C# Automatic Properties - Why Do I Have To Write "get; set;"?

Asked 16 years ago Modified 8 years, 1 month ago Viewed 46k times

53

If both get and set are compulsory in C# automatic properties, why do I have to bother specifying "get; set;" at all?

c# c#-3.0 automatic-properties

Share

Improve this question

Follow

edited Dec 19, 2008 at 10:54

AnthonyWJones

189k ● 35 ● 235 ● 307

asked Dec 4, 2008 at 13:10

Ben Aston

55.6k ● 69 ● 218 ● 344

9 Answers

Sorted by: Highest score (default)

68

Because you might want a read-only property:

```
public int Foo { get; private set; }
```

Or Write-only property:

```
public int Foo { private get; set; }
```

Share Improve this answer

Follow

edited Nov 12, 2016 at 19:39

Vivek Nuna

1

answered Dec 4, 2008 at 13:12

Brian Genisio

48.1k ● 16 ● 128 ● 168

3 You can also add all sorts of modifiers, like protected etc. – Tigraine Dec 4, 2008 at 13:15

Yeah, and what other people said... you need a way to distinguish between a field and a property. – Brian Genisio Dec 4, 2008 at 13:19

1 A little side note: there is the concept of readonly fields. The framework will make sure that these are only written once. It is different from private setters or getters which can be written if you have access. – Cristian Libardo Dec 4, 2008 at 13:24

Also, see my answer, it's about more than access levels, the compiler treats fields and properties differently. – Binary Worrier Dec 4, 2008 at 14:08

3 @BrianGenisio These properties are not really read-only, but rather write-privately, which were close enough until now, however in C# 6.0 (Visual Studio 2015) we have real read-only properties (as answered here). Yay! – Robert Synoradzki Dec 5, 2014 at 9:30

58

ERROR: A property or indexer may not be passed as an out or ref parameter

If you didn't specify `{get; set;}` then the compiler wouldn't know if it's a field or a property. This is important becasue while they "look" identical the compiler treats them differently. e.g. Calling "InitAnInt" on the property raises an error.

```
class Test
{
    public int n;
    public int i { get; set; }
    public void InitAnInt(out int p)
    {
        p = 100;
    }
    public Test()
    {
        InitAnInt(out n); // This is OK
        InitAnInt(out i); // ERROR: A property or inde
                          // as an out or ref paramete
    }
}
```

You shouldn't create public fields/Variables on classes, you never know when you'll want to change it to have get & set accessors, and then you don't know what code you're going to break, especially if you have clients that program against your API.

Also you can have different access modifiers for the get & set, e.g. {get; **private** set;} makes the get public and the the set private to the declaring class.

Share Improve this answer Follow

edited Feb 14, 2016 at 20:48

answered Dec 4, 2008 at 14:06

Binary Worrier

51.7k • 20 • 142 • 186

20

Just thought I would share my findings on this topic.

Coding a property like the following, is a .net 3.0 shortcut call “**auto-implemented property**”.

```
public int MyProperty { get; set; }
```

This saves you some typing. The long way to declare a property is like this:

```
private int myProperty;
public int MyProperty
{
  get { return myProperty; }
  set { myProperty = value; }
}
```

When you use the “auto-implemented property” the compiler generates the code to wire up the get and set to some “k_BackingField”. Below is the disassembled code using Reflector.

```
public int MyProperty
{
    [CompilerGenerated]
```

```
    get
    {
        return this.<MyProperty>k__BackingField;
    }
    [CompilerGenerated]
    set
    {
        this.<MyProperty>k__BackingField = value;
    }
}
```

disassembled C# code from IL

Also wires up a method for the setter and getter.

```
[CompilerGenerated]
public void set_MyProperty(int value)
{
    this.<MyProperty>k__BackingField = value;
}
[CompilerGenerated]
public int get_MyProperty()
{
    return this.<MyProperty>k__BackingField;
}
```

disassembled C# code from IL

When you declare a read only auto-implemented property, by setting the setter to private:

```
 public int MyProperty { get; private set; }
```

All the compiler does flag the "**set**" as private. The setter and getter method say the same.

```
public int MyProperty
{
    [CompilerGenerated]
    get
    {
        return this.<MyProperty>k__BackingField;
    }
    private [CompilerGenerated]
    set
    {
        this.<MyProperty>k__BackingField = value;
    }
}
```

disassembled C# code from IL

So I am not sure why the framework require both the get; and set; on an auto-implemented property. They could have just not written the set and setter method if it was not supplied. But there may be some compiler level issue that makes this difficult, I don't know.

If you look at the long way of declaring a read only property:

```
public int myProperty = 0;
public int MyProperty
{
    get { return myProperty; }
}
```

And then look at the disassembled code. The setter is not there at all.

```
public int Test2
{
    get
    {
        return this._test;
    }
}

public int get_Test2()
{
    return this._test;
}
```

disassembled C# code from IL

Share Improve this answer

Follow

answered Dec 4, 2008 at 17:08

Ron Todosichuk

284 ● 1 ● 5

5 The private set method is required with auto-properties because otherwise you'd never be able to set the value to anything, which would be pointless. You can exclude the setter from a non-auto property because the backing field provides a way to change the value internally.
– Jeromy Irvine Dec 4, 2008 at 17:41

Good point, that is correct. Because you do not have a private variable, while using an auto-implemented property, you have know way of setting a value if it was not there.
– Ron Todosichuk Dec 4, 2008 at 20:10

17

Because you need some way to distinguish it from plain fields.

It's also useful to have different access modifiers, e.g.

```
public int MyProperty { get; private set; }
```

Share Improve this answer

Follow

answered Dec 4, 2008 at 13:13

Cristian Libardo

9,248 ● 3 ● 37 ● 41

But public fields are useless. – Daniel Earwicker Dec 4, 2008 at 14:56

Agreed. The syntax can probably be partly attributed to mr anders not wanting to introduce a new keyword in the language. – Cristian Libardo Dec 4, 2008 at 16:40

1 Earwicker: I found public fields useful in my Vector2f class -- they sped the program a lot (compared to properties), and the class is simple enough that I'm never going to need to change the implementation. – Stefan Monov Jan 14, 2010 at 13:13

5

The compiler needs to know if you want it to generate a getter and/or a setter, or perhaps are declaring a field.

Share Improve this answer

Follow

answered Dec 4, 2008 at 13:12

Kris

41.8k ● 9 ● 76 ● 101

If the property didn't have accessors, how would the compiler separate it from a field? And what would

2

separate it from a field?

Share Improve this answer

Follow

answered Dec 4, 2008 at 13:13

Rune Grimstad

36.2k • 10 • 65 • 77

2

Well, obviously you need a way of disambiguating between fields and properties. But are required keywords really necessary? For instance, it's clear that these two declarations are different:

```
public int Foo;
public int Bar { }
```

That could work. That is, it's a syntax that a compiler could conceivably make sense of.

But then you get to a situation where an empty block has semantic meaning. That seems precarious.

Share Improve this answer

Follow

answered Dec 4, 2008 at 22:26

Robert Rossney

96.6k • 24 • 148 • 218

2

Since no one mentioned it... you could make the auto-property virtual and override it:

```
public virtual int Property { get; set; }
```

If there was no get/set, how would it be overridden? Note that you are allowed to override the getter and not the setter:

```
public override int Property { get { return int.MinVal
```

Share Improve this answer

Follow

edited May 23, 2017 at 12:01

Community Bot

1 ● 1

answered Sep 30, 2012 at 0:51

Zaid Masud

13.4k ● 9 ● 69 ● 88

2

Also, because ever since C# 6.0 (in Visual Studio 2015, at the time of this answer available in version Ultimate Preview) you may implement a true read-only property:

```
public string Name { get; }
public string Name { get; } = "This won't change even
```

... as opposed to currently imperfect workaround with public getter/private setter pair:

```
public string Name { get; private set; }

public Constructor() { Name="As initialised"; }
public void Method() { Name="This might be changed int
not."; }
```

Example of the above below (compiled and executable online here).

```
using System;

public class Propertier {
    public string ReadOnlyPlease { get; private set; }

    public Propertier()  { ReadOnlyPlease="As initiali
    public void Method() { ReadOnlyPlease="This might
    public override string ToString() { return String.
[{0}]",ReadOnlyPlease); }
}

public class Program {
    static void Main() {
        Propertier p=new Propertier();
        Console.WriteLine(p);

//      p.ReadOnlyPlease="Changing externally!";
//      Console.WriteLine(p);

        // error CS0272: The property or indexer `Prop
cannot be used in this context because the set accesso
        // That's good and intended.

        // But...
        p.Method();
        Console.WriteLine(p);
    }
}
```

Other tasty news about C# 6.0 available as official preview video here.

Share Improve this answer

Follow

answered Dec 4, 2014 at 10:25

Robert Synoradzki

1,996 ● 18 ● 22