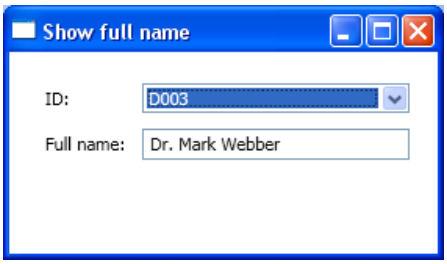
LAB 4: Interfaces and inheritance

Getting started

In this lab you will develop the *Doctor* and *Patient* classes you used in the Appointments system previous labs. The application which you will be working with is a simple window which includes a combo box which lists the ID of all people in the system, including doctors and patients. When the user selects a name from the combo box, the full name of the person is to be displayed in a text box. The names of doctors and patients will be displayed in different formats:



As in previous labs you will look **only at the code for the C# model classes**, and use the GUI simply to test the program. The GUI will be given to you – you will need to create the model classes. The model classes are similar to the *Doctor* and *Patient* classes in previous labs, but have some changes to their specification.

Task 1: Using an interface

Each item in the combo box can be an instance of either *Doctor* or *Patient*. Both classes must have an *ID* property which can be displayed in the combo box, and both must provide a method *GetFullName* which is used to get the text for the text box.

Another way of saying this is that to be displayed in the combo box the *Doctor* and *Patient* classes must **fulfil a contract** which says that they will provide:

- A property called *ID* of type string
- A method called *GetFullName* which returns a string

This will be ensured by making these classes implement an **interface** called *IPerson* which declares the appropriate property and method.

Open the project:

1. Start a VM which has Visual Studio 2010 installed, and start Visual Studio.

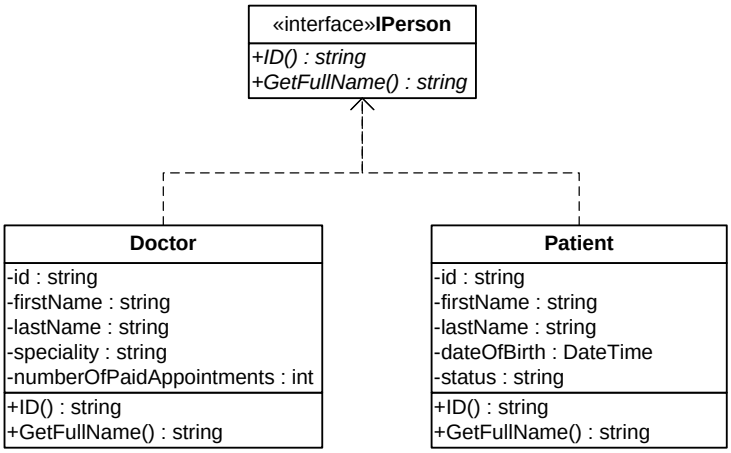
2. Download the file *lab4.zip* from Blackboard, and extract the contents.

3. In Visual Studio, select **File>Open Project**.. and browse to find the file *Appointments.sln* inside the *task1/Appointments* folder, then click **Open**.

4. Build the project. You will get errors, as the model classes are not included in this project.

Add model classes and interface:

5. Create an interface *IPerson* and classes *Doctor* and *Patient* which meet the following specification:



IPerson:

- *ID* is a property of type string
- *GetFullName* has a return type of string and takes no parameters

Doctor:

- *id, firstName, lastName* and *speciality* instance variables should be encapsulated in read/write properties
- *numberOfPaidAppointments* instance variable should be encapsulated in a read-only property
- There should be a constructor which takes string parameters which are used to set *id*, *firstName*, *lastName* and *speciality*. The value of *numberOfPaidAppointments* should be set to zero in the constructor.
- The *GetFullName* method should return a string containing *"Dr ".* and the values of *firstName* and *lastName*.
- This class should implement *IPerson*

Patient:

- All instance variables should be encapsulated in read/write properties
- There should be a constructor which takes five string parameters which are used to set all the instance variable values. The string representing date of birth will need to be converted within the constructor to type *DateTime*, using an appropriate method of the .NET framework ***Convert*** class.

- The *GetFullName* method should return a string containing the values of *lastName* and *firstName* separated by a comma.
- This class should implement *IPerson*

NOTE: The *ID* **property** is shown in the class diagram as a **method,** which is not strictly correct. Actually, UML does not have a specific way of showing properties, which are only found in .NET languages. Often we don't include properties in class diagrams as they are really just a way of accessing instance variables. However, to show properties in an interface we need to show them as methods, as interfaces can't include instance variables.

Test:

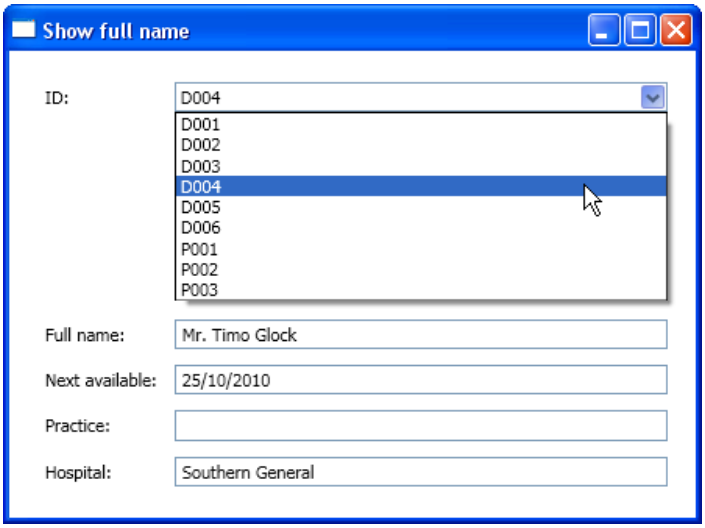
1. Run the program. You should see the **Show Full Name** window.
2. Select *D003* in the combo box. Check that the text box shows *Dr. Mark Webber*.
3. Select *P001* in the combo box. Check that the text box shows *Vettel, Sebastien*.

Questions:

- What do you think is the **reference type** for the objects displayed in the combo box?
- What three things must the code for a class include in order to implement the *IPerson* interface?

Task 2: Creating derived classes

In this task you will add the ability to deal with other types of person by adding a *Surgeon* class and a *GeneralPractitioner* class. These are both specific kinds of doctor, so the classes will be derived from the *Doctor* class. The Show Full Name window is extended to show more details of the selected person.



Open the project:

1. Start a VM which has Visual Studio 2010 installed, and start Visual Studio.

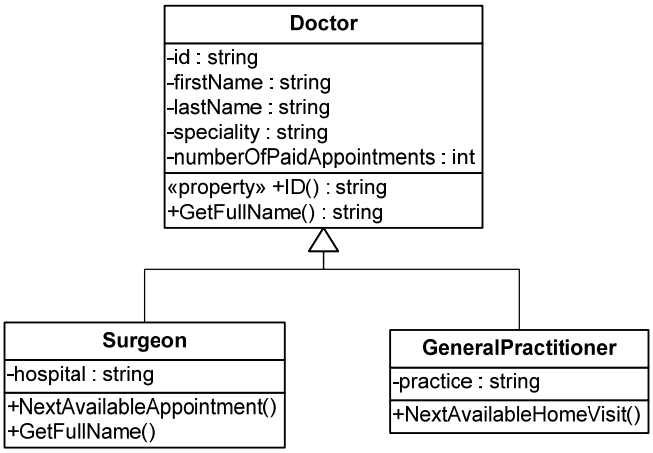
2. Download the file *lab4.zip* from Blackboard, and extract the contents.

3. In Visual Studio, select **File>Open Project**.. and browse to find the file *Appointments.sln* inside the *task2/Appointments* folder, then click **Open**.

4. Build the project. You will get errors, as not all of the model classes are included in this project.

Add new model classes:

5. Create classes *Surgeon* and *GeneralPractitioner* which meet the following specification. The *Doctor* and *Patient* classes and *IPerson* interface from task 1 are already in the project.



Surgeon:

- Should be derived from *Doctor*
- *hospital* instance variable should be encapsulated in a read/write property
- There should be a constructor which takes parameters which are used to set *id*, *firstName*, *lastName* ,*speciality* and *hospital*, using the base class constructor to set all but *hospital*.
- The *GetFullName* method should **override** the method with the same name in *Doctor*, and return a string containing *"Mr ".* and the values of *firstName* and *lastName*
- The *NextAvailableAppointment* method should return a *DateTime* object representing a date **one week** from now – you should look at the MSDN documentation for *DateTime* to find how to do this.

GeneralPractitioner:

- Should be derived from *Doctor*
- *practice* instance variable should be encapsulated in a read/write property
- There should be a constructor which takes parameters which are used to set *id*, *firstName*, *lastName* ,*speciality* and *practice*, using the base class constructor to set all but *practice*.
- The *NextAvailableHomeVisit* method should return a *DateTime* object representing a date **one day** from now

Doctor:

- The *GetFullName* method should be declared as *virtual* so that it can be overridden.

Test:

4. Run the program. You should see the **Show Full Name** window.

5. Select *D003* in the combo box. Check that the text box shows *Dr. Mark Webber*, that a practice name is shown and that the Next Available box shows tomorrow's date.

6. Select *D006* in the combo box. Check that the text box shows *Mr Robert Kubica*, that a hospital name is shown and that the Next Available box shows a date a week from now.

7. Select *P001* in the combo box. Check that the text box shows *Vettel, Sebastien* and that no other information is shown.

Questions:

- What do you think is the **reference type** for the objects displayed in the combo box?
- What are the types of the three objects you are displaying in steps 5-7 above?
- In step 5 above, the *GetFullName* method from which class is used to display *Dr. Mark Webber?*
- In step 6 above, the *GetFullName* method from which class is used to display *Mr Robert Kubica?*
- Does the *Surgeon* class implement *IPerson*? How do you know?
- Draw an object diagram for the object(s) used to display information in step 5 above.

Task 3: Implementing a framework interface

The .NET framework includes many interfaces which you can implement in your classes. Implementing an interface can allow your classes to interact with other framework classes or the .NET runtime in useful ways.

One example is the *IDisposable* interface, which declares a single method, *Dispose*. Here's a situation where this is useful:

- Your program opens a file on disk to write some data
- Opening a file uses an **operating system resource** (called a file handle)
- Something goes wrong while writing and your program crashes
- The file handle is left using up memory - file handles are not managed by .NET and so are not garbage collected

C# has a special code block, the **using** block, which helps deal with this situation. In this example, a *StreamWriter* object is created in a using block. If something goes wrong, the *Dispose* method of *StreamWriter* is called immediately and will destroy the file handle so that its memory is freed up. If nothing goes wrong, *Dispose* is called as soon as the using block ends. This works because *StreamWriter* implements *IDisposable*. Any object which implements *IDisposable* can be used like this.

```
using (StreamWriter sw = new StreamWriter("TestFile.txt"))
{
    // write to file
}
```

In this task you will create a class which implements *IDisposable*, and you will observe that its *Dispose* method is called at the end of a using block.

Create a project:

1. In Visual Studio, select **File>New Project**.. and create a new C# console application called *task3*.

2. Add a class called *MyClass* as follows:

```
class MyClass : IDisposable
{
    public MyClass()
    {
        Console.WriteLine(
         "CREATED A NEW OBJECT WHICH MIGHT USE SYSTEM RESOURCES...");
    }

    public void Dispose()
    {
        Console.WriteLine("CLEANED UP...");
    }
}
```

3. Add the following code to the *Main* method in the *Program* class:

```
using (MyClass myObj = new MyClass())
{
    Console.WriteLine("inside USING block");
}
Console.ReadLine();
```

Test:

4. Run the program. Observe the output in the console window. You should see three messages.

Questions:

What part of *MyClass* is executed at the start of the using block in *Main*?

What part of *MyClass* is executed at the end of the using block in *Main*?

How does the runtime know that it can call the *Dispose* method of *MyClass*?