Programming in .NET

Microsoft Development Center Serbia programming course

Lesson 4 – Collections and iterators

Collections

The C# Collection classes are a set of classes designed specifically for grouping together objects and performing tasks on them. A number of collection classes are available with C# and we will be looking at the key classes in this chapter.

Both the `List<T>` and `ArrayList` classes have properties very similar to C# arrays. One key advantage of these classes over arrays is that they can grow and shrink as the number of stored objects changes.

The `List<T>` class is contained with the `System.Collections.Generic` namespace whilst the `ArrayList` class is contained within the `System.Collections` namespace.

The syntax for creating a `List<T>` collection is as follows:

```
List<type> name = new List<type>();
```

An `ArrayList` object is created in a similar manner, although without the type argument:

```
ArrayList name = new ArrayList();
```

With the above syntax in mind we can now create a `List<T>` object called colorList:

```
using System;
using System.Collections.Generic;

public class Lists
{
        static void Main()
        {
                List<string> colorList = new List<string>();
        }
}
```

Once a List object has been created there are a number of methods which may be called to perform tasks on the list. Once such method is the `Add()` method which, as the name suggests, is used to add items to the list object:

```
List<string> colorList = new List<string>();

colorList.Add ("Red");
colorList.Add ("Green");
colorList.Add ("Yellow");
colorList.Add ("Purple");
colorList.Add ("Orange");
```

Individual items in a list may be accessed using the index value of the item (keeping in mind that the first item is index 0, the second index 1 and so on). The index value is placed in square brackets after the list name. For example, to access the second item in the colorList object:

```
Console.WriteLine (colorList[1]);
```

A list item value can similarly be changed using the index combined with the assignment operator. For example, to change the color from Yellow to Ingido:

A list item value can similarly be changed using the index combined with the assignment operator. For example, to change the color from Yellow to Ingido:

```
colorList[2] = "Indigo";
```

All the items in a list may be accessed using a `foreach` loop. For example:

```
foreach (string color in colorList)
{
        Console.WriteLine ( color );
}
```

When compiled and executed, the above code will output each of the color strings in the colorList object.

Items may be removed from a list using the `Remove()` method. This method takes the value of the item to be removed as an argument. For example, to remove the "Red" string from the colorList object:

```
colorList.Remove("Red");
```

It is important to note that items in a list may be duplicated. In the case of duplicated items, the `Remove()` method will only remove the first matching instance.

Previously we used the `Add()` method to add items to a list. The `Add()` method, however, only adds items to the end of a list. Sometimes it is necessary to insert a new item at a specific location in a list. The `Insert()` method is provided for this specific purpose. `Insert()` takes two arguments, an integer indicating the index location of the insertion and the item to be inserted at that location. For example, to insert an item at location 2 in our example list:

```
colorList.Insert(2, "White");
```

There is no way to tell C# to automatically sort a list as items are added. If the items in a list are required to be always sorted into order the `Sort()` method should be called after new items are added:

```
colorList.Sort();
```

A number of methods are provided with the List and `ArrayList` classes for the purposes of finding items. The most basic method is the `Contains()` method, which when called on a `List` or `ArrayList` object returns true if the specified item is found in the list, or false if it is not.

The `IndexOf()` method returns the index value of a matching item in a list. For example, the following code sample will output the value 2, which is the index position of the "Yellow" string:

```
List<string> colorList = new List<string>();

colorList.Add ("Red");
colorList.Add ("Green");
colorList.Add ("Yellow");
colorList.Add ("Purple");
colorList.Add ("Orange");

Console.WriteLine(colorList.IndexOf("Yellow"));
```

If the item is not found in the List a value of -1 is returned by the `IndexOf()` method.

This technique could be used to replace a specific value with another. For example, without knowing in advance the index value of the "Yellow" string we can change to "Black":

```
colorList[colorList.IndexOf("Yellow")] = "Black";
```

The `LastIndexOf()` method returns the index value of the last item in the list to match the specified item. This is particularly useful when a list contains duplicate items.

There are two class members that are useful for obtaining information about a C# List or `ArrayList` collection object. The Capacity property can be used to identify how many items a collection can store without having to resize.

The `Count` property, on the other hand, identifies how many items are currently stored in the list. For obvious reasons, `Capacity` will always exceed `Count`.

In instances where a large gap exists between Count and Capacity the excess capacity may be removed with a call the `TrimExcess()` method.

All the items in a list may be removed using the `Clear()` method:

```
colorList.Clear();
```

The `Clear()` method removes the items from the list and sets the `Count` property to zero. The Capacity property, however, remains unchanged. To remove the capacity of a list follow the `Clear()` method call with a call to `TrimExcess()`.

Iterators

An *enumerator* is a read-only, forward-only cursor over a sequence of values. In essence, it allows you to access all the elements in a sequence of items without caring about what kind of sequence it is – an array, a list a linked list, or none of the above. It's really effective for building data pipelines, where an item enters the pipeline and goes through a series of transformations or filters before coming out at the other end. Interfaces used for the iterator pattern are:

```
System.Collections.IEnumerator
System.Collections.Generic.IEnumerator<T>
```

In order to consume iterators, `foreach` statement is used. All standard collection classes in .NET can be used with `foreach` statement to iterate over all elements.

```
      // prints all elements in the collection
      // Every collection implements IEnumerable
      foreach (int x in new int[4] { 4, 3, 2, 1 })
      {
          Console.WriteLine(x);
      }
```

`Foreach` statement also works with other collections like `List<T>`. There is an overload constructor for `List<T>` that accepts `IEnumerable<T>` that will create a list out of the iterated elements. So the following example shows that static array can be used to construct a list.

```
      // every collection supports IEnumerable
      int[] static_array = new int[3] { 1, 2, 3 };

      List<int> dynamic_array = dynamic_array = new List<int>(static_array);

      // foreach works on both static and dynamic arrays
      foreach (var x in dynamic array)
          Console.WriteLine(x);
```

Even string support `IEnumerable<char>`, which we can see with the following example:

```
      foreach (char c in "We like deers :)")
          Console.WriteLine(c);
```

One important aspect of the iterator pattern is that we don't return all of the data in one go. The code that iterates over elements just asks for one element at a time. That means we need to keep track of how far we've already gone through our collection.

`IEnumerable<T>` interface defines one method `GetEnumerator()` (other than a base interface which is a non-generic version of `IEnumerable`) and looks like this:

```
public interface IEnumerable<out T> : IEnumerable
{
    IEnumerator<T> GetEnumerator();
}
```

How really `foreach` statement will reveal the next example. Last example will really be converted into the code:

```
  using (var enumerator = "We like deers :)".GetEnumerator())
  {
      while (enumerator.MoveNext())
      {
          var element = enumerator.Current;

          Console.WriteLine(element);
      }
  }
```

`IEnumerator<T>` is an object that is supposed to keep state of how much progress iterator made through our collection. And as we see from this `foreach` example, enumerator is an object that defines method `MoveNext()` and property `Current` that are used to make a progress to the next element and return that element to be used in `foreach` loop.

Let's suppose that we want to implement our own collection `ABCArray` that supports `foreach` statement and returns characters "A", "B", "C". It would be defined like this:

```
public class ABCArray : IEnumerable<string>
{
    private string[] x = new string[3] { "A", "B", "C" };

    public IEnumerator<string> GetEnumerator()
    {
         ...
    }

    IEnumerator IEnumerable.GetEnumerator()
    {
        return GetEnumerator();
    }
}
```

One could argue that the same class `ABCArray` can implement `IEnumerator<string>` as well and just return this from `GetEnumerator()`, but that is a bad approach – if `GetEnumerator` is called several times, several independent iterators should be returned. For instance, we should be able to use two `foreach` statements, one inside another, to get all possible pairs of values. The two iterators need to be independent, which suggests we need to create a new object each time `GetEnumerator` is called. Therefore much better idea is to use internal class `ABCArrayEnumerator` to implement `IEnumerator<string>`. The following code shows full implementation of `ABCArray` and `ABCArrayEnumerator` types.

```
public class ABCArray : IEnumerable<string>
{
    private string[]  letters = new string[3] { "A", "B", "C" };
```

```
        public IEnumerator<string> GetEnumerator()
        {
            return new MyArrayEnumerator(this);
        }

        IEnumerator IEnumerable.GetEnumerator()
        {
            return GetEnumerator();
        }

        // Iterator for ABCArray class
        internal class ABCArrayEnumerator : IEnumerator<string>
        {

            public ABCArrayEnumerator(ABCArray parent)
            {
                _parent = parent;
            }

            private ABCArray _parent;
            private int _cursor = -1;


            public string Current
            {
                get
                {
                    return _parent._letters[_cursor];
                }
            }

            public void Dispose()
            {
                _cursor = -1;
            }

            object System.Collections.IEnumerator.Current
            {
                get { return this.Current; }
            }

            public bool MoveNext()
            {
                // return true if successfully advanced to the next element
                // but advances the cursor first
                return (++_cursor < _parent._letters.Length);
            }

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

Important thing to not here is that `ABCArrayEnumerator` class actually holds the state of the cursor used to make progress through the elements and that returned value is used from `ABCArray` object itself that is passed through constructor to `ABCArrayEnumerator`.

Although the code is straightforward, this is a lot of code to do simple thing. Starting with C# 2.0, `yield return` is introduced which reduces the code presented above by far. With `yield return` the same `ABCArray` class can be implemented as:

```
    public class ABCArray : IEnumerable<string>
    {
```

```
        public IEnumerator<string> GetEnumerator()
        {
            yield return "A";
            yield return "B";
            yield return "C";
        }

        IEnumerator IEnumerable.GetEnumerator()
        {
            return GetEnumerator();
        }
    }
```

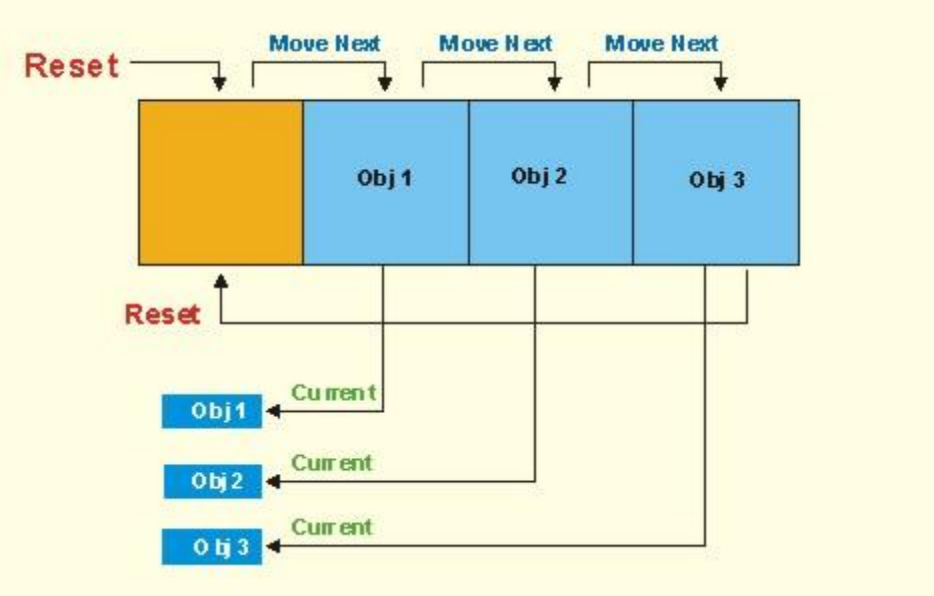
To be clear, this replaces the whole `ABCArrayEnumerator` class completely. Yield return tells C# compiler that this isn't a normal method but the one implemented with an iterator block. In order to use yield return a method or property needs to be declared with a return type `IEnumerable`, `IEnumerator`, `IEnumerable<T>` or `IEnumerator<T>` and needs to use the matching type (object for first two or specific T in last two) to the interface returned. In order to break iteration yield break can be used.

```
    public static IEnumerable<int> UsingYieldStatement(bool breakEarly)
    {
        yield return 42;

        // do some work

        yield return 43;

        if (breakEarly)
            yield break;

        // will not be called
        yield return 3;
    }
```

Normal return statements are not allowed within iterator blocks, only yield return. Another restriction is that catch block cannot be used within iterator blocks, only try and finally like in the following example

```
    public static IEnumerable<string> UsingYieldWithTry()
    {
        try
        {
            yield return "OK";
        }
        // catch { ... }    not allowed
        finally
        {
             //... it's ok to do work here
        }
        yield return "still OK";
    }
```

The following sequence diagram shows how the iterator is making progress through the elements.



Iterator pattern doesn't have to be used for iteration through a collection. The following example shows how a method can return dates in equal intervals between start and end date without putting all elements in a single array.

```
static IEnumerable<DateTimeOffset> SplitTimeFrame
      (DateTimeOffset startTime, DateTimeOffset endTime, TimeSpan interval)
{
    DateTimeOffset current = startTime;
    while(current + interval < endTime)
    {
        yield return current;
        newStart += interval;
    }
    yield return current;
}
```

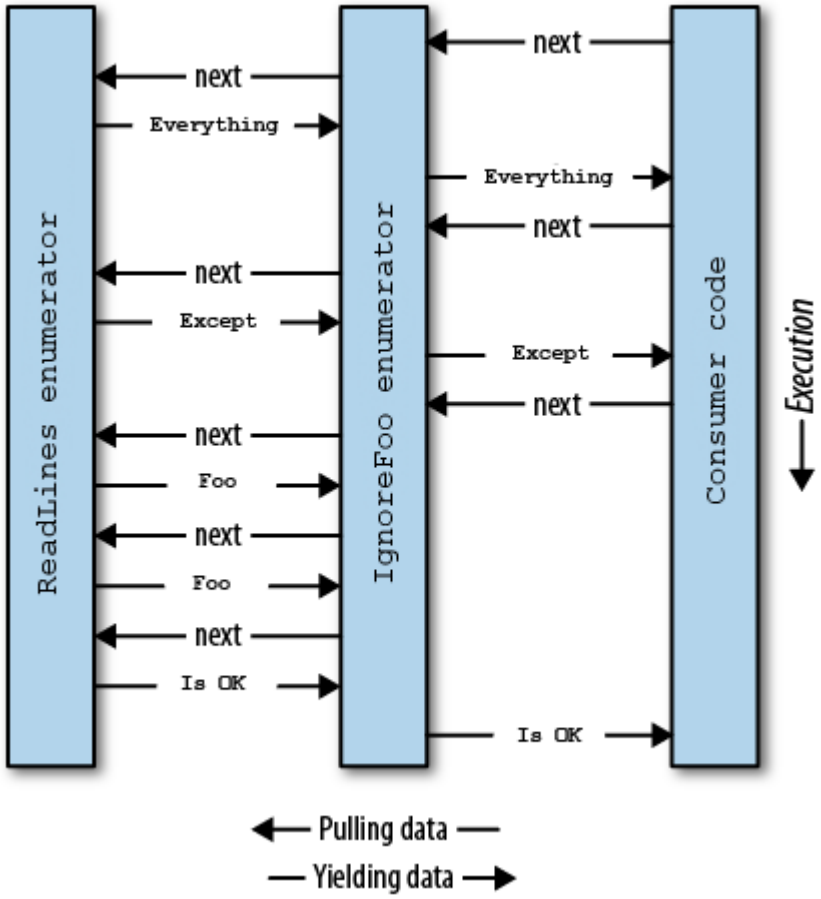
Another real life example of using yield return is iterating over lines in a file. `TextReader` is a type that reads lines over a text file with `ReadLine()` method which will return null once we reach the end of the file. This kind of a code is very common and now we can write it only once and pipeline it through a series of methods that can transform or filter lines before they reach the end. In order to create `TextReader` for a specific file we can use `File.OpenText` static method. The following example reads all lines from a text file ignores some specific lines and outputs the rest.

```
static IEnumerable<string> ReadLines(string filename)
{
    using (System.IO.TextReader reader = System.IO.File.OpenText(filename))
    {
        string line;
        while ((line = reader.ReadLine()) != null)
        {
            yield return line;
        }
    }
}

static IEnumerable<string> IgnoreFoo(IEnumerable<string> lines)
{
```

```
        foreach (string line in lines)
        {
            if (line != "Foo")
                yield return line;
        }
    }

    static void UseReadLines()
    {
        foreach (string line in IgnoreFoo(ReadLines("foo.txt")))
            Console.WriteLine(line);
    }
```

And with the following text in our foo.txt, the execution will look like on the chart below:

```
Everything
Except
Foo
Foo
Is OK
```



Recap

The purpose of this lesson was to show how powerful yield return and iterator blocks can be. Instead of returning all elements at once, they are postponed until requested by the client code. `foreach` statement can be used on all already provided .NET collection so any new data collection or type that makes sense to

support `foreach` statement should do so because that greatly improves usability and readability of that collection.