SIT771 Object Oriented Development

Pass Task 3.2: Validating Accounts

Overview

For this task, you will pick up from the Bank Account program (2.2P), and add some much needed validation to the Withdraw and Deposit methods, as well as creating a small UI for the program.

Submission Details

Submit the following files to OnTrack.

- Your program code (*Program.cs*, *Account.cs*)
- A screen shot of your program running

You want to focus on the use control flow for validation and user interaction.

Instructions

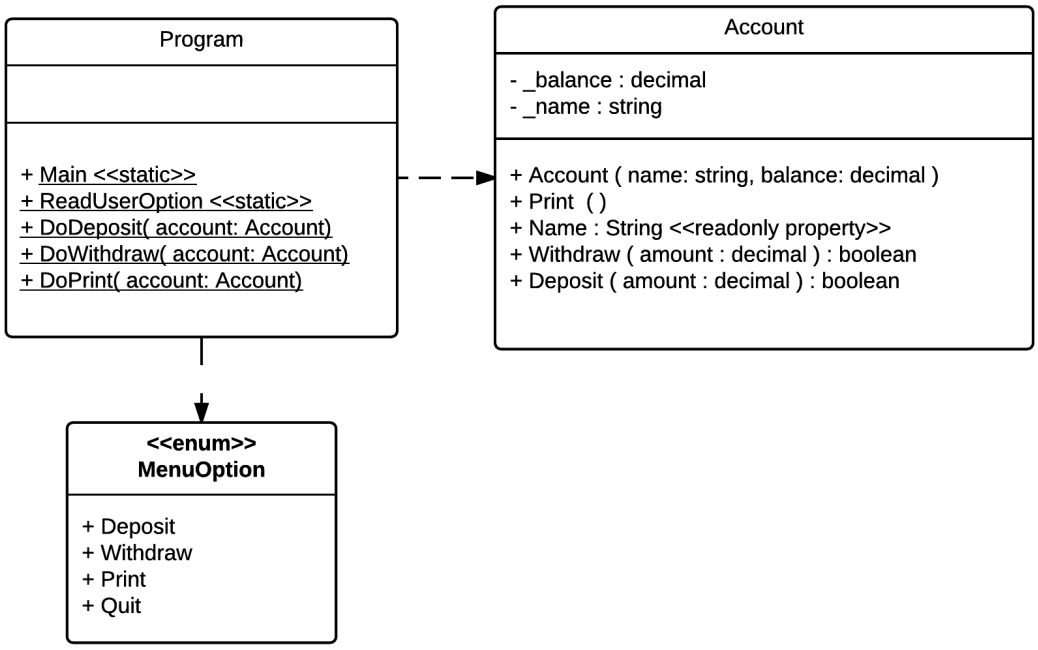

Figure: Iteration 2, showing the Program and Account classes

Let's add some validations to the Deposit method in the `Account` class.

We probably shouldn't be able to deposit negative amount of money! So let's add a simple check to ensure that the `amountToDeposit` is greater than 0:

```
public bool Deposit(decimal amountToDeposit)
{
  if (amountToDeposit > 0)
  {
    Balance += amountToDeposit;
    return true;
  }
  return false;
}
```

We've just added a validation! Now, users can't deposit negative values!

Something else we have changed is that we are now returning a boolean value - this is so that we can check if the Deposit was successful or not! So if the `amountToDeposit` is valid, then we change the `Balance` and return `true` to indicate that this succeeded. Returning false means that the deposit failed.

This is an example of a method that returns a value. The caller can then use the result returned for their own purpose. In this case they will know if the deposit or withdraw worked as expected.

1. Add validations to the `Withdraw` method yourself!

   Ensure that the user cannot withdraw more funds than they have, and that they cannot withdraw a negative amount.

2. Now that we have some validations in the Account class's methods, let's edit our `Program.cs` file so that the user has some control over the bank account. Create a new **enumeration** called `MenuOption`, with the options `Withdraw`, `Deposit`, `Print`, and `Quit`.

   Place this outside of the Program class, either before or after the Program class code. Keep the options in that order, this way we know that `Withdraw` will map to the integer value 0, `Deposit` will map to 1, `Print` will map toi 2, and `Quit` will map to 3. You have done this in task 3.1P.

3. Create a new `static ReadUserOption` method that returns a `MenuOption`.

   This method will show the menu to the user and read in their selection. It should be very similar to the method you wrote in task 3.1P.

1. The `ReadUserOption` method should:
    - create a **do … while** loop
    - Show the user a prompt
    - Use `Console.ReadLine` and `Convert.ToInt32` to read in the user's selection and convert it to an integer. Store this in a variable for later use.
    - End the **do … while** loop, having the above code loop **while** the value entered by the user is less than 1 or larger than 4.
    - Return the matching enumeration

    > **Tip:**
    >
    > See pass task 3.1P for more details on how to write the `ReadUserOption` method.

2. Switch to your terminal and use `skm dotnet run` to compile and test your code.
3. Switch back into VS Code, and open the **Program.cs** file.
4. We can now edit the `Main` method, and have it respond to the option the user selected.
5. Add in a `switch` statement, with a `case` for each of the `MenuOption` values. Start with just a `Console.WriteLine` to print out which option they chose.

   Compile and run to make sure things are working as expected.
6. One by one, add functionality to support the deposit, withdraw, and print options. In each case, do the following:
    - Create a separate static method call to contain the code for this task: `DoDeposit`, `DoWithdraw` or `DoPrint`. This method should accept the account object as a parameter. For example:

      ```
      private static void DoDeposit(Account account)
      {
          //...
      }
      ```

    - Call this new method from within the appropriate **case** of the **switch** statement in `Main`, passing across the account object. For example:

```
//..
case MenuOption.Deposit:
    DoDeposit(account);
    break;
// ...
```

In your program, don’t use Jake’s account, instead use the account object you created.

- Code required actions in each of the **Do…** methods.

  DoWithdraw and *DoDeposit* will be responsible for performing a deposit or withdraw on the passed in account. You will need to use `Console.ReadLine` and `Convert.ToDecimal` to ask how much they'd like to withdraw/deposit, and call the correct method on the Account object with the amount specified.

  ```
  Remember you can declare a `decimal` local variable to store the amount t

  Use the value returned by the `Withdraw` and `Deposit` method to let the
  ```

  In *DoPrint* call print on the account.

7. Run the program and perform a few deposits, withdraws and prints and create a screenshot of your program running.

Remember to backup your work and to keep a copy of the things you submit to OnTrack.

Task Discussion

For this task you need to discuss the use of control flow with your tutor. Here are some guides on what to prepare for:

- 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?