How do I clone a generic list in C#?

Ask Question

511

105

I have a generic list of objects in C#, and wish to clone the list. The items within the list are cloneable, but there doesn't seem to be an option to do `list.Clone()` .

Is there an easy way around this?

c# generics list clone

edited Dec 3 '12 at 6:26
Peter Mortensen
14.1k 19 88 114

asked Oct 21 '08 at 16:47
Fiona
2,689 4 14 8

38 You should say if you're looking for a deep copy or a shallow copy – orip Nov 23 '08 at 10:25

9 What are deep and shallow copies? – Colonel Panic Sep 27 '12 at 11:03

4 @ColonelPanic en.wikipedia.org/wiki/Object_copy#Shallow_copy –

Home

PUBLIC

Stack Overflow

Tags

Users

copy. Eg a shallow copy of a list will have the same elements, but will be a different list. – orip Dec 18 '12 at 22:15

26 Answers

340

You can use an extension method.

```
static class Extensions
{
    public static IList<T> Clone<T>(this IList<T> listToCl
    {
        return listToClone.Select(item => (T)item.Clone())
    }
}
```

edited Sep 23 '13 at 19:11

nawfal
44.6k 36 260 307

answered Oct 21 '08 at 16:58

ajm
3,539 1 12 7

64 I think List.ConvertAll might do this in faster time, since it can pre-allocate the entire array for the list, versus having to resize all the time. – MichaelGG Oct 21 '08 at 17:43

2 @MichaelGG, what if you don't want to Convert but just Clone/Duplicate the items in the list? Would this work? || var clonedList = ListOfStrings.ConvertAll(p => p); – IbrarMumtaz Aug 17 '14 at 15:42

no further cast needed: List<MyType> cloned = listToClone.Clone(); – Plutoz May 15 '15 at 7:02

2 this is deep cloning – George Birbilis Jun 14 '15 at 13:34

453

If your elements are value types, then you can just do:

```
List<YourType> newList = new List<YourType>(oldList);
```

However, if they are reference types and you want a deep copy (assuming your elements properly implement `ICloneable`), you could do something like this:

```
List<ICloneable> oldList = new List<ICloneable>();
List<ICloneable> newList = new List<ICloneable>(oldList.Co

oldList.ForEach((item) =>
    {
        newList.Add((ICloneable)item.Clone());
    });
```

Obviously, replace `ICloneable` in the above generics and cast with whatever your element type is that implements `ICloneable`.

If your element type doesn't support `ICloneable` but does have a copy-constructor, you could do this instead:

```
List<YourType> oldList = new List<YourType>();
List<YourType> newList = new List<YourType>(oldList.Count)
```

Personally, I would avoid `ICloneable` because of the need to guarantee a deep copy of all members. Instead, I'd suggest the copy-constructor or a factory method like `YourType.CopyFrom(YourType itemToCopy)` that returns a new instance of `YourType`.

Any of these options could be wrapped by a method (extension or otherwise).

edited Oct 21 '08 at 17:06

answered Oct 21 '08 at 16:54

Jeff Yates
53.3k 16 131 176

1 I think List<T>.ConvertAll might look nicer than creating a new list and doing a foreach+add. – MichaelGG Oct 21 '08 at 17:42

2 @Dimitri: No, that's not true. The problem is, when `ICloneable` was defined, the definition never stated whether the clone was deep or shallow, so you cannot determine which type of Clone operation will be done when an object implements it. This means that if you want to do a deep clone of `List<T>`, you will have to do it without `ICloneable` to be sure it is a deep copy. – Jeff Yates Sep 10 '10 at 14:36

5 Why not use the AddRange method? (`newList.AddRange(oldList.Select(i => i.Clone())` or `newList.AddRange(oldList.Select(i => new YourType(i)`) – phoog Dec 21 '10 at 16:00

4 @phoog: I think that it is a little less readable/understandable when scanning the code, that's all. Readability wins for me. – Jeff Yates Dec 22 '10 at 15:18

to change can sometimes be a *major* performance drain, increasing memory usage by orders of magnitude. – supercat Sep 23 '13 at 21:25

76

```
public static object DeepClone(object obj)
{
  object objResult = null;
  using (MemoryStream  ms = new MemoryStream())
  {
    BinaryFormatter  bf =   new BinaryFormatter();
    bf.Serialize(ms, obj);

    ms.Position = 0;
    objResult = bf.Deserialize(ms);
  }
  return objResult;
}
```

This is one way to do it with C# and .NET 2.0. Your object requires to be `[Serializable()]` . The goal is to lose all references and build new ones.

edited Jul 9 '15 at 13:21

bluish
14.5k 18 94 150

answered Oct 21 '08 at 17:43

Patrick Desjardins
89.8k 77 271 327

11 +1 - i like this answer - it is quick, dirty, nasty and very

3 Quick! but: Why dirty? – raiserle Dec 12 '13 at 11:54

2 This deep clones and is fast and easy. Carefull on other suggestions on this page. I tried several and they don't deep clone. – RandallTo May 29 '15 at 3:14

2 Only negative aspect, if you can call it that, is that your classes have to be marked Serializable for this to work. – Tuukka Haapaniemi Sep 4 '15 at 9:38

75

For a shallow copy, you can instead use the GetRange method of the generic List class.

```
List<int> oldList = new List<int>( );
// Populate oldList...

List<int> newList = oldList.GetRange(0, oldList.Count);
```

Quoted from: Generics Recipes

edited Apr 7 '18 at 14:15
Jochem Broekhoff
20 7

answered Oct 21 '08 at 16:52
Anthony Potts
4,973 5 35 54

33 You can also achieve this by using the List<T>'s contructor to specify a List<T> from which to copy from. eg var shallowClonedList = new List<MyObject>(originalList); –

20

After a slight modification you can also clone:

```
public static T DeepClone<T>(T obj)
{
    T objResult;
    using (MemoryStream ms = new MemoryStream())
    {
        BinaryFormatter bf = new BinaryFormatter();
        bf.Serialize(ms, obj);
        ms.Position = 0;
        objResult = (T)bf.Deserialize(ms);
    }
    return objResult;
}
```

edited Dec 3 '12 at 6:25

Peter Mortensen
14.1k 19 88 114

answered Jul 20 '11 at 9:26

Ajith
225 2 3

Do not forget the T should be serializable, otherwise you get System.Runtime.Serialization.SerializationException. – Bence Végert Nov 23 '17 at 11:28

Good answer. **One hint:** You could add `if (!obj.GetType().IsSerializable) return default(T);` as the first statement which prevents the exception. And if you change it to an extension method, you could even use the Elvis operator like `var b = a?.DeepClone();` (given `var a = new List<string>() { "a", "b" };` for example). – Matt Feb 22 '18 at 14:53

18

```
Microsoft (R) Roslyn C# Compiler version 2.3.2.62116
Loading context from 'CSharpInteractive.rsp'.
Type "#help" for more information.
> var x = new List<int>() { 3, 4 };
> var y = x.ToList();
> x.Add(5)
> x
List<int>(3) { 3, 4, 5 }
> y
List<int>(2) { 3, 4 }
>
```

answered Sep 25 '17 at 0:35

Xavier John

3,696 3 20 26

2 Simplest solution by far – corvuszero Nov 14 '17 at 22:20

15 A little warning this is a shallow copy ... This will create two list objects, but the objects inside will be the same. I.e. changing one property will change the same object / property in the original list. – Mark G Jan 2 '18 at 13:08

15

Unless you need an actual clone of every single object inside your `List<T>`, the best way to clone a list is to create a new list with the old list as the collection parameter.

```
List<T> myList = ...;
List<T> cloneOfMyList = new List<T>(myList);
```

answered Jul 10 '15 at 14:09

Jader Feijo
1,091 1 9 11

4 aka shallow copy – developerbmw Jul 27 '15 at 20:27

I agree with user49126, I'm seeing that it is a shallow copy and changes made to one list are reflected in the other list. – Seidleroni Nov 19 '15 at 15:30

@Seidleroni, you are wrong. The changes made to the list itens are afected on the other list, changes in the list itself are not. – Wellington Zanelli Apr 19 '16 at 17:53

This is shallow copy. – Elliot Chen Aug 5 '16 at 17:58

How is this a shallow copy? – mko Sep 27 '18 at 11:41

13

Use AutoMapper (or whatever mapping lib you prefer) to clone is simple and a lot maintainable.

Define your mapping:

```
Mapper.CreateMap<YourType, YourType>();
```

Do the magic:

```
YourTypeList.ConvertAll(Mapper.Map<YourType, YourType>);
```

answered Feb 13 '13 at 23:20

Derek Liang

13

And you know the type:

```
List<int> newList = new List<int>(oldList);
```

If you don't know the type before, you'll need a helper function:

```
List<T> Clone<T>(IEnumerable<T> oldList)
{
    return newList = new List<T>(oldList);
}
```

The just:

```
List<string> myNewList = Clone(myOldList);
```

edited Apr 15 '13 at 13:08

Lucas B
7,218 5 30 49

answered Oct 21 '08 at 16:54

James Curran
86.7k 30 158 244

15 This doesn't clone the elements. – Jeff Yates Oct 21 '08 at 16:57

12 Keep in mind, this only works for value types. – Dan Bechard Oct 25 '12 at 18:29

```
List<T> newList = JsonConvert.DeserializeObject<T>
(JsonConvert.SerializeObject(listToCopy))
```

Possibly not the most efficient way to do it, but unless you're doing it 100s of 1000s of times you may not even notice the speed difference.

answered Nov 1 '13 at 14:43

ProfNimrod
2,673 1 21 39

3 It's not about the speed difference, it's about the readability. If I came to this line of code I would slap my head and wonder why they introducted a third-party library to serialize and then deserialize an object which I would have no idea why it's happening. Also, this wouldn't work for a model list with objects that have a circular structure. – Jonathon Cwik Feb 4 '15 at 16:41

This code worked excellently for me for deep cloning. The app is migrating document boilerplate from Dev to QA to Prod. Each object is a packet of several document template objects, and each document in turn is comprised of a list of paragraph objects. This code let me serialize the .NET "source" objects and immediately deserialize them to new "target" objects, which then get saved to a SQL database in a different environment. After tons of research, I found lots of stuff, much of which was too cumbersome, and decided to try this. This short and flexible approach was "just right"! – Developer63 Nov 7 '15 at 7:01

```
StreamingContext(StreamingContextStates.Clone));
        binaryFormatter.Serialize(memStream, objtype); mem
SeekOrigin.Begin);
        lstfinal = binaryFormatter.Deserialize(memStream);
    }

    return lstfinal;
}
```

edited Apr 25 '11 at 16:22

Cody Gray ♦
197k 36 388 479

answered Apr 25 '11 at 12:18

pratik
31 1

3

```
public class CloneableList<T> : List<T>, ICloneable where
{
  public object Clone()
  {
    var clone = new List<T>();
    ForEach(item => clone.Add((T)item.Clone()));
    return clone;
  }
}
```

answered Oct 7 '11 at 7:04

Peter
85 1 8

```
        try
        {
            Type sourceType = typeof(TEntity);
            foreach(var o1 in o1List)
            {
                TEntity o2 = new TEntity();
                foreach (PropertyInfo propInfo in (sourceT
                {
                    var val = propInfo.GetValue(o1, null);
                    propInfo.SetValue(o2, val);
                }
                retList.Add(o2);
            }
            return retList;
        }
        catch
        {
            return retList;
        }
    }
```

answered Apr 10 '16 at 7:40

shahrooz.bazrafshan
61 4

2

You could also simply convert the list to an array using `ToArray`, and then clone the array using `Array.Clone(...)`. Depending on your needs, the methods included in the Array class could meet your needs.

edited Jan 15 '15 at 15:34

Avi
15.2k 13 54 93

STILL change the values in the original list. – Bernoulli Lizard Feb 2 '17 at 15:34

you can use var clonedList = ListOfStrings.ConvertAll(p => p); as given by @IbrarMumtaz .... Works effectively... Changes to one list are kept to itself and doesn't to reflect in another – zainul Feb 13 '17 at 7:01

2

You can use extension method:

```
namespace extension
{
    public class ext
    {
        public static List<double> clone(this List<double>
        {
            List<double> kop = new List<double>();
            int x;
            for (x = 0; x < t.Count; x++)
            {
                kop.Add(t[x]);
            }
            return kop;
        }
    };

}
```

You can clone all objects by using their value type members for example, consider this class:

```
public class matrix
{
    public List<List<double>> mat;
    public int rows,cols;
```

```
        int x;
        // I assume I have clone method for List<double>
        for(x=0;x<this.mat.count;x++)
        {
            copy.mat.Add(this.mat[x].clone());
        }
        // then mat is cloned
        return copy; // and copy of original is returned
    }
};
```

Note: if you do any change on copy (or clone) it will not affect the original object.

edited Sep 26 '16 at 9:08

Athafoud
2,106 2 29 45

answered Jun 7 '13 at 11:37

user2463322
21 1

2

If you need a cloned list with the same capacity, you can try this:

```
public static List<T> Clone<T>(this List<T> oldList)
{
    var newList = new List<T>(oldList.Capacity);
    newList.AddRange(oldList);
    return newList;
}
```

21 1

2

My friend Gregor Martinovic and I came up with this easy solution using a JavaScript Serializer. There is no need to flag classes as Serializable and in our tests using the Newtonsoft JsonSerializer even faster than using BinaryFormatter. With extension methods usable on every object.

Standard .NET JavascriptSerializer option:

```
public static T DeepCopy<T>(this T value)
{
    JavaScriptSerializer js = new JavaScriptSerializer();

    string json = js.Serialize(value);

    return js.Deserialize<T>(json);
}
```

Faster option using Newtonsoft JSON:

```
public static T DeepCopy<T>(this T value)
{
    string json = JsonConvert.SerializeObject(value);

    return JsonConvert.DeserializeObject<T>(json);
}
```

edited May 16 '17 at 12:22
Peter Mortensen

Private members are not cloned using the JSON method. stackoverflow.com/a/78612/885627 – himanshupareek66 Nov 22 '17 at 15:08

1

I've made for my own some extension which converts ICollection of items that not implement IClonable

```
static class CollectionExtensions
{
    public static ICollection<T> Clone<T>(this ICollection
    {
        var array = new T[listToClone.Count];
        listToClone.CopyTo(array,0);
        return array.ToList();
    }
}
```

answered Jul 3 '13 at 12:41

wudzik
18.6k 12 67 88

seems some collections (e.g. DataGrid's SelectedItems at Silverlight) skip the implementation of CopyTo which is a problem with this approach – George Birbilis Jun 14 '15 at 13:35

I use automapper to copy an object. I just setup a mapping that maps one object to itself. You can wrap this operation

1,170 2 16 32

1

Using a cast may be helpful, in this case, for a shallow copy:

```
IList CloneList(IList list)
{
    IList result;
    result = (IList)Activator.CreateInstance(list.GetType(
    foreach (object item in list) result.Add(item);
    return result;
}
```

applied to generic list:

```
List<T> Clone<T>(List<T> argument) => (List<T>)CloneList(a
```

answered Feb 27 at 9:51

Thomas Cerny

11 1

0

The following code should transfer onto a list with minimal changes.

Basically it works by inserting a new random number from a greater range with each successive loop. If there exist numbers already that are the same or higher than it, shift those random numbers up one so they transfer into the

```
private int[] getRandomUniqueIndexArray(int length, int co
{
    if(count > length || count < 1 || length < 1)
        return new int[0];

    int[] toReturn = new int[count];
    if(count == length)
    {
        for(int i = 0; i < toReturn.Length; i++) toReturn[
        return toReturn;
    }

    Random r = new Random();
    int startPos = count - 1;
    for(int i = startPos; i >= 0; i--)
    {
        int index = r.Next(length - i);
        for(int j = startPos; j > i; j--)
            if(toReturn[j] >= index)
                toReturn[j]++;
        toReturn[i] = index;
    }

    return toReturn;
}
```

edited May 16 '17 at 11:56

Peter Mortensen
14.1k 19 88 114

answered Sep 3 '15 at 11:23

Adam Lewis
1 1

```
[ProtoContract(ImplicitFields = ImplicitFields.AllPublic)]
public class Person
{
       ...
      Job JobDescription
       ...
}

[ProtoContract(ImplicitFields = ImplicitFields.AllPublic)]
public class Job
{...
}

private static readonly Type stringType = typeof (string);

public static class CopyFactory
{
    static readonly Dictionary<Type, PropertyInfo[]> Prope
PropertyInfo[]>();

    private static readonly MethodInfo CreateCopyReflectio

    static CopyFactory()
    {
        CreateCopyReflectionMethod =
typeof(CopyFactory).GetMethod("CreateCopyReflection", Bind
BindingFlags.Public);
    }

    public static T CreateCopyReflection<T>(T source) wher
    {
        var copyInstance = new T();
        var sourceType = typeof(T);

        PropertyInfo[] propList;
        if (ProperyList.ContainsKey(sourceType))
            propList = ProperyList[sourceType];
        else
        {
            propList = sourceType.GetProperties(BindingFla
```

```
                value != null && prop.PropertyType.IsClass
stringType ?
CreateCopyReflectionMethod.MakeGenericMethod(prop.Property
object[] { value }) : value, null);
        }

        return copyInstance;
    }
```

I measured it in a simple way, by using the Watcher class.

```
var person = new Person
{
    ...
};

for (var i = 0; i < 1000000; i++)
{
   personList.Add(person);
}
var watcher = new Stopwatch();
watcher.Start();
var copylist = personList.Select(CopyFactory.CreateCopyRe
watcher.Stop();
var elapsed = watcher.Elapsed;
```

RESULT: With inner object PersonInstance - 16.4, PersonInstance = null - 5.6

CopyFactory is just my test class where I have dozen of tests including usage of expression. You could implement this in another form in an extension or whatever. Don't forget about caching.

I didn't test serializing yet, but I doubt in an improvement with a million classes. I'll try something fast protobuf/newton

I recently tested the Protocol Buffers serializer with the DeepClone function out of the box. It wins with 4.2 seconds on a million simple objects, but when it comes to inner objects, it wins with the result 7.4 seconds.

```
Serializer.DeepClone(personList);
```

SUMMARY: If you don't have access to the classes, then this will help. Otherwise it depends on the count of the objects. I think you could use reflection up to 10,000 objects (maybe a bit less), but for more than this the Protocol Buffers serializer will perform better.

edited May 16 '17 at 12:02
Peter Mortensen
14.1k 19 88 114

answered Dec 18 '15 at 23:56
Roma Borodov
306 2 9

0

There is a simple way to clone objects in C# using a JSON serializer and deserializer.

You can create an extension class:

```
using Newtonsoft.Json;

static class typeExtensions
{
```

To clone and object:

```
obj clonedObj = originalObj.jsonCloneObject;
```

edited May 16 '17 at 12:23

Peter Mortensen
14.1k 19 88 114

answered Jan 20 '17 at 8:43

Albert arnau
1 3

0

```
//try this
List<string> ListCopy= new List<string>(OldList);
//or try
List<T> ListCopy=OldList.ToList();
```

edited Feb 18 '18 at 4:58

community wiki
2 revs
Steve

I'll be lucky if anybody ever reads this... but in order to not

```
    T Clone();
}
```

Then I specified the extension:

```
public static List<T> Clone<T>(this List<T> listToClone) wl
{
    return listToClone.Select(item => (T)item.Clone()).ToL
}
```

And here is an implementation of the interface in my A/V marking software. I wanted to have my Clone() method return a list of VidMark (while the ICloneable interface wanted my method to return a list of object):

```
public class VidMark : IMyCloneable<VidMark>
{
    public long Beg { get; set; }
    public long End { get; set; }
    public string Desc { get; set; }
    public int Rank { get; set; } = 0;

    public VidMark Clone()
    {
        return (VidMark)this.MemberwiseClone();
    }
}
```

And finally, the usage of the extension inside a class:

```
private List<VidMark> _VidMarks;
private List<VidMark> _UndoVidMarks;

//Other methods instantiate and fill the lists
```

edited Jan 21 at 17:22

answered Jan 21 at 17:15

John Kurtz
347 4 15

0

For a deep copy, ICloneable is the correct solution, but here's a similar approach to ICloneable using the constructor instead of the ICloneable interface.

```
public class Student
{
  public Student(Student student)
  {
    FirstName = student.FirstName;
    LastName = student.LastName;
  }

  public string FirstName { get; set; }
  public string LastName { get; set; }
}

// wherever you have the list
List<Student> students;

// and then where you want to make a copy
List<Student> copy = students.Select(s => new Student(s)).
```

you'll need the following library where you make the copy

```
using System.Lin
```

answered May 13 at 21:04

ztorstri
23 6