Lecture 7 - Working with interfaces

**IEnumerable**

Whenever we work with collection of objects, we might find ourselves in need to iterate the collection. The best way to iterate through a collection is by implementing the Iterator pattern. (refer: [Understanding and Implementing the Iterator Pattern in C# and C++](http://www.codeproject.com/Articles/362986/Understanding-and-Implementing-the-Iterator-Patter)[[^](http://www.codeproject.com/Articles/362986/Understanding-and-Implementing-the-Iterator-Patter)]). C# provides a very clean construct of foreach statement to iterate in a read only manner over a collection.

C# also provides us the possibility of using the same foreach construct and all the enumeration techniques on our custom collection objects by implementing the IEnumerable interface. So let us see how we can implementIEnumerable interface with our custom collection classes.

Using the code

Enumerating the Collection classes

Before starting the discussion let us see how we can use the Built-in classes and iterate over them. Lets start by looking into the ArrayList class that implements IEnumerable and see how we can have read only iteration over that using foreach statement.

*// Let us first see how we can enumerate an object implementing IEnumerable*

ArrayList list = new ArrayList();

list.Add("1");

list.Add(2);

list.Add("3");

list.Add('4');

foreach (object s in list)

{

Console.WriteLine(s);

}

Enumerating the Generic Collection classes

The Arraylist class is a generalized class that let us keep a collection. We can also have a generic class in which we can provide the type along with the data. Iterating over generic collection classes is also possible because they implementIEnumerable<T> interface. Lets see how we can enumerate over a generic collection.

*// Let us first see how we can enumerate an object implementing IEnumerable<T>*

List<string> listOfStrings = new List<string>();

listOfStrings.Add("one");

listOfStrings.Add("two");

listOfStrings.Add("three");

listOfStrings.Add("four");

foreach (string s in listOfStrings)

{

Console.WriteLine(s);

}

Now our objective is to have our own custom collection class and a generic collection class that should implement the IEnumerable and IEnumerable<T> interface respectively to provide the possibility of enumerating over them.

Understanding the yield keyword

Before jumping into the Implementation of these classes, we need to understand a very important keywordyield which actually facilitate the enumeration over collection. yield statement is used while returning a value from a function.

A normal method call like the one shown below will return only the first value no matter how many times it is called.

static int SimpleReturn()

{

return 1;

return 2;

return 3;

}

static void Main(string[] args)

{

*// Lets see how simeple return works*

Console.WriteLine(SimpleReturn());

Console.WriteLine(SimpleReturn());

Console.WriteLine(SimpleReturn());

Console.WriteLine(SimpleReturn());

}

The reason for this is that the normal return statement does not preserve the state of the function while returning. i.e. every call to this function is a new call and it will return the first value only.

Where as if I replace the return keyword by yield return then the function will become capable of saving its state while returning the value. i.e. when the function is called second time, it will continue the processing from where is has returned in the previous call.

static IEnumerable<int> YieldReturn()

{

yield return 1;

yield return 2;

yield return 3;

}

static void Main(string[] args)

{

*// Lets see how yield return works*

foreach (int i in YieldReturn())

{

Console.WriteLine(i);

}

}

When we run the above code it will return 1,2 and then 3. The only catch while using the yield return statement is that the function should return an IEnumerable and should be called from an iteration block i.e. foreachstatement.

Implementing IEnumerable in our custom Collection class

Now in our custom collection classes, if we define a function that will iterate over all the elements in the collection and return then using the yield keyword, we will be able to get hold of all the elements in the the collection.

So let us define our own MyArrayList class and implement IEnumerable interface, which will force us to implement the GetEnumerator function. This function will iterate over the collection and do a yield return on all the elements.

class MyArrayList : IEnumerable

{

object[] m\_Items = null;

int freeIndex = 0;

public MyArrayList()

{

*// For the sake of simplicity lets keep them as arrays*

*// ideally it should be link list*

m\_Items = new object[100];

}

public void Add(object item)

{

*// Let us only worry about adding the item*

m\_Items[freeIndex] = item;

freeIndex++;

}

*// IEnumerable Member*

public IEnumerator GetEnumerator()

{

foreach (object o in m\_Items)

{

*// Lets check for end of list (its bad code since we used arrays)*

if(o == null)

{

break;

}

*// Return the current element and then on next function call*

*// resume from next element rather than starting all over again;*

yield return o;

}

}

}

This class will now let us enumerate all the elements using a foreach stemement.

static void Main(string[] args)

{

*//Let us now go ahead and use our custom MyArrayList with IEnumerable implemented*

MyArrayList myList = new MyArrayList();

myList.Add("1");

myList.Add(2);

myList.Add("3");

myList.Add('4');

foreach (object s in myList)

{

Console.WriteLine(s);

}

}

**Note:** This class is neither complete not a very good implementation. The only purpose of the sample implementation is to demonstrate the implementation of IEnumerable interface.

Implementing IEnumerable<T> in our custom Generic Collection class

Let us now take this approach a little further and define a generic collection class capable of being enumerated. To do this we need to implement IEnumerable<T> interface.

class MyList<T> : IEnumerable<T>

{

T[] m\_Items = null;

int freeIndex = 0;

public MyList()

{

*// For the sake of simplicity lets keep them as arrays*

*// ideally it should be link list*

m\_Items = new T[100];

}

public void Add(T item)

{

*// Let us only worry about adding the item*

m\_Items[freeIndex] = item;

freeIndex++;

}

#region IEnumerable<T> Members

public IEnumerator<T> GetEnumerator()

{

foreach (T t in m\_Items)

{

*// Lets check for end of list (its bad code since we used arrays)*

if (t == null) *// this wont work is T is not a nullable type*

{

break;

}

*// Return the current element and then on next function call*

*// resume from next element rather than starting all over again;*

yield return t;

}

}

#endregion

#region IEnumerable Members

System.Collections.IEnumerator System.Collections.IEnumerable.GetEnumerator()

{

*// Lets call the generic version here*

return this.GetEnumerator();

}

#endregion

}

This class will now let us enumrate all the elements using a foreach stemement.

static void Main(string[] args)

{

*// Let us first see how we can enumerate an custom MyList<t> class implementing IEnumerable<T>*

MyList<string> myListOfStrings = new MyList<string>();

myListOfStrings.Add("one");

myListOfStrings.Add("two");

myListOfStrings.Add("three");

myListOfStrings.Add("four");

foreach (string s in myListOfStrings)

{

Console.WriteLine(s);

}

}

</t>

So now we have a collection class and a generic collectio class that implement IEnumerable andIEnumerable<T> respectively. Althogh These class is neither complete not a very good implementation but they do serve the purpose of the article i.e. to demonstrate the implementation of IEnumerable interface.

**IComparable**

It's easy to sort a list of strings or integers by just calling the List.Sort() method, but how can we sort two objects and based on what field?

Let's look at a small example and see how we can solve the problem using IComparable<> and IComparerinterfaces.

## Example

Let's create a simple Employee class with two fields, Name and Salary.

Hide   Copy Code

class Employee

{

public string Name { get; set; }

public int Salary { get; set; }

}

Now create a List of Employees and call Sort() method of a List.

Hide   Copy Code

*// Use Collection Initializers( C# 3.0 ) to initialize the List*

List<Employee> empList = new List<Employee>()

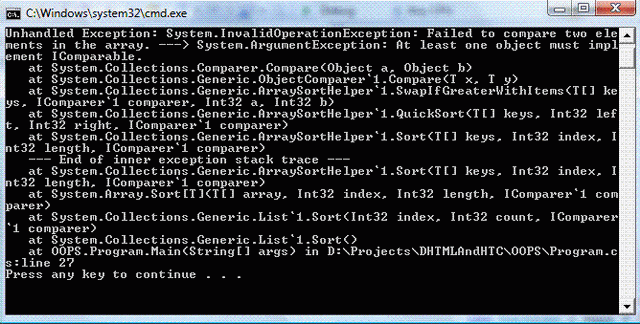
{ new Employee { Name = "a", Salary = 14000 },

new Employee { Name = "b", Salary = 13000 }

};

empList.Sort();

Oops! we got an exception!



The Exception says we need to implement IComparable<> interface,   
List.Sort() sorts any class that implements IComparable<> Interface which has one method calledCompareTo().

Let's implement IComparable<> interface to our Employee class:

Hide   Copy Code

class Employee : IComparable<Employee>

{

public string Name { get; set; }

public int Salary { get; set; }

#region IComparable<Employee> Members

public int CompareTo( Employee other )

{

if ( this.Salary < other.Salary ) return 1;

else if ( this.Salary > other.Salary ) return -1;

else return 0;

}

#endregion

}

In the above code, we are sorting objects based on salary of employee in descending order, by implementingCompareTo() method of IComparable interface which takes Employee reference as a parameter.  
Now, calling empList.Sort() gives no exception and empList is well sorted by salary.

But sometimes, we may need to sort a list of objects when class does not implement IComparable<> interface and also we may need various kinds of sorting on that class like:

1. Sort Employees by Salary in Ascending Order
2. Sort Employees by Salary in Descending Order
3. Sort Employees by Name

To solve this problem, .NET provides a special interface called IComparer<> which has a method Compare(), takes two object parameters X, Y and returns an int.  
Use of IComparer<> interface tells List how exactly you want to sort.

Hide   Shrink http://www.codeproject.com/images/arrow-up-16.png   Copy Code

class Employee\_SortBySalaryByAscendingOrder : IComparer<Employee>

{

#region IComparer<Employee> Members

public int Compare( Employee x, Employee y )

{

if ( x.Salary > y.Salary ) return 1;

else if ( x.Salary < y.Salary ) return -1;

else return 0;

}

#endregion

}

class Employee\_SortBySalaryByDescendingOrder : IComparer<Employee>

{

#region IComparer<Employee> Members

public int Compare( Employee x, Employee y )

{

if ( x.Salary < y.Salary ) return 1;

else if ( x.Salary > y.Salary ) return -1;

else return 0;

}

#endregion

}

class Employee\_SortByName : IComparer<Employee>

{

#region IComparer<Employee> Members

public int Compare( Employee x, Employee y )

{

return string.Compare( x.Name, y.Name );

}

#endregion

}

The above code introduces three classes by implementing Compare() method of IComparer interface.

Now, how do we use this? We just have to pass the reference of these classes as a object parameter to Sort()method as shown below:

Hide   Copy Code

*// Use Collection Initializers( C# 3.0 ) to initialize the List*

List<Employee> empList = new List<Employee>()

{ new Employee { Name = "a", Salary = 14000 },

new Employee { Name = "b", Salary = 13000 }

};

Employee\_SortBySalaryByAscendingOrder eAsc =

new Employee\_SortBySalaryByAscendingOrder();

*// Sort Employees by salary by ascending order.*

empList.Sort( eAsc );

Employee\_SortBySalaryByDescendingOrder eDsc =

new Employee\_SortBySalaryByDescendingOrder();

*// Sort Employees by salary by descending order.*

empList.Sort( eDsc );

Employee\_SortByName eName = new Employee\_SortByName();

*// Sort Employees by their names.*

empList.Sort( eName );

## Conclusion

Sorting Lists is simple as long as you sort basic elements like strings and integers for which comparison classes are defined.

Usage of IComparable<> and IComparer<> interface helps to sort Lists of objects on custom classes easily.

**Questions:**

1. **What interface should we implement to iterate through collection?**
2. **What is the difference between IComparer and IComparable?**
3. **How to sort collection using different criterias?**
4. **What is the main rule of implementing the interface?**
5. **What elements can we have in interface?**