IEnumerable and IEnumerator are implementation of the iterator pattern in .NET. I'll explain the iterator pattern and the problem it aims to solve in detail shortly. But if you're looking for a quick, pragmatic tip, remember that when a class implements IEnumerable, it can be enumerated. This means you can use a foreach block to iterate over that type.

In C#, all collections (eg lists, dictionaries, stacks, queues, etc) are enumerable because they implement the IEnumerable interface. So are strings. You can iterate over a string using a foreach block to get every character in the string.

Iterator Pattern

Consider the following implementation of a List class. (This is an over-simplified example and not a proper/full implementation of the List class).

```
public class List
{
    public object[] Objects;

    public List()
    {
        Objects = new object[100];
    }

    public void Add(object obj)
    {
        Objects[Objects.Count] = obj;
    }
}
```

The problem with this implementation is that the List class is exposing its internal structure (object[]) for storing data. This violates the *information hiding* principle of object-oriented programming. It gives the outside world intimate knowledge of the design of this class. If tomorrow we decide to replace the array with a binary search tree, all the code that directly reference the **Objects** array need to modified.

So, objects should not expose their internal structure. This means we need to modify our List class and make the **Objects** array private:

```
public class List
{
    private object[]  _objects;

    public List()
    {
        _objects = new object[100];
    }

    public void Add(object obj)
    {
        _objects[_objects.Count] = obj;
    }
}
```

Note that I renamed **Objects** to **_objects** because by convention private fields in C# should be named using camel notation prefixed with an underline.
So, with this change, we're hiding the internal structure of this class from the outside. But this leads to a new different problem: how are we going to iterate over this list? We no longer have access to the **Objects** array, and we cannot use it in a loop.
That's when the *iterator pattern* comes into the picture. It provides a mechanism to traverse an object irrespective of how it is internally represented.
IEnumerable and IEnumerator interfaces in .NET are implementations of the iterator pattern. So, let's see how these interfaces work, and how to implement them in our List class here.

IEnumerable interface represents an object that can be enumerated, like the List class here. It has one method:

```
public interface IEnumerable
{
    IEnumerator GetEnumerator();
}
```

The **GetEnumerator** method here returns an **IEnumerator** object, which can be used to iterate (or enumerate) the given object. Here is the declaration of the IEnumerator interface:

```
public interface IEnumerator
{
    bool MoveNext();
    object Current { get; }
    void Reset();
}
```

With this, the client code can use the MoveNext() method to iterate the given object and use the Current property to access one element at a time. Here is an example:

```
var enumerator = list.GetEnumerator();
while (enumerator.MoveNext())
{
      Console.WriteLine(enumerator.Current);
}
```

Note that with this interface, the client of our class no longer knows about its internal structure. It doesn't know if we have an array or a binary search tree or some other data structure in the List class. It simply calls GetEnumerator, receives an enumerator and uses that to enumerate the List. If we change the internal structure, this client code will not be affected whatsoever.

So, the iterator pattern provides a mechanism to iterate a class without being coupled to its internal structure.

Implementing IEnumerable and IEnumerator

So, now let's see how we can implement the IEnumerable interface on our List class. First, we need to change our List class as follows:

```
public class List : IEnumerable
{
    private object[] _objects;

    public List()
    {
        _objects = new object[100];
    }

    public void Add(object obj)
    {
        _objects[_objects.Count] = obj;
    }

    public IEnumerator GetEnumerator()
    {
    }
}
```

So I added the IEnumerable interface at the declaration of the class and also created the GetEnumerator method. This method should return an instance of a class that implements IEnumerator. So, we’re going to create a new class called ListEnumerator.

```
public class List : IEnumerable
{
    private object[] _objects;

    public List()
    {
        _objects = new object[100];
    }
```

```

        public void Add(object obj)
        {
            _objects[_objects.Count] = obj;
        }

        public IEnumerator  GetEnumerator()
        {
            return new ListEnumerator();
        }

        private class ListEnumerator  :  IEnumerator
        {
        }
    }
```

So, I modified the GetEnumerator method to return a new ListEnumerator. I also declared the ListEnumerator class, but I haven't implemented the members of the IEnumerator interface yet. That will come shortly.

You might ask: "Mosh, why are you declaring ListEnumerator as a nested private class? Aren't nested classes ugly?" The ListEnumerator class is part of the implementation of our List class. As you'll see shortly, It'll have intimate knowledge of the internal structure of the List class. If tomorrow I replace the array with a binary search tree, I need to modify ListEnumerator to support this. I don't want anywhere else in the code to have a reference to the ListEnumerator; otherwise, the internals of the List class will be leaked to the outside again.

Alright, so let's quickly recap up to this point. I implemented IEnumerable on our List class and defined the GetEnumerator method. This method returns a new ListEnumerator that the clients will use to iterate the List. I declared ListEnumerator as a private nested class inside List.

Now, it's time to complete the implementation of ListEnumerator. It's pretty easy:

```
public class ListEnumerator : IEnumerator
{
    private int _currentIndex = -1;

    public bool MoveNext()
    {
        _currentIndex++;

        return (_currentIndex < _objects.Count);
    }

    public object Current
    {
        get
        {
            try
            {
                return _objects[_currentIndex];
            }
            catch (IndexOutOfRangeException)
            {
                throw new InvalidOperationException();
            }
    }

    public void Reset()
    {
        _currentIndex = -1;
    }
}
```

Let's examine this class bit by bit.

The **_currentIndex** field is used to maintain the position of the current element in the list. Initially, it is set to -1, which is before the first element in the list. As we call the MoveNext method, it is incremented by one.
The **MoveNext** method returns a boolean value to indicate if we've reached the end of the list or not. Note that here in the MoveNext method, we have a reference to _objects. This is why I told our ListEnumerator has intimate knowledge of the internal structure of the List. It knows we're using an object[] there. If we replace the array with a binary search tree, we need to modify the MoveNext method. There are different traversal algorithms for trees.
The **Current** property returns the current element in the list. I've used a try/catch block here, incase the client of the List class tries to access the Current property before calling the MoveNext method. In this case, _currentIndex will be -1 and accessing _objects[-1] will throw IndexOutOfRangeException. I've caught this exception and re-thrown a more meaningful exception (InvalidOperationException). The reason for that is because I don't want the clients of the list to know anything about the fact that we're using an array with an index. So, IndexOutOfRange is too detailed for the clients of the List class to know and should be replaced with InvalidOperationException.
And finally, in the **Reset** method, we set _currentIndex back to -1, so we can re-iterate the List from the beginning, if we want.
So, let's review. I modified our List class to hide its internal structure by making the object[] private. With this, I had to implement the IEnumerable interface so that the clients of the List could enumerate it without knowing about its internal structure. IEnumerable interface has only a single method: GetEnumerator, which is used by the clients to enumerate the List. I created another class called ListEnumerator that knows how to iterate the List. It implements a standard interface (IEnumerator) and hides the details of how the List is enumerated.

The beauty of IEnumerable and IEnumerator is that we'll end up with a simple and consistent mechanism to iterate any objects, irrespective of their internal structure. All we need to is:

```
var enumerator = list.GetEnumerator();
while (enumerator.MoveNext())
{
      Console.WriteLine(enumerator.Current);
}
```
Any changes in the internals of our enumerable classes will be protected from leaking outside. So the client code will not be affected, and this means: more loosely-coupled software.

Generic IEnumerable<T> and IEnumerator<T>

In the examples in this post, I showed you the non-generic versions of these interfaces. These interfaces were originally added to .NET v1, but later Microsoft introduced the generic version of these interfaces to prevent the additional cost of boxing/unboxing. If you're not familiar with generics, check out my video on YouTube.

Misconception about IEnumerable and Foreach

A common misconception about IEnumerable is that it is used so we can iterate over the underlying class using a foreach block. While this is true on the surface, the foreach block is simply a syntax sugar to make your code neater. IEnumerable, as I explained earlier, is the implementation of the iterator pattern and is used to give the ability to iterate a class without knowing its internal structure.

In the examples earlier in this post, we used IEnumerable/IEnumerator as follows:

```
var enumerator = list.GetEnumerator();
while (enumerator.MoveNext())
{
       Console.WriteLine(enumerator.Current);
}
```
So, as you see, we can still iterate the list using a while loop. But with a foreach block, our code looks cleaner:

```
foreach (var item in list)
{
      Console.WriteLine(item);
```

```
    }
```

When you compile your code, the compiler translates your foreach block to a while loop like the earlier example. So, under the hood, it’ll use the IEnumerator object returned from GetEnumerator method.

So, while you can use the foreach block on any types that implements IEnumerable, IEnumerable is not designed for the foreach block!

https://programmingwithmosh.com/net/ienumerable-and-ienumerator/