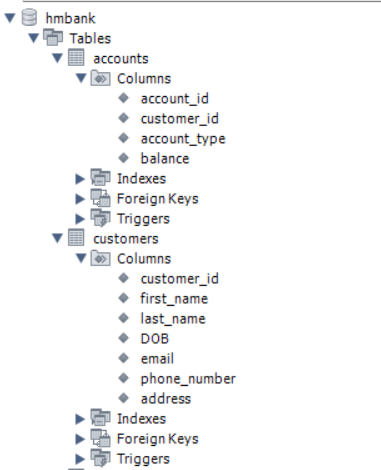
*• Description*

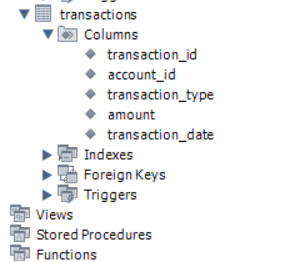
*• Date and Time*

*• TransactionType(Withdraw, Deposit, Transfer)*

*• TransactionAmount*







*4. Create three child classes that inherit the Account class and each class must contain below*

*mentioned attribute:*

*• SavingsAccount: A savings account that includes an additional attribute for interest rate.*

*Saving account should be created with minimum balance 500.*

*• CurrentAccount: A Current account that includes an additional attribute for*

*overdraftLimit(credit limit).*

*• ZeroBalanceAccount: ZeroBalanceAccount can be created with Zero balance.*

*5. Create ICustomerServiceProvider interface/abstract class with following functions:*

*• get\_account\_balance(account\_number: long): Retrieve the balance of an account given*

*its account number. should return the current balance of account.*

*• deposit(account\_number: long, amount: float): Deposit the specified amount into the*

*account. Should return the current balance of account.*

*• withdraw(account\_number: long, amount: float): Withdraw the specified amount from*

*the account. Should return the current balance of account.*

*o A savings account should maintain a minimum balance and checking if the*

*withdrawal violates the minimum balance rule.*

*o Current account customers are allowed withdraw overdraftLimit and available*

*account balance. withdraw limit can exceed the available balance and should*

*not exceed the overdraft limit.*

*• transfer(from\_account\_number: long, to\_account\_number: int, amount: float):*

*Transfer money from one account to another. both account number should be validate*

*from the database use getAccountDetails method.*

*• getAccountDetails(account\_number: long): Should return the account and customer*

*details.*

*• getTransations(account\_number: long, FromDate:Date, ToDate: Date): Should return*

*the list of transaction between two dates.*

*6. Create IBankServiceProvider interface/abstract class with following functions:*

*• create\_account(Customer customer, long accNo, String accType, float balance): Create*

*a new bank account for the given customer with the initial balance.*

*• listAccounts(): Array of BankAccount: List all accounts in the bank.(List[Account]*

*accountsList)*

*• getAccountDetails(account\_number: long): Should return the account and customer*

*details.*

*• calculateInterest(): the calculate\_interest() method to calculate interest based on the*

*balance and interest rate.*

*7. Create CustomerServiceProviderImpl class which implements ICustomerServiceProvider*

*provide all implementation methods. These methods do not interact with database directly.*

*8. 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*

*9. Create IBankRepository interface/abstract class which include following methods to interact*

*with database.*

*• createAccount(customer: Customer, accNo: long, accType: String, balance: float):*

*Create a new bank account for the given customer with the initial balance and store in*

*database.*

*• listAccounts(): List<Account> accountsList: List all accounts in the bank from database.*

*• calculateInterest(): the calculate\_interest() method to calculate interest based on the*

*balance and interest rate.*

*• getAccountBalance(account\_number: long): Retrieve the balance of an account given*

*its account number. should return the current balance of account from database.*

*• deposit(account\_number: long, amount: float): Deposit the specified amount into the*

*account. Should update new balance in database and return the new balance.*

*• withdraw(account\_number: long, amount: float): Withdraw amount should check the*

*balance from account in database and new balance should updated in Database.*

*o A savings account should maintain a minimum balance and checking if the*

*withdrawal violates the minimum balance rule.*

*o Current account customers are allowed withdraw overdraftLimit and available*

*account balance. withdraw limit can exceed the available balance and should*

*not exceed the overdraft limit.*

*• transfer(from\_account\_number: long, to\_account\_number: int, amount: float):*

*Transfer money from one account to another. check the balance from account in*

*database and new balance should updated in Database.*

*getAccountDetails(account\_number: long): Should return the account and customer*

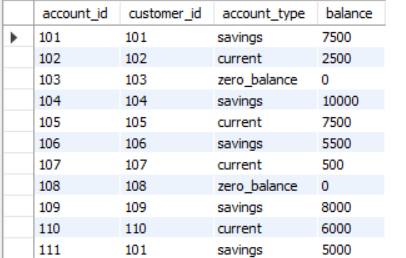
*details from databse.*

*• getTransations(account\_number: long, FromDate:Date, ToDate: Date): Should return*

*the list of transaction between two dates from database.*

*10. Create BankRepositoryImpl class which implement the IBankRepository interface/abstract class*

*and provide implementation of all methods and perform the database operations.*

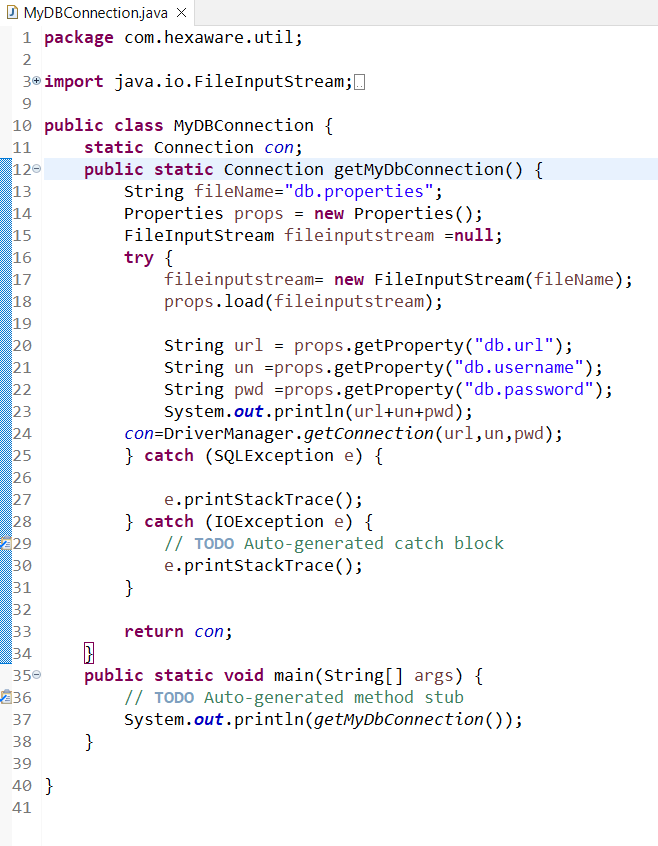




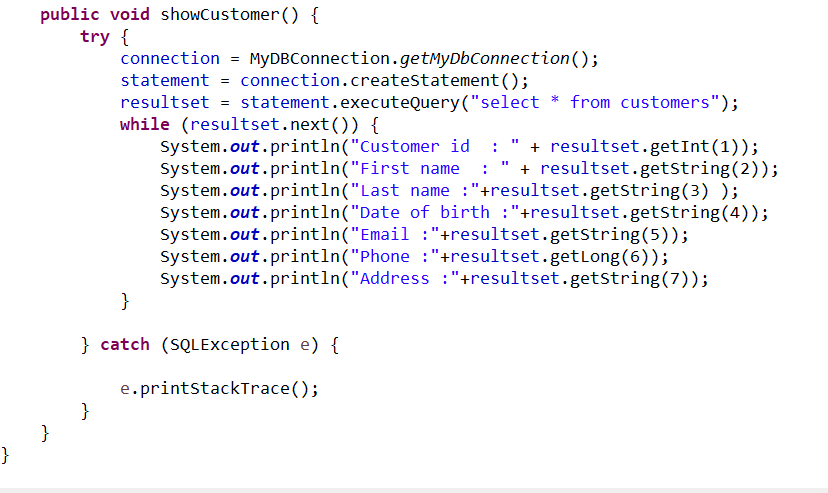
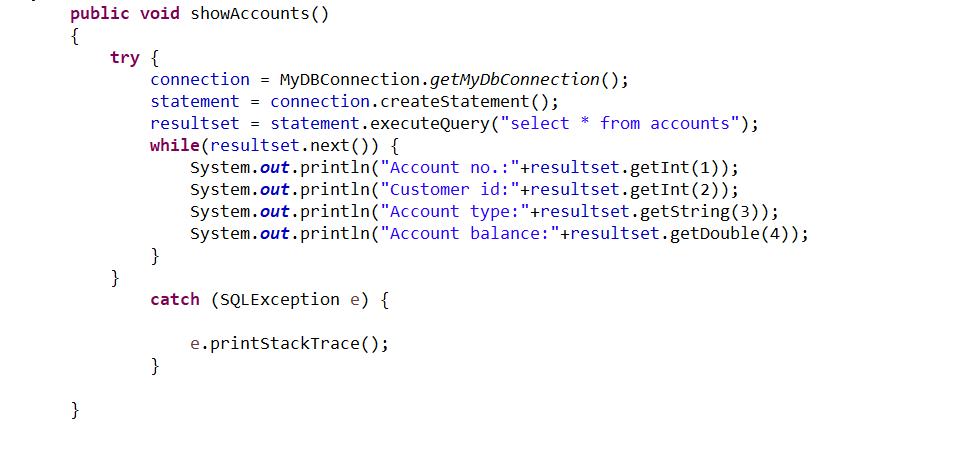
*11. Create DBUtil class and add the following method.*

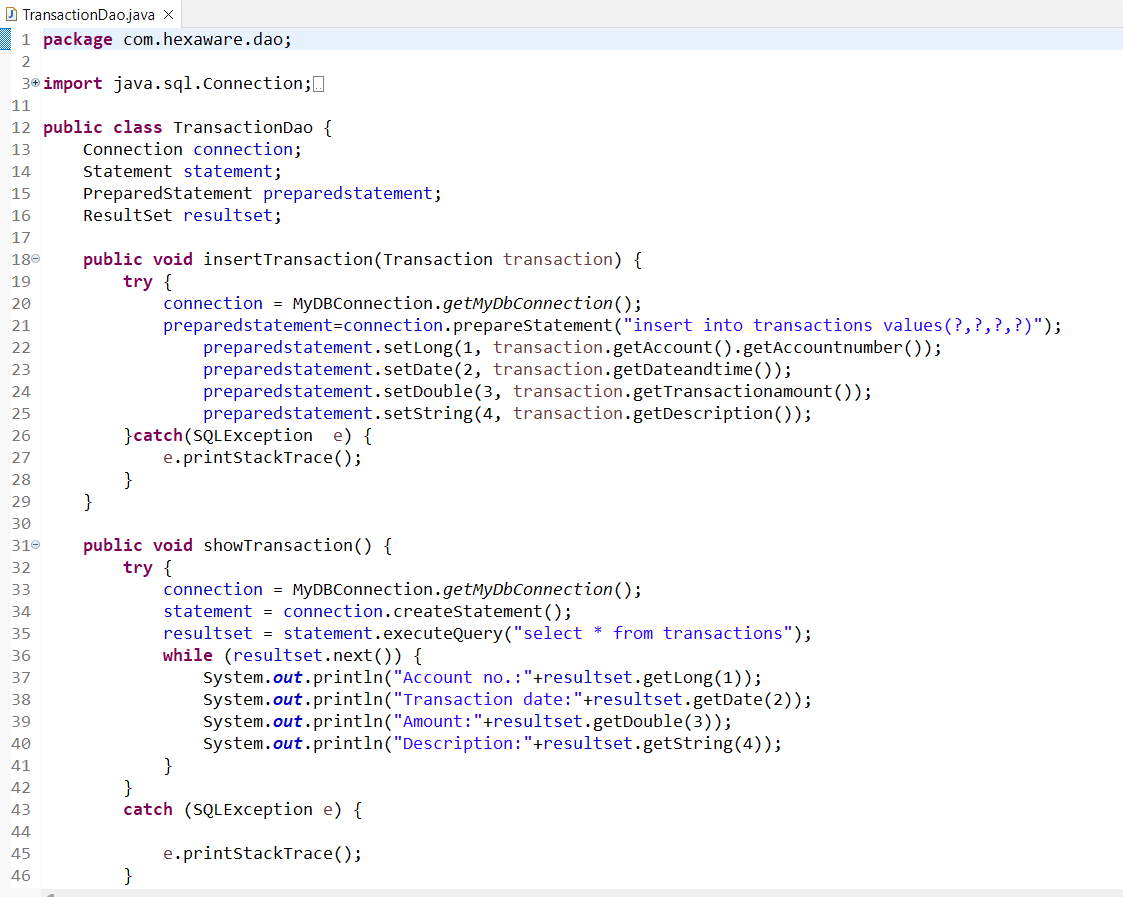
*• static getDBConn():Connection Establish a connection to the database and return*

*Connection reference*







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

*• main method to simulate the banking system. Allow the user to interact with the system*

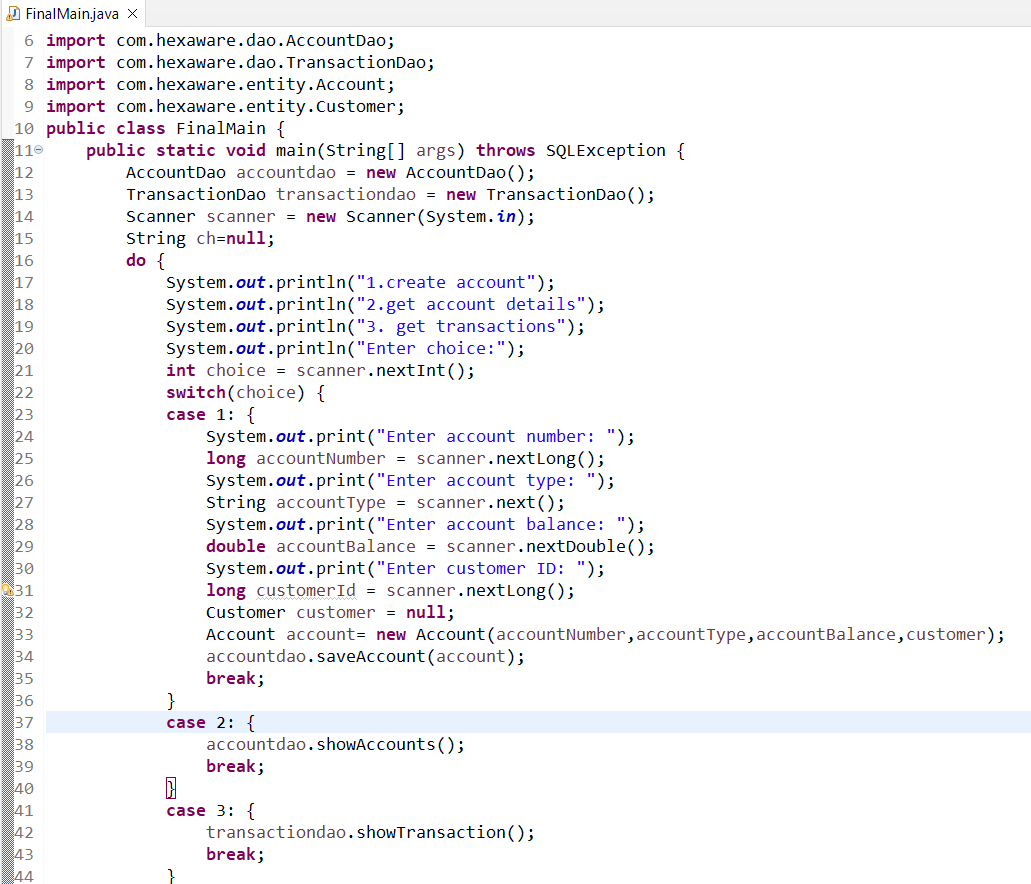
*by entering choice from menu such as "create\_account", "deposit", "withdraw",*

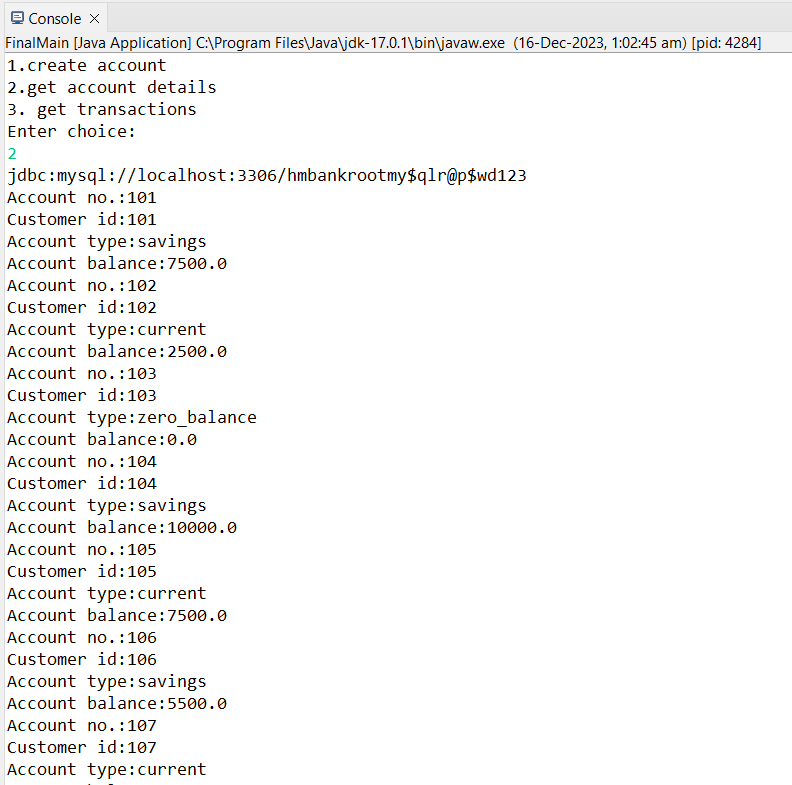
*"get\_balance", "transfer", "getAccountDetails", "ListAccounts", "getTransactions" and*

*"exit."*

*• create\_account should display sub menu to choose type of accounts and repeat this*

*operation until user exit.*





*13. Place the interface/abstract class in service package and interface/abstract class implementation*

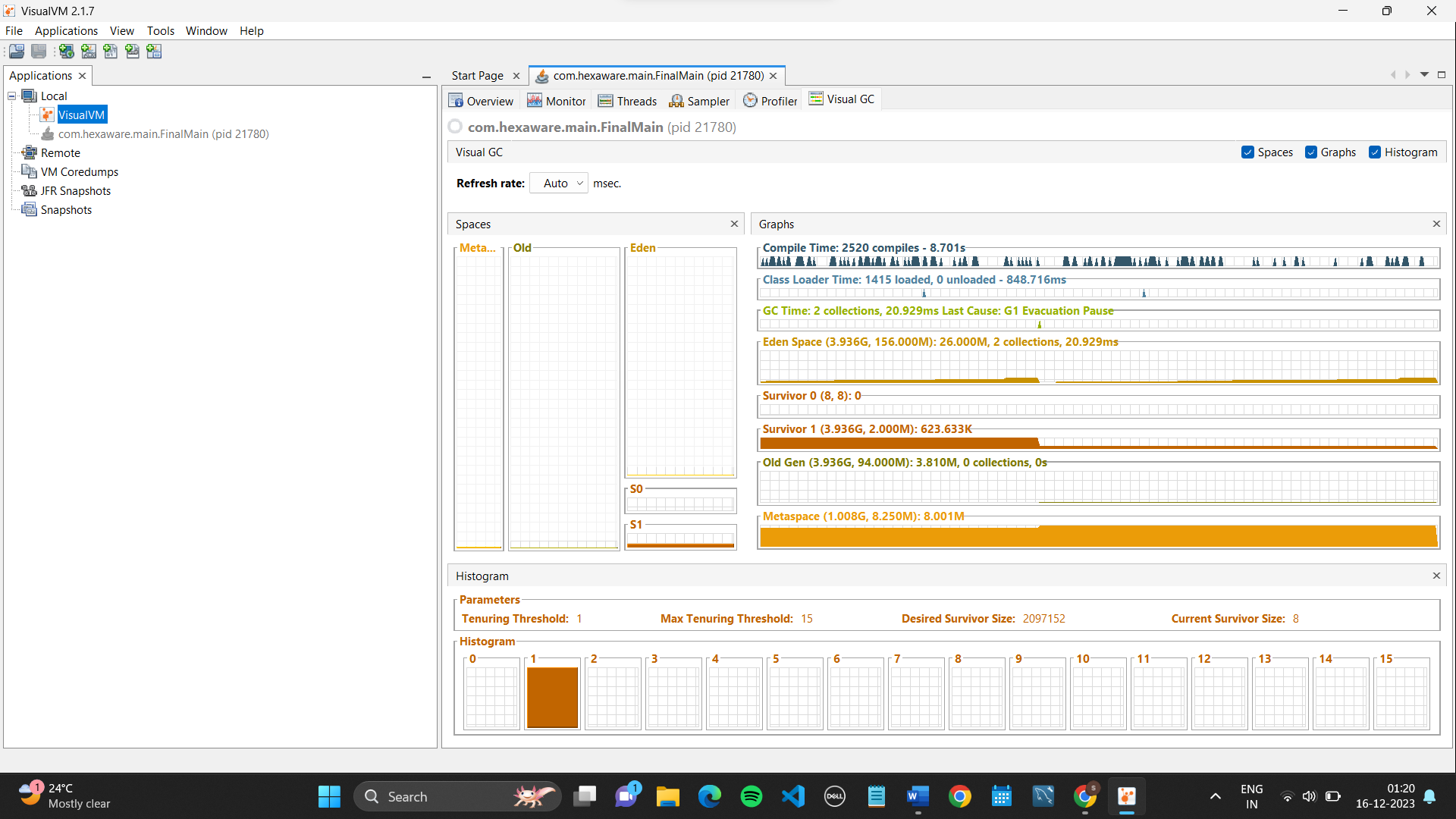
*class, account class in bean package and Bank class in app package.*

*OUTPUT: Same as task 11*

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

*SQLException*

*OUTPUT: Same as task 12*

**