Deep cloning objects

Ask Question

2003

586

I want to do something like:

```
MyObject myObj = GetMyObj(); // Create and fill a new object
MyObject newObj = myObj.Clone();
```

And then make changes to the new object that are not reflected in the original object.

I don't often need this functionality, so when it's been necessary, I've resorted to creating a new object and then copying each property individually, but it always leaves me with the feeling that there is a better or more elegant way of handling the situation.

How can I clone or deep copy an object so that the cloned object can be modified without any changes being reflected in the original object?

c# .net clone

edited Dec 16 '15 at 9:42
poke
217k 46 338 402

asked Sep 17 '08 at 0:06
NakedBrunch
30.4k 12 67 94

72 May be useful: "Why Copying an Object is a terrible thing to do?" agiledeveloper.com/articles/cloning072002.htm – Pedro77 Dec 7 '11 at 11:56

17 You should have a look at AutoMapper – Daniel Little Dec 19 '12 at 0:36

3 Your solution is far more complex, I got lost reading it... hehehe. I'm using an DeepClone interface. public interface IDeepCloneable<T> { T DeepClone(); } – Pedro77 Aug 9 '13 at 14:12

41 Answers

1 2 next

1582 ✓

Whilst the standard practice is to implement the `ICloneable` interface (described here, so I won't regurgitate), here's a nice deep clone object copier I found on The Code Project a while ago and incorporated it in our stuff.

As mentioned elsewhere, it does require your objects to be serializable.

```
using System;
using System.IO;
using System.Runtime.Serialization;
using System.Runtime.Serialization.Formatters.Binary;

/// <summary>
/// Reference Article http://www.codeproject.com/KB/tips/Serializec
/// Provides a method for performing a deep copy of an object.
/// Binary Serialization is used to perform the copy.
/// </summary>
public static class ObjectCopier
{
    /// <summary>
    /// Perform a deep Copy of the object.
    /// </summary>
    /// <typeparam name="T">The type of object being copied.</typep
    /// <param name="source">The object instance to copy.</param>
    /// <returns>The copied object.</returns>
    public static T Clone<T>(T source)
    {
        if (!typeof(T).IsSerializable)
        {
            throw new ArgumentException("The type must be serializa
        }

        // Don't serialize a null object, simply return the default
        if (Object.ReferenceEquals(source, null))
        {
            return default(T);
        }

        IFormatter formatter = new BinaryFormatter();
        Stream stream = new MemoryStream();
        using (stream)
        {
            formatter.Serialize(stream, source);
            stream.Seek(0, SeekOrigin.Begin);
            return (T)formatter.Deserialize(stream);
        }
    }
}
```

The idea is that it serializes your object and then deserializes it into a fresh object. The benefit is that you don't have to concern yourself about cloning everything when an object gets too complex.

And with the use of extension methods (also from the originally referenced source):

In case you prefer to use the new extension methods of C# 3.0, change the method to have the following signature:

```
public static T Clone<T>(this T source)
{
   //...
}
```

Now the method call simply becomes `objectBeingCloned.Clone();` .

EDIT (January 10 2015) Thought I'd revisit this, to mention I recently started using (Newtonsoft) Json to do this, it should be lighter, and avoids the overhead of [Serializable] tags. (**NB** @atconway has pointed out in the comments that private members are not cloned using the JSON method)

```
/// <summary>
/// Perform a deep Copy of the object, using Json as a serialisatic
Private members are not cloned using this method.
/// </summary>
/// <typeparam name="T">The type of object being copied.</typeparam
/// <param name="source">The object instance to copy.</param>
/// <returns>The copied object.</returns>
public static T CloneJson<T>(this T source)
{
    // Don't serialize a null object, simply return the default for
    if (Object.ReferenceEquals(source, null))
    {
        return default(T);
    }

    // initialize inner objects individually
    // for example in default constructor some list property initic
values,
    // but in 'source' these items are cleaned -
    // without ObjectCreationHandling.Replace default constructor v
to result
    var deserializeSettings = new JsonSerializerSettings {ObjectCre
ObjectCreationHandling.Replace};

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

edited May 23 '17 at 12:18

community wiki
19 revs, 12 users 65%
johnc

22 stackoverflow.com/questions/78536/cloning-objects-in-c/… has a link to the code above [and references two other such implementations, one of

which is more appropriate in my context] – Ruben Bartelink Feb 4 '09 at 13:13

93 Serialization/deserialization involves significant overhead that isn't necessary. See the ICloneable interface and .MemberWise() clone methods in C#. – 3Dave Jan 28 '10 at 17:28

15 @David, granted, but if the objects are light, and the performance hit when using it is not too high for your requirements, then it is a useful tip. I haven't used it intensively with large amounts of data in a loop, I admit, but I have never seen a single performance concern. – johnc Jan 29 '10 at 0:21

15 @Amir: actually, no: `typeof(T).IsSerializable` is also true if the type has been marked with the `[Serializable]` attribute. It doesn't have to implement the `ISerializable` interface. – Daniel Gehriger Jun 3 '11 at 15:25

11 Just thought I'd mention that whilst this method is useful, and I've used it myself many a time, it's not at all compatible with Medium Trust - so watch out if you're writing code that needs compatibility. BinaryFormatter access private fields and thus cannot work in the default permissionset for partial trust environments. You could try another serializer, but make sure your caller knows that the clone may not be perfect if the incoming object relies on private fields. – Alex Norcliffe Oct 17 '11 at 11:35

Home

PUBLIC

Stack Overflow

Tags

Users

Jobs

1

As nearly all of the answers to this question have been unsatisfactory or plainly don't work in my situation, I have authored AnyClone which is entirely implemented with reflection and solved all of the needs here. I was unable to get serialization to work in a complicated scenario with complex structure, and `IClonable` is less than ideal - in fact it shouldn't even be necessary.

Standard ignore attributes are supported using `[IgnoreDataMember]`, `[NonSerialized]`. Supports complex collections, properties without setters, readonly fields etc.

I hope it helps someone else out there who ran into the same problems I did.

answered Nov 16 '18 at 6:40

Michael Brown
768 11 19

0

The generic approaches are all technically valid, but I just wanted to add a note from myself since we rarely actually need a real deep copy, and I would strongly oppose using generic deep copying in actual business applications since that makes it so you might have many places where the objects are copied and then modified explicitly, its easy to get lost.

In most real-life situations also you want to have as much granular control over the copying process as possible since you are not only coupled to the data access framework but also in practice the copied business objects should rarely be 100% the same. Think an example referenceId's used by the ORM to identify object references, a full deep copy will also copy this id's so while in-memory the objects will be different, as soon as you submit it to the datastore, it will complain, so you will have to modify this properties manually after copying anyway and if the object changes you need to adjust it in all of the places which use the generic deep copying.

Expanding on @cregox answer with ICloneable, what actually is a deep copy? Its just a newly allocated object on the heap that is identical to the original object but occupies a different memory space, as such rather than using a generic cloner functionality why not just create a new object?

I personally use the idea of static factory methods on my domain objects.

Example:

```
    public class Client
    {
        public string Name { get; set; }

        protected Client()
        {
        }

        public static Client Clone(Client copiedClient)
        {
            return new Client
            {
                Name = copiedClient.Name
            };
        }
    }

    public class Shop
    {
        public string Name { get; set; }

        public string Address { get; set; }

        public ICollection<Client> Clients { get; set; }

        public static Shop Clone(Shop copiedShop, string newAddress,
clients)
        {
            var copiedClients = new List<Client>();
            foreach (var client in copiedShop.Clients)
            {
                copiedClients.Add(Client.Clone(client));
            }

            return new Shop
            {
                Name = copiedShop.Name,
                Address = newAddress,
                Clients = copiedClients
            };
        }
    }
```

If someone is looking how he can structure object instantiation while retaining full control over the copying process that's a solution that I have been personally very successful with. The protected constructors also make it so, other developers are forced to use the factory methods which gives a neat single point of object instantiation encapsulating the construction logic inside of the object. You can also overload the method and have several clone logic's for different places if necessary.

edited Nov 13 '18 at 20:17

answered Oct 12 '18 at 12:59

Piotr Jerzy Mamenas
192 2 20

▲
234
▼

I wanted a cloner for very simple objects of mostly primitives and lists. If your object is out of the box JSON serializable then this method will do the trick. This requires no modification or implementation of interfaces on the cloned class, just a JSON serializer like JSON.NET.

```
public static T Clone<T>(T source)
{
    var serialized = JsonConvert.SerializeObject(source);
    return JsonConvert.DeserializeObject<T>(serialized);
}
```

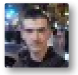
Also, you can use this extension method

```
public static class SystemExtension
{
    public static T Clone<T>(this T source)
    {
        var serialized = JsonConvert.SerializeObject(source);
        return JsonConvert.DeserializeObject<T>(serialized);
        }
    }
```

edited Sep 14 '18 at 6:16

Vineet Choudhary
4,927 2 32 61

answered Apr 3 '13 at 13:31



craastad
3,520 4 23 42

12 the solutiojn is even faster than the BinaryFormatter solution, .NET Serialization Performance Comparison – esskar Mar 12 '14 at 10:25

3 Thanks for this. I was able to do essentially the same thing with the BSON serializer that ships with the MongoDB driver for C#. – Mark Ewer Jun 18 '14 at 0:58

3 This is the best way for me, However, I use `Newtonsoft.Json.JsonConvert` but it is the same – Pierre Feb 4 '15 at 12:20

1 For this to work the object to clone needs to be serializable as already mentioned - this also means for example that it may not have circular dependencies – radomeit Feb 22 '18 at 10:03

1 I think this is the best solution as the implementation can be applied on most programming languages. – mr5 Jan 2 at 7:58

1

C# Extension that'll support for "not **ISerializable**" types too.

```
public static class AppExtensions
{
      public static T DeepClone<T>(this T a)
      {
          using (var stream = new MemoryStream())
          {
              var serializer = new System.Xml.Serialization.XmlSeri
              serializer.Serialize(stream, a);
              stream.Position = 0;
              return (T)serializer.Deserialize(stream);
          }
      }
}
```

Usage

```
var obj2 = obj1.DeepClone()
```

answered May 3 '18 at 8:18

Sameera R.
2,792 1 24 42

20

The short answer is you inherit from the ICloneable interface and then implement the .clone function. Clone should do a memberwise copy and perform a deep copy on any member that requires it, then return the resulting object. This is a recursive operation (it requires that all members of the class you want to clone are either value types or implement ICloneable and that their members are either value types or implement ICloneable, and so on).

For a more detailed explanation on Cloning using ICloneable, check out this article.

The *long* answer is "it depends". As mentioned by others, ICloneable is not supported by generics, requires special considerations for circular class references, and is actually viewed by some as a "mistake" in the .NET Framework. The serialization method depends on your objects being serializable, which they may not be and you may have no control over. There is still much debate in the community over which is the "best" practice. In reality, none of the solutions are the one-size fits all best practice for all situations like ICloneable was originally interpreted to be.

See the this Developer's Corner article for a few more options (credit to Ian).

edited Apr 9 '18 at 22:46
Johann
2,952 2 28 35

answered Sep 17 '08 at 0:14
Zach Burlingame
10.5k 14 50 64

1 ICloneable doesn't have a generic interface, so it is not recommended to use that interface. – Karg Sep 17 '08 at 0:15

1

Yet another JSON.NET answer. This version works with classes that don't implement ISerializable.

```
public static class Cloner
{
    public static T Clone<T>(T source)
    {
        if (ReferenceEquals(source, null))
            return default(T);

        var settings = new JsonSerializerSettings { ContractResolver
ContractResolver() };

        return JsonConvert.DeserializeObject<T>(JsonConvert.Serializ
settings), settings);
    }

    class ContractResolver : DefaultContractResolver
    {
        protected override IList<JsonProperty> CreateProperties(Type
MemberSerialization memberSerialization)
        {
```

```
      var props = type.GetProperties(BindingFlags.Public | Bin
| BindingFlags.Instance)
        .Select(p => base.CreateProperty(p, memberSerializat
        .Union(type.GetFields(BindingFlags.Public | BindingF
BindingFlags.Instance)
          .Select(f => base.CreateProperty(f, memberSerial
        .ToList();
      props.ForEach(p => { p.Writable = true; p.Readable = tru
      return props;
    }
  }
}
```

answered Mar 14 '18 at 11:37

Matthew Watson
73.8k 6 93 180

101

After much much reading about many of the options linked here, and possible solutions for this issue, I believe all the options are summarized pretty well at *Ian P*'s link (all other options are variations of those) and the best solution is provided by *Pedro77*'s link on the question comments.

So I'll just copy relevant parts of those 2 references here. That way we can have:

The best thing to do for cloning objects in c sharp!

First and foremost, those are all our options:

- Manually with **ICloneable**, which is *Shallow* and not *Type-Safe*
- **MemberwiseClone**, which uses ICloneable
- **Reflection** by using Activator.CreateInstance and recursive MemberwiseClone

- **Serialization**, as pointed by johnc's preferred answer
- **Intermediate Language**, which I got no idea how works
- **Extension Methods**, such as this custom clone framework by Havard Straden
- **Expression Trees**

The article Fast Deep Copy by Expression Trees has also performance comparison of cloning by Serialization, Reflection and Expression Trees.

Why I choose *ICloneable* (i.e. manually)

Mr Venkat Subramaniam (redundant link here) explains in much detail why.

All his article circles around an example that tries to be applicable for most cases, using 3 objects: *Person*, *Brain* and *City*. We want to clone a person, which will have its own brain but the same city. You can either picture all problems any of the other methods above can bring or read the article.

This is my slightly modified version of his conclusion:

> Copying an object by specifying `New` followed by the class name often leads to code that is not extensible. Using clone, the application of prototype pattern, is a better way to achieve this. However, using clone as it is provided in C# (and Java) can be quite problematic as well. It is better to provide a protected (non-public) copy constructor and invoke that from the clone method. This gives us the ability to delegate the task of creating an object to an instance of a class itself, thus providing extensibility and also, safely creating the objects using the protected copy constructor.

Hopefully this implementation can make things clear:

```
public class Person : ICloneable
{
    private final Brain brain; // brain is final since I do not want
                // any transplant on it once created!
    private int age;
    public Person(Brain aBrain, int theAge)
    {
        brain = aBrain;
        age = theAge;
    }
    protected Person(Person another)
    {
        Brain refBrain = null;
        try
        {
            refBrain = (Brain) another.brain.clone();
            // You can set the brain in the constructor
        }
        catch(CloneNotSupportedException e) {}
        brain = refBrain;
        age = another.age;
    }
    public String toString()
    {
        return "This is person with " + brain;
        // Not meant to sound rude as it reads!
    }
    public Object clone()
    {
        return new Person(this);
    }
    …
}
```

Now consider having a class derive from Person.

```
public class SkilledPerson extends Person
{
    private String theSkills;
    public SkilledPerson(Brain aBrain, int theAge, String skills)
    {
        super(aBrain, theAge);
        theSkills = skills;
    }
    protected SkilledPerson(SkilledPerson another)
    {
        super(another);
        theSkills = another.theSkills;
    }

    public Object clone()
    {
        return new SkilledPerson(this);
    }
    public String toString()
    {
        return "SkilledPerson: " + super.toString();
    }
}
```

You may try running the following code:

```
public class User
{
    public static void play(Person p)
    {
        Person another = (Person) p.clone();
        System.out.println(p);
        System.out.println(another);
    }
    public static void main(String[] args)
    {
        Person sam = new Person(new Brain(), 1);
        play(sam);
        SkilledPerson bob = new SkilledPerson(new SmarterBrain(), 1,
        play(bob);
    }
}
```

The output produced will be:

```
This is person with Brain@1fcc69
This is person with Brain@253498
SkilledPerson: This is person with SmarterBrain@1fef6f
SkilledPerson: This is person with SmarterBrain@209f4e
```

Observe that, if we keep a count of the number of objects, the clone as implemented here will keep a correct count of the number of objects.

edited Jan 23 '18 at 20:30

Michael Freidgeim
13.5k 6 91 115

answered Sep 26 '12 at 20:18

cregox
11.2k 8 61 100

5 MS recommends not using `ICloneable` for public members. "Because callers of Clone cannot depend on the method performing a predictable cloning operation, we recommend that ICloneable not be implemented in public APIs." msdn.microsoft.com/en-us/library/… However, based on the explanation given by Venkat Subramaniam in your linked article, I think it makes sense to use in this situation *as long as the creators of the ICloneable objects have a deep understanding of which properties should be deep vs. shallow copies* (i.e. deep copy Brain, shallow copy City) – BateTech Jan 9 '15 at 16:57

0

I found a new way to do it that is Emit.

We can use Emit to add the IL to app and run it. But I dont think its a good way for I want to perfect this that I write my answer.

The Emit can see the official document and Guide

You should learn some IL to read the code. I will write the code that can copy the property in class.

```
public static class Clone
{
    // ReSharper disable once InconsistentNaming
    public static void CloneObjectWithIL<T>(T source, T los)
    {
        //see http://lindexi.oschina.io/lindexi/post/C-
%E4%BD%BF%E7%94%A8Emit%E6%B7%B1%E5%85%8B%E9%9A%86/
        if (CachedIl.ContainsKey(typeof(T)))
        {
            ((Action<T, T>) CachedIl[typeof(T)])(source, los);
            return;
        }
        var dynamicMethod = new DynamicMethod("Clone", null, new[] {
typeof(T) });
        ILGenerator generator = dynamicMethod.GetILGenerator();

        foreach (var temp in typeof(T).GetProperties().Where(temp =>
temp.CanWrite))
        {
            //do not copy static that will except
            if (temp.GetAccessors(true)[0].IsStatic)
            {
                continue;
            }

            generator.Emit(OpCodes.Ldarg_1);// los
            generator.Emit(OpCodes.Ldarg_0);// s
            generator.Emit(OpCodes.Callvirt, temp.GetMethod);
            generator.Emit(OpCodes.Callvirt, temp.SetMethod);
        }
        generator.Emit(OpCodes.Ret);
        var clone = (Action<T, T>) dynamicMethod.CreateDelegate(type
        CachedIl[typeof(T)] = clone;
        clone(source, los);
    }

    private static Dictionary<Type, Delegate> CachedIl { set; get; }
Dictionary<Type, Delegate>();
}
```

The code can be deep copy but it can copy the property. If you want to make it to deep copy that you can change it for the IL is too hard that I cant do it.

answered Aug 8 '17 at 0:44

lindexi
2,487 1 8 40

16

1. Basically you need to implement ICloneable interface and then realize object structure copying.
2. If it's deep copy of all members, you need to insure (not relating on solution you choose) that all children are clonable as well.
3. Sometimes you need to be aware of some restriction during this process, for example if you copying the ORM objects most of frameworks allow only one object attached to the session and you MUST NOT make clones of this object, or if it's possible you need to care about session attaching of these objects.

Cheers.

edited Jul 3 '17 at 7:25
Christian Davén
9,912 9 45 63

answered Sep 17 '08 at 0:11
dimarzionist
9,244 4 17 21

4 ICloneable doesn't have a generic interface, so it is not recommended to use that interface. – Karg Sep 17 '08 at 0:13

1

A mapper performs a deep-copy. Foreach member of you object it creates a new object and assign all of its values. It works recursively on each non-primitive inner member.

I suggest you one of the fastest, currently actively developed ones. I suggest UltraMapper https://github.com/maurosampietro/UltraMapper

Nuget packages: https://www.nuget.org/packages/UltraMapper/

edited May 8 '17 at 7:05

answered Apr 23 '17 at 9:16

Mauro Sampietro
1,763 14 37

3

If your Object Tree is Serializeable you could also use something like this

```
static public MyClass Clone(MyClass myClass)
{
    MyClass clone;
    XmlSerializer ser = new XmlSerializer(typeof(MyClass), _xmlAttri
    using (var ms = new MemoryStream())
    {
        ser.Serialize(ms, myClass);
        ms.Position = 0;
        clone = (MyClass)ser.Deserialize(ms);
    }
    return clone;
}
```

be informed that this Solution is pretty easy but it's not as performant as other solutions may be.

And be sure that if the Class grows, there will still be only those fields cloned, which also get serialized.

edited Mar 21 '17 at 14:14

Hakam Fostok
5,771 8 44 70

answered Apr 20 '15 at 13:51

LuckyLikey
1,493 15 33

30

Well I was having problems using ICloneable in Silverlight, but I liked the idea of seralization, I can seralize XML, so I did this:

```
static public class SerializeHelper
{
    //Michael White, Holly Springs Consulting, 2009
    //michael@hollyspringsconsulting.com
    public static T DeserializeXML<T>(string xmlData) where T:new()
    {
        if (string.IsNullOrEmpty(xmlData))
            return default(T);

        TextReader tr = new StringReader(xmlData);
        T DocItms = new T();
        XmlSerializer xms = new XmlSerializer(DocItms.GetType());
        DocItms = (T)xms.Deserialize(tr);

        return DocItms == null ? default(T) : DocItms;
    }

    public static string SerializeObjectToXML<T>(T xmlObject)
    {
        StringBuilder sbTR = new StringBuilder();
        XmlSerializer xmsTR = new XmlSerializer(xmlObject.GetType())
        XmlWriterSettings xwsTR = new XmlWriterSettings();

        XmlWriter xmwTR = XmlWriter.Create(sbTR, xwsTR);
        xmsTR.Serialize(xmwTR,xmlObject);

        return sbTR.ToString();
    }

    public static T CloneObject<T>(T objClone) where T:new()
    {
        string GetString = SerializeHelper.SeralizeObjectToXML<T>(ob
        return SerializeHelper.DeserializeXML<T>(GetString);
    }
}
```

edited Mar 21 '17 at 14:11

Hakam Fostok
5,771 8 44 70

answered Dec 2 '09 at 17:39

Michael White

309 3 2

-1

I know that this question and answer sits here for a while and following is not quite answer but rather observation, to which I came across recently when I was checking whether indeed privates are not being cloned (I wouldn't be myself if I have not ;) when I happily copy-pasted @johnc updated answer.

I simply made myself extension method (which is pretty much copy-pasted form aforementioned answer):

```
public static class CloneThroughJsonExtension
{
    private static readonly JsonSerializerSettings DeserializeSettin
JsonSerializerSettings { ObjectCreationHandling = ObjectCreationHand

    public static T CloneThroughJson<T>(this T source)
    {
        return ReferenceEquals(source, null) ? default(T) :
JsonConvert.DeserializeObject<T>(JsonConvert.SerializeObject(source)
DeserializeSettings);
    }
}
```

and dropped naively class like this (in fact there was more of those but they are unrelated):

```
public class WhatTheHeck
{
    public string PrivateSet { get; private set; } // matches ctor p

    public string GetOnly { get; } // matches ctor param name

    private readonly string _indirectField;
    public string Indirect => $"Inception of: {_indirectField} "; //
```

```
name
    public string RealIndirectFieldVaule => _indirectField;

    public WhatTheHeck(string privateSet, string getOnly, string ind
    {
        PrivateSet = privateSet;
        GetOnly = getOnly;
        _indirectField = indirect;
    }
}
```

and code like this:

```
var clone = new WhatTheHeck("Private-Set-Prop cloned!", "Get-Only-Pr
"Indirect-Field clonned!").CloneThroughJson();
Console.WriteLine($"1. {clone.PrivateSet}");
Console.WriteLine($"2. {clone.GetOnly}");
Console.WriteLine($"3.1. {clone.Indirect}");
Console.WriteLine($"3.2. {clone.RealIndirectFieldVaule}");
```

resulted in:

```
1. Private-Set-Prop cloned!
2. Get-Only-Prop cloned!
3.1. Inception of: Inception of: Indirect-Field cloned!
3.2. Inception of: Indirect-Field cloned!
```

I was whole like: WHAT THE F... so I grabbed Newtonsoft.Json Github repo and started to dig. What it comes out, is that: while deserializing a type which happens to have only one ctor and its param names match (case insensitive) public property names they will be passed to ctor as those params. Some clues can be found in the code here and here.

Bottom line

I know that it is rather not common case and example code is bit abusive, but hey! It got me by surprise when I was checking whether there is any dragon waiting in the bushes to jump out and bite me in the ass. ;)

edited May 23 '17 at 12:10

Community ♦
1 1

answered Feb 1 '17 at 19:30

gaa
931 7 20

19

The best is to implement an **extension method** like

```
public static T DeepClone<T>(this T originalObject)
{ /* the cloning code */ }
```

and then use it anywhere in the solution by

```
var copy = anyObject.DeepClone();
```

We can have the following three implementations:

1. By Serialization (the shortest code)
2. By Reflection - **5x faster**
3. By Expression Trees - **20x faster**

All linked methods are well working and were deeply tested.

edited May 23 '17 at 11:55

Community ♦
1 1

answered Aug 3 '16 at 22:24

frakon
873 1 9 19

26

If you're already using a 3rd party application like ValueInjecter or Automapper, you can do something like this:

```
MyObject oldObj; // The existing object to clone

MyObject newObj = new MyObject();
newObj.InjectFrom(oldObj); // Using ValueInjecter syntax
```

Using this method you don't have to implement ISerializable or ICloneable on your objects. This is common with the MVC/MVVM pattern, so simple tools like this have been created.

see the valueinjecter deep cloning solution on CodePlex.

edited Sep 19 '16 at 22:57
Stacked
3,667 3 44 59

answered Oct 15 '12 at 17:55
Michael Cox
1,083 12 13

4

I think you can try this.

```
MyObject myObj = GetMyObj(); // Create and fill a new object
MyObject newObj = new MyObject(myObj); //DeepClone it
```

answered Aug 19 '16 at 16:47
Sudhanva Kotabagi
108 8

5

Here a solution fast and easy that worked for me without relaying on Serialization/Deserialization.

```
public class MyClass
{
    public virtual MyClass DeepClone()
    {
        var returnObj = (MyClass)MemberwiseClone();
        var type = returnObj.GetType();
        var fieldInfoArray = type.GetRuntimeFields().ToArray();

        foreach (var fieldInfo in fieldInfoArray)
        {
            object sourceFieldValue = fieldInfo.GetValue(this);
            if (!(sourceFieldValue is MyClass))
            {
                continue;
            }

            var sourceObj = (MyClass)sourceFieldValue;
            var clonedObj = sourceObj.DeepClone();
            fieldInfo.SetValue(returnObj, clonedObj);
        }
        return returnObj;
    }
}
```

EDIT: requires

```
    using System.Linq;
    using System.Reflection;
```

That's How I used it

```
public MyClass Clone(MyClass theObjectIneededToClone)
{
    MyClass clonedObj = theObjectIneededToClone.DeepClone();
}
```

edited Jul 29 '16 at 14:15

answered Jul 29 '16 at 13:44

Daniele D.
1,975 3 24 34

5

Code Generator

We have seen a lot of ideas from serialization over manual implementation to reflection and I want to propose a totally different approach using the [CGbR Code Generator](). The generate clone method is memory and CPU efficient and therefor 300x faster as the standard DataContractSerializer.

All you need is a partial class definition with `ICloneable` and the generator does the rest:

```
public partial class Root : ICloneable
{
    public Root(int number)
    {
        _number = number;
    }
    private int _number;

    public Partial[] Partials { get; set; }

    public IList<ulong> Numbers { get; set; }

    public object Clone()
    {
        return Clone(true);
    }

    private Root()
    {
    }
}

public partial class Root
{
    public Root Clone(bool deep)
    {
        var copy = new Root();
        // All value types can be simply copied
        copy._number = _number;
        if (deep)
        {
            // In a deep clone the references are cloned
            var tempPartials = new Partial[Partials.Length];
            for (var i = 0; i < Partials.Length; i++)
            {
                var value = Partials[i];
                value = value.Clone(true);
                tempPartials[i] = value;
            }
            copy.Partials = tempPartials;
            var tempNumbers = new List<ulong>(Numbers.Count);
            for (var i = 0; i < Numbers.Count; i++)
            {
                var value = Numbers[i];
                tempNumbers.Add(value);
            }
            copy.Numbers = tempNumbers;
        }
        else
        {
            // In a shallow clone only references are copied
            copy.Partials = Partials;
            copy.Numbers = Numbers;
        }
        return copy;
    }
}
```

Note: Latest version has a more null checks, but I left them out for better understanding.

edited Jun 9 '16 at 21:24

answered Jun 9 '16 at 20:56

Toxantron
1,640 6 15

11

Keep things simple and use AutoMapper as others mentioned, it's a simple little library to map one object to another... To copy an object to another with the same type, all you need is three lines of code:

```
MyType source = new MyType();
Mapper.CreateMap<MyType, MyType>();
MyType target = Mapper.Map<MyType, MyType>(source);
```

The target object is now a copy of the source object. Not simple enough? Create an extension method to use everywhere in your solution:

```
public static T Copy<T>(this T source)
{
    T copy = default(T);
    Mapper.CreateMap<T, T>();
    copy = Mapper.Map<T, T>(source);
    return copy;
}
```

By using the extension method, the three lines become one line:

```
MyType copy = source.Copy();
```

edited May 28 '16 at 11:23

answered May 28 '16 at 11:02

Stacked
3,667 3 44 59

6

This method solved the problem for me:

```
private static MyObj DeepCopy(MyObj source)
        {

            var DeserializeSettings = new JsonSerializerSettings {
ObjectCreationHandling = ObjectCreationHandling.Replace };

            return JsonConvert.DeserializeObject<MyObj >
(JsonConvert.SerializeObject(source), DeserializeSettings);

        }
```

Use it like this: `MyObj a = DeepCopy(b);`

answered Apr 12 '16 at 13:43

JerryGoyal
11.6k 7 81 90

38

Simple extension method to copy all the public properties. Works for any objects and **does not** require class to be `[Serializable]` . Can be extended for other access level.

```
public static void CopyTo( this object S, object T )
{
    foreach( var pS in S.GetType().GetProperties() )
    {
        foreach( var pT in T.GetType().GetProperties() )
        {
            if( pT.Name != pS.Name ) continue;
            ( pT.GetSetMethod() ).Invoke( T, new object[]
            { pS.GetGetMethod().Invoke( S, null ) } );
        }
    };
}
```

edited Apr 7 '16 at 12:59

SanyTiger

549 1 7 21

answered Mar 16 '11 at 11:38

Konstantin Salavatov

3,119 2 18 19

12 This, unfortunately, is flawed. It's equivalent to calling objectOne.MyProperty = objectTwo.MyProperty (i.e., it will just copy the reference across). It will not clone the values of the properties. – Alex Norcliffe Oct 18 '11 at 0:59

1 to Alex Norcliffe : author of question asked about "copying each property" rather then cloning. in most cases exact duplication of properties is not needed. – Konstantin Salavatov Mar 28 '12 at 9:41

1 i think about using this method but with recursion. so if the value of a property is a reference, create a new object and call CopyTo again. i just see one problem, that all used classes must have a constructor without parameters. Anybody tried this already? i also wonder if this will actually work with properties containing .net classes like DataRow and DataTable? – Koryu Jul 25 '13 at 9:22

7

As I couldn't find a cloner that meets all my requirements in different projects, I created a deep cloner that can be configured and adapted to different code structures instead of adapting my code to meet the cloners requirements. Its achieved by adding annotations to the code that shall be cloned or you just leave the code as it is to have the default behaviour. It uses reflection, type caches and is based on fasterflect. The cloning process is very fast for a huge amount of data and a high object hierarchy (compared to other reflection/serialization based algorithms).

https://github.com/kalisohn/CloneBehave

Also available as a nuget package: https://www.nuget.org/packages/Clone.Behave/1.0.0

For example: The following code will deepClone Address, but only perform a shallow copy of the _currentJob field.

```
public class Person
{
  [DeepClone(DeepCloneBehavior.Shallow)]
  private Job _currentJob;

  public string Name { get; set; }

  public Job CurrentJob
  {
    get{ return _currentJob; }
    set{ _currentJob = value; }
  }

  public Person Manager { get; set; }
}

public class Address
{
  public Person PersonLivingHere { get; set; }
}

Address adr = new Address();
adr.PersonLivingHere = new Person("John");
adr.PersonLivingHere.BestFriend = new Person("James");
adr.PersonLivingHere.CurrentJob = new Job("Programmer");

Address adrClone = adr.Clone();

//RESULT
adr.PersonLivingHere == adrClone.PersonLivingHere //false
adr.PersonLivingHere.Manager == adrClone.PersonLivingHere.Manager //
adr.PersonLivingHere.CurrentJob == adrClone.PersonLivingHere.Current
adr.PersonLivingHere.CurrentJob.AnyProperty ==
adrClone.PersonLivingHere.CurrentJob.AnyProperty //true
```

edited Jan 27 '16 at 12:53

answered Jan 25 '16 at 17:45

kalisohn

221 3 6

3

Ok, there are some obvious example with reflection in this post, BUT reflection is usually slow, until you start to cache it properly.

if you'll cache it properly, than it'll deep clone 1000000 object by 4,6s (measured by Watcher).

```
static readonly Dictionary<Type, PropertyInfo[]> ProperyList = new D
PropertyInfo[]>();
```

than you take cached properties or add new to dictionary and use them simply

```
foreach (var prop in propList)
{
        var value = prop.GetValue(source, null);
        prop.SetValue(copyInstance, value, null);
}
```

full code check in my post in another answer

https://stackoverflow.com/a/34365709/4711853

edited May 23 '17 at 11:47

Community ♦

1 1

answered Dec 19 '15 at 8:17

Roma Borodov
296 2 9

162

The reason not to use ICloneable is **not** because it doesn't have a generic interface. The reason not to use it is because it's vague. It doesn't make clear whether you're getting a shallow or a deep copy; that's up to the implementer.

Yes, `MemberwiseClone` makes a shallow copy, but the opposite of `MemberwiseClone` isn't `Clone` ; it would be, perhaps, `DeepClone` , which doesn't exist. When you use an object through its ICloneable interface, you can't know which kind of cloning the underlying object performs. (And XML comments won't make it clear, because you'll get the interface comments rather than the ones on the object's Clone method.)

What I usually do is simply make a `Copy` method that does exactly what I want.

edited Oct 7 '15 at 18:37

answered Sep 17 '08 at 1:12

Ryan Lundy
158k 31 162 196

27 Your example illustrates the problem. Suppose you have a Dictionary<string, Customer>. Should the cloned Dictionary have the *same* Customer objects as the original, or *copies* of those Customer objects? There are reasonable use cases for either one. But ICloneable doesn't make clear which one you'll get. That's why it's not useful. – Ryan Lundy Jan 12 '11 at 18:53

7

Q. Why would I choose this answer?

- Choose this answer if you want the fastest speed .NET is capable of.
- Ignore this answer if you want a really, really easy method of cloning.

In other words, go with another answer unless you have a performance bottleneck that needs fixing, and you can prove it with a profiler.

10x faster than other methods

The following method of performing a deep clone is:

- 10x faster than anything that involves serialization/deserialization;
- Pretty darn close to the theoretical maximum speed .NET is capable of.

And the method ...

For ultimate speed, you can use **Nested MemberwiseClone to do a deep copy**. Its almost the same speed as copying a value struct, and is much faster than (a) reflection or (b) serialization (as described in other answers on this page).

Note that **if** you use **Nested MemberwiseClone for a deep copy**, you have to manually implement a ShallowCopy for each nested level in the class, and a DeepCopy which calls all said ShallowCopy methods to create a complete clone. This is simple: only a few lines in total, see the demo code below.

Here is the output of the code showing the relative performance difference for 100,000 clones:

- 1.08 seconds for Nested MemberwiseClone on nested structs
- 4.77 seconds for Nested MemberwiseClone on nested classes
- 39.93 seconds for Serialization/Deserialization

Using Nested MemberwiseClone on a class almost as fast as copying a struct, and copying a struct is pretty darn close to the theoretical maximum speed .NET is capable of.

```
Demo 1 of shallow and deep copy, using classes and MemberwiseClone:
  Create Bob
    Bob.Age=30, Bob.Purchase.Description=Lamborghini
  Clone Bob >> BobsSon
  Adjust BobsSon details
    BobsSon.Age=2, BobsSon.Purchase.Description=Toy car
  Proof of deep copy: If BobsSon is a true clone, then adjusting Bob
not affect Bob:
    Bob.Age=30, Bob.Purchase.Description=Lamborghini
  Elapsed time: 00:00:04.7795670,30000000

Demo 2 of shallow and deep copy, using structs and value copying:
  Create Bob
    Bob.Age=30, Bob.Purchase.Description=Lamborghini
  Clone Bob >> BobsSon
  Adjust BobsSon details:
    BobsSon.Age=2, BobsSon.Purchase.Description=Toy car
  Proof of deep copy: If BobsSon is a true clone, then adjusting Bob
not affect Bob:
    Bob.Age=30, Bob.Purchase.Description=Lamborghini
  Elapsed time: 00:00:01.0875454,30000000

Demo 3 of deep copy, using class and serialize/deserialize:
  Elapsed time: 00:00:39.9339425,30000000
```

To understand how to do a deep copy using MemberwiseCopy, here is the demo project that was used to generate the times above:

```
// Nested MemberwiseClone example.
// Added to demo how to deep copy a reference class.
[Serializable] // Not required if using MemberwiseClone, only used f
using serialization.
public class Person
{
    public Person(int age, string description)
    {
        this.Age = age;
        this.Purchase.Description = description;
    }
    [Serializable] // Not required if using MemberwiseClone
    public class PurchaseType
    {
        public string Description;
        public PurchaseType ShallowCopy()
        {
            return (PurchaseType)this.MemberwiseClone();
        }
    }
    public PurchaseType Purchase = new PurchaseType();
    public int Age;
    // Add this if using nested MemberwiseClone.
    // This is a class, which is a reference type, so cloning is mor
    public Person ShallowCopy()
    {
        return (Person)this.MemberwiseClone();
    }
    // Add this if using nested MemberwiseClone.
    // This is a class, which is a reference type, so cloning is mor
    public Person DeepCopy()
    {
            // Clone the root ...
        Person other = (Person) this.MemberwiseClone();
            // ... then clone the nested class.
        other.Purchase = this.Purchase.ShallowCopy();
        return other;
    }
}
// Added to demo how to copy a value struct (this is easy - a deep c
default)
public struct PersonStruct
{
    public PersonStruct(int age, string description)
    {
        this.Age = age;
        this.Purchase.Description = description;
    }
    public struct PurchaseType
    {
        public string Description;
    }
    public PurchaseType Purchase;
    public int Age;
    // This is a struct, which is a value type, so everything is a c
    public PersonStruct ShallowCopy()
    {
        return (PersonStruct)this;
    }
    // This is a struct, which is a value type, so everything is a c
    public PersonStruct DeepCopy()
    {
        return (PersonStruct)this;
    }
}
// Added only for a speed comparison.
public class MyDeepCopy
{
    public static T DeepCopy<T>(T obj)
    {
        object result = null;
        using (var ms = new MemoryStream())
        {
            var formatter = new BinaryFormatter();
            formatter.Serialize(ms, obj);
            ms.Position = 0;
            result = (T)formatter.Deserialize(ms);
            ms.Close();
        }
        return (T)result;
    }
}
```

Then, call the demo from main:

```
void MyMain(string[] args)
{
    {
        Console.Write("Demo 1 of shallow and deep copy, using classe:
MemberwiseCopy:\n");
        var Bob = new Person(30, "Lamborghini");
        Console.Write("  Create Bob\n");
        Console.Write("    Bob.Age={0}, Bob.Purchase.Description={1}'
Bob.Purchase.Description);
        Console.Write("  Clone Bob >> BobsSon\n");
        var BobsSon = Bob.DeepCopy();
        Console.Write("  Adjust BobsSon details\n");
        BobsSon.Age = 2;
        BobsSon.Purchase.Description = "Toy car";
        Console.Write("    BobsSon.Age={0}, BobsSon.Purchase.Descrip
BobsSon.Age, BobsSon.Purchase.Description);
        Console.Write("  Proof of deep copy: If BobsSon is a true cl
BobsSon details will not affect Bob:\n");
        Console.Write("    Bob.Age={0}, Bob.Purchase.Description={1}
Bob.Purchase.Description);
        Debug.Assert(Bob.Age == 30);
        Debug.Assert(Bob.Purchase.Description == "Lamborghini");
        var sw = new Stopwatch();
        sw.Start();
        int total = 0;
        for (int i = 0; i < 100000; i++)
        {
            var n = Bob.DeepCopy();
            total += n.Age;
        }
        Console.Write("  Elapsed time: {0},{1}\n\n", sw.Elapsed, tot
    }
    {
        Console.Write("Demo 2 of shallow and deep copy, using struct
        var Bob = new PersonStruct(30, "Lamborghini");
        Console.Write("  Create Bob\n");
        Console.Write("    Bob.Age={0}, Bob.Purchase.Description={1}
Bob.Purchase.Description);
        Console.Write("  Clone Bob >> BobsSon\n");
        var BobsSon = Bob.DeepCopy();
        Console.Write("  Adjust BobsSon details:\n");
        BobsSon.Age = 2;
        BobsSon.Purchase.Description = "Toy car";
        Console.Write("    BobsSon.Age={0}, BobsSon.Purchase.Descrip
BobsSon.Age, BobsSon.Purchase.Description);
        Console.Write("  Proof of deep copy: If BobsSon is a true cl
BobsSon details will not affect Bob:\n");
        Console.Write("    Bob.Age={0}, Bob.Purchase.Description={1}
Bob.Purchase.Description);
        Debug.Assert(Bob.Age == 30);
        Debug.Assert(Bob.Purchase.Description == "Lamborghini");
        var sw = new Stopwatch();
        sw.Start();
        int total = 0;
        for (int i = 0; i < 100000; i++)
        {
            var n = Bob.DeepCopy();
            total += n.Age;
        }
        Console.Write("  Elapsed time: {0},{1}\n\n", sw.Elapsed, tot
    }
    {
        Console.Write("Demo 3 of deep copy, using class and serializ
        int total = 0;
        var sw = new Stopwatch();
        sw.Start();
        var Bob = new Person(30, "Lamborghini");
        for (int i = 0; i < 100000; i++)
        {
            var BobsSon = MyDeepCopy.DeepCopy<Person>(Bob);
            total += BobsSon.Age;
        }
        Console.Write("  Elapsed time: {0},{1}\n", sw.Elapsed, total
    }
    Console.ReadKey();
}
```

Again, note that **if** you use **Nested MemberwiseClone for a deep copy**, you have to manually implement a ShallowCopy for each nested level in the class, and a DeepCopy which calls all said ShallowCopy methods to create a complete clone. This is simple: only a few lines in total, see the demo code above.

Value types vs. References Types

Note that when it comes to cloning an object, there is is a big difference between a "**struct**" and a "**class**":

- If you have a "**struct**", it's a **value type** so you can just copy it, and the contents will be cloned (but it will only make a shallow clone unless you use the techniques in this post).
- If you have a "**class**", it's a **reference type**, so if you copy it, all you are doing is copying the pointer to it. To create a true clone, you have to be more creative, and use differences between value types and references types which creates another copy of the original object in memory.

See differences between value types and references types.

Checksums to aid in debugging

- Cloning objects incorrectly can lead to very difficult-to-pin-down bugs. In production code, I tend to implement a checksum to double check that the object has been cloned properly, and hasn't been corrupted by another reference to it. This checksum can be switched off in Release mode.
- I find this method quite useful: often, you only want to clone parts of the object, not the entire thing.

Really useful for decoupling many threads from many other threads

One excellent use case for this code is feeding clones of a nested class or struct into a queue, to implement the producer / consumer pattern.

- We can have one (or more) threads modifying a class that they own, then pushing a complete copy of this class into a `ConcurrentQueue` .
- We then have one (or more) threads pulling copies of these classes out and dealing with them.

This works extremely well in practice, and allows us to decouple many threads (the producers) from one or more threads (the consumers).

And this method is blindingly fast too: if we use nested structs, it's 35x faster than serializing/deserializing nested classes, and allows us to take advantage of all of the threads available on the machine.

Update

Apparently, ExpressMapper is as fast, if not faster, than hand coding such as above. I might have to see how they compare with a profiler.

edited Sep 18 '15 at 18:12

answered Jul 4 '15 at 17:24

Contango
41.7k 47 190 238

26

I've just created `CloneExtensions` **library** project. It performs fast, deep clone using simple assignment operations generated by Expression Tree runtime code compilation.

How to use it?

Instead of writing your own `Clone` or `Copy` methods with a tone of assignments between fields and properties make the program do it for yourself, using Expression Tree. `GetClone<T>()` method marked as extension method allows you to simply call it on your instance:

```
var newInstance = source.GetClone();
```

You can choose what should be copied from `source` to `newInstance` using `CloningFlags` enum:

```
var newInstance
    = source.GetClone(CloningFlags.Properties | CloningFlags.Collect
```

What can be cloned?

- Primitive (int, uint, byte, double, char, etc.), known immutable types (DateTime, TimeSpan, String) and delegates (including Action, Func, etc)
- Nullable
- T[] arrays
- Custom classes and structs, including generic classes and structs.

Following class/struct members are cloned internally:

- Values of public, not readonly fields
- Values of public properties with both get and set accessors
- Collection items for types implementing ICollection

How fast it is?

The solution is faster then reflection, because members information has to be gathered only once, before `GetClone<T>` is used for the first time for given type `T` .

It's also faster than serialization-based solution when you clone more then couple instances of the same type `T` .

and more...

Read more about generated expressions on documentation.

Sample expression debug listing for `List<int>` :

```
.Lambda
#Lambda1<System.Func`4[System.Collections.Generic.List`1[System.Int3
(
    System.Collections.Generic.List`1[System.Int32] $source,
    CloneExtensions.CloningFlags $flags,

System.Collections.Generic.IDictionary`2[System.Type,System.Func`2[Sy
 $initializers) {
    .Block(System.Collections.Generic.List`1[System.Int32] $target) 
        .If ($source == null) {
            .Return #Label1 { null }
        } .Else {
            .Default(System.Void)
        };
        .If (
            .Call $initializers.ContainsKey(.Constant<System.Type>
(System.Collections.Generic.List`1[System.Int32]))
        ) {
            $target = (System.Collections.Generic.List`1[System.Int3
($initializers.Item[.Constant<System.Type>
(System.Collections.Generic.List`1[System.Int32])]
```

```
            ).Invoke((System.Object)$source)
        } .Else {
            $target = .New System.Collections.Generic.List`1[System.
        };
        .If (
            ((System.Byte)$flags & (System.Byte).Constant<CloneExtens
(Fields)) == (System.Byte).Constant<CloneExtensions.CloningFlags>(Fi
        ) {
            .Default(System.Void)
        } .Else {
            .Default(System.Void)
        };
        .If (
            ((System.Byte)$flags & (System.Byte).Constant<CloneExtens
(Properties)) == (System.Byte).Constant<CloneExtensions.CloningFlags
        ) {
            .Block() {
                $target.Capacity = .Call CloneExtensions.CloneFactory
                    $source.Capacity,
                    $flags,
                    $initializers)
            }
        } .Else {
            .Default(System.Void)
        };
        .If (
            ((System.Byte)$flags & (System.Byte).Constant<CloneExtens
(CollectionItems)) == (System.Byte).Constant<CloneExtensions.Cloning
(CollectionItems)
        ) {
            .Block(
                System.Collections.Generic.IEnumerator`1[System.Int3
                System.Collections.Generic.ICollection`1[System.Int3
                $var1 = (System.Collections.Generic.IEnumerator`1[Sy
$source.GetEnumerator();
                $var2 = (System.Collections.Generic.ICollection`1[Sy
                .Loop  {
                    .If (.Call $var1.MoveNext() != False) {
                        .Call $var2.Add(.Call CloneExtensions.CloneF
                                $var1.Current,
                                $flags,


                         $initializers))
                    } .Else {
                        .Break #Label2 { }
                    }
            }
            .LabelTarget #Label2:
        }
    } .Else {
        .Default(System.Void)
    };
    .Label
        $target
    .LabelTarget #Label1:
}
```

}

what has the same meaning like following c# code:

```
(source, flags, initializers) =>
{
    if(source == null)
        return null;

    if(initializers.ContainsKey(typeof(List<int>))
        target = (List<int>)initializers[typeof(List<int>)].Invoke((
    else
        target = new List<int>();

    if((flags & CloningFlags.Properties) == CloningFlags.Properties)
    {
        target.Capacity = target.Capacity.GetClone(flags, initialize
    }

    if((flags & CloningFlags.CollectionItems) == CloningFlags.Collec
    {
        var targetCollection = (ICollection<int>)target;
        foreach(var item in (ICollection<int>)source)
        {
            targetCollection.Add(item.Clone(flags, initializers));
        }
    }

    return target;
}
```

Isn't it quite like how you'd write your own `Clone` method for `List<int>` ?

edited Sep 8 '15 at 15:26

answered Dec 24 '13 at 22:56

MarcinJuraszek
109k 11 140 216

2 What are the chances of this getting on NuGet? It seems like the best solution. How does it compare to NClone? – crush May 28 '14 at 19:56

1

When using Marc Gravells protobuf-net as your serializer the accepted answer needs some slight modifications, as the object to copy won't be attributed with `[Serializable]` and, therefore, isn't serializable and the Clone-method will throw an exception.
I modified it to work with protobuf-net:

```
public static T Clone<T>(this T source)
{
    if(Attribute.GetCustomAttribute(typeof(T), typeof(ProtoBuf.Proto
           == null)
    {
        throw new ArgumentException("Type has no ProtoContract!", "s
    }

    if(Object.ReferenceEquals(source, null))
    {
        return default(T);
    }

    IFormatter formatter = ProtoBuf.Serializer.CreateFormatter<T>();
    using (Stream stream = new MemoryStream())
    {
        formatter.Serialize(stream, source);
        stream.Seek(0, SeekOrigin.Begin);
        return (T)formatter.Deserialize(stream);
    }
}
```

This checks for the presence of a `[ProtoContract]` attribute and uses protobufs own formatter to serialize the object.

answered Aug 22 '15 at 11:36

Basti M
5,342 3 25 43

15

If you want true cloning to unknown types you can take a look at fastclone.

That's expression based cloning working about 10 times faster than binary serialization and maintaining complete object graph integrity.

That means: if you refer multiple times to the same object in your hierachy, the clone will also have a single instance beeing referenced.

There is no need for interfaces, attributes or any other modification to the objects being cloned.

edited Aug 12 '15 at 20:07

answered Feb 16 '15 at 11:30

Michael Sander
2,041 18 23

1 I've tried your solution and it seems to work well, thanks! I think this answer should be upvoted more times. Manually implementing ICloneable

is tedious and error-prone, using reflection or serialization is slow if performance is important and you need to copy thousands of objects during a short period of time. – nightcoder Jul 28 '15 at 15:25

1

It's unbelievable how much effort you can spend with IClonable interface - especially if you have heavy class hierarchies. Also MemberwiseClone works somehow oddly - it does not exactly clone even normal List type kind of structures.

And of course most interesting dilemma for serialization is to serialize back references - e.g. class hierarchies where you have child-parent relationships. I doubt that binary serializer will be able to help you in this case. (It will end up with recursive loops + stack overflow).

I somehow liked solution proposed here: [How do you do a deep copy of an object in .NET (C# specifically)?](https://stackoverflow.com/questions/78536/deep-cloning-objects)

however - it did not support Lists, added that support, also took into account re-parenting. For parenting only rule which I have made that field or property should be named "parent", then it will be ignored by DeepClone. You might want to decide your own rules for back-references - for tree hierarchies it might be "left/right", etc...

Here is whole code snippet including test code:

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

namespace TestDeepClone
{
    class Program
    {
        static void Main(string[] args)
        {
            A a = new A();
            a.name = "main_A";
            a.b_list.Add(new B(a) { name = "b1" });
            a.b_list.Add(new B(a) { name = "b2" });

            A a2 = (A)a.DeepClone();
            a2.name = "second_A";

            // Perform re-parenting manually after deep copy.
            foreach( var b in a2.b_list )
                b.parent = a2;


            Debug.WriteLine("ok");

        }
    }

    public class A
    {
        public String name = "one";
        public List<String> list = new List<string>();
        public List<String> null_list;
        public List<B> b_list = new List<B>();
        private int private_pleaseCopyMeAsWell = 5;

        public override string ToString()
        {
            return "A(" + name + ")";
        }
    }

    public class B
    {
        public B() { }
        public B(A _parent) { parent = _parent; }
        public A parent;
        public String name = "two";
    }


    public static class ReflectionEx
    {
        public static Type GetUnderlyingType(this MemberInfo member)
        {
            Type type;
```

```
            switch (member.MemberType)
            {
                case MemberTypes.Field:
                    type = ((FieldInfo)member).FieldType;
                    break;
                case MemberTypes.Property:
                    type = ((PropertyInfo)member).PropertyType;
                    break;
                case MemberTypes.Event:
                    type = ((EventInfo)member).EventHandlerType;
                    break;
                default:
                    throw new ArgumentException("member must be if t
PropertyInfo or EventInfo", "member");
            }
            return Nullable.GetUnderlyingType(type) ?? type;
        }

        /// <summary>
        /// Gets fields and properties into one array.
        /// Order of properties / fields will be preserved in order 
class / struct. (MetadataToken is used for sorting such cases)
        /// </summary>
        /// <param name="type">Type from which to get</param>
        /// <returns>array of fields and properties</returns>
        public static MemberInfo[] GetFieldsAndProperties(this Type 
        {
            List<MemberInfo> fps = new List<MemberInfo>();
            fps.AddRange(type.GetFields());
            fps.AddRange(type.GetProperties());
            fps = fps.OrderBy(x => x.MetadataToken).ToList();
            return fps.ToArray();
        }

        public static object GetValue(this MemberInfo member, object
        {
            if (member is PropertyInfo)
            {
                return (member as PropertyInfo).GetValue(target, nul
            }
            else if (member is FieldInfo)
            {
                return (member as FieldInfo).GetValue(target);
            }
            else
            {
                throw new Exception("member must be either PropertyI
            }
        }

        public static void SetValue(this MemberInfo member, object ta
        {
            if (member is PropertyInfo)
            {
                (member as PropertyInfo).SetValue(target, value, null
            }
            else if (member is FieldInfo)
            {
                (member as FieldInfo).SetValue(target, value);
            }
            else
            {
                throw new Exception("destinationMember must be either
FieldInfo");
            }
        }

        /// <summary>
        /// Deep clones specific object.
        /// Analogue can be found here: https://stackoverflow.com/que
do-you-do-a-deep-copy-an-object-in-net-c-specifically
        /// This is now improved version (list support added)
        /// </summary>
        /// <param name="obj">object to be cloned</param>
        /// <returns>full copy of object.</returns>
        public static object DeepClone(this object obj)
        {
            if (obj == null)
                return null;

            Type type = obj.GetType();

            if (obj is IList)
            {
                IList list = ((IList)obj);
                IList newlist = (IList)Activator.CreateInstance(obj.G
list.Count);

                foreach (object elem in list)
                    newlist.Add(DeepClone(elem));

                return newlist;
            } //if
            if (type.IsValueType || type == typeof(string))
            {
                return obj;
            }
            else if (type.IsArray)
            {
                Type elementType = Type.GetType(type.FullName.Replac
string.Empty));
                var array = obj as Array;
                Array copied = Array.CreateInstance(elementType, arr

                for (int i = 0; i < array.Length; i++)
                    copied.SetValue(DeepClone(array.GetValue(i)), i)

                return Convert.ChangeType(copied, obj.GetType());
            }
            else if (type.IsClass)
            {
                object toret = Activator.CreateInstance(obj.GetType(

                MemberInfo[] fields = type.GetFieldsAndProperties();
                foreach (MemberInfo field in fields)
                {
                    // Don't clone parent back-reference classes. (U
naming 'parent'
                    // to indicate child's parent class.
                    if (field.Name == "parent")
                    {
                        continue;
                    }

                    object fieldValue = field.GetValue(obj);

                    if (fieldValue == null)
                        continue;

                    field.SetValue(toret, DeepClone(fieldValue));
                }

                return toret;
            }
            else
            {
                // Don't know that type, don't know how to clone it.
                if (Debugger.IsAttached)
                    Debugger.Break();
                return null;
            }
        } //DeepClone
    }

}
```

edited May 23 '17 at 12:18

Community ♦

1 1

answered Apr 24 '15 at 19:39

TarmoPikaro

1,892 1 17 31

1 2 next

protected by casperOne Jun 27 '12 at 17:00

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?