Deep cloning objects

Asked 11 years ago Active 24 days ago Viewed 760k times

2096

605

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
233k 51 362 428

asked Sep 17 '08 at 0:06
NakedBrunch
31k 12 68 95

74 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

stackoverflow.com/questions/8025890/… Another solution... – Felix K. Mar 16 '12 at 16:39

18 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

@Pedro77: A concern I have with `IDeepCloneable` is that not all collections of references to things that can be deep-cloned should be; the proper

45 Answers

1 2 next

1633

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/SerializedObjectCloner.aspx
/// 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.</typeparam>
    /// <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 serializable.",
nameof(source));
        }

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

```
            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 serialisation method. NOTE:
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 that object
    if (Object.ReferenceEquals(source, null))
    {
        return default(T);
```

```
    // without ObjectCreationHandling.Replace default constructor values will be added
to result
    var deserializeSettings = new JsonSerializerSettings {ObjectCreationHandling =
ObjectCreationHandling.Replace};

    return JsonConvert.DeserializeObject<T>(JsonConvert.SerializeObject(source),
deserializeSettings);
}
```

edited Aug 22 at 12:18

community wiki
20 revs, 14 users 64%
johnc

23 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

97 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

17 @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

16 @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

258

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)
{
```

```
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
5,284 2 32 65

answered Apr 3 '13 at 13:31

craastad
3,775 5 25 42

13 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

169

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.

I'm not clear why ICloneable is considered vague. Given a type like Dictionary(Of T,U), I would expect that ICloneable.Clone should do whatever level of deep and shallow copying is necessary to make the new dictionary be an independent dictionary that contains the same T's and U's (struct contents, and/or object references) as the original. Where's the ambiguity? To be sure, a generic ICloneable(Of T), which inherited ISelf(Of T), which included a "Self" method, would be much better, but I don't see ambiguity on deep vs shallow cloning. – supercat Jan 12 '11 at 18:35

28 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

@Kyralessa The Microsoft MSDN article actually states this very problem of not knowing if you are requesting a deep or shallow copy. – crush May 28 '14 at 19:05

The answer from the duplicate stackoverflow.com/questions/129389/… describes Copy extension, based on recursive MembershipClone – Michael Freidgeim Jan 23 '18 at 12:15

106

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**

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;
    }
```

```
    {
        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);
```

```
            SkilledPerson bob = new SkilledPerson(new SmarterBrain(), 1, "Writer");
            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
15k 6 97 120

answered Sep 26 '12 at 20:18

cregox
11.6k 8 61 102

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

First off, I'm far from an expert in this topic (public APIs). I *think* for once that MS remark makes a lot of sense. And I don't think it's safe to assume the **users** of that API will have such a deep understanding. So, it only makes sense implementing it on a **public API** if it really won't matter for whoever is going to use it. I *guess* having some kind of UML very explicitly making the distinction on each property could help. But I'd like to hear from someone with more experience. :P – cregox Jan 10 '15 at 3:45

You can use the CGbR Clone Generator and get a similar result without manually writing the code. – Toxantron Jun 9 '16 at 20:53

Intermediate Language implementation is useful – Michael Freidgeim Jan 23 '18 at 12:35

There's no final in C# – Konrad Sep 5 '18 at 11:31

5 .Net doesn't have copy constructors. – Pop Catalin Sep 17 '08 at 0:45

47 Sure it does: new MyObject(objToCloneFrom) Just declare a ctor which takes the object to clone as a parameter. – Nick Sep 17 '08 at 11:49

29 It's not the same thing. You have to add it to every class manually, and you don't even know if you're garantueeing a deep copy. – Dave Van den Eynde Jun 4 '09 at 8:01

14 +1 for copy ctor. You have to manually write a clone() function for each type of object too, and good luck with that when your class hierarchy gets a few levels deep. – Andrew Grant Sep 15 '09 at 0:50

3 With copy constructors you lose hierarchy though. agiledeveloper.com/articles/cloning072002.htm – Will Nov 6 '11 at 21:10

40

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
3,319 2 20 20

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

properties containing .net classes like DataRow and DataTable? – Koryu Jul 25 '13 at 9:22

32

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 SeralizeObjectToXML<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>(objClone);
        return SerializeHelper.DeserializeXML<T>(GetString);
    }
}
```

29

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.CollectionItems);
```

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?

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.Int32],CloneExtensions.Clo
(
    System.Collections.Generic.List`1[System.Int32] $source,
    CloneExtensions.CloningFlags $flags,

System.Collections.Generic.IDictionary`2[System.Type,System.Func`2[System.Object,System.Ob
 $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]))
        ) {
            $target = (System.Collections.Generic.List`1[System.Int32]).Call
($initializers.Item[.Constant<System.Type>
(System.Collections.Generic.List`1[System.Int32])]
            ).Invoke((System.Object)$source)
        } .Else {
            $target = .New System.Collections.Generic.List`1[System.Int32]()
        };
        .If (
            ((System.Byte)$flags & (System.Byte).Constant<CloneExtensions.CloningFlags>
(Fields)) == (System.Byte).Constant<CloneExtensions.CloningFlags>(Fields)
        ) {
            .Default(System.Void)
        } .Else {
            .Default(System.Void)
        };
        .If (
            ((System.Byte)$flags & (System.Byte).Constant<CloneExtensions.CloningFlags>
(Properties)) == (System.Byte).Constant<CloneExtensions.CloningFlags>(Properties)
```

```
                    $initializers)
            }
        } .Else {
            .Default(System.Void)
        };
        .If (
            ((System.Byte)$flags & (System.Byte).Constant<CloneExtensions.CloningFlags>
(CollectionItems)) == (System.Byte).Constant<CloneExtensions.CloningFlags>
(CollectionItems)
        ) {
            .Block(
                System.Collections.Generic.IEnumerator`1[System.Int32] $var1,
                System.Collections.Generic.ICollection`1[System.Int32] $var2) {
                $var1 = (System.Collections.Generic.IEnumerator`1[System.Int32]).Call
$source.GetEnumerator();
                $var2 = (System.Collections.Generic.ICollection`1[System.Int32])$target;
                .Loop  {
                    .If (.Call $var1.MoveNext() != False) {
                        .Call $var2.Add(.Call CloneExtensions.CloneFactory.GetClone(
                                $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:

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

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

    if((flags & CloningFlags.CollectionItems) == CloningFlags.CollectionItems)
    {
        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
111k 11 152 225

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

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 14:15

Not at all, you wrong about reflection, you should simply cache this properly. Check my answer below stackoverflow.com/a/34368738/4711853 – Roma Borodov Dec 19 '15 at 9:32

```
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
3,966 3 46 62

answered Oct 15 '12 at 17:55
Michael Cox
1,103 12 13

21

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
950 1 10 19

the Func at runtime, please check if you have some solution.In fact I have seen issue only with complex objects with deep hierarchy, simple one easily get copied – Mrinal Kamboj Dec 24 '17 at 14:36

ExpressionTree implementation seems very good. It even works with circular references and private members. No attributes needed. Best answer I've found. – N73k Jul 1 at 20:29

21

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

3,082 2 32 35

answered Sep 17 '08 at 0:14

Zach Burlingame

10.7k 14 51 65

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

Your solution works until it needs to handle circular references, then things start to complicate, it's better to try implement deep cloning using deep serialization. – Pop Catalin Sep 17 '08 at 0:46

Unfortunately, not all objects are serializable either, so you can't always use that method either. Ian's link is the most comprehensive answer so far. – Zach Burlingame Sep 17 '08 at 0:56

+1 for mentioning the Brad Abrams article. – Merlyn Morgan-Graham Nov 6 '11 at 8:33

17

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
10.5k 9 47 64

answered Sep 17 '08 at 0:11

dimarzionist
9,434 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

Simple and concise answers are the best. – DavidGuaita Apr 20 '18 at 0:45

16

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
2,258 18 26

This one seems to be pretty useful – LuckyLikey Apr 20 '15 at 13:53

It's easier to start working from one code snapshot than for overall system, especially closed one. It's quite understandable that no library can solve all problems with one shot. Some relaxations should be made. – TarmoPikaro Apr 24 '15 at 19:56

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

12

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

3,966 3 46 62

Be careful with this one, it performs really poorly. I ended up switching to johnc answer which is as short as this one and performs a lot better. – Agorilla Apr 11 '17 at 7:42

@Agorilla, how do you know the performance pool? – Bigeyes Apr 14 '18 at 12:22

This only does a shallow copy. – N73k Jul 1 at 18:27

```
static public IEnumerable<SpotPlacement> CloneList(List<SpotPlacement> spotPlacements)
{
    foreach (SpotPlacement sp in spotPlacements)
    {
        yield return (SpotPlacement)sp.Clone();
    }
}
```

And at another place:

```
public object Clone()
{
    OrderItem newOrderItem = new OrderItem();
    ...
    newOrderItem._exactPlacements.AddRange(SpotPlacement.CloneList(_exactPlacements));
    ...
    return newOrderItem;
}
```

I tried to come up with oneliner that does this, but it's not possible, due to yield not working inside anonymous method blocks.

Better still, use generic List<T> cloner:

```
class Utility<T> where T : ICloneable
{
    static public IEnumerable<T> CloneList(List<T> tl)
    {
        foreach (T t in tl)
        {
            yield return (T)t.Clone();
        }
    }
}
```

edited Dec 3 '12 at 17:23

Peter Mortensen
14.4k 19 88 117

answered Sep 30 '09 at 9:51

Daniel Mošmondor
14.4k 11 51 92

9

- 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
```

```
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 BobsSon details will
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 for speed comparison
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 more difficult.
```

```
    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 copy happens by
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 clone by default.
    public PersonStruct ShallowCopy()
    {
        return (PersonStruct)this;
    }
    // This is a struct, which is a value type, so everything is a clone by default.
    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())
        {
```

```
            return (T)result;
        }
    }
```

Then, call the demo from main:

```
void MyMain(string[] args)
{
    {
        Console.Write("Demo 1 of shallow and deep copy, using classes and
MemberwiseCopy:\n");
        var Bob = new Person(30, "Lamborghini");
        Console.Write("  Create Bob\n");
        Console.Write("    Bob.Age={0}, Bob.Purchase.Description={1}\n", Bob.Age,
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.Description={1}\n",
BobsSon.Age, BobsSon.Purchase.Description);
        Console.Write("  Proof of deep copy: If BobsSon is a true clone, then adjusting
BobsSon details will not affect Bob:\n");
        Console.Write("    Bob.Age={0}, Bob.Purchase.Description={1}\n", Bob.Age,
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, total);
    }
    {
        Console.Write("Demo 2 of shallow and deep copy, using structs:\n");
        var Bob = new PersonStruct(30, "Lamborghini");
```

```
            BobsSon.Age = 2;
            BobsSon.Purchase.Description = "Toy car";
            Console.Write("    BobsSon.Age={0}, BobsSon.Purchase.Description={1}\n",
    BobsSon.Age, BobsSon.Purchase.Description);
            Console.Write("  Proof of deep copy: If BobsSon is a true clone, then adjusting
    BobsSon details will not affect Bob:\n");
            Console.Write("    Bob.Age={0}, Bob.Purchase.Description={1}\n", Bob.Age,
    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, total);
        }
        {
            Console.Write("Demo 3 of deep copy, using class and serialize/deserialize:\n");
            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

- 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

@Lasse V. Karlsen. Yes, you're absolutely correct, I've updated the answer to make this clearer. This method can be used to make deep copies of structs *and* classes. You can run the included example demo code to show how its done, it has an example of deep cloning a nested struct, and another example of deep cloning a nested class. – Contango Jul 4 '15 at 17:51

8

In general, you implement the ICloneable interface and implement Clone yourself. C# objects have a built-in MemberwiseClone method that performs a shallow copy that can help you out for all the primitives.

For a deep copy, there is no way it can know how to automatically do it.

answered Sep 17 '08 at 0:09

HappyDude
1,260 3 12 13

ICloneable doesn't have a generic interface, so it is not recommended to use that interface. – Karg Sep 17 '08 at 0:12

8

I've seen it implemented through reflection as well. Basically there was a method that would iterate through the members of an object and appropriately copy them to the new object. When it reached reference types or collections I think it did a recursive call on itself. Reflection is expensive, but it worked pretty well.

answered Oct 19 '10 at 13:01

xr280xr
6,855 4 56 94

8

Here is a deep copy implementation:

```
public static object CloneObject(object opSource)
{
    //grab the type and create a new instance of that type
    Type opSourceType = opSource.GetType();
```

```
    //iterate over the properties and if it has a 'set' method assign it from the source
TO the target
    foreach (PropertyInfo item in opPropertyInfo)
    {
        if (item.CanWrite)
        {
            //value types can simply be 'set'
            if (item.PropertyType.IsValueType || item.PropertyType.IsEnum ||
item.PropertyType.Equals(typeof(System.String)))
            {
                item.SetValue(opTarget, item.GetValue(opSource, null), null);
            }
            //object/complex types need to recursively call this method until the end of
the tree is reached
            else
            {
                object opPropertyValue = item.GetValue(opSource, null);
                if (opPropertyValue == null)
                {
                    item.SetValue(opTarget, null, null);
                }
                else
                {
                    item.SetValue(opTarget, CloneObject(opPropertyValue), null);
                }
            }
        }
    }
    //return the new item
    return opTarget;
}
```

answered Sep 6 '11 at 7:38

dougajmcdonald
13.8k 7 44 81

2 This looks like memberwise clone because does not aware of reference type properties – sll Nov 6 '11 at 10:17

1 If you want blindingly fast performance, don't go for this implementation: it uses reflection, so it won't be that fast. Conversely, "premature optmization is the of all evil", so ignore the performance side until after you've run a profiler. – Contango Dec 30 '11 at 17:30

8

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();
```

```
adr.PersonLivingHere.CurrentJob.AnyProperty ==
adrClone.PersonLivingHere.CurrentJob.AnyProperty //true
```

edited Jan 27 '16 at 12:53

answered Jan 25 '16 at 17:45

kalisohn

241 3 6

This method solved the problem for me:

7

```
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

13.7k 9 88 101

I like Copyconstructors like that:

6

```
    public AnyObject(AnyObject anyObject)
    {
        foreach (var property in typeof(AnyObject).GetProperties())
        {
            property.SetValue(this, property.GetValue(anyObject));
```

```
        }
    }
```

If you have more things to copy add them

edited Mar 6 '15 at 15:39

answered Mar 6 '15 at 13:48

LuckyLikey
1,836 16 35

Code Generator

6

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 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

1,702 6 17

```
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.

6

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
132 1 9

5

Follow these steps:

- Define an `ISelf<T>` with a read-only `Self` property that returns `T`, and `ICloneable<out T>`, which derives from `ISelf<T>` and includes a method `T Clone()`.
- Then define a `CloneBase` type which implements a `protected virtual generic VirtualClone` casting `MemberwiseClone` to the passed-in type.
- Each derived type should implement `VirtualClone` by calling the base clone method and then doing whatever needs to be done to properly clone those aspects of the derived type which the parent VirtualClone method hasn't yet handled.

For maximum inheritance versatility, classes exposing public cloning functionality should be `sealed`, but derive from a base class which is otherwise identical except for the lack of cloning. Rather than passing variables of the explicit clonable type, take a parameter of type `ICloneable<theNonCloneableType>`. This will allow a routine that expects a cloneable derivative of `Foo` to work with a cloneable derivative of `DerivedFoo`, but also allow the creation of non-cloneable derivatives of `Foo`.

edited Jun 4 '13 at 14:29

Mifeet
9,141 3 38 85

answered Dec 7 '11 at 21:24

supercat
60.2k 4 127 163

4

I have created a version of the accepted answer that works with both '[Serializable]' and '[DataContract]'. It has been a while since I wrote it, but if I remember correctly [DataContract] needed a different serializer.

```
        /// <summary>
        /// Perform a deep Copy of an object that is marked with '[Serializable]' or
    '[DataContract]'
        /// </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 Clone<T>(T source)
        {
            if (typeof(T).IsSerializable == true)
            {
                return CloneUsingSerializable<T>(source);
            }

            if (IsDataContract(typeof(T)) == true)
            {
                return CloneUsingDataContracts<T>(source);
            }

            throw new ArgumentException("The type must be Serializable or use
    DataContracts.", "source");
        }


        /// <summary>
        /// Perform a deep Copy of an object that is marked with '[Serializable]'
        /// </summary>
        /// <remarks>
        /// Found on http://stackoverflow.com/questions/78536/cloning-objects-in-c-sharp
        /// Uses code found on CodeProject, which allows free use in third party apps
        /// - http://www.codeproject.com/KB/tips/SerializedObjectCloner.aspx
        /// </remarks>
        /// <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 CloneUsingSerializable<T>(T source)
        {
            if (!typeof(T).IsSerializable)
            {
                throw new ArgumentException("The type must be serializable.", "source");
            }

            // Don't serialize a null object, simply return the default for that object
```

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


        /// <summary>
        /// Perform a deep Copy of an object that is marked with '[DataContract]'
        /// </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 CloneUsingDataContracts<T>(T source)
        {
            if (IsDataContract(typeof(T)) == false)
            {
                throw new ArgumentException("The type must be a data contract.", "source");
            }

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

            DataContractSerializer dcs = new DataContractSerializer(typeof(T));
            using(Stream stream = new MemoryStream())
            {
                using (XmlDictionaryWriter writer =
XmlDictionaryWriter.CreateBinaryWriter(stream))
                {
                    dcs.WriteObject(writer, source);
                    writer.Flush();
                    stream.Seek(0, SeekOrigin.Begin);
                    using (XmlDictionaryReader reader =
XmlDictionaryReader.CreateBinaryReader(stream, XmlDictionaryReaderQuotas.Max))
                    {
                        return (T)dcs.ReadObject(reader);
                    }
```

```
    /// Helper function to check if a class is a [DataContract]
    /// </summary>
    /// <param name="type">The type of the object to check.</param>
    /// <returns>Boolean flag indicating if the class is a DataContract (true) or not
(false) </returns>
    public static bool IsDataContract(Type type)
    {
        object[] attributes = type.GetCustomAttributes(typeof(DataContractAttribute),
false);
        return attributes.Length == 1;
    }

}
```

answered Apr 11 '14 at 16:06

Jeroen Ritmeijer

2,545 3 20 30

4

To clone your class object you can use the Object.MemberwiseClone method,

just add this function to your class :

```
public class yourClass
{
    // ...
    // ...

    public yourClass DeepCopy()
    {
        yourClass othercopy = (yourClass)this.MemberwiseClone();
        return othercopy;
    }
}
```

then to perform a deep independant copy, just call the DeepCopy method :

Chtiwi Malek
7,483 1 54 55

3 MemberwiseClone method creates a shallow copy NOT a deep copy. msdn.microsoft.com/en-us/library/… – odyth Dec 21 '14 at 1:38

4

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 Dictionary<Type,
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
336 2 9

Calling `prop.GetValue(...)` is still reflection and can't be cached. In an expression tree its compiled though, so faster – Tseng Sep 28 '16 at 14:17

```
    MyClass clone;
    XmlSerializer ser = new XmlSerializer(typeof(MyClass), _xmlAttributeOverrides);
    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

Hakan Fıstık

6,715 8 48 73

answered Apr 20 '15 at 13:51

LuckyLikey

1,836 16 35

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?