University of Glasgow | School of Computing Science

Assessed Coursework

Course Name	**Programming**			
Coursework Number	**1**			
Deadline	**Time:**	**4.30pm**	**Date:**	**13 November 2015**
% Contribution to final course mark	**10**		**This should take at most this many hours:**	**15**
Solo or Group ✓	**Solo**	✓	**Group**	
Submission Instructions	**Via Moodle – see Page 8**			
Who Will Mark This? ✓	**Lecturer ✓**	**Tutor ✓**	**Other**	
Feedback Type? ✓	**Written ✓**	**Oral**	**Both**	
Individual or Generic? ✓	**Generic**	**Individual**	**Both ✓**	
Other Feedback Notes				
Please Note: This Coursework cannot be Re-Done				

Code of Assessment Rules for Coursework Submission

Deadlines for the submission of coursework which is to be formally assessed will be published in course documentation, and work which is submitted later than the deadline will be subject to penalty as set out below. The primary grade and secondary band awarded for coursework which is submitted after the published deadline will be calculated as follows:

(i) Work submitted not more than five working days after the deadline will be assessed in the usual way. The primary grade and secondary band so determined will then be reduced by two secondary bands for each working day (or part of a working day) the work was submitted late.

(ii) Work submitted more than five working days after the deadline will be awarded Grade H.

Penalties for late submission of coursework will not be imposed if good cause is established for the late submission. You should submit documents supporting good cause via MyCampus.

Penalty for non-adherence to Submission Instructions is 2 bands.

MSc / PGDip Information Technology / Software Development

Programming – 2015-16

Assessed Exercise 1

Handed out: Tuesday 27 October *Due*: Friday 13 November at 4.30pm

This exercise is worth 10% of the overall mark for the Programming course

Application Specification

Lilybank Wine Merchants sells bottles of wine at a variety of prices on a sale or return basis. Their customers are allowed to trade on credit (that is, they are allowed to build up a debt to the wine merchant which can be settled later), and have a Customer Account record in which details of the balance owed to the wine merchants are stored. The wine merchants want to automate their invoicing procedure. They require an interface that will allow a sales assistant to enter details of a series of sales and/or returns, and update the customer balance as appropriate.

A single sale or a single return involves one or more bottles of a single type of wine at a single price. Suitable details should be input by the sales assistant who should then click a button to indicate that the transaction is to be processed. The interface should then display the cost of the sale (or the amount to be refunded in the case of a return), and the current balance (which is the total cost of all sales and returns to date, plus the initial customer balance).

The details to be input by the sales assistant are:

- the type of wine – its 'name';
- the number of bottles purchased or returned;
- the price of one bottle of that type of wine.

There should be two buttons: one to signal that a sale is to be processed, and the other to signal that a return is to be processed. Pressing either button indicates to the program that the user has entered the correct details as per the list above.

Details should be input through text fields, with informative labelling. Calculated costs should also be displayed in text fields, but these should be set as un-editable. A sample suitable GUI interface is displayed below:

This example shows the situation for customer Jane Smith, with an initial credit balance of £17.49 (that is, the wine merchant owed her £17.49 at the outset), and after a purchase of 15 bottles of champagne at £18.99 per bottle. Note that the initial credit balance is no longer showing, and the wine name, quantity and price details were cleared from the text fields when the sale button was pressed.

Functionality to be provided by the program

1. On running the program, the name of the customer and the customer's current balance (the amount currently owing to the wine merchant) are to be obtained via two input dialog boxes (from the `JOptionPane` class). If the user closes either dialog box or presses 'Cancel', then the program should terminate. Otherwise, the first dialog box should return a `String` value which indicates the name of the customer. If this string is empty, again the program should terminate. Otherwise the customer's name should appear in the title bar of the main GUI window.

   The second dialog box should return a `double` value representing the current balance in the customer account. If the user supplies non-numerical input, then an error message should be displayed via a separate dialog box and the dialog box requesting the balance should reappear, giving another chance to supply data in the correct format. This should continue until the balance has been entered correctly (unless the program terminates due to the dialog box being closed or cancelled). Note that each execution of the program involves just a single customer.

   Negative initial balances correspond to the case that the customer is in credit (i.e., the wine merchant actually owes the customer money). The initial balance should be confirmed to the user (e.g., in the "Current balance" text field as in the above screenshot) when the main GUI window initially displays.

2. The user should be able to process a sales transaction at any time. Sales are implemented by entering:
   - the name of the wine;
   - the number of bottles to be purchased;
   - the price of a bottle of the wine;

   and then pressing the 'Process Sale' button. When valid data is entered, the cost of the transaction should be calculated as:

   $$\text{number of bottles} \times \text{cost of one bottle}$$

   and the customer account balance should be increased by the cost of the sale.

   The "name", "quantity" and "price" textfields should be cleared after processing a sale.

3. The user should be able to process a return transaction at any time. Returns are implemented by entering:
   - the name of the wine;
   - the number of bottles to be returned;
   - the price of a bottle of the wine;

   and then pressing the Process Return button. When valid data is entered, the refund given by the transaction should be calculated as:

   $$\text{number of bottles} \times \text{cost of one bottle} \times (1\text{-service charge})$$

and the customer account balance should be decreased by the refund amount.

Currently Lilybank Wine Merchants levy a 20% service charge for accepting returned goods. The program should round all prices to two decimal places (so returning a single bottle of wine costing £3.97 should cause the balance to be decreased by £3.18).

The "name", "quantity" and "price" textfields should be cleared after processing a return.

4. When a valid sale or return is processed, the cost of the sale or the amount to be refunded should be displayed in the appropriate text box (labelled 'Amount of transaction' in the sample GUI) – you are not required to display a negative sign in the case of a return. The value of the current total balance of the customer account should be displayed in the text box labelled 'Current balance'. If the current balance is negative it should NOT be displayed with a 'minus sign', but instead should be displayed as a positive value followed by the characters 'CR'. (This means that the customer is in credit, i.e., the wine merchant actually owes the customer money.)

   The name of the wine that has been purchased / returned should be confirmed back to the user, e.g., along the lines of the example shown on Page 2.

   Ensure that all financial quantities are displayed to two decimal places (note: the user need not be forced to use as many as two decimal places when supplying the price of a wine bottle). Also, ensure that Current Balance reflects the initial balance plus/minus *all* the sales and returns that have been carried out during the program's execution (and not just the initial balance plus/minus the value of the most recent sale or return, for example).

5. If invalid data is entered for either the price or the number of bottles (i.e., the price is not a positive double, or the quantity is not a positive integer) then an error message should be displayed using a `JOptionPane` component and no further processing should take place for that transaction (however there is no need for the program to terminate in such a case). No validation is expected with regard to the name of the wine, except that the program should check that the wine name is non-empty. The "name", "quantity" and "price" textfields should be cleared if invalid data is entered.

Getting started

Launch Eclipse. You should ensure that your workspace is set to `H:\workspace\Prog`. Use "Switch Workspace->Other" under the File menu to check your workspace location. If you need to change it, note that Eclipse will restart.

Create a new Java project called AE1 in your workspace. You must ensure that in the project creation wizard, you choose the option "use project folder as root for sources and class files". If you find that folder AE1 has sub-folders src and bin, you have chosen the wrong option and should delete the project and create it again.

Program Design

Your program should consist of the following four classes:

The class ***CustomerAccount*** should define the Customer Account object. Its purpose is to store the customer's name and current balance, and to perform operations on the latter variable to update it in the case of a sale or return. It is one of the Model classes in the Model – View – Controller style of programming.

It should have a class constant representing the percentage service charge for returns. The types of the instance variables will be (you must choose your own names):

```
int       // represents the current balance
String    // represents the name of the account holder
```

Regarding the type of the variable to store the current balance, for the purposes of this exercise, full credit will be given if `double` is used, however as was the case in an earlier lab exercise, the best practice is to use `int` for financial computations, so here the balance could be represented by a variable of type `int` that gives the number of pence in the current balance.

Methods will be

```
Constructor: initialises the instance variables

A method to process the sale of numBottles bottles of wine with cost
per bottle costBottle.
It updates the account balance.
It returns the total cost of the transaction as a double.

A method to process the return of numBottles bottles of wine with
cost per bottle costBottle.
It updates the account balance.
It returns the total cost of the transaction as a double.

Methods to return the values of the instance variables.
```

The class ***Wine*** should be used to represent information about a wine type that is purchased / returned. It is another Model class in the Model – View – Controller style of programming.

The types of the instance variables will be (you must choose your own names):

```
String    // represents the name of the wine
double    // represents the price of a bottle of the wine
int       // represents the quantity of the wine
```

There will be a constructor to initialise instance variables, together with methods to return their values.

The GUI class ***LWMGUI*** will provide the user interface and methods to handle events. It is the View / Controller class. It will contain methods for laying out the components, listening for events, and processing sales and returns.

When the user presses either the sale or return button, the details that the program needs to capture from the textfields are the same regardless of the operation to be performed. Thus a

single method can be used to create a `Wine` object, which can then be passed to further methods to process the sale or return as appropriate.

The class ***`AssEx1`*** will contain the main method. Its purpose will be to (i) get the customer name and starting balance from the user via two input dialog boxes, (ii) create a `CustomerAccount` object on the basis of this information, and (iii) instantiate and display a `LWMGUI` object, passing the `CustomerAccount` object as a parameter.

Note that the first parameter of any call to a `JOptionPane.showInputDialog` constructor should be `null` when this constructor is called from the main method of the class `AssEx1`.

Hints on Development

You should design your program incrementally, testing each method as it is developed. When you have got a particular stage working make sure that you save a copy of your working file under a different name so that you can revert to an older working version if necessary.

(A) First design `LWMGUI` without the `CustomerAccount` or `Wine` classes and without any serious event handling methods. Write a version of the `actionPerformed` method that simply prints out a confirmation of which button has been pressed to the system output window and test that each button click obtains the response you expect.

(B) Create the `Wine` class. A `Wine` object will be created and returned by a method in `LWMGUI` that is responsible for getting details from the textfields when either the sale or return button is pressed. This object will then be passed to the relevant method in `LWMGUI` that is responsible for processing the sale or return as appropriate.

(C) Write the `CustomerAccount` class. Create a `CustomerAccount` object in the `main` method of `AssEx1` by getting relevant information from the input dialog boxes. Pass it to the `LWMGUI` constructor and store it in an instance variable of the latter.

(D) Implement the sales and returns functionality. These should be developed independently. Sales are marginally simpler than returns, so it might be better to organise sales processing first.

Write the methods in the `CustomerAccount` class to process the transactions. These methods should

- calculate and return the cost of the transaction;
- update the value of the balance.

In the `LWMGUI` class, the event handler code should use methods in the `CustomerAccount` class to process the transactions, and the event handler code should update the GUI accordingly.

Testing

You are responsible for creating various test data and checking that your program executes correctly when given this input. You will naturally be testing functionality as you develop the program using statements such as `System.err.print`. However you should not submit

the output of such statements, and moreover you should ensure that these statements are removed from your submitted code. Instead you should submit a few screen dumps of your final results, with each screen dump chosen to demonstrate the correctness of as many aspects as possible of the functionality – see under 'Report' for more details regarding this.

Report

Write a short summary (no more than one page, excluding screen dumps) of the final state of your program in a Word file. It should indicate how closely your program matches the specification. There should be a statement detailing the inputs that you have used to test that the program is working correctly before submitting the final version. This statement should include a description of the results you would expect when executing the program with your data. The report should include:

- Your ***name and student number*** at the top.
- Any assumptions you have made that have affected your implementation, e.g., the length of a wine name might be an assumption you need to make.
- Any known deficiencies, e.g., missing functionality or discrepancies between your expected and actual results and how you might correct them. If your program does meet the specification, all that is needed is ONE sentence stating this.
- Details of the test data used and the results expected and obtained should be provided. Read the requirements carefully. One screen dump can provide proof of several aspects of functionality You should test the following aspects of the functionality and provide corresponding screen dumps:

Function	*Proof*
Program gets customer name and initial balance from input dialog boxes.	Screen dump of the GUI before any transactions have been made.
Program validates number input from the GUI.	One screen dump of a GUI with invalid data and a JOptionPane error message
Program calculates the amount of a single transaction correctly, and displays the amount and the updated balance correctly in the GUI	Screen dumps of the main GUI after one transaction (e.g., a sale) leaving a normal balance and one transaction (e.g., a return) leaving a credit balance

Use ALT+Print Screen to capture a screenshot of the relevant active window from your program, and then dump the screenshot to Word using CTRL-v.

Code listing document

Produce a pdf file containing your code listing. This should be done as follows. Double-click on the "Cygwin" icon on the desktop, and enter the following commands:

```
cd H:/workspace/Prog/AE1
code2pdf –o AE1 *.java
```

You will find that `AE1.pdf` appears in your `AE1` folder. This will contain a listing of all of your .java files. If you have extraneous .java files in that folder that do not form part of your

submission, simply remove them before issuing the above commands. Alternatively, you can do the following:

```
cd H:/workspace/Prog/AE1
code2pdf –o AE1 AssEx1.java LWMGUI.java CustomerAccount.java
Wine.java
```

Note that these instructions apply to the lab machines. Creating acceptable pdf output from a PC when off campus is possible (though done at your own risk): see the Programming Moodle page for more information.

Submission

All submission for this exercise is to be done electronically – no hard-copy submissions are required. You need to submit a zip file containing the following:

- Your report on the program, with screenshots, contained in `AE1report.doc` or `AE1report.docx`
- Your code listing, contained in `AE1.pdf`
- Your Java source files:
  - `AssEx1.java`
  - `LWMGUI.java`
  - `CustomerAccount.java`
  - `Wine.java`

To create the zip file, select the above files (either by choosing them from within your `AE1` folder, or by selecting all files from within another folder where they are the only files present). Note that, to select multiple files, hold down the Control key whilst left-clicking on the file icons. Then right-click on one of the highlighted files and choose "7-Zip -> Add to archive". From the wizard, choose "zip" as the archive format and ensure that the filename is `AE1_<family name>_<given name>.zip`. (e.g., `AE1_Smith_Jane.zip`). You should find that `AE1_<family name>_<given name>.zip` has now been created. You can double-click on this file to check its contents.

On the Programming Moodle page (http://moodle2.gla.ac.uk/course/view.php?id=1831), you will find under the Section labelled "Assessment" a small submission box icon labelled "Assessed Exercise 1". Click on the link. Follow the on-screen instructions to upload your zip file `AE1_<family name>_<given name>.zip`.

You can submit versions as many times as you like prior to the deadline. As a result it would be sensible to try out the submission mechanism with early versions of your code well before the deadline just to ensure that you are familiar with the process, to avoid any last-minute panic.

Notes:

- Do NOT alter the files in `Workspace\Prog\AE1` after the deadline, so that if there is a problem with the zip file then you can resubmit your files, and their date of last modification will still be prior to the deadline.
- If any print statements have been included for debugging purposes, please remember to comment them out before submission.

Techniques

The code constituting your solution to this exercise should involve only techniques that have been covered in the Programming course. ***That is, classes or methods that have not been covered in the course should not be used.***

Marking Scheme

The assignment will be marked in bands on the scale A1,A2,……..F3,G1,G2,H. Marks will be awarded according to the quality of three separate elements:

Element	**Weighting**	**Contents**
Functionality	40%	Proportion of the specified functionality implemented, and program correctness.
Design	34%	Choice of methods and instance variables within classes, design of methods
Documentation and Testing	26%	Readability of code (comments, layout, indentation), quality of status report, selection of test data

Notes:

- *We are expecting you to stick rigidly to the specification. Don't spend time implementing functionality that was not asked for – it will not receive additional credit and your time would be better spent on other tasks!*
- *If major aspects of the functionality are not even attempted then the credit given for the Design, Documentation and Testing will be reduced. So for example if only half of the functionality is attempted then it will only be possible to obtain half marks for Design, Documentation and Testing.*

Bands will awarded according to the following broad criteria:

A1,A2,A3,A4,A5	**Excellent** in all three elements. At the A5 level there may be minor errors in several elements
B1,B2,B3	**Very Good**. Possibly not all functionality implemented but the design and documentation of what was done was very good
C1,C2,C3	**Good** Definitely of MSc standard but too many errors to attract a B level band. A fully functional program with poor design and documentation would only attract a C level band
D1,D2,D3	**Satisfactory** Diploma standard. Maybe some major aspect of functionality missing and weak design or documentation
E1,E2,E3	**Weak** Some functionality but not much
F1….G2	**Poor** Very little done
H	Nothing submitted