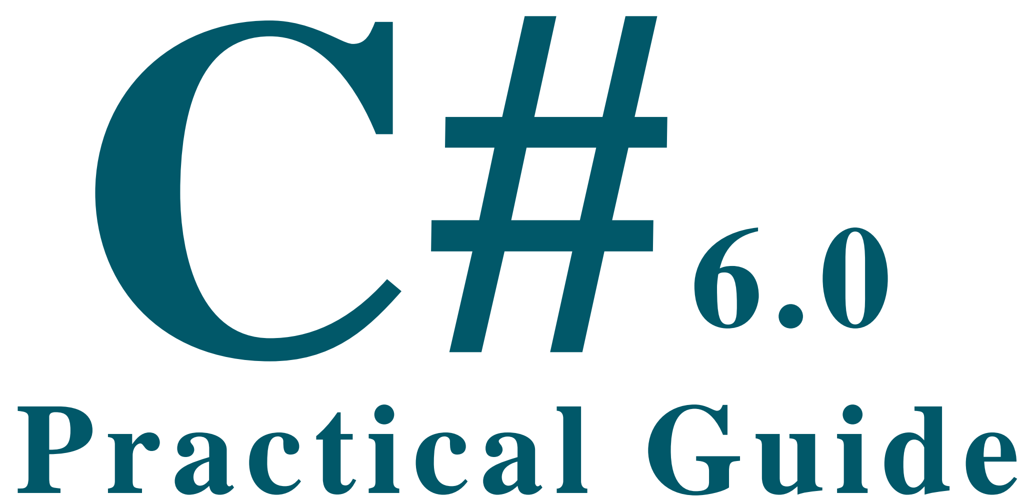


By: Mukesh Kumar

www.mukeshkumar.net

Disclaimer & Copyright

Copyright © 2016 by mukeshkumar.net

All rights reserved. Share this eBook as it is, don't reproduce, republish, change or copy. This is a free eBook and you are free to give it away (in unmodified form) to whomever you wish. No part of this publication may be reproduced, distributed, or transmitted in any form or by any means, including photocopying, recording, or other electronic or mechanical methods, without the prior written permission from the author.

The information provided within this eBook is for general informational purposes only. While we try to keep the information up-to-date and correct, there are no representations or warranties, express or implied, about the completeness, accuracy, reliability, suitability or availability with respect to the information, products, services, or related graphics contained in this eBook for any purpose. Any use of this information is at your own risk

The methods describe within this eBook are the author's personal thoughts. They are not intended to be a definitive set of instructions for this project. You may discover there are other methods and materials to accomplish the same end result.

Author

Mukesh Kumar (Microsoft MVP)

About Author

He is a **Software Developer, Microsoft MVP** having Master's degree in Computer Science. He is also **C# Corner MVP, Blogger, Writer** and has more than 4+ Years of extensive experiences with designing and developing enterprise scale application on Microsoft Technologies like C#, Asp.Net, MVC, WCF, Web API, JQuery, AngularJS, LINQ etc.

He is passionate author on www.mukeshkumar.net and believes in **"Think for new"**.

He will be happy to hear from his readers, kindly provides your valuable feedbacks & comments!

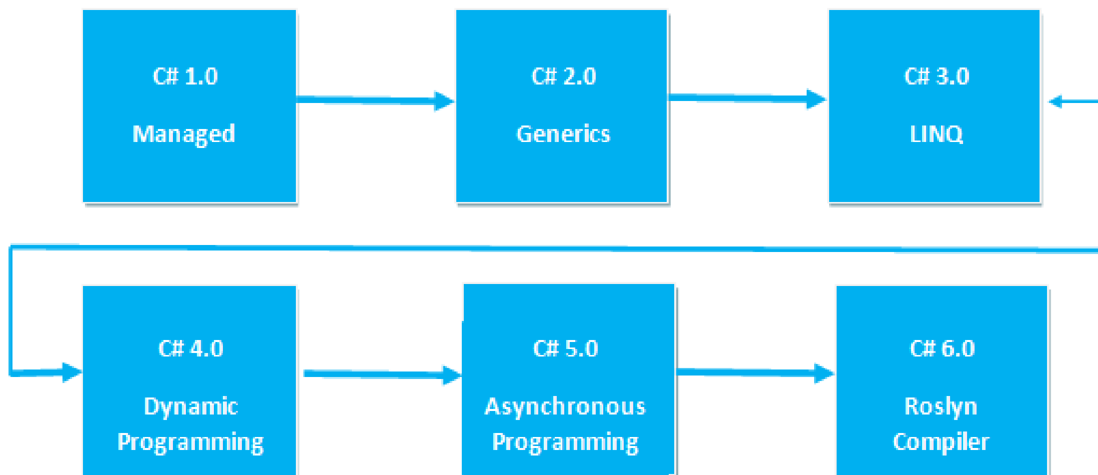
He can be contacted by email either **info@mukeshkumar.net**. You can also follow him on Facebook, Twitter, LinkedIn and Google+.

 ©2016 by Mukesh Kumar

C# Evolution

C# introduced in 2000 with Visual Studio .Net. It became very popular because it gives you facilities to design and develop enterprise application in Object Oriented Programming Manner with .Net Framework. C# 6.0 is the evolution from C# 1.0. C# was introduced with different versions and each version has their own enhance features.

But C# 6.0 has not introduced any new feature but it has introduced so many features which can improve our coding time.



As we know, C# 6.0 was introduced with Visual Studio 2015. There are several features introduced in several versions of C#, like LINQ was introduced with C# 3.0. Dynamic features were introduced with C# 4.0. Async and Await were introduced with C# 5.0 etc.

In **C# 6.0**, the most emphasis was given to syntactical improvements rather than new features to add on. These features will help us to write the code faster and in a simple way.

 ©2016 by Mukesh Kumar

We can download Visual Studio 2015 or if you have Visual Studio 2013, you need to download Roslyn Package from **here**. Roslyn is the .NET compiler, which is an open source and integrated with Visual Studio 2015.

Several new syntactical improvements with C# 6.0 were introduced and these are as follows:

1. Using Static
2. Auto property Initializer
3. Index Initializer
4. String Interpolation
5. Expression Bodied Members
6. Getter Only Auto Properties
7. Exception Filters
8. Null Conditional Operators
9. nameOf Expression
10. Await in Catch/Finally Block

 ©2016 by Mukesh Kumar

Using Static

Now, we can define the namespace via using static to access the property or the method of the static class. We need to add only using static directive before the namespace of the class. Before C# 6.0, we use ReadLine(), WriteLine() methods of the Console class using defining System.Console.ReadLine() and System.Console.WriteLine() explicitly.

As per MSDN

*"Using is now supported on specific classes in a feature called the using static directive, which renders static methods available in global scope**, without a type prefix of any kind"*

So in C# 6.0, we use it like below while declaring the namespace. It is because, System.Console is static class.

```
using static System.Console;
```

When we define the System.Console class with static in using section, we can access all the properties and methods of the static class System.Console without writing the Console class name again and again.

After using "Using static", we can write our code simple as **"WriteLine("XYZ");"** rather than **"Console.WriteLine("XYZ");"**

Given below is the example where you will see what we did in C# 5.0 and what we can do in C# 6.0.

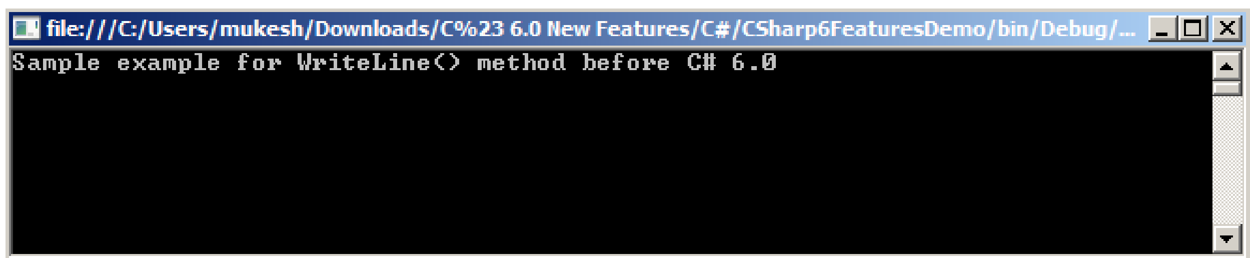
Before C# 6.0

```
using System;
namespace CSharp6FeaturesDemo
{
```


©2016 by Mukesh Kumar

```
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Sample example for WriteLine() method before C# 6.0");
            Console.ReadLine();
        }
    }
}
```

Before C# 6.0, For using the WriteLine() method, we have to use Console.WriteLine(). WriteLine() method cannot be accessed without the reference of the Console class. After running the program, the output is as following.



In C# 6.0

C# 6.0 provides the flexibility to use static class without the reference of the class every time. We only need to define it in using static statement and we can access it.

```
using System;
using static System.Console;
using static CSharp6FeaturesDemo.MyClass;
namespace CSharp6FeaturesDemo
{
    class Program
    {
        static void Main(string[] args)
        {
```

 ©2016 by Mukesh Kumar

```
            WriteLine("This is a demo for C# 6.0 New Features: Using Static");
            Hello();
            ReadLine();
        }
    }
    static class MyClass
    {
        public static void Hello()
        {
            WriteLine("This is Main Method");
        }
    }
}
```

We can also apply using static directive to our own custom static classes to access their static members. In the example given above, we can see Hello() method which is a static method defined in static class MyClass.

```
static class MyClass
{
    public static void Hello()
    {
        WriteLine("This is Main Method");
    }
}
```

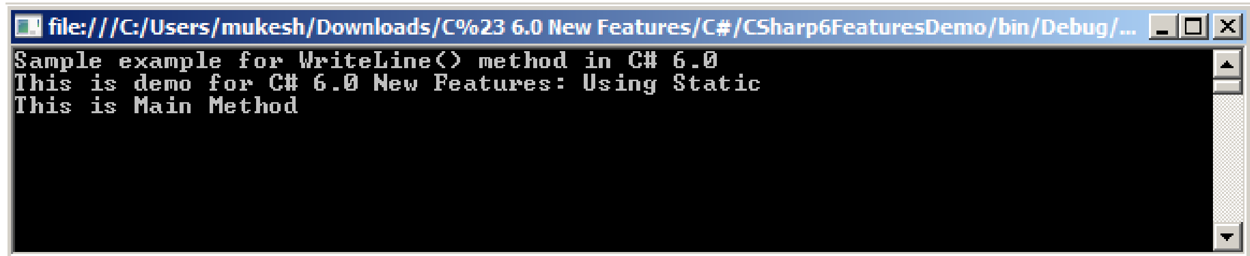
So, Using Static Directive Reduces Noise within Your Code like why to write the static class name Console again and again and it also reduce your development time.

Here, we only define the namespace in using statement with static keyword, as shown below:

```
using static CSharp6FeaturesDemo.MyClass;
```

 ©2016 by Mukesh Kumar

When we run the program, output will be as like following.



See the following image, which shows clear picture about "using static" term.

```
using System;

namespace ConsoleApplication1
{
    1 reference
    static class TestClass
    {
        1 reference
        public static void TestMethod()
        {
            Console.WriteLine("I am Test Method.");
            Console.ReadLine();
        }
    }
    0 references
    class Program
    {
        0 references
        static void Main(string[] args)
        {
            Console.WriteLine("Square root of 16 is " + Math.Sqrt(16));
            TestClass.TestMethod();
        }
    }
}
```

Without Using Static

```
using System;
using static System.Console;
using static System.Math;

namespace ConsoleApplication1
{
    1 reference
    static class TestClass
    {
        1 reference
        public static void TestMethod()
        {
            WriteLine("I am Test Method.");
            ReadLine();
        }
    }
    0 references
    class Program
    {
        0 references
        static void Main(string[] args)
        {
            WriteLine("Square root of 16 is " + Sqrt(16));
            TestClass.TestMethod();
        }
    }
}
```

With Using Static

©2016 by Mukesh Kumar

Pay Attention in Case of Ambiguity or avoid using static directive would be better in some case.

Since sometime using static directive can break your code because eliminating the type qualifier doesn't significantly reduce the clarity of the code, even though there is less code. Think of the scenario below where the code will certainly break

```
using static System.IO.Directory
using static System.IO.File
using System;
namespace CSharp6FeaturesDemo
{
    class Program
    {
        private static void Encrypt(string filename)
        {
            //Following Exists method exists in both the namespaces
System.IO.Directory and System.IO.File
            if (!Exists(filename)) // LOGIC ERROR: Using Directory rather than File
            {
                throw new ArgumentException("The file does not exist.",
                  nameof(filename));
            }
            //Other logic goes here.
        }
    }
}
```

In above the code calls the **Directory.Exists** but **File.Exists** is what's needed here. In other words, although the code is certainly readable, it's incorrect and, at least in this case, avoiding the using static syntax is probably better.

An additional feature of the using static directive is its behavior with extension methods as Extension methods aren't moved into global scope. But there are ways to do it in c# 6.

Pay attention to "static methods available in global scope" as extension methods are not part of global scope. So we discuss more on Using static with Extension methods later in the book.

 ©2016 by Mukesh Kumar

Auto Property Initializer

To access the internal members of a class, we use the properties. Properties have their getter and setter methods. Before C# 6.0, we could not directly define the value of the property. To do this, we basically used the local member variables but C# 6.0 provides the flexibility to initialize the value.

With C# 6.0, we can directly assign the value of the property at the time of defining the new property.

Before C# 6.0

Earlier, we generally used constructor to initialize the value of the Property. See the following example, as we can see that we have created multiple properties and to initialize the value, constructor will help.

```
using System;
using static System.Console;
namespace CSharp6FeaturesDemo
{
    public class Program
    {
        static void Main(string[] args)
        {
            Employee emp = new Employee();
            Console.WriteLine("Employee Id is " + emp.EmployeeId);
            Console.WriteLine("Employee Full Name is " + emp.FullName);
            Console.ReadLine();
        }
    }
    public class Employee
    {
        public Guid EmployeeId { get; set; }
```

 ©2016 by Mukesh Kumar

```
        public string FirstName { get; set; }
        public string LastName { get; set; }

        public string FullName { get; set; }
        public Employee()
        {
          //Before C# 6.0, Initialize the values in Constructor
            EmployeeId = Guid.NewGuid();
            FirstName = "Mukesh";
            LastName = "Kumar";
            FullName = string.Format("{0} {1}", FirstName, LastName);
        }
    }
}
```

In C# 6.0

C# 6.0 is very smooth and we don't need to worry about how and where we will initialize the value of the property. You can directly assign the value of the property after = sign. It will not give you any type of exception and run smoothly.

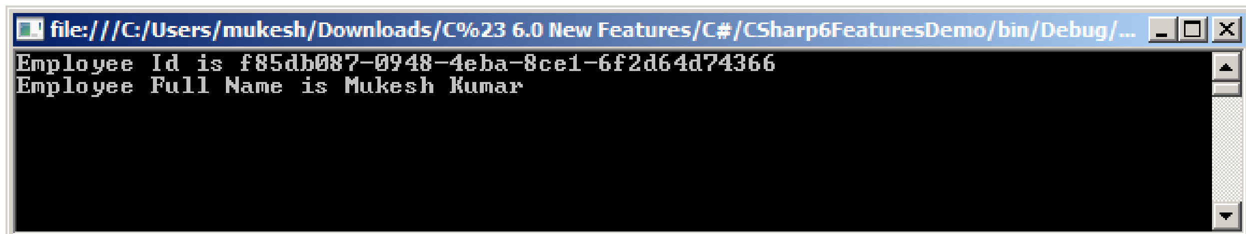
In the example given below, we can see that **EmployeeId** is generating new **GUID**. It is clearly visible that **FirstName** and **LastName** are initialized with their values.

```
using System;
using static System.Console;
namespace CSharp6FeaturesDemo
{
    public class Program
    {
        static void Main(string[] args)
        {
            Employee emp = new Employee();
            WriteLine("Employee Id is " + emp.EmployeeId);
            WriteLine("Employee Full Name is " + emp.FullName);
```


©2016 by Mukesh Kumar

```
            ReadLine();
        }
    }
    public class Employee
    {
        //In C# 6.0, Initialize the values directly to properties
        public Guid EmployeeId { get; set; } = Guid.NewGuid();
        public string FirstName { get; set; } = "Mukesh";
        public string LastName { get; set; } = "Kumar";
        public string FullName { get { return string.Format("{0} {1}", FirstName,
LastName); } }

    }
}
```

When we run the sample program, the output will be as following image shown.



 ©2016 by Mukesh Kumar

Index Initializer

C# 6.0 provides the new way to initialize the collection. We can create index based collections like dictionaries, hash tables etc. As we know that dictionary is the key-value pair and we need to assign the values for their corresponding keys. We have some different way to create the key/value pair before c# 6.0.

See the following example how we use the key\value pair for a dictionary object in C# before C# 6.0.

Before C# 6.0

```
using System;
using System.Collections.Generic;
namespace CSharp6FeaturesDemo
{
    public class Program
    {
        static void Main(string[] args)
        {
            Dictionary<int, string> myDictionary = new Dictionary<int, string>() {
                {1, "Mukesh Kumar"},
                {2, "Rahul Rathor"},
                {3, "Yaduveer Saini"},
                {4, "Banke Chamber"}
            };

            foreach (var item in myDictionary)
            {
                Console.WriteLine("The " + item.Key + " Number Employee is " +
item.Value + "\n");
            }
            Console.ReadLine();
        }
```


©2016 by Mukesh Kumar

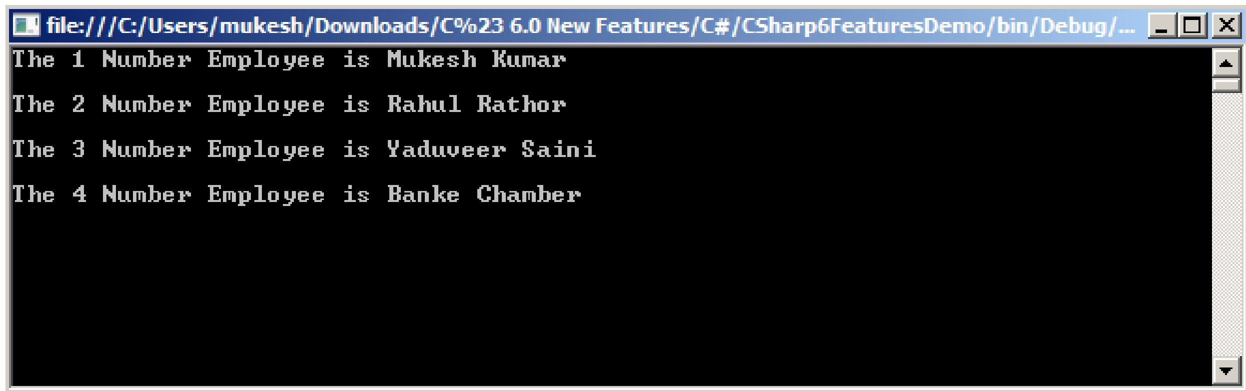
```
    }
}
```

In C# 6.0

But with C# 6.0, we logically define that the value for index 1 is "Mukesh Kumar" and so on. You can see the following example which clears your all doubts.

```
using System;
using System.Collections.Generic;
using static System.Console;
namespace CSharp6FeaturesDemo
{
    public class Program
    {
        static void Main(string[] args)
        {
            Dictionary<int, string> myDictionary = new Dictionary<int, string>()
            {
                [1] = "Mukesh Kumar",
                [2] = "Rahul Rathor",
                [3] = "Yaduveer Saini",
                [4] = "Banke Chamber"
            };

            foreach (var item in myDictionary)
            {
                WriteLine("The " + item.Key + " Number Employee is " + item.Value +
"\n");
            }
            ReadLine();
        }
    }
}
```

 ©2016 by Mukesh Kumar



©2016 by Mukesh Kumar

String Interpolation

For string concatenation, we generally use **String.Format** where we pass the array of the index and then pass the parameters value for it. It is sometimes very confusing when we are working with multiple parameters value with different index of position.

Following is the simple example of getting the full name to use first name and last name. Here we are using the **String.Format** and after that we need to specify the index array for the parameters value.

Before C# 6.0

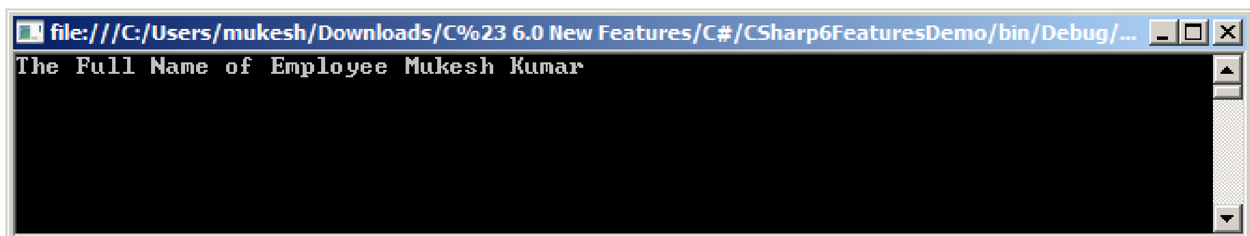
```
using System;
using System.Collections.Generic;

namespace CSharp6FeaturesDemo
{
    public class Program
    {
        static void Main(string[] args)
        {
            string firstName = "Mukesh";
            string lastName = "Kumar";

            Console.WriteLine("The Full Name of Employee " + string.Format("{0} {1}", 
firstName, lastName));
            Console.ReadLine();
        }
    }
}
```

C# 6.0 has introduced a very great feature where we don't need to pass the array index, but we can direct pass our variable but be sure we are using the **$ sign** before to start.

 ©2016 by Mukesh Kumar

In C# 6.0

```
using System;
using System.Collections.Generic;
using static System.Console;
namespace CSharp6FeaturesDemo
{
    public class Program
    {
        static void Main(string[] args)
        {
            string firstName = "Mukesh";
            string lastName = "Kumar";

            WriteLine($"The Full Name of Employee {firstName} {lastName}");
            ReadLine();
        }
    }
}
```



 ©2016 by Mukesh Kumar

Expression Bodied Members

Lambda Expressions are very useful when you are working with **LINQ** query. But in C# 6.0 you can use lambda expressions with properties and functions to increase your productivity and write clean code.

Using lambda expressions, you can define the method in one line, it means use only for simple and short logical methods or properties. I believe when you are working with large application and you need to get single value based on their condition then there we can use the expression bodied members. We can directly create a function and also get the output based on their parameters values.

Before C# 6.0

Earlier we generally use to create a separate method to complete any simple task; it might be one liner or more than one line. But it gets quite complex and takes time. See the following example where the two methods **GetFullName** and **AddTwoNumber** are doing simple task like concatenating two strings and add two numbers. But to complete this task, we had written two separate methods.

```
using System;
using System.Collections.Generic;

namespace CSharp6FeaturesDemo
{
    public class Program
    {
        public static string GetFullName(string firstName, string lastName)
        {
            return string.Format("{0} {1}", firstName, lastName);
        }
        public static int AddTwoNumber(int firstNumber, int secondNumber)
        {
            return firstNumber + secondNumber;
        }
        static void Main(string[] args)
        {
```

 ©2016 by Mukesh Kumar

```
            string firstName = "Mukesh";
            string lastName = "Kumar";
            int firstNumber = 10;
            int secondNumber = 20;

            Console.WriteLine("Full Name is "+GetFullName(firstName, lastName));
            Console.WriteLine("Sum of Two Number is "+AddTwoNumber(firstNumber, 
secondNumber));
            Console.ReadLine();
        }
    }
}
```

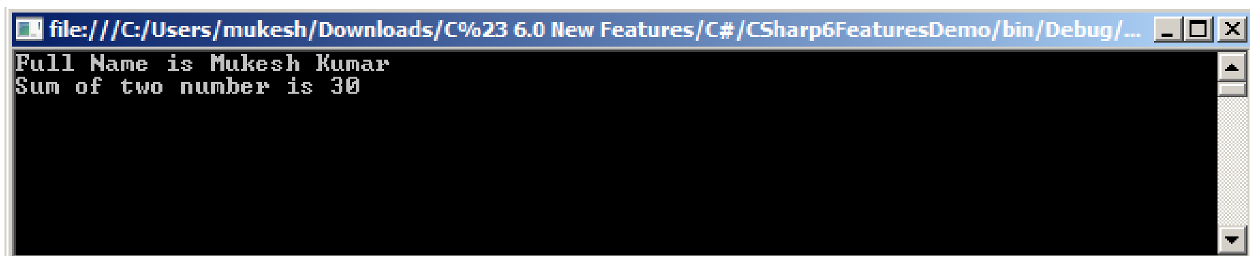
In C# 6.0

C# 6.0 does not required to create separate methods for only single task. It can be defined and completed at that time when it is defining. Using the lambda expression, we can simply write linear method.

```
using System;
using System.Collections.Generic;
using static System.Console;
namespace CSharp6FeaturesDemo
{
    public class Program
    {
        public static string GetFullName(string firstName, string lastName) => 
firstName + " " + lastName;
        public static int AddTwoNumber(int firstNumber, int secondNumber) => 
firstNumber + secondNumber;

        static void Main(string[] args)
        {
            string firstName = "Mukesh";
            string lastName = "Kumar";
            int firstNumber = 10;
            int secondNumber = 20;
```

 ©2016 by Mukesh Kumar

```
            WriteLine("Full Name is "+GetFullName(firstName, lastName));
            WriteLine("Sum of two number is "+AddTwoNumber(firstNumber, 
secondNumber));
            ReadLine();
        }
    }
}
```



 ©2016 by Mukesh Kumar

Getter Only Auto Properties

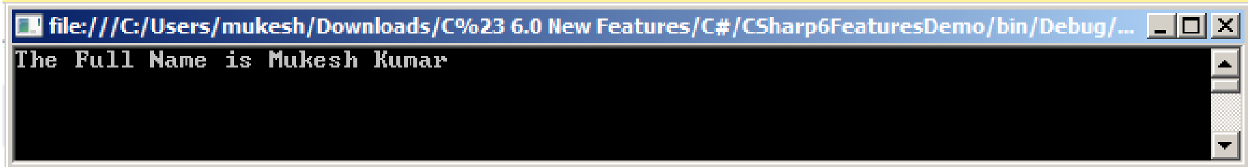
When you are working with properties, Before C# 6.0 we need to define the getter and setter both; but in C# 6.0 we can define only the getter for properties and make it read only. Before C# 6.0 you need to define the both when going to create properties. Sometimes, it is not required to create the setter but we need to define it. But there is not any restriction to make such type of coding with C# 6.0. You can only define the getter for the properties and make it read-only.

See the following example where **FirstName** and **LastName** have their getter with their values.

In C# 6.0

```
using System;
using System.Collections.Generic;

namespace CSharp6FeaturesDemo
{
    public class Program
    {
        string FirstName { get; } = "Mukesh";
        string LastName { get; } = "Kumar";

        public string FullName = string.Empty;
        public Program()
        {
            FullName = FirstName + " " + LastName;
        }
        static void Main(string[] args)
        {
            Program prog = new Program();
            Console.WriteLine("The Full Name is " + prog.FullName);
            Console.ReadLine();
        }
    }
}
```

 ©2016 by Mukesh Kumar



 ©2016 by Mukesh Kumar

Exception Filters

It is available with VB but when new compiler "**Roslyn**" is introduced. This feature added with it. Exception Filter is used to specify the catch block on the basis of some conditional argument. Generally we define only one catch block and inside that we make different conditions for throwing the different types of exceptions.

But in C# 6.0 you can check the message and show the exception message as per your conditions.

Before C# 6.0

```
using System;
using System.Collections.Generic;

namespace CSharp6FeaturesDemo
{
    public class Program
    {
        static void Main(string[] args)
        {
            int errorCode = 404;
            try
            {
                throw new Exception(errorCode.ToString());
            }
            catch (Exception ex)
            {
                if (ex.Message.Equals("404"))
                    Console.WriteLine("This is Http Error");
                else if (ex.Message.Equals("401"))
                    Console.WriteLine("This is Unathorized Error");
                else
                    Console.WriteLine("This is some different exception");

                Console.ReadLine();
```

 ©2016 by Mukesh Kumar

```
            }


        }
    }
}
```

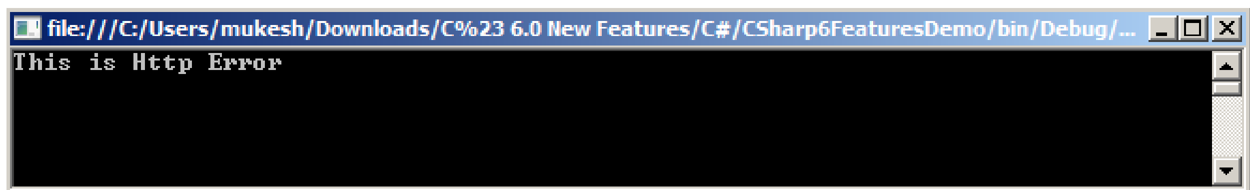
In C# 6.0

With C# 6.0, we can check the exception message and on the basis of exception can be defined in catch block. It can be used with multiple catch blocks and every catch block will be checked with own conditional message.

```
using System;
using System.Collections.Generic;
using static System.Console;
namespace CSharp6FeaturesDemo
{
    public class Program
    {
        static void Main(string[] args)
        {
            int errorCode = 404;
            try
            {
                throw new Exception(errorCode.ToString());
            }
            catch (Exception ex) when (ex.Message.Equals("404"))
            {
                WriteLine("This is Http Error");
            }
            catch (Exception ex) when (ex.Message.Equals("401"))
            {
                WriteLine("This is Unathorized Error");
            }
            catch (Exception ex) when (ex.Message.Equals("403"))
            {
```

 ©2016 by Mukesh Kumar

```
                WriteLine("Forbidden");
            }
            ReadLine();
        }

    }
}
```



You can see that on the basis of the error code the catch block is going to execute.

 ©2016 by Mukesh Kumar

Null Conditional Operators

The null-conditional operator translates to checking whether the operand is null prior to invoking the method or property. We check the object is null or not so that NullReferenceException could not occurs or handled properly. But to implement this we need to write some extra code which sometimes makes the code lengthy.

But with C# 6.0, you don't need to write some extra code, you can use null conditional operator, which is nothing but only a **question mark [?] sign.**

Before C# 6.0

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

namespace CSharp6FeaturesDemo
{
    public class Program
    {

        static void Main(string[] args)
        {
            List<Employee> employees = new List<Employee>();
            Program prog = new Program();
            if (employees.FirstOrDefault() != null)
            {
                //This code will not hit because of employees is null;
                Console.WriteLine(employees.First().Name);
            }
            else
            {
                Employee emp = new Employee();
                emp.EmployeeId = 10;
                emp.Name = "Mukesh Kumar";
                emp.Address = "New Delhi";
                employees.Add(emp);
```

 ©2016 by Mukesh Kumar

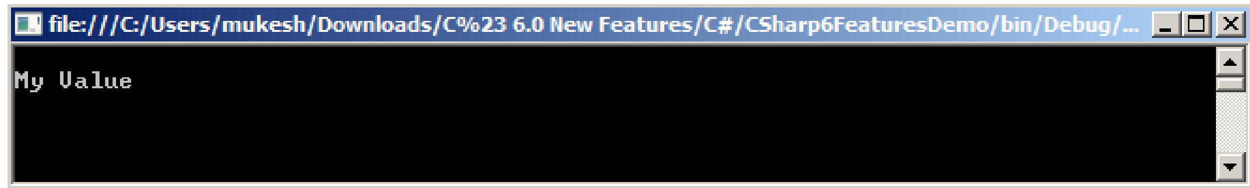
```
                Console.WriteLine(employees.First().Name);
            }
            Console.ReadLine();
        }
    }
    public class Employee
    {
        public int EmployeeId { get; set; }
        public string Name { get; set; }
        public string Address { get; set; }

    }
}
```

In C# 6.0

```
using System;
using System.Collections.Generic;
using System.Linq;
using static System.Console;

namespace CSharp6FeaturesDemo
{
    public class Program
    {

        static void Main(string[] args)
        {
            List<Employee> employees = new List<Employee>();
            Program prog = new Program();

            //No need to check null in if condition
            //null operator ? will check and return null if value is not there
            WriteLine(employees.FirstOrDefault()?.Name);

            //set the default value if value is null
            WriteLine(employees.FirstOrDefault()?.Name ?? "My Value");
```

 ©2016 by Mukesh Kumar

```
            ReadLine();
        }
    }
    public class Employee
    {
        public int EmployeeId { get; set; }
        public string Name { get; set; }
        public string Address { get; set; }

    }
}
```



See, First value is not going to print anything and also not throwing any error. It is because the? Sign is included with the statement which handles the null exception. And for the second statement the default value "My Value" will be going to print because of the value is null.

Chaining using Null Conditional Operator

What makes the null-conditional operator more useful is that it can be chained. If, for example, you invoke **string[] names = person?.Name?.Split(' ')**, Split will only be invoked if both person and **person.Name** are not null. When chained, if the first operand is null, the expression evaluation is short-circuited, and no further invocation within the expression call chain will occur.

 ©2016 by Mukesh Kumar

nameOf Expression

nameOf expression feature for C# 6.0 has introduced with CTP3 update. Sometime there is requirement to pass the parameter name as it is as string in our code. It is called Magic strings. It is normal string which is mapped to code.

In big application where thousands of line written, if we pass the string value inside the "" [double quotes] and we need to change the parameter name as well as magic string name then it is very hard to find out each and replace to everyone. But using nameOf expression, we don't need to care about magic string inside the code. Just change the parameter name at every place.

Before C# 6.0

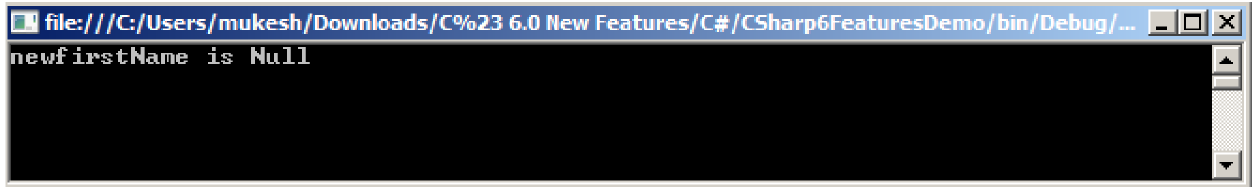
```
using System;
using System.Collections.Generic;
using System.Linq;
using static System.Console;
namespace CSharp6FeaturesDemo
{
    public class Program
    {
        private void SayHello(string firstName)
        {
            if (firstName != null)
            {
                Console.WriteLine("Hello " + firstName);
            }
            else
            {
                Console.WriteLine("firstName is Null");
            }
        }
        static void Main(string[] args)
        {
            Program objProgram = new Program();
```

 ©2016 by Mukesh Kumar

```
            objProgram.SayHello("Mukesh");
            Console.ReadLine();
        }
    }
}
```

In C# 6.0

```
using System;
using System.Collections.Generic;
using System.Linq;
using static System.Console;
namespace CSharp6FeaturesDemo
{
    public class Program
    {
        private void SayHello(string newfirstName)
        {
            if (newfirstName != null)
            {
                Console.WriteLine("Hello " + newfirstName);
            }
            else
            {
                Console.WriteLine(nameof(newfirstName) + " is Null");
            }
        }
        static void Main(string[] args)
        {
            Program objProgram = new Program();
            objProgram.SayHello(null);
            Console.ReadLine();
        }
    }
}
```



 ©2016 by Mukesh Kumar

Await in Catch/Finally Block

As we know, Async and Await are used for asynchronous programming.

In C# 6.0

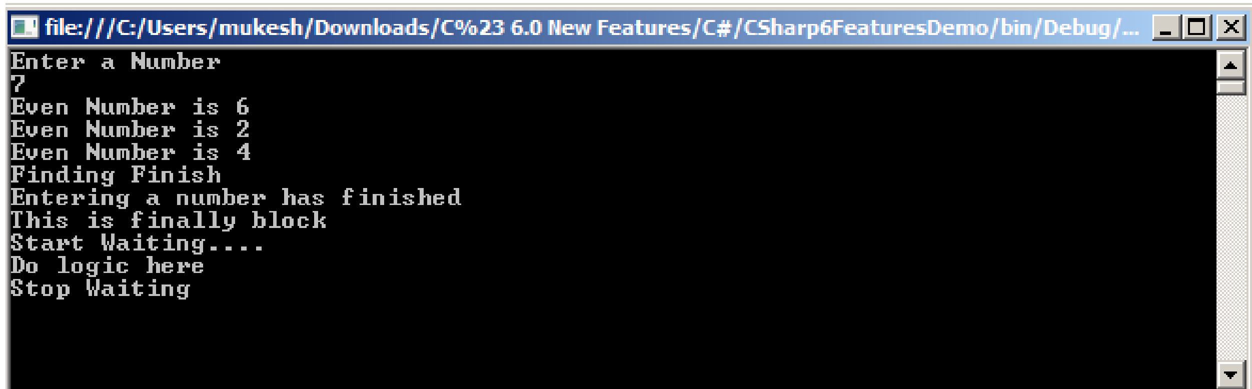
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using static System.Console;
namespace CSharp6FeaturesDemo
{
    public class Program
    {
        public static Task GetEvenNumber(int lastNumber)
        {
            return Task.Factory.StartNew(() =>
            {
                Random random = new Random();

                for (int i = 0; i < lastNumber; i++)
                {
                    var randomNumber = random.Next(1, lastNumber);
                    if (randomNumber % 2 == 0)
                    {
                        System.Threading.Thread.Sleep(1000);
                        Console.WriteLine("Even Number is " + randomNumber);
                    }
                }
                Console.WriteLine("Finding Finish");
            });
        }
        private static async void PrintEvenNumber()
        {
            try
```

 ©2016 by Mukesh Kumar

```
            {
                Console.WriteLine("Enter a Number ");
                int lastNumber = Convert.ToInt32(Console.ReadLine());
                await GetEvenNumber(lastNumber);
                Console.WriteLine("Entering a number has finished");
            }
            catch (Exception ex)
            {
                Console.WriteLine("This is catch block");
                await TestWaiting();
                Console.WriteLine(ex.Message.ToString());

            }
            finally
            {
                Console.WriteLine("This is finally block");
                await TestWaiting();
            }

        }

        private static Task TestWaiting()
        {
            return Task.Factory.StartNew(() =>
            {
                Console.WriteLine("Start Waiting....");
                // do some application log logic
               Console.WriteLine("Stop Waiting");
                System.Threading.Thread.Sleep(10000);
                Console.WriteLine("Stop Waiting");
            });

        }
        static void Main(string[] args)
        {
            Program objProgram = new Program();
```

 ©2016 by Mukesh Kumar

```
            PrintEvenNumber();
            Console.ReadLine();
        }
    }
}
```


file:///C:/Users/mukesh/Downloads/C%23 6.0 New Features/C#/CSharp6FeaturesDemo/bin/Debug/...

```
Enter a Number
7
Even Number is 6
Even Number is 2
Even Number is 4
Finding Finish
Entering a number has finished
This is finally block
Start Waiting....
Do logic here
Stop Waiting
```

Thanks

©2016 by Mukesh Kumar