LINQ's Distinct() on a particular property

Ask Question

903

274

I am playing with LINQ to learn about it, but I can't figure out how to use Distinct when I do not have a simple list (a simple list of integers is pretty easy to do, this is not the question). What I if want to use Distinct on a list of an Object on *one* or *more* properties of the object?

Example: If an object is `Person`, with Property `Id`. How can I get all Person and use `Distinct` on them with the property `Id` of the object?

```
Person1: Id=1, Name="Test1"
Person2: Id=1, Name="Test1"
Person3: Id=2, Name="Test2"
```

How can I get just Person1 and Person3? Is that possible?

If it's not possible with LINQ, what would be the best way to have a list of `Person` depending on some of its properties in .NET 3.5?

c# linq .net-3.5 distinct

edited Jan 16 '16 at 18:34
Peter Mortensen
13.9k 19 87 113

asked Jan 28 '09 at 20:45
Patrick Desjardins
89.1k 77 271 327

Home

PUBLIC

Stack Overflow

Tags

Users

Jobs

Teams
Q&A for work

Learn More

20 Answers

1064

EDIT: This is now part of MoreLINQ.

What you need is a "distinct-by" effectively. I don't believe it's part of LINQ as it stands, although it's fairly easy to write:

```
public static IEnumerable<TSource> DistinctBy<TSource, TKey>
    (this IEnumerable<TSource> source, Func<TSource, TKey> keySelec
{
    HashSet<TKey> seenKeys = new HashSet<TKey>();
    foreach (TSource element in source)
    {
        if (seenKeys.Add(keySelector(element)))
        {
            yield return element;
        }
    }
}
```

So to find the distinct values using just the `Id` property, you could use:

```
var query = people.DistinctBy(p => p.Id);
```

And to use multiple properties, you can use anonymous types, which implement equality appropriately:

```
var query = people.DistinctBy(p => new { p.Id, p.Name });
```

Untested, but it should work (and it now at least compiles).

It assumes the default comparer for the keys though - if you want to pass in an equality comparer, just pass it on to the `HashSet` constructor.

edited Nov 4 '15 at 12:25
Thijs
1,788 4 21 48

answered Jan 28 '09 at 21:17
Jon Skeet
1098k 697 7996 8477

7 Source to DistinctBy: code.google.com/p/morelinq/source/browse/MoreLinq/DistinctBy.cs – Contango Feb 6 '13 at 19:46

1 @ashes999: I'm not sure what you mean. The code is present in the answer *and* in the library - depending on whether you're happy to take on a dependency. – Jon Skeet Feb 19 '13 at 17:07

6 @ashes999: If you're only doing this in a single place, ever, then sure, using `GroupBy` is simpler. If you need it in more than one place, it's much cleaner (IMO) to encapsulate the intention. – Jon Skeet Feb 19 '13 at 17:29

4 @MatthewWhited: Given that there's no mention of `IQueryable<T>` here, I don't see how it's relevant. I agree that this wouldn't be suitable for EF etc, but within LINQ to Objects I think it's *more* suitable than `GroupBy`. The context of the question is always important. – Jon Skeet Jan 22 '17 at 17:10

2 The project moved on github, here's the code of DistinctBy: github.com/morelinq/MoreLINQ/blob/master/MoreLinq/DistinctBy.cs – Phate01 Mar 10 '18 at 12:24

0

You can use DistinctBy() for getting Distinct records by an object property. Just add the following statement before using it:

> using Microsoft.Ajax.Utilities;

and then use it like following:

```
var listToReturn = responseList.DistinctBy(x => x.Index).ToList();
```

where 'Index' is the property on which i want the data to be distinct.

answered Mar 27 at 6:04

Harry .Naeem
528 3 5 23

55

Use:

```
List<Person> pList = new List<Person>();
/* Fill list */

var result = pList.Where(p => p.Name != null).GroupBy(p => p.Id).Sel
grp.FirstOrDefault());
```

The `where` helps you filter the entries (could be more complex) and the `groupby` and `select` perform the distinct function.

edited Dec 31 '18 at 2:33

SteveCav
5,658 38 51

answered Feb 14 '12 at 12:52
karcsi
653 5 5

Perfect, and works without extending Linq or using another dependency. – DavidScherer Mar 11 at 13:19

1

Override **Equals(object obj)** and **GetHashCode()** methods:

```
class Person
{
    public int Id { get; set; }
    public int Name { get; set; }

    public override bool Equals(object obj)
    {
        return ((Person)obj).Id == Id;
        // or:
        // var o = (Person)obj;
        // return o.Id == Id && o.Name == Name;
    }
    public override int GetHashCode()
    {
        return Id.GetHashCode();
    }
}
```

and then just call:

```
List<Person> distinctList = new[] { person1, person2, person3 }.Dist
```

answered Sep 27 '18 at 20:31
Waldemar Gałęzinowski

412 5 13

However GetHashCode() should be more advanced (to count also the Name), this answer is probably best by my opinion. Actually, to archive the target logic, there no need to override the GetHashCode(), Equals() is enough, but if we need performance, we have to override it. All comparison algs, first check hash, and if they are equal then call Equals(). – Oleg Skripnyak Oct 28 '18 at 5:30

Also, there in Equals() the first line should be "if (!(obj is Person)) return false". But best practice is to use separate object casted to a type, like "var o = obj as Person;if (o==null) return false;" then check equality with o without casting – Oleg Skripnyak Oct 28 '18 at 5:48

-3

Please give a try with below code.

```
var Item = GetAll().GroupBy(x => x .Id).ToList();
```

edited Jul 16 '18 at 7:24
Alien
5,523 3 11 28

answered Jul 16 '18 at 5:26
Mohamed Hammam
1 1

1 A short answer is welcome, however it won't provide much value to the latter users who are trying to understand what's going on behind the problem. Please spare some time to explain what's the real issue to cause the problem and how to solve. Thank you ~ – Hearen Jul 16 '18 at 5:46

30

Solution first group by your fields then select firstordefault item.

```
 List<Person> distinctPeople = allPeople
.GroupBy(p => p.PersonId)
.Select(g => g.FirstOrDefault())
.ToList();
```

answered Jul 13 '17 at 8:33

cahit beyaz
2,880 1 18 20

70

You could also use query syntax if you want it to look all LINQ-like:

```
var uniquePeople = from p in people
                   group p by new {p.ID} //or group by new {p.ID, p.I
                   into mygroup
                   select mygroup.FirstOrDefault();
```

edited Jan 20 '17 at 19:31

burnttoast11
774 14 31

answered Mar 6 '12 at 18:28

Chuck Rostance
5,169 1 19 16

3 Hmm my thoughts are both the query syntax and the fluent API syntax are just as LINQ like as each other and its just preference over which ones people use. I myself prefer the fluent API so I would consider that more LINK-Like but then I guess that's subjective – Max Carroll Jan 5 '18 at 15:57

LINQ-Like has nothing to do with preference, being "LINQ-like" has to do with looking like a different query language being embedded into C#, I prefer the fluent interface, coming from java streams, but it is NOT LINQ-Like. – Ryan The Leach Oct 2 '18 at 4:11

3

If you don't want to add the MoreLinq library to your project just to get the `DistinctBy` functionality then you can get the same end result using the overload of Linq's `Distinct` method that takes in an `IEqualityComparer` argument.

You begin by creating a generic custom equality comparer class that uses lambda syntax to perform custom comparison of two instances of a generic class:

```
public class CustomEqualityComparer<T> : IEqualityComparer<T>
{
    Func<T, T, bool> _comparison;
    Func<T, int> _hashCodeFactory;

    public CustomEqualityComparer(Func<T, T, bool> comparison, Func<
hashCodeFactory)
    {
        _comparison = comparison;
        _hashCodeFactory = hashCodeFactory;
    }

    public bool Equals(T x, T y)
    {
        return _comparison(x, y);
    }

    public int GetHashCode(T obj)
    {
        return _hashCodeFactory(obj);
    }
}
```

Then in your main code you use it like so:

```
Func<Person, Person, bool> areEqual = (p1, p2) => int.Equals(p1.Id, 

Func<Person, int> getHashCode = (p) => p.Id.GetHashCode();

var query = people.Distinct(new CustomEqualityComparer<Person>(areEq
```

Voila! :)

The above assumes the following:

- Property `Person.Id` is of type `int`
- The `people` collection does not contain any null elements

If the collection could contain nulls then simply rewrite the lambdas to check for null, e.g.:

```
Func<Person, Person, bool> areEqual = (p1, p2) =>
{
    return (p1 != null && p2 != null) ? int.Equals(p1.Id, p2.Id) : f
};
```

EDIT

This approach is similar to the one in Vladimir Nesterovsky's answer but simpler.

It is also similar to the one in Joel's answer but allows for complex comparison logic involving multiple properties.

However, if your objects can only ever differ by `Id` then another user gave the correct answer that all you need to do is override the default implementations of `GetHashCode()` and `Equals()` in your `Person` class and then just use the out-of-the-box `Distinct()` method of Linq to filter out any duplicates.

edited Aug 22 '16 at 18:03

answered Aug 22 '16 at 17:45

Caspian Canuck
1,061 1 10 17

2

```
List<Person>lst=new List<Person>
        var result1 = lst.OrderByDescending(a => a.ID).Select(a =>ne
{ID=a.ID,Name=a.Name} ).Distinct();
```

answered May 16 '16 at 10:42

Arindam
169 1 5

1611

> What if I want to obtain a distinct list based on *one* or *more* properties?

Simple! You want to group them and pick a winner out of the group.

```
List<Person> distinctPeople = allPeople
  .GroupBy(p => p.PersonId)
  .Select(g => g.First())
  .ToList();
```

If you want to define groups on multiple properties, here's how:

```
List<Person> distinctPeople = allPeople
  .GroupBy(p => new {p.PersonId, p.FavoriteColor} )
  .Select(g => g.First())
  .ToList();
```

edited Jan 17 '16 at 15:33

answered Jan 29 '09 at 14:39

Amy B
88.6k 18 120 166

@ErenErsonmez sure. With my posted code, if deferred execution is desired, leave off the ToList call. – Amy B Jan 17 '12 at 12:34

5 Very nice answer! Realllllly helped me in Linq-to-Entities driven from a sql view where I couldn't modify the view. I needed to use FirstOrDefault() rather than First() - all is good. – Alex KeySmith May 16 '12 at 14:14

6 I tried it and it should change to Select(g => g.FirstOrDefault()) – user585440 Jan 6 '16 at 23:38

1 @DavidB you might want to make a note that for LinqToEntities it should be FirstOrDefault instead. First is not supported. :) – Johny Skovdal Aug 29 '16 at 16:48

15 @ChocapicSz Nope. Both `Single()` and `SingleOrDefault()` each throw when the source has more than one item. In this operation, we expect the possibility that each group may have more then one item. For that matter, `First()` is preferred over `FirstOrDefault()` because each group must have at least one member.... unless you're using EntityFramework, which can't figure out that each group has at least one member and demands `FirstOrDefault()` . – Amy B Jul 17 '17 at 13:41

24

You can do this with the standard `Linq.ToLookup()` . This will create a collection of values for each unique key. Just select the first item in the collection

```
Persons.ToLookup(p => p.Id).Select(coll => coll.First());
```

edited Jan 16 '16 at 19:07

Peter Mortensen
13.9k 19 87 113

answered Jan 20 '15 at 15:01

David Fahlander
1,934 1 9 10

Good one! Pretty neat and simple solution that is barely even "hacky". :-) – Jonas Jan 9 at 20:27

Much better answer and neat – abdul qayyum Feb 14 at 10:32

3

When we faced such a task in our project we defined a small API to compose comparators.

So, the use case was like this:

```
var wordComparer = KeyEqualityComparer.Null<Word>().
    ThenBy(item => item.Text).
    ThenBy(item => item.LangID);
...
source.Select(...).Distinct(wordComparer);
```

And API itself looks like this:

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

public static class KeyEqualityComparer
{
    public static IEqualityComparer<T> Null<T>()
    {
        return null;
    }

    public static IEqualityComparer<T> EqualityComparerBy<T, K>(
        this IEnumerable<T> source,
        Func<T, K> keyFunc)
    {
        return new KeyEqualityComparer<T, K>(keyFunc);
    }

    public static KeyEqualityComparer<T, K> ThenBy<T, K>(
        this IEqualityComparer<T> equalityComparer,
        Func<T, K> keyFunc)
    {
        return new KeyEqualityComparer<T, K>(keyFunc, equalityCompar
    }
}

public struct KeyEqualityComparer<T, K>: IEqualityComparer<T>
{
    public KeyEqualityComparer(
        Func<T, K> keyFunc,
        IEqualityComparer<T> equalityComparer = null)
    {
        KeyFunc = keyFunc;
        EqualityComparer = equalityComparer;
    }

    public bool Equals(T x, T y)
    {
        return ((EqualityComparer == null) || EqualityComparer.Equal
                EqualityComparer<K>.Default.Equals(KeyFunc(x), KeyFu
    }

    public int GetHashCode(T obj)
    {
        var hash = EqualityComparer<K>.Default.GetHashCode(KeyFunc(o

        if (EqualityComparer != null)
        {
            var hash2 = EqualityComparer.GetHashCode(obj);

            hash ^= (hash2 << 5) + hash2;
        }

        return hash;
    }

    public readonly Func<T, K> KeyFunc;
    public readonly IEqualityComparer<T> EqualityComparer;
}
```

More details is on our site: *IEqualityComparer in LINQ*.

edited Jan 16 '16 at 18:55
Peter Mortensen
13.9k 19 87 113

answered Jul 10 '14 at 21:00
Vladimir Nesterovsky
412 5 11

2

The best way to do this that will be compatible with other .NET versions is to override Equals and GetHash to handle this (see Stack Overflow question *This code returns distinct values. However, what I want is to return a strongly typed collection as opposed to an anonymous type*), but if you need something that is generic throughout your code, the solutions in this article are great.

edited May 23 '17 at 12:02
Community ♦
1 1

answered Oct 21 '13 at 0:47
gcoleman0828
858 3 27 47

5

In case you need a Distinct method on multiple properties, you can check out my PowerfulExtensions library. Currently it's in a very young stage, but already you can use methods like Distinct, Union, Intersect, Except on any number of properties;

This is how you use it:

```
using PowerfulExtensions.Linq;
...
var distinct = myArray.Distinct(x => x.A, x => x.B);
```

edited Jan 16 '16 at 18:51

Peter Mortensen
13.9k 19 87 113

answered Aug 15 '13 at 20:20

Andrzej Gis
6,953 10 56 98

16

The following code is functionally equivalent to [Jon Skeet's answer](#).

Tested on .NET 4.5, should work on any earlier version of LINQ.

```
public static IEnumerable<TSource> DistinctBy<TSource, TKey>(
  this IEnumerable<TSource> source, Func<TSource, TKey> keySelector)
{
  HashSet<TKey> seenKeys = new HashSet<TKey>();
  return source.Where(element => seenKeys.Add(keySelector(element)))
}
```

Incidentially, check out [Jon Skeet's latest version of DistinctBy.cs on Google Code](#).

edited May 23 '17 at 12:10

Community ♦
1 1

answered Feb 6 '13 at 19:56

Contango
41.7k 47 190 238

3 This gave me a "sequence has no values error", but Skeet's answer produced the correct result. – What Would Be Cool Apr 22 '14 at 23:42

11

I've written an article that explains how to extend the Distinct function so that you can do as follows:

```
var people = new List<Person>();

people.Add(new Person(1, "a", "b"));
people.Add(new Person(2, "c", "d"));
people.Add(new Person(1, "a", "b"));

foreach (var person in people.Distinct(p => p.ID))
    // Do stuff with unique list here.
```

Here's the article: *Extending LINQ - Specifying a Property in the Distinct Function*

edited Jan 16 '16 at 18:44
Peter Mortensen
13.9k 19 87 113

answered Mar 11 '09 at 12:21
Timothy Khouri
21.9k 16 73 120

3 Your article has an error, there should be a <T> after Distinct: public static IEnumerable<T> Distinct(this... Also it does not look like it will work (nicely) on more that one property i.e. a combination of first and last names. – row1 Mar 17 '10 at 10:01

2 +1, a minor error is not a reason enough for downvote, that just so silly, callled a typo often. And I'm yet to see a generic function that will work for any number of property! I hope the downvoter has downvoted every other answer in this thread as well. But hey what is this second type being object?? I object ! – nawfal Nov 22 '12 at 12:08

3

Personally I use the following class:

```
public class LambdaEqualityComparer<TSource, TDest> :
    IEqualityComparer<TSource>
{
    private Func<TSource, TDest> _selector;

    public LambdaEqualityComparer(Func<TSource, TDest> selector)
    {
        _selector = selector;
    }

    public bool Equals(TSource obj, TSource other)
    {
        return _selector(obj).Equals(_selector(other));
    }

    public int GetHashCode(TSource obj)
    {
        return _selector(obj).GetHashCode();
    }
}
```

Then, an extension method:

```
public static IEnumerable<TSource> Distinct<TSource, TCompare>(
    this IEnumerable<TSource> source, Func<TSource, TCompare> select
{
    return source.Distinct(new LambdaEqualityComparer<TSource, TComp
}
```

Finally, the intended usage:

```
var dates = new List<DateTime>() { /* ... */ }
var distinctYears = dates.Distinct(date => date.Year);
```

The advantage I found using this approach is the re-usage of `LambdaEqualityComparer` class for other methods that accept an `IEqualityComparer` . (Oh, and I leave the `yield` stuff to the original LINQ implementation...)

answered Oct 30 '15 at 18:59
Joel
5,209 4 38 52

55

I think it is enough:

```
list.Select(s => s.MyField).Distinct();
```

answered Jan 23 '15 at 14:54
Ivan
1,325 13 7

31 What if he needs back his full object, not just that particular field? – Festim Cahani Aug 12 '15 at 13:44

1 What exactly object of the several objects that have the same property value? – donRumatta Sep 3 '15 at 10:45

0

You should be able to override Equals on person to actually do Equals on Person.id. This ought to result in the behavior you're after.

answered Jan 28 '09 at 20:49
GWLlosa
15.2k 16 66 101

5

You can do it (albeit not lightning-quickly) like so:

```
people.Where(p => !people.Any(q => (p != q && p.Id == q.Id)));
```

That is, "select all people where there isn't another different person in the list with the same ID."

Mind you, in your example, that would just select person 3. I'm not sure how to tell which you want, out of the previous two.

answered Jan 28 '09 at 20:47

mquander
55k 13 86 119