# Tutorial Learning materials and some C#7 new features

- Learning material for C#
- String interpolation
- Generics
- The 'is' expression
- The 'case' clause in switch statements (new pattern matching feature)

## 1. LinkedIn Learning

There is a lot of material available that you should familiarise yourselves with so you can progress with learning C# as quickly as possible.

Go to this website and log in to LinkedIn Learning
https://www.westminster.ac.uk/current-students/studies/study-skills-and-training/digital-skills/linkedin-learning

These tutorials will be helpful with Visual Studio and C#. They are generally between 1 and 5 hours, and I recommend you work though as many as you feel you need. The following should get you started.

C# Essential Training

Learning C#

Computer Science Principles Lab: C#

Finally, you can explore more new features in C#7 with this tutorial.

C# 7 First Look

## 2. String interpolation, and the 'is', 'case' clauses in 'switch' statements.

**(Chapter1 'C# 7 and .NET Core Cookbook', online via Library Search)**

Create a new project call it Tutorial2

Make sure you include the following using statement in your project.

```
using System.Collections.Generic;
```

**a) Create two new classes, one for Student and another for Professor.**

```
public class Student
{
  public string Name { get; set; }
```

```
    public string LastName { get; set; }
    public List<int> CourseCodes { get; set; }
  }


public class Professor
  {
    public string Name { get; set; }
    public string LastName { get; set; }
    public List<string> TeachesSubjects { get; set; }
  }
```

**Did you notice the shortcut to creating getters and setters. In your 'Main' program create instances of Student and Professor to familiarise yourself with them (see step 4 below for the answer).**

**b) Follow these instructions and also check out the book chapter 1.**
**(This may be difficult during tutorial times due to simultaneous viewing restrictions)**

1. In the `Tutorial2` class, create a new method called `OutputInformation()` that takes a person object as parameter.

```
public void OutputInformation(object person)
{

}
```

2. Inside this method, we would need to check what type of object is passed to it. Traditionally, we would need to do the following:

```
if (person is Student)
{
  Student student = (Student)person;
  WriteLine($"Student {student.Name} {student.LastName}
            is enrolled for courses {String.Join<int>(
            ", ", student.CourseCodes)}");
}
```

**//!!Notice the $ sign in the WriteLine. We call this string**
**// interpolation.**

```
if (person is Professor)
{
  Professor prof = (Professor)person;
  WriteLine($"Professor {prof.Name} {prof.LastName}
            teaches
{String.Join<string>(",",prof.TeachesSubjects)}");
}
```

3. We have two `if` statements. We are expecting either a `Student` object or a `Professor`object. The complete `OutputInformation()` method should look as follows:

```
public void OutputInformation(object person)
{
  if (person is Student)
  {
    Student student = (Student)person;
    WriteLine($"Student {student.Name} {student.LastName}
              is enrolled for courses {String.Join<int>
              (", ", student.CourseCodes)}");
  }
  if (person is Professor)
  {
    Professor prof = (Professor)person;
    WriteLine($"Professor {prof.Name} {prof.LastName}
              teaches {String.Join<string>
              (",", prof.TeachesSubjects)}");
    }
  }
```

4. Calling this method from the `static void Main` is easy enough. The objects are similar, but differ in the list they contain. A `Student` object exposes a list of course codes, while a `Professor` exposes a list of subjects taught to students.

```
static void Main(string[] args)
{
  Tutorial2 t2 = new Tutorial2();

  Student student = new Student();
  student.Name = "Dirk";
  student.LastName = "Strauss";
  student.CourseCodes = new List<int> { 203, 202, 101 };

  t2.OutputInformation(student);

  Professor prof = new Professor();
  prof.Name = "Reinhardt";
  prof.LastName = "Botha";
  prof.TeachesSubjects = new List<string> {
       "Mobile Development", "Cryptography" };

  t2.OutputInformation(prof);
}
```

5. Run the console application and see the `OutputInformation()` method in action.

6. While the information we see in the console application is what we expect, we can simplify the code in the `OutputInformation()` method much more with pattern matching. To do this, modify the code as follows:

```
if (person is Student student)
{

}
if (person is Professor prof)
{

}
```

7. The first `if` expression checks to see if the object `person` is of type `Student`. If so, it stores that value in the `student` variable. The same logic is true for the second `if` expression. If true, the value of `person` is stored inside the variable `prof`. For code execution to reach the code between the curly braces of each `if` expression, the condition had to evaluate to true. We can, therefore, dispense with the cast of the `person` object to a `Student` or `Professor`type, and just use the `student` or `prof` variable directly, like so:

```
if (person is Student student)
{
  WriteLine($"Student {student.Name} {student.LastName}
            is enrolled for courses {String.Join<int>
            (", ", student.CourseCodes)}");
}
if (person is Professor prof)
{
  WriteLine($"Professor {prof.Name} {prof.LastName}
            teaches {String.Join<string>
            (",", prof.TeachesSubjects)}");
}
```

8. Running the console application again, you will see that the output is exactly the same as before. We have, however, written better code that uses type pattern matching to determine the correct output to display.

9. Patterns, however, don't stop there. You can also use them in constant patterns, which are the simplest type of pattern to use. Let's take a look at the check for the constant `null`. With pattern matching we can enhance our `OutputInformation()` method as follows:

```
public void OutputInformation(object person)
{
  if (person is null)
  {
    WriteLine($"Object {nameof(person)} is null");
```

```
  }
}
```

10. Change the code that is calling the `OutputInformation()` method and set it to `null`.

```
Student student = null;
```

11. Run your console application and see the message displayed.

It is good practice to use the `nameof` keyword here. If the variable name `person` ever has to change, the corresponding output will be changed also.

12. Lastly, `switch` statements in C# 7.0 have been improved to make use of pattern matching. **C# 7.0 allows us to switch on anything, not just primitive types and strings.** The `case` clauses now make use of patterns, which is really exciting. Let's have a look at how to implement this in the following code examples. We will keep using the `Student` and `Professor` types to illustrate the concept of pattern matching in `switch` statements. Modify the `OutputInformation()` method and include the boilerplate `switch` statement as follows. The `switch` statement still has defaults, but it can now do so much more.

```
public void OutputInformation(object person)
{
  switch (person)
  {
    default:
      WriteLine("Unknown object detected");
    break;
  }
}
```

13. We can expand the `case` statement to check for the `Professor` type. If it matches an object to the `Professor` type, it can act on that object and use it as a `Professor` type in the body of the `case` statement. This means we can call the `Professor`-specific `TeachesSubjects`property. We do it like this:

```
switch (person)
{
  case Professor prof:
    WriteLine($"Professor {prof.Name} {prof.LastName}
              teaches {String.Join<string>
              (",", prof.TeachesSubjects)}");
  break;
  default:
    WriteLine("Unknown object detected");
  break;
}
```

14. We can also do the same for `Student` types. Change the code of the `switch` as follows:

```
switch (person)
{
  case Student student:
    WriteLine($"Student {student.Name} {student.LastName}
              is enrolled for courses {String.Join<int>
              (", ", student.CourseCodes)}");
  break;
  case Professor prof:
    WriteLine($"Professor {prof.Name} {prof.LastName}
              teaches {String.Join<string>
              (",", prof.TeachesSubjects)}");
  break;
  default:
    WriteLine("Unknown object detected");
  break;
}
```

15. One final (and great) feature of `case` statements remains to be illustrated. We can also implement a `when` condition. The `when`condition simply evaluates to a Boolean and further filters the input that it triggers on. To see this in action, change the `switch` accordingly:

```
switch (person)
{
  case Student student when
(student.CourseCodes.Contains(203)):
  WriteLine($"Student {student.Name} {student.LastName}
            is enrolled for course 203.");
  break;
  case Student student:
  WriteLine($"Student {student.Name} {student.LastName}
            is enrolled for courses {String.Join<int>
            (", ", student.CourseCodes)}");
  break;
  case Professor prof:
  WriteLine($"Professor {prof.Name} {prof.LastName}
            teaches {String.Join<string>(",",
            prof.TeachesSubjects)}");
  break;
  default:
    WriteLine("Unknown object detected");
  break;
}
```

16. Lastly, to come full circle and check for null values, we can modify our `switch` statement to cater for those too. The completed `switch` statement is, therefore, as follows:

```
switch (person)
{
  case Student student when
```

```
(student.CourseCodes.Contains(203)):
            WriteLine($"Student {student.Name} {student.LastName}
                      is enrolled for course 203.");
          break;
          case Student student:
          WriteLine($"Student {student.Name} {student.LastName}
                    is enrolled for courses {String.Join<int>
                    (", ", student.CourseCodes)}");
          break;
          case Professor prof:
          WriteLine($"Professor {prof.Name} {prof.LastName}
                    teaches {String.Join<string>
                    (",", prof.TeachesSubjects)}");
          break;
          case null:
            WriteLine($"Object {nameof(person)} is null");
          break;
          default:
            WriteLine("Unknown object detected");
          break;
        }
```

17. Running the console application again, you will see that the first case statement containing the `when` condition is triggered for the `Student` type.

End document