C# - Can publicly inherited methods be hidden (e.g. made private to derived class)

Asked 16 years, 3 months ago Modified 6 years, 9 months ago Viewed 60k times

74

Suppose I have BaseClass with public methods A and B, and I create DerivedClass through inheritance.

e.g.

```
public DerivedClass : BaseClass {}
```

Now I want to develop a method C in DerivedClass that uses A and B. Is there a way I can override methods A and B to be private in DerivedClass so that only method C is exposed to someone who wants to use my DerivedClass?

c# class derived

Share
Improve this question
Follow

edited Nov 30, 2008 at 5:56

Don Kirkby
56.5k • 27 • 219 • 301

asked Sep 19, 2008 at 23:29

Lyndon
909 • 3 • 8 • 9

10 Answers

Sorted by: Highest score (default)

86

It's not possible, why?

In C#, it is forced upon you that if you inherit public methods, you must make them public. Otherwise they expect you not to derive from the class in the first place.

Instead of using the is-a relationship, you would have to use the has-a relationship.

The language designers don't allow this on purpose so that you use inheritance more properly.

For example one might accidentally confuse a class Car to derive from a class Engine to get it's functionality. But an Engine is functionality that is used by the car. So you would want to use the has-a relationship. The user of the Car does not want to have access to the interface of the Engine. And the Car itself should not confuse the Engine's methods with it's own. Nor Car's future derivations.

So they don't allow it to protect you from bad inheritance hierarchies.

What should you do instead?

Instead you should implement interfaces. This leaves you free to have functionality using the has-a relationship.

Other languages:

In C++ you simply specify a modifier before the base class of private, public or protected. This makes all members of the base that were public to that specified access level. It seems silly to me that you can't do the same in C#.

The restructured code:

```
interface I
{
    void C();
}

class BaseClass
{
    public void A() { MessageBox.Show("A"); }
    public void B() { MessageBox.Show("B"); }
}

class Derived : I
{
    public void C()
    {
        b.A();
        b.B();
    }

    private BaseClass b;
}
```

I understand the names of the above classes are a little moot :)

Other suggestions:

Others have suggested to make A() and B() public and throw exceptions. But this doesn't make a friendly class for people to use and it doesn't really make sense.

Share

Improve this answer

Follow

edited Aug 5, 2013 at 9:10

Daniel Daranas

22.6k ● 9 ● 65 ● 121

answered Sep 19, 2008 at 23:56

Brian R. Bondy

347k ● 126 ● 602 ● 640

> Interfaces are definitely one recommended way to go, but it shouldn't be the only step. You still have to be aware of if your code is violating the Single Responsibility Principle or not.

Even implementing Interfaces doesn't prevent you from writing monolithic classes that are large and unwieldy. – Jason Olson Sep 20, 2008 at 4:56

1 Another reason to not hide public methods is that it may break the interface contracts with consumers. For example, hiding the Remove() method from a base class that implements IList would break any code that expected that method to be present based on the interface. – Peter Gluck Jul 5, 2012 at 17:46

9 Unfortunately, the inability to do this means that when you're using somebody else's badly-designed library, it's impossible to insulate your codebase from it. A great example is the WebControl class in System.Web --- it has tons of public properties that abuse inline styles, and because they rolled all that into the base class, there's no way to create a CleanWebControl class that inherits it and hides the bad design. Hiding somebody else's bad design would be useful, if it was possible, especially when the bad design is in the .NET Framework itself. – Sean Werkema Oct 1, 2012 at 17:37

44

When you, for instance, try to inherit from a `List<object>`, and you want to hide the direct `Add(object _ob)` member:

```
// the only way to hide
[Obsolete("This is not supported in this class.", true)]
public new void Add(object _ob)
{
    throw NotImplementedException("Don't use!!");
}
```

It's not really the most preferable solution, but it does the job. Intellisense still accepts, but at compile time you get an error:

> error CS0619: 'TestConsole.TestClass.Add(TestConsole.TestObject)' is obsolete: 'This is not supported in this class.'

Share

Improve this answer

Follow

edited Jun 27, 2016 at 13:25

Athafoud

3,000 ● 3 ● 42 ● 61

answered Jul 1, 2010 at 10:06

nono

441 ● 4 ● 3

6 Beautiful! This is what I needed. I've got a badly structured bad class that I can't change (owned by someone else) who has got members I need to hide. This works great. Note that C++ lets you do this. – Mark Lakata Oct 1, 2012 at 18:02

Welp, afaik `new` can be (accidentally?) bypassed depending on how your variable is declared so I wouldn't consider this as safe. (`new` generally isn't a good idea) – jeromej Jul 19, 2022 at 15:05

That sounds like a bad idea. Liskov would not be impressed.

If you don't want consumers of DerivedClass to be able to access methods DeriveClass.A() and DerivedClass.B() I would suggest that DerivedClass should implement some public interface IWhateverMethodCIsAbout and the consumers of DerivedClass should actually be talking to IWhateverMethodCIsAbout and know nothing about the implementation of BaseClass or DerivedClass at all.

Share Improve this answer Follow

answered Sep 19, 2008 at 23:35

Hamish Smith

8,181 ● 1 ● 38 ● 49

I did a little research into interfaces and maybe I'm missing something big here, but if I defined the interface, I'd want to implement it using methods A and B anyway. A and B are helper functions to implement larger functions in the derived classes. – Lyndon Sep 19, 2008 at 23:59

Lyndon, hopefully the code in the accepted answer makes it clear that DerivedClass is still a subclass of BaseClass and you can still use function A and B in DerivedClass to implement C but the outside world doesn't get to interact with DerivedClass as a DerivedClass and uses the interface instead – Hamish Smith Sep 20, 2008 at 0:48

What you need is composition not inheritance.

```
class Plane
{
  public Fly() { .. }
  public string GetPilot() {...}
}
```

Now if you need a special kind of Plane, such as one that has PairOfWings = 2 but otherwise does everything a plane can.. You inherit plane. By this you declare that your derivation meets the contract of the base class and can be substituted without blinking wherever a base class is expected. e.g. LogFlight(Plane) would continue to work with a BiPlane instance.

However if you just need the Fly behavior for a new Bird you want to create and are not willing to support the complete base class contract, you compose instead. In this case, refactor the behavior of methods to reuse into a new type Flight. Now create and hold references to this class in both Plane and Bird. You don't inherit because the Bird does not support the complete base class contract... (e.g. it cannot provide GetPilot()).

For the same reason, **you cannot reduce the visibility of base class methods when you override..** you can override and make a base private method public in the derivation but not vice versa. e.g. In this example, if I derive a type of Plane "BadPlane" and then override and "Hide" GetPilot() - make it private; a client method LogFlight(Plane p) will work for most Planes but will blow up for "BadPlane" if the implementation of LogFlight happens to need/call GetPilot(). **Since all derivations of a base class are expected to be 'substitutable' wherever a base class param is expected, this has to be disallowed.**

Share Improve this answer Follow

answered Sep 20, 2008 at 5:18

Gishu

137k ● 47 ● 226 ● 311

1 the need to hide base class methods and properties arises when you don't have control on the source of the base class. Suppose you have to override an entire mechanism of the base class which does not suffice your needs. You make your own methods, override others but also want to hide some which belong to the original mechanism and which might get used by other programmers. You might choose aggregation/wrapping the "base" class, but that also is not always acceptable because it breaks the inheritance chain. Marking such methods as obsolete is the best sollution – Radu Simionescu Nov 17, 2015 at 9:51

5

@Brian R. Bondy pointed me to an interesting article on Hiding through inheritance and the **new** keyword.

http://msdn.microsoft.com/en-us/library/aa691135(VS.71).aspx

So as workaround I would suggest:

```
class BaseClass
{
    public void A()
    {
        Console.WriteLine("BaseClass.A");
    }

    public void B()
    {
        Console.WriteLine("BaseClass.B");
    }
}

class DerivedClass : BaseClass
{
    new public void A()
    {
        throw new NotSupportedException();
    }

    new public void B()
```

```
    {
        throw new NotSupportedException();
    }

    public void C()
    {
        base.A();
        base.B();
    }
}
```

This way code like this will throw a **NotSupportedException**:

```
    DerivedClass d = new DerivedClass();
    d.A();
```

Share

Improve this answer

Follow

edited Dec 22, 2010 at 21:25

Michael Myers ♦

192k ● 47 ● 295 ● 295

answered Sep 19, 2008 at 23:34

Jorge Ferreira

97.8k ● 25 ● 126 ● 134

1 If you use DerivedClass d = new DerivedClass(); d.A() you will still be able to access the base class' A() implementation. – Brian R. Bondy Sep 19, 2008 at 23:41

Jorge, I just tried this and as Brian pointed out, A and B would still be exposed. – Lyndon Sep 19, 2008 at 23:42

1 if you cast a DerivedClass Instance to BaseClass, you will be calling A and B from the base class. It would work better with virtual methods in the base class – Radu Simionescu Nov 17, 2015 at 9:59

3

The only way to do this that I know of is to use a Has-A relationship and only implement the functions you want to expose.

Share Improve this answer Follow

answered Sep 19, 2008 at 23:31

Nescio

28.4k ● 10 ● 55 ● 76

1

Hiding is a pretty slippery slope. The main issues, IMO, are:

- It's dependent upon the design-time declaration type of the instance, meaning if you do something like BaseClass obj = new SubClass(), then call obj.A(), hiding is defeated. BaseClass.A() will be executed.

- Hiding can very easily obscure behavior (or behavior changes) in the base type. This is obviously less of a concern when you own both sides of the equation, or if calling 'base.xxx' is part of your sub-member.
- If you actually *do* own both sides of the base/sub-class equation, then you should be able to devise a more manageable solution than institutionalized hiding/shadowing.

Share Improve this answer Follow

answered Sep 20, 2008 at 3:13

Jared
2,015 ● 15 ● 14

1

I would say that if you have a codebase that you are wanting to do this with, it is not the best designed code base. It's typically a sign of a class in one level of the heirarchy needing a certain public signature while another class derived from that class doesn't need it.

An upcoming coding paradigm is called "Composition over Inheritance." This plays directly off of the principles of object-oriented development (especially the Single Responsibility Principle and Open/Closed Principle).

Unfortunately, the way a lot of us developers were taught object-orientation, we have formed a habit of immediately thinking about inheritance instead of composition. We tend to have larger classes that have many different responsibilities simply because they might be contained with the same "Real World" object. This can lead to class hierarchies that are 5+ levels deep.

An unfortunate side-effect that developers don't normally think about when dealing with inheritance is that inheritance forms one of the strongest forms of dependencies that you can ever introduce into your code. Your derived class is now strongly dependant on the class it was inherited from. This can make your code more brittle in the long run and lead to confounding problems where changing a certain behavior in a base class breaks derived classes in obscure ways.

One way to break your code up is through interfaces like mentioned in another answer. This is a smart thing to do anyways as you want a class's external dependencies to bind to abstractions, not concrete/derived types. This allows you to change the implementation without changing the interface, all without effecting a line of code in your dependent class.

I would much rather than maintain a system with hundreds/thousands/even more classes that are all small and loosely-coupled, than deal with a system that makes heavy use of polymorphism/inheritance and has fewer classes that are more tightly coupled.

Perhaps the **best** resource out there on object-oriented development is Robert C. Martin's book, Agile Software Development, Principles, Patterns, and Practices.

Share Improve this answer Follow

answered Sep 20, 2008 at 4:54

Jason Olson

3,696 ● 2 ● 24 ● 24

0

If they're defined public in the original class, you cannot override them to be private in your derived class. However, you could make the public method throw an exception and implement your own private function.

Edit: Jorge Ferreira is correct.

Share Improve this answer Follow

answered Sep 19, 2008 at 23:31

Serafina Brocious

30.6k ● 12 ● 91 ● 115

0

While the answer to the question is "no", there is one tip I wish to point out for others arriving here (given that the OP was sort of alluding to assembly access by 3rd parties). When others reference an assembly, Visual Studio should be honoring the following attribute so it will not show in intellisense (hidden, but can STILL be called, so beware):

```
[System.ComponentModel.EditorBrowsable(System.ComponentModel.EditorBrowsableStat
```

If you had no other choice, you should be able to use `new` on a method that hides a base type method, return `=> throw new NotSupportedException();`, and combine it with the attribute above.

Another trick depends on NOT inheriting from a base class if possible, where the base has a corresponding interface (such as `IList<T>` for `List<T>`). Implementing interfaces "explicitly" will also hide those methods from intellisense on the class type. For example:

```
public class GoodForNothing: IDisposable
{
    void IDisposable.Dispose() { ... }
}
```

In the case of `var obj = new GoodForNothing()`, the `Dispose()` method will not be available on `obj`. However, it WILL be available to anyone who explicitly type-casts `obj` to `IDisposable`.

In addition, you could also wrap a base type instead of inheriting from it, then hide some methods:

```
public class MyList<T> : IList<T>
{
    List<T> _Items = new List<T>();
    public T this[int index] => _Items[index];
    public int Count => _Items.Count;
    public void Add(T item) => _Items.Add(item);

[System.ComponentModel.EditorBrowsable(System.ComponentModel.EditorBrowsableStat
    void ICollection<T>.Clear() => throw new InvalidOperationException("No you
may not!"); // (hidden)
    /*...etc...*/
}
```

Share

Improve this answer

Follow

edited Feb 27, 2018 at 11:06

answered Feb 27, 2018 at 10:52

James Wilkins

7,357 ● 4 ● 52 ● 78