Practical Task 3.2

(Pass Task)

General Instructions

In this practical task, you will continue your work on developing the banking system that you began in Task 2.2. You will refine the functionality of the `Account` class and add some important validation to its `Withdraw` and `Deposit` methods. In addition, you will focus on the use of control flow in order to implement a primitive user interface (UI) for the program.

Create a new C# Console Application project. You should import the `Account` class written earlier as part of Task 2.2. Another new class must replace the default `Program` class. Name it as `BankSystem`. Its purpose is to test the `Account` class and also provide the UI to control an associated account.

1. Because the user should not be able to deposit a negative amount of money, you need to add a simple check to ensure that the input of the `Deposit` method is greater than zero. Indeed, the user must know whether the deposit operation was successful or not. This can be achieved by changing the output of the method. It should now return a value of type ***boolean***. That is, if the amount to deposit is valid, then we change the `_balance` and return `true` to indicate that this succeeded. Returning `false` means that the deposit failed. The caller can then use the result returned for their own purpose.

   Similarly, you must ensure that the user cannot withdraw more funds than they have, and that they cannot withdraw a negative amount. This will require you to alter the `Withdraw` method of the class.

2. You now can focus on the `BankSystem` class to implement the UI so that the user gets some control over the affiliated bank account. Create a new ***enumeration*** called `MenuOption`, with the following options:
   - 1. `Withdraw`
   - 2. `Deposit`
   - 3. `Print`, and
   - 4. `Quit`.

   Place this outside of the `BankSystem` class, i.e. either before or after the `BankSystem` class code. Keep the options in that specific order as this way we know that `Withdraw` will map to the integer value 0, `Deposit` will map to 1, `Print` will map to 2, and `Quit` will map to 3.

   Create a new ***static*** `ReadUserOption` method that returns a result of type `MenuOption`. This method will show the menu to the user and read in the choice they made. It should consist of a ***do ... while*** loop, show the user a prompt, and use `Console.ReadLine` and `Convert.ToInt32` methods to read in the user's choice and convert it to an integer. The integer result should be recorded in a variable for later use.

   Note that the value entered by the user cannot be less than 1 or larger than 4. The menu should keep asking the user to choose a valid option from the list until such option is entered. In other words, invalid codes must be ignored by the menu and it must repeat the possible options again. The `ReadUserOption` method must return a valid code that matches the enumeration.

3. Now, edit the `Main` method and have it respond to the option the user selected. Add in a ***switch statement***, with a case for each of the `MenuOption` values. Do not forget to confirm the option selected by the user in `ReadUserOption` by printing a message to the terminal. One by one, add functionality to support the **deposit**, **withdraw**, and **print** operations on the account.

   For each option create a separate ***static*** method. Each method must take an object of the `Account` class as a parameter. The three methods are as follows:

   - **void DoDeposit(Account account)**

     Performs a deposit operation on the specified `account`. The method requests for the amount that the user wants to deposit, then forwards the request to the `account`.

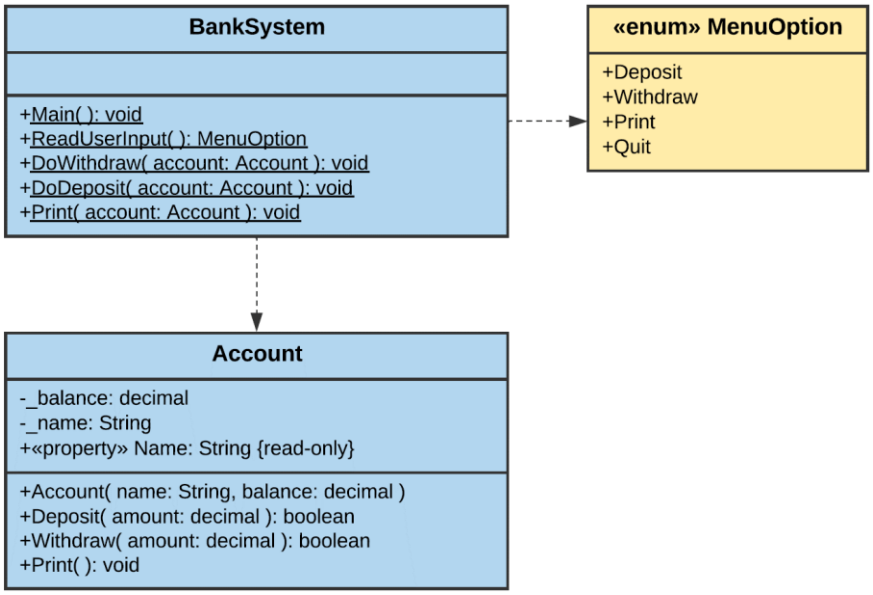
- **void DoWithdraw(Account account)**

  Performs a withdraw operation on the specified `account`. The method requests for the amount that the user wants to withdraw, then forwards the request to the `account`.

- **void DoPrint(Account account)**

  Calls `Print` on the specified `account`.

  The following UML diagram should help you to understand the design of the current version of the banking system.



Note that in `DoDeposit` / `DoWithdraw` you may need to use a decimal local variable to store the amount to deposit or withdraw, before you pass it to the `Account` object to process. Use the Boolean value returned by the `Withdraw` or `Deposit` method to let the user know if the call was successful.

4. Complete and test your program for potential logical issues and runtime errors.
5. Prepare to discuss the following with your tutor:
   - Explain how selection is used within the Accounts and for the menu.
   - Why use an enumeration for the menu? What are the advantages, and other approaches?
   - How is repetition used in the program? What capabilities does this give us?
   - How can control flow help increase the capability of our objects?

Further Notes

- Study the way to program and use classes and objects in C# by reading Sections 2.3-2.6 of SIT232 Workbook available in CloudDeakin → Resources.
- Section 2.2.3 of the Workbook gives a sketch of how to complete the UI asked in this task. Additionally, Section 2.7 introduces enumerated data types that you will need to implement the MenuOption enumeration.
- If you seriously struggle with the remaining concepts used in this practical task, we recommend you to start reading the entire Sections 1 and 2 of SIT232 Workbook.
- In this unit, we will use Microsoft Visual Studio 2017 to develop C# programs. Find the instructions to install the community version of Microsoft Visual Studio 2017 available on the SIT232 unit web-page in CloudDeakin in Resources → Software → Visual Studio Community 2017. You however are free to use another IDE, e.g. Visual Studio Code, if you prefer that.

Marking Process and Discussion

To get your task completed, you must finish the following steps strictly on time.

- Make sure that your program implements the required functionality. It must compile and have no runtime errors. Programs causing compilation or runtime errors will not be accepted as a solution. You need to test your program thoroughly before submission. Think about potential errors where your program might fail.
- Submit the expected code files as an answer to the task via OnTrack submission system. Cloud students must record a short video explaining their work and solution to the task. Upload the video to one of accessible resources, and refer to it for the purpose of marking. You must provide a working link to the video to your marking tutor in OnTrack.
- On-campus students must meet with their marking tutor to demonstrate and discuss their solution in one of the dedicated practical sessions. Be on time with respect to the specified discussion deadline.
- Answer all additional questions that your tutor may ask you. Questions are likely to cover lecture notes, so attending (or watching) lectures should help you with this **compulsory** interview part. Please, come prepared so that the class time is used efficiently and fairly for all students in it. You should start your interview as soon as possible as if your answers are wrong, you may have to pass another interview, still before the deadline. Use available attempts properly.

Note that we will not accept your solution after the submission deadline and will not discuss it after the discussion deadline. If you fail one of the deadlines, you fail the task and this reduces the chance to pass the unit. Unless extended for all students, the deadlines are strict to guarantee smooth and on-time work throughout the unit.

Remember that this is your responsibility to keep track of your progress in the unit that includes checking which tasks have been marked as completed in the OnTrack system by your marking tutor, and which are still to be finalised. When marking you at the end of the unit, we will solely rely on the records of the OnTrack system and feedback provided by your tutor about your overall progress and quality of your solutions.