Properties in C# with Examples

Why do we need Properties in C#?

In order to encapsulate and protect the data members (i.e. fields or variables) of a class, we use properties in C#. The Properties in C# are used as a mechanism to set and get the values of data members of a class outside of that class. If a class contains any values in it and if we want to access those values outside of that class, then we can provide access to those values in 2 different ways. They are as follows:

1. By storing the value under a public variable, we can give direct access to the value outside of the class.
2. By storing that value in a private variable, we can also give access to that value outside of the class by defining a property for that variable.

What is a Property in C#?

A Property in C# is a member of a class that is used to set and get the data from a data field (i.e. variable) of a class. The most important point that you need to remember is that a property in C# is never used to store any data, it just acts as an interface or medium to transfer the data. We use the Properties as they are public data members of a class, but they are actually special methods called accessors.

What are Accessors in C#?

The Assessors are nothing but special methods which are used to set and get the values from the underlying data member (i.e. variable) of a class. Assessors are of two types. They are as follows:

1. **Set Accessor**
2. **Get Accessor**

What is a Set Accessor?

The **set** accessor is used to set the data (i.e. value) into a data field i.e. a variable of a class. This set accessor contains a fixed variable named **value**. Whenever we call the property to set the data, whatever data (value) we are supplying will come and store inside the variable called **value** by default. Using a set accessor, we cannot get the data.

Syntax: set { Data_Field_Name = value; }

What is Get Accessor?

The get accessor is used to get the data from the data field i.e. variable of a class. Using the get accessor, we can only get the data, we cannot set the data.

Syntax: get {return Data_Field_Name;}

Example to Understand Properties in C#:

In the below example, I have shown you the use of Properties in C#. Here, we have created two classes i.e. Employee and Program and we want to access the Employee class data members inside the Program class. In the Employee class, we have created two private data members (i.e. **_EmpId** and **_EmpName**) to hold the Employee Id and Name of the Employee and as we mark these two variables as private, so we cannot access directly these two members from outside the Employee class. We cannot access them directly from the Program class. Then for these two data members, we have created two public properties i.e. **EmpId** and **EmpName** to get and set the Employee ID and Name respectively. The point that you need to remember is properties are not going to store the value, rather they are just transferring the values. The variables are going to store the data. Further, the following example code is self-explained, so please go through the comments line.

```
using System;
namespace PropertyDemo
{
   public class Employee
   {
      //Private Data Members
      private int _EmpId;
      private string _EmpName;

      //Public Properties
      public int EmpId
      {
         //The Set Accessor is used to set the _EmpId private variable value
         set
         {
            _EmpId = value;
         }
         //The Get Accessor is used to return the _EmpId private variable value
         get
         {
            return _EmpId;
         }
      }
      public string EmpName
      {
         //The Set Accessor is used to set the _EmpName private variable value
         set
         {
            _EmpName = value;
         }
```

```
        //The Get Accessor is used to return the _EmpName private variable value
        get
        {
            return _EmpName;
        }
    }
}
class Program
{
    static void Main(string[] args)
    {
        Employee employee = new Employee();
        //We cannot access the private data members
        //So, using public properties (SET Accessor) we are setting
        //the values of private data members
        employee.EmpId = 101;
        employee.EmpName = "Pranaya";

        //Using public properties (Get Accessor) we are Getting
        //the values of private data members
        Console.WriteLine("Employee Details:");
        Console.WriteLine("Employee id:" + employee.EmpId);
        Console.WriteLine("Employee name:" + employee.EmpName);
        Console.ReadKey();
    }
}
}
```

Output:

```
Employee Details:
Employee id:101
Employee name:Pranaya
```

Now, you may have one question. Why not make the variables public? Why are we are creating the variables as private and why we are creating public properties for them? The answer is to achieve the Encapsulation Principle. We will discuss this in detail when we discuss the **Encapsulation Principle in C#**.

What are the Different types of Properties Supported by C#.NET?

The C#.NET supports four types of properties. They are as follows

1. **Read-Only Property**
2. **Write Only Property**
3. **Read Write Property**
4. **Auto-Implemented Property**

Let us understand each of the above properties in detail with examples.

What is Read-only Property in C#?

The Read-Only Property is used to read the data from the data field i.e. read the data of a variable. Using this Read-Only Property, we cannot set the data into the data field. This property will contain only one accessor i.e. get accessor.

Syntax:

```
AccessModifier Datatype PropertyName
{
    get {return DataFieldName;}
}
```

What is Write only Property in C#?

The Write-Only Property is used to write the data into the data field i.e. write the data to a variable of a class. Using this Write-Only Property, we cannot read the data from the data field. This property will contain only one accessor i.e. set accessor.

Syntax:

```
AccessModifier Datatype PropertyName
{
     set {DataFieldName = value;}
}
```

What is Read Write Property in C#?

The Read-Write Property is used for both reading the data from the data field as well as writing the data into the data field of a class. This property will contain two accessors i.e. set and get. The set accessor is used to set or write the value to a data field and the get accessor is read the data from a variable.

Syntax:

```
AccessModifier Datatype PropertyName
{
    set {DataFieldName = value;}
    get {return DataFieldName;}
}
```

Note: Whenever we create a property for a variable, the data type of the property must be the same as the data type of the variable. A property can never accept any argument.

Example to understand the Read and Write Property in C#

In the below example, within the Employee class, we have created four private variables and for private each variable we have created public properties. And we have created each property with both set and get accessors which will make them read and write property and using these properties we can perform both read and write operations. The point that you need to remember is the data type of the property and the data of the corresponding variables must be the same, or else you will get a compile-time error. Then from the Main method, we create an instance of the Employee class, and then by using the public properties we are setting the getting values.

```
using System;
namespace PropertyDemo
{
    public class Employee
    {
        //Private Data Members
        private int _EmpId, _Age;
        private string _EmpName, _Address;

        //Public Properties
        public int EmpId
        {
            //The Set Accessor is used to set the _EmpId private variable value
            set
            {
                _EmpId = value;
            }
            //The Get Accessor is used to return the _EmpId private variable value
            get
            {
                return _EmpId;
            }
        }

        public int Age
        {
            //The Set Accessor is used to set the _Age private variable value
            set
            {
                _Age = value;
            }
            //The Get Accessor is used to return the _Age private variable value
            get
            {
                return _Age;
            }
        }
        public string EmpName
```

```
    {
        //The Set Accessor is used to set the _EmpName private variable value
        set
        {
            _EmpName = value;
        }
        //The Get Accessor is used to return the _EmpName private variable value
        get
        {
            return _EmpName;
        }
    }
    public string Address
    {
        //The Set Accessor is used to set the _Address private variable value
        set
        {
            _Address = value;
        }
        //The Get Accessor is used to return the _Address private variable value
        get
        {
            return _Address;
        }
    }
}
class Program
{
    static void Main(string[] args)
    {
        Employee employee = new Employee();
        //We cannot access the private data members
        //So, using public properties (SET Accessor) we are setting
        //the values of private data members
        employee.EmpId = 101;
        employee.Age = 101;
        employee.EmpName = "Pranaya";
        employee.Address = "BBSR, Odisha, India";

        //Using public properties (Get Accessor) we are Getting
        //the values of private data members
        Console.WriteLine("Employee Details:");
        Console.WriteLine($"Id: {employee.EmpId}");
        Console.WriteLine($"Name: {employee.EmpName}");
        Console.WriteLine($"Age: {employee.Age}");
        Console.WriteLine($"Address: {employee.Address}");
```

```
            Console.ReadKey();
        }
    }
}
```

Output:

```
Employee Details:
Id: 101
Name: Pranaya
Age: 101
Address: BBSR, Odisha, India
```

In the above example, we declare the data fields i.e. variables of Employee class as private. As a result, these data fields or variables are not accessible directly from outside the Employee class. So, here, in the Program class which is outside of the Employee class, we transferred the data into the data field or variables with the help of properties.

Example to understand the Read-Only and Write-Only Properties in C#:

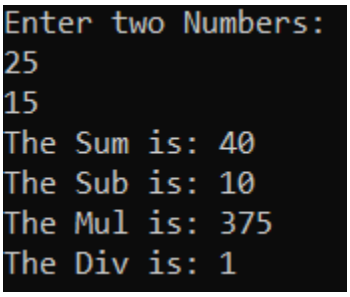
In the below example, within the Calculator class, we have created three private variables. Then for these three private variables, we have created two write-only properties (property with only set accessor) for _Number1 and _Number2 variables and one read-only property (property with only get accessor) for _Result variable. Using the write-only property we can only set the values and using only the read-only property we can get the value. Then from the Main method of the Program class, we create an instance of the Calculator class and access the read-only and write-only properties.

```
using System;
namespace PropertyDemo
{
    public class Calculator
    {
        int _Number1, _Number2, _Result;

        //Write-Only Properties
        //Only Set Accessor, No Get Accessor
        public int SetNumber1
        {
            set
            {
                _Number1 = value;
            }
        }
```

```
    public int SetNumber2
    {
        set
        {
            _Number2 = value;
        }
    }

    //Read-Only Property
    //Only Get Accessor, No Set Accessor
    public int GetResult
    {
        get
        {
            return _Result;
        }
    }
    public void Add()
    {
        _Result = _Number1 + _Number2;
    }
    public void Sub()
    {
        _Result = _Number1 - _Number2;
    }
    public void Mul()
    {
        _Result = _Number1 * _Number2;
    }
    public void Div()
    {
        _Result = _Number1 / _Number2;
    }
}
class Program
{
    static void Main(string[] args)
    {
        Calculator calculator = new Calculator();
        Console.WriteLine("Enter two Numbers:");
        calculator.SetNumber1 = int.Parse(Console.ReadLine());
        calculator.SetNumber2 = int.Parse(Console.ReadLine());

        calculator.Add();
        Console.WriteLine($"The Sum is: {calculator.GetResult}");
```

```
            calculator.Sub();
            Console.WriteLine($"The Sub is: {calculator.GetResult}");

            calculator.Mul();
            Console.WriteLine($"The Mul is: {calculator.GetResult}");

            calculator.Div();
            Console.WriteLine($"The Div is: {calculator.GetResult}");
            Console.ReadKey();
        }
    }
}
```

Output:



What are the advantages of using Properties in C#?

1. Properties will provide the abstraction to the data fields.
2. They also provide security to the data fields.
3. Properties can also validate the data before storing it in the data fields.

Note: When we will discuss the **Encapsulation OOPs Principle**, at that time, I will explain the above points with practical examples.

What is the Default Accessibility Specifier of Accessors in C#?

The default accessibility specifier of the accessor is the same as the accessibility specifier of the property. For example:

```
public int EmpId
{
    set { _EmpId = value; }
    get {  return _EmpId; }
}
```

In the above example, the property Empid is declared as public. So, the set and get accessor will be public. If the property is private then both set and get accessors will also be private.

What are symmetric and asymmetric accessors in C#?

If the accessibility specifier of the accessors (both get and set) are the same within a property accessibility specifier then the accessors are known as symmetric accessors. On the other hand, if the accessibility specifier of the accessors is not the same as a property accessibility specifier, then the accessors are known as asymmetric accessors. For example:

```
public int EmpId
{
    protected set { _EmpId = value; }
    get { return _EmpId; }
}
```

In the above property, the set accessor is declared as protected while the get accessor is public by default, so they are known as asymmetric. In general, asymmetric accessors are used in the inheritance process. We will discuss this in detail when we discuss **Inheritance OOPs Principle** in C#.

We can also write the Read-only property using two accessors as follows.

```
public int EmpId
{
    private set { _EmpId = value; }
    get { return _EmpId; }
}
```

We can also write the Write only property using two accessors as follows.

```
public int EmpId
{
    set { _EmpId = value; }
    private get { return _EmpId; }
}
```

Note: The point that you need to remember is once you declare the accessor as private, then you cannot access that accessor from outside the class.

What are Auto-Implemented Properties in C#?

If you do not have any additional logic while setting and getting the data from a data field i.e. from a variable of a class, then you can make use of the auto-implemented properties which was introduced as part of C# 3.0. The Auto-Implemented Property in C# reduces the amount of code that we have to write. When we use auto-implemented properties, then the C# compiler implicitly creates a private, anonymous field or variable for that property behind the scene which is going to hold the data.

Syntax: **Access specifier Datatype Property_Name { get; set; }**
Example: **public int A { get; set; }**

Example to understand Auto Implemented Properties in C#:

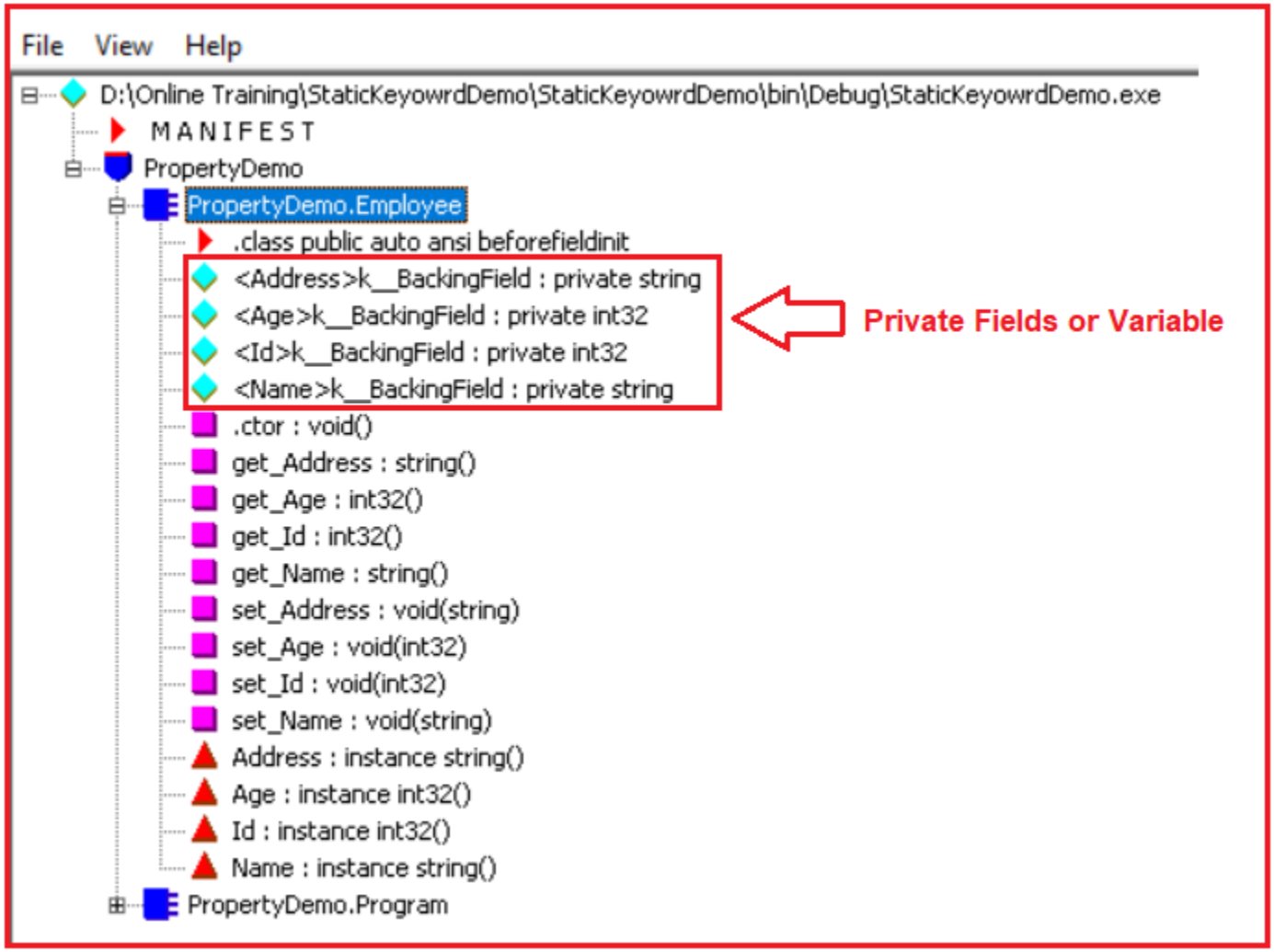
In the below example, I am showing the use of Auto Implemented Properties in C#. Please observe the Employee class. In the Employee class, we have not created any private data fields or variables to hold the data. But we have created four Auto Implemented Properties. When we create Auto Implemented Properties, behind the scene, the compiler will create the private anonymous field for each property to hold the data.

```
using System;
namespace PropertyDemo
{
   public class Employee
   {
      public int Id { get; set; }
      public int Age { get; set; }
      public string Name { get; set; }
      public string Address { get; set; }

   }
   class Program
   {
      static void Main(string[] args)
      {
         Employee employee = new Employee();
         employee.Id = 101;
         employee.Age = 101;
         employee.Name = "Pranaya";
         employee.Address = "BBSR, Odisha, India";

         Console.WriteLine("Employee Details:");
         Console.WriteLine($"Id: {employee.Id}");
         Console.WriteLine($"Name: {employee.Name}");
         Console.WriteLine($"Age: {employee.Age}");
         Console.WriteLine($"Address: {employee.Address}");
         Console.ReadKey();
      }
   }
}
```

Output:

```
Employee Details:
Id: 101
Name: Pranaya
Age: 101
Address: BBSR, Odisha, India
```

Now, if you verify the IL code of the Employee class using the ILDASM tool, then you will see that four private variables are created behind scenes by the compiler as shown in the below image.



Why do we need Properties in C# Real-time Applications?

Declaring the class fields or variables as public and exposing those fields or variables to the outside world (which means outside of the class) is bad as we do not have any control over what gets assigned and what gets returned. Let's understand this with one example.

```
using System;
namespace PropertyDemo
{
   public class Student
   {
      public int ID;
      public string Name;
      public int PassMark;
```

```
    }
    class Program
    {
        static void Main(string[] args)
        {
            Student student = new Student();
            student.ID = -100;
            student.Name = null;
            student.PassMark = 0;
            Console.WriteLine($"ID = {student.ID}, Name = {student.Name}, PassMark =
{student.PassMark}");
            Console.ReadKey();
        }
    }
}
```

Output:

```
ID = -100, Name = , PassMark = 0
```

Problems with the above public fields are as follows

1. An ID value should always be a non-negative number.
2. The name cannot be set to NULL.
3. If a student's name is missing then we should return "No Name".
4. The PassMark value should always be read-only.

The Programming Languages like C++, and Java does not have the concept properties and such programming languages use getter and setter methods to encapsulate and protect fields.

Example using Setter and Getter Methods in C#:

Let's rewrite the previous example using setter and getter methods to achieve the above requirements. For each variable or data field, we need to write setter or getter methods as per our requirements. Here, we have written setter and getter methods for _ID and _Name variables to set and get the ID and Name values. On the other hand, we have only getter methods for the _PassMark variable, so from outside the class, we cannot set the value for PassMark. Again, within the setter and getter properties, we have also written logic to validate the data before storing and returning.

```
using System;
namespace PropertyDemo
{
    public class Student
    {
        private int _ID;
        private string _Name;
```

```
    private int _PassMark = 35;
    public void SetID(int ID)
    {
        if (ID < 0)
        {
            throw new Exception("ID value should be greater than zero");
        }
        _ID = ID;
    }
    public int GetID()
    {
        return _ID;
    }
    public void SetName(string Name)
    {
        if (string.IsNullOrEmpty(Name))
        {
            throw new Exception("Name should not be empty");
        }
        _Name = Name;
    }
    public string GetName()
    {
        if (string.IsNullOrEmpty(_Name))
        {
            return "No Name";
        }
        return _Name;
    }
    public int GetPassMark()
    {
        return _PassMark;
    }
}
class Program
{
    static void Main(string[] args)
    {
        Student student = new Student();
        student.SetID(101);
        student.SetName("Pranaya");

        Console.WriteLine($"ID = {student.GetID()}");
        Console.WriteLine($"Name = {student.GetName()}");
        Console.WriteLine($"Pass Mark = {student.GetPassMark()}");
        Console.ReadKey();
```

```
    }
  }
}
```

Output:

```
ID = 101
Name = Pranaya
Pass Mark = 35
```

Example using Properties in C#:

The advantage of properties over the traditional getter() and setter() methods is that we can access them as they are public fields, not methods. Let's rewrite the same program using properties to achieve the same requirements.

```
using System;
namespace PropertyDemo
{
   public class Student
   {
      private int _ID;
      private string _Name;
      private int _PassMark = 35;
      public int ID
      {
         set
         {
            if (value < 0)
            {
               throw new Exception("ID value should be greater than zero");
            }
            _ID = value;
         }
         get
         {
            return _ID;
         }
      }
      public string Name
      {
         set
         {
            if (string.IsNullOrEmpty(value))
            {
               throw new Exception("Name should not be empty");
```

```
            }
            _Name = value;
        }
        get
        {
            return string.IsNullOrEmpty(_Name) ? "No Name" : _Name;
        }
    }
    public int PassMark
    {
        get
        {
            return _PassMark;
        }
    }
}
class Program
{
    static void Main(string[] args)
    {
        Student student = new Student();
        student.ID = 101;
        student.Name = "Pranaya";

        Console.WriteLine($"ID = {student.ID}");
        Console.WriteLine($"Name = {student.Name}");
        Console.WriteLine($"Pass Mark = {student.PassMark}");
        Console.ReadKey();
    }
}
}
```

Output:

```
ID = 101
Name = Pranaya
Pass Mark = 35
```

Here, in this article, I try to explain **Properties in C#** with Examples. I hope you understood the need and use of Properties in C#.