Practical Task 7.1

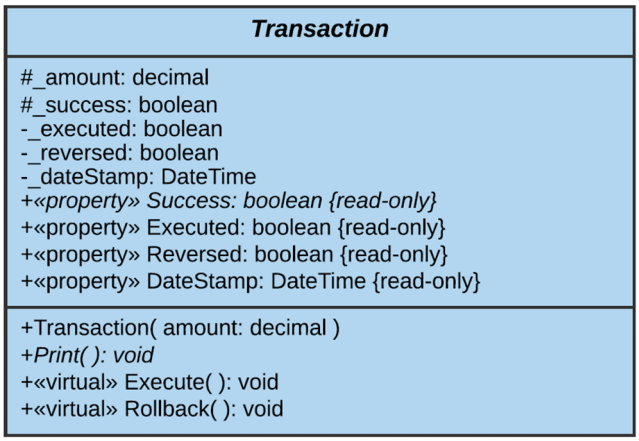
(Pass Task)

Submission deadline: 10:00am Monday, September 13
Discussion deadline: 10:00pm Friday, October 1

General Instructions

In this practical task, you will take a look at the banking system again and create an ***abstract*** `Transaction` class for which the `DepositTransaction`, `WithdrawTransaction`, and `TransferTransaction` classes will inherit from. By doing this, you will reduce code duplication and simplify the code in certain parts of your program. This task finalizes the project on the banking system.

1. To start your work, use the existing project that you should have already completed in Task 6.2. Extend the project by adding the required `Transaction` class and the corresponding source code file. This class must be ***abstract*** and meet the design illustrated by the following UML diagram.



   Make sure that you interpret the above diagram correctly as this will immediately affect your program code. For example, making a method's signature written in an ***italic font style*** has a special sense. Spend some time to understand why certain attributes are now marked as ***protected*** and why several properties and methods are specified as ***abstract*** or ***virtual***. Think about how these facts may impact a ***derived (child) class*** of the `Transaction` class.

   Note the following facts about the `Transaction` class. The `Transaction`'s constructor is parameterized by a single argument and its purpose is to set the initial value of the `_amount` field. The `Print` method is just a placeholder that different transactions can ***override*** and specify (i.e., they can provide a particular implementation). Similarly, the `Success` is a placeholder for the property to be defined in derived transactions to indicate whether the respective operation has been successfully completed.

   The new `_datestamp` field and its associated `DateStamp` property are introduced to record the time when the latest of the two transaction's operations (i.e., the `Execute` or the `Rollback`) was attempted. You can get the current time using the `DateTime.Now` property of the `DateTime` native .NET class. This implies that the `Execute` method may not only update the `_executed` field to **`true`**, but also set the `_datestamp` to the current time. The `Rollback` method may operate similarly to the `Execute`.

2. Now, modify the `DepositTransaction`, `WithdrawTransaction`, and `TransferTransaction` classes to make them inheriting from the `Transaction`. The structure of these classes should not change significantly; so, you only need to make it working correctly with the new `Transaction` class.

   Note that the constructor in each of these classes should call the ***base class*** constructor, feeding it with the amount as the input, and then, in its body, it must set up the account(s) as necessary.

   The `Execute` method should ***override*** the method from the **base** class. Its code should first call the `Execute` method of the base class and then proceed with the transaction itself with regard to the given account. The `Rollback` is similar to the `Execute`, but should reverse the order of operations. Do not forget to implement the other methods and properties for each transaction class adhering to the design of the ***base (parent) class***.

3. Now, when the three types of transactions are all inheriting from the `Transaction` class, you can record a list of transactions associated with the bank. Modify the `Bank` class by adding a new **private** field `_transactions` of `List<Transaction>` type.

   Replace the three `ExecuteTransaction` methods with a single method that accepts (as the input) a transaction of the base `Transaction` class. When called, this method must add the transaction to the list of transactions and invoke the `Execute` method on that transaction.

4. Add a `RollbackTransaction` method, which is to act similarly to the `ExecuteTransaction`. It must take an instance of the `Transaction` class as an argument and attempt to reverse it via the call of the `Rollback` method.

5. Finally, create a public void `PrintTranscationHistory` method that iterates through the `_transactions`, asking each one to `Print`. Note that the `Print` method should state the status of the transaction along with the timestamp of the last operation on it.

   Add a menu option in the `BankSystem` to print the numbered transaction history through the call of the `PrintTranscationHistory` method. Once the whole list is displayed, ask the user whether he/she wants to rollback a specific transaction. This functionality should be implemented by a new method of the `BankSystem` class, e.g. `DoRollback`. Therefore, introduce this method and make sure that it properly serves this option. The user should be able to indicate the transaction via its index in the displayed list. If an attempt to rollback is unsuccessful, i.e. the operation throws an exception, the user must be notified about it. This should return the control back to the main menu.

6. Check the banking system and make sure that all operations of the `Bank` class can be executed correctly and the bank does not lose money as different transactions are processed. Test your program for potential logical issues and runtime errors.

7. Prepare to discuss the following with your tutor:
   - Explain how inheritance and polymorphism are used in this solution?
   - How can a single `ExecuteTransaction` method perform any kind of transaction?
   - What changes would you need to make to the Bank to include a new transaction type?
   - What are the advantages we get through inheritance?
   - What advantages does polymorphism provide?

Further Notes

- Explore Section 6.2 of the SIT232 Workbook to learn the concept of polymorphism, how it is achieved, and how to apply it in solving problems. Section 6.3 explains how abstract methods and classes provide the ability to express the desirable semantics of inheritance and how to use them. The Workbook is available in CloudDeakin → Learning Resources.

- Explore Section 7 of the Workbook to learn about how to interpret UML class diagrams.
- The following links give some additional notes on polymorphism, abstract methods and classes, overriding and others aspects that you need to complete this task.
  - https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/protected
  - https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/polymorphism
  - https://www.tutorialspoint.com/csharp/csharp_polymorphism.htm
  - https://www.c-sharpcorner.com/UploadFile/ff2f08/understanding-polymorphism-in-C-Sharp/

Submission Instructions and Marking Process

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

- Make sure your programs implement the required functionality. They 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 programs thoroughly before submission. Think about potential errors where your programs might fail.
- **Submit** the expected code files as a solution to the task via OnTrack submission system.
- Once your solution is accepted by your tutor, you will be invited to **continue its discussion and answer relevant theoretical questions through a face-to-face interview**. Specifically, you will need to meet with the tutor to demonstrate and discuss the solution in one of the dedicated practical sessions (run online via MS Teams for cloud students and on-campus for students who selected to join classes at Burwood). Please, come prepared so that the class time is used efficiently and fairly for all students in it. Be on time with respect to the specified discussion deadline.

  You will also need to **answer all additional questions** that your tutor may ask you. Questions will cover the lecture notes; so attending (or watching) the lectures should help you with this **compulsory** discussion part. You should start the discussion as soon as possible as if your answers are wrong, you may have to pass another round, 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 the quality of your solutions.