Lab 05 - Class Inheritance

Instructions:

- **The lab requires completing a few tasks.**
- **Your submissions must be submitted to the GitHub repository in the Lab05 directory.**
- **Cheating of any kind is prohibited and will not be tolerated.**
- **Violating and failing to follow any of the rules will result in an automatic zero (0) for the lab.**

TO ACKNOWLEDGE THAT YOU HAVE READ AND UNDERSTOOD THE INSTRUCTIONS ABOVE, AT THE BEGINNING OF YOUR SUBMISSION(S), ADD A COMMENT THAT CONSISTS OF YOUR NAME AND THE DATE.

Grading

Task	Maximum Points	Points Earned
1	2.5	
2	2.5	
Total	5	

Note: solutions will be provided for tasks colored blue only.

Your objective is to create an application for a bank that allows users to start and perform transactions on checking and savings accounts. To accomplish your objective, you must create two classes named *Checking* and *Savings*. These classes must inherit (publicly) the *Account* class from lab04 and they **must** contain

- **Checking**
  - A private field named *fee*.
  - A public default constructor that initializes *fee*, *balance*, *accountnumber* to 0, 0 and *acc_num_gen*, respectively, and increments *acc_num_gen* by 1.
  - A public overloaded constructor that takes a float parameter that initializes *fee*, *balance* and *accountnumber* to 0, the parameter and *acc_num_gen*, respectively, and increments *acc_num_gen* by 1. If the parameter is less than 0, *balance* is initialized to 0.
  - A public overridden `deposit()` method. It adds the parameter minus *fee* to *balance*, assigns 0 to *fee*, and returns true if the parameter is not negative and the parameter minus *fee* is not negative; otherwise, it subtracts the parameter from *fee* and returns false if the parameter is not negative; otherwise, it returns false.
  - A public overridden `withdraw()` method. It assigns *balance* the difference of the parameter from *balance* and returns true if the parameter is not negative and *balance* minus the parameter is not negative; otherwise, it adds 5 and the parameter minus *balance* to *fee*, assigns 0 to *balance* and returns false if the parameter is not negative and *balance* minus the parameter is negative.
  - A public getter method for *fee* named `Fee()`.
  - A public overridden toString method or friend ostream operator if applicable, that displays the string

    "*x*:\n$ *y*"

    where *x* and *y* are the values of *accountnumber* and *balance* minus *fee*, respectively.
- **Savings**
  - A private float field named *interest*.
  - A public default constructor that initializes *interest*, *balance* and *accountnumber* to 0, 0, and *acc_num_gen*, respectively, and increments *acc_num_gen* by 1.
  - A public overloaded constructor that takes a float parameter that initializes *interest*, *balance* and *accountnumber* to 0, the parameter and *acc_num_gen*, respectively, and increments *acc_num_gen* by 1. If the parameter is less than 0, *balance* is initialized to 0.
  - A public getter method for *interest* named `interest()`.
  - A public method named `accumulate()` that takes no parameter. It adds the product 1.00426 and the sum of *balance* and *interest* to *interest*.
  - A public overridden toString method or friend ostream operator if applicable, that displays the string

    "*x*:\n$ *y*"

    where *x* and *y* are the values of *accountnumber* and *balance* plus *interest*, respectively.

Afterward, define a function named *app()* that takes no parameters. It prompts a menu to a user until the user chooses the quit option. The other menu options must be to '`start an account`' and '`perform a transaction`'. Likewise, it displays the number of accounts created before the menu.

If the user chooses the '`start an account`' option, it should prompt the user for the type of account and the initial balance. Afterward, it displays the account object created.

If the user chooses the '`perform a transaction`' option, it should initially prompt the user for the account number. If the account is found, it should prompt the user for the type of transaction (deposit or withdraw) and the amount. If the transaction is successful, it should display the new balance of the account; otherwise, it should state that an error occurred.

Last, after each prompt cycle, invoke `accumulate()` for each *Savings* object.

Task 1

Write the above program in C++.

Task 2

Write the above program in Ruby.