Diving into OOP (Day 7): Properties in C# (A Practical Approach)

Table of Contents:

Introduction:

My article of the series “Diving into OOP” will explain all about properties, its uses and indexers in C#. We’ll follow the same way of learning i.e. less theory and more practical. I’ll try to explain the concept in-depth.

Properties (The definition):

Let’s start with the definition taken from MSDN

*“*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.*”*

Roadmap:

Let’s recall our road map,

Properties (The explanation):

Being a C# programmer, I must say that properties are something a C# programmer is blessed to use in his code. If we analyze properties internals, they are very different from normal variables; properties are internally methods that do not have their own memory like variables have. We can leverage property to write our custom code whenever we access a property. We can access the code when we call/execute properties or at the time of declaration too, but this is not possible with variables. A property in easy language is a class member and is encapsulated and abstracted from the end developer who is accessing the property. A property can contain lots of code that an end user does not know. An end user only cares to use that property like a variable.
Let’s start with some coding now.
Note: Each and every code snippet in this article is tried and tested.

Lab1:

Create a simple console application and name it “Properties”. Add a class named “Properties” to it. You can choose the names of project and class as per your wish. Now try to create a property in that class like shown below,

Properties.cs:

```
namespace Properties
{
    public class Properties
    {
        public string Name{}
    }
}
```

Try to run/build the application, what do we get?

Output:

```
Error	'Properties.Properties.Name': property or indexer must have at least one accessor
```

In above example, we created a property named Name of type string. The error we got is very self-explanatory; it says a property must have an accessor, i.e. a get or a set. This means we need something in our property that gets accessed, whether to set the property value or get the property value, unlike variables, properties cannot be declared without an accessor.
.

Lab2:

Let's assign a get accessor to our property and try to access that property in Main method of program.cs file.

Properties.cs:

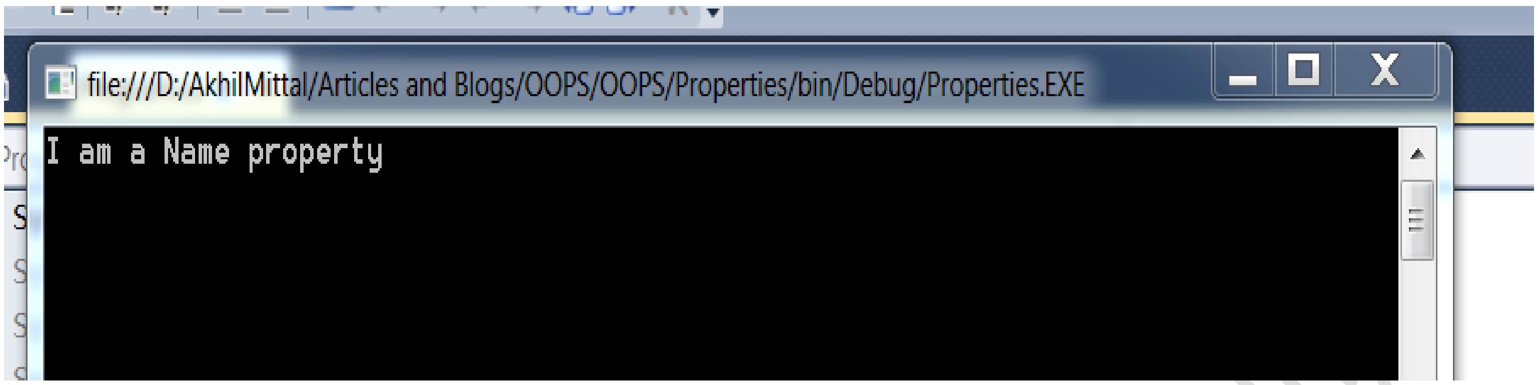
```
namespace Properties
{
    public class Properties
    {
        public string Name
        {
            get
            {
                return "I am a Name property";
            }
        }
    }
}
```

Program.cs:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Properties
{
    class Program
    {
        static void Main(string[] args)
        {
            Properties properties=new Properties();
            Console.WriteLine(properties.Name);
            Console.ReadLine();
        }
    }
}
```

Try to run/build the application, what do we get?

Output:


file:///D:/AkhilMittal/Articles and Blogs/OOPS/OOPS/Properties/bin/Debug/Properties.EXE

```
I am a Name property
```

It says “I am a Name property”. This signifies that our get successor got called when I tried to access the property or tried to fetch the value of a Property.

Get accessor

Now declare one more property in Properties.cs class, and name it Age that returns an integer. It calculates the age of a person by calculating the difference between his date of birth and current date,

Properties.cs:

```
using System;

namespace Properties
{
    public class Properties
    {
        public string Name
        {
            get
            {
                 return "I am a Name property";
            }
        }

        public int Age
        {
            get
            {
                DateTime dateOfBirth=new DateTime(1984,01,20);
                DateTime currentDate = DateTime.Now;
                int age = currentDate.Year - dateOfBirth.Year;
                return age;
            }
        }
    }
}
```

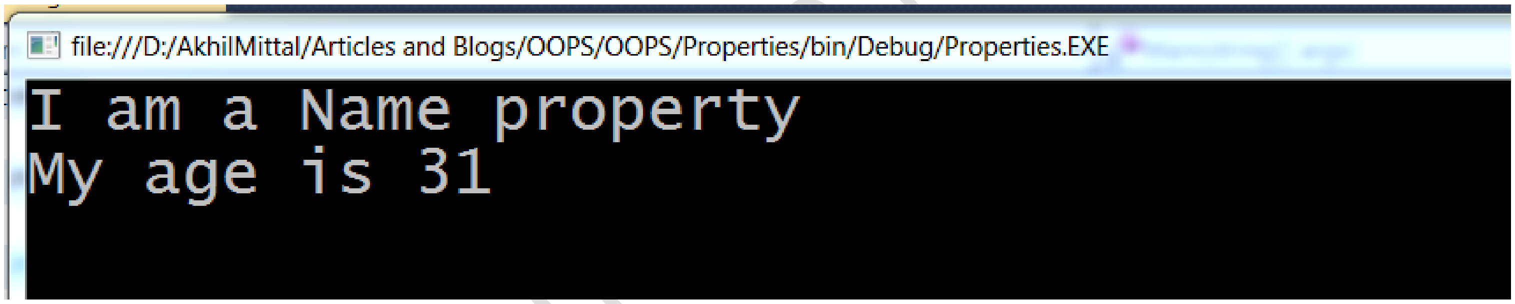
Call the “Age” property in the same way as done for Name.

Program.cs:

```
using System;

namespace Properties
{
    class Program
    {
        static void Main(string[] args)
        {
            Properties properties=new Properties();
            Console.WriteLine(properties.Name);
            Console.WriteLine("My age is " + properties.Age);
            Console.ReadLine();
        }
    }
}
```

Try to run/build the application, what do we get?

Output:



```
file:///D:/AkhilMittal/Articles and Blogs/OOPS/OOPS/Properties/bin/Debug/Properties.EXE
I am a Name property
My age is 31
```

It returns the correct age subjective to date of birth provided. Did you notince something here? Our property contains some code and logic to calculate age, but the caller i.e. Program.cs is not aware of the logic.it only cares about using that Property. Therefore we see that a property encapsulates and abstracts its functionality from the end user, in our case it's a developer.

Point to remember:

Get accessor is only used to read a property value. A property having only "get" cannot be set with any value from the caller.

This means a caller/end user can only access that property in read mode.

Set accessor

Let's start with a simple example.

Lab1:

Properties.cs:

```
using System;
namespace Properties
{
    public class Properties
    {
        public string Name
        {
            get { return "I am a Name property"; }
        }

        public int Age
        {
            get
            {
                DateTime dateOfBirth = new DateTime(1984, 01, 20);
                DateTime currentDate = DateTime.Now;
                int age = currentDate.Year - dateOfBirth.Year;
                Console.WriteLine("Get Age called");
                return age;
            }
            set
            {
                Console.WriteLine("Set Age called " + value);
            }
        }
    }
}
```

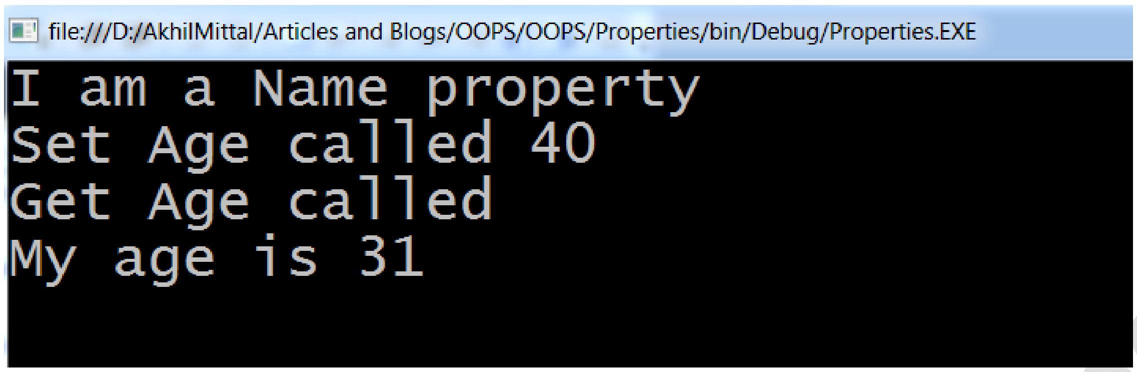
Call the “Age” property in the same way as done for Name.

Program.cs:

```
using System;

namespace Properties
{
    class Program
    {
        static void Main(string[] args)
        {
            Properties properties=new Properties();
            Console.WriteLine(properties.Name);
            properties.Age = 40;
            Console.WriteLine("My age is " + properties.Age);
            Console.ReadLine();
        }
    }
}
```

Run the application,

Output:


file:///D:/AkhilMittal/Articles and Blogs/OOPS/OOPS/Properties/bin/Debug/Properties.EXE

```
I am a Name property
Set Age called 40
Get Age called
My age is 31
```

In the above given example, I made few minor changes in get accessor, i.e. just printing that control is in "Get accessor" and introduced a "Set" in Age property too. Everything else remains same. Now when I call Name property, it works as was working earlier.Since we used "Set" so now we are allowed to set the value of a property. When I do **properties.Age = 40;** that means I am setting value 40 to that property.We can say a property can also be used to assign some value.In this case Set accessor is called, as soon as we set a value to property.Later on when we access that same property, again our get accessor gets called which returns value with some custom logic. We have a drawback here, as we see, whenever we call get we get the same vale and not the value that we assigned to that property i.e. because get has its custom fixed logic.Let's try to overcome this situation.

Lab2:

The example I am about to explain makes use of a private variable. But you can also use Automatic Properties to achieve same. I'll purposely make use of variable to have a better understanding.

Properties.cs:

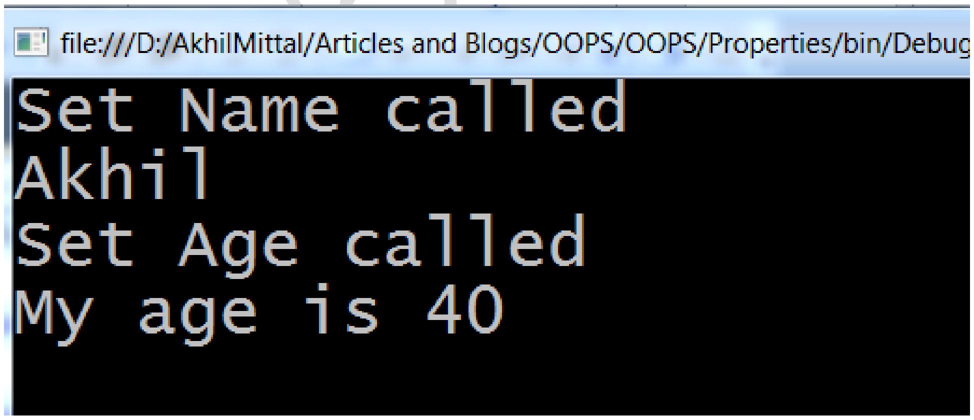
```
using System;
namespace Properties
{
    public class Properties
    {
        private string name;
        private int age;

        public string Name
        {
            get { return name; }
            set
            {
                Console.WriteLine("Set Name called ");
                name = value;
            }
        }
        public int Age
        {
            get { return age; }
            set
            {
                Console.WriteLine("Set Age called ");
                age = value;
            }
        }
    }
}
```

Program.cs:

```
using System;

namespace Properties
{
    class Program
    {
        static void Main(string[] args)
        {
            Properties properties=new Properties();
            properties.Name = "Akhil";
            Console.WriteLine(properties.Name);
            properties.Age = 40;
            Console.WriteLine("My age is " + properties.Age);
            Console.ReadLine();
        }
    }
}
```

Run the application,

Output:



file:///D:/AkhilMittal/Articles and Blogs/OOPS/OOPS/Properties/bin/Debug

```
Set Name called
Akhil
Set Age called
My age is 40
```

Now you see, we get the same value that we assigned to Name and Age property .When we access these properties get accessor is called and it returns the same vale as we set them for. Here properties internally make use of local variable to hold and sustain the value.

In day to day programming, we normally create a Public property that can be accessed outside the class. However the variable it is using internally could be a private.

Point to remember:

The variable used for property should be of same data type as the data type of the property.

In our case we used variables, name and age, they share same datatype as their respective properties do. We don't use variables as there might be scenarios in which we do not have control over those variables, end user can change them at any point of code without maintaining the change stack. Moreover one major use of properties is user can associate some logic or action when some change on the variable occurs, therefore when we use properties, we can easily track the value changes in variable.
When using Automatic Properties, they do this internally, i.e. we don't have to define an extra variable to do so,like shown below,

Lab3:

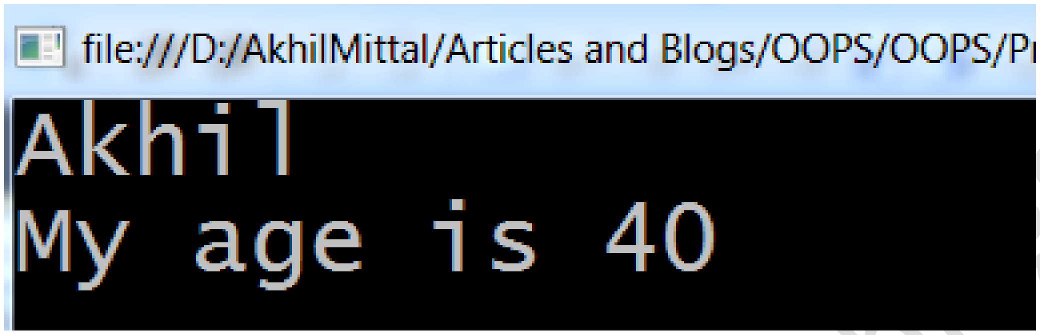
Properties.cs:

```
using System;
namespace Properties
{
    public class Properties
    {
        private string name;
        private int age;

        public string Name { get; set; }

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

Program.cs:

```
using System;

namespace Properties
{
    class Program
    {
        static void Main(string[] args)
        {
            Properties properties=new Properties();
            properties.Name = "Akhil";
            Console.WriteLine(properties.Name);
            properties.Age = 40;
            Console.WriteLine("My age is " + properties.Age);
            Console.ReadLine();
        }
    }
}
```
Run the application,

Output:


file:///D:/AkhilMittal/Articles and Blogs/OOPS/OOPS/Pr

```
Akhil
My age is 40
```

Here
```
    public string Name { get; set; }
    public int Age { get; set; }
```

are automatic properties.

I hope now you know how to define a property and use it.

Readonly

A property can be made read-only by only providing the get accessor. We do not provide a set accessor, if we do not want our property to be initialized or to be set from outside the scope of class.

Properties.cs:

```
using System;
namespace Properties
{
    public class Properties
    {
        private string name="Akhil";
        private int age=32;

        public string Name
        {
            get { return name; }
        }
        public int Age { get { return age; } }
    }
}
```

Program.cs:

```
using System;

namespace Properties
{
    class Program
    {
        static void Main(string[] args)
        {
            Properties properties=new Properties();
            properties.Name = "Akhil";
            Console.WriteLine(properties.Name);
            properties.Age = 40;
            Console.WriteLine("My age is " + properties.Age);
            Console.ReadLine();
        }
    }
}
```

Build the application, we get following output

```
Error     Property or indexer 'Properties.Properties.Age' cannot be assigned to -- it is read only
Error     Property or indexer 'Properties.Properties.Name' cannot be assigned to -- it is read only
```

In main method of Program class, we tried to set the value of Age and Name property by,

```
            properties.Name = "Akhil";
            properties.Age = 40;
```

But since they were marked read-only i.e. only with get accessor, we encountered a compile time error.

Write-Only

A property can also be made write-only i.e. vice versa to read-only. In this case you'll be only allowed to set the value of the property but can't access it because we don't have get accessor in it.

Properties.cs:

```
using System;
namespace Properties
{
    public class Properties
    {
        private string name;
        private int age;

        public string Name
        {
            set { name=value; }
        }

        public int Age { set { age = value; } }
    }
}
```

Program.cs:

```
using System;

namespace Properties
{
    class Program
    {
        static void Main(string[] args)
        {
            Properties properties=new Properties();
            properties.Name = "Akhil";
            Console.WriteLine(properties.Name);
            properties.Age = 40;
            Console.WriteLine("My age is " + properties.Age);
            Console.ReadLine();
        }
    }
}
```

Build the application, we get following output

Error The property or indexer 'Properties.Properties.Age' cannot be used in this context because it lacks the get accessor
Error The property or indexer 'Properties.Properties.Name' cannot be used in this context because it lacks the get accessor

In the above mentioned example, our property is marked only with set accessor, but we tried to access those properties in our main program with,

```
Console.WriteLine(properties.Name);
Console.WriteLine("My age is " + properties.Age);
```

That means we tried to call get accessor of property which is not defined, so we again ended up in a compile time error.

Insight of Properties in C#

Lab1:

Can we define properties as two different set of pieces? The answer is NO.

Properties.cs:

```
using System;
namespace Properties
{
    public class Properties
    {
        private string name;

        public string Name
        {
            set { name=value; }
        }

        public string Name
        {
            get { return name; }
        }
    }
}
```

Build the project, we get compile time error,

Error   The type 'Properties.Properties' already contains a definition for 'Name'

Here I tried to create a single property segregated in two different accessor.Compile treats a property name as a single separate property, so we cannot define a property with two names having different accessor.

Lab2:

Can we define properties same as an already defined variable? The answer is NO.

Properties.cs:

```
using System;
namespace Properties
{
    public class Properties
    {
        private string name;

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

Build the project; we get compile time error,

Error   The type 'Properties.Properties' already contains a definition for 'name'

Again, we cannot have a variable and a property with the same name. They may differ on the grounds of case sensitivity, but they cannot share a same common name with the same case because at the time of accessing them, compiler may get confused that whether you are trying to access a property or a variable.

Properties vs Variables

It is a conception that variables are faster in execution that properties. I do not deny about this but this may not be true ion every case or can vary case to case. A property, like I explained internally executes a function/method whereas a variable uses/initializes memory when used. At times properties are not slower than variables as the property code is internally rewritten to memory access.

To summarize, MSDN explains this theory better than me,

Point of difference	Variable	Property
Declaration	Single declaration statement	Series of statements in a code block
Implementation	Single storage location	Executable code (property procedures)
Storage	Directly associated with variable's value	Typically has internal storage not available outside the property's containing class or module Property's value might or might not exist as a stored element [1]
Executable code	None	Must have at least one procedure
Read and write access	Read/write or read-only	Read/write, read-only, or write-only
Custom actions (in addition to accepting or returning value)	Not possible	Can be performed as part of setting or retrieving property value

Table reference: https://msdn.microsoft.com/en-us/library/sk5e8eth(VS.80).aspx

Static Properties:

Like variables and methods, a property can also be marked static,

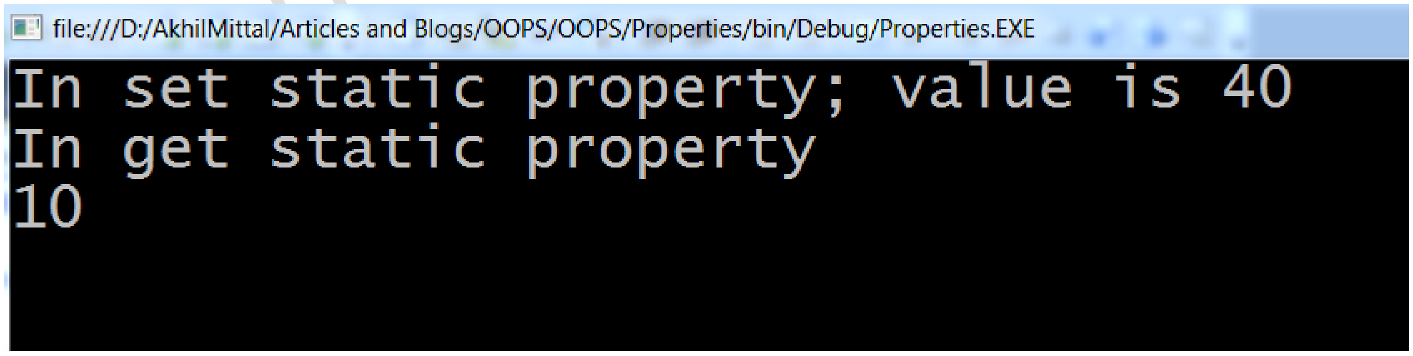
Properties.cs:

```
using System;
namespace Properties
{
    public class Properties
    {
        public static int Age
        {
            set
            {
                Console.WriteLine("In set static property; value is " + value);
            }
            get
            {
                Console.WriteLine("In get static property");
                return 10;
            }
        }
    }
}
```

Program.cs:

```
using System;

namespace Properties
{
    class Program
    {
        static void Main(string[] args)
        {
            Properties.Age = 40;
            Console.WriteLine(Properties.Age);
            Console.ReadLine();
        }
    }
}
```

Output:



file:///D:/AkhilMittal/Articles and Blogs/OOPS/OOPS/Properties/bin/Debug/Properties.EXE

```
In set static property; value is 40
In get static property
10
```

In above example, I created a static Age property. When I tried to access it, you can see it is accessed via class name, like all static members are subjected to. So properties also inherit the static functionality like all c# members, no matter it is variable or a method. They'll be accessed via class name only.

Properties return type:

Lab1:

Properties.cs:

```
using System;

namespace Properties
{
  public class Properties
    {
        public void AbsProperty
        {
            get
            {
                Console.WriteLine("Get called");
            }
        }
    }
}
```

Compile the program.

Output is a compile time error,

```
Error	'AbsProperty': property or indexer cannot have void type
```

Point to remember:

A property cannot have a void return type.

Lab2:

Just try to return a value from "set" accessor,

Properties.cs:

```
using System;

namespace Properties
{
  public class Properties
    {
        public int Age
        {
            set { return 5; }
        }
    }
}
```

Compile the program,

Error Since 'Properties.Properties.Age.set' returns void, a return keyword must not be followed by an object expression

Here compiler understands “set” accessor as a method that returns void and takes a parameter to initialize the value. So set cannot be expected to return a value.

If we just leave return statement empty, and remove 5, we do not get any error and code compiles,

```
using System;

namespace Properties
{
  public class Properties
    {
        public int Age
        {
            set { return ; }
        }
    }
}
```

Value Keyword

We have a reserved keyword named value.

```
using System;

namespace Properties
{
  public class Properties
    {
        public string Name
        {
            set { string value; }
        }
    }
}
```

Just compile the above given code, we get a compile time error as follows,

Error A local variable named 'value' cannot be declared in this scope because it would give a different meaning to 'value', which is already used in a 'parent or current' scope to denote something else

This signifies that “value” is a reserved keyword here. So one cannot declare a variable named value in “set” accessor as it may give different meaning to already reserved keyword value.

Abstract Properties:

Lab1:

Yes, we can also have abstract properties; let’s see how it works,

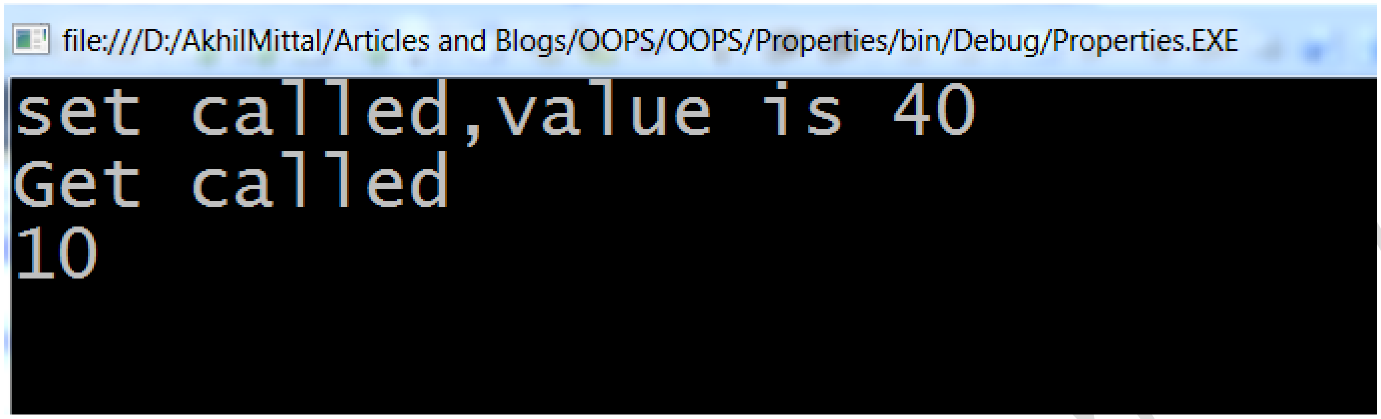
Properties.cs:

```
using System;

namespace Properties
{
    public abstract class BaseClass
    {
        public abstract int AbsProperty { get; set; }
    }

    public class Properties : BaseClass
    {
        public override int AbsProperty
        {
            get
            {
                Console.WriteLine("Get called");
                return 10;
            }
            set { Console.WriteLine("set called,value is " + value); }
        }
    }
}
```

Program.cs:

```
using System;

namespace Properties
{
    class Program
    {
        static void Main(string[] args)
        {
            Properties prop=new Properties();
            prop.AbsProperty = 40;
            Console.WriteLine(prop.AbsProperty);
            Console.ReadLine();
        }
    }
}
```

Output:


file:///D:/AkhilMittal/Articles and Blogs/OOPS/OOPS/Properties/bin/Debug/Properties.EXE

```
set called,value is 40
Get called
10
```

In above example, I just created a base class named “BaseClass” and defined an abstract property named Absproperty. Since the property is abstract it follows the rules of being abstract as well. I inherited my “Properties” class from BaseClass and given the body to that abstract property. Since the property was abstract I have to override it in my derived class to add functionality to it. So I used override keyword in my derived class.

In base class, abstract property has no body at all, neither for “get” and nor for “set”, so we have to implement both the accessor in our derived class, like shown in “Properties” class.

Point to remember:

If one do not mark property defined in derived class as override, it will by default be considered as new.

For more understanding follow http://www.codeproject.com/Articles/774578/Diving-in-OOP-Day-Polymorphism-and-Inheritance-Dyn article on new and override.

Lab2:

Properties.cs:

```
using System;

namespace Properties
{
    public abstract class BaseClass
    {
        public abstract int AbsProperty { get; }
    }

    public class Properties : BaseClass
    {
        public override int AbsProperty
        {
            get
            {
                Console.WriteLine("Get called");
                return 10;
            }
            set { Console.WriteLine("set called,value is " + value); }
        }
    }
}
```

Program.cs:

```
using System;

namespace Properties
{
    class Program
    {
        static void Main(string[] args)
        {
            Properties prop=new Properties();
            prop.AbsProperty = 40;
            Console.WriteLine(prop.AbsProperty);
            Console.ReadLine();
        }
    }
}
```

Output:

Compile time error,

```
Error	'Properties.Properties.AbsProperty.set': cannot override because 'Properties.BaseClass.AbsProperty' does not have an overridable set accessor
```

In above lab example, I just removed “set” from AbsProperty in Base class.All the code remains same.Now here we are trying to override the set accessor too in derived class, that is missing in base class, therefore compiler will not allow you to override a successor that is not declared in base class, hence resulted into a compile time error.

Point to remember:

You cannot override an accessor that is not defined in a base class abstract property.

Properties in Inheritance

Just follow the given code,

Properties.cs:

```
using System;

namespace Properties
{
  public class PropertiesBaseClass
    {
        public int Age
        {
            set {}
        }
    }

    public class PropertiesDerivedClass:PropertiesBaseClass
    {
        public int Age
        {
            get { return 32; }
        }
    }
}
```

Program.cs:

```
namespace Properties
{
    class Program
    {
        static void Main(string[] args)
        {
            PropertiesBaseClass pBaseClass=new PropertiesBaseClass();
            pBaseClass.Age = 10;
            PropertiesDerivedClass pDerivedClass=new PropertiesDerivedClass();
            ((PropertiesBaseClass) pDerivedClass).Age = 15;
            pDerivedClass.Age = 10;
        }
    }
}
```

As you can see in above given code, in Properties.cs file I created two classes one is Base i.e. PropertiesBaseClass and second in Derived i.e. PropertiesDerivedClass. I purposely declared set accessor in Base class and get in Derived class for the same property name i.e. Age. Now this case may give you the feeling that when compiled, our code of property Age will become one, i.e. it will take set from Base class and get from derived class and combine it into a single entity of Age property.But this is practically not the case.Compiler treats both these properties differently, and does not consider them to be same. In this case the property in derived class actually hides the property in base class , they are not the same but independent properties.The same concept of method hiding applies here too. You can read about hiding in http://www.codeproject.com/Articles/774578/Diving-in-OOP-Day-Polymorphism-and-Inheritance-Dyn.
To use the property of base class from a derived class object, you need to cast it to base class and then use it.
When you compile the above code, you get a compile time error as follows,

Error Property or indexer 'Properties.PropertiesDerivedClass.Age' cannot be assigned to -- it is read only

i.e. we can do `((PropertiesBaseClass) pDerivedClass).Age = 15;`

but we cannot do `pDerivedClass.Age = 10;` because derived class property has no “set” accessor.

Summary:

Let’s recall all the points that we have to remember,

- The variable used for property should be of same data type as the data type of the property.
- A property cannot have a void return type.
- If one do not mark property defined in derived class as override, it will by default be considered as new.
- You cannot override an accessor that is not defined in a base class abstract property.
- Get accessor is only used to read a property value. A property having only get cannot be set with any value from the caller.

Conclusion:

In this article we learnt a lot about properties in c#. I hope you now understand properties well. In my next article I’ll explain all about indexers in C#.

Keep coding and enjoy reading
Also do not forget to rate/comment/like my article if it helped you by any means, this helps me to get motivated and encourages me to write more and more.

Read more:

- Learning MVC series.
- C# and Asp.Net Questions (All in one)
- MVC Interview Questions
- C# and Asp.Net Interview Questions and Answers
- Web Services and Windows Services Interview Questions

For more technical articles you can reach out to A Practical Approach.