Refactoring your code to SRP

Previously we had these questions:

- What is the role of the Main method?
- What is the role of the Program.cs file?
- What should the `Greeting()` method do? - how any things etc.?

Question 01:

The Main method is used to deliver the flow of the program, it does this by:

- reading user input
- print information to the screen
- call methods to process business logic

Note that is doesn't actually do any of the logic itself.

Question 02:

The Program.cs file is the main file of the program which holds the Main method - and only the main method.

Program is a class and this class should only do one thing.

If there is a need to put in an extra method - for example a console menu - then that can be placed in the Program.cs file, because it is still only delivering the UI, but this is optional.

Question 03:

The `Greeting()` method should only deliver the greeting. The method should take an argument rather than asking for information from an external source, which it can't guarantee the content of.

By setting a parameter of a certain type, the method can process it's logic without fail.

The `Greeting()` method should also not be in the Program class - it doesn't belong there as it does not deliver any UI. It delivers data.

So we can create a new class called `Messages` and place it in there.

Now we can call this method by doing the following: `Messages.Greeting()`

Now we have altered the Main method without breaking the program and your final code should look like this:

```
public class Program
{
    public static void Main(string[] args)
    {
        Console.WriteLine("What is your name?");
        string name = Console.ReadLine();
        Console.WriteLine(Messages.Greeting(name));
    }
}

public class Messages
{
    public static string Greeting(string name)
    {
      return $"Your name is {name}";
    }
}
```

SOLID Principle 2 - Open Closed Principle (cont...)

Ok - so we were talking about OCP, so what would we do with that?

So if we now want to add an extra greeting, say for example we want to say:

```
Good day, my name is ${name};
```

or

```
Good Afternoon, my name is ${name};
```

How would we add that to our program without guaranteeing that it won't break the code already have?

You may say: "We could add the methods to the Message class?"

You would be right and you would be sort of wrong.

or rather right with caution...

- Right - yes you can add extra methods to a class
- Sort of wrong - how do you know that this class won't break anything else in the program when you add these methods?

Up until this point we have just created classes when we needed classes, but we really want to base these classes on something - so that they do not just turn into some mixed bag we can't keep track off.

Introdroducing Interfaces

Interfaces allow us to stipulate what a class is made up off. An Interface specifies the properties and methods that a class should implement and when an Interface is linked to a class, the class must implement everything the Interface requires - it is a contract.

Now a class can still have its own methods, but that is something you should avoid.

If you need to add extra things to a class, create another interface first.

For example you may do something like this:

```
namespace ocp_trials
{
    public interface IMessages
    {
        string Greeting(string name);
    }

    public interface INoonMessages
    {
        string AfternoonGreeting(string name);
    }

    public class Messages: IMessages, INoonMessages
    {
        public string Greeting(string name)
        {
          return $"Your name is {name}";
        }

        public string AfternoonGreeting(string name)
        {
          return $"Good afternoon, your name is {name}";
        }
    }
}
```

You can see we have 2 interfaces and 2 classes, one class implements one interface and the other implements both. However the body of the methods that the classes need to implement is different - the names of the methods are the same.

Previously you may have done Inheritance and used polymorphism on a method - in this case like the inheritance model, you would define an abstract class.
The downsides of that approach is that you can only inherit from a single class and your subclasses are now also dependant on that parent class.

With an Interface, the class could still function as it is without it being tied to the interface - so you are not coupling the classes to each other.

As far as the implentstion goes in the Main method - you can specify that the type of the variable is the interface model and then if you want to change the return value of the method, you only need to change the class which is the interface is implementing - the rest stays the same.

So now we are able to extend the classes - but we are still modifying the class and in this case because the methods are similar that is ok.

However what if we have a more complex solution?

For that we need to know the different classes we use within C#.

1. A POCO class - This is a class like how you know it. Nothing fancy, but not dependant on anything.
2. A MODEL class - This is a class that has no methods in it, only properties and it is often used to match a datastream structure.

When you get some data in your app from an external source you create a MODEL class so you can display this data. (We will look at how to this later)

But the MODEL class is never implemented as an instance - it is a MODEL.

From this model, we can create different Interfaces and these interfaces may or may not have methods that they want to have imlemented. So now you may have some data off all the different employees of a company and those employees have different roles and attributes.

So you may create an interface for the following:

- Technician
- Sales person
- Accounts

All these people's details will come from the same base MODEL class, but they will be implemented as their own class based on the interfaces that are created for them.

For the people above you would create 3 interfaces (one for each type) so that they would be able to be implemented. This way you are not modifying any of the classes already created, but you are making multiple classes based from the original Model.

Now what if you want to create a new role - for example a managers role - you could now create an interface for the manager and you are then extending your program without modifying the code of the roles that are already implemented.

We are about to look at a code example, but before you we do that be aware that while we going to be coding the open closed principle, we are also applying the single responsibility princinple.

Let's have a look at an example:

```
// CODE
```

SOLID and GUI

So why are the SOLID principles in a GUI class?

The reason for this is so that we able to make our code portable. The idea is that we may not know what type of UI our classes are going to interact with. So we want to decouple from any UI as much as possible.

Now you will see later on that we are going to have code tightly coupled to our UI (which is what we don't want) but that is a process we are building towards with techniques we will learn later.

UI cannot be tested by automated tests. So by having our business logic code as much separated as possible we will be able to test our code and check that it works without having to manually press buttons and interact with the UI.

The SOLID principles are principles - they are not rules, although a lot of places in industry apply them to keep a tidy code base.