

Diving into OOP (Day 8): Indexers in C# (A Practical Approach)

Table of Contents:

Table of Contents:	1
Introduction:	1
Indexers in C# (The definition)	2
Roadmap:	2
Indexers (The explanation):	2
Lab 1:	3
Lab 2:	4
Lab 3:	5
Data-Types in Indexers	6
Lab 1:	6
Indexers in interfaces	7
Indexers in Abstract class	10
Indexer Overloading	11
Point to remember:	13
Static Indexers?	13
Point to remember:	13
Inheritance/Polymorphism in Indexers	13
.Net Framework and Indexers	15
Point to remember	15
Properties vs Indexers	16
Conclusion:	16

Introduction:

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

Indexers in C# (The definition)

Let's take the definition from <https://msdn.microsoft.com/en-us/library/6x16t2tx.aspx>,

"Indexers allow instances of a class or struct to be indexed just like arrays. Indexers resemble [properties](#) except that their accessors take parameters."

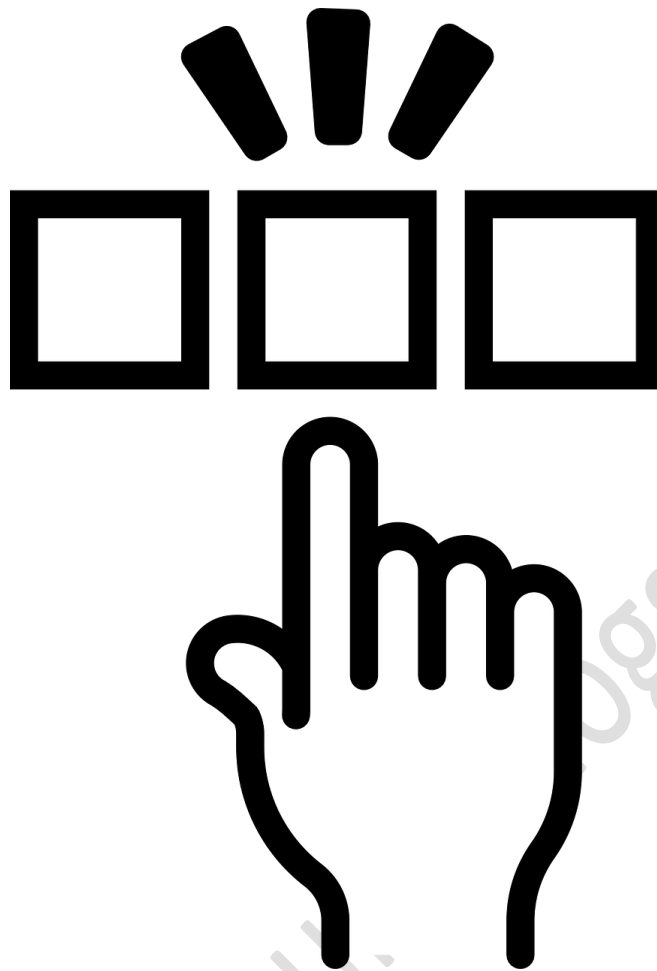
Roadmap:

Let's recall our road map,



1. **Diving in OOP (Day 1): Polymorphism and Inheritance(Early Binding/Compile Time Polymorphism)**
2. [Diving in OOP \(Day 2\): Polymorphism and Inheritance \(Inheritance\)](#)
3. [Diving in OOP \(Day 3\): Polymorphism and Inheritance \(Dynamic Binding/Run Time Polymorphism\)](#)
4. [Diving in OOP \(Day 4\): Polymorphism and Inheritance \(All about Abstract classes in C#\)](#)
5. [Diving in OOP \(Day 5\): All about access modifiers in C# \(Public/Private/Protected/Internal/Sealed/Constants/Readonly Fields\)](#)
6. [Diving in OOP \(Day 6\): Understanding Enum in C# \(A Practical Approach\)](#)
7. [Diving into OOP \(Day 7\): Properties in C# \(A Practical Approach\).](#)
8. **Diving into OOP (Day 8): Indexers in C# (A Practical Approach).**

Indexers (The explanation):



Like definition says, indexers allow us to leverage the capability of accessing the class objects as an array. For better understanding, create a console application named **Indexers** and add a class to it named **Indexer**. We'll use this class and project to learn Indexers. Make the class public, do not add any code for now and in **Program.cs** add following code,

Lab 1:

```
namespace Indexers
{
    class Program
    {
        static void Main(string[] args)
        {
            Indexer indexer=new Indexer();
            indexer[1] = 50;
        }
    }
}
```

Compile the code. We get,

Error Cannot apply indexing with [] to an expression of type 'Indexers.Indexer'

I just created an object of Indexer class and tried to use that object as an array. Since actually it was not an array, it resulted as a compile time error.

Lab 2:

Indexer.cs

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

namespace Indexers
{
    public class Indexer
    {
        public int this[int indexValue]
        {
            set
            {
                Console.WriteLine("I am in set : Value is " + value + " and indexValue is " +
indexValue);
                Console.ReadLine();
            }
        }
    }
}
```

Program.cs:

```
namespace Indexers
{
    class Program
    {
        static void Main(string[] args)
        {
            Indexer indexer=new Indexer();
            indexer[1] = 50;
        }
    }
}
```

Output:

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

I am in set : Value is 50 and indexValue is 1

Here we just made a use of indexer to index my object of the class Indexer. Now my object can be used as an array to access different object values.

Implementation of indexers is derived from a property known as "this". It takes an integer parameter indexValue. Indexers are different from properties. In properties when we want to initialize or assign a value, the "set" accessor if defined automatically gets called. And the keyword "value" in "set" accessor was used to hold or keep track of the assigned value to our property. In above example, `indexer[1] = 50;` calls the "set" accessor of "this" property i.e. an indexer therefore 50 becomes value and 1 becomes index of that value.

Lab 3:

Indexer.cs

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

namespace Indexers
{
    public class Indexer
    {
        public int this[int indexValue]
        {
            set
            {
                Console.WriteLine("I am in set : Value is " + value + " and indexValue is " +
indexValue);
            }
            get
            {
                Console.WriteLine("I am in get and indexValue is " + indexValue);
                return 30;
            }
        }
    }
}
```

Program.cs:

```
using System;

namespace Indexers
{
    class Program
    {
        static void Main(string[] args)
        {

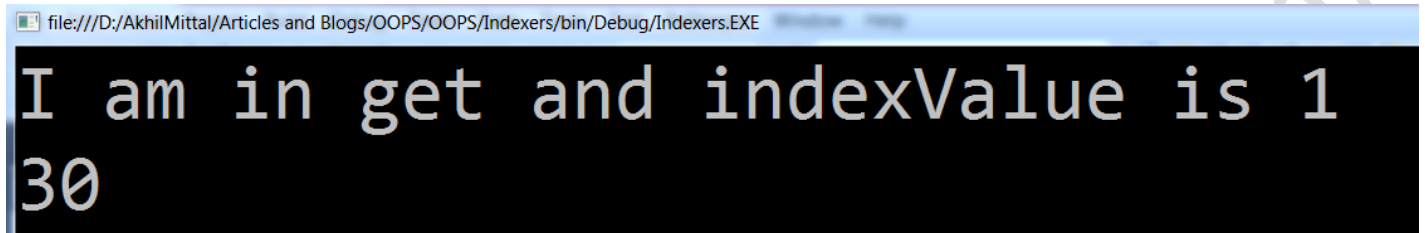
```

```

        Indexer indexer=new Indexer();
        Console.WriteLine(indexer[1]);
        Console.ReadKey();
    }
}

```

Output:



In the above code snippet, I used get as well, to access the value of indexer. Properties and Indexers work on same set of rules. There is a bit difference on how we use them. When we do `indexer[1]` that means “get” accessor is called, and when we assign some value to `indexer[1]` that means “set” accessor is called. While implementing indexer code we have to take care that when we access indexer it is accessed in the form of a variable and that too an array parameter.

Data-Types in Indexers

Lab 1:

Indexer.cs

```

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

namespace Indexers
{
    public class Indexer
    {
        public int Index;
        public int this[string indexValue]
        {
            set
            {
                Console.WriteLine("I am in set : Value is " + value + " and indexValue is " +
indexValue);
                Index = value;
            }
            get
            {

```

```

        Console.WriteLine("I am in get and indexValue is " + indexValue);
        return Index;
    }
}
}

```

Program.cs:

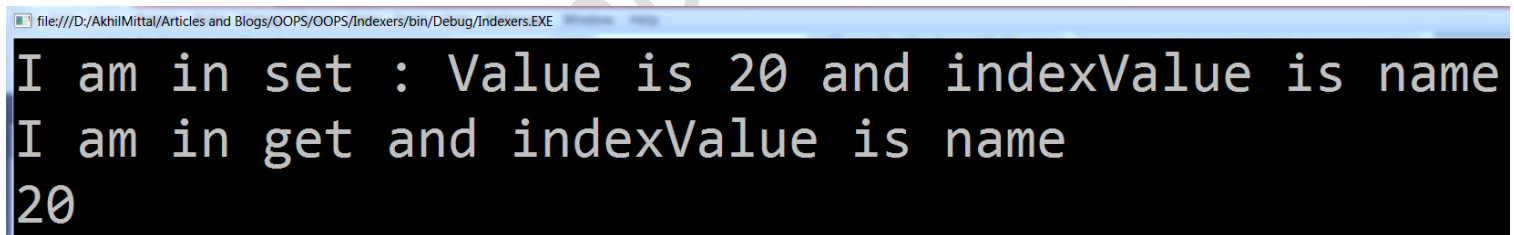
```

using System;

namespace Indexers
{
    class Program
    {
        static void Main(string[] args)
        {
            Indexer indexer=new Indexer();
            indexer["name"]=20;
            Console.WriteLine(indexer["name"]);
            Console.ReadKey();
        }
    }
}

```

Output:



```

file:///D:/AkhilMittal/Articles and Blogs/OOPS/OOPS/Indexers/bin/Debug/Indexers.EXE
I am in set : Value is 20 and indexValue is name
I am in get and indexValue is name
20

```

The “this” property i.e. indexers have return value. In our example the return value was integer. The square brackets along with “this” can also hold other data types and not only integer. In the above mentioned example I tried to explain this using string parameter type for “this” : `public int this[string indexValue]`,

The string parameter “indexValue” has a value “name”, like we passed in Main method of Program.cs. So one can have more than one indexers in a class deciding what should be the data type of the parameter value of array. An indexer, like properties follow same rules of inheritance and polymorphism.

Indexers in interfaces

Like Properties and Methods, Indexers can also be declared in Interfaces.

For practical implementation, just create an interface named IIndexers having following code,

```
namespace Indexers
{
    interface IIndexers
    {
        string this[int indexerValue] { get; set; }
    }
}
```

Here, an indexer is declared with an empty get and set accessor, that returns string values.

'Now we need a class that implements this interface. You can define a class of your choice and implement that through IIndexers interface,

Indexer.cs:

```
using System;

namespace Indexers
{
    public class IndexerClass:IIndexers
    {
        readonly string[] _nameList = { "AKhil","Bob","Shawn","Sandra" };

        public string this[int indexerValue]
        {
            get
            {
                return _nameList[indexerValue];
            }
            set
            {
                _nameList[indexerValue] = value;
            }
        }
    }
}
```

The class has a default array of strings that hold names. Now we can implement interface defined indexer in this class to write our custom logic to fetch names on the base of indexerValue. Let's call this in our main method,

Program.cs:

```
using System;

namespace Indexers
{
    class Program
    {
        static void Main(string[] args)
        {
            IIndexers iIndexer=new IndexerClass();
        }
    }
}
```

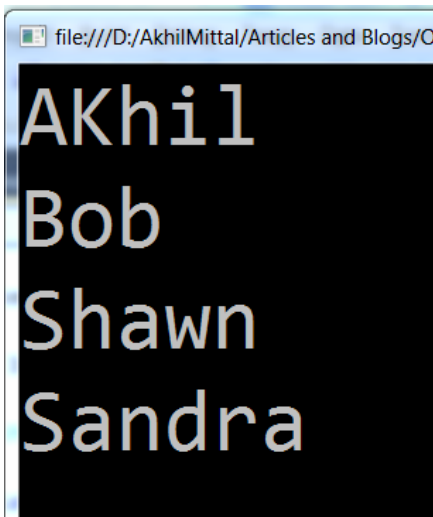


```

        Console.WriteLine(iIndexer[0]);
        Console.WriteLine(iIndexer[1]);
        Console.WriteLine(iIndexer[2]);
        Console.WriteLine(iIndexer[3]);
        Console.ReadLine();
    }
}

```

Run the application. Output,



In main method, we took Interface reference to create an object of IndexerClass, and we accessed that object array through indexer values like an array. It gives the names one by one.

Now if I want to access “set” accessor as well, I can easily do that. To check this, just add two more lines where you set the value in indexer,

```

iIndexer[2] = "Akhil Mittal";
Console.WriteLine(iIndexer[2]);

```

I set the value of 2nd element as a new name, let's see the output,



Indexers in Abstract class

Like we used indexers in Interfaces, we can also use indexers in abstract class. I'll use the same logic of source code that we used in interfaces, so that you can relate how it works in abstract class as well. Just define a new class that should be abstract and should contain an abstract indexer with empty get and set,

AbstractBaseClass:

```
namespace Indexers
{
    public abstract class AbstractBaseClass
    {
        public abstract string this[int indexerValue] { get; set; }
    }
}
```

Define derived class, inheriting from abstract class,

IndexerClass:

We here use override in indexer to override the abstract indexer declared in abstract class.

```
using System;

namespace Indexers
{
    public class IndexerClass:AbstractBaseClass
    {
        readonly string[] _nameList = { "AKhil", "Bob", "Shawn", "Sandra" };

        public override string this[int indexerValue]
        {
            get
            {
                return _nameList[indexerValue];
            }
            set
            {
                _nameList[indexerValue] = value;
            }
        }
    }
}
```

Program.cs:

We'll use reference of abstract class to create an object of Indexer class.

```
using System;

namespace Indexers
```

```

{
    class Program
    {
        static void Main(string[] args)
        {
            AbstractBaseClass absIndexer=new IndexerClass();
            Console.WriteLine(absIndexer[0]);
            Console.WriteLine(absIndexer[1]);
            Console.WriteLine(absIndexer[2]);
            Console.WriteLine(absIndexer[3]);
            absIndexer[2] = "Akhil Mittal";
            Console.WriteLine(absIndexer[2]);

            Console.ReadLine();
        }
    }
}

```

Output:



All of the above code is self-explanatory. You can explore more scenarios by yourself for more detailed understanding.

Indexer Overloading

Indexer.cs

```

using System;

namespace Indexers
{
    public class Indexer
    {
        public int this[int indexerValue]
        {

```

```

        set
        {
            Console.WriteLine("Integer value " + indexerValue + " " + value);
        }
    }

    public int this[string indexerValue]
    {
        set
        {
            Console.WriteLine("String value " + indexerValue + " " + value);
        }
    }

    public int this[string indexerValue, int indexerintValue]
    {
        set
        {
            Console.WriteLine("String and integer value " + indexerValue + " " +
indexerintValue + " " + value);
        }
    }
}
}

```

Program.cs:

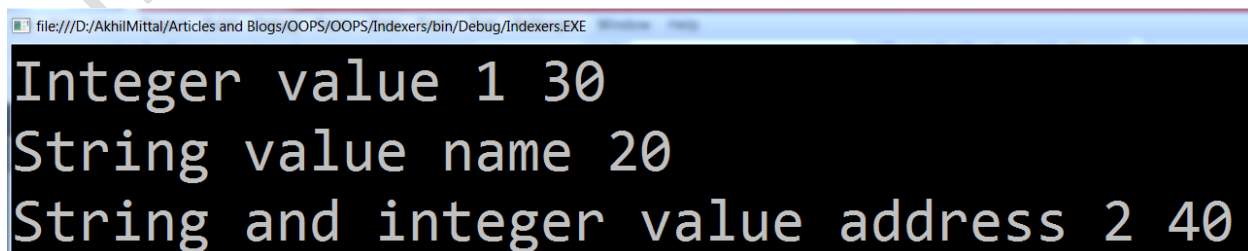
```

using System;

namespace Indexers
{
    class Program
    {
        static void Main(string[] args)
        {
            Indexer indexer=new Indexer();
            indexer[1] = 30;
            indexer["name"]=20;
            indexer["address",2] = 40;
            Console.ReadLine();
        }
    }
}

```

Output:



```

file:///D:/AkhiMittal/Articles and Blogs/OOPS/OOPS/Indexers/bin/Debug/Indexers.EXE
Integer value 1 30
String value name 20
String and integer value address 2 40

```

In above example, we see that an indexer's signature is actually count of actual parameters and data types irrespective of the names of the arguments/parameters or return value of the indexers. This allows us to overload indexers like we do in method overloading. You can read more about method overloading in <http://www.codeproject.com/Articles/771455/Diving-in-OOP-Day-Polymorphism-and-Inheritance-Part>. Here now we have overloaded indexers that takes integer, string integer and string combined as actual parameters. Like methods cannot be overloaded on the basis of return types, so indexers are. Indexers follow same methodology of overload like methods do.

Point to remember:

Like indexers, we cannot overload properties. Properties are more like knowing by name and indexers on the other hand is more like knowing by signature.

Static Indexers?

In the example that we discussed in last section, just add a static keyword to the indexer signature,

```
public static int this[int indexerValue]
{
    set
    {
        Console.WriteLine("Integer value " + indexerValue + " " + value);
    }
}
```

Compile the program; we get a compile time error,

Error The modifier 'static' is not valid for this item

The error clearly indicates that an indexer cannot be marked static. An indexer can only be a class instance member but not static, on the other hand a property can be static too.

Point to remember:

Properties can be static but indexers cannot be.

Inheritance/Polymorphism in Indexers

Indexer.cs

```
using System;

namespace Indexers
{
    public class IndexerBaseClass
    {
```

```

        public virtual int this[int indexerValue]
        {
            get
            {
                Console.WriteLine("Get of IndexerBaseClass; indexer value: " + indexerValue);
                return 100;
            }
            set
            {
                Console.WriteLine("Set of IndexerBaseClass; indexer value: " + indexerValue +
                    " set value " + value);
            }
        }
    }
}
public class IndexerDerivedClass:IndexerBaseClass
{
    public override int this[int indexerValue]
    {
        get
        {
            int dValue = base[indexerValue];
            Console.WriteLine("Get of IndexerDerivedClass; indexer value: " +
                indexerValue + " dValue from base class indexer: " + dValue);
            return 500;
        }
        set
        {
            Console.WriteLine("Set of IndexerDerivedClass; indexer value: " +
                indexerValue + " set value " + value);
            base[indexerValue] = value;
        }
    }
}
}
}

```

Program.cs:

```

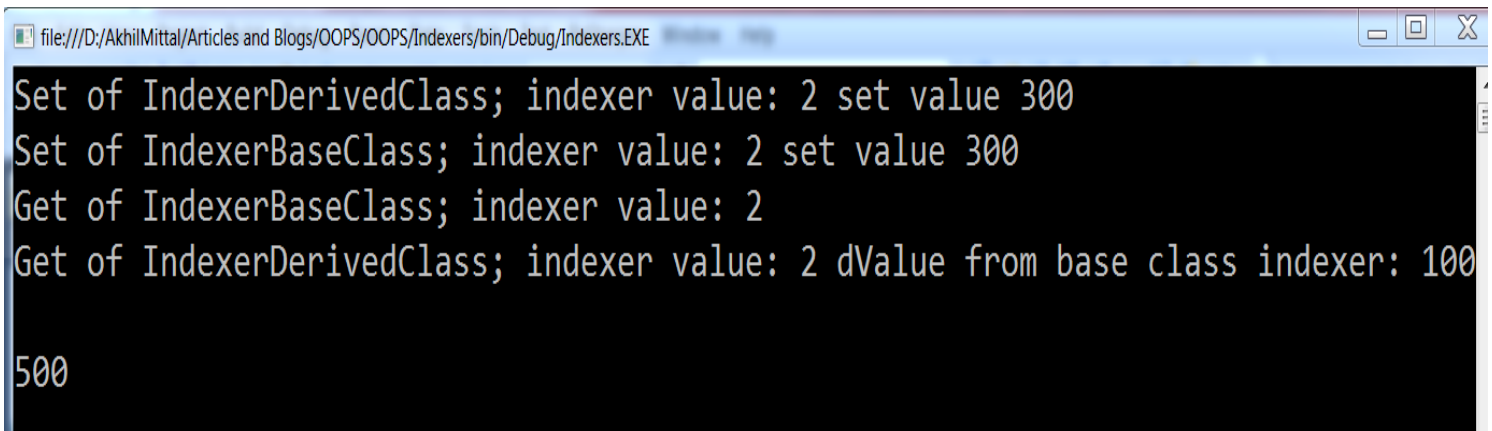
using System;

namespace Indexers
{
    class Program
    {
        static void Main(string[] args)
        {
            IndexerDerivedClass indexDerived=new IndexerDerivedClass();
            indexDerived[2] = 300;
            Console.WriteLine(indexDerived[2]);
            Console.ReadLine();
        }
    }
}

```

```
}  
}
```

Output:



```
file:///D:/AkhilMittal/Articles and Blogs/OOPS/OOPS/Indexers/bin/Debug/Indexers.EXE  
Set of IndexerDerivedClass; indexer value: 2 set value 300  
Set of IndexerBaseClass; indexer value: 2 set value 300  
Get of IndexerBaseClass; indexer value: 2  
Get of IndexerDerivedClass; indexer value: 2 dValue from base class indexer: 100  
500
```

The example code taken above explains run time polymorphism and inheritance in indexers. I created a base class named `IndexerBaseClass` having an indexer with its own get and set like we discussed in prior examples. There after a derived class is created named `IndexerDerivedClass`, this derives from `IndexerBaseClass` and overrides “this” indexer from base class, note that base class indexer is marked virtual, so we can override it in derived class by marking it “override” in derived class. The example makes call to indexer of base class. Sometimes when we need to override code in derived class in the derived class, we may require the base class indexer should be called first. This is just a situation. The same rule of run time polymorphism applies here, we declare base class indexer and virtual and derived class one as override. In “set” accessor of derived class, we can call base class indexer as `base[indexerValue]`. Also this value is used to initialize the derived class indexer as well. So the value is stored in “value” keyword too. So, `indexDerived[2]` in `Main()` method of `Program.cs` gets replaced to `base[2]` in “set” accessor. Whereas In “get” accessor it is vice versa, we require to put `base[indexerValue]` to right hand side of equal sign. The “get” accessor in base class returns a value, i.e. 100, which we get in `dValue` variable.

.Net Framework and Indexers

Indexers play a crucial role in .net framework. Indexers are widely used in .Net Framework inbuilt classes, libraries such as collections and enumerable. Indexers are used in collections that are searchable like Dictionary, Hashtable, List, ArrayList etc.

Point to remember

Dictionary in C# largely uses indexers to have a starting parameter as an indexer argument.

Classes like ArrayList and List use indexers internally to provide functionality of arrays for fetching and using the elements.

Properties vs Indexers

I have already explained a lot about properties and indexers, to summarize, let me point to an [MSDN](#) link for better understanding,

Property	Indexer
Allows methods to be called as if they were public data members.	Allows elements of an internal collection of an object to be accessed by using array notation on the object itself.
Accessed through a simple name.	Accessed through an index.
Can be a static or an instance member.	Must be an instance member.
A get accessor of a property has no parameters.	A get accessor of an indexer has the same formal parameter list as the indexer.
A set accessor of a property contains the implicit value parameter.	A set accessor of an indexer has the same formal parameter list as the indexer, and also to the value parameter.
Supports shortened syntax with Auto-Implemented Properties (C# Programming Guide) .	Does not support shortened syntax.

Table taken from MSDN: <https://msdn.microsoft.com/en-us/library/4bsztef7.aspx>

Conclusion:

With this article we completed almost all the scenarios related to an indexer. We did a lot of hands-on lab to clear our concepts. I hope my readers now know by heart about these basic concepts and will never forget them. These may also help you in cracking C# interviews.



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](#).

<http://csharpnpulse.blogspot.in/>