What is the difference between a field and a property?

Asked 11 years, 1 month ago Active 6 months ago Viewed 420k times

1042

In C#, what makes a field different from a property, and when should a field be used instead of a property?

c# properties field

302

edited Feb 18 '18 at 16:08

community wiki
6 revs, 6 users 100%
Anonymous

31 Microsoft directly answers this question (for all .NET languages) as part of its Member Design Guidelines. For specifics see the articles Property Design and Field Design. Note there is a distinction between **instance** members and **static** members. – DavidRR Jan 2 '14 at 19:12

7 Properties, fields, and methods. Oh my! and Why properties matter are two useful posts that explain when you should use properties vs fields. – Steven Wexler May 13 '14 at 22:29

31 Answers

1 2 next

901

Properties expose fields. Fields should (almost always) be kept private to a class and accessed via get and set properties. Properties provide a level of abstraction allowing you to change the fields while not affecting the external way they are accessed by the things that use your class.

```
public class MyClass
{
    // this is a field.  It is private to your class and stores the actual data.
    private string _myField;

    // this is a property. When accessed it uses the underlying field,
    // but only exposes the contract, which will not be affected by the underlying field
    public string MyProperty
    {
```

```
        get
        {
            return _myField;
        }
        set
        {
            _myField = value;
        }
    }

    // This is an AutoProperty (C# 3.0 and higher) - which is a shorthand syntax
    // used to generate a private field for you
    public int AnotherProperty{get;set;}
}
```

@Kent points out that Properties are not required to encapsulate fields, they could do a calculation on other fields, or serve other purposes.

@GSS points out that you can also do other logic, such as validation, when a property is accessed, another useful feature.

edited Feb 25 '17 at 20:21

community wiki
8 revs, 4 users 66%
Cory

175 It's worth noting that properties are not required to encapsulate fields. There could be no field at all behind the property. It might be a calculation or return a constant or whatever. – Kent Boogaart Nov 17 '08 at 10:18

8 "while not affecting the external way they are accessed by the things that use your class." forgive me if I'm incorrectly understanding, then, but why the need for access modifiers in front of properties at all, if the field behind it seems to handle this? i.e. why make a property anything other than public? – Chucky Jun 20 '14 at 11:01

15 Your answer was right before the edits and oddly-upvoted incorrect comments. A property should always encapsulate one or more fields, and should never do any heavy lifting or validation. If you need a property such a UserName or Password to have validation, change their type from strings to Value Objects. There is an unspoken contract between a class-creator and the consumer. Fields hold state, Properties expose state using one or more fields, Voids change state (heavy lifting), and Functions perform queries(heavy lifting).This is not stone, just loose expectations. – Suamere Nov 6 '15 at 16:26

6 @jpaugh If I am a class consumer, I follow contracts set by the class creator. If a property is `string`, my contract is: assign any chars up to ~2bil length. If a property is `DateTime`, my contract is: assign any numbers within the limits of DateTime, which I can look up. If the creator adds constraints to the setters, those constraints are not communicated. But if, instead, the creator changes the type from `string` to `Surname`, then their new Surname class communicates the constraints, and the property `public Surname LastName` doesn't have setter validation. Also, `Surname` is reusable. – Suamere Feb 3 '17 at 16:27

4 And since `Surname`, in my example, is reusable, you don't need to worry about later on copy/pasting those validations in a property setter to other places in code. Nor wondering if the validation of a Surname is in multiple places if you ever make changes to the business rules for Surnames. Check

out the link I posted about Value Objects – Suamere Feb 3 '17 at 16:29

245

Object orientated programming principles say that, the internal workings of a class should be hidden from the outside world. If you expose a field you're in essence exposing the internal implementation of the class. Therefore we wrap fields with Properties (or methods in Java's case) to give us the ability to change the implementation without breaking code depending on us. Seeing as we can put logic in the Property also allows us to perform validation logic etc if we need it. C# 3 has the possibly confusing notion of autoproperties. This allows us to simply define the Property and the C#3 compiler will generate the private field for us.

```
public class Person
{
   private string _name;

   public string Name
   {
      get
      {
         return _name;
      }
      set
      {
         _name = value;
      }
   }
   public int Age{get;set;} //AutoProperty generates private field for us
}
```

answered Nov 17 '08 at 9:05 community wiki danswain

80 +1 for mentioning autoproperties - I think this is something many of the answers here (and elsewhere) have forgotten to bring in. Without this explanation, it can still be pretty hard to grasp what `public int myVar { get; set; }` really stands for (and I presume that it's the reason for at least 50% of the hits this question gets). – Priidu Neemre Oct 24 '13 at 8:48

6 +1 also for mentioning auto, and mentioning how it works ("AutoProperty generates private field for us") This was the answer I've been looking for to a question I had. When researching I didn't see on MSDN's page about them any indication that a private field was created and was causing confusion. I guess that's what this means? "Attributes are permitted on auto-implemented properties but obviously not on the backing fields since those are not accessible from your source code. If you must use an attribute on the backing field of a property, just create a regular property." but wasn't sure. – Nyra Jul 18 '14 at 14:05

2 Note that the given example doesn't encapsulate squat. This property gives 100 % full access to the private field, so this isn't object-oriented at all. You

might as well have a public field in this case. Granted, it helps (marginally) to refactor code in the future, but any IDE worth it's mettle can transform a field to a property with a few keystrokes. The answer might be technically correct about how properties work, but it does not give a good "OOP explanation" to their uses. – sara Feb 1 '16 at 16:05

2 @kai I agree that the answer over-simplified things and is not showing all the power of an auto-property, however I disagree that this is not object-oriented. You may want to check the difference between fields and properties. Fields cannot be virtual, and `virtual` itself is part of object-oriented programming. – Gobe Feb 9 '16 at 0:12

160

An important difference is that interfaces can have properties but not fields. This, to me, underlines that properties should be used to define a class's public interface while fields are meant to be used in the private, internal workings of a class. As a rule I rarely create public fields and similarly I rarely create non-public properties.

answered Nov 17 '08 at 13:44

community wiki
Hans Løken

95

I'll give you a couple examples of using properties that might get the gears turning:

- **Lazy Initialization:** If you have a property of an object that's expensive to load, but isn't accessed all that much in normal runs of the code, you can delay its loading via the property. That way, it's just sitting there, but the first time another module tries to call that property, it checks if the underlying field is null - if it is, it goes ahead and loads it, unknown to the calling module. This can greatly speed up object initialization.
- **Dirty Tracking:** Which I actually learned about from my own question here on StackOverflow. When I have a lot of objects which values might have changed during a run, I can use the property to track if they need to be saved back to the database or not. If not a single property of an object has changed, the IsDirty flag won't get tripped, and therefore the saving functionality will skip over it when deciding what needs to get back to the database.

edited Jul 5 '18 at 4:13

community wiki
4 revs, 3 users 78%
Chris

1 A question about dirty tracking: what if I could change the field directly– I don't know if that can be done, I could say: "the object does not need to be saved if not a single FIELD of an object has changed" thus dirty tracking would not be a difference, am I missing something? – juanpastas Dec 10 '12 at 17:29

2 @juanpastas: The advantage of properties with regard to dirty tracking is that if property setters will set a "dirty" flag, then in the scenario where the flag isn't set code won't have to inspect the values of any properties to see if they might have changed. By contrast, if an object exposes its attributes as

fields, then the contents of all fields must be compared against the previous value (which not only adds time to do the comparison, but also means the code must *have* the previous value). – supercat May 7 '13 at 6:31

49

Using Properties, you can throw an event, when the value of the property is changed (aka. PropertyChangedEvent) or before the value is changed to support cancelation.

This is not possible with (direct access to) fields.

```
public class Person {
 private string _name;

 public event EventHandler NameChanging;
 public event EventHandler NameChanged;

 public string Name{
  get
  {
     return _name;
  }
  set
  {
     OnNameChanging();
     _name = value;
     OnNameChanged();
  }
 }

 private void OnNameChanging(){
   EventHandler localEvent = NameChanging;
   if (localEvent != null) {
     localEvent(this,EventArgs.Empty);
   }
 }

 private void OnNameChanged(){
   EventHandler localEvent = NameChanged;
   if (localEvent != null) {
     localEvent(this,EventArgs.Empty);
   }
 }
}
```

edited Jun 26 '09 at 6:38 community wiki

2 revs
Jehof

3 I took a long time to find this. This is a MVVM. Thank you ! :) – user5039044 Oct 26 '16 at 9:48

43

Since many of them have explained with technical pros and cons of `Properties` and `Field` , it's time to get into real time examples.

1. Properties allows you to set the read-only access level

Consider the case of `dataTable.Rows.Count` and `dataTable.Columns[i].Caption` . They come from the class `DataTable` and both are public to us. The difference in the access-level to them is that we cannot set value to `dataTable.Rows.Count` but we can read and write to `dataTable.Columns[i].Caption` . Is that possible through `Field` ? No!!! This can be done with `Properties` only.

```
public class DataTable
{
    public class Rows
    {
       private string _count;

       // This Count will be accessable to us but have used only "get" ie, readonly
       public int Count
       {
           get
           {
              return _count;
           }
       }
    }

    public class Columns
    {
        private string _caption;

        // Used both "get" and "set" ie, readable and writable
        public string Caption
        {
           get
           {
              return _caption;
           }
           set
           {
              _caption = value;
```

```
            }
        }
    }
}
```

2. Properties in PropertyGrid

You might have worked with `Button` in Visual Studio. Its properties are shown in the `PropertyGrid` like `Text` , `Name` etc. When we drag and drop a button, and when we click the properties, it will automatically find the class `Button` and filters `Properties` and show that in `PropertyGrid` (where `PropertyGrid` won't show `Field` even though they are public).

```
public class Button
{
    private string _text;
    private string _name;
    private string _someProperty;

    public string Text
    {
        get
        {
           return _text;
        }
        set
        {
           _text = value;
        }
   }

   public string Name
   {
        get
        {
           return _name;
        }
        set
        {
           _name = value;
        }
   }

   [Browsable(false)]
   public string SomeProperty
   {
        get
        {
```

```
            return _someProperty;
        }
        set
        {
            _someProperty= value;
        }
    }
```

In `PropertyGrid` , the properties `Name` and `Text` will be shown, but not `SomeProperty` . Why??? Because Properties can accept Attributes. It does not show in case where `[Browsable(false)]` is false.

3. Can execute statements inside Properties

```
public class Rows
{
    private string _count;


    public int Count
    {
        get
        {
            return CalculateNoOfRows();
        }
    }

    public int CalculateNoOfRows()
    {
         // Calculation here and finally set the value to _count
         return _count;
    }
}
```

4. Only Properties can be used in Binding Source

Binding Source helps us to decrease the number of lines of code. `Fields` are not accepted by `BindingSource` . We should use `Properties` for that.

5. Debugging mode

Consider we are using `Field` to hold a value. At some point we need to debug and check where the value is getting null for that field. It will be difficult to do where the number of lines of code are more than 1000. In such situations we can use `Property` and can set debug mode inside `Property` .

```
public string Name
{
     // Can set debug mode inside get or set
     get
     {
        return _name;
     }
     set
     {
        _name = value;
     }
}
```

edited Jul 5 '18 at 4:13

community wiki
7 revs, 4 users 95%
Sarath Avanavu

2 Read my first sentence in my answer. I have specifically told that I am not going to repeat everything again here. That makes no sense!!! The people will first look at the description first, then the examples. The marked answer gives the description well, but I added with some real-time scenarios and examples which makes sense. Please make sure you think from reader's point of view before commenting @Dawid Ferenczy – Sarath Avanavu Jul 4 '15 at 13:02

1 I have read it, but you didn't read my previous comment obviously: "*But you noticed, that you're just providing an usage examples, since difference between fields and properties was already described, so forgot my comment, please :)*". – Dawid Ferenczy Rogožan Jul 15 '15 at 17:29

DIFFERENCES - USES (when and why)

31

A **field** is a variable that is declared directly in a class or struct. A class or struct may have instance fields or static fields or both. Generally, you should use fields *only for variables that have private or protected accessibility*. Data that your class exposes to client code *should be provided through methods, properties* and indexers. By using these constructs for indirect access to internal fields, you can guard against invalid input values.

A **property** is a member that provides a flexible mechanism to read, write, or compute the value of a private field. Properties can be used as if they are public data members, but they are actually special methods called *accessors*. This enables data to be accessed easily and still helps promote the *safety and flexibility of methods*. Properties enable a class to expose a public way of getting and setting values, while hiding implementation or verification code. A get property accessor is used to return the property value, and a set accessor is used to assign a new value.

answered Sep 12 '13 at 14:18

community wiki

makiSTB

12

Properties have the primary advantage of allowing you to change the way data on an object is accessed without breaking it's public interface. For example, if you need to add extra validation, or to change a stored field into a calculated you can do so easily if you initially exposed the field as a property. If you just exposed a field directly, then you would have to change the public interface of your class to add the new functionality. That change would break existing clients, requiring them to be recompiled before they could use the new version of your code.

If you write a class library designed for wide consumption (like the .NET Framework, which is used by millions of people), that can be a problem. However, if you are writing a class used internally inside a small code base (say <= 50 K lines), it's really not a big deal, because no one would be adversely affected by your changes. In that case it really just comes down to personal preference.

answered Nov 17 '08 at 10:09

community wiki
Scott Wisniewski

11

Properties support asymmetric access, i.e. you can have either a getter and a setter or just one of the two. Similarly properties support individual accessibility for getter/setter. Fields are always symmetric, i.e. you can always both get and set the value. Exception to this is readonly fields which obviously cannot be set after initialization.

Properties may run for a very long time, have side effects, and may even throw exceptions. Fields are fast, with no side effects, and will never throw exceptions. Due to side effects a property may return a different value for each call (as may be the case for DateTime.Now, i.e. DateTime.Now is not always equal to DateTime.Now). Fields always return the same value.

Fields may be used for out / ref parameters, properties may not. Properties support additional logic – this could be used to implement lazy loading among other things.

Properties support a level of abstraction by encapsulating whatever it means to get/set the value.

Use properties in most / all cases, but try to avoid side effects.

answered Nov 17 '08 at 11:40

community wiki
Brian Rasmussen

1 Properties should never have side effects. Even the debugger assumes it can evaluate them safely. – Craig Gidney Jun 26 '09 at 6:44

11

In the background a property is compiled into methods. So a `Name` property is compiled into `get_Name()` and `set_Name(string value)`. You can see this if you study the compiled code. So there is a (very) small performance overhead when using them. Normally you will always use a Property if you expose a field to the outside, and you will often use it internally if you need to do validation of the value.

edited Jul 25 '12 at 8:37

community wiki
2 revs, 2 users 67%
Rune Grimstad

7

When you want your private variable(field) to be accessible to object of your class from other classes you need to create properties for those variables.

for example if I have variables named as "id" and "name" which is private but there might be situation where this variable needed for read/write operation outside of the class. At that situation , property can help me to get that variable to read/write depending upon the get/set defined for the property. A property can be a readonly / writeonly / readwrite both.

here is the demo

```
class Employee
{
    // Private Fields for Employee
    private int id;
    private string name;

    //Property for id variable/field
    public int EmployeeId
    {
       get
       {
          return id;
       }
       set
       {
          id = value;
       }
    }

    //Property for name variable/field
```

```
    public string EmployeeName
    {
       get
       {
          return name;
       }
       set
       {
          name = value;
       }
    }
}

class MyMain
{
    public static void Main(string [] args)
    {
       Employee aEmployee = new Employee();
       aEmployee.EmployeeId = 101;
       aEmployee.EmployeeName = "Sundaran S";
    }
}
```

edited Dec 29 '13 at 7:21

community wiki
2 revs, 2 users 98%
Petryanu

6

The second question here, "when should a field be used instead of a property?", is only briefly touched on in this other answer and kinda this one too, but not really much detail.

In general, all the other answers are spot-on about good design: prefer exposing properties over exposing fields. While you probably won't **regularly** find yourself saying "wow, imagine how much worse things would be if I had made this a field instead of a property", it's **so much** more rare to think of a situation where you would say "wow, thank God I used a field here instead of a property."

But there's one advantage that fields have over properties, and that's their ability to be used as "ref" / "out" parameters. Suppose you have a method with the following signature:

```
public void TransformPoint(ref double x, ref double y);
```

and suppose that you want to use that method to transform an array created like this:

```
System.Windows.Point[] points = new Point[1000000];
Initialize(points);
```

Here's I think the fastest way to do it, since X and Y are properties:

```
for (int i = 0; i < points.Length; i++)
{
    double x = points[i].X;
    double y = points[i].Y;
    TransformPoint(ref x, ref y);
    points[i].X = x;
    points[i].Y = y;
}
```

And that's going to be pretty good! Unless you have measurements that prove otherwise, there's no reason to throw a stink. But I believe it's not technically guaranteed to be as fast as this:

```
internal struct MyPoint
{
    internal double X;
    internal double Y;
}

// ...

MyPoint[] points = new MyPoint[1000000];
Initialize(points);

// ...

for (int i = 0; i < points.Length; i++)
{
    TransformPoint(ref points[i].X, ref points[i].Y);
}
```

Doing some measurements myself, the version with fields takes about 61% of the time as the version with properties (.NET 4.6, Windows 7, x64, release mode, no debugger attached). The more expensive the `TransformPoint` method gets, the less pronounced that the difference becomes. To repeat this yourself, run with the first line commented-out and with it not commented-out.

Even if there were no performance benefits for the above, there are other places where being able to use ref and out parameters might be beneficial, such as when calling the Interlocked or Volatile family of methods. *Note: In case this is new to you, Volatile is basically a*

way to get at the same behavior provided by the `volatile` keyword. As such, like `volatile`, it doesn't magically solve all thread-safety woes like its name suggests that it might.

I definitely don't want to seem like I'm advocating that you go "oh, I should start exposing fields instead of properties." The point is that if you need to regularly use these members in calls that take "ref" or "out" parameters, especially on something that might be a simple value type that's unlikely to ever need any of the value-added elements of properties, an argument can be made.

edited May 23 '17 at 11:47

community wiki
5 revs
Joe Amenta

4

Also, properties allow you to use logic when setting values.

So you can say you only want to set a value to an integer field, if the value is greater than x, otherwise throw an exception.

Really useful feature.

answered Nov 17 '08 at 10:46

community wiki
GurdeepS

4

If you are going to use thread primitives you are forced to use fields. Properties can break your threaded code. Apart from that, what cory said is correct.

edited Nov 24 '08 at 7:34

community wiki
2 revs
Jonathan C Dickinson

1 since when? lock your backing field within the property and it's the equivilant – Sekhat Nov 17 '08 at 14:42

1 Properties are methods, and are not inlined by any CIL JIT today. If you are going to use thread primitives like Interlocked you need to have fields. Check your sources. Admittedly 'locking' was the wrong word to use. – Jonathan C Dickinson Nov 24 '08 at 7:34

(This should really be a comment, but I can't post a comment, so please excuse if it is not appropriate as a post).

4

I once worked at a place where the recommended practice was to use public fields instead of properties when the equivalent property def would just have been accessing a field, as in :

```
get { return _afield; }
set { _afield = value; }
```

Their reasoning was that the public field could be converted into a property later in future if required. It seemed a little strange to me at the time. Judging by these posts, it looks like not many here would agree either. What might you have said to try to change things ?

Edit : I should add that all of the code base at this place was compiled at the same time, so they might have thought that changing the public interface of classes (by changing a public field to a property) was not a problem.

edited Jun 26 '09 at 6:20

community wiki
2 revs
Moe Sisko

4

Technically, i don't think that there is a difference, because properties are just wrappers around fields created by the user or automatically created by the compiler.The purpose of properties is to enforce encapsuation and to offer a lightweight method-like feature. It's just a bad practice to declare fields as public, but it does not have any issues.

edited Nov 1 '13 at 8:04

community wiki
2 revs
Brahim Boulkriat

4

Fields are **ordinary member variables** or member instances of a class. Properties are an **abstraction to get and set their values**. Properties are also called accessors because they offer a way to change and retrieve a field if you expose a field in the class as private. Generally, you should declare your member variables private, then declare or define properties for them.

```
class SomeClass
{
   int numbera; //Field

   //Property
  public static int numbera { get; set;}

}
```

answered Aug 10 '15 at 20:03 community wiki Vasim Shaikh

3

Properties encapsulate fields, thus enabling you to perform additional processing on the value to be set or retrieved. It is typically overkill to use properties if you will not be doing any pre- or postprocessing on the field value.

answered Nov 17 '08 at 10:05 community wiki Erik Burger

1 no, I always use properties, it allows you the flexibility of changing the implementation at anytime without breaking your API. – Sekhat Nov 17 '08 at 14:39

3

IMO, Properties are just the "SetXXX()" "GetXXX()" functions/methods/interfaces pairs we used before, but they are more concise and elegant.

answered Dec 16 '13 at 15:03 community wiki Junchao Xu

3

Traditionally private fields are set via getter and setter methods. For the sake of less code you can use properties to set fields instead.

answered Apr 1 '15 at 1:30 community wiki Chris Paine

3

when you have a class which is "Car". The properties are color,shape..

Where as fields are variables defined within the scope of a class.

answered Apr 1 '15 at 3:09

community wiki
user406580

3

From Wikipedia -- Object-oriented programming:

> Object-oriented programming (OOP) is a programming paradigm based on the concept of "objects", which are data structures that contain **data, in the form of fields**, often known as attributes; and **code, in the form of procedures, often known as methods**. *(emphasis added)*

Properties are actually part of an object's behavior, but are designed to give consumers of the object the illusion/abstraction of working with the object's data.

answered Aug 26 '15 at 10:58

community wiki
Zev Spitz

3

My design of a field is that a field needs to be modified only by its parent, hence the class. Result the variable becomes private, then to be able to give the right to read the classes / methods outside I go through the system of property with only the Get. The field is then retrieved by the property and read-only! If you want to modify it you have to go through methods (for example the constructor) and I find that thanks to this way of making you secure, we have better control over our code because we "flange". One could very well always put everything in public so every possible case, the notion of variables / methods / classes etc ... in my opinion is just an aid to the development, maintenance of the code. For example, if a person resumes a code with public fields, he can do anything and therefore things "illogical" in relation to the objective, the logic of why the code was written. It's my point of view.

When i use a classic model private field / public readonly properties,for 10 privates fields i should write 10 publics properties! The code can be really big faster. I discover the private setter and now i only use public properties with a private setter. The setter create in background a private field.

That why my old classic programming style was:

```
public class MyClass
{
 private int _id;
 public int ID { get { return _id; } }
 public MyClass(int id)
 {
```

```
  _id = id;
 }
}
```

My new programming style:

```
public class MyClass
{
 public int ID { get; private set; }
 public MyClass(int id)
 {
  ID = id;
 }
}
```

edited Dec 9 '16 at 16:59

community wiki
3 revs
Tony Pinot

3

Fields are the variables in classes. Fields are the data which you can encapsulate through the use of access modifiers.

Properties are similar to Fields in that they define states and the data associated with an object.

Unlike a field a property has a special syntax that controls how a person reads the data and writes the data, these are known as the get and set operators. The set logic can often be used to do validation.

edited Oct 24 '18 at 8:15

community wiki
2 revs, 2 users 55%
Neil Meyer

3

Though fields and properties look to be similar to each other, they are 2 completely different language elements.

1. **Fields are the only mechanism how to store data on class level.** Fields are conceptually variables at class scope. If you want to store some data to instances of your classes (objects) you need to use fields. There is no other choice. Properties can't store any data even though, it may look they are able to do so. See bellow.
2. **Properties on the other hand never store data.** They are just the pairs of methods (get and set) that can be syntactically called in a similar way as fields and in most cases they access (for read or write) fields, which is the source of some confusion. But because

property methods are (with some limitations like fixed prototype) regular C# methods they can do whatever regular methods can do. It means they can have 1000 lines of code, they can throw exceptions, call another methods, can be even virtual, abstract or overridden. What makes properties special, is the fact that C# compiler stores some extra metadata into assemblies that can be used to search for specific properties - widely used feature.

Get and set property methods has the following prototypes.

```
PROPERTY_TYPE get();

void set(PROPERTY_TYPE value);
```

So it means that properties can be 'emulated' by defining a field and 2 corresponding methods.

```
class PropertyEmulation
{
    private string MSomeValue;

    public string GetSomeValue()
    {
        return(MSomeValue);
    }

    public void SetSomeValue(string value)
    {
        MSomeValue=value;
    }
}
```

Such property emulation is typical for programming languages that don't support properties - like standard C++. In C# there you should always prefer properties as the way how to access to your fields.

Because only the fields can store a data, it means that more fields class contains, more memory objects of such class will consume. On the other hand, adding new properties into a class doesn't make objects of such class bigger. Here is the example.

```
class OneHundredFields
{
        public int Field1;
        public int Field2;
        ...
        public int Field100;
}

OneHundredFields Instance=new OneHundredFields() // Variable 'Instance' consumes
```

```
100*sizeof(int) bytes of memory.

class OneHundredProperties
{
    public int Property1
    {
        get
        {
            return(1000);
        }
        set
        {
            // Empty.
        }
    }

    public int Property2
    {
        get
        {
            return(1000);
        }
        set
        {
            // Empty.
        }
    }

    ...

    public int Property100
    {
        get
        {
            return(1000);
        }
        set
        {
            // Empty.
        }
    }
}

OneHundredProperties Instance=new OneHundredProperties() // !!!!! Variable 'Instance'
consumes 0 bytes of memory. (In fact a some bytes are consumed becasue every object
contais some auxiliarity data, but size doesn't depend on number of properties).
```

Though property methods can do anything, in most cases they serve as a way how to access objects' fields. If you want to make a field accessible to other classes you can do by 2 ways.

1. Making fields as public - not advisable.
2. Using properties.

Here is a class using public fields.

```
class Name
{
    public string FullName;
    public int YearOfBirth;
    public int Age;
}

Name name=new Name();

name.FullName="Tim Anderson";
name.YearOfBirth=1979;
name.Age=40;
```

While the code is perfectly valid, from design point of view, it has several drawbacks. Because fields can be both read and written, you can't prevent user from writing to fields. You can apply `readonly` keyword, but in this way, you have to initialize readonly fields only in constructor. What's more, nothing prevents you to store invalid values into your fields.

```
name.FullName=null;
name.YearOfBirth=2200;
name.Age=-140;
```

The code is valid, all assignments will be executed though they are illogical. `Age` has a negative value, `YearOfBirth` is far in future and doesn't correspond to Age and `FullName` is null. With fields you can't prevent users of `class Name` to make such mistakes.

Here is a code with properties that fixes these issues.

```
class Name
{
    private string MFullName="";
    private int MYearOfBirth;

    public string FullName
    {
        get
```

```
        {
            return(MFullName);
        }
        set
        {
            if (value==null)
            {
                throw(new InvalidOperationException("Error !"));
            }

            MFullName=value;
        }
    }

    public int YearOfBirth
    {
        get
        {
            return(MYearOfBirth);
        }
        set
        {
            if (MYearOfBirth<1900 || MYearOfBirth>DateTime.Now.Year)
            {
                throw(new InvalidOperationException("Error !"));
            }

            MYearOfBirth=value;
        }
    }

    public int Age
    {
        get
        {
            return(DateTime.Now.Year-MYearOfBirth);
        }
    }

    public string FullNameInUppercase
    {
        get
        {
            return(MFullName.ToUpper());
        }
    }
}
```

The updated version of class has the following advantages.

1. `FullName` and `YearOfBirth` are checked for invalid values.
2. `Age` is not writtable. It's callculated from `YearOfBirth` and current year.
3. A new property `FullNameInUppercase` converts `FullName` to UPPER CASE. This is a little contrived example of property usage, where properties are commonly used to present field values in the format that is more appropriate for user - for instance using current locale on specific numeric of `DateTime` format.

Beside this, properties can be defined as virtual or overridden - simply because they are regular .NET methods. The same rules applies for such property methods as for regular methods.

C# also supports indexers which are the properties that have an index parameter in property methods. Here is the example.

```
class MyList
{
    private string[]                MBuffer;

    public MyList()
    {
        MBuffer=new string[100];
    }

    public string this[int Index]
    {
        get
        {
            return(MBuffer[Index]);
        }
        set
        {
            MBuffer[Index]=value;
        }
    }
}

MyList   List=new MyList();

List[10]="ABC";
Console.WriteLine(List[10]);
```

Since C# 3.0 allows you to define automatic properties. Here is the example.

```
class AutoProps
{
    public int Value1
    {
        get;
        set;
    }

    public int Value2
    {
        get;
        set;
    }
}
```

Even though `class AutoProps` contains only properties (or it looks like), it can store 2 values and size of objects of this class is equal to `sizeof(Value1)+sizeof(Value2)` =4+4=8 bytes.

The reason for this is simple. When you define an automatic property, C# compiler generates automatic code that contains hidden field and a property with property methods accessing this hidden field. Here is the code compiler produces.

Here is a code generated by the **ILSpy** from compiled assembly. Class contains generated hidden fields and properties.

```
internal class AutoProps
{
    [CompilerGenerated]
    [DebuggerBrowsable(DebuggerBrowsableState.Never)]
    private int <Value1>k__BackingField;

    [CompilerGenerated]
    [DebuggerBrowsable(DebuggerBrowsableState.Never)]
    private int <Value2>k__BackingField;

    public int Value1
    {
        [CompilerGenerated]
        get
        {
            return <Value1>k__BackingField;
        }
        [CompilerGenerated]
        set
        {
            <Value1>k__BackingField = value;
        }
```

```
    }

    public int Value2
    {
        [CompilerGenerated]
        get
        {
            return <Value2>k__BackingField;
        }
        [CompilerGenerated]
        set
        {
            <Value2>k__BackingField = value;
        }
    }
}
```

So, as you can see, the compiler still uses the fields to store the values - since fields are the only way how to store values into objects.

So as you can see, though properties and fields have similar usage syntax they are very different concepts. Even if you use automatic properties or events - hidden fields are generated by compiler where the real data are stored.

If you need to make a field value accessible to the outside world (users of your class) don't use public or protected fields. Fields always should be marked as private. Properties allow you to make value checks, formatting, conversions etc. and generally make your code safer, more readable and more extensible for future modifications.

answered Jun 26 at 13:24 community wiki Timmy_A

2

Properties are special kind of class member, In properties we use a predefined Set or Get method.They use accessors through which we can read, written or change the values of the private fields.

For example, let us take a class named `Employee` , with private fields for name, age and Employee_Id. We cannot access these fields from outside the class , but we can access these private fields through properties.

Why do we use properties?

Making the class field public & exposing it is risky, as you will not have control what gets assigned & returned.

To understand this clearly with an example lets take a student class who have ID, passmark, name. Now in this example some problem with public field

- ID should not be -ve.
- Name can not be set to null
- Pass mark should be read only.
- If student name is missing No Name should be return.

To remove this problem We use Get and set method.

```
// A simple example
public class student
{
    public int ID;
    public int passmark;
    public string name;
}

public class Program
{
    public static void Main(string[] args)
    {
       student s1 = new student();
       s1.ID = -101; // here ID can't be -ve
       s1.Name = null ; // here Name can't be null
    }
}
```

Now we take an example of get and set method

```
public class student
{
    private int _ID;
    private int _passmark;
    private string_name ;
    // for id property
    public void SetID(int ID)
    {
        if(ID<=0)
        {
            throw new exception("student ID should be greater then 0");
        }
        this._ID = ID;
```

```
    }
    public int getID()
    {
        return_ID;
    }
}
public class programme
{
    public static void main()
    {
        student s1 = new student ();
        s1.SetID(101);
    }
    // Like this we also can use for Name property
    public void SetName(string Name)
    {
        if(string.IsNullOrEmpty(Name))
        {
            throw new exeception("name can not be null");
        }
        this._Name = Name;
    }
    public string GetName()
    {
        if( string.IsNullOrEmpty(This.Name))
        {
            return "No Name";
        }
        else
        {
            return this._name;
        }
    }
        // Like this we also can use for Passmark property
    public int Getpassmark()
    {
        return this._passmark;
    }
}
```

edited Aug 18 '16 at 17:52

community wiki
2 revs, 2 users 76%
ahmed alkhatib

Additional info: By default, get and set accessors are as accessible as the property itself. You can control/restrict accessor accessibility

2

individually (for get and set) by applying more restrictive access modifiers on them.

Example:

```
public string Name
{
    get
    {
        return name;
    }
    protected set
    {
        name = value;
    }
}
```

Here get is still publicly accessed (as the property is public), but set is protected (a more restricted access specifier).

edited Feb 5 '17 at 13:32

community wiki
2 revs, 2 users 88%
code_doctor

2

Properties are used to expose field. They use accessors(set, get) through which the values of the private fields can be read, written or manipulated.

Properties do not name the storage locations. Instead, they have accessors that read, write, or compute their values.

Using properties we can set validation on the type of data that is set on a field.

For example we have private integer field age on that we should allow positive values since age cannot be negative.

We can do this in two ways using getter and setters and using property.

```
Using Getter and Setter

   // field
   private int _age;

   // setter
   public void set(int age){
     if (age <=0)
```

```
        throw new Exception();

      this._age = age;
    }

    // getter
    public int get (){
      return this._age;
    }
```

Now using property we can do the same thing. In the value is a key word

```
    private int _age;

    public int Age{
    get{
        return this._age;
    }

    set{
       if (value <= 0)
         throw new Exception()
       }
    }
```

Auto Implemented property if we don't logic in get and set accessors we can use auto implemented property.

When u**se auto-implemented property compiles creates a private, anonymous field** that can only be accessed through get and set accessors.

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

Abstract Properties An abstract class may have an abstract property, which should be implemented in the derived class

```
public abstract class Person
   {
      public abstract string Name
      {
         get;
         set;
      }
      public abstract int Age
      {
         get;
         set;
```

```
        }
    }

// overriden something like this
// Declare a Name property of type string:
  public override string Name
  {
     get
     {
        return name;
     }
     set
     {
        name = value;
     }
  }
```

We can privately set a property In this we can privately set the auto property(set with in the class)

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

You can achieve same with this code. In this property set feature is not available as we have to set value to field directly.

```
private int myProperty;
public int MyProperty
{
    get { return myProperty; }
}
```

edited Jun 14 '17 at 17:50

community wiki
2 revs
Nayas Subramanian

2

Think about it : You have a room and a door to enter this room. If you want to check how who is coming in and secure your room, then you should use properties otherwise they won't be any door and every one easily come in w/o any regulation

```
class Room {
   public string sectionOne;
   public string sectionTwo;
}

Room r = new Room();
r.sectionOne = "enter";
```

People is getting in to sectionOne pretty easily, there wasn't any checking

```
class Room
{
   private string sectionOne;
   private string sectionTwo;

   public string SectionOne
   {
      get
      {
        return sectionOne;
      }
      set
      {
        sectionOne = Check(value);
      }
   }
}

Room r = new Room();
r.SectionOne = "enter";
```

Now you checked the person and know about whether he has something evil with him

answered Jun 26 '17 at 14:48 community wiki user8080469

2

The vast majority of cases it's going to be a property name that you access as opposed to a variable name (**field**) The reason for that is it's considered good practice in .NET and in C# in particular to protect every piece of data within a class, whether it's an instance variable or a static variable (class variable) because it's associated with a class.

Protect all of those variables with corresponding properties which allow you to define, **set and get** accessors and do things like validation when you're manipulating those pieces of data.

But in other cases like Math class (System namespace), there are a couple of static properties that are built into the class. one of which is the math constant PI

eg. Math.PI

and because PI is a piece of data that is well-defined, we don't need to have multiple copies of PI, it always going to be the same value. So static variables are sometimes used to share data amongst object of a class, but the are also commonly used for constant information where you only need one copy of a piece of data.

answered Feb 27 '18 at 11:33 community wiki Nermien Barakat

1 2 next