

When to use IList and when to use List

Asked 16 years, 4 months ago Modified 2 years, 10 months ago

Viewed 190k times



213



c#

.net



I know that IList is the interface and List is the concrete type but I still don't know when to use each one. What I'm doing now is if I don't need the Sort or FindAll methods I use the interface. Am I right? Is there a better way to decide when to use the interface or the concrete type?

Share

Improve this question

Follow

asked Aug 19, 2008 at 23:09



Rismo

6,577 ● 11 ● 38 ● 33

2 If anyone is still wondering, I find the best answers here:

stackoverflow.com/questions/400135/listt-or-ilstt

– Crismogram May 22, 2018 at 3:13

12 Answers

Sorted by:

Highest score (default)



There are two rules I follow:

212



- Accept the most basic type that will work
- Return the richest type your user will need

So when writing a function or method that takes a collection, write it not to take a List, but an `ICollection<T>`, an `IEnumerable<T>`, or `IEnumerator<T>`. The generic interfaces will still work even for heterogeneous lists because `System.Object` can be a `T` too. Doing this will save you headache if you decide to use a Stack or some other data structure further down the road. If all you need to do in the function is foreach through it, `IEnumerable<T>` is really all you should be asking for.

On the other hand, when returning an object out of a function, you want to give the user the richest possible set of operations without them having to cast around. So in that case, if it's a `List<T>` internally, return a copy as a `List<T>`.

Share Improve this answer

edited Aug 20, 2008 at 1:58

Follow

answered Aug 20, 2008 at 1:33



Lee

18.7k ● 6 ● 60 ● 61

55 You shouldn't treat input/output types any differently. Input and output types should *both* be the most basic type (preferably interface) that will support clients needs. Encapsulation relies on telling clients as little about the implementation of your class as possible. If you return a concrete List, you can't then change to some other better

type without forcing all of your clients to re-compile/update.

– [Ash](#) Sep 12, 2010 at 8:53

17 I disagree with the 2 rules... I would use most primitive type and specially when returning in this case `ICollection` (better `IEnumerable`) and you should work with `List` in your function inside. Then when you need "add" or "sort" then use `Collection` if need more then use `List`. So my hard rule would be: START always with `ICollection` and if you need more then extend... – [ethem](#) Apr 26, 2013 at 18:25

3 For your convenience, the "two rules" have a name: [robustness principle \(a.k.a. Postel's law\)](#). – [easoncxz](#) Jun 13, 2015 at 11:12

8 I very strongly disagree about Point #2, especially if this is at a service/api boundary. Returning modifiable collections can give the impression that the collections are "live" and calling methods like `Add()` and `Remove()` may have effects beyond just the collection. Returning a read-only interface such as `ICollection` is often the way to go for data-retrieval methods. Your consumer can project it into a richer type as-needed. – [STW](#) Sep 23, 2016 at 22:29

1 Alas, `ICollection` is the only collection interface that allows fast indexed access `[]`. Often one of my requirements is returning such an indexed collection. My other requirement is being able to change the actual collection (list, array, ...) without API changes, meaning the return type should be an interface. These requirements bring us to `ICollection`. Unfortunately, this also grants `Add()`, among other methods (which even *throws* for arrays!). So the collection interfaces are less than ideal. Alternatively, return a `ReadOnlyCollection`, e.g. by calling `resultList.AsReadOnly()`. – [Timo](#) Oct 24, 2016 at 12:01



Microsoft guidelines as checked by FxCop discourage use of `List<T>` in public APIs - prefer `ICollection<T>`.

64



Incidentally, I now almost always declare one-dimensional arrays as `ICollection<T>`, which means I can consistently use the `ICollection<T>.Count` property rather than `Array.Length`. For example:



```
public interface IMyApi
{
    ICollection<int> GetReadOnlyValues();
}

public class MyApiImplementation : IMyApi
{
    public ICollection<int> GetReadOnlyValues()
    {
        List<int> myList = new List<int>();
        ... populate list
        return myList.AsReadOnly();
    }
}

public class MyMockApiImplementationForUnitTests : IMy
{
    public ICollection<int> GetReadOnlyValues()
    {
        ICollection<int> testValues = new int[] { 1, 2, 3 };
        return testValues;
    }
}
```

Share Improve this answer

answered Sep 17, 2008 at 17:08

Follow



to StackOverflow

125k ● 33 ● 210 ● 341

4 I like this explanation / example the most! – [JonH](#) Nov 4, 2009 at 18:12



40



IEnumerable

You should try and use the least specific type that suits your purpose.

`IEnumerable` is less specific than `IList`.

You use `IEnumerable` when you want to loop through the items in a collection.

IList

`IList` implements `IEnumerable`.

You should use `IList` when you need access by index to your collection, add and delete elements, etc...

List

`List` implements `IList`.

Share Improve this answer

Follow

edited Mar 21, 2020 at 6:40



[Ola Ström](#)

5,271 ● 6 ● 30 ● 49

answered Jul 20, 2013 at 11:07



[rajesh](#)

401 ● 4 ● 2

-
- 9 Excellent, clear answer, which I marked as helpful. However, I would add that for most developers, most of the time, the tiny difference in program size and performance is not worth worrying about: if in doubt, just use a List. – [Graham Laight](#)
Sep 10, 2014 at 8:57
-



There's an important thing that people always seem to overlook:

33



You can pass a plain array to something which accepts an `IList<T>` parameter, and then you can call `IList.Add()` and will receive a runtime exception:



Unhandled Exception: System.NotSupportedException: Collection was of a fixed size.

For example, consider the following code:

```
private void test(IList<int> list)
{
    list.Add(1);
}
```

If you call that as follows, you will get a runtime exception:

```
int[] array = new int[0];
test(array);
```

This happens because using plain arrays with `IList<T>` violates the Liskov substitution principle.

For this reason, if you are calling `IList<T>.Add()` you may want to consider requiring a `List<T>` instead of an `IList<T>`.

Share Improve this answer

answered Oct 30, 2013 at 13:47

Follow

8	11	22	1
21	2	7	12
3	24	9	6
10	5	4	23

[Matthew Watson](#)

109k ● 12 ● 174 ● 295

2 This is trivially true for every interface. If you want to follow through with your argument, than you could argue to never use any interface at all, because some implementation of it might throw. If you, on the other hand, consider the suggestion given by OP to prefer `List<T>` over `IList<T>`, you should also be aware of the reasons why `IList<T>` is recommended. (For example blogs.msdn.microsoft.com/kcwalina/2005/09/26/...)

– [Micha Wiedenmann](#) Aug 29, 2017 at 7:49

4 @MichaWiedenmann My answer here is specific to when you are calling `IList<T>.Add()`. I'm not saying that you *shouldn't* use `IList<T>` - I'm just pointing out a possible pitfall. (I tend to use `IEnumerable<T>` or `ReadOnlyList<T>` or `ReadOnlyCollection<T>` in preference to `IList<T>` if I can.) – [Matthew Watson](#) Aug 29, 2017 at 8:06



I would agree with Lee's advice for taking parameters, but not returning.

24



If you specify your methods to return an interface that means you are free to change the exact implementation later on without the consuming method ever knowing. I thought I'd never need to change from a `List<T>` but had



to later change to use a custom list library for the extra functionality it provided. Because I'd only returned an `ICollection<T>` none of the people that used the library had to change their code.

Of course that only need apply to methods that are externally visible (i.e. public methods). I personally use interfaces even in internal code, but as you are able to change all the code yourself if you make breaking changes it's not strictly necessary.

Share Improve this answer

answered Aug 20, 2008 at 9:12

Follow



ICR

14.2k ● 4 ● 51 ● 78



9



It's always best to use the lowest base type possible. This gives the implementer of your interface, or consumer of your method, the opportunity to use whatever they like behind the scenes.



For collections you should aim to use `IEnumerable` where possible. This gives the most flexibility but is not always suited.

Share Improve this answer

answered Aug 19, 2008 at 23:41

Follow



tgmdbm

1,563 ● 12 ● 19

- 1 It is always best to **accept** the lowest base type possible. Returning is a different story. Choose what options are likely to be useful. So you think your client might want to use

indexed access? Keep them from `ToList()` -ing your returned `IEnumerable<T>` that was already a list, and return an `IList<T>` instead. Now, clients can benefit from what you can provide without effort. – [Timo](#) Oct 24, 2016 at 12:06



5



If you're working within a single method (or even in a single class or assembly in some cases) and no one outside is going to see what you're doing, use the fullness of a List. But if you're interacting with outside code, like when you're returning a list from a method, then you only want to declare the interface without necessarily tying yourself to a specific implementation, especially if you have no control over who compiles against your code afterward. If you started with a concrete type and you decided to change to another one, even if it uses the same interface, you're going to break someone else's code unless you started off with an interface or abstract base type.

Share Improve this answer

answered Aug 19, 2008 at 23:20

Follow



[Mark Cidade](#)

99.8k ● 33 ● 229 ● 237



4



You are most often better off using the most general usable type, in this case the `IList` or even better the `IEnumerable` interface, so that you can switch the implementation conveniently at a later time.



However, in .NET 2.0, there is an annoying thing - `ICollection` does not have a **Sort()** method. You can use a supplied adapter instead:

```
ArrayList.Adapter(list).Sort()
```

Share Improve this answer

edited Sep 17, 2008 at 17:22

Follow

answered Sep 17, 2008 at 17:14



petr k.

8,100 ● 7 ● 43 ● 52



2

I don't think there are hard and fast rules for this type of thing, but I usually go by the guideline of using the lightest possible way until absolutely necessary.



For example, let's say you have a `Person` class and a `Group` class. A `Group` instance has many people, so a `List` here would make sense. When I declare the list object in `Group` I will use an `ICollection<Person>` and instantiate it as a `List`.



```
public class Group {  
    private ICollection<Person> people;  
  
    public Group() {  
        this.people = new List<Person>();  
    }  
}
```

And, if you don't even need everything in `ICollection` you can always use `IEnumerable` too. With modern compilers and processors, I don't think there is really any speed difference, so this is more just a matter of style.

Share Improve this answer

answered Aug 19, 2008 at 23:17

Follow



[swilliams](#)

48.8k ● 27 ● 102 ● 130

5 why not make it a just a List in the first place? I still don't understand why bonus you get from making it a ICollection then in the constructor you make it into a List<> – [chobo2](#) Dec 19, 2011 at 20:56 ✎

I agree, if you are explicitly creating a List<T> object then you lose the advantage of the interface? – [The_Butcher](#) Sep 18, 2017 at 13:15

This still doesn't seem to cover the reasoning to use ICollection over List specifically. – [Bonez024](#) Feb 2, 2022 at 19:46



2



You should use the interface only if you need it, e.g., if your list is casted to an ICollection implementation other than List. This is true when, for example, you use NHibernate, which casts ICollections into an NHibernate bag object when retrieving data.



If List is the only implementation that you will ever use for a certain collection, feel free to declare it as a concrete List implementation.

Share Improve this answer

answered Aug 19, 2008 at 23:20

Follow



Jon Limjap

95.3k ● 15 ● 103 ● 153



In situations I usually come across, I rarely use IList directly.

1



Usually I just use it as an argument to a method



```
void ProcessArrayData(IList almostAnyTypeOfArray)
{
    // Do some stuff with the IList array
}
```

This will allow me to do generic processing on almost any array in the .NET framework, unless it uses IEnumerable and not IList, which happens sometimes.

It really comes down to the kind of functionality you need. I'd suggest using the List class in most cases. IList is best for when you need to make a custom array that could have some very specific rules that you'd like to encapsulate within a collection so you don't repeat yourself, but still want .NET to recognize it as a list.

Share Improve this answer

answered Aug 19, 2008 at 23:17

Follow



Dan Herbert

103k ● 51 ● 192 ● 221



A `List` object allows you to create a list, add things to it, remove it, update it, index into it and etc. `List` is used

1

whenever you just want a generic list where you specify object type in it and that's it.



`IList` on the other hand is an Interface. Basically, if you want to create your own custom `List`, say a list class called `BookList`, then you can use the Interface to give you basic methods and structure to your new class.

`IList` is for when you want to create your own, special sub-class that implements `List`.

Another difference is: `IList` is an Interface and cannot be instantiated. `List` is a class and can be instantiated. It means:

```
IList<string> list1 = new IList<string>(); // this is  
  
IList<string> list2 = new List<string>(); // this wil  
List<string> list3 = new List<string>(); // this wil
```

Share Improve this answer

Follow

edited Feb 16, 2022 at 17:30



UkFLSUI

5,672 ● 6 ● 37 ● 49

answered Aug 19, 2013 at 4:53



Javid

27 ● 2