Lab 7: Extension Methods

Programming in C# for Visual Studio .NET Platform II

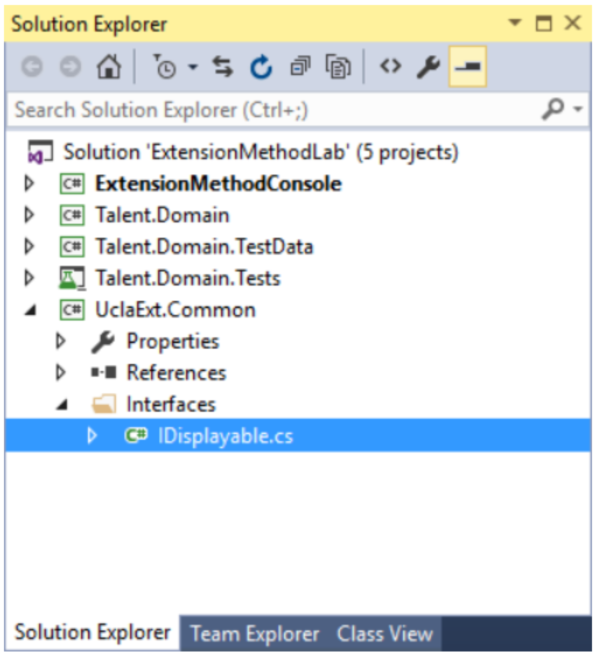
Objective

This exercise explores a feature called extension methods that was added in the 3.0 version of .Net. It is useful in its own right, but also provides an essential pre-requisite to the implementation of LINQ, which we will explore in subsequent lessons.

Solution Setup

Step 1: Download the ExtensionMethods.zip file and unzip it to your C:\UclaExt directory. Open the LinqLab.sln solution in Visual Studio and confirm that you see an `ExtensionMethodConsole` console Application and the `Talent.Domain`, `Talent.Domain.TestData and Talent.Domain.Tests` projects from the prior labs. You should be able to run all the unit tests successfully.

Step 2: Create a UclaExt.Common class library project. Some of the C# types (interfaces, classes, etc. that we are going to be building in this and future exercises are very general. We might want to use these types for applications other than the Talent application. To allow us to share these types across different projects, we should create a new class library, which I will name UclaExt.Common. Then if I want to re-use these features in future projects, I can just include the UclaExt.Common project in my new solution. Create a new class library called UclaExt.Common and add a project reference to the Talent.Domain project that references the UclaExt.Common project. Then add a new folder to the Common project called Interfaces and move the IDisplayable interface definition from the Talent.Domain project to the Common project's Interfaces folder.



The IDisplayable interface implementation should look like this (be sure to double-check that the namespace is UclaExt.Common.Interfaces and change it, if necessary):

```
namespace Talent.Domain
{
    public interface IDisplayable
    {
        string Display();
    }
}
```

Then you can remove the IDisplayable interface implementation from the Talent.Domain project – you will need to add using directives to UclaExt.Common.Interface to the Domain class definitions that implement the IDisplayable interface to get the Talent.Domain project to compile and run.

Step 4: Extension Methods. Suppose you wish to extend the functionality of a class or interface with a method. If the type you wish to extend is a class and is part of your application, you would probably choose to either modify the class by adding a new method, or perhaps derive a subclass from the original class, so the derived class can provide the additional method. However, there are many cases where this is not possible:

- You do not have access to the source code and the class is sealed. This would be the case if you wanted to extend many of the .Net framework classes.
- The class is used by many different assemblies, and the functionality you need is specific to your usage, so you don't want to pollute the class for other usages with irrelevant methods.
- It is an interface that you wish to extend with a method implementation. An interface can only define the signature of a method, and not provide an implementation.

As a concrete example, let's suppose that we are tired of having to write a `foreach` loop to print the contents of a collection of objects to the console like this:

```
foreach (var p in store.People)
{
    Console.WriteLine(p.Display());
}
```

It might be nice if we had a method on each class that implements the IEnumerable<IDisplayable> interface that can perform the looping for us in a more concise fashion.

Since we will be creating a number of similar methods, and they might be re-usable across other applications, create an Extensions folder in the UclaExt.Common project.

Now create a public static class inside the Extensions folder, which we will call `IEnumerableExtensions` like this:

```
using System;
using System.Collections;
using System.Collections.Generic;
using UclaExt.Common.Interfaces;

namespace UclaExt.Common.Extensions
{
    public static class IEnumerableExtensions
    {
        /// <summary>
        /// Prints a sequence of items to the
        /// console using the ToString method.
        /// </summary>
        /// <param name="items">IEnumerable<object> sequence of items</param>
        public static void PrintToConsole(IEnumerable<object> items)
        {
            foreach (IDisplayable idisp in items)
            {
                Console.WriteLine(idisp.ToString());
            }
        }


    }
}
```

I've chosen to make the parameter of type `IEnumerable<object>`, so it can work with all the .Net collection classes that implement `IEnumerable`, rather than just `List<T>`. Then we will be able to list all the objects to the console by calling the single line like this:

```
IEnumerableExtensions.PrintToConsole(store.People);
```

(You will, of course, need to add a project reference in the console project that references the UclaExt.Common project, and add a using directive for the UclaExt.Common.Extensions namespace to the Program.cs file.) Try substituting this for the foreach loop and verify that it works as well. This should not be very surprising, since all we have really done is encapsulate the loop in a static method that performs the looping for us.

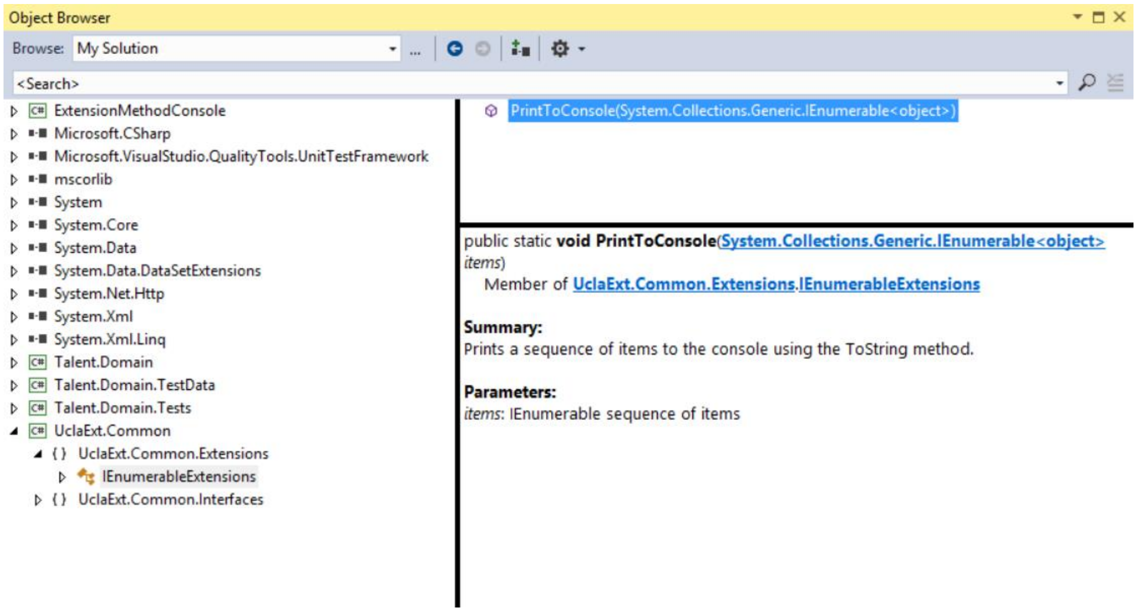
However, C# allows us to do this even more concisely with a language feature called an ***extension method***. To convert the static `PrintToConsole()` method to an extension method, **change** the method to:

```
public static void PrintToConsole(this IEnumerable<object> items)
{
    foreach (var item in items)
    {
        Console.WriteLine(item.ToString());
    }
}
```

Where all I have done is add the keyword "this" in front of the first parameter. Then I can call the extension method from Main using the single line:

```
store.People.PrintToConsole();
```

To the developer, the method actually looks like it is available on the List<Person> collection class, since we can call it with what looks like a standard method call syntax. Intellisense can even find the method, and since I provided an XML comment on the method, intellisense even shows the method summary. In fact, you can even open the Object Browser pane in VS and navigate to the XML documentation for the method:



public static **void PrintToConsole**(System.Collections.Generic.IEnumerable<object> *items*)
Member of UclaExt.Common.Extensions.IEnumerableExtensions

Summary:
Prints a sequence of items to the console using the ToString method.

Parameters:
items: IEnumerable sequence of items

Think about what this extension method does:

We normally cannot implement any kind of method on an interface, but an extension method allows us to do something almost equivalent. (Though an extension method will not be able to access private properties of the classes that implement the interface.)

Step 5: Implement IDisplayable for the Person class and add another `foreach` loop to the Main method in the console application, such that the person information is displayed something like this:

```
List of People and Age:
Ms. Judith Olivia Dench(81)
Mr. Stephen Frears(74)
Mr. Stephen John Coogan(50)
Ms. Sophie Kennedy Clark
Mr. Chris Buck
Ms. Jennifer Lee
Ms. Kristen Bell(35)
Josh Gad
```

Where the number in parentheses in the age and the parentheses are omitted if the Age is null.

With these modifications in place, we can add another extension method that operates on collections of IDisplayable objects.

Step 6: An extension method for IDisplayable collections. Now, in the same `ExtensionMethods` class, add the following method:

```
/// <summary>
/// Prints a sequence of IDisplayable items to the
/// console using the Display method.
/// </summary>
/// <param name="items">IEnumerable sequence of IDisplayable items</param>
public static void PrintToConsole(this IEnumerable<IDisplayable> items)
{
    foreach (IDisplayable idisp in items)
    {
        Console.WriteLine(idisp.Display());
    }
}
```

This is an overload of the previous method that is specific to collections of `IDisplayable` objects, like our People collection.

Now the same store.People.PrintToConsole(); line will call this method, because parameter IEnumerable<IDisplayable> is a better match than IEnumerable<Object>.

Now run the application and you should see that we get the more verbose display of employee information for each listed employee, since Person implements `IDisplayable`.

Step 8: Extension Methods for LINQ. Open MSDN (http://msdn.microsoft.com/library) and find the documentation on the class `List<T>`. Below the list of normal methods, you will find a list of dozens of Extension Methods, almost all of which are "Defined by Enumerable". If you look at the `Enumerable` class, you will see that it is a public static class in the `System.Linq` namespace with the description "Provides a set of static methods for querying objects that implement `IEnumerable<T>`". You should recognize that these are simply a large set of extension methods for the `IEnumerable<T>` interface. These methods implement a powerful set of methods that allow you to query any collection of objects in .Net that implements the

`IEnumerable<T>` interface which Microsoft calls ***Language Integrated Query***, or ***LINQ***. The primary reason that Microsoft chose to implement the extension method feature in .Net 3.0 was precisely to support LINQ.

Summary

In this lab we examined extension methods and learned how they can be used to extend the behavior of classes that we cannot (or don't want to) subclass or even to provide an implementation of methods that can be applied to interfaces. We can create extension methods in our own programs for our own purposes, but the driving force behind the extension method feature is that it provided a way for Microsoft to implement LINQ.

In the next exercise we will explore some of the methods in LINQ and see how to work with the LINQ extension methods to manipulate collections in .Net.

Finished Solution

You can download and run my finished solution from https://github.com/entrotech/Lab07-ExtensionMethods