Arrays, Structures, Classes, Objects, Methods, Functions and Reference Types

This week the concepts of data structures have been introduced. Just to recap.

An array holds multiple values of a single data type and its size should (usually) be declared when the array is created. A structure is a data structure which can hold more than one type of data. It helps you to make a single variable hold related data of various data types. The *struct* keyword is used for creating a structure. A structure can also contain *constructors*, *constants*, *fields*, *methods*, *properties*, *indexers*, *operators*, *events*, and *nested types*.

A class is a blueprint for a data type and enables you to create your own custom types by grouping together variables of other types, methods and events. Objects are instances of a class and can be stored in either a named variable or in an array or collection. In OO programming method is a subroutine (or procedure or function) associated with a class. A function is a piece of code which is called by name and data can be passed into it to operate on. It can also optionally return data. Both functions and methods allows you to encapsulate a piece of code to call it from other parts of your code. You may run into a situation where you need to repeat a piece of code, from multiple places, and this is where they come in.

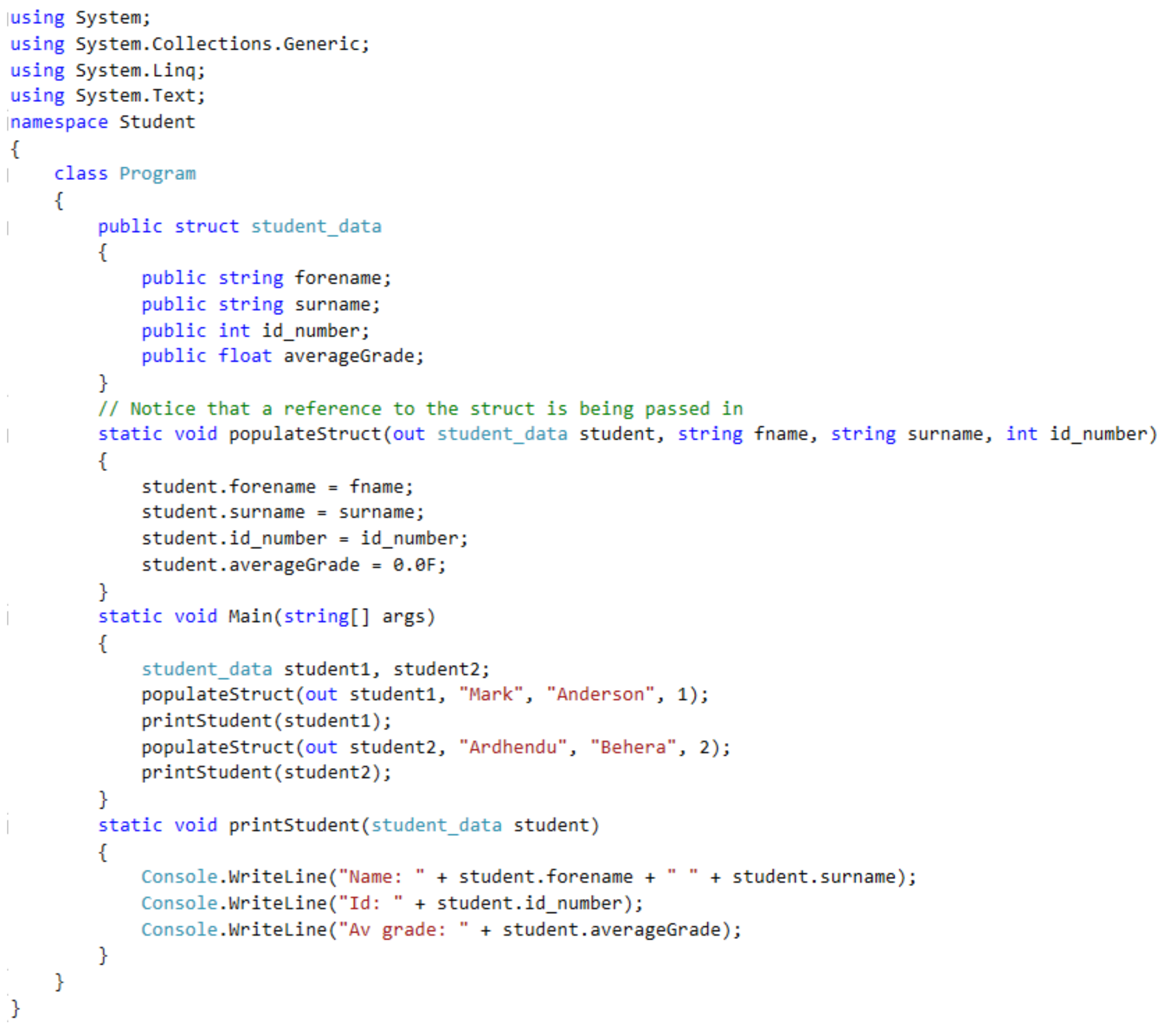
In C#, there are two kind of types: reference type and value type. For example, numeric data types such as integers, floats, etc., boolean, enumerations and user defined structures are value types whereas class, interfaces and delegates are reference type. Objects, strings and dynamics are built-in reference types. Variables of reference types store references to their data, while variables of value types directly contain their data.

The sample code below creates a structure to hold student data. Each student has a forename, a surname, a student id and an average grade. The method `populateStruct` takes four parameters to populate a student data structure. Note that the first parameter is a reference (using `out` in this case). This means that we can change the values in the method and those changes will be reflected in the structure outside of the `populateStruct` method. This is why we can print out the contents of the structure in the `main()` method and still see the updated values. This is called pass-by-reference. If the method has to be written as pass-by-value (removing the `out` in the function signature) then the changes will be lost outside the method.

```
static void populateStruct(out student_data student,
```

Try removing the `out` in the above line, recompiling the code and re-running the program. You will get output suggesting that the structures aren't holding any data.

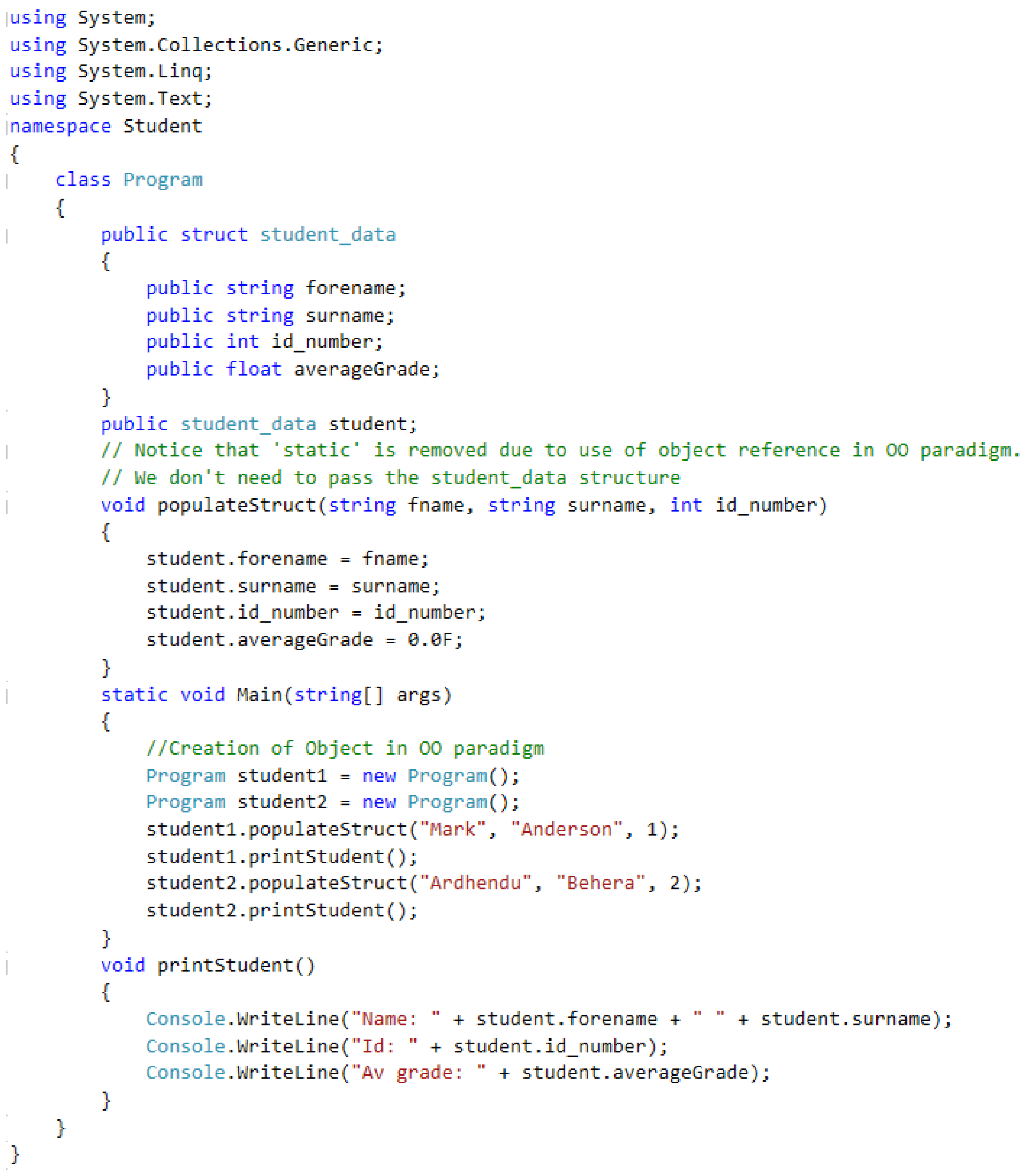
The `printStudent` method takes one parameter (a value of `student_data`) and return `void`. It prints the name, id and average grade on the console.



```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace Student
{
    class Program
    {
        public struct student_data
        {
            public string forename;
            public string surname;
            public int id_number;
            public float averageGrade;
        }
        // Notice that a reference to the struct is being passed in
        static void populateStruct(out student_data student, string fname, string surname, int id_number)
        {
            student.forename = fname;
            student.surname = surname;
            student.id_number = id_number;
            student.averageGrade = 0.0F;
        }
        static void Main(string[] args)
        {
            student_data student1, student2;
            populateStruct(out student1, "Mark", "Anderson", 1);
            printStudent(student1);
            populateStruct(out student2, "Ardhendu", "Behera", 2);
            printStudent(student2);
        }
        static void printStudent(student_data student)
        {
            Console.WriteLine("Name: " + student.forename + " " + student.surname);
            Console.WriteLine("Id: " + student.id_number);
            Console.WriteLine("Av grade: " + student.averageGrade);
        }
    }
}
```

Example 1: Structure definition, pass-by-value and pass-by-reference of structure data to methods.

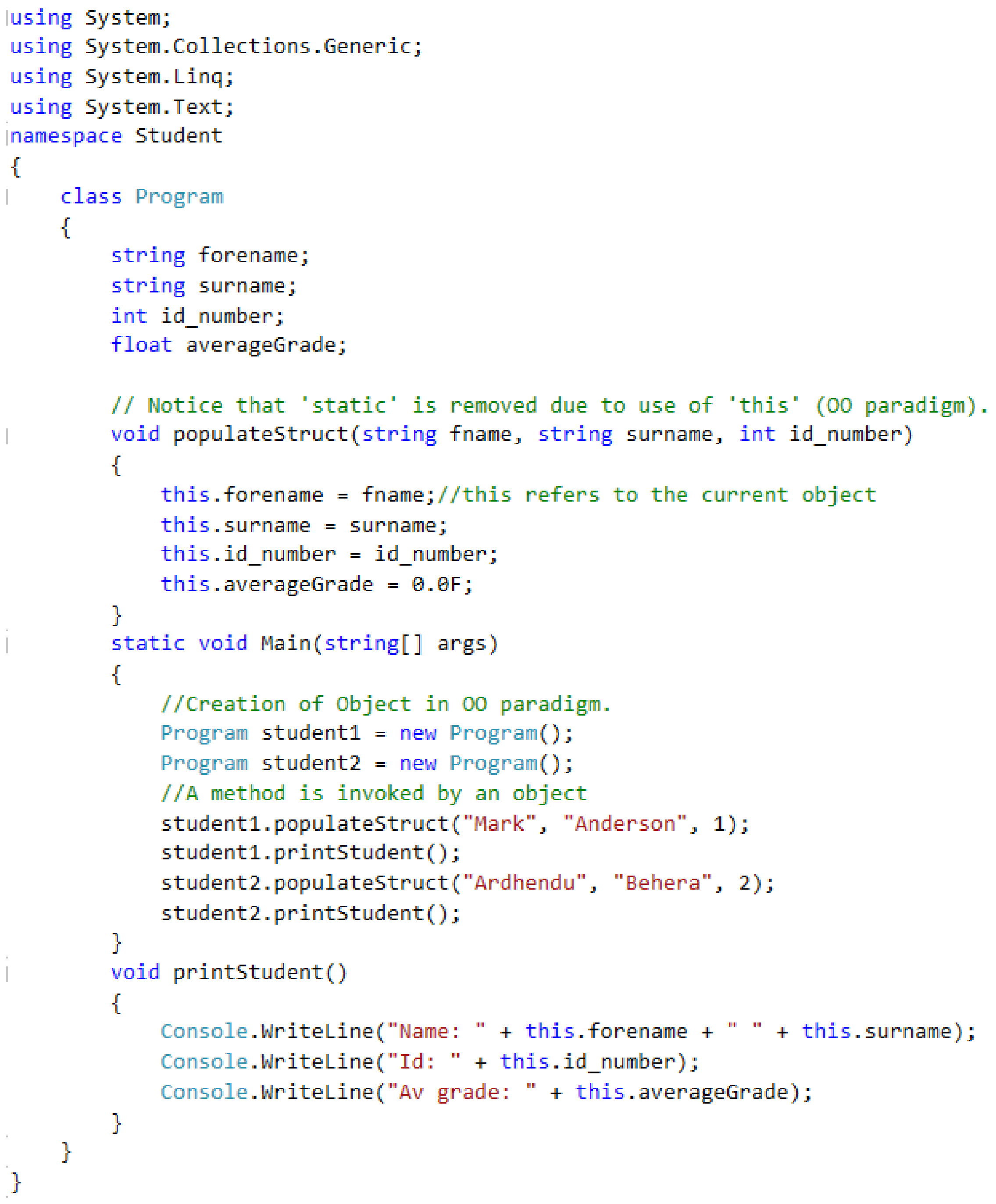
The above program code uses the `class Program`, however there is no object is created and used for the `class Program`. Therefore, it is not really Object-Oriented style of programming. Let's re-write the same program by using object and is given below. In this code, we create a variable named `student` to hold the structure `student_data`. This `student` variable is used in method `populateStruct` and `printStudent` for assigning and printing, respectively. Look closely how the same code can be written in different programming style.



```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace Student
{
    class Program
    {
        public struct student_data
        {
            public string forename;
            public string surname;
            public int id_number;
            public float averageGrade;
        }
        public student_data student;
        // Notice that 'static' is removed due to use of object reference in OO paradigm.
        // We don't need to pass the student_data structure
        void populateStruct(string fname, string surname, int id_number)
        {
            student.forename = fname;
            student.surname = surname;
            student.id_number = id_number;
            student.averageGrade = 0.0F;
        }
        static void Main(string[] args)
        {
            //Creation of Object in OO paradigm
            Program student1 = new Program();
            Program student2 = new Program();
            student1.populateStruct("Mark", "Anderson", 1);
            student1.printStudent();
            student2.populateStruct("Ardhendu", "Behera", 2);
            student2.printStudent();
        }
        void printStudent()
        {
            Console.WriteLine("Name: " + student.forename + " " + student.surname);
            Console.WriteLine("Id: " + student.id_number);
            Console.WriteLine("Av grade: " + student.averageGrade);
        }
    }
}
```

Example 2: Object-Oriented style of programming for the same code in example 1.

Now, let’s re-write again the same code in Object-Oriented style without using the `structure` and is shown below. The keyword `this` here refers to the current object and you might have encountered in java programming.



```csharp
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace Student
{
    class Program
    {
        string forename;
        string surname;
        int id_number;
        float averageGrade;

        // Notice that 'static' is removed due to use of 'this' (OO paradigm).
        void populateStruct(string fname, string surname, int id_number)
        {
            this.forename = fname;//this refers to the current object
            this.surname = surname;
            this.id_number = id_number;
            this.averageGrade = 0.0F;
        }
        static void Main(string[] args)
        {
            //Creation of Object in OO paradigm.
            Program student1 = new Program();
            Program student2 = new Program();
            //A method is invoked by an object
            student1.populateStruct("Mark", "Anderson", 1);
            student1.printStudent();
            student2.populateStruct("Ardhendu", "Behera", 2);
            student2.printStudent();
        }
        void printStudent()
        {
            Console.WriteLine("Name: " + this.forename + " " + this.surname);
            Console.WriteLine("Id: " + this.id_number);
            Console.WriteLine("Av grade: " + this.averageGrade);
        }
    }
}
```

Example 3: Object-Oriented style of programming for the same code in example 1 using keyword `this` and without using a `structure`.

This shows the same problem can be implemented using different programming style. In this portfolio, we will use the style described in Example 1.

Exercises

EX1:

Using the code in Example 1 as a template, amend the `student_data` structure so that a student also has a programme title and a programme code. Both of these fields should be strings. Amend `populateStruct` and `printStudent` to incorporate these new field (so you can create a student and supply the programme name/title, and these new features are also included in the print out for a student.

EX2:

Amend the `student_data` program above to hold an array of four students (rather than individual student) in the `main` method. In your `main` method, you can populate the array using four calls to the `populateStruct` method, e.g.

```
populateStruct(students[0], "Mark", "Anderson", 1);
populateStruct(students[1], "Ardhendu", "Behera", 2);
populateStruct(students[2], "Tom", "Jones", 3);
populateStruct(students[3], "Ewan", "Evans", 4);
```

Write a new `printAllStudent` method which uses a loop to print out all the elements in the array. As a couple of hints, remember that the variable which is just the name of the array (i.e. students in the above snippet of code) is a pointer to the first element. Also, the signature for your `printAllStudent` method should be

```
static void printAllStudent(ref student_data students){

//Add your code here
}
```

EX3

Extend the previous program by reading number of students from the console and then dynamically create a `student_data` array of size equal to the number entered (e.g. 5).

```
student_data[] students = new student_data[5];
```

For each student, enter their name, surname and id via console (you have done this in portfolio 1). In the end print all students' data. For example, one would see the bellow message in the terminal when you run your code.

```
Enter the number of students:
5
Person 1, please enter your name:
Mark
Person 1, please enter your surname:
Anderson
.
.
.
```

EX4:
Amend the `student_data` program above (Example 1) to add a new structure which will hold module data. In your `module_data` structure, you should include a module code, a module title and a module mark. Assume a student only takes six modules. Extend your `student_data` structure to hold six `module_data` items.

Referring to `populateStruct`, create a new method to populate the modules.

```
static void populateModule(out module_data module, string mcode, string mname, int score)
{
        // Your code goes here
}
```

You can now extend the `main` method to create six modules (in a similar way to how the students are created) and add those modules to your students. You should end up with an array that contains `student_data` structure, each of which have an array that contains `module_data` structure!

EX5 (Optional):
Amend the `student_data` program so that the average score is calculated from the grades held in each `module_data` structure for each `student_data` record.

EX6 (Optional):
Amend the previous `student_data` program (EX5) so that the average score is assigned a grade as a string (**Fail**: 0-29, **Narrow Fail**: 30-39, **Pass**: 40-49, **Good**: 50-59, **Very Good**: 60-69, **Excellent**: 70-84 and **Outstanding**: 85-100).