

# SIT771 Object Oriented Development

## Pass Task 5.3: Many Bank Accounts

---

### Overview

For this task, we're going to be adding many accounts to our Bank program - to do this, we'll introduce a new Bank class and make use of C# List.

### Submission Details

Submit the following files to OnTrack.

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

### Instructions

In this task, you're going to be created a new Bank class in a file named **Bank.cs**. This will be used to play the role of the Bank, which will have a number of accounts that it manages.

To start off, let's add a the Bank class.

1. Create a new C# file named `Bank.cs`, in it, add the Bank class which meets the following UML diagram.

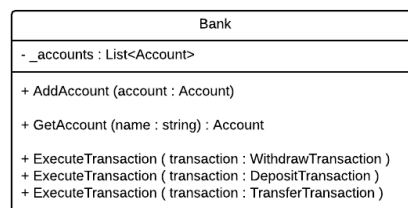


Figure: Bank uml

2. Implement the `AddAccount` , and `GetAccount` methods:

- `AddAccount`: Add the `account` which is passed in, into the Bank's list of accounts.
  - To do this, using the `Add` method on the `_accounts` object, and pass in the argument.
- `GetAccount`: Iterate through `_accounts`, and return the first bank which has the same name as the passed in `name` parameter.
  - Note: If the list of accounts doesn't have a matching account, you will need to return `null` . The `null` value is a special marker that indicates **no object**. So in this case, we are returning "no account object" as there are none with the required name.
  - Note: To get this working you will need to add a **readonly** `Name` property to the `Account` class. The get method for this will return the value from the `_name` field.
  - Users of `GetAccount` will then have to check if they got an account or no account ( `null` ) when they called `GetAccount` .

Here is the pseudocode for how we can find an account.

- Loop for each account
  - If the current account has the matching name...
    - Return the current account
- After the loop... return null as no account had a matching name

3. Notice in the UML diagram, `ExecuteTransaction` is listed three times but with different parameters. This is known as **Method Overloading**. When the `ExecuteTransaction` method is called, the correct method is identified by determining the type of the argument which is being passed in. As you can see, we accept the types `WithdrawTransaction` , `DepositTransaction` , and `TransferTransaction` .

The bodies of each `ExecuteTransaction` methods are the same, they call `Execute` on the `transaction` parameter.

4. Now that we've got this Bank class, we need to refactor `Main` to use it!

- In `Main` , add a new Bank object.
- Add new `MenuOption` values - `NewAccount` and the code so that the user can see, select, and run this option. It will:
  - Ask the user for the name of the account and its starting balance
  - Create a new Account object
  - Use the `AddAccount` method on the bank object to add the new account.

5. Next, we need a way of getting the account(s) for the other menu options. To do this we can add a `FindAccount` static method in the `Program`. This can be implemented as shown below.

```
public class Program
{
    //...
    private static Account FindAccount(Bank fromBank)
    {
        Console.WriteLine("Enter account name: ");
        String name = Console.ReadLine();
        Account result = fromBank.GetAccount(name);

        if ( result == null )
        {
            Console.WriteLine($"No account found with name {name}");
        }

        return result;
    }
}
```

6. Change `DoDeposit` to make use of the `Bank`. You need to change the parameters to be passed a `Bank` object, not an account object. `DoDeposit` can then use the `Bank` to find the account to deposit to. Here is a *start* for this code, with some comments for guidance:

```
private static void DoDeposit(Bank toBank)
{
    Account toAccount = FindAccount(toBank);
    if (toAccount == null) return;

    // Read in the amount
    // Create the Deposit transaction
    // Tell toBank to run the transaction
    // Ask the transaction to Print
}
```

7. Compile and run your program and make sure you can create an account, and deposit money into it.
8. Make similar changes to the `DoWithdraw` and `DoTransfer` methods. With Transfer, you will need to find both the from and to accounts, and you want to make sure that neither is null.
9. Create a screenshot of your program running! Make sure to demonstrate the different actions are working.

Remember to backup your work, and make sure you keep a copy of everything 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 the `Bank` works when you have multiple accounts.
- Demonstrate you can deposit and withdraw from many 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?