How to restrict access to nested class member to enclosing class?

Asked 15 years, 1 month ago Modified 4 years, 3 months ago Viewed 28k times

81

Is it possible to specify that members of a nested class can be accessed by the enclosing class, but not other classes ?

Here's an illustration of the problem (of course my actual code is a bit more complex...) :

```
public class Journal
{
    public class JournalEntry
    {
        public JournalEntry(object value)
        {
            this.Timestamp = DateTime.Now;
            this.Value = value;
        }

        public DateTime Timestamp { get; private set; }
        public object Value { get; private set; }
    }

    // ...
}
```

I would like to prevent client code from creating instances of `JournalEntry`, but `Journal` must be able to create them. If I make the constructor public, anyone can create instances... but if I make it private, `Journal` won't be able to !

Note that the `JournalEntry` class must be public, because I want to be able to expose existing entries to client code.

Any suggestion would be appreciated !

UPDATE: Thanks everyone for your input, I eventually went for the public `IJournalEntry` interface, implemented by a private `JournalEntry` class (despite the last requirement in my question...)

c# nested-class access-levels

Share edited Nov 3, 2009 at 3:27 asked Nov 3, 2009 at 2:03

Improve this question

Follow

Thomas Levesque
292k ● 72 ● 635 ● 766

2 You could make the JournalEntry(object) constructor 'internal'; this would prevent other assemblies from instantiating journal entries but other classes in the same assembly can still make them; however if you're the author of the assembly this might be good enough. – John K Nov 3, 2009 at 2:24

4 yes, I thought of that, but I'd prefer to be unable to create instances even in the same assembly... thanks anyway ! – Thomas Levesque Nov 3, 2009 at 3:17

7 Answers

Sorted by: Highest score (default)

94

Actually there is a complete and simple solution to this problem that doesn't involve modifying the client code or creating an interface.

This solution is actually faster than the interface-based solution for most cases, and easier to code.

```
public class Journal
{
  private static Func<object, JournalEntry> _newJournalEntry;

  public class JournalEntry
  {
    static JournalEntry()
    {
      _newJournalEntry = value => new JournalEntry(value);
    }
    private JournalEntry(object value)
    {
      ...
```

Share

Improve this answer

Follow

edited Jun 14, 2016 at 21:44

Benjamin Hodgson
44.5k ● 17 ● 112 ● 166

answered Nov 3, 2009 at 6:29

Ray Burns
62.8k ● 12 ● 145 ● 142

3 Nice and original approach... I have to remember that one. Thanks ! – Thomas Levesque Nov 3, 2009 at 13:08

2 Spiffy. The accepted answer has the same general idea but it comes at the bottom of a bunch of other alternatives, so look here! – aggieNick02 Apr 22, 2013 at 15:54

11 The problem I found with this approach is that because static constructors in .NET are lazily invoked, you might find yourself having your `_newJournalEntry` being null - but `System.Runtime.CompilerServices.RuntimeHelpers.RunClassConstructor` can save the day. :) – Regent Feb 18, 2016 at 16:08

2 @Regent can throw in an empty static void method as well to get static ctor to be called. Not beautiful, but effective. – joelmdev Jun 20, 2019 at 17:15

4 Don't call us, we'll call you. And then you can call us. – Kyle Delaney Apr 22, 2020 at 17:50

61

If your class is not too complex, you could either use an interface which is publicly visible and make the actual implementing class private, or you could make a protected constructor for the `JornalEntry` class and have a private class `JornalEntryInstance` derived from `JornalEntry` with a public constructor which is actually instantiated by your `Journal`.

```
public class Journal
{
    public class JournalEntry
    {
        protected JournalEntry(object value)
        {
            this.Timestamp = DateTime.Now;
            this.Value = value;
        }

        public DateTime Timestamp { get; private set; }
        public object Value { get; private set; }
    }

    private class JournalEntryInstance: JournalEntry
    {
        public JournalEntryInstance(object value): base(value)
        { }
    }
    JournalEntry CreateEntry(object value)
    {
        return new JournalEntryInstance(value);
    }
}
```

If your actual class is too complex to do either of that and you can get away with the constructor being not completely invisible, you can make the constructor internal so it is only visible in the assembly.

If that too is infeasible, you can always make the constructor private and use reflection to call it from your journal class:

```
typeof(object).GetConstructor(new Type[] { }).Invoke(new Object[] { value });
```

Now that I think about it, another possibility would use a private delegate in the containing class which is set from the inner class

```
public class Journal
{
    private static Func<object, JournalEntry> EntryFactory;
    public class JournalEntry
    {
        internal static void Initialize()
        {
            Journal.EntryFactory = CreateEntry;
        }
        private static JournalEntry CreateEntry(object value)
        {
            return new JournalEntry(value);
        }
        private JournalEntry(object value)
        {
            this.Timestamp = DateTime.Now;
            this.Value = value;
        }

        public DateTime Timestamp { get; private set; }
        public object Value { get; private set; }
    }

    static Journal()
    {
        JournalEntry.Initialize();
    }

    static JournalEntry CreateEntry(object value)
    {
        return EntryFactory(value);
    }
}
```

This should give you your desired visibility levels without needing to resort on slow reflection or introducing additional classes / interfaces

Share

Improve this answer

Follow

edited Sep 2, 2020 at 8:42

Pang

10.1k ● 146 ● 85 ● 124

answered Nov 3, 2009 at 2:18

Grizzly

20.1k ● 4 ● 64 ● 79

The interface is a good solution, don't know why I didn't think of it... Thanks ! – Thomas Levesque Nov 3, 2009 at 3:22

6 The first approach has a big disadvantage: Other classes can inherit from JournalEntry as well and then call the constructor. – Tim Pohlmann Aug 31, 2015 at 12:16

3 @TimPohlmann you can now make the JournayEntry ctor `private protected` which should provide some defence against inheritors. – mcintyre321 Aug 16, 2018 at 15:50

What is value suppose to represent? – c0dezer019 Jun 9, 2022 at 20:27

33

Make `JournalEntry` a *private nested type*. Any public members will be visible only to the enclosing type.

```
public class Journal
{
    private class JournalEntry
    {
    }
}
```

If you need to make `JournalEntry` objects available to other classes, expose them via a public interface:

```
public interface IJournalEntry
{
}

public class Journal
{
    public IEnumerable<IJournalEntry> Entries
    {
        get { ... }
    }

    private class JournalEntry : IJournalEntry
    {
    }
}
```

Share Improve this answer Follow

answered Nov 3, 2009 at 2:37

Sam Harwell
99.7k ● 22 ● 214 ● 282

14

A simpler approach is to just use an `internal` constructor, but make the caller prove who they are by supplying a reference that **only the legitimate caller** could know (we don't need to be concerned about non-public reflection, because if the caller has access to non-public reflection then we've already lost the fight - they can access a `private` constructor directly); for example:

```
class Outer {
    // don't pass this reference outside of Outer
    private static readonly object token = new object();

    public sealed class Inner {
        // .ctor demands proof of who the caller is
        internal Inner(object token) {
            if (token != Outer.token) {
                throw new InvalidOperationException(
                    "Seriously, don't do that! Or I'll tell!");
```

```
            }
            // ...
        }
    }

    // the outer-class is allowed to create instances...
    private static Inner Create() {
        return new Inner(token);
    }
}
```

Share Improve this answer Follow

answered May 2, 2013 at 13:46

Marc Gravell
1.1m ● 272 ● 2.6k ● 3k

3

In this case you could either:

1. Make the constructor internal - this stops those outside this assembly creating new instances or...
2. Refactor the `JournalEntry` class to use a public interface and make the actual `JournalEntry` class private or internal. The interface can then be exposed for collections while the actual implementation is hidden.

I mentioned internal as a valid modifier above however depending on your requirements, private may be the better suited alternative.

Edit: Sorry I mentioned private constructor but you've already dealt with this point in your question. My apologies for not reading it correctly!

Share Improve this answer Follow

answered Nov 3, 2009 at 2:07

Paul Mason
1,129 ● 8 ● 15

0

For generic nested class =)

I know this is an old question and it has already an accepted answer, nevertheless for those google swimmers who may have a similar scenario to mine this answer may provide some help.

I came across this question for I needed to implement the same feature as the OP. For my first scenario this and this answers worked just fine. Nevertheless I needed also to expose a nested generic class. The problem is that you can not expose a delegate type field (the factory field) with opened generic parameters without making

your own class generic, but obviously this is not what we want, so, here is my solution for such scenario:

```
public class Foo
{
    private static readonly Dictionary<Type, dynamic> _factories = new
Dictionary<Type, dynamic>();

    private static void AddFactory<T>(Func<Boo<T>> factory)
        => _factories[typeof(T)] = factory;

    public void TestMeDude<T>()
    {
        if (!_factories.TryGetValue(typeof(T), out var factory))
        {
            Console.WriteLine("Creating factory");
            RuntimeHelpers.RunClassConstructor(typeof(Boo<T>).TypeHandle);
            factory = _factories[typeof(T)];
        }
        else
        {
            Console.WriteLine("Factory previously created");
        }

        var boo = (Boo<T>)factory();
        boo.ToBeSure();
    }

    public class Boo<T>
    {
        static Boo() => AddFactory(() => new Boo<T>());

        private Boo() { }

        public void ToBeSure() => Console.WriteLine(typeof(T).Name);
    }
}
```

We have **Boo** as our internal nested class with a private constructor and we mantain on our parent class a dictionary with these generic factories taking advantage of **dynamic**. So, each time **TestMeDude** is called, Foo searches for whether the factory for T has already been created, if not it creates it calling nested class' static constructor.

Testing:

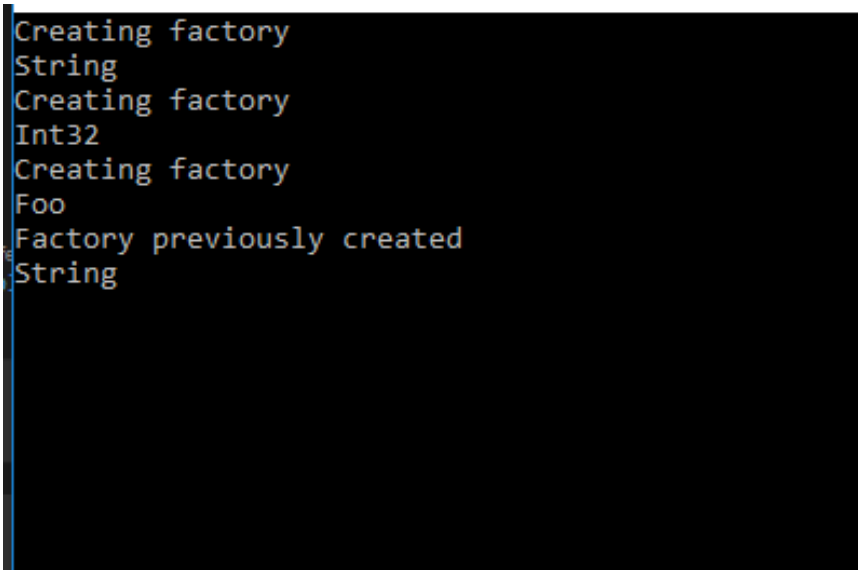
```
private static void Main()
{
    var foo = new Foo();

    foo.TestMeDude<string>();
    foo.TestMeDude<int>();
    foo.TestMeDude<Foo>();

    foo.TestMeDude<string>();
```

```
    Console.ReadLine();
}
```

The output is:



Share Improve this answer Follow

answered Feb 23, 2018 at 16:05

taquion
2,757 ● 2 ● 19 ● 33

0

The solution Grizzly suggested does make it a bit hard to create the nested class somewhere else but not impossible,like Tim Pohlmann wrote someone can still inherit it and use the inheriting class ctor.

I'm taking advantage of the fact that nested class can access the container private properties, so the container asks nicely and the nested class gives access to the ctor.

```
 public class AllowedToEmailFunc
{
    private static Func<long, EmailPermit> CreatePermit;

    public class EmailPermit
    {
        public static void AllowIssuingPermits()
        {
            IssuegPermit = (long userId) =>
            {
                return new EmailPermit(userId);
            };
        }

        public readonly long UserId;

        private EmailPermit(long userId)
        {
```

```
            UserId = userId;
        }
    }

    static AllowedToEmailFunc()
    {
        EmailPermit.AllowIssuingPermits();
    }

    public static bool AllowedToEmail(UserAndConf user)
    {
        var canEmail = true; /// code checking if we can email the user
        if (canEmail)
        {
            return IssuegPermit(user.UserId);
        }
        else
        {
            return null
        }

    }
}
```

This solution is not something I would do on a regular day on the job, not because it will lead to problems in other places but because it's unconventional (I've never seen it before) so it might cause other developers pain .

Share Improve this answer Follow

answered Jul 16, 2018 at 5:08

user1852503

5,557 ● 2 ● 22 ● 29