Lab: Defining Classes

Problems for exercises and homework for the "C# OOP Basics" course @ SoftUni".

You can check your solutions here: https://judge.softuni.bg/Contests/674/Defining-Classes-Lab

Problem 1. Bank Account

Create a **class** named **BankAccount**.

The class should have **private fields** for:

- **id: int**
- **balance: decimal**

The class should also have **public properties** for:

- **Id: int**
- **Balance: decimal**

You should be able to use the class like this:

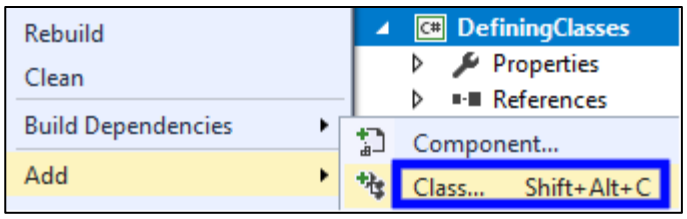
```
static void Main(string[] args)
{
    BankAccount acc = new BankAccount();

    acc.Id = 1;
    acc.Balance = 15;

    Console.WriteLine($"Account {acc.Id}, balance {acc.Balance}");
}
```

Solution

Create a **new class** and ensure **proper naming**



Problem 2. Bank Account Methods

Create a class **BankAccount** (you can use class from previous task)

The class should have private fields for:

- **id: int**
- **balance: decimal**

The class should also have properties for:

- **Id: int**
- **Balance: decimal**
- **Deposit(decimal amount): void**
- **Withdraw(decimal amount): void**

Override the method **ToString()**.

You should be able to use the class like this:

```
static void Main(string[] args)
{
    BankAccount acc = new BankAccount();

    acc.Id = 1;
    acc.Deposit(15);
    acc.Withdraw(10);

    Console.WriteLine(acc);
}
```

Solution

Create a method **Deposit(decimal amount)**

```
public void Deposit(decimal amount)
{
    this.Balance += amount;
}
```

Create a method **Withdraw(decimal amount)**

```
public void Withdraw(decimal amount)
{
    this.Balance -= amount;
}
```

Override the method **ToString()**

```
public override string ToString()
{
    return $"Account {this.Id}, balance {this.Balance}";
}
```

Problem 3. Test Client

Create a test client that tests your **BankAccount** class.

Support the **following commands**:

- **Create {Id}**
- **Deposit {Id} {amount}**
- **Withdraw {Id} {amount}**
- **Print {Id}**
- **End**

If you try to create an account with an existing Id, print **"Account already exists".**

If you try to perform an operation on a **non-existing account**, print **"Account does not exist"**.

If you try to withdraw an amount larger than the balance, print **"Insufficient balance"**.

The Print command should print **"Account ID{id}, balance {balance}"**. Round the balance to the second digit after the decimal separator.

Examples

Input	Output
Create 1 Create 1 Deposit 1 20 Withdraw 1 30 Withdraw 1 10 Print 1 End	Account already exists Insufficient balance Account ID1, balance 10.00
Deposit 2 20 Withdraw 2 30 Print 2 End	Account does not exist Account does not exist Account does not exist

Solution

Create a **`Dictionary<int, BankAccount>`** to store existing accounts

Create the input loop:

```
var cmdArgs = command.Split();

var cmdType = cmdArgs[0];
switch (cmdType)
{
    case "Create":
        Create(cmdArgs, accounts);
        break;
    case "Deposit":
        Deposit(cmdArgs, accounts);
        break;
    case "Withdraw":
        Withdraw(cmdArgs, accounts);
        break;
    case "Print":
        Print(cmdArgs, accounts);
        break;
}
```

Check the **type of command** and **execute** accordingly (***optional:*** *you can create a separate method for each command*)

Implement the **Create** command:

```
private static void Create(string[] cmdArgs, Dictionary<int, BankAccount> accounts)
{
    var id = int.Parse(cmdArgs[1]);
    if (accounts.ContainsKey(id))
    {
        Console.WriteLine("Account already exists");
    }
    else
    {
        var acc = new BankAccount();
        acc.ID = id;
        accounts.Add(id, acc);
    }
}
```

Implement the rest of the commands following the same logic.

Problem 4. Person Class

Create a **Person** class.

The class should have **private fields** for:

- **`name: string`**
- **`age: int`**
- **`accounts: List<BankAccount>`**

The class should have **constructors**:

- **`Person(string name, int age)`**
- **`Person(string name, int age, List<BankAccount> accounts)`**

The class should also have **public methods** for:

- **`GetBalance(): decimal`**

Solution

Create the class as usual:

```
public class Person
{
    private string name;
    private int age;
    private List<BankAccount> accounts;
}
```

Create a constructor with two parameters:

```
public Person(string name, int age)
{
    this.name = name;
    this.age = age;
    this.accounts = new List<BankAccount>();
}
```

Create a constructor with three parameters:

```
public Person(string name, int age, List<BankAccount> accounts)
{
    this.name = name;
    this.age = age;
    this.accounts = accounts;
}
```

Create method **`GetBalance()`**:

```
public decimal GetBalance()
{
    return this.accounts
}
```

Optional: You can take advantage of **constructor chaining**:

```
public Person(string name, int age)
    : this(name, age, new List<BankAccount>())
{ }

public Person(string name, int age, List<BankAccount> accounts)
{
    this.name = name;
    this.age = age;
    this.accounts = accounts;
}
```