IEnumerable vs List - What to Use? How do they work?

Asked 9 years ago Active 2 months ago Viewed 408k times

616

267

I have some doubts over how Enumerators work, and LINQ. Consider these two simple selects:

```
List<Animal> sel = (from animal in Animals
                    join race in Species
                    on animal.SpeciesKey equals race.SpeciesKey
                    select animal).Distinct().ToList();
```

or

```
IEnumerable<Animal> sel = (from animal in Animals
                           join race in Species
                           on animal.SpeciesKey equals race.SpeciesKey
                           select animal).Distinct();
```

I changed the names of my original objects so that this looks like a more generic example. The query itself is not that important. What I want to ask is this:

```
foreach (Animal animal in sel) { /*do stuff*/ }
```

1. I noticed that if I use `IEnumerable`, when I debug and inspect "sel", which in that case is the IEnumerable, it has some interesting members: "inner", "outer", "innerKeySelector" and "outerKeySelector", these last 2 appear to be delegates. The "inner" member does not have "Animal" instances in it, but rather "Species" instances, which was very strange for me. The "outer" member does contain "Animal" instances. I presume that the two delegates determine which goes in and what goes out of it?
2. I noticed that if I use "Distinct", the "inner" contains 6 items (this is incorrect as only 2 are Distinct), but the "outer" does contain the correct values. Again, probably the delegated methods determine this but this is a bit more than I know about IEnumerable.
3. Most importantly, which of the two options is the best performance-wise?

The evil List conversion via `.ToList()`?

Or maybe using the enumerator directly?

By using our site, you acknowledge that you have read and understand our Cookie Policy, Privacy Policy, and our Terms of Service.

c# linq list ienumerable

edited Sep 12 '12 at 13:53
svick
184k 42 304 425

asked Sep 2 '10 at 15:05
Axonn
3,909 5 25 34

10 Answers

679

`IEnumerable` describes behavior, while List is an implementation of that behavior. When you use `IEnumerable`, you give the compiler a chance to defer work until later, possibly optimizing along the way. If you use ToList() you force the compiler to reify the results right away.

Whenever I'm "stacking" LINQ expressions, I use `IEnumerable`, because by only specifying the behavior I give LINQ a chance to defer evaluation and possibly optimize the program. Remember how LINQ doesn't generate the SQL to query the database until you enumerate it? Consider this:

```
public IEnumerable<Animals> AllSpotted()
{
    return from a in Zoo.Animals
           where a.coat.HasSpots == true
           select a;
}

public IEnumerable<Animals> Feline(IEnumerable<Animals> sample)
{
    return from a in sample
           where a.race.Family == "Felidae"
           select a;
}

public IEnumerable<Animals> Canine(IEnumerable<Animals> sample)
{
    return from a in sample
           where a.race.Family == "Canidae"
           select a;
}
```

By using our site, you acknowledge that you have read and understand our Cookie Policy, Privacy Policy, and our Terms of Service.

```
var Leopards = Feline(AllSpotted());
var Hyenas = Canine(AllSpotted());
```

So is it faster to use List over `IEnumerable` ? Only if you want to prevent a query from being executed more than once. But is it better overall? Well in the above, Leopards and Hyenas get converted into *single SQL queries each*, and the database only returns the rows that are relevant. But if we had returned a List from `AllSpotted()` , then it may run slower because the database could return far more data than is actually needed, and we waste cycles doing the filtering in the client.

In a program, it may be better to defer converting your query to a list until the very end, so if I'm going to enumerate through Leopards and Hyenas more than once, I'd do this:

```
List<Animals> Leopards = Feline(AllSpotted()).ToList();
List<Animals> Hyenas = Canine(AllSpotted()).ToList();
```

edited Nov 22 '15 at 20:53
Amirhossein Mehrvarzi
7,443 4 30 54

answered Sep 2 '10 at 15:36
Chris Wenham
19.1k 10 53 67

1 Hi and thanks for answering ::-). You gave me a very good example of how a case when clearly the IEnumerable case is performance-advantaged. Any idea regarding that other part of my question? Why the Enumerable is "split" into "inner" and "outer"? This happens when I inspect the element in debug/break mode via mouse. Is this perhaps Visual Studio's contribution? Enumerating on the spot and indicating input and output of the Enum? – Axonn Sep 2 '10 at 17:12

11 I think they refer to the two sides of a join. If you do "SELECT * FROM Animals JOIN Species..." then the inner part of the join is Animals, and the outer part is Species. – Chris Wenham Sep 2 '10 at 20:16

9 When I've read the answers about: **IEnumerable<T> vs IQueryable<T>** I saw the analogical explanation, so that IEnumerable automatically forces the runtime to use LINQ to Objects to query the collection. So I'm confused between these 3 types. stackoverflow.com/questions/2876616/… – Bronek Jan 6 '14 at 20:55

3 @Bronek The answer you linked is true. `IEnumerable<T>` will be LINQ-To-Objects after the first part meaning all spotted would have to be returned to run Feline. On the other hand an `IQuertable<T>` will allow the query to be refined, pulling down only Spotted Felines. – Nate Aug 20 '14 at 17:49

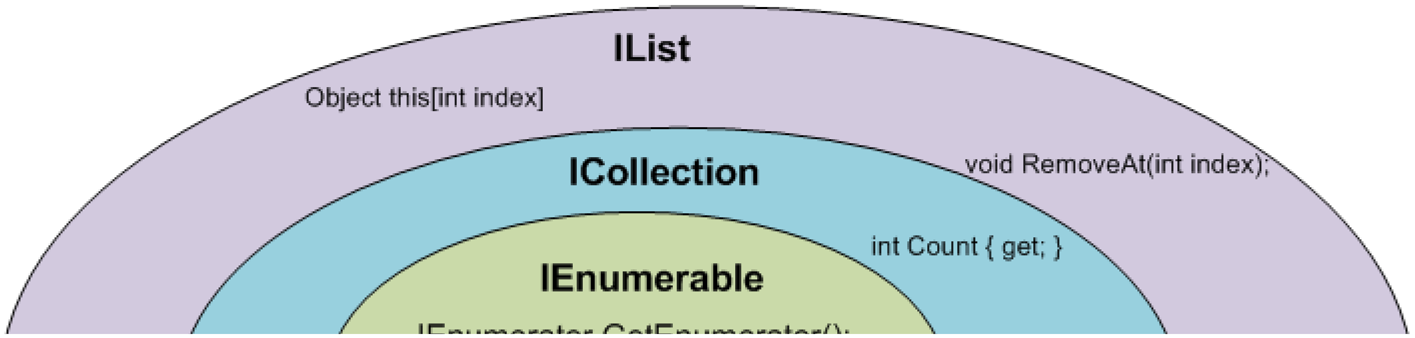
17 This answer is very misleading! @Nate's comment explains why. If you're using IEnumerable<T>, the filter is going to happen on the client side no matter what. – Hans Jan 26 '15 at 23:58

By using our site, you acknowledge that you have read and understand our Cookie Policy, Privacy Policy, and our Terms of Service.

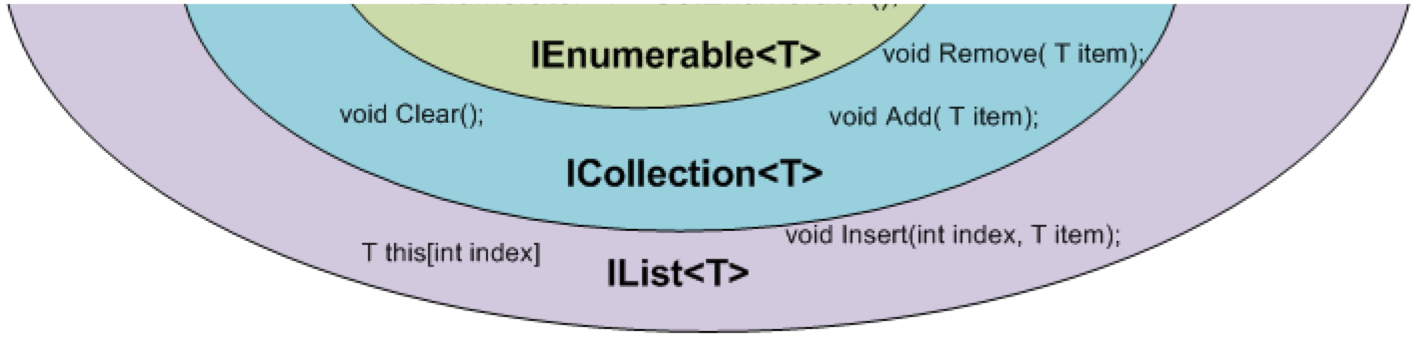
139

By using our site, you acknowledge that you have read and understand our Cookie Policy, Privacy Policy, and our Terms of Service.

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, you should never have a member of your own implementations with the concrete type List/List.



By using our site, you acknowledge that you have read and understand our Cookie Policy, Privacy Policy, and our Terms of Service.



edited Jun 4 '15 at 22:58

answered Jun 4 '15 at 14:07

rubStackOverflow
2,443 1 19 27

19 It should be pointed that this article is only for the public facing parts of your code, not the internal workings. `List` is an implementation of `IList` and as such has extra functionality on top of those in `IList` (e.g. `Sort` , `Find` , `InsertRange`). If you force yourself to use `IList` over `List` , you loose these methods that you may require – Jonathan Twite Mar 24 '16 at 9:07

2 Don't forget `IReadOnlyCollection<T>` – Dandré Jul 10 '16 at 7:53

124

A class that implement `IEnumerable` allows you to use the `foreach` syntax.

Basically it has a method to get the next item in the collection. It doesn't need the whole collection to be in memory and doesn't know how many items are in it, `foreach` just keeps getting the next item until it runs out.

This can be very useful in certain circumstances, for instance in a massive database table you don't want to copy the entire thing into memory before you start processing the rows.

Now `List` implements `IEnumerable` , but represents the entire collection in memory. If you have an `IEnumerable` and you call `.ToList()` you create a new list with the contents of the enumeration in memory.

Your linq expression returns an enumeration, and by default the expression executes when you iterate through using the `foreach` . An `IEnumerable` linq statement executes when you iterate the `foreach` , but you can force it to iterate sooner using `.ToList()` .

Here's what I mean:

By using our site, you acknowledge that you have read and understand our Cookie Policy, Privacy Policy, and our Terms of Service.

```
        where ....
        select item;

// this will iterate through the entire linq statement:
int count = things.Count();

// this will stop after iterating the first one, but will execute the linq again
bool hasAnyRecs = things.Any();

// this will execute the linq statement *again*
foreach( var thing in things ) ...

// this will copy the results to a list in memory
var list = things.ToList()

// this won't iterate through again, the list knows how many items are in it
int count2 = list.Count();

// this won't execute the linq statement - we have it copied to the list
foreach( var thing in list ) ...
```

answered Sep 2 '10 at 15:24
Keith
99.2k 62 246 364

2 But what happens if you execute a foreach on an IEnumerable **without converting it to a List first**? Does it bring the whole collection in memory? Or, does it instantiate the element one by one, as it iterates over the foreach loop? thanks – Pap Apr 3 '16 at 15:21

@Pap the latter: it executes again, nothing is automatically cached in memory. – Keith Apr 3 '16 at 15:34

seems like the key diff is 1) whole thing in memory or not. 2) IEnumerable let me use `foreach` while List will go by say index. Now, if I'd like to know the **count/length** of `thing` beforehand, IEnumerable won't help, right? – Jeb50 Sep 4 '18 at 23:49

@Jeb50 Not exactly - both `List` and `Array` implement `IEnumerable`. You can think of `IEnumerable` as a lowest common denominator that works for both in memory collections and large ones that get one item at a time. When you call `IEnumerable.Count()` you might be calling a fast `.Length` property or going through the whole collection - the point is that with `IEnumerable` you don't know. That can be a problem, but if you're just going to `foreach` it then you don't care - your code will work with an `Array` or `DataReader` the same. – Keith Sep 5 '18 at 11:01

1 @MFouadKajj I don't know what stack you're using, but it's almost certainly not making a request with each row. The server runs the query and calculates the starting point of the result set, but doesn't get the whole thing. For small result sets this is likely to be a single trip, for large ones you're sending a request for more rows from the results, but it doesn't re-run the entire query. – Keith Dec 3 '18 at 14:52

By using our site, you acknowledge that you have read and understand our Cookie Policy, Privacy Policy, and our Terms of Service.

78

> IEnumerable is read-only and List is not.

See [Practical difference between List and IEnumerable]()

edited May 23 '17 at 12:34
Community ♦
1 1

answered Dec 9 '14 at 8:30
CAD bloke
5,839 2 44 89

As a follow up, is that because of the Interface aspect or because of the List aspect? i.e. is IList also readonly? – Jason Masters Feb 26 at 2:20

IList is not read-only - [docs.microsoft.com/en-us/dotnet/api/…]() IEnumerable is read-only because it lacks any methods to add or remove anything once it is constructed, it is one of the base interfaces which IList extends (see link) – CAD bloke Feb 26 at 2:24

65

The most important thing to realize is that, using Linq, the query does not get evaluated immediately. It is only run as part of iterating through the resulting `IEnumerable<T>` in a `foreach` - that's what all the weird delegates are doing.

So, the first example evaluates the query immediately by calling `ToList` and putting the query results in a list.
The second example returns an `IEnumerable<T>` that contains all the information needed to run the query later on.

In terms of performance, the answer is *it depends*. If you need the results to be evaluated at once (say, you're mutating the structures you're querying later on, or if you don't want the iteration over the `IEnumerable<T>` to take a long time) use a list. Else use an `IEnumerable<T>`. The default should be to use the on-demand evaluation in the second example, as that generally uses less memory, unless there is a specific reason to store the results in a list.

edited Jan 9 '15 at 3:50
h4ck3rm1k3
1,790 19 31

answered Sep 2 '10 at 15:08
thecoop
36.3k 11 109 168

Hi and thanks for answering ::-). This cleared up almost all my doubts. Any idea why the Enumerable is "split" into "inner" and "outer"? This happens when I inspect the element in debug/break mode via mouse. Is this perhaps Visual Studio's contribution? Enumerating on the spot and indicating input and output of the Enum? – Axonn Sep 2 '10 at 17:10

5 That's the `Join` doing it's work - inner and outer are the two sides of the join. Generally, don't worry about what's actually in the `IEnumerables`, as it will be completely different from your actual code. Only worry about the actual output when you iterate over it :) – thecoop Sep 3 '10 at 10:24

By using our site, you acknowledge that you have read and understand our Cookie Policy, Privacy Policy, and our Terms of Service.

38

The advantage of IEnumerable is deferred execution (usually with databases). The query will not get executed until you actually loop through the data. It's a query waiting until it's needed (aka lazy loading).

If you call ToList, the query will be executed, or "materialized" as I like to say.

There are pros and cons to both. If you call ToList, you may remove some mystery as to when the query gets executed. If you stick to IEnumerable, you get the advantage that the program doesn't do any work until it's actually required.

answered Sep 2 '10 at 15:13
Matt Sherman
5,657 4 30 52

18

I will share one misused concept that I fell into one day:

```
var names = new List<string> {"mercedes", "mazda", "bmw", "fiat", "ferrari"};

var startingWith_M = names.Where(x => x.StartsWith("m"));

var startingWith_F = names.Where(x => x.StartsWith("f"));


// updating existing list
names[0] = "ford";

// Guess what should be printed before continuing
print( startingWith_M.ToList() );
print( startingWith_F.ToList() );
```

Expected result

```
// I was expecting
print( startingWith_M.ToList() ); // mercedes, mazda
print( startingWith_F.ToList() ); // fiat, ferrari
```

Actual result

By using our site, you acknowledge that you have read and understand our Cookie Policy, Privacy Policy, and our Terms of Service.

Explanation

As per other answers, the evaluation of the result was deferred until calling `ToList` or similar invocation methods for example `ToArray` .

So I can rewrite the code in this case as:

```
var names = new List<string> {"mercedes", "mazda", "bmw", "fiat", "ferrari"};

// updating existing list
names[0] = "ford";

// before calling ToList directly
var startingWith_M = names.Where(x => x.StartsWith("m"));

var startingWith_F = names.Where(x => x.StartsWith("f"));

print( startingWith_M.ToList() );
print( startingWith_F.ToList() );
```

Play arround

https://repl.it/E8Ki/0

edited May 23 '18 at 16:26
maxisam
19.1k 8 60 73

answered Nov 26 '16 at 8:03
amd
15.1k 5 37 55

1 That's because of linq methods (extension) which in this case come from IEnumerable where only create a query but not execute it (behind the scenes the expression trees are used). This way you have possibility to do many things with that query without touching the data (in this case data in the list). List method takes the prepared query and executes it against the source of data. – Bronek Dec 17 '16 at 23:54

1 Actually, I read all the answers, and yours was the one I up-voted, because it clearly states the difference between the two without specifically talking about LINQ/SQL. It is essential to know all this BEFORE you get to LINQ/SQL. Admire. – BeemerGuy Dec 28 '16 at 13:09

That is an important difference to explain but your "expected result" isn't really expected. You're saying it like it's some sort of gotcha rather than design. – Neme Jan 22 '17 at 0:14

@Neme, yes It was my expectation before I understand how `IEnumerable` works, but now Isn't more since I know how ;) – amd Jan 22 '17 at 7:54

By using our site, you acknowledge that you have read and understand our Cookie Policy, Privacy Policy, and our Terms of Service.

15

If all you want to do is enumerate them, use the `IEnumerable`.

Beware, though, that changing the original collection being enumerated is a dangerous operation - in this case, you will want to `ToList` first. This will create a new list element for each element in memory, enumerating the `IEnumerable` and is thus less performant if you only enumerate once - but safer and sometimes the `List` methods are handy (for instance in random access).

edited Sep 3 '10 at 14:38

answered Sep 2 '10 at 15:09
Daren Thomas
44.7k 37 136 184

1 I'm not sure it's safe to say that generating a list means lower performance. – Steven Sudit Sep 2 '10 at 15:24

@ Steven: indeed as thecoop and Chris said, sometimes it may be necessary to use a List. In my case, I have concluded it isn't. @ Daren: what do you mean by "this will create a new list for each element in memory"? Maybe you meant a "list entry"? ::-). – Axonn Sep 2 '10 at 17:09

@Axonn yes, I ment list entry. fixed. – Daren Thomas Sep 3 '10 at 6:30

@Steven If you plan to iterate over the elements in the `IEnumerable`, then creating a list first (and iterating over that) means you iterate over the elements *twice*. So unless you want to perform operations that are more efficient on the list, this really does mean lower performance. – Daren Thomas Sep 3 '10 at 6:31

3 @jerhewet: it is never a good idea to modify a sequence being iterated over. Bad things will happen. Abstractions will leak. Demons will break into our dimension and wreak havoc. So yes, `.ToList()` helps here ;) – Daren Thomas Jun 7 '11 at 7:21

5

In addition to all the answers posted above, here is my two cents. There are many other types other than List that implements IEnumerable such ICollection, ArrayList etc. So if we have IEnumerable as parameter of any method, we can pass any collection types to the function. Ie we can have method to operate on abstraction not any specific implementation.

edited Dec 22 '17 at 23:43

answered Jun 15 '17 at 23:22
Ananth
4,565 21 73 105

1

There are many cases (such as an infinite list or a very large list) where IEnumerable cannot be transformed to a List. The most obvious examples are all the prime numbers, all the users of facebook with their details, or all the items on ebay.

By using our site, you acknowledge that you have read and understand our Cookie Policy, Privacy Policy, and our Terms of Service.

surely run me out of memory, no matter how big my computer was. No computer can by itself contain and handle such a huge amount of data.

[Edit] - Needless to say - it's not "either this or that". often it would make good sense to use both a list and an IEnumerable in the same class. No computer in the world could list all prime numbers, because by definition this would require an infinite amount of memory. But you could easily think of a `class PrimeContainer` which contains an `IEnumerable<long> primes`, which for obvious reasons also contains a `SortedList<long> _primes`. all the primes calculated so far. the next prime to be checked would only be run against the existing primes (up to the square root). That way you gain both - primes one at a time (IEnumerable) and a good list of "primes so far", which is a pretty good approximation of the entire (infinite) list.

edited Jun 10 at 12:59

answered Mar 11 at 8:06

LongChalk
236 2 6

By using our site, you acknowledge that you have read and understand our Cookie Policy, Privacy Policy, and our Terms of Service.