How do you get the index of the current iteration of a foreach loop?

Ask Question

772

128

Is there some rare language construct I haven't encountered (like the few I've learned recently, some on Stack Overflow) in C# to get a value representing the current iteration of a foreach loop?

For instance, I currently do something like this depending on the circumstances:

```
int i = 0;
foreach (Object o in collection)
{
    // ...
    i++;
}
```

c# foreach

edited Mar 23 at 8:51

AustinWBryan

2,381 2 14 32

asked Sep 4 '08 at 1:38

Matt Mitchell
24.6k 32 102 172

1 foreach casting retrieval is generally not going to me more optimized than just using index-based access on a collection, though in many cases it will be equal. The purpose of foreach is to make your code readable, but it (usually) adds a layer of indirection, which isn't free. – Brian Mar 26 '10 at 15:32

8 I would say the primary purpose of `foreach` is to provide a common iteration mechanism for all collections regardless of whether they are indexable (`List`) or not (`Dictionary`). – Brian Gideon Jul 23 '10 at 15:59

2 Hi Brian Gideon - definitely agree (this was a few years ago and I was far less experienced at the time). However, while `Dictionary` isn't indexable, an iteration of `Dictionary` does traverse it in a particular order (i.e. an Enumerator is indexable by the fact it yields elements sequentially). In this sense, we could say that we are not looking for the index within the collection, but rather the index of the current enumerated element within the enumeration (i.e. whether we are at the first or fifth or last enumerated element). – Matt Mitchell May 3 '11 at 2:28

4 foreach also allows the compiler to skip bounds checking each array access in the compiled code. Using for with an index will make the runtime check whether your index access is safe. – IvoTops Aug 22 '12 at 9:00

1 But it's false. If you don't change the iteration variable of a for loop within the loop, the compiler knows what its bounds are and doesn't need to check them again. This is such a common case that any decent compiler will implement it. – Jim Balter Oct 26 '13 at 0:52

33 Answers

1 2 next

The `foreach` is for iterating over collections that implement `IEnumerable` . It does this by calling `GetEnumerator` on the collection, which will return an `Enumerator` .

488

This Enumerator has a method and a property:

- MoveNext()
- Current

`Current` returns the object that Enumerator is currently on, `MoveNext` updates `Current` to the next object.

The concept of an index is foreign to the concept of enumeration, and cannot be done.

Because of that, most collections are able to be traversed using an indexer and the for loop construct.

I greatly prefer using a for loop in this situation compared to tracking the index with a local variable.

edited Jan 21 at 12:11

Moriya
4,785 2 19 42

answered Sep 4 '08 at 1:46

FlySwat
116k 63 231 300

112 "Obviously, the concept of an index is foreign to the concept of enumeration, and cannot be done." -- This is nonsense, as the answers by David B and bcahill make clear. An index is an enumeration over a range, and there's no reason one cannot enumerate two things in parallel ... that's exactly what the indexing form of Enumerable.Select does. – Jim Balter Oct 26 '13 at 0:57

9 Basic code example: `for(var i = 0; i < myList.Count; i ++) {System.Diagnostics.Debug.WriteLine(i);}` – Chad Hedgcock Apr 14 '15 at 17:53

4 @Pretzel The statement which I quoted is (obviously) not correct, and I explained why. The fact Linked List "doesn't have an index" is completely irrelevant and shows extraordinary confusion. – Jim Balter Feb 2 '16 at 0:34

15 @JimBalter: That's the first link I read. I don't see how it supports your position. I also don't hurl ad-homs and divisive words at people when replying. I cite as an example your use of "nonsense", "extraordinary confusion", "completely wrong and confused", "I got 37 upvotes", etc. (I think your ego is a bit on the line here.) Instead, I'd like to politely ask that you not browbeat others with your 'argumentum ad verecundiam' and I'd like to instead encourage you to think of ways to support people here on StackOverflow. I'd say Jon Skeet is a model citizen here in that regard. – Pretzel Feb 2 '16 at 21:19

7 Pretzel: Your use of Jon Skeet as a model citizen is mistaken, as he will browbeat (and downvote) someone who doesn't agree with him, as I've experienced. Jim Balter: Some of your comments were overly aggressive and you could have presented your points with less anger and more education. – Suncat2000 Aug 12 '16 at 19:22

511

Ian Mercer posted a similar solution as this on Phil Haack's blog:

```
foreach (var item in Model.Select((value, i) => new { i, value }))
{
    var value = item.value;
    var index = item.i;
}
```

This gets you the item (`item.value`) and its index (`item.i`) by using this overload of Linq's `Select` :

> the second parameter of the function [inside Select] represents the index of the source element.

The `new { i, value }` is creating a new anonymous object.

Heap allocations can be avoided by using `ValueTuple` if you're using C# 7.0 or later:

```
foreach (var item in Model.Select((value, i) => ( value, i )))
{
    var value = item.value;
    var index = item.i;
}
```

You can also eliminate the `item.` by using automatic destructuring:

```
<ol>
foreach ((MyType value, Int32 i) in Model.Select((value, i) => ( val
{
    <li id="item_@i">@value</li>
}
</ol>
```

edited Jan 25 at 6:50

Dai
74.4k 13 120 207

answered Jul 11 '12 at 16:52

bcahill
5,290 1 11 10

7 That solution is nice for the case of a Razor template where the neatness of the template is a non-trivial design concern and you are also wanting to use the index of every item that is enumerated. However, do bear in mind that the object allocation from 'wrapping' adds expense (in space and time) on top of the (unavoidable) incrementing of an integer. – David Bullock Feb 25 '14 at 11:14

3 pure awesomeness, but where is that documented.... I mean two parameters in the Select delegate, I never saw that... and I can't find a documentation explanation..... experimenting with that expression I saw that value doesn't mean anything and only the position matters, first parameter the element, second parameter the index. – mjsr Jul 15 '15 at 20:25

6 @mjsr The documentation is here. – Thorkil Holm-Jacobsen Sep 15 '15 at 12:52

13 Sorry - that's clever, but is it really more readable than creating an index outside the foreach and incrementing it each loop? – jbyrd Oct 17 '17 at 20:13

5 With the later C# versions so you also use tuples, so you'll have something like this: foreach (var (item, i) in Model.Select((v, i) => (v, i))) Allows you to access the item and index (i) directly inside the for-loop with tuple deconstruction. – Haukman Jan 8 '18 at 22:25

0

i want to discuss this question more theoretically (since it has already enough practical answers)

.net has a very nice abstraction model for groups of data (a.k.a. collections)

- At the very top, and the most abstract, you have an `IEnumerable` it's just a group of data that you can enumerate. It doesn't matter HOW you enumerate, it's just that you can enumerate some data. And that enumeration is done by a completely different object, an `IEnumerator`

these interfaces are defined is as follows:

```
//
// Summary:
//     Exposes an enumerator, which supports a simple iteration over
collection.
public interface IEnumerable
{
    //
    // Summary:
    //     Returns an enumerator that iterates through a collection.
    //
    // Returns:
    //     An System.Collections.IEnumerator object that can be used
    //     the collection.
```

```
    IEnumerator GetEnumerator();
}

//
// Summary:
//     Supports a simple iteration over a non-generic collection.
public interface IEnumerator
{
    //
    // Summary:
    //     Gets the element in the collection at the current positio
    //
    // Returns:
    //     The element in the collection at the current position of
    object Current { get; }

    //
    // Summary:
    //     Advances the enumerator to the next element of the collec
    //
    // Returns:
    //     true if the enumerator was successfully advanced to the n
if
    //     the enumerator has passed the end of the collection.
    //
    // Exceptions:
    //   T:System.InvalidOperationException:
    //     The collection was modified after the enumerator was crea
    bool MoveNext();
    //
    // Summary:
    //     Sets the enumerator to its initial position, which is bef
element
    //     in the collection.
    //
    // Exceptions:
    //   T:System.InvalidOperationException:
    //     The collection was modified after the enumerator was crea
    void Reset();
}
```

- as you might have noticed, the `IEnumerator` interface doesn't "know" what an index is, it just knows what element it's currently pointing to, and how to move to the next one.

- now here is the trick: `foreach` considers every input collection an `IEnumerable`, even if it is a more concrete implementation like an `IList<T>` (which inherits from `IEnumerable`), it will only see the abstract interface `IEnumerable`.
- what `foreach` is actually doing, is calling `GetEnumerator` on the collection, and calling `MoveNext` until it returns false.
- so here is the problem, you want to define a concrete concept "Indices" on an abstract concept "Enumerables", the built in `foreach` construct doesn't give you that option, so your only way is to define it yourself, either by what you are doing originally (creating a counter manually) or just use an implementation of `IEnumerator` that recognizes indices AND implement a `foreach` construct that recognizes that custom implementation.

personally i would create an extension method like this

```
public static class Ext
{
    public static void FE<T>(this IEnumerable<T> l, Action<int, T> a
    {
        int counter = 0;
        foreach (var item in l)
        {
            act(counter, item);
            counter++;
        }
    }
}
```

and use it like this

```
var x = new List<string>() { "hello", "world" };
x.FE((ind, ele) =>
{
    Console.WriteLine($"{ind}: {ele}");
});
```

this also avoids any unnecessary allocations seen in other answers.

answered Dec 15 '18 at 2:12

bigworld12
593 6 26

8

Just add your own index. Keep it simple.

```
int i = 0;
foreach (var item in Collection)
{
    item.index = i;
    ++i;
}
```

answered May 21 '18 at 20:55

conterio
513 1 5 16

25

Using LINQ, C# 7, and the `System.ValueTuple` NuGet package, you can do this:

```
foreach (var (value, index) in collection.Select((v, i)=>(v, i))) {
    Console.WriteLine(value + " is at index " + index);
}
```

You can use the regular `foreach` construct and be able to access the value and index directly, not as a member of an object, and keeps both fields only in the scope of the loop. For these reasons, I believe this is the best solution if you are able to use C# 7 and `System.ValueTuple` .

edited Dec 18 '17 at 15:57

answered Sep 10 '17 at 20:56

Pavel
510 1 11 15

4

The leading answer states:

"Obviously, the concept of an index is foreign to the concept of enumeration, and cannot be done."

While this is true of the current C# version, this is not a conceptual limit.

The creation of a new C# language feature by MS could solve this, along with support for a new Interface IIndexedEnumerable

```
foreach (var item in collection with var index)
{
    Console.WriteLine("Iteration {0} has value {1}", index, item);
}

//or, building on @user1414213562's answer
foreach (var (item, index) in collection)
{
    Console.WriteLine("Iteration {0} has value {1}", index, item);
}
```

If foreach is passed an IEnumerable and can't resolve an IIndexedEnumerable, but it is asked with var index, then the C# compiler can wrap the source with an IndexedEnumerable object, which adds in the code for tracking the index.

```
interface IIndexedEnumerable<T> : IEnumerable<T>
{
    //Not index, because sometimes source IEnumerables are transient
```

```
    public long IterationNumber { get; }
}
```

Why:

- Foreach looks nicer, and in business applications is rarely a performance bottleneck
- Foreach can be more efficient on memory. Having a pipeline of functions instead of converting to new collections at each step. Who cares if it uses a few more CPU cycles, if there are less CPU cache faults and less GC.Collects
- Requiring the coder to add index tracking code, spoils the beauty
- It's quite easy to implement (thanks MS) and is backward compatible

While most people here are not MS, this is a correct answer, and you can lobby MS to add such a feature. You could already build your own iterator with an extension function and use tuples, but MS could sprinkle the syntactic sugar to avoid the extension function

edited Sep 20 '17 at 1:16

answered Nov 11 '16 at 4:11

Todd

10.4k 4 27 43

1 @Pavel I updated the answer to be clear. This answer was provided to counter the leading answer which states "Obviously, the concept of an index is foreign to the concept of enumeration, and cannot be done." – Todd Sep 20 '17 at 1:08

C# 7 finally gives us an elegant way to do this:

10

```
static class Extensions
{
    public static IEnumerable<(int, T)> Enumerate<T>(
        this IEnumerable<T> input,
        int start = 0
    )
    {
        int i = start;
        foreach (var t in input)
        {
            yield return (i++, t);
        }
    }
}

class Program
{
    static void Main(string[] args)
    {
        var s = new string[]
        {
            "Alpha",
            "Bravo",
            "Charlie",
            "Delta"
        };

        foreach (var (i, t) in s.Enumerate())
        {
            Console.WriteLine($"{i}: {t}");
        }
    }
}
```

answered Jul 21 '17 at 13:52

Paul Mitchell

2,815 14 20

76

Finally C#7 has a decent syntax for getting an index inside of a `foreach` loop (i. e. tuples):

```
foreach (var (item, index) in collection.WithIndex())
{
    Debug.WriteLine($"{index}: {item}");
}
```

A little extension method would be needed:

```
public static IEnumerable<(T item, int index)> WithIndex<T>(this IEn
   => self.Select((item, index) => (item, index));
```

edited Oct 19 '16 at 16:59

answered Oct 12 '16 at 11:12

user1414213562

1,178 8 9

2 This answer is underrated, having the tuples is much cleaner – Todd Sep 20 '17 at 1:15

1 Modified to handle null collections: `public static IEnumerable<(T item, int index)> WithIndex<T>(this IEnumerable<T> self) => self?.Select((item, index) => (item, index)) ?? new List<(T, int)>();` – 2Toad Nov 25 '18 at 5:33

3

You can write your loop like this:

```
var s = "ABCDEFG";
foreach (var item in s.GetEnumeratorWithIndex())
```

```
{
    System.Console.WriteLine("Character: {0}, Position: {1}", item.V
}
```

After adding the following struct and extension method.

The struct and extension method encapsulate Enumerable.Select functionality.

```
public struct ValueWithIndex<T>
{
    public readonly T Value;
    public readonly int Index;

    public ValueWithIndex(T value, int index)
    {
        this.Value = value;
        this.Index = index;
    }

    public static ValueWithIndex<T> Create(T value, int index)
    {
        return new ValueWithIndex<T>(value, index);
    }
}

public static class ExtensionMethods
{
    public static IEnumerable<ValueWithIndex<T>> GetEnumeratorWithIn
IEnumerable<T> enumerable)
    {
        return enumerable.Select(ValueWithIndex<T>.Create);
    }
}
```

answered Aug 23 '16 at 15:48

Satisfied
31 2

Why foreach ?!

7

The simplest way is using **for** instead of foreach **if you are using List** .

```
for(int i = 0 ; i < myList.Count ; i++)
{
    // Do Something...
}
```

OR if you want use foreach :

```
foreach (string m in myList)
{
     // Do something...
}
```

you can use this to khow index of each Loop :

```
myList.indexOf(m)
```

answered May 26 '16 at 21:41

Parsa
399 6 16

5 indexOf solution is invalid for list with duplicates and is also very slow. – tymtam Nov 17 '16 at 0:03

1 The issue to be avoided is where you traverse the IEnumerable multiple times, e.g. to get the count of items and then each item. This has implications when the IEnumerable is the result of a database query for example. – David Clarke Jan 12 '18 at 1:18

This doesn't answer your specific question, but it DOES provide you

1

with a solution to your problem: use a for loop to run through the object collection. then you will have the current index you are working on.

```
// Untested
for (int i = 0; i < collection.Count; i++)
{
    Console.WriteLine("My index is " + i);
}
```

edited Apr 2 '16 at 2:11

answered Nov 3 '15 at 21:00

BKSpurgeon

13k 4 53 48

5

If the collection is a list, you can use List.IndexOf, as in:

```
foreach (Object o in collection)
{
    // ...
    @collection.IndexOf(o)
}
```

edited Nov 4 '15 at 3:06

Andrew Medico

17.8k 8 47 70

answered Jan 5 '15 at 18:08

ssaeed

188 4 11

12 And now the algorithm is O(n^2) (if not worse). I would think *very* carefully before using this. Its also a duplicate of @crucible 's answer – BradleyDotNET Jan 5 '15 at 18:26

1 @BradleyDotNET is exactly right, don't use this version. – Oskar Jan 15 '16 at 11:16

1 Be careful with this! If you've got a duplicated item in your list, it will get the position of the first one! – Sonhja Jul 15 '16 at 9:00

22

There's nothing wrong with using a counter variable. In fact, whether you use `for`, `foreach` `while` or `do`, a counter variable must somewhere be declared and incremented.

So use this idiom if you're not sure if you have a suitably-indexed collection:

```
var i = 0;
foreach (var e in collection) {
   // Do stuff with 'e' and 'i'
   i++;
}
```

Else use this one if you *know* that your *indexable* collection is O(1) for index access (which it will be for `Array` and probably for `List<T>` (the documentation doesn't say), but not necessarily for other types (such as `LinkedList`)):

```
// Hope the JIT compiler optimises read of the 'Count' property!
for (var i = 0; i < collection.Count; i++) {
   var e = collection[i];
   // Do stuff with 'e' and 'i'
}
```

It should never be necessary to 'manually' operate the `IEnumerator` by invoking `MoveNext()` and interrogating `Current` - `foreach` is

saving you that particular bother ... if you need to skip items, just use a `continue` in the body of the loop.

And just for completeness, depending on what you were *doing* with your index (the above constructs offer plenty of flexibility), you might use Parallel LINQ:

```
// First, filter 'e' based on 'i',
// then apply an action to remaining 'e'
collection
    .AsParallel()
    .Where((e,i) => /* filter with e,i */)
    .ForAll(e => { /* use e, but don't modify it */ });

// Using 'e' and 'i', produce a new collection,
// where each element incorporates 'i'
collection
    .AsParallel()
    .Select((e, i) => new MyWrapper(e, i));
```

We use `AsParallel()` above, because it's 2014 already, and we want to make good use of those multiple cores to speed things up. Further, for 'sequential' LINQ, you only get a `ForEach()` extension method on `List<T>` and `Array` ... and it's not clear that using it is any better than doing a simple `foreach`, since you are still running single-threaded for uglier syntax.

edited May 23 '17 at 11:33

Community ♦

1 1

answered Feb 25 '14 at 14:07

David Bullock

3,973 2 27 39

I built this in LINQPad:

3

```
var listOfNames = new List<string>(){"John","Steve","Anna","Chris"};

var listCount = listOfNames.Count;

var NamesWithCommas = string.Empty;

foreach (var element in listOfNames)
{
    NamesWithCommas += element;
    if(listOfNames.IndexOf(element) != listCount -1)
    {
        NamesWithCommas += ", ";
    }
}

NamesWithCommas.Dump();  //LINQPad method to write to console.
```

You could also just use `string.join` :

```
var joinResult = string.Join(",", listOfNames);
```

edited Jul 28 '15 at 11:58

Peter Mortensen
13.9k 19 87 113

answered Oct 25 '13 at 16:14

Warren LaFrance
173 12

5 This is O(n*n). – Jim Balter Oct 26 '13 at 1:00

1 Can someone explain to me what's this O(n*n) thing ? – Axel Jan 9 '17 at 21:59

I don't think this should be quite efficient, but it works:

3

```
@foreach (var banner in Model.MainBanners) {
    @Model.MainBanners.IndexOf(banner)
}
```

edited Jul 28 '15 at 11:54

Peter Mortensen
13.9k 19 87 113

answered Mar 22 '13 at 13:52

Bart Calixto
12.8k 8 63 103

7 That presupposes the list does not have duplicates, of course... – Michael Sorens Mar 22 '13 at 14:32

8

This is how I do it, which is nice for its simplicity/brevity, but if you're doing a lot in the loop body `obj.Value` , it is going to get old pretty fast.

```
foreach(var obj in collection.Select((item, index) => new { Index = 
}) {
    string foo = string.Format("Something[{0}] = {1}", obj.Index, ob
    ...
}
```

edited Jul 28 '15 at 11:52

Peter Mortensen
13.9k 19 87 113

answered Aug 20 '10 at 22:45

Ian Henry
19.4k 4 43 58

57

Literal Answer -- warning, performance may not be as good as just using an `int` to track the index. At least it is better than using `IndexOf`.

You just need to use the indexing overload of Select to wrap each item in the collection with an anonymous object that knows the index. This can be done against anything that implements IEnumerable.

```
System.Collections.IEnumerable collection = Enumerable.Range(100, 10

foreach (var o in collection.OfType<object>().Select((x, i) => new {
{
    Console.WriteLine("{0} {1}", o.i, o.x);
}
```

edited Nov 27 '13 at 7:44

SteveC
6,705 19 72 128

answered Sep 16 '08 at 21:50

Amy B
88.6k 18 120 166

3 The only reason to use OfType<T>() instead of Cast<T>() is if some items in the enumeration might fail an explicit cast. For object, this will never be the case. – dahlbyk Jun 26 '09 at 0:09

11 Sure, except for the other reason to use OfType instead of Cast - which is that I never use Cast. – Amy B Jun 26 '09 at 2:02

Using @FlySwat's answer, I came up with this solution:

32

```
//var list = new List<int> { 1, 2, 3, 4, 5, 6 }; // Your sample collec

var listEnumerator = list.GetEnumerator(); // Get enumerator

for (var i = 0; listEnumerator.MoveNext() == true; i++)
{
  int currentItem = listEnumerator.Current; // Get current item.
  //Console.WriteLine("At index {0}, item is {1}", i, currentItem); /
with i and  currentItem
}
```

You get the enumerator using `GetEnumerator` and then you loop using a `for` loop. However, the trick is to make the loop's condition `listEnumerator.MoveNext() == true` .

Since the `MoveNext` method of an enumerator returns true if there is a next element and it can be accessed, making that the loop condition makes the loop stop when we run out of elements to iterate over.

answered Oct 7 '13 at 22:49

Gezim
3,917 9 43 70

9 There's no need to compare listEnumerator.MoveNext() == true. That's like asking the computer if true == true? :) Just say if listEnumerator.MoveNext() { } – Zesty Nov 7 '13 at 8:12

6 @Zesty, you're absolutely correct. I felt that it's more readable to add it in this case especially for people who aren't used to entering anything other than i < blahSize as a condition. – Gezim Nov 7 '13 at 17:36

1 You should dispose the enumerator. – Antonín Lejsek Dec 16 '17 at 21:02

Better to use keyword `continue` safe construction like this

5

```
int i=-1;
foreach (Object o in collection)
{
    ++i;
    //...
    continue; //<--- safe to call, index will be increased
    //...
}
```

edited Jul 26 '13 at 14:49

ardila

1,005 11 21

answered Aug 20 '10 at 22:30

user426810

51 1 1

0

Here is another solution to this problem, with a focus on keeping the syntax as close to a standard `foreach` as possible.

This sort of construct is useful if you are wanting to make your views look nice and clean in MVC. For example instead of writing this the usual way (which is hard to format nicely):

```
<%int i=0;
foreach (var review in Model.ReviewsList) { %>
   <div id="review_<%=i%>">
       <h3><%:review.Title%></h3>
   </div>
   <%i++;
} %>
```

You could instead write this:

```
<%foreach (var review in Model.ReviewsList.WithIndex()) { %>
   <div id="review_<%=LoopHelper.Index()%>">
```

```
        <h3><%:review.Title%></h3>
    </div>
<%} %>
```

I've written some helper methods to enable this:

```
public static class LoopHelper {
    public static int Index() {
        return (int)HttpContext.Current.Items["LoopHelper_Index"];
    }
}

public static class LoopHelperExtensions {
    public static IEnumerable<T> WithIndex<T>(this IEnumerable<T> th
        return new EnumerableWithIndex<T>(that);
    }

    public class EnumerableWithIndex<T> : IEnumerable<T> {
        public IEnumerable<T> Enumerable;

        public EnumerableWithIndex(IEnumerable<T> enumerable) {
            Enumerable = enumerable;
        }

        public IEnumerator<T> GetEnumerator() {
            for (int i = 0; i < Enumerable.Count(); i++) {
                HttpContext.Current.Items["LoopHelper_Index"] = i;
                yield return Enumerable.ElementAt(i);
            }
        }

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

In a non-web environment you could use a `static` instead of `HttpContext.Current.Items` .

This is essentially a global variable, and so you cannot have more than one WithIndex loop nested, but that is not a major problem in this use case.

answered Apr 8 '13 at 22:05

mike nelson
12.4k 9 47 54

80

I disagree with comments that a `for` loop is a better choice in most cases.

`foreach` is a useful construct, and not replaceble by a `for` loop in all circumstances.

For example, if you have a **DataReader** and loop through all records using a `foreach` it automatically calls the **Dispose** method and closes the reader (which can then close the connection automatically). This is therefore safer as it prevents connection leaks even if you forget to close the reader.

(Sure it is good practise to always close readers but the compiler is not going to catch it if you don't - you can't guarantee you have closed all readers but you can make it more likely you won't leak connections by getting in the habit of using foreach.)

There may be other examples of the implicit call of the `Dispose` method being useful.

edited Jul 25 '12 at 10:07

SteveC
6,705 19 72 128

answered Jun 25 '09 at 23:49

mike nelson
12.4k 9 47 54

---

2 Thanks for pointing this out. Rather subtle. You can get more information at pvle.be/2010/05/foreach-statement-calls-dispose-on-ienumerator and

msdn.microsoft.com/en-us/library/aa664754(VS.71).aspx. – Mark Meuer Jun 17 '11 at 14:22

2

I wasn't sure what you were trying to do with the index information based on the question. However, in C#, you can usually adapt the IEnumerable.Select method to get the index out of whatever you want. For instance, I might use something like this for whether a value is odd or even.

```
string[] names = { "one", "two", "three" };
var oddOrEvenByName = names
    .Select((name, index) => new KeyValuePair<string, int>(name, ind
    .ToDictionary(kvp => kvp.Key, kvp => kvp.Value);
```

This would give you a dictionary by name of whether the item was odd (1) or even (0) in the list.

answered Sep 14 '11 at 19:03

Kasey Speakman
2,483 24 36

3

For interest, Phil Haack just wrote an example of this in the context of a Razor Templated Delegate (http://haacked.com/archive/2011/04/14/a-better-razor-foreach-loop.aspx)

Effectively he writes an extension method which wraps the iteration in an "IteratedItem" class (see below) allowing access to the index as well as the element during iteration.

```
public class IndexedItem<TModel> {
  public IndexedItem(int index, TModel item) {
    Index = index;
```

```
    Item = item;
  }

  public int Index { get; private set; }
  public TModel Item { get; private set; }
}
```

However, while this would be fine in a non-Razor environment if you are doing a single operation (i.e. one that could be provided as a lambda) it's not going to be a solid replacement of the for/foreach syntax in non-Razor contexts.

answered Apr 18 '11 at 23:25

Matt Mitchell

24.6k 32 102 172

2

I just had this problem, but thinking around the problem in my case gave the best solution, unrelated to the expected solution.

It could be quite a common case, basically, I'm reading from one source list and creating objects based on them in a destination list, however, I have to check whether the source items are valid first and want to return the row of any error. At first-glance, I want to get the index into the enumerator of the object at the Current property, however, as I am copying these elements, I implicitly know the current index anyway from the current destination. Obviously it depends on your destination object, but for me it was a List, and most likely it will implement ICollection.

i.e.

```
var destinationList = new List<someObject>();
foreach (var item in itemList)
{
  var stringArray = item.Split(new char[] { ';', ',' },
StringSplitOptions.RemoveEmptyEntries);
```

```
  if (stringArray.Length != 2)
  {
    //use the destinationList Count property to give us the index in
list
    throw new Exception("Item at row " + (destinationList.Count + 1)
problem.");
  }
  else
  {
    destinationList.Add(new someObject() { Prop1 = stringArray[0], P
stringArray[1]});
  }
}
```

Not always applicable, but often enough to be worth mentioning, I think.

Anyway, the point being that sometimes there is a non-obvious solution already in the logic you have...

edited Mar 23 '11 at 12:34

answered Mar 23 '11 at 12:19

nicodemus13

1,563 1 11 27

2

How about something like this? Note that myDelimitedString may be null if myEnumerable is empty.

```
IEnumerator enumerator = myEnumerable.GetEnumerator();
string myDelimitedString;
string current = null;

if( enumerator.MoveNext() )
    current = (string)enumerator.Current;
```

```
while( null != current)
{
    current = (string)enumerator.Current; }

    myDelimitedString += current;

    if( enumerator.MoveNext() )
        myDelimitedString += DELIMITER;
    else
        break;
}
```

edited Oct 29 '10 at 19:27

answered Oct 29 '10 at 19:00

Matt Towers

21 2

3

My solution for this problem is an extension method `WithIndex()` ,

http://code.google.com/p/ub-dotnet-utilities/source/browse/trunk/Src/Utilities/Extensions/EnumerableExtensions.cs

Use it like

```
var list = new List<int> { 1, 2, 3, 4, 5, 6 };

var odd = list.WithIndex().Where(i => (i.Item & 1) == 1);
CollectionAssert.AreEqual(new[] { 0, 2, 4 }, odd.Select(i => i.Index
CollectionAssert.AreEqual(new[] { 1, 3, 5 }, odd.Select(i => i.Item)
```

edited Sep 14 '10 at 21:10

answered Sep 11 '10 at 14:11

ulrichb
14.9k 6 62 80

24

You could wrap the original enumerator with another that does contain the index information.

```
foreach (var item in ForEachHelper.WithIndex(collection))
{
    Console.Write("Index=" + item.Index);
    Console.Write(";Value= " + item.Value);
    Console.Write(";IsLast=" + item.IsLast);
    Console.WriteLine();
}
```

Here is the code for the `ForEachHelper` class.

```
public static class ForEachHelper
{
    public sealed class Item<T>
    {
        public int Index { get; set; }
        public T Value { get; set; }
        public bool IsLast { get; set; }
    }

    public static IEnumerable<Item<T>> WithIndex<T>(IEnumerable<T> e
    {
        Item<T> item = null;
        foreach (T value in enumerable)
        {
            Item<T> next = new Item<T>();
            next.Index = 0;
            next.Value = value;
            next.IsLast = false;
            if (item != null)
            {
```

```
                next.Index = item.Index + 1;
                yield return item;
            }
            item = next;
        }
        if (item != null)
        {
            item.IsLast = true;
            yield return item;
        }
    }
}
```

edited Jul 23 '10 at 13:02

answered Jul 20 '10 at 19:15

Brian Gideon
40.6k 10 86 134

7 @Lucas: No, but it will return the index of the current foreach iteration. That was the question. – Brian Gideon Jul 23 '10 at 13:32

11

```
int index;
foreach (Object o in collection)
{
    index = collection.indexOf(o);
}
```

This would work for collections supporting `IList` .

edited Jan 2 '10 at 3:50

sth
169k 42 249 336

answered Dec 31 '09 at 11:49

Sachin
832 6 7

62 Two problems: 1) This is `O(n^2)` since in most implementations `IndexOf` is `O(n)` . 2) This fails if there are duplicate items in the list. – CodesInChaos Sep 14 '11 at 19:08

13 Note: O(n^2) means this could be disastrously slow for a big collection. – O'Rooney Nov 14 '11 at 0:58

17 God, I hope you did not use that! :(It DOES use that variable you did't want to create - in fact, it will create n+1 ints because that function has to create one in order to return, too, - and that indexof search is much, much slower than one integer increment operation in every step. Why won't people vote this answer down? – canahari May 19 '13 at 22:42

11 Don't use this answer, I found the hard truth mentioned in one of the comments. "This fails if there are duplicate items in the list."!!! – Bruce May 23 '14 at 7:32

17

Here's a solution I just came up with for this problem

Original code:

```
int index=0;
foreach (var item in enumerable)
{
    blah(item, index); // some code that depends on the index
    index++;
}
```

Updated code

```
enumerable.ForEach((item, index) => blah(item, index));
```

Extension Method:

```
    public static IEnumerable<T> ForEach<T>(this IEnumerable<T> enum
int> action)
    {
        var unit = new Unit(); // unit is a new type from the reacti
(http://msdn.microsoft.com/en-us/devlabs/ee794896.aspx) to represent
you can't return a void
        enumerable.Select((item, i) =>
            {
                action(item, i);
                return unit;
            }).ToList();

        return pSource;
    }
```

edited Dec 31 '09 at 11:31

answered Dec 31 '09 at 5:26

mat3

699 6 12

10

It's only going to work for a List and not any IEnumerable, but in LINQ there's this:

```
IList<Object> collection = new List<Object> {
    new Object(),
    new Object(),
    new Object(),
    };

foreach (Object o in collection)
{
    Console.WriteLine(collection.IndexOf(o));
}

Console.ReadLine();
```

@Jonathan I didn't say it was a great answer, I just said it was just showing it was possible to do what he asked :)

@Graphain I wouldn't expect it to be fast - I'm not entirely sure how it works, it could reiterate through the entire list each time to find a matching object, which would be a helluvalot of compares.

That said, List might keep an index of each object along with the count.

Jonathan seems to have a better idea, if he would elaborate?

It would be better to just keep a count of where you're up to in the foreach though, simpler, and more adaptable.

edited Sep 4 '08 at 2:51

answered Sep 4 '08 at 2:39

crucible
2,104 2 20 33

4 Not sure on the heavy downvoting. Sure performance makes this prohibitive but you did answer the question! – Matt Mitchell Oct 2 '09 at 21:15

2 Another issue with this is that it only works if the items in the list are unique. – CodesInChaos Sep 14 '11 at 19:12

1 2 next

protected by AZ_ Dec 7 '17 at 10:54

Thank you for your interest in this question. Because it has attracted low-quality or spam answers that had to be removed, posting an answer now

requires 10 reputation on this site (the association bonus does not count).

Would you like to answer one of these unanswered questions instead?