

# Lab 3: Generics

## *Programming in C# for Visual Studio .NET Platform II*

### Objective

In this exercise, we will learn about generics. The idea of Generics is to provide a way to re-use algorithms that work with objects such that the same algorithms can be applied to various object types.

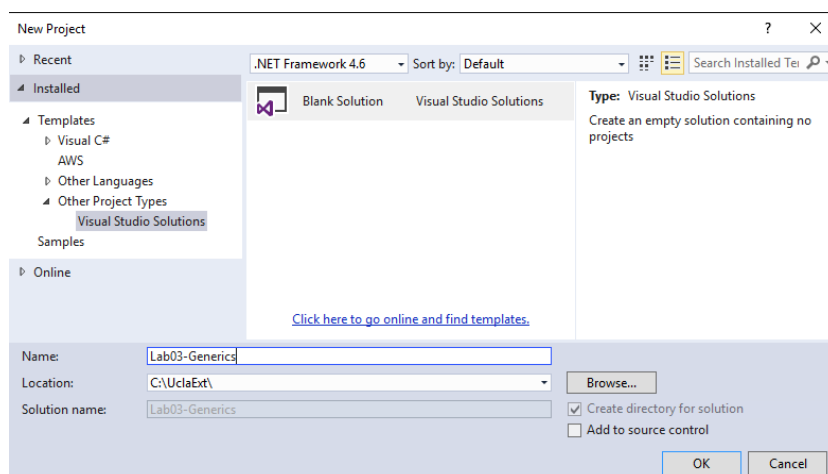
We will show what generics are and show a few good examples of when we would want to use them. Rather than a “deep dive” on all the things you can do with generics, we just explore the basics of what generics are, and talk about a few examples of using generics that will prove useful throughout the course. For a more thorough explanation, [Griffiths] and [Troelsen] each have very good chapters that cover generics in depth.

In the course of this lesson, we will also cover several great C# language features:

- Nullable Types
- Generic Collections, such as List<T>
- Filtering and sorting collections using **LINQ** or Language Integrated Query, a powerful set of tools for working with collections of objects in C#.

***It is important to work through this entire exercise and understand these concepts, as you will be using them all the time.***

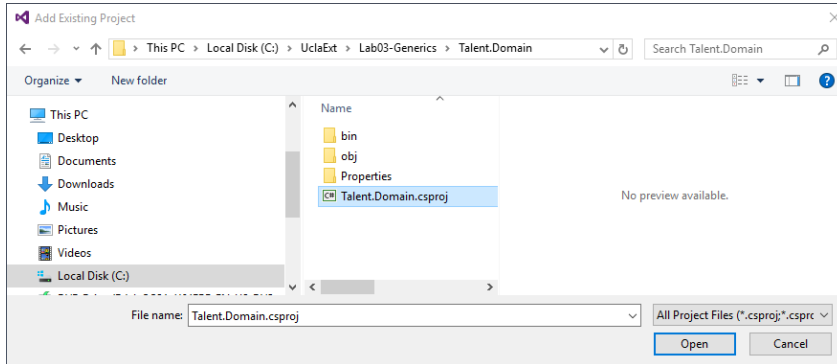
**Step 1: Create a new blank Visual Studio solution called Lab03-Generics in the UclaExt directory.**



**Step 2: Copy the projects Talent.Domain and Talent.TestData from Lab2 to the new solution.**

To do this, just copy the folders Talent.Domain and Talent.Domain.TestData from the Lab 2 folder to the new solution folder. (I would recommend using my solution to Lab2 to copy from.)

Then, in Solution Explorer, select the solution node, right-click and select Add | Existing Project from the context menu, then use the Add Existing Project dialog to navigate to the c:\UclaExt\Lab03-Generics\Talent.Domain\Talent.Domain.csproj file and Click Open to add it to the solution.

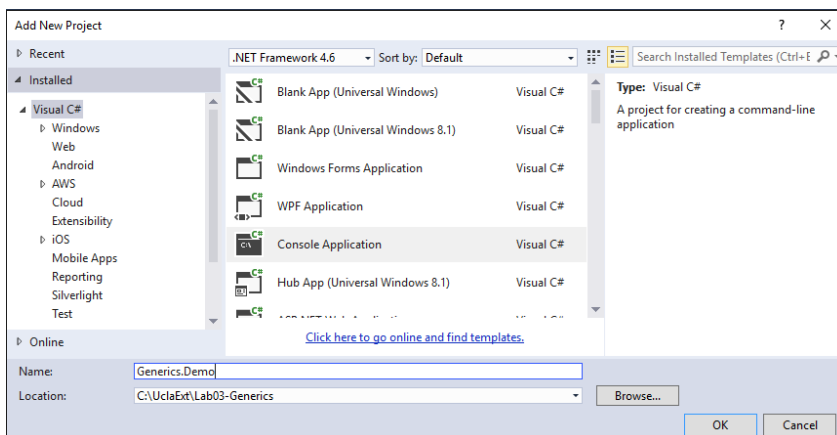


Repeat this procedure to add the Talent.Domain.TestData project to the solution as well, taking special care to make sure you are including both projects from the folders under the Lab03-Generics directory.

There should already be a project reference from the Talent.Domain.TestData project to the Talent.Domain project, but add this if it is missing.

You should be able to build the solution, but if you try to start it, you will get an error message saying “A Project with an Output Type of Class Library cannot be started directly.” This is because our solution only includes two class library projects, and we need to have an executable project of some sort for an executable application.

### Step 3: Add a new console application to the solution called Generics.Demo.



You can then set it to be the solution's start-up project by selecting the project in solution explorer, and choosing Set as Startup Project from the context menu. In solution explorer, the start-up project node will show in bold type.

Add project references to the Generics.Demo project to both the Talent.Domain and Talent.Domain.TestData projects.

Modify the Main method in Program.cs by adding a few lines of code:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Generics.Demo
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Lab 03 - Generics:");

            Console.ReadLine();
        }
    }
}
```

Now you should be able to compile and run the solution.

**Step 4: The IntPair class.** To start, let's create a simple class in the Generics.Demo project that contains two integers. This is pretty straightforward. Create the class IntPair in the Generics.Demo project. The code should look like this:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Generics.Demo
{
    public class IntPair
    {
        public IntPair(int item1, int item2)
        {
            Item1 = item1;
            Item2 = item2;
        }

        public int Item1 { get; set; }
        public int Item2 { get; set; }

        public void Swap()
        {
            int temp = Item1;
            Item1 = Item2;
            Item2 = temp;
        }

        public override string ToString()
        {

```

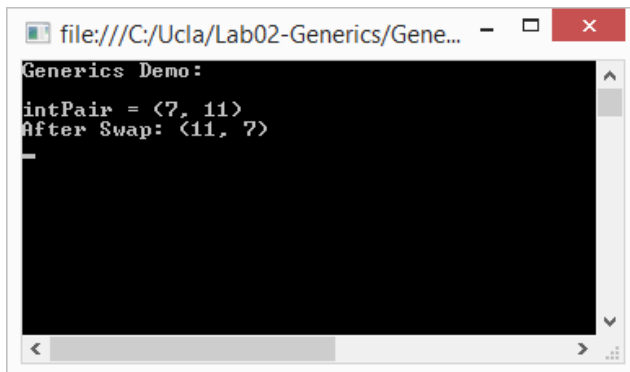
```
        return "(" + Item1.ToString() + ", " + Item2.ToString() + ");"  
    }  
}
```

If you look at this class, you can see that all it does is contain a pair of integers accessible via the Item1 and Item2 properties, along with a method that allows us to swap which integer is represented by the two properties.

We can exercise this class in our console application, by making the changes shown in bold blue to the Main method:

```
using System;  
using System.Collections.Generic;  
using System.Linq;  
using System.Text;  
using System.Threading.Tasks;  
  
namespace Generics.Demo  
{  
    class Program  
    {  
        static void Main(string[] args)  
        {  
            Console.WriteLine("Lab 02 - Generics:");  
  
            DemoWithoutGenerics();  
  
            Console.ReadLine();  
        }  
  
        private static void DemoWithoutGenerics()  
        {  
            IntPair intPair = new IntPair(7, 11);  
            Console.WriteLine("\r\nintPair = " + intPair);  
            intPair.Swap();  
            Console.WriteLine("After Swap: " + intPair);  
        }  
    }  
}
```

You should be able to compile and run the application, and see a console output like this:



```
file:///C:/Ucla/Lab02-Generics/Gene...  
Generics Demo:  
intPair = <7, 11>  
After Swap: <11, 7>
```

This is a reasonable class, and is **type-safe**, in that the compiler will not allow you to set either the Item1 or Item2 properties to anything except an integer.

**Step 5: The StringPair class.** If we then find a need for a class that represents a pair of strings, we would need to create another new class that is structurally similar, except that it works with strings, instead of ints. The code should look like this:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Generics.Demo
{
    public class StringPair
    {
        public StringPair(string item1, string item2)
        {
            Item1 = item1;
            Item2 = item2;
        }

        public string Item1 { get; set; }
        public string Item2 { get; set; }

        public void Swap()
        {
            string temp = Item1;
            Item1 = Item2;
            Item2 = temp;
        }

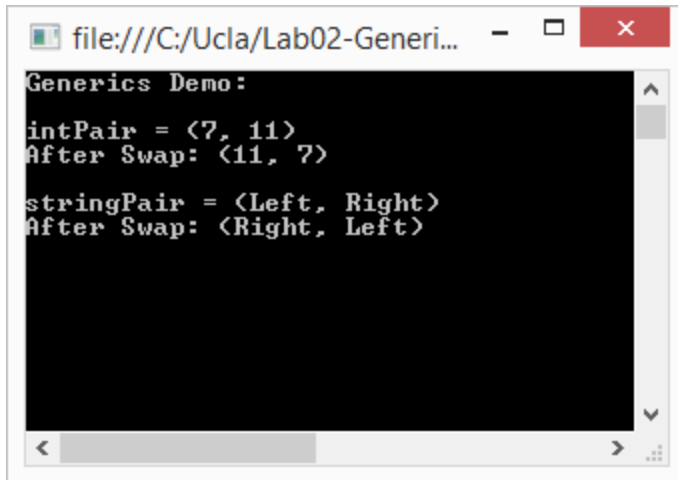
        public override string ToString()
        {
            return "(" + Item1.ToString() + ", " + Item2.ToString() + ")";
        }
    }
}
```

This class operates exactly like the IntPair class, except that it works on strings rather than ints.

You can add the following lines of code to the code in the DemoWithoutGenerics method:

```
StringPair stringPair = new StringPair("Left", "Right");
Console.WriteLine("\r\nstringPair = " + stringPair);
stringPair.Swap();
Console.WriteLine("After Swap: " + stringPair);
```

And then compile and run the application to see that it gives the results you would expect:



This all well and good, but you can imagine that at some point you might want similar classes for pairs of doubles, or Shows, or People, and so on. Each time you would need to create a whole new class that is functionally similar, but works with a different type. This could get pretty tedious and clutter your code with a large number of classes that do essentially the same thing on different types.

**Step 6: Generic Classes.** C# has a feature called **Generic Classes** that provide an elegant way to specify a class definition such that the compiler can create many versions of the class – each version tailored for working with a specific type. Create a new Pair class like this:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Generics.Demo
{
    public class Pair<T>
    {
        public Pair(T item1, T item2)
        {
            Item1 = item1;
            Item2 = item2;
        }

        public T Item1 { get; set; }
        public T Item2 { get; set; }

        public void Swap()
        {
            T temp = Item1;
            Item1 = Item2;
            Item2 = temp;
        }

        public override string ToString()
        {
            return "(" + Item1.ToString() + ", " + Item2.ToString() + ")";
        }
    }
}
```

You will notice that the class name is `Pair<T>`, where T in angle brackets is called a **Type Parameter**. Each place where `IntPair` declared an identifier, we have replaced it with the letter T.

When you want to write code that uses such a generic class, you need to tell the compiler what it should replace T with to create the actual class you need. For example to create an instance of `Pair` that works for integers, you can declare, instantiate and initialize it like this:

```
Pair<int> intPairG = new Pair<int>(7, 11);
```

At compile time, the compiler sees that you will be needing a class that looks like the class definition of `Pair<T>` where T is replaced with `int` throughout the class definition, to create a class just like `IntPair` that we created earlier. I like to think of this as something similar to a Microsoft Word mail merge, where the class definition has a placeholder, T, that gets merged with the various types that your program actually needs, and stamps out the specific class definitions you will need as separate classes. The class definition is called an open (or unbound) generic class, while the specific classes that get generated by the compiler after the type parameter replacement are called closed (or constructed) classes. If you create multiple `Pair<int>` instances in your code, the compiler will only create a single constructed class that will be used by all instances of `Pair<int>`.

Now add the following method to the body of the `Program` class after the `DemoWithoutGenerics` method:

```
private static void DemoGenericPair()
{
    Pair<int> intPairG = new Pair<int>(7, 11);
    Console.WriteLine(String.Format("\r\nintPair: {0}", intPairG));
    intPairG.Swap();
    Console.WriteLine(String.Format("After Swap: {0}", intPairG));

    Pair<string> stringPairG = new Pair<string>("Left", "Right");
    Console.WriteLine(String.Format("\r\nstringPair: {0}", stringPairG));
    stringPairG.Swap();
    Console.WriteLine(String.Format("After Swap: {0}", stringPairG));

    Pair<Genre> genrePair = new Pair<Genre>(
        new Genre { GenreId = 1, Code = "Prog", Name = "Programming", DisplayOrder = 10
    },
        new Genre { GenreId = 2, Code = "Adv", Name = "Adventure", DisplayOrder = 10 });
    Console.WriteLine(String.Format("\r\ngenrePair: {0}", genrePair));
    genrePair.Swap();
    Console.WriteLine(String.Format("After Swap: {0}", genrePair));
}
```

Then add a call to this `DemoGenericPair` method to the `Main` method, and comment out the previous call to `DomainWithoutGenerics`. You will also need to add a using directive to the other using directives in `Program.cs`

```
using Talent.Domain;
```

You should be able to compile and run the code to verify that all this works:

```

Generics Demo:
intPair = <7, 11>
After Swap: <11, 7>

stringPair = <Left, Right>
After Swap: <Right, Left>

intPair: <7, 11>
After Swap: <11, 7>

stringPair: <Left, Right>
After Swap: <Right, Left>

prodPair: <Programming, Adventure>
After Swap: <Adventure, Programming>

```

This is a nice feature to save us from having to write a lot of classes that are essentially similar except for the underlying types that the class manipulates, but it becomes much more important in situations where we want to build a class library for use by other code, where we don't have any idea ahead of time what the users of our class library will need to use in place of the type parameter T.

Though this example presented a generic class, the concept can also apply to structs, interfaces, methods, events and delegates in basically the same fashion.

**Step 7: Nullable<T>.** You can create your own generic classes as we just did in the previous steps. However, you will probably find that taking advantage of existing generic classes, such as those in the Base Class Library, is even more convenient.

A really good example of a generic in the Base Class library is the generic type `Nullable<T>`. As you know, **reference types** like variables referring to strings and custom classes often are not initialized such that the reference does not point to a constructed object, and we say that the reference variable is **null**. In many contexts, this means “unspecified” or “nothing”. If you do not initialize a **value type** variable (like an `int` or `Boolean`), it will be initialized with a default value for you (see <http://msdn.microsoft.com/en-us/library/83fhsxwc.aspx> for the default values by type). So there is no way for you to set something like an `int` to a value that means “unspecified”, but we will often need a way to do so.

The BCL defines a generic struct called `Nullable<T>` which addresses this need, where T can be any value type that we want to make “nullable”, and the `Nullable<T>` struct wraps this class up inside a type that has a `HasValue` property indicating if the value is “null” or not, plus a `Value` property that can be used to get the value only if it is not null.



Add the following method to the Program class below the method from the previous step, add a call to this method to the Main method and comment out the call to the DemoGenericPair method.

Try uncommenting the lines in bold/blue and see that an exception is thrown.

```
private static void DemoNullable()
{
    Console.WriteLine("\r\nUsing Nullable<T>:\r\n");

    Console.WriteLine("\r\nWhen int? is assigned a value:");

    int? nonNullIndex = -4;
    // Equivalent "Long Form" syntax:
    //Nullable<int> nonNullIndex = new Nullable<int>(-4);

    // HasValue returns true
    Console.WriteLine("nonNullIndex.HasValue = " + (nonNullIndex.HasValue ? "T" : "F"));

    // We can get value
    Console.WriteLine("nonNullIndex.Value = " + nonNullIndex.Value);

    // And can cast to int
    Console.WriteLine("nonNullIndex as int = " + (int)nonNullIndex);

    Console.WriteLine("\r\nWhen int? is NOT assigned a value:");

    int? nullIndex = null;
    //Nullable<int> nullIndex = new Nullable<int>(); // previous line is shortcut for
this

    // HasValue returns False
    Console.WriteLine("nullIndex.HasValue = " + (nullIndex.HasValue ? "T" : "F"));

    // If we try to get value or explicitly cast to int when HasValue is false, a
    // System.InvalidOperationException is thrown
    //Console.WriteLine("nullIndex.Value = " + nullIndex.Value);

    // Explicit casting will also fail
    //Console.WriteLine("nullIndex as int = " + (int)nullIndex);

    Console.WriteLine("\r\nSafely getting value from int?:");

    Console.WriteLine("nullIndex = " + (nullIndex.HasValue ? nullIndex.Value.ToString() :
    "No Value"));
    Console.WriteLine("nonNullIndex = " + (nonNullIndex.HasValue ?
    nonNullIndex.Value.ToString() : "No Value"));

    Console.WriteLine("nullIndex = " + nullIndex.GetValueOrDefault());
    Console.WriteLine("nonNullIndex = " + nonNullIndex.GetValueOrDefault());
}
```

This code shows you how to use the Nullable<T> type, which we will use extensively in our development – so it is important to know this particular type.

However, this example also shows how Microsoft was able to provide us with the ability to instantly create a nullable variant of any value type we might define by simply inserting the appropriate type parameter in the variable declaration like this:

```
Nullable<MyValueType> a;
```

(It's actually not very often that you will define your own value types, since they are usually for fairly simple values, and the BCL has already provided most of the value types you would normally need.)

(The rest of this step is just illustrative – you don't need to put this in your lab solution.)

Just to illustrate how easy it was for Microsoft to create `Nullable<T>`, I created a `Nillable<T>` generic class that does almost everything that `Nullable<T>` does like this:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Generics.Demo
{
    /// <summary>
    /// Cheap imitation of the System.Nullable <T>
    /// class for demonstration purposes"/>"/>
    /// </summary>
    /// <typeparam name="T"></typeparam>
    public struct Nillable<T>
    {
        private bool hasValue;
        private T _internalValue;

        public bool HasValue
        {
            get { return _hasValue; }
        }

        public T Value
        {
            get
            {
                if (!this.HasValue)
                    throw new InvalidOperationException(
                        "Cannot get value of Nillable<T> when HasValue is false.");
                return _internalValue;
            }
        }

        public Nillable(T internalValue)
        {
            internalValue = internalValue;
            _hasValue = true;
        }

        public T GetValueOrDefault()
        {
            return this.HasValue ? this.Value : default(T);
        }
    }
}
```

This does most of what the real `Nullable<T>` implementation does (except for allowing explicit casting or supporting the syntactic sugar), so I can add the following code

```
private static void DemoNillable()
{
    Console.WriteLine("\r\nUsing the Nillable<T> class: ");
}
```

```
Nullable<int> p = new Nullable<int>(5);
Nullable<int> q = new Nullable<int>();

Console.WriteLine("p = " + (p.HasValue ? p.Value.ToString() : "No Value"));
Console.WriteLine("q = " + (q.HasValue ? q.Value.ToString() : "No Value"));
}
```

To demonstrate that it is pretty much equivalent.

**Step 8: Use the `List<T>` collection class and generic interfaces.** The BCL also includes a very important set of generic types for various types of collections. The generic collection classes can be found in the `System.Collections.Generic` namespace, which VS references by default in most project templates, since the generic collections are almost always preferred over their non-generic counterparts. These include `List<T>`, `Queue<T>`, `Stack<T>`, `Dictionary<TKey, TValue>`, etc., which are each designed to have different combinations of sets of related capabilities defined by several interfaces.

For example, the `List<T>` class is a general purpose indexed collection, meaning that objects in the collection have an integer-valued “position” within the collection (the first value has an index of 0). The `List<T>` class implements the `IEnumerable<T>` and `IEnumerable` interfaces, which allow iteration through the items with the `foreach` statement. It also implements the `ICollection<T>` and `ICollection` interfaces, which provide capabilities that most collections need, such as a `Count` property which returns the number of items in the collection and methods to `Add` and `Remove` Items, `Clear` the collection, etc. The `ICollection<T>` and `ICollection` interfaces build on the `ICollection<T>` and `ICollection` interfaces by adding the `Item` property to get the object at a specified index position within the list, and the `Insert()` and `RemoveAt()` methods which allow inserting or removing an object at a particular indexed position within the collection. In addition, there are many, many extension methods that operate on any `List<T>` collection.

Using the `List<T>` class is very simple. We do not need to create our own collection class explicitly. Instead, we just need to make sure that our code includes a `using System.Collections.Generic;` directive, and then declare instances of the `List<T>` class with the appropriate type parameter and start using them.

If you open the `DomainObjectStore` class definition, it looks like this:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Talent.Domain.TestData
{
    public class DomainObjectStore
    {
        #region Fields

        List<MpaaRating> _mpaaRatings;
        List<Genre> _genres;
        List<CreditType> _creditTypes;
        List<HairColor> _hairColors;
        List<EyeColor> _eyeColors;
        List<Show> _shows;
        List<Person> _people;
        List<ShowGenre> _showGenres;
        List<Credit> _credits;

        #endregion

        #region Constructor

        public DomainObjectStore()
        {
            mpaaRatings = LoadMpaaRatings();
            _genres = LoadGenres();
            _creditTypes = LoadCreditTypes();
            _hairColors = LoadHairColors();
            _eyeColors = LoadEyeColors();
            _shows = LoadShows();
            _people = LoadPeople();
            _showGenres = LoadShowGenres(_shows);
            _credits = LoadCredits(_shows, _people);
        }

        #endregion

        #region Properties

        public List<MpaaRating> MpaaRatings { get { return _mpaaRatings; } }
        public List<Genre> Genres { get { return _genres; } }
        public List<CreditType> CreditTypes { get { return _creditTypes; } }
        public List<HairColor> HairColors { get { return _hairColors; } }
        public List<EyeColor> EyeColors { get { return _eyeColors; } }
        public List<Show> Shows { get { return _shows; } }
        public List<Person> People { get { return _people; } }
        public List<ShowGenre> ShowGenres { get { return _showGenres; } }
        public List<Credit> Credits { get { return _credits; } }

        #endregion

        #region Load Methods

        ...

        #endregion

        #region Helper Methods

        ...

        #endregion
    }
}
```

```
}  
}
```

We use a `List<T>` collection to hold strongly-typed collections of each object type, effectively creating an in-memory “database” of domain objects for us to work with. Each collection is made accessible outside the class by a corresponding read-only property, and the elided `Load` Methods are called by the constructor to create the sample objects, populate the collections, and set up the navigation properties that represent relationships among the various related objects.

Because we used `List<T>` collection objects:

1. There is no way to add anything except instances of `Genre` to the collection. In most cases, the line of code that tries to add the object will not even compile. (Though if the object type is not determined until runtime, then it is still possible to get a runtime exception.)
2. All of the features built into `List<T>` are available with no coding. We didn’t have to create any kind of custom class to get all this.
3. When working with objects from the collection, we do not need to cast to the correct type and are guaranteed not to have `Invalid Cast Exceptions` due to incorrect object types in the collection.

Add the following lines to your `Main` method to get started:

```
private static void DemoGenericList()  
{  
    Console.WriteLine("\r\nList<Genre>");  
    DomainObjectStore dos = new DomainObjectStore();  
  
    foreach (var g in dos.Genres)  
    {  
        Console.WriteLine(String.Format("\t({0}) {1}",  
            g.Code, g.Name));  
    }  
}
```

Modify your `Main` method to just call this method, and verify that it lists the `Genres`. We already did this in the previous lab, but now you should have a better idea of how it works.

`List<T>` has a vast number of features. If you look up `List<T>` in the MSDN documentation, you can see that it has well over 100 methods. Exploring these capabilities in depth would take us a few lessons to do (see chapters 5, 9 and 10 of Griffiths, for the details), so we will just look at the few things we will need to build our projects in the next few steps.

**Step 9: `IEnumerable<T>` and `foreach`.** In the previous lab on interfaces, we encountered the `IEnumerable` and `IEnumerator` (non-generic) interfaces, which allowed a `foreach` loop to iterate over an `ArrayList` of objects. The `List<T>` class implements the generic version of these interfaces, `IEnumerable<T>` and `IEnumerator<T>`, which provide type-safe iteration. Any

collection class that implements the `IEnumerable<T>` interface can be traversed using a `foreach` loop like this (the `foreach` construct requires that the collection class implement either `IEnumerable` or `IEnumerable<T>` and uses `IEnumerable<T>` if available):

```
foreach(var g in genres)
{
    Console.WriteLine(String.Format("{0} {1}",
        g.Code, g.Name));
}
```

The details of how `IEnumerable<T>` is implemented aren't too important to us right now, since our main goal right now is to just see how we can use generic collections. In general, collection classes that implement `IEnumerable` might not traverse the collection in any particular order, though the `List<T>` class does implement `IEnumerable<T>` such that the `foreach` loop starts with index 0 and progresses through the items in order by index.

**Step 10: Sorting the Person Collection revisited.** In the previous lab, we implemented a default sorting capability on the `Person` class by implementing the `IComparable` interface like this:

```
#region IComparable

public int CompareTo(object obj)
{
    Person other = obj as Person;
    if (other == null)
    {
        return -1;
    }
    int nameCompare = String.Compare(this.LastFirstName,
        other.LastFirstName, true);
    if(nameCompare != 0)
    {
        return nameCompare;
    }
    int suffixCompare = String.Compare(this.Suffix, other.Suffix, true);
    if(suffixCompare != 0)
    {
        return suffixCompare;
    }
    if(this.DateOfBirth.HasValue && other.DateOfBirth.HasValue)
    {
        return this.DateOfBirth.Value < other.DateOfBirth.Value ? -1 : 1;
    }
    return 0;
}

#endregion
```

We can improve upon this by implementing the `IComparable<Person>` interface instead. First modify the class declaration:

```
public class Person : IComparable<Person>
```

Then replace the implementation of `IComparable` with this:

```
#region IComparable<Person>

public int CompareTo(Person other)
{
```

```
int nameCompare = String.Compare(this.LastFirstName,
    other.LastFirstName, true);
if (nameCompare != 0)
{
    return nameCompare;
}
int suffixCompare = String.Compare(this.Suffix, other.Suffix, true);
if (suffixCompare != 0)
{
    return suffixCompare;
}
if (this.DateOfBirth.HasValue && other.DateOfBirth.HasValue)
{
    return this.DateOfBirth.Value < other.DateOfBirth.Value ? -1 : 1;
}
return 0;
}

#endregion
```

This is both safer and more concise, since:

1. When we write code that attempts to use the Sort (or other similar methods) on a collection of objects with incompatible types, intellisense can detect this problem before we even compile the application, and
2. Since the CompareTo method is passed an argument of the correct type, we not need to attempt to cast `other` to an object of the correct type and decide what to do if the cast fails.

**Step 13:** Suppose now that we want to provide other ways of sorting people – say by Age. When we implemented the `IComparable<Person>` interface we had to choose one way to sort people and pretty much “baked” the comparison rule into our class – there really isn’t a way to programmatically choose an alternate comparison rule when you use `IComparable`. If you go to the documentation for `List<T>.Sort()` you can see that there is an override that take two arguments, an array and an instance of an **`IComparer`**. The `IComparer` can be a class of our own design that implements the `IComparer` interface. So let’s create a new class just for this purpose called `PersonAgeComparer`. Create a new class in the library project called `PersonAgeComparer`. You will need to add a using statement to include the `System.Collections` namespace, and then you can implement the `IComparer` interface. When VS supplies the method stub, you will see that the interface has a single `Compare(object x, object y)`. Very much like the `CompareTo` method of the `IComparable` interface. Here is my complete implementation of the `PersonAgeComparer` class:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Collections;

namespace Talent.Domain
{
    public class PersonAgeComparer : IComparer<Person>
    {
```

```
public int Compare(Person p1, Person p2)
{
    int returnValue = 0;
    if(p1.DateOfBirth.HasValue && p2.DateOfBirth.HasValue)
    {
        returnValue = -1 *
            DateTime.Compare(p1.DateOfBirth.Value,
                p2.DateOfBirth.Value);
        if( returnValue != 0)
        {
            return returnValue;
        }
    }
    if(p1.DateOfBirth.HasValue ^ p2.DateOfBirth.HasValue)
    {
        // If only one has a DateOfBirth, it appears
        // after the one with null DateOfBirth.
        return p1.DateOfBirth.HasValue ? 1 : -1;
    }
    // If both have same DateOfBirth or both are
    // missing, fall back to default sort order by name
    return p1.CompareTo(p2);
}
}
```

Though the comparison rule itself just compares DateOfBirth values, there is a bit of code required to deal with the possibility that either or both of the arguments x and y are of a suitable type for comparison.

Now that we have a suitable Comparer class, we can use the List<T>.Sort(IComparer<T> comparer) method to sort our employee list. Just add the following code to the end of the Main method body:

```
Console.WriteLine("\r\nSorting People by Age");
dos.People.Sort(new PersonAgeComparer());
foreach (Person p in dos.People)
{
    Console.WriteLine(p.LastFirstName + " Age: " +
        ((p.Age.HasValue) ? p.Age.Value.ToString() : "Unknown"));
}
```

Run the code and verify that the people list is now sorted by Age (employees with null Age appear first).

Using this technique, we could create as many different “Comparer” classes as we want with various algorithms for comparing employees. At runtime, we can then choose which comparer to use, giving us a lot of flexibility.

Once again, we use an existing .Net interface and just provided an implementation to make our Person class (and all its subclasses) fully integrated with the .Net framework’s sorting capabilities.



## Summary

In this exercise, we took a step back from the first two exercises to look at some C# features that we used without explanation in creating our domain model or will find indispensable in working with domain objects as the course progresses:

- Generics: We looked at the basic concept of generics, then looked at two examples of generics types, `Nullable<T>` and `List<T>` that the Base Class Library implements for us.
- The `Nullable<T>` type allows us to have value types that can assume a “null” values when we need to be able to say that the value is not assigned or “null”. You need to recognize that a primitive type keyword followed by a question mark is an alias for the corresponding nullable type, for example `DateType?` is exactly equivalent to the `Nullable<DateTime>`.
- Though there are several generic collections, we will use `List<T>` most of the time. This is a great collection type, combining type safety, efficiency and all the power of LINQ.
- We saw how the concept of Generics also applies to interfaces, allowing for type-safe interfaces.

Each of these topics would require an entire lesson to explore in detail, but if we did that, we would never get to our goal of building a complete application, so the intent here is just to get a taste for the capabilities available to us, and learn to use the few features we will need in the upcoming lessons. I’d encourage you to read more about any of these topics that interest you. You can download my final solution for this exercise from the GitHub repository at <https://github.com/entrotech/Lab03-Generics>.

## Coming Attractions

Now that we have a domain model and know how to manipulate the domain objects, we will abandon our console applications for testing parts of our application and learn how to test properly with the MS Test unit testing framework. Unit testing is all about testing the smallest pieces of our application individually in an automated fashion. The idea is to gain confidence that all the small parts of our application work perfectly before we try to get them to work together.

## Further Reading

### Generics:

The core C# references [Griffiths], [Albahari], [Michaelis] all have good coverage of Generics.

Jeremy Clark has a good presentation that is available as a set of PowerPoint slides with an accompanying article and sample code, that he Titled “T, Earl Grey, Hot: Generics in .Net”. You

can find links to these materials on his Download page at  
<http://www.jeremybytes.com/Downloads.aspx>