

The results are in! See what nearly 90,000 developers picked as their most loved, dreaded, and desired coding languages and more in the 2019 Developer Survey.

Fast way to convert a two dimensional array to a List (one dimensional)

[Ask Question](#)

▲
23
▼

I have a two dimensional array and I need to convert it to a List (same object). I don't want to do it with `for` or `foreach` loop that will take each element and add it to the List. Is there some other way to do it?

★

12

c#

arrays

list

type-conversion

edited Sep 17 '18 at 14:06



[Micha Wiedenmann](#)

10.6k 13 64 106

asked Feb 27 '11 at 9:26



[Yanshof](#)

4,572 12 68 141

1 What is that list supposed to contain? – [BoltClock](#) ♦ Feb 27 '11 at 9:28

4 Is your 2D array rectangular(`T[,]`) or jagged(`T[][]`)? – [CodesInChaos](#)
Feb 27 '11 at 9:30

[Home](#)[PUBLIC](#)[Stack Overflow](#)[Tags](#)[Users](#)[Jobs](#)**Teams**

Q&A for work

[Learn More](#)

- 2 Why do you want to avoid loops? – [CodesInChaos](#) Feb 27 '11 at 9:41
- 2 And do you want a short or a fast solution? – [CodesInChaos](#) Feb 27 '11 at 10:09
- 2 Yanshof: Could you address the comments in your accepted answer? You claim you want a "fast" solution (given the title) but you've accepted the slowest of the three answers. – [Jon Skeet](#) Feb 27 '11 at 10:15

3 Answers



29



To Convert `double[,]` to `List<double>` ? If you are looking for a one-liner, here goes

```
double[, ] d = new double[, ]
{
    {1.0, 2.0},
    {11.0, 22.0},
    {111.0, 222.0},
    {1111.0, 2222.0},
    {11111.0, 22222.0}
};
List<double> lst = d.Cast<double>().ToList()
```

But, if you are looking for something efficient, I'd rather say you don't use this code.

Please follow either of the two answers mentioned below. Both are implementing much much better techniques.



edited Sep 8 '13 at 9:57

answered Feb 27 '11 at 9:43



[naveen](#)

37.6k 38 140 217

-
- 3 Aside from everything else, that will end up boxing every `double` in the array... it'll perform poorly. – [Jon Skeet](#) Feb 27 '11 at 9:48
-
- 1 @Danny: I'm not really sure how this method is any clearer or easier to understand than a `for` loop, which the OP explicitly wishes to avoid. Not to mention the title says "Fast". – [Cody Gray](#) ♦ Feb 27 '11 at 9:53 
-
- 6 In my quick benchmark of a 1000 x 1000 array, this performs over 30 *times* as slowly as the `for` loop or the `Buffer.BlockCopy` solution. I'm pretty surprised it's been accepted, given the "Fast" part of the title. – [Jon Skeet](#) Feb 27 '11 at 9:59
-
- 2 @downvoters: thanks for letting me know that I know less than JonSkeet. :) Please understand that I am not deleting the answer, because OP used this code somewhere and is happy with it. Tell me, how many of you downvoters work at enterprise level? funny – [naveen](#) Sep 5 '13 at 15:24
-
- 2 @naveen: I see no real evidence of that, and given that the OP can use any of the answers by just copying and pasting them and adjusting to his variable names, they're all equally "fast" by that definition. Even if you think that's the most likely intention, your answer provides no indication of the inefficiency involved which would be appropriate in order to serve *all* readers rather than just the original poster. – [Jon Skeet](#) Sep 5 '13 at 19:09 
-

Well, you can make it use a "blit" sort of copy, although it does mean making an extra copy :(

54

```
double[] tmp = new double[array.GetLength(0) * array.GetLength(1)];
Buffer.BlockCopy(array, 0, tmp, 0, tmp.Length * sizeof(double));
List<double> list = new List<double>(tmp);
```

If you're happy with a single-dimensional array of course, just ignore the last line :)

`Buffer.BlockCopy` is implemented as a native method which I'd expect to use extremely efficient copying after validation. The `List<T>` constructor which accepts an `IEnumerable<T>` is optimized for the case where it implements `IList<T>` , as `double[]` does. It will create a backing array of the right size, and ask it to copy itself into that array. Hopefully that will use `Buffer.BlockCopy` or something similar too.

Here's a quick benchmark of the three approaches (for loop, `Cast<double>().ToList()` , and `Buffer.BlockCopy`):

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Linq;

class Program
{
    static void Main(string[] args)
    {
        double[,] source = new double[1000, 1000];
        int iterations = 1000;

        Stopwatch sw = Stopwatch.StartNew();
        for (int i = 0; i < iterations; i++)
        {
            UsingCast(source);
        }
        sw.Stop();
        Console.WriteLine("LINQ: {0}", sw.ElapsedMilliseconds);

        GC.Collect();
        GC.WaitForPendingFinalizers();

        sw = Stopwatch.StartNew();
        for (int i = 0; i < iterations; i++)
        {
            UsingForLoop(source);
        }
        sw.Stop();
        Console.WriteLine("For loop: {0}", sw.ElapsedMilliseconds);
    }
}
```

```

GC.Collect();
GC.WaitForPendingFinalizers();

sw = Stopwatch.StartNew();
for (int i = 0; i < iterations; i++)
{
    UsingBlockCopy(source);
}
sw.Stop();
Console.WriteLine("Block copy: {0}", sw.ElapsedMilliseconds)
}

static List<double> UsingCast(double[,] array)
{
    return array.Cast<double>().ToList();
}

static List<double> UsingForLoop(double[,] array)
{
    int width = array.GetLength(0);
    int height = array.GetLength(1);
    List<double> ret = new List<double>(width * height);
    for (int i = 0; i < width; i++)
    {
        for (int j = 0; j < height; j++)
        {
            ret.Add(array[i, j]);
        }
    }
    return ret;
}

static List<double> UsingBlockCopy(double[,] array)
{
    double[] tmp = new double[array.GetLength(0) * array.GetLength(1)];
    Buffer.BlockCopy(array, 0, tmp, 0, tmp.Length * sizeof(double));
    List<double> list = new List<double>(tmp);
    return list;
}
}

```

Results (times in milliseconds);

LINQ: 253463
For loop: 9563
Block copy: 8697

EDIT: Having changed the for loop to call `array.GetLength()` on each iteration, the for loop and the block copy take around the same time.

edited Feb 27 '11 at 11:59

answered Feb 27 '11 at 9:44



Jon Skeet

1099k 698 8003
8480

-
- 2 The main problem with that one is that it can leave a big temporary array on the large object heap. – [CodesInChaos](#) Feb 27 '11 at 9:51

@CodeInChaos: Absolutely. It's a pain we can't tell `List<T>` to just use the given array :(I think it's still likely to be faster than looping though. – [Jon Skeet](#) Feb 27 '11 at 9:57

-
- 1 The problem with telling `List<T>` to use a certain array is that we could tell several lists to use the same array. Not sure how big a problem that's be in practice. – [CodesInChaos](#) Feb 27 '11 at 10:02

@CodeInChaos: Yup, that's why there's no way of doing it. It's probably the right decision on the part of the BCL team - it's just irritating for things like this :) – [Jon Skeet](#) Feb 27 '11 at 10:10

One interesting observation on the looping solution is that it's twice as slow if one swaps the inner and outer loop. Most likely due to CPU caches working better if you read/write sequentially. – [CodesInChaos](#) Feb 27 '11 at 11:05

A for loop is the fastest way.

11

You may be able to do it with LINQ, but that will be slower. And while you don't write a loop yourself, under the hood there is still a loop.

- For a jagged array you can probably do something like
`arr.SelectMany(x=>x).ToList()` .
- ~~On `T[,]` you can simply do `arr.ToList()` since the `IEnumerable<T>` of `T[,]` returns all elements in the 2D array.~~
 Looks like the 2D array only implements `IEnumerable` but not `IEnumerable<T>` so you need to insert a `Cast<double>` like yetanothercoder suggested. That will make it even slower due to boxing.

The only thing that can make the code faster than the naive loop is calculating the number of elements and constructing the List with the correct capacity, so it doesn't need to grow.

If your array is rectangular you can obtain the size as `width*height` , with jagged arrays it can be harder.

```
int width=1000;
int height=3000;
double[,] arr=new double[width,height];
List<double> list=new List<double>(width*height);
int size1=arr.GetLength(1);
int size0=arr.GetLength(0);
for(int i=0;i<size0;i++)
{
    for(int j=0;j<size1;j++)
        list.Add(arr[i,j]);
}
```

In theory it might be possible to use private reflection and unsafe code to make it a bit faster doing a raw memory copy. But I strongly advice against that.

edited Feb 27 '11 at 10:06

answered Feb 27 '11 at 9:29

**CodesInChaos**

90.2k 14 173 229

Could you give a sample of what you're thinking about in the `for` loop so I can benchmark it against my `Buffer.BlockCopy` approach? I'd *expect* mine to be faster, but I want to make sure I'm testing the right thing... –

[Jon Skeet](#) Feb 27 '11 at 9:47

I think should be `arr.Cast<T>().ToList()`? – [Cheng Chen](#) Feb 27 '11 at 9:49



Looks like yours if about twice as fast @Jon – [CodesInChaos](#) Feb 27 '11 at 9:57

@Danny That's why I edited it while you were writing your comment. – [CodesInChaos](#) Feb 27 '11 at 9:58

-
- 1 [@CodeInChaos](#): You can optimize yours somewhat by not calling `GetLength` on every iteration... but in my tests, `Buffer.BlockCopy` is still a bit faster. – [Jon Skeet](#) Feb 27 '11 at 10:00
-