Lab 2: Interfaces

Programming in C# for Visual Studio .NET Platform II

Objective

We have seen how inheritance allows us to abstract the common state and behaviors of very closely related classes into a common base class. This is helpful when the classes are really specialized versions of the same sort of object. However, it is often the case that very dissimilar classes should have identical methods, even though the classes may represent fundamentally different things. In this exercise, we explore C# Interfaces which provide a way to formally define a capability that classes may want to support. Once an interface is defined, various classes that choose to support the interface may be explicitly declared to support the interface and then provide methods, properties, etc. to implement this functionality as appropriate for that class. Users of classes that support a particular interface can then expect such classes to support the corresponding capability via the interfaces required methods and properties.

Here, we look at how interfaces work, and then look at some important examples that you will use often in your work with C#.

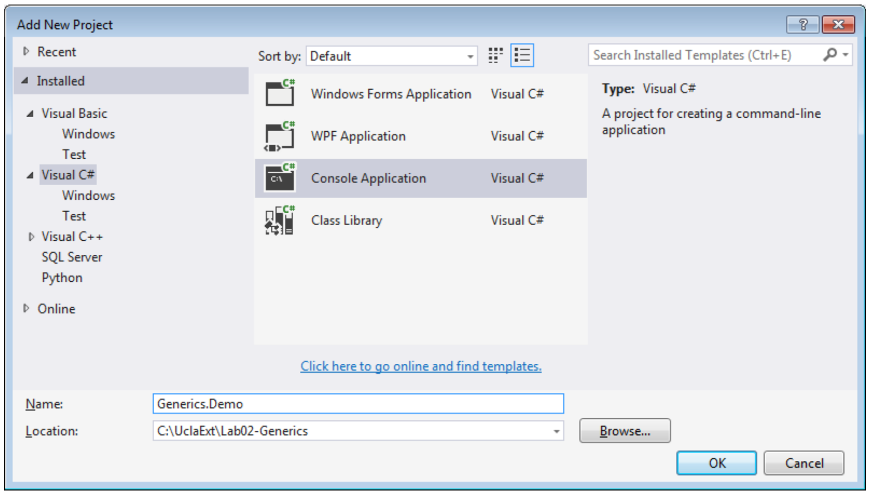
Procedure

Step 1: (removed)

Step 2: Download the starter solution. Download the zip file for the solution to Lab01 from the GitHub repository at https://github.com/entrotech/Lab01-DomainModel and unzip it to a folder C:\UclaExt\Lab01.

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 Interfaces.Demo.


Add New Project
Recent
Installed
Visual Basic
Windows
Test
Visual C#
Windows
Test
Visual C++
SQL Server
Python
Online
Sort by: Default
Search Installed Templates (Ctrl+E)
Windows Forms Application Visual C#
WPF Application Visual C#
Console Application Visual C#
Class Library Visual C#
Type: Visual C#
A project for creating a command-line application
Click here to go online and find templates.
Name: Generics.Demo
Location: C:\UclaExt\Lab02-Generics
Browse...
OK
Cancel

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 Interfaces.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;
using Talent.Domain;
using Talent.Domain.TestData;

namespace Interfaces.Demo
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Lab 02 - Interfaces:");

            DomainObjectStore store = new DomainObjectStore();


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

This creates the test data from the previous lab by instantiating a DomainObjectStore. If Visual Studio complains about the DomainObjectStore, check your project references and using directives.

Now you should be able to compile and run the solution – though it won't do anything interesting yet.

Defining, Implementing and using an Interface

In this part of the lab, we will review the basic concepts associated with Interfaces. Interfaces are applicable when objects of different classes "can do" something. In these cases, we define an **interface**, which represents a contract for a set of capabilities that the classes are expected to implement. As an example, suppose we expect that our application will need the ability to display a fairly detailed description of several different types of objects that otherwise aren't very closely related. To illustrate, we will create an IDisplayable interface that should be supported by classes that might need to display a human-readable string describing the object. We'll implement the IDisplayable interface for a few of our domain classes.

Step 4: Add a Display method to the LookupBase class definition.

```
public virtual string Display()
{
    return (Code ?? "") + " - "
        + (Name ?? "");
}
```

Of course, the Display method you implement on the LookupBase class will be inherited by each of the derived class, so you can now add a foreach loop to the Main method to display the Genres. (The virtual keyword, will allow us to override the implementation of the Display method on a derived class in the next step.)

```
Console.WriteLine("\r\nDisplay Genres:");
foreach (Genre genre in store.Genres)
{
    Console.WriteLine(genre.Display());
}
```

Step 5: Override the Display method for the MpaaRating class. Suppose we want the Display method to include the Description when we use it on an MpaaRating object. All we need to do is override the Display method in the MpaaRating class definition. You can even re-use the Display method by calling it from the MpaaRating.Display method override:

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

namespace Talent.Domain
{
    /// <summary>
    /// MPAA Rating, as defined by the MPAA at http://www.mpaa.org/film-ratings/
    /// </summary>
    public class MpaaRating : LookupBase
    {
        private string _description;

        public string Description
        {
            get { return _description; }
            set { _description = value; }
        }

        public override string ToString()
        {
            return Code;
        }

        public override string Display()
        {
            string msg = base.Display();
            if(Description != null)
            {
                msg += "\r\n\t" + Description;
            }
            return msg;
        }

    }
}
```

Note that:

- We need to use the override keyword when implementing the Display method.
- We can use the string returned by LookupBase.Display and just append the Description property.

Now add a foreach loop to the Main method that displays the MpaaRating descriptions. The results in the console should look like this:

```
Display MPAA Ratings:
G - General Audiences
        All ages admitted.
PG - Parental Guidance Suggested
        Some material may not be suitable for children.
PG-13 - Parents Are Strongly Cautioned
        Parents are urged to be cautious.
R - Restricted
        People under 17 years must be accompanied.
NC-17 - Adults Only
        Exclusively for adults.
UR - Unrated
        Not submitted for rating.
```

Step 6: Polymorphism using common base class. One of the key principles of Object-Oriented Programming is polymorphism. Polymorphism allows us to call a method on an abstraction (i.e. a base class or interface) of an object and have the object's concrete class determine exactly how that method is implemented. To illustrate, we can create an instance of any LookupBase object and call its Display method. If the object happens to be an MpaaRating, the Display method will behave differently than for other LookupBase-derived classes (like Genre) by including the description. This is often most useful when we need to work with a collection of objects that all derived from the same base class. Add the following code to the Main method:

```
Console.WriteLine("\r\nPolymorphism via Base class:");
List<LookupBase> lookups = new List<LookupBase>();
lookups.AddRange(store.Genres);
lookups.AddRange(store.MpaaRatings);
foreach (LookupBase lookup in lookups)
```

```
{
    Console.WriteLine(lookup.Display());
}
```

Note that we can create a List<LookupBase> collection. Even though we cannot explicitly create instances of type LookupBase (since this class is abstract), we can still create a collection of LookupBase objects that can hold a collection of objects, where each object is an instance of one of LookupBase's derived classes – i.e., Genres, MpaaRatings, etc.

We already have a collection of Genres and MpaaRatings in our store, and we can use the AddRange method to add both collections to the lookups collection, so our lookups collection is now a mixed collection of Genres and MpaaRatings.

The foreach loop allows us to iterate over this collection and call the Display method on each object to give the following result:

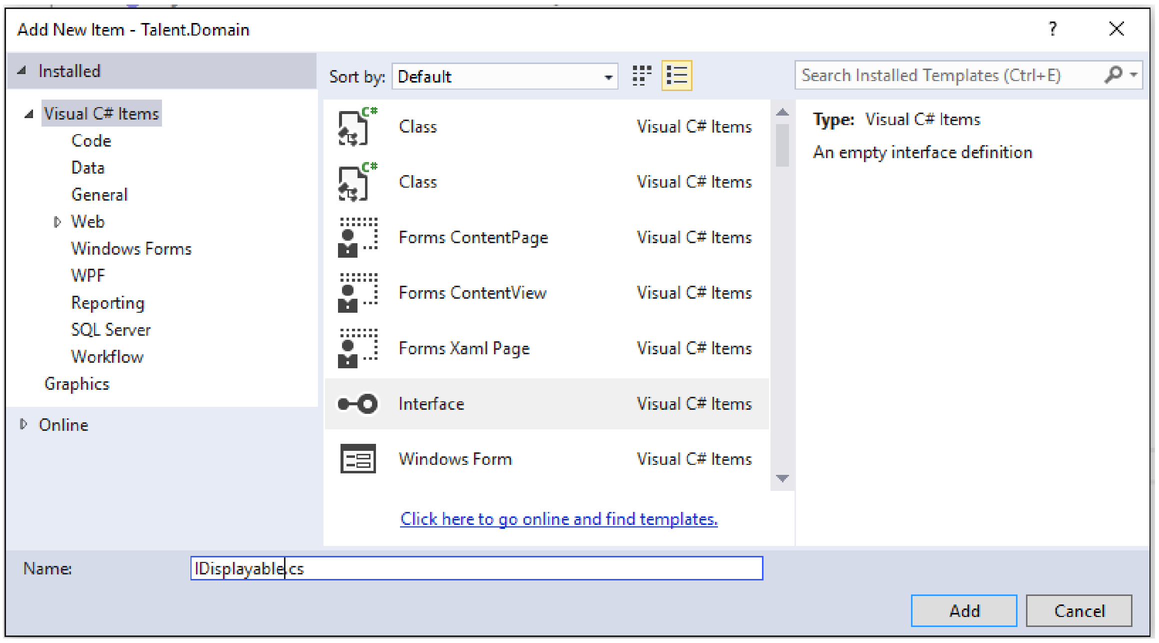
```
Polymorphism via Base class:
Dra - Drama
Com - Comedy
Sus - Suspense
Act - Action
Doc - Documentary
Ani - Animation
Adv - Adventure
G - General Audiences
        All ages admitted.
PG - Parental Guidance Suggested
        Some material may not be suitable for children.
PG-13 - Parents Are Strongly Cautioned
        Parents are urged to be cautious.
R - Restricted
        People under 17 years must be accompanied.
NC-17 - Adults Only
        Exclusively for adults.
UR - Unrated
        Not submitted for rating.
```

Polymorphism refers to the fact that as we iterated through this collection, each object executed the Display method as appropriate for its exact type.

Step 7: Create the IDisplayable interface. The ability to display a string with a fairly detailed description of the object is the kind of capability that we might want to have for a number of different classes in our system that really don't have much else in common. We might want to implement the Display method for Show, Person or other classes that do not have a common base class.

Interfaces allow us to formally define a named set of related capabilities in the form of methods, properties and events in .Net. Various classes that intend to support this capability can then implement the interface members and we will be able to use the interface to access these capabilities without regard to the concrete class type.

In the Talent.Domain project create an IDisplayable interface.


Add New Item - Talent.Domain

Class	Visual C# Items
Class	Visual C# Items
Forms ContentPage	Visual C# Items
Forms ContentView	Visual C# Items
Forms Xaml Page	Visual C# Items
Interface	Visual C# Items
Windows Form	Visual C# Items

Click here to go online and find templates.

Type: Visual C# Items

An empty interface definition

Name: IDisplayable.cs

Modify the code to look like this:

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

Note that the interface will need to be public to be accessible from our console application. Also note that the Display() method has no method body, since interfaces just represent a contract and do not contain any kind of implementation – it is up to the classes that implement the interface to do all of the work.

Step 8: Modify the Show class to implement the IDisplayable interface. First modify the class declaration to:

```
public class Show : IDisplayable
```

This instructs the compiler that we intend to implement the IDisplayable interface. If you try to compile at this point, you will get a compile error indicating that Show does not implement interface member Display(). An easy way to start implementing an interface is to double-click on the interface name in the class declaration, right-click and select “Implement Interface | Implement Interface” from the context menu. This will insert the following code at the end of the body of the class definition:

```
public string Display()
{
    throw new NotImplementedException();
}
```

Visual Studio knows what interface members need to be implemented to satisfy the IDisplayable interface and automatically creates method "stubs" for the interface members you need to implement, but it is up to you as the class designer to provide the implementation, so VS just throws an exception if the method is called.

Replace the Display method implementation such that the method will return the show title, followed by the Release year in parentheses:

```
public string Display()
{
    string msg = Title ?? "";
    if(ReleaseDate.HasValue)
    {
        msg += "(" + ReleaseDate.Value.Year + ")";
    }
    return msg;
}
```

Then add a foreach loop to the Main method that calls the Display method for each show and writes the result to the Console:

```
Console.WriteLine("\r\nDisplay Shows:");
foreach(Show show in store.Shows)
{
    Console.WriteLine(show.Display());
}
```

Step 9: Implement IDisplayable for the LookupBase class. This class already implements the Display method from Step 4, but we still want to explicitly declare that the LookupBase class implements IDisplayable, so the compiler will recognize that the Display method in intended to be an implementation of the IDisplayable interface. All we need to do is add IDisplayable to the class definition:

```
public abstract class LookupBase : IDisplayable
{
```

Note that:

- The signature of the Display method must exactly match the IDisplayable definition, or intelisense will alert you about the mismatch by telling you that the method IDisplayable.Display() is not implemented.
- We do not need to add IDisplayable to the classes derived from LookupBase – it is "inherited".

Step 10: Polymorphism via interfaces. Now that we have a formal interface defined, we can use IDisplayable's method on any type of class that supports this interface. To illustrate, we can create a collection of objects that support IDisplayable and then iterate over this collection to display a detailed listing. Try this:

```
Console.WriteLine("\r\nPolymorphism via Interface:");
List<IDisplayable> displayables = new List<IDisplayable>();
displayables.AddRange(store.Shows);
displayables.AddRange(store.MpaaRatings);
foreach (IDisplayable displayable in displayables)
{
    Console.WriteLine(displayable.Display());
}
```

This example is very similar to Step 6, where we displayed information on objects that share a common base class with an implementation of the Display method. Now though, we can display information on objects where all they have in common is that they provide an implementation of the interface we need.

Note that we can create a List<IDisplayable> collection. Even though we cannot explicitly create instances of type IDisplayable (since you cannot create an instance of an interface), we can still create a collection of IDisplayable objects that can hold a collection of objects, where each object is an instance of a class that implements the IDisplayable interface.

We already have collections of Shows and MpaaRatings in our store, and we can use the AddRange method to add both collections to the displayables collection, so our displayables collection is now a mixed collection of Shows and MpaaRatings.

The foreach loop allows us to iterate over this collection and call the Display method on each object to give the following result:

```
Polymorphism via Interface:
Philomena(2013)
Frozen(2013)
G - General Audiences
        All ages admitted.
PG - Parental Guidance Suggested
        Some material may not be suitable for children.
PG-13 - Parents Are Strongly Cautioned
        Parents are urged to be cautious.
R - Restricted
        People under 17 years must be accompanied.
NC-17 - Adults Only
        Exclusively for adults.
UR - Unrated
        Not submitted for rating.
```

The point here is that even though Shows and MpaaRatings are two completely different things, each supports the IDisplayable capability and can be displayed by calling the Display method. So implementing the IDisplayable interface allows quite different types of objects to respond to the Display method – each in its own way. Any object instantiated from IDisplayable “can do” the operations required to support the IDisplayable contract.

Using Existing Interfaces

In the preceding steps, we created and used an interface “from scratch”, doing three basic tasks:

1. Define the capability in the form of a contract called an interface,

2. Add the capability to one or more class definitions by implementing the interface, and
3. Use the interface from "client" code. We could just call the class's interface methods directly, or, more interestingly, create reference variable of the interface type (IDisplayable), and call the interface's methods from this reference, allowing us to polymorphically call the interface method on objects of completely different types (provided that each type implements the interface).

Now we look at a few situations where some of these steps are pre-built in the .Net Framework Class Library, and we only need to do part of the work.

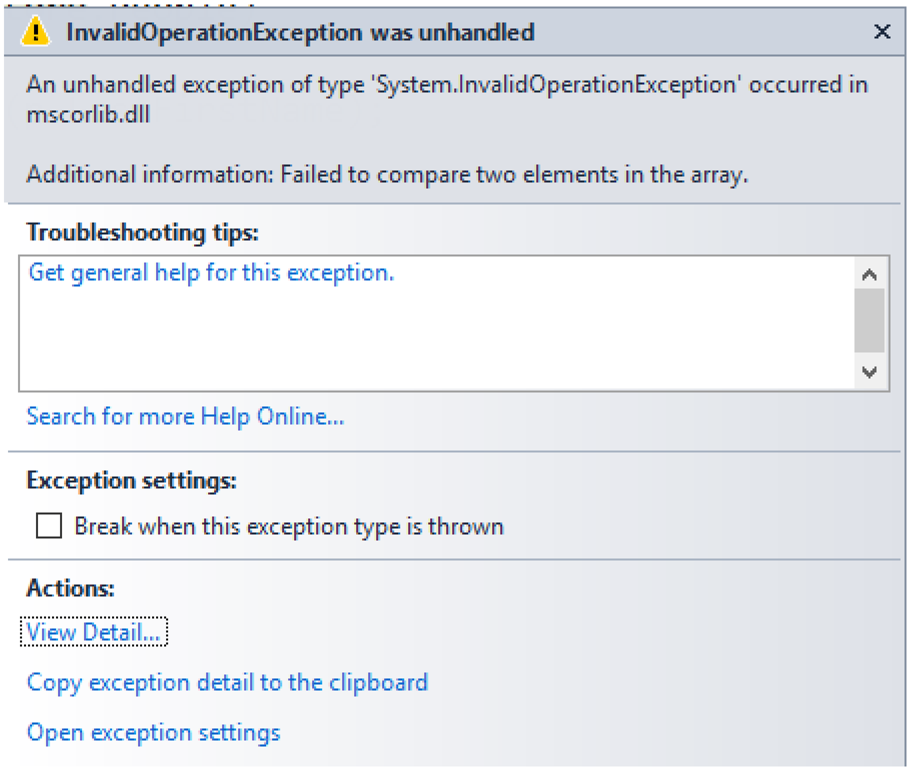
The use of interfaces is pervasive in the .Net Framework Class Library. The FCL includes hundreds of defined interfaces that we can use to integrate our own classes with built-in FCL functionality.

Step 11: **Providing a default sorting mechanism for classes.** Sorting algorithms have been studied extensively for decades in computer science and there is no reason for us to "re-invent the wheel" when it comes to sorting objects in our applications. The FCL incorporates extensive and highly optimized sorting algorithms ready for us to use with our own objects, if we simply supply an ordering rule that can be used to compare two objects and determine which should come first when sorted.

For example, let's look at the Sort() method of the List<T> class. If you look at the MSDN documentation for the Sort method, you see that it takes an existing List<T> collection and sorts the collection "in place" so the object are re-ordered within the List. So we can try to sort People using the following code:

```
Console.WriteLine("\r\nSorting People");
store.People.Sort();
foreach(Person p in store.People)
{
    Console.WriteLine(p.LastFirstName);
}
```

When you try to run this in the debugger, though, you will get the following exception:



InvalidOperationException was unhandled

An unhandled exception of type 'System.InvalidOperationException' occurred in mscorlib.dll

Additional information: Failed to compare two elements in the array.

Troubleshooting tips:

Get general help for this exception.

Search for more Help Online...

Exception settings:

☐ Break when this exception type is thrown

Actions:

View Detail...

Copy exception detail to the clipboard

Open exception settings

If you go to the MSDN documentation for the Sort() method, you will see a remark that says:

> *This method uses the default comparer Comparer<T>.Default for type* T *to determine the order of list elements.*
> *The Comparer<T>.Default property checks whether type* T *implements the IComparable<T> generic interface and uses that implementation, if available. If not, Comparer<T>.Defaultchecks whether type* T *implements the IComparable interface. If type* T *does not implement either interface, Comparer<T>.Default throws anInvalidOperationException.*

This method looks at the class (or interface) for the objects in our collection and tries to find an IComparable or IComparable<T> interface that it can use to compare any two objects to determine which should come first in the sorted collection. In other words, we need to supply a default sorting rule by implementing one of these interfaces. We'll implement the IComparable interface.

Step 12: Implement the IComparable interface for our Person class. Modify the class declaration for the Person class to implement the IComparable interface:

```
Public class Person : IDisplayable, IComparable
```

We just add IComparable to class declaration. We can highlight the word IComparable, right-click and select "Implement Interface | Implement Interface" and VS will create the method stubs for the interface. Modify the code for the resulting CompareTo method to look 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
```

Take a look at this code for a moment. The CompareTo(object obj) method is an instance method that accepts a reference to another object. When the Sort method (or any other code that might exploit the IComparable interface) calls the CompareTo method, it passes the method a reference to another object that the current object should be compared to. If the current instance is "less than" obj, then CompareTo should return a value less than one. If the current instance is "greater than" `obj`, then the CompareTo method should return a value greater than one, and if the current instance and `obj` are "equal", then CompareTo should return zero. It is up to us to decide what it means for one person to be "greater than" another person for purposes of sorting some sort of collection of people. In this case, I decided that I want the people to be sorted alphabetically by last name, then first name. If two employees have the same first and last names, then employees with the same name will be sorted by Suffix in alphabetical order, and if they are still the same, sort by DateOfBirth, if available. Also notice that `obj` refers to a reference type that might not be an employee at all, and I really can't come up with a sensible way of comparing employees to, say, products. So the common practice is to simply check to make sure `obj` is of an appropriate type, and, if not, just make an arbitrary decision about whether incomparable types are to be considered less than or greater than any employee. Now build and run the application and you should see that the employees are sorted by LastName then FirstName.

In this case, the .NET framework defined the IComparable interface for us, and the List<T>.Sort() method knew how to use the IComparable interface, so all we had to do was augment our Person class by implementing this interface to make it sortable. In fact, virtually all of the .Net collection classes that are sortable use the IComparable interface, so our Person class "can do" a comparison with other objects derived from EmployeeBase, so we can sort collections of employees in an Array, ArrayList, List, generic list, and a slew of other types of collections. This is very easy – and very powerful.

Step 13: In the previous step we looked at a case where the .Net framework defined an interface (IComparable) for us to implement in our own classes. Now let's look at two interfaces that .Net defines and implements in various classes for us to use, called **IEnumerable** and **IEnumerator**. Look up the description of IEnumerable on MSDN and you can see that collection classes that implement a GetEnumerator() method that returns an object that implements the IEnumerator interface. In particular, we can call the GetEnumerator() method to get an IEnumerator object for the collection, then we can call the IEnumerator's MoveNext() method repeatedly to progress from one item to the next in the collection, and use the GetCurrent() method to get a reference to the "current" item in the collection. MoveNext will eventually return false when there are no more items in the collection. Most of the collection classes in the .Net FCL implement IEnumerable, including ArrayList, so we can add the following code to our Main method to try it out:

```
Console.WriteLine("\r\nUse the IEnumerator interface:");
Console.WriteLine("\r\nUse the IEnumerator interface:");
ArrayList arrayList = new ArrayList()
{
    17, 45.67, 18.333M, store.People[0], "My String"
};
IEnumerator myEnumerator = arrayList.GetEnumerator();
while (myEnumerator.MoveNext())
{
    Console.WriteLine(myEnumerator.Current.ToString());
}
```

Since this code uses the older ArrayList collection class, you will need to add a using directive for the System.Collections class to the top of the Program.cs file.

Run this code and you should see a list of the ArrayList items (using the ToString() method to convert to a string for display):

```
Use the IEnumerator interface:
17
45.67
18.333
Kristen Bell
My String
```

In fact, the **`foreach`** operator in C# uses the IEnumerable interface, so only collections that implement IEnumerable can use the `foreach` operator. The above code is equivalent to

```
Console.WriteLine("\r\nUse foreach:\r\n");
foreach(var person in store.People)
{
    Console.WriteLine(person.ToString());
}
```

Add this to your main method to try it out. We will encounter several uses of the IEnumerable interface in subsequent lessons.

Note that in the case of IEnumerable, .Net provided the interface definition and a bunch of classes that implement IEnumerable, so all we need to do is use the methods that IEnumerable guarantees to take advantage of the enumeration capability. The `foreach` statement is just "syntactic sugar" that makes using the IEnumerable capability more concise.

Summary

Interfaces formally define a capability that a type (class) may choose to support. The interface itself does not implement any functionality, but instead should be thought of as a **capability** that a class might support. It is essentially a contract defining what operations a class can support (it could also define properties and events as well, though we haven't shown any examples in this exercise). When a class (or one of its base classes) is designed to **implement** an interface, then any code that uses this class can safely call interface methods on instances of the class, and instances of the class can be "upcast" to reference variables that have the interface name as their type (e.g., IDisplayable displayable = new Genre() as IDisplayable).

Though we can define our own interfaces, then design classes that implement this interface, and additional classes that consume the interface (as we did for IDisplayable), we will often use an interface that is already part of the Framework Class Library and either implement the interface in our own classes (as for IComparable), or take advantage of FCL classes that implement pre-defined interfaces for us to use (as for IEnumerable).

For more detailed information on interfaces, see the few pages in Chapter 3 of [Griffiths] on interfaces, or Chapter 7 of [Michaelis], or the MSDN documentation at these pages:

- https://msdn.microsoft.com/en-us/library/ms173156.aspx
- https://msdn.microsoft.com/en-us/library/87d83y5b.aspx

Or an entire chapter on interfaces from an older book on C#:

- https://msdn.microsoft.com/en-us/library/orm-9780596521066-01-13.aspx

My final solution to this exercise can be downloaded from the GitHub repository at https://github.com/entrotech/Lab02-Interfaces.

Homework

After completing this lab, Implement the IDisplayable interface for the Person class such that it lists the person's Full Name, followed by their Age in parentheses, then add code to the Main method to list all the people by calling the Display method for each.

Implement the IComparable interface for the Show class to sort alphabetically by Title and then by Release Year.