CM0570
Program Design and Development
Assignment 2 of 2
Academic year 2016-17

Dr M J Brockway, Module Tutor

July 2016

Issued Teaching week 6 in classes and on eLP.

Due End teaching week 12, Sunday, December 11, 23:59; work to be submitted electronically via eLP.

Marks and feedback available early semester 2 via e-mail.

Learning Outcomes This part of the assignment assesses, wholly or partly, the following learning outcomes (taken from the Module Descriptor):

3. Understand and employ object oriented programming and make use of collection classes and be able to implement data structures such as lists, queues and trees together with algorithms to process these structures (e.g. searching and sorting).
4. Understand algorithm complexity and the big-O notation, and use this knowledge to select suitable data structures for a particular scenario.

This assignment is weighted at 50% of the marks for this module and wholly an *individual* assignment.

1 Scenario

Bigg City Bank provide banking services in Bigg City. The main things they are concerned with are *Clients* and *Accounts*. There are *Current Accounts* and

Savings Accounts. A client may have at most one of each type of account. There are no other types of account. The following information is stored about a **client**:

- ClientID: an integer; must be unique;
- FamilyName: a string; must not be null or empty;
- FirstName: a string; must not be null or empty;
- HouseNumber: an integer ≥ 1;
- Postcode: a string consisting of two capital letters, one or two digits, a space, one or two digits and two capital letters. Lowercase letters (if entered) should be converted to the corresponding capital letters.

Accounts. Certain information which is stored is common to both types of account:

- AccountID: a string; wholly numeric, 8 digits long. 00000000 (all 0s) is the only invalid combination. A leading zero is significant. Must be unique.
- Owner: the client ID of the client to whom this account belongs. The client ID must actually exist.
- SortCode: a string; two digits, a hyphen, two digits, a hyphen, two digits. None of the pairs of digits can be 00 (although either one of the digits can be a 0, it is not possible for both to be 0).
- Balance: an integer: an amount in pence; may be negative in case of a current account.

For *current accounts*, information is stored on the *overdraft limit*. This is an integer, representing a whole number of pounds. It is set to a value which is specified when the account is created, but may subsequently be changed. The value must be ≥ 0.

No additional information is stored for *savings accounts*, but such accounts pay interest on the balance at the end of each month.

Both types of accounts are associated with a list of *transactions*. A given transaction may only be associated with one account.

Transactions. There are two types of transactions: credits (which add money to an account) and debits (which remove money from an account). The information stored about either type of transaction is as follows:

- AccountID: a string; the account associated with the transaction.
- Amount: an integer: stored in pence but entered in pounds and pence: eg 12.45 is stored as 1245.
- DateAndTime: a Date; the date-and-time when the transaction occurred.

2 Functionality

Bigg City Bank require a system that is capable of performing the following tasks:

1. adding records for clients, validating the information according to the descriptions given;
2. creating a current account or savings account for an existing client;
3. entering transactions; all of the transactions associated with an account are stored.

3 Your Tasks

A contract programmer has written a *partial* implementation of the required functionality which you are, for 70 marks, required to extend as specified below. For 30 marks you are asked write on aspects of the implementation.

The partial implementation has the following features.

- Classes `Client`, `Account`, `CurrentAccount` are provided, encapsulating data in accordance with the specifications.
- A `Transaction` class is provided, encapulating a transaction on an account. There are two attributes: the ID of the account to which the transaction will be applied, and an amount of money in pence: a **positive** value indicates a credit transaction, a **negative** value a debit.
- Classes `MainMenu` and `TransactionDlg` provide a GUI, but need further implementation work from you.
- Client details are loaded from a text file `clients.txt` and stored in a `java.util.Map` in the `MainMenu` class.
- Current account details are loaded from a text file `currentAccounts.txt` and stored in a `java.util.Map` in the `MainMenu` class.
- The main menu has buttons triggering functions for reading in data from the files (to be done once at system startup), listing client details on the console, listing accounts on the console;
- There are also buttons *debit* and *credit*: these both are intended to create *transaction records*, instances of `Transaction` - with a positive amount in case of credits and a negative amount in case of debits. These buttons cause a custom dialogue (a `TransactionDlg` instance) to show, that collects the required data from the user. The user clicks *submit* to (repeatedly) create credit (or debit) transactions, and *hide* to return to the main manu. However, the functionality to actually validate the data, create the

transaction records and add these to a list of *pending transactions* has still to be implemented.

- There is a *process transactions* button that does nothing at the moment. It is intended that when clicked, the list of emphpending transactions will be iterated through and the transaction records (credits and debits) applied to the accounts.

This implementation is provided in `Assign2_partial.zip`. You are required to extend it, as below. Note that you are not required to support savings accounts at all - these are beyond the scope of the present assignment.

3.1 Your implementation tasks

In particular you are *not* asked to provide main-menu commands to add a client or account: these can be added by editing the data files for now, and this job is postponed. You *are* asked to ...

1. Complete the functionality to create transactions (debit and credit) on click of the relevant button. In each case, a Transaction object should be created (with a negative amount in case of a debit) and added to a the linked-list of *pending transactions*. The partial implmentation displays the dialogue which collects the required data, but you need to implement the function which validates it, creates the Transaction instance and adds it to the list of *pending transactions*.
2. Add functionality to process the list of transactions by iterating though it. A debit that has a negative amount or will violate the credit limit should cause a message to be displayed on the console (hint: make use of the **CurrentAccount debit** function and its boolean return value); otherwise the transaction should be applied to the account and then *moved* from the *pending transactions* list to the end of the separate *completed transactions* list.
3. Add to the main menu command buttons to list on the console *pending transactions* and *completed transactions*.
4. Add a command button to save data: this should use PrintWriter instances to save updated account records to text file `currentAccount.txt`, and pending transactions to a file `transactions.txt` in a suitable format.
5. Functionality to load client and account data was provided. Extend this so that pending transactions are also reloaded on click of 'Load Data'. Use `java.util.Scanner` and decompose all this functionality into helper functions.

Start from the partial implementation in `Assign2_partial.zip`, downloadable from the web page. Look carefully at the TODO comments - these will guide you.

Code should be of good quality; variables, methods and classes should be appropriately named, and appropriate use made of comments including Javadoc comments.

You may leave the transactions lists as a java.util.LinkedList but may wish to experiment with ArrayList or Set implementations. *You are expected to discuss your choice of data structures in your report.*

3.2 Your report

This is worth 30%, should be around 1000-1200 words and should have two roughly equal sections:

1. Storage of Client and Account data: write a critique of the use of HashMaps by the given partial implementation and discuss possible alternatives. If ArrayLists were to be used, how would the code to access a record with a particular key have to change? (Give a code snippet as an example.) What would be the implications for performance? What is the effect of the *Hash* implmentation of the List interface on the order of storage of records? How might a *Tree* implementation (TreeMap) be different? What other possibilities for storage of client and account data are there?
2. Storage of transaction data: write a critique of the use of a List, and LinkedList in particular. Discuss possible alternatives. Might an ArrayList be a better option? What are the pros and cons. Might a Set implementation (eg HashSet, TreeSet) be feasible? Why/whynot?

You should consider both efficiency, computational complexity, and overall suitability of the data structures you employed and consider reasonable alternatives within the Java collections API that might have been used instead.

4 Marking Scheme

4.1 Implementation

Up to 70 marks will be awarded for the implementation: 50 marks for *functionality* plus 20 marks for *quality of the coding*.

Functionality marks break down as

- listing of pending transactions, completed transactions: 10
- creation of transaction records: 10
- processing of transactions: 15

- saving of account data, pending transaction data; and (re)loading of pending transaction data: 15 marks

Full credit will be given for each aspect that has been correctly implemented. Partial credit will be given for a reasonable but not wholly correct implementation. A mark below half marks will be given if significant features are missing. No credit will be given for features which have not been implemented at all or which do not compile.

Code quality marks will be awarded as follows:

16-20 Good use of commenting throughout, including Javadoc comments for the vast majority of classes, methods and variables. Code is well structured and clear; classes, methods and variables are almost all well named.

12-15 Generally a good attempt, making use of comments, and where the majority of classes, variables and methods have been appropriately named. However there may be several omissions of Javadoc comments, and the code, while generally of good quality, may show some inadequacies of structure.

8-11 A fair attempt; the code is of reasonable quality. However, there may be several problems with structure, or very little use has been made of comments, or the naming of classes, methods and variables is unsatisfactory in a significant number of cases.

4-7 The code is of poor quality, having significant weaknesses in two or more of the following: structure and clarity, use of comments, naming of classes, methods and variables.

0-3 Very poor coding which is hard to understand. Little use of comments. Poor naming of almost all classes, methods and variables.

4.2 Report

24-30 Outstanding report, giving full and detailed coverage of the efficiency and other reasons for the choice of data structure.

20-23 An excellent report, giving comprehensive discussion of the efficiency and other reasons for the choice of data structure. There may be some omissions or errors, but these are minor and do not detract significantly from the report as a whole.

16-19 A good report. Efficiency issues are discussed and so are other reasons for the choice of data structure. However there may be a number of omissions and errors in both sections, and there is an effect on the overall report quality as a result.

12-15 An adequate report. There is coverage of both efficiency and other aspects, but with a number of significant errors and omissions. Alternatively, there is excellent coverage of one of either efficiency or other aspects, but little coverage of the other

10-11 A poor report, but one which still covers either efficiency or other aspects to a reasonable level, but with little coverage of the other of these. Alternatively, a report with many errors and omissions in both sections.

0-9 Wholly inadequate, with little attempt to justify the choice of data structure either on the basis of efficiency or any other reasons.

4.3 Academic Misconduct

This is *individual* work and must be solely your own. Any sources you make use of must be referenced correctly using the guidelines in *Cite Them Right*, available on the University website. Any plagiarism or collusion will be dealt with according to University regulations.

5 Handing in your work

You should submit the following electronically, using the links provided on Blackboard:

1. a `.zip` file containing all your Java source code files. If packages have been used the files should be stored in an appropriate directory structure. The zip file should also include, in addition to the code, a plain text (`.txt`) file giving instructions necessary for compiling and running your application. *It should not be necessary for an IDE to be used to run the application. It should be possible to run it by compiling the java sources and launching MainMenu from a commend-line.*
2. your report. This should be submitted via Turnitin on Blackboard using the link provided, and must be in an acceptable format word docx, odt, or (preferably) PDF.

These should be submitted by due date.

If you need to make any assumptions in order to implement the system, these should be detailed in your report, together with the reasons for making them.