Practical Task 6.2

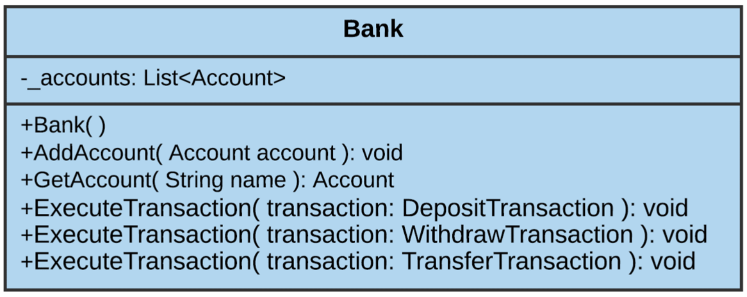
(Pass Task)

Submission deadline: 10:00am Monday, May 4
Discussion deadline: 10:00am Saturday, May 16

General Instructions

This practical task is an extension of Task 5.2. This time, you will need to add multiple accounts to the banking system which you are developing. It will require you to introduce a Bank class, a new entity capable to store and manage a collection of associated bank accounts. For your code solution, you should create a new C# Console Application project but import the classes completed in the previous iteration of the system.

1. Add the Bank class to the project and make sure that it meets the design illustrated by the UML diagram below.



```
Bank
-_accounts: List<Account>
+Bank( )
+AddAccount( Account account ): void
+GetAccount( String name ): Account
+ExecuteTransaction( transaction: DepositTransaction ): void
+ExecuteTransaction( transaction: WithdrawTransaction ): void
+ExecuteTransaction( transaction: TransferTransaction ): void
```

You should record related accounts in the designated `List<Account>` data collection. To satisfy the design requirements, implement the following two methods:

– **void AddAccount(Account account)**

  Adds the specified account into the list of affiliated bank accounts.

– **Account GetAccount(String name)**

  Searches through the list of affiliated bank accounts and returns the first account whose name matches the given string argument. This method returns ***null*** if no account matches the searching criteria.

Note that the ***null*** value employed in the second method acts as a special marker to indicate the status of 'no appropriate object'. This is exactly the situation when the searching process must signal about the absence of the required account name. The caller of the `GetAccount` method then has to check whether the return value is different to ***null*** and proceed accordingly.

Notice that the UML diagram lists other three methods, with all three having the same name. Specifically, the `ExecuteTransaction` method has a different argument type in each of its versions. This is known as ***Method Overloading***, a concept that you should have already encountered in Task 5.1. When the `ExecuteTransaction` method is invoked, the right version is identified by determining the type of the argument which is being passed in. As you can see, it accepts the following types:

– `DepositTransaction`,

– `WithdrawTransaction`, and

– `TransferTransaction`.

Indeed, you should keep the same body for all three versions as they must call the `Execute` method on the respective transaction parameter.

2. Now, when you have the Bank class defined, you need to make the Main method to use a new Bank object. Therefore, navigate to the BankSystem.cs and add a new menu option. Designate this option as "Add new account" and make sure that the corresponding code is adjusted so that the user can see, select, and run this option. Specifically, it must do the following.
   - Ask the user for the name of the account and its starting balance.
   - Create a new Account object.
   - Call the AddAccount method to add the new account to the bank presented in the banking system.
3. When the new menu option is tested and ready, you need to produce a mechanism to find accounts for operations provided in other menu options, i.e. in Withdraw, Deposit, Transfer, and Print. This can be realized by introducing FindAccount, a new ***private static*** method placed in the BankSystem class. Its parameter is as follows.
   - **Account FindAccount (Bank bank)**

     Requests the user to enter an account name and delegates the search for the account to the specified Bank object. If no account matches the name, this method notifies the user about the outcome. Regardless the outcome, this method returns the result of the search operations performed by the bank.
4. You now need to change the DoDeposit method in the BankSystem class to make use of the Bank class. In particular, this will ask you to alter the input argument of the DoDeposit, which must be now a Bank class object, not an Account class object. This method then should address to the FindAccount of the bank and examine its result.

   Compile and run your program. Make sure that you can create an account, and deposit money into it.
5. When the DoDeposit works as expected without any flaws, make similar changes to the DoWithdraw, DoTransfer, and then to DoPrint methods. Note that in order to run the transfer transaction, the user will need to specify two accounts: one account to debit and one account to credit. The search operation for both accounts must not return ***null*** to have the corresponding transfer transaction valid.
6. Compile and run the program to make sure that it works correctly. Elaborate on testing and debugging of the new class. Think about how the user can potentially act to crash your program. Focus on possible scenarios.
7. Prepare to discuss the following with your tutor:
   - Explain how the Bank works when you have multiple accounts.
   - Explain how FindAccount works in relation to DoDeposit, DoWithdraw, and DoTransfer. What happens if an account cannot be found in any of these steps?

Further Notes

- Explore Section 3.4.2 of the Workbook to study the concept of method overloading; that is, the way of defining methods with the same name but a different set of arguments.
- Explore Sections 4.2-4.6 of the Workbook to get explanation of how to implement object/class relationships that you will need to code and integrate the Bank class.
- Section 2.2.3 of the Workbook gives a sketch of how to complement the existing UI. Additionally, Section 2.7 introduces enumerated data types that you will need to implement the MenuOption enumeration.
- The following links give some additional notes on method overloading in C#:
  - https://www.c-sharpcorner.com/UploadFile/8a67c0/method-overloading-and-method-overriding-in-C-Sharp/
  - https://docs.microsoft.com/en-us/dotnet/standard/design-guidelines/member-overloading

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.