## LAB 6: Working with collections

Getting started

In this lab you will first look, in **task 1**, at the way some common abstract data types store objects, using some classes which provide simple implementations of these types.

In **task 2** you will develop a new *HealthCentre* class which uses .NET Framework collection types to store a collection of *Patient* objects. This class will be used in a window which lists patients and allows the user to search for the details of a specific patient.
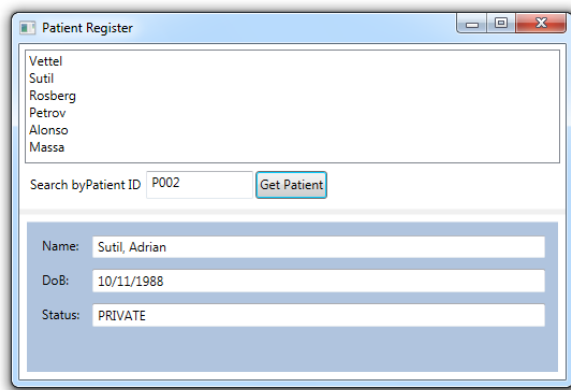
As in previous labs you will look **only at the code for the C# model classes**, and use the GUI simply to test the program. The GUI will be given to you – you will need to create the model classes.

Task 1: Examining abstract data types

This task uses the classes in a namespace called *SimpleCollections* which is not part of the .NET Framework but which is provided for you.

Open the project:

1. Start a VM which has Visual Studio 2012 installed, and start Visual Studio.

2. Download the file *lab6.zip* from GCULearn, and extract the contents.

3. In Visual Studio, open the *CollectionsLab* solution which is inside the *task1* folder. This solution contains two projects, *CollectionsLab* and *SimpleCollections*. The latter provides the abstract data type implementations.

4. Build the solution.

Examining an array list:

5. Open *Program.cs* in the *CollectionsLab* project. Note the following:

    *   there is a **using** reference to the *SimpleCollections* namespace.
    *   Some instances of *Patient* and *Doctor* classes are created – these classes implement the interface *IPerson*.
    *   An instance of *Object* is also created
    *   An instance, called *people*, of a class *SimpleArrayList* is created
    *   The *Patient* and *Doctor* classes objects are added to people, by calling the *Add* method.

6. Open *SimpleArrayList.cs* in the *SimpleCollections* project. Look at the instance variables.

**Question**: ***What do you think is the purpose of each of the three instance variables of SimpleArrayList?***

7. Click in the margin beside the last line of code to set a breakpoint. When you run the program, it will stop at that line.
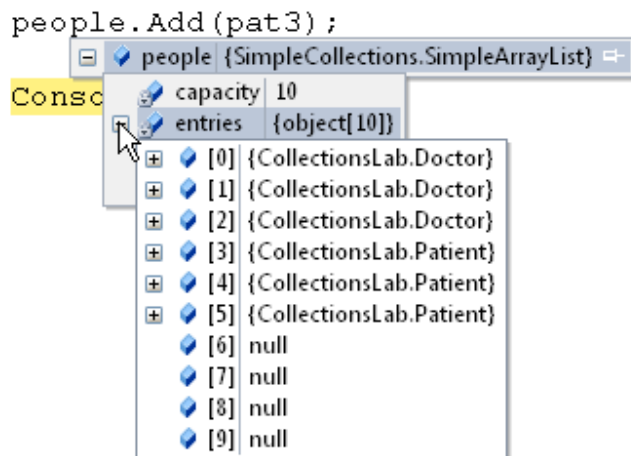
```
27      people.Add(pat3);
28
29      Console.ReadLine();
30
```
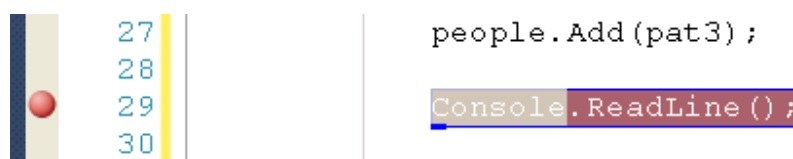
8. Run the program using the Start Debugging option. The program will stop at the breakpoint.

```
27      people.Add(pat3);
28
29      Console.ReadLine();
30
```
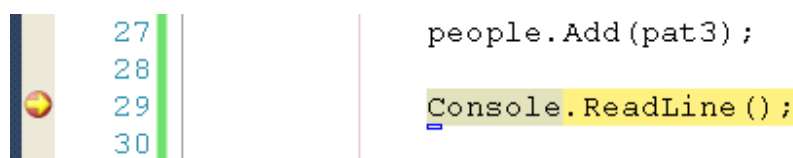
9. Place the cursor over the name of a reference to people. An **inspector** should pop-up:

```
people.Add(pat3);
        ⊞ ● people {SimpleCollections.SimpleArrayList} ▭
Console.ReadLine();
```
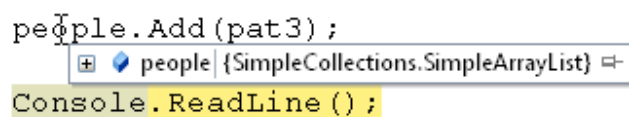
This will allow you to explore the structure of this *SimpleArrayList* object. Click the + sign to expand the view. You should see the instance variables. Expand *entries*. You should now see a representation of an array of objects. Expand each object to see its instance variables.

```
people.Add(pat3);
        ☐  ● people {SimpleCollections.SimpleArrayList} ᆏ
Consc        🖉 capacity  10
        ☐  🖉 entries    {object[10]}
       ⤷        ⊞ ● [0] {CollectionsLab.Doctor}
                 ⊞ ● [1] {CollectionsLab.Doctor}
                 ⊞ ● [2] {CollectionsLab.Doctor}
                 ⊞ ● [3] {CollectionsLab.Patient}
                 ⊞ ● [4] {CollectionsLab.Patient}
                 ⊞ ● [5] {CollectionsLab.Patient}
                   ● [6] null
                   ● [7] null
                   ● [8] null
                   ● [9] null
```

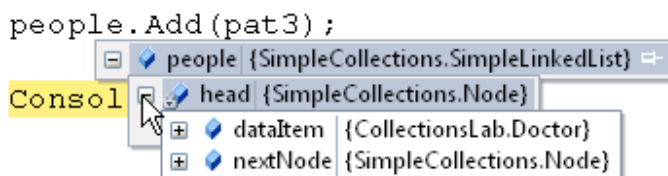**Question**: *How does a SimpleArrayList store objects?*

Question: *What kind of objects can it store?*

Question: *How many objects can it store?*

Question: *How many objects does it currently contain?*

Examining a linked list:

1. Open *Program.cs* in the *CollectionsLab* project.

2. Change the type of people to *SimpleLinkedList*. Do not change anything else. Set the same breakpoint as in the previous task if it is not already set.

3. Run the program and inspect *people* as before. The structure should be different from before.

```
people.Add(pat3);
        ☐  ● people {SimpleCollections.SimpleLinkedList} ᆏ
Consol  ☐  🖉 head {SimpleCollections.Node}
       ⤷        ⊞ ● dataItem  {CollectionsLab.Doctor}
                 ⊞ ● nextNode {SimpleCollections.Node}
```

4. Expand *people*, and then expand *head*. Expand *dataItem* – you should see the instance variables of a *Doctor* object.

5. Expand *nextNode.* You should see another *dataItem* and *nextNode.* Keep expanding each *nextNode* until you reach the end of the list.

Question: *How does a SimpleLinkedList store objects?*

**Question**: *How many objects can it store?*

Question: *How many objects does it currently contain?*

Examining a dictionary:

1. Open *Program.cs* in the *CollectionsLab* project.

2. Change the type of people to *SimpleDictionary*.

3. Change each call to *people.Add* to *people.Put*, and supply two parameters: the ID property of the object, and the object itself:

```
people.Put(doc1.ID, doc1);
people.Put(doc2.ID, doc2);
...
```

4. Run the program and inspect *people* as before. The structure should be different again.

```
people.Put(pat3.ID, pat3);
```
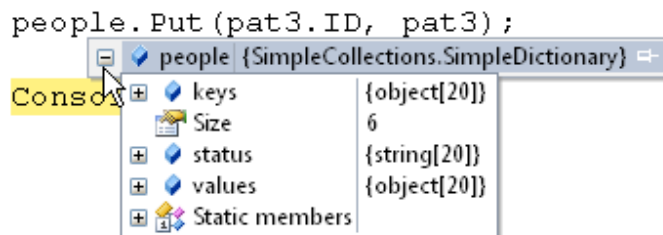

Expand keys and values and look at the contents.

Question: *How does a Dictionary store objects?*

Question: *What does 'keys' contain?*

Question: *What do you notice about the order in which keys and objects are stored?*

Task 2: Using .NET Framework collections

In this task you will create a *HealthCentre* class which contains a collection of all registered patients. The *HealthCentre* class will also contain a method to allow a specific *Patient* object to be found by ID.

Open the project:

1. Start a VM which has Visual Studio 2012 installed, and start Visual Studio.

2. Download the file *lab6.zip* from GCULearn, and extract the contents.

3. In Visual Studio, open the *HealthCentre* solution which is inside the *task2* folder.

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

Add a new model class:

The project should already contain a *Patient* class. You will add a new *HealthCentre* class. These classes are represented in this class diagram. This diagram was generated from code by Visual Studio, and unlike standard UML diagrams it shows .NET specific features, including properties.
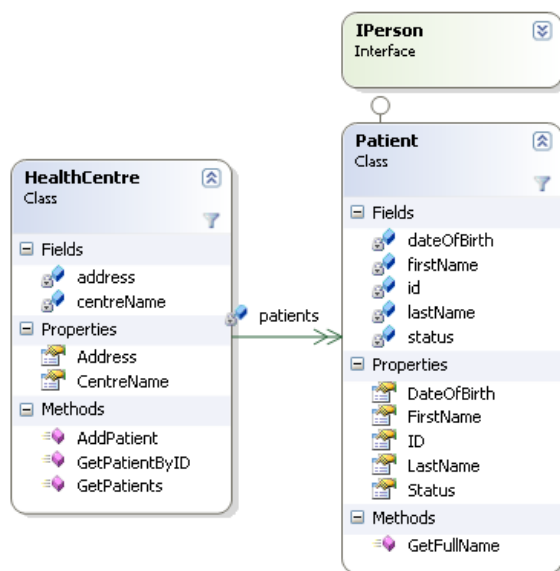
5. Create a new class *HealthCentre* which meets the following specification.

    HealthCentre:
    - Instance variables *centreName* and *address*, encapsulated in read/write properties
    - Instance variable *patients* of type *List<Patient>* which defines a one-to-many relationship with *Patient.*
    - Constructor which takes parameters which are used to set *centreName* and *address* and initialises *patients* as an empty list.
    - *AddPatient* method which takes a *Patient* object as a parameter and adds it to *patients* by calling the *Add* method of the *List* class.
    - *GetPatientByID* method which takes a *string* object named *ID* as a parameter and returns the *Patient* object with that *ID* value by calling the *Find* method of the *List* class with a suitable **predicate** defined using a **lambda expression.**
    - *GetPatients* method which simply returns the *List* object.

You may want to look at the MSDN documentation for *System.Collections.Generic.List* class.

Test:

1. Run the program. You should see the **Patient Register** window.

2. Check that the text box shows the last names of 6 patients.

3. Select *P002* in the text box and click the button. Check that the form shows the details of patient Adrian Sutil.

4. Select *P005* in the text box and click the button. Check that the form shows the details of patient Fernando Alonso.

5. Select *P007* in the text box and click the button. Check that the form boxes are blank as there is no patient with this ID.

Questions:
- ***How does the List class store the objects it holds?***
- ***What is the advantage of using a generic collection such as List<Patient>?***
- ***What types of object can be stored in a list of type List<Patient>?***

Use Dictionary instead of List:

You will now change your *HealthCentre* class so that it uses a Dictionary to store patients.

1. Modify *HealthCentre* as follows:

    HealthCentre:
    - Change the type of *patients* to *Dictionary<string, Patient>*
    - Modify the constructor so that it initialises patients as an empty dictionary.
    - Modify *AddPatient* so that it uses the *Add* method of the *Dictionary* class, using the *ID* property of the *Patient* parameter as the key.
    - Modify *GetPatientByID* so that it uses the *ID* parameter as the key to find the required *Patient*.
    - Modify *GetPatients* so that it returns the **values** of the *Dictionary*. The return type of the method should be changed to *ICollection<Patient>*

You will need to look at the MSDN documentation for *System.Collections.Generic. Dictionary* class to see how to use the *Add* method, find an object using a key and get the values as a collection.

Test:

The program should work exactly as before – repeat the previous tests.

Questions:
- ***How does the Dictionary class store the objects it holds?***
- ***Which version of HealthCentre will be more efficient when you need to find a patient if there is a very large number of patients? Why?***

Programming challenge

This challenge involves the *IEnumerable* interface. Implementing *IEnumerable* allows a collection class to be traversed using the **foreach** loop.

Open the *CollectionsLab solution* you worked with in Task 1. In the *SimpleCollections* project there is a class *SimpleEnumerableArrayList*. This class contains a class *SimpleArrayEnumerator*, which can traverse the array list (and which is used by the foreach loop). Traversing an array list simply means moving through the array one item at a time.

SimpleArrayEnumerator has an instance variable index which keeps track of the current position in the list (initially -1 – i.e. just before the first list entry) and can access the variable *entries* which is the array used by the list to store objects.

Your challenge is to work out how to complete the following:

- **MoveNext** – method which increments the current position, returns false if this is beyond the last entry in the array, returns true otherwise

- **Current** – property which returns the object stored at the current position in the array

Find these and complete the code. You can test by changing the type of *people* in *Program* to *SimpleEnumerableArrayList* and adding the following code:

```
foreach (IPerson p in people)
{
    Console.WriteLine(p.GetFullName());
}
```

You should see a list of names displayed in the console. If you can build and run the solution, but see no output in the console, then your code is not working correctly.