When To Use IEnumerable, ICollection, IList And List

I think that the question when to use *IEnumerable*, *ICollection*, *IList* or *List* is a common one that hasn’t often being answered in an easy way. I not only want to do this within this article, but I also want to give you some further information to understand the things more deeply.

If you get the understanding for the principle, you’ll be automatically really confident when you have to do this decision for the next time.

If you only want to know when to use which type, scroll down and have a look at the table providing the scenarios and the relevant types. I strongly recommend reading of the entire article to get a deeper understanding.  
  
Let’s first take a look at the individual types and more importantly its members. It’s generally a good idea to have a look at the types in question if you want to decide which type you should use in a specific situation in your code.

IEnumerable

First of all, it is important to understand, that there are two different interfaces defined in the .NET base class library. There is a non-generic *IEnumerable* interface and there is a generic type-safe *IEnumerable<T>* interface.

The IEnumerable interface is located in the *System.Collections* namespace and contains only a single method definition. The interface definition looks like this:

[?](http://www.claudiobernasconi.ch/2013/07/22/when-to-use-ienumerable-icollection-ilist-and-list/)

|  |
| --- |
| public interface IEnumerable  {      IEnumerator GetEnumerator();  } |

The GetEnumerator method must return an instance of an object of a class which implements the *IEnumerator* interface. We won’t have a look at the definition of the [IEnumerator](http://msdn.microsoft.com/en-us/library/system.collections.ienumerator.aspx" \t "_blank) interface this time, but if you are interested, please visit the [official msdn documentation](http://msdn.microsoft.com/en-us/library/system.collections.ienumerator.aspx).

It is important to know that the C# language **foreach** keyword works with all types that implement the *IEnumerable* interface. Only in C# [it also works with things that don’t explicitly implement IEnumerable or IEnumerable<T>](http://msdn.microsoft.com/en-us/library/9yb8xew9(v=vs.90).aspx). I believe you have been using the foreach keyword many times and without worrying about the reason why and how it worked with that type.

IEnumerable<T>

Let’s now take a look at the definition of the generic and type-safe version called *IEnumerable<T>*which is located in the*System.Collections.Generic*namespace*:*

[?](http://www.claudiobernasconi.ch/2013/07/22/when-to-use-ienumerable-icollection-ilist-and-list/)

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

As you can see the *IEnumerable<T>* interface inherits from the *IEnumerable* interface. Therefore a type which implements *IEnumerable<T>* has also to implement the members of *IEnumerable*.

*IEnumerable<T>* defines a single method *GetEnumerator* which returns an instance of an object that implements the *IEnumerator<T>* interface. We won’t have a look at this interface this time. Please take a look at the [official msdn documentation](http://msdn.microsoft.com/en-us/library/78dfe2yb.aspx) if you would like to get some more information.

ICollection

As you can imagine, there are also two versions of ICollection which are System.Collections.ICollection and the generic version System.Collections.Generic.ICollection<T>.

Let’s take a look at the definition of the ICollection interface type:

[?](http://www.claudiobernasconi.ch/2013/07/22/when-to-use-ienumerable-icollection-ilist-and-list/)

|  |
| --- |
| public interface ICollection : IEnumerable  {      int Count { get; }      bool IsSynchronized { get; }      Object SyncRoot { get; }        void CopyTo(Array array, int index);  } |

*ICollection* inherits from *IEnumerable*. You therefore have all members from the *IEnumerable* interface implemented in all classes that implement the *ICollection* interface.

I won’t go much into details of the defined methods and properties this time. I just want to let you know about the official description from the msdn documentation:

Defines size, enumerators, and synchronization methods for all nongeneric collections.

ICollection<T>

When we look at the generic version of ICollection, you’ll recognize that it does not look exactly the same as the non-generic equivalent:

[?](http://www.claudiobernasconi.ch/2013/07/22/when-to-use-ienumerable-icollection-ilist-and-list/)

|  |
| --- |
| public interface ICollection<T> : IEnumerable<T>, IEnumerable  {      int Count { get; }      bool IsReadOnly { get; }        void Add(T item);      void Clear();      bool Contains(T item);      void CopyTo(T[] array, int arrayIndex);      bool Remove(T item);  } |

The official msdn documentation looks like this:

Defines methods to manipulate generic collections.

In fact, we have some more methods to add, remove and clear a collection. The way synchronization was implemented differs also. I believe that this happened because the generic version of this interface was introduced with .NET 2.0 whereas the non-generic version was already introduced in .NET 1.1.

IList

The *IList* interface has of course a non-generic and a generic version. We start with looking at the non-generic *IList* interface:

[?](http://www.claudiobernasconi.ch/2013/07/22/when-to-use-ienumerable-icollection-ilist-and-list/)

|  |
| --- |
| public interface IList : ICollection, IEnumerable  {      bool IsFixedSize { get; }      bool IsReadOnly { get; }      Object this[int index] { get; set; }        int Add(Object value);      void Clear();      bool Contains(Object value);      int IndexOf(Object value);      void Insert(int index, Object value);      void Remove(Object value);      void RemoveAt(int index);  } |

*IList* implements *ICollection* and *IEnumerable*. In addition it provides method definitions for adding and removing elements and to clear the collection. It also provides methods for handling the positioning of the elements within the collection. It also provides an object indexer to allow the user to access the collection with square brackets like:

[?](http://www.claudiobernasconi.ch/2013/07/22/when-to-use-ienumerable-icollection-ilist-and-list/)

|  |
| --- |
| myList[elementIndex] |

IList<T>

Now let’s take a look at the generic version of IList:

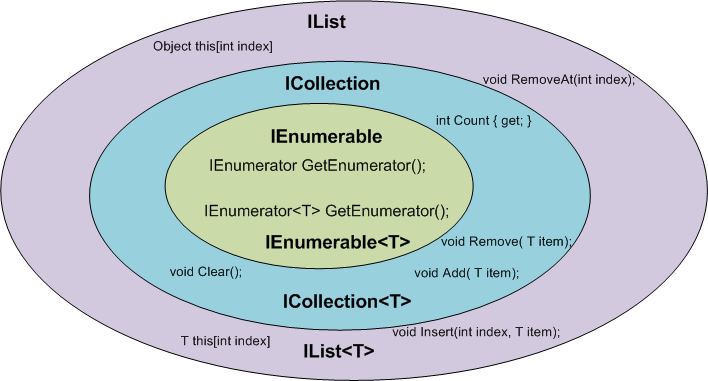
[?](http://www.claudiobernasconi.ch/2013/07/22/when-to-use-ienumerable-icollection-ilist-and-list/)

|  |
| --- |
| public interface IList<T> : ICollection<T>, IEnumerable<T>, IEnumerable  {      T this[int index] { get; set; }        int IndexOf(T item);      void Insert(int index, T item);      void RemoveAt(int index);  } |

As we mentioned before when discussing the *ICollection<T>* interface, there are some more methods defined in the *ICollection<T>* interface than in the *ICollection* interface. Therefore the member list of the *IList<T>* interface is a bit shorter than the non-generic equivalent. We only have some new methods for accessing a collection with specific positioning.

Conclusion

Take a look at the following graphic. Not every interface member is displayed, but it should be enough to give you an overview about all types we discussed.

[](http://cdn.claudiobernasconi.ch/wp-content/uploads/2013/07/Zeichnung-IEnumerable-ICollection-IList.png)

Which type should you depend on?

Now that we have looked at all of the interfaces in question, we are ready to decide which interface we should depend on in a certain situation. Generally, it’s a great idea to depend only on things we really need. I am going to show you how this decision can be made very easily and what you can expect to gain if you do so.

If you use a narrower interface type such as *IEnumerable* instead of *IList*, you protect your code against breaking changes. If you use *IEnumerable*, the caller of your method can provide any object which implements the *IEnumerable* interface. These are nearly all collection types of the base class library and in addition many custom defined types. The caller code can be changed in the future and your code won’t break that easily as it would if you had used *ICollection* or even worse *IList*.

If you use a wider interface type such as *IList*, you are more in danger of breaking code changes. If someone wants to call your method with a custom defined object which only implements *IEnumerable,* it simply won’t work and will result in a compilation error.

Generally you should always use that type that provides a contract for only the methods you really use.

The following table gives you an overview of how you can decide which type you should depend on.

|  |  |
| --- | --- |
| **Interface** | **Scenario** |
| IEnumerable, IEnumerable<T> | The only thing you want is to iterate over the elements in a collection. You only need read-only access to that collection. |
| ICollection, ICollection<T> | You want to modify the collection or you care about its size. |
| IList, IList<T> | You want to modify the collection and you care about the ordering and / or positioning of the elements in the collection. |
| List, List<T> | Since in object oriented design you want to [depend on abstractions instead of implementations](http://en.wikipedia.org/wiki/Dependency_inversion_principle), you should never have a member of your own implementations with the concrete type List/List. |