

C# Concepts

Delegates

A delegate is a named type that defines a particular kind of method. Just as a class definition lays out all the members for the given kind of object it defines, the delegate lays out the method signature for the kind of method it defines.

Based on this statement, a delegate is a function pointer and it defines what that function looks like.

A delegate can be seen as a placeholder for a/some method(s).

By defining a delegate, you are saying to the user of your class, *"Please feel free to assign, any method that matches this signature, to the delegate and it will be called each time my delegate is called"*.

Typical use is of course events. All the OnEventX *delegate* to the methods the user defines.

Delegates are useful to offer to the **user** of your objects some ability to customize their behavior. Most of the time, you can use other ways to achieve the same purpose and I do not believe you can ever be **forced** to create delegates. It is just the easiest way in some situations to get the thing done.

Delegates is a type that hold references to methods with a particular parameter list and return type. Programmers can invoke the method through the delegate instance.

Syntax:

```
<access_modifier> delegate <return_data_type> delegate_name (parameters_list)
```

Where:

- The ***access_modifier*** specifies the accessibility of delegates.
- The ***return_data_type*** specifies the returning data type of output from delegates.
- The ***delegate_name*** specifies the name of delegate.
- The ***parameters_list*** specifies the inputs parameters list of delegates.

Sample:

```
public delegate int PerformCalculation(int x, int y);
```

Key point to remember

- Delegates are like C++ function pointers but are type safe.
- Delegates allow methods to be passed as parameters.
- A method must have the same return type as the delegate.
- Delegates can be used to define callback methods.
- Delegates are especially used for implementing events and the call-back methods.
- Delegates can be chained together; for example, multiple methods can be called on a single event.

Following example demonstrates declaration, instantiation, and use of a delegate that can be used to reference methods that take two integer parameters and returns an integer value.

```
namespace delegatesSample
{
    class Program
    {
        //Declaring a delegate
        public delegate int Calculator(int n1, int n2);

        static void Main(string[] args)
        {
            ////create delegate instances
            Calculator c1 = new Calculator(Add);

            //calling the methods using the delegate objects
            int result = c1(2, 3);

            Console.WriteLine(result);
            Console.Read();
        }

        public static int Add(int a1, int a2)
```

```
{  
    return a1 + a2;  
}  
}  
}
```

When the above code is compiled and executed, it produces the following result:

5

There are three types of delegates that can be used in C#.

- Single delegate
- Multicast delegate
- Generic delegate

Single Delegate

Single delegate can be used to invoke a single method.

In following example, a delegate Calculator invokes a method **Add()**.

```
namespace delegatesSample  
{  
    class Program  
    {  
        //Declaring a delegate  
        public delegate int Calculator(int n1, int n2);  
        static void Main(string[] args)  
        {  
            ////create delegate instances  
            Calculator c1 = new Calculator(Add);  
            //calling the methods using the delegate objects  
            int result = c1(2, 3);  
            Console.WriteLine(result);  
        }  
    }  
}
```

```

        Console.Read();
    }
    public static int Add(int a1, int a2)
    {
        return a1 + a2;
    }
}
}

```

When the above code is compiled and executed, it produces the following result:

```
5
```

Multicast Delegate

Multicast delegate can be used to invoke the multiple methods. The delegate instance can do multicasting (adding new method on existing delegate instance) using the "+" operator and "-" operator can be used to remove a method from a delegate instance. All methods will invoke in sequence as they are assigned.

In Following example, a delegate instance **MulPlusDel** invokes the methods **Add()**, **SubDel()** and **CrossDel()**.

```

namespace delegatesSample
{
    class Program
    {
        //Declaring a delegate
        public delegate void Calculator(int n1, int n2);
        static void Main(string[] args)
        {
            //create delegate instances
            Calculator addDel = new Calculator(Add);
            Calculator SubDel = new Calculator(Subtract);
            Calculator CrossDel = new Calculator(Multiply);
            Calculator MulPlusDel, multiMinusHiDel;
            // The three delegates, addDel, SubDel and CrossDel, are combined to MulPlusDel.
            MulPlusDel = addDel + SubDel + CrossDel;

```

```

//Remove SubDel from the MulPlusDel delegate
multiMinusHiDel = MulPlusDel - SubDel;
MulPlusDel(5, 2);
multiMinusHiDel(6, 2);

Console.Read();
}
public static void Add(int a1, int a2)
{
    Console.WriteLine(a1 + a2);
}
public static void Subtract(int a1, int a2)
{
    Console.WriteLine(a1 - a2);
}
public static void Multiply(int a1, int a2)
{
    Console.WriteLine(a1 * a2);
}
}
}

```

When the above code is compiled and executed, it produces the following result:

```

7
3
10
8
12

```

Generic Delegate

Generic Delegate was introduced in .NET **3.5** that don't require to define the delegate instance in order to invoke the methods.

There are three types of generic delegates:

- Action
- Func
- Predicate

Action Delegate

Encapsulates a method that has a single parameter and does not return a value. In other words, Action generic delegate, points to a method that takes up to 16 Parameters and returns void.

In following example, we create one Action delegate "Action<string>" with single input parameter as a string.

```
namespace delegatesSample
{
    class Program
    {
        static void Main(string[] args)
        {
            Action<string> actdel=new action<string>(ModelNumber);

            ActDel("2013");
            Console.Read();
        }
        public static void ModelNumber(string year)
        {
            Console.WriteLine("vehicle Model number = " + year);
        }
    }
}
```

When the above code is compiled and executed, it produces the following result:

```
Vehicle model number = 2013
```

Func Delegate

Encapsulates a method that has no parameters and returns a value of the type. In other words, The generic **Func** delegate is used when we want to point to a method that returns a value. **Always remember that the final parameter of Func<> is always the return value of the method.** For example **Func <int, int, string>**, this version of the Func<> delegate will take 2 int parameters and returns a string value.

Note

Last parameter of func delegate is return value of method.

In following example, we create one Func delegate Func <int, int, string> with two input parameter as an int and return value type is string as its last parameter while instantiate Func delegate.

```
namespace delegatesSample
{
    class Program
    {
        static void Main(string[] args)
        {
            Func<int, int, string> FucDel = new Func<int, int, string>(AddNumber);
            Console.WriteLine(FucDel(4, 5));
            Console.Read();
        }
        public static string AddNumber(int n1, int n2)
        {
            return "Total = " + (n1 + n2);
        }
    }
}
```

When the above code is compiled and executed, it produces the following result:

```
Total  = 9
```

Predicate Delegate

The Predicate delegate defines a method that can be called on arguments and always returns **Boolean** type result.

In following example, we create one Predicate delegate Predicate<string> with one input parameter as a string and its return bool type value.

```

namespace delegatesSample
{
    class Program
    {
        static void Main(string[] args)
        {
            Predicate<string> PreDel = new Predicate<string>(IsStringNull);

            Console.WriteLine(PreDel(""));
            Console.Read();
        }

        public static bool IsStringNull(string input)
        {
            if(string.IsNullOrEmpty(input))
            {
                return true;
            }
            else
            {
                return false;
            }
        }
    }
}

```

When the above code is compiled and executed, it produces the following result:

```
true
```

Interview Questions And Answers

Question 1 : What you mean by delegate in C#?

Answer: Delegates are type safe pointers unlike function pointers as in C++. Delegate is used to represent the reference of the methods of some return type and parameters.

Question 2 : What are the types of delegates in C#?

Answer:

Below are the uses of delegates in C# :

- Single Delegate
- Multicast Delegate
- Generic Delegate

Question 3 : What are the three types of Generic delegates in C#?

Answer:

Below are the three types of generic delegates in C# :

- Func
- Action
- Predicate

Question 4 : What are the differences between events and delegates in C#?

Answer: Main difference between event and delegate is event will provide one more of encapsulation over delegates. So when you are using events destination will listen to it but delegates are naked, which works in subscriber/destination model.

Question 5 : Can we use delegates for asynchronous method calls in C#?

Answer: Yes. We can use delegates for asynchronous method calls.

Question 6 : What are the uses of delegates in C#?

Answer:

Below are the list of uses of delegates in C# :

- Callback Mechanism
- Asynchronous Processing
- Abstract and Encapsulate method
- Multicasting

Question 7 : Define Multicast Delegate in C#?

Answer: A delegate with multiple handlers are called as multicast delegate.

http://iqdotnet.com/CSharp/CSharp_Delegates_With_Example

What is the use of ObservableCollection in .net?

ObservableCollection is a collection that allows code outside the collection be aware of when changes to the collection (add, move, remove) occur. It is used heavily in WPF and Silverlight but its use is not limited to there. Code can add event handlers to see when the collection has changed and then react through the event handler to do some additional processing. This may be changing a UI or performing some other operation.

The code below doesn't really do anything but demonstrates how you'd attach a handler in a class and then use the event args to react in some way to the changes. WPF already has many operations like refreshing the UI built in so you get them for free when using ObservableCollection

```
class Handler
{
    private ObservableCollection<string> collection;

    public Handler()
    {
        collection = new ObservableCollection<string>();
        collection.CollectionChanged += HandleChange;
    }

    private void HandleChange(object sender, NotifyCollectionChangedEventArgs e)
    {
        foreach (var x in e.NewItems)
        {
            // do something
        }

        foreach (var y in e.OldItems)
```

```

{
    //do something
}
if (e.Action == NotifyCollectionChangedAction.Move)
{
    //do something
}
}
}

```

sealed (C# Reference)

Visual Studio 2015

[Other Versions](#)



Updated: July 20, 2015

For the latest documentation on Visual Studio 2017 RC, see [Visual Studio 2017 RC Documentation](#).

When applied to a class, the `sealed` modifier prevents other classes from inheriting from it. In the following example, class `B` inherits from class `A`, but no class can inherit from class `B`.

```

class A {}
sealed class B : A {}

```

You can also use the `sealed` modifier on a method or property that overrides a virtual method or property in a base class. This enables you to allow classes to derive from your class and prevent them from overriding specific virtual methods or properties.

Example

In the following example, `Z` inherits from `Y` but `Z` cannot override the virtual function `F` that is declared in `X` and sealed in `Y`.

C#

```

class X

```

```

{
    protected virtual void F() { Console.WriteLine("X.F"); }
    protected virtual void F2() { Console.WriteLine("X.F2"); }
}
class Y : X
{
    sealed protected override void F() { Console.WriteLine("Y.F"); }
    protected override void F2() { Console.WriteLine("Y.F2"); }
}
class Z : Y
{
    // Attempting to override F causes compiler error CS0239.
    // protected override void F() { Console.WriteLine("C.F"); }

    // Overriding F2 is allowed.
    protected override void F2() { Console.WriteLine("Z.F2"); }
}

```

When you define new methods or properties in a class, you can prevent deriving classes from overriding them by not declaring them as [virtual](#).

It is an error to use the [abstract](#) modifier with a sealed class, because an abstract class must be inherited by a class that provides an implementation of the abstract methods or properties.

When applied to a method or property, the `sealed` modifier must always be used with [override](#).

Because structs are implicitly sealed, they cannot be inherited.

For more information, see [Inheritance](#).

For more examples, see [Abstract and Sealed Classes and Class Members](#).

Example

C#

```

sealed class SealedClass
{
    public int x;
    public int y;
}

class SealedTest2
{
    static void Main()
    {
        SealedClass sc = new SealedClass();
        sc.x = 110;
        sc.y = 150;
        Console.WriteLine("x = {0}, y = {1}", sc.x, sc.y);
    }
}

```

```
}  
}  
// Output: x = 110, y = 150
```

In the previous example, you might try to inherit from the sealed class by using the following statement:

```
class MyDerivedC: SealedClass {} // Error
```

The result is an error message:

```
'MyDerivedC' cannot inherit from sealed class 'SealedClass'.
```

<https://msdn.microsoft.com/en-in/library/88c54tsw.aspx>

Partial Classes and Methods (C# Programming Guide)

Visual Studio 2015

[Other Versions](#)



Updated: July 20, 2015

For the latest documentation on Visual Studio 2017 RC, see [Visual Studio 2017 RC Documentation](#).

It is possible to split the definition of a [class](#) or a [struct](#), an [interface](#) or a method over two or more source files. Each source file contains a section of the type or method definition, and all parts are combined when the application is compiled.

Partial Classes

There are several situations when splitting a class definition is desirable:

- When working on large projects, spreading a class over separate files enables multiple programmers to work on it at the same time.
- When working with automatically generated source, code can be added to the class without having to recreate the source file. Visual Studio uses this approach when it creates Windows Forms, Web service wrapper code, and so on. You can create code that uses these classes without having to modify the file created by Visual Studio.
- To split a class definition, use the [partial](#) keyword modifier, as shown here:

C#

```
public partial class Employee
{
    public void DoWork()
    {
    }
}

public partial class Employee
{
    public void GoToLunch()
    {
    }
}
```

The `partial` keyword indicates that other parts of the class, struct, or interface can be defined in the namespace. All the parts must use the `partial` keyword. All the parts must be available at compile time to form the final type. All the parts must have the same accessibility, such as `public`, `private`, and so on.

If any part is declared abstract, then the whole type is considered abstract. If any part is declared sealed, then the whole type is considered sealed. If any part declares a base type, then the whole type inherits that class.

All the parts that specify a base class must agree, but parts that omit a base class still inherit the base type. Parts can specify different base interfaces, and the final type implements all the interfaces listed by all the partial declarations. Any class, struct, or interface members declared in a partial definition are available to all the other parts. The final type is the combination of all the parts at compile time.

Note

The `partial` modifier is not available on delegate or enumeration declarations.

The following example shows that nested types can be partial, even if the type they are nested within is not partial itself.

C#

```
class Container
{
    partial class Nested
    {
        void Test() { }
    }
    partial class Nested
    {
        void Test2() { }
    }
}
```

At compile time, attributes of partial-type definitions are merged. For example, consider the following declarations:

C#

```
[SerializableAttribute]
partial class Moon { }

[ObsoleteAttribute]
partial class Moon { }
```

They are equivalent to the following declarations:

C#

```
[SerializableAttribute]
[ObsoleteAttribute]
class Moon { }
```

The following are merged from all the partial-type definitions:

- XML comments
- interfaces
- generic-type parameter attributes
- class attributes
- members

For example, consider the following declarations:

C#

```
partial class Earth : Planet, IRotate { }
partial class Earth : IRevolve { }
```

They are equivalent to the following declarations:

C#

```
class Earth : Planet, IRotate, IRevolve { }
```

Restrictions

There are several rules to follow when you are working with partial class definitions:

- All partial-type definitions meant to be parts of the same type must be modified with `partial`. For example, the following class declarations generate an error:

C#

```
public partial class A { }
//public class A { } // Error, must also be marked partial
```

- The `partial` modifier can only appear immediately before the keywords `class`, `struct`, or `interface`.
- Nested partial types are allowed in partial-type definitions as illustrated in the following example:

C#

```
partial class ClassWithNestedClass
{
    partial class NestedClass { }
}

partial class ClassWithNestedClass
{
    partial class NestedClass { }
}
```

- All partial-type definitions meant to be parts of the same type must be defined in the same assembly and the same module (.exe or .dll file). Partial definitions cannot span multiple modules.
- The class name and generic-type parameters must match on all partial-type definitions. Generic types can be partial. Each partial declaration must use the same parameter names in the same order.

- The following keywords on a partial-type definition are optional, but if present on one partial-type definition, cannot conflict with the keywords specified on another partial definition for the same type:
 - [public](#)
 - [private](#)
 - [protected](#)
 - [internal](#)
 - [abstract](#)
 - [sealed](#)
 - base class
 - [new](#) modifier (nested parts)
 - generic constraints

For more information, see [Constraints on Type Parameters](#).

Example 1

Description

In the following example, the fields and the constructor of the class, `CoOrds`, are declared in one partial class definition, and the member, `PrintCoOrds`, is declared in another partial class definition.

Code

C#

```
public partial class CoOrds
{
    private int x;
    private int y;

    public CoOrds(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
}
```

```

public partial class CoOrds
{
    public void PrintCoOrds()
    {
        Console.WriteLine("CoOrds: {0},{1}", x, y);
    }
}

class TestCoOrds
{
    static void Main()
    {
        CoOrds myCoOrds = new CoOrds(10, 15);
        myCoOrds.PrintCoOrds();

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
// Output: CoOrds: 10,15

```

Example 2

Description

The following example shows that you can also develop partial structs and interfaces.

Code

C#

```

partial interface ITest
{
    void Interface_Test();
}

partial interface ITest
{
    void Interface_Test2();
}

partial struct S1
{
    void Struct_Test() { }
}

partial struct S1
{
    void Struct_Test2() { }
}

```

Partial Methods

A partial class or struct may contain a partial method. One part of the class contains the signature of the method. An optional implementation may be defined in the same part or another part. If the implementation is not supplied, then the method and all calls to the method are removed at compile time.

Partial methods enable the implementer of one part of a class to define a method, similar to an event. The implementer of the other part of the class can decide whether to implement the method or not. If the method is not implemented, then the compiler removes the method signature and all calls to the method. The calls to the method, including any results that would occur from evaluation of arguments in the calls, have no effect at run time. Therefore, any code in the partial class can freely use a partial method, even if the implementation is not supplied. No compile-time or run-time errors will result if the method is called but not implemented.

Partial methods are especially useful as a way to customize generated code. They allow for a method name and signature to be reserved, so that generated code can call the method but the developer can decide whether to implement the method. Much like partial classes, partial methods enable code created by a code generator and code created by a human developer to work together without run-time costs.

A partial method declaration consists of two parts: the definition, and the implementation. These may be in separate parts of a partial class, or in the same part. If there is no implementation declaration, then the compiler optimizes away both the defining declaration and all calls to the method.

```
// Definition in file1.cs
partial void onNameChanged();

// Implementation in file2.cs
partial void onNameChanged()
{
    // method body
}
```

- Partial method declarations must begin with the contextual keyword [partial](#) and the method must return [void](#).
- Partial methods can have [ref](#) but not [out](#) parameters.
- Partial methods are implicitly [private](#), and therefore they cannot be [virtual](#).
- Partial methods cannot be [extern](#), because the presence of the body determines whether they are defining or implementing.
- Partial methods can have [static](#) and [unsafe](#) modifiers.

- Partial methods can be generic. Constraints are put on the defining partial method declaration, and may optionally be repeated on the implementing one. Parameter and type parameter names do not have to be the same in the implementing declaration as in the defining one.
- You can make a [delegate](#) to a partial method that has been defined and implemented, but not to a partial method that has only been defined.

<https://msdn.microsoft.com/en-IN/library/wa80x488.aspx>

extern (C# Reference)

Visual Studio 2015

[Other Versions](#)



Updated: July 20, 2015

For the latest documentation on Visual Studio 2017 RC, see [Visual Studio 2017 RC Documentation](#).

The `extern` modifier is used to declare a method that is implemented externally. A common use of the `extern` modifier is with the `DllImport` attribute when you are using Interop services to call into unmanaged code. In this case, the method must also be declared as `static`, as shown in the following example:

```
[DllImport("avifil32.dll")]
private static extern void AVIFileInit();
```

The `extern` keyword can also define an external assembly alias, which makes it possible to reference different versions of the same component from within a single assembly. For more information, see [extern alias](#).

It is an error to use the [abstract](#) and `extern` modifiers together to modify the same member. Using the `extern` modifier means that the method is implemented outside the C# code, whereas using the `abstract` modifier means that the method implementation is not provided in the class.

The `extern` keyword has more limited uses in C# than in C++. To compare the C# keyword with the C++ keyword, see [Using extern to Specify Linkage in the C++ Language Reference](#).

Example

Example 1. In this example, the program receives a string from the user and displays it inside a message box. The program uses the `MessageBox` method imported from the `User32.dll` library.

C#

```
//using System.Runtime.InteropServices;
class ExternTest
{
    [DllImport("User32.dll", CharSet=CharSet.Unicode)]
    public static extern int MessageBox(IntPtr h, string m, string c, int
type);

    static int Main()
    {
        string myString;
        Console.WriteLine("Enter your message: ");
        myString = Console.ReadLine();
        return MessageBox((IntPtr)0, myString, "My Message Box", 0);
    }
}
```

Example

Example 2. This example illustrates a C# program that calls into a C library (a native DLL).

1. Create the following C file and name it `cmdll.c`:

```
// cmdll.c
// Compile with: /LD
int __declspec(dllexport) SampleMethod(int i)
{
    return i*10;
}
```

Example

2. Open a Visual Studio x64 (or x32) Native Tools Command Prompt window from the Visual Studio installation directory and compile the `cmdll.c` file by typing **`cl /LD cmdll.c`** at the command prompt.
3. In the same directory, create the following C# file and name it `cm.cs`:

```
// cm.cs
```

```

using System;
using System.Runtime.InteropServices;
public class MainClass
{
    [DllImport("Cmdll.dll")]
    public static extern int SampleMethod(int x);

    static void Main()
    {
        Console.WriteLine("SampleMethod() returns {0}.", SampleMethod(5));
    }
}

```

Example

3. Open a Visual Studio x64 (or x32) Native Tools Command Prompt window from the Visual Studio installation directory and compile the `cm.cs` file by typing:

csc cm.cs (for the x64 command prompt)

—or—

csc /platform:x86 cm.cs (for the x32 command prompt)

This will create the executable file `cm.exe`.

4. Run `cm.exe`. The `SampleMethod` method passes the value 5 to the DLL file, which returns the value multiplied by 10. The program produces the following output:

`SampleMethod() returns 50.`

<https://msdn.microsoft.com/en-in/library/e59b22c5.aspx>

Why we need two trees, visual and logic tree

Here is how [the docs](#) explain what it means that `ContentElements` are not really in the visual tree:

Content elements (derived classes of `ContentElement`) are not part of the visual tree; they do not inherit from `Visual` and have no visual representation. In order to appear in a UI at all, a `ContentElement` must be hosted within a content host that is a `Visual`, usually a `FrameworkElement`. You can conceptualize that the content host is somewhat like a "browser" for the content and chooses how to display that content within the screen region the host controls. Once the content is hosted, the content can be made a participant in certain tree processes that are normally associated with the visual tree. Generally the `FrameworkElement` host class includes implementation code that adds any hosted `ContentElement` to the event route through subnodes of the content logical tree, even though the hosted content is not part of the true visual tree. This is

necessary so that a **ContentElement** can source a routed event that routes to any element other than itself.

So what does all this mean? Well, it means that you can't always just use **VisualTreeHelper** to traverse the visual tree. If you pass a **ContentElement** to **VisualTreeHelper**'s **GetParent** or **GetChild** methods, an exception will be thrown because **ContentElement** does not derive from **Visual** or **Visual3D**. In order to walk up the visual tree, you need to check each element along the way to see if it descends from **Visual** or **Visual3D**, and if it does not, then you must temporarily walk up the logical tree until you encounter another visual object. For example, here's some code which walks up to the root element in a visual tree:

[Hide](#) [Copy Code](#)

```
DependencyObject FindVisualTreeRoot(DependencyObject initial)
{
    DependencyObject current = initial;
    DependencyObject result = initial;

    while (current != null)
    {
        result = current;
        if (current is Visual || current is Visual3D)
        {
            current = VisualTreeHelper.GetParent(current);
        }
        else
        {
            // If we're in Logical Land then we must walk
            // up the logical tree until we find a
            // Visual/Visual3D to get us back to Visual Land.
            current = LogicalTreeHelper.GetParent(current);
        }
    }

    return result;
}
```

This code walks up the logical tree when necessary, as seen in the **else** clause. This is useful if, say, the user clicks on a **Run** element within a **TextBlock** and in your code you need to walk up the visual tree starting at that **Run**. Since the **Run** class descends from **ContentElement**, the **Run** is not "really" in the visual tree so we need to walk up out of "Logical Land" until we encounter the **TextBlock** which contains the **Run**. At that point we will be back in "Visual Land" since **TextBlock** is not a **ContentElement** subclass (i.e. it is a real part of a visual tree).

The Logical Tree

The logical tree represents the essential structure of your UI. It closely matches the elements you declare in XAML, and excludes most visual elements created internally to help render the elements you declared. WPF uses the logical tree to determine several things including dependency property value inheritance, resource resolution, and more.

Working with the logical tree is not nearly as clear-cut as the visual tree. For starters, the logical tree can contain objects of any type. This differs from the visual tree, which only contains instances of `DependencyObject` subclasses. When working with the logical tree, you must keep in mind a leaf node in the tree (a terminus) can be of any type. Since `LogicalTreeHelper` only works with `DependencyObject` subclasses, you need to be careful about type checking the objects while walking down the tree. For example:

[Hide](#) [Copy Code](#)

```
void WalkDownLogicalTree(object current)
{
    DoSomethingWithObjectInLogicalTree(current);

    // The logical tree can contain any type of object, not just
    // instances of DependencyObject subclasses. LogicalTreeHelper
    // only works with DependencyObject subclasses, so we must be
    // sure that we do not pass it an object of the wrong type.
    DependencyObject depObj = current as DependencyObject;

    if (depObj != null)
        foreach(object logicalChild in LogicalTreeHelper.GetChildren(depObj))
            WalkDownLogicalTree(logicalChild);
}
```

A given `Window/Page/Control` will have one visual tree, but can contain any number of logical trees. Those logical trees are not connected to each other, so you cannot just use `LogicalTreeHelper` to navigate between them. In this article, I refer to the top level control's logical tree as the "main logical tree" and all of the other logical trees within it as "logical islands". Logical islands are really just regular logical trees, but I think that the term "island" helps to convey the fact that they are not connected to the main logical tree.

This weirdness can all be boiled down to one word: **templates**.

Controls and data objects have no intrinsic visual appearance; instead they rely on templates to explain how they should render. A template is like a cookie-cutter which can be "expanded" to create real live visual elements used to render something. The elements that are part of an expanded template, hereafter referred to as "template elements", form their own logical tree which is disconnected from the logical tree of the object for which they were created. Those little logical trees are what I refer to as "logical islands" in this article.

You have to write extra code if you need to jump between logical islands/trees. Bridging those logical islands together, while walking **up** logical trees, involves making use of the `TemplatedParent` property of `FrameworkElement` or `FrameworkContentElement`. `TemplatedParent` returns the element which has the template applied to it, and, thus, contains a logical island. Here is a method which finds the `TemplatedParent` of any element:

<https://www.codeproject.com/Articles/21495/Understanding-the-Visual-Tree-and-Logical-Tree-in>

<http://www.informit.com/articles/article.aspx?p=2115888&seqNum=2>

Fundamentals of Garbage Collection

In the common language runtime (CLR), the garbage collector serves as an automatic memory manager. It provides the following benefits:

- Enables you to develop your application without having to free memory.
- Allocates objects on the managed heap efficiently.
- Reclaims objects that are no longer being used, clears their memory, and keeps the memory available for future allocations. Managed objects automatically get clean content to start with, so their constructors do not have to initialize every data field.
- Provides memory safety by making sure that an object cannot use the content of another object.

Fundamentals of memory

The following list summarizes important CLR memory concepts.

- Each process has its own, separate virtual address space. All processes on the same computer share the same physical memory, and share the page file if there is one.
- By default, on 32-bit computers, each process has a 2-GB user-mode virtual address space.
- As an application developer, you work only with virtual address space and never manipulate physical memory directly. The garbage collector allocates and frees virtual memory for you on the managed heap.

If you are writing native code, you use Win32 functions to work with the virtual address space. These functions allocate and free virtual memory for you on native heaps.

- Virtual memory can be in three states:
 - **Free.** The block of memory has no references to it and is available for allocation.
 - **Reserved.** The block of memory is available for your use and cannot be used for any other allocation request. However, you cannot store data to this memory block until it is committed.
 - **Committed.** The block of memory is assigned to physical storage.

- Virtual address space can get fragmented. This means that there are free blocks, also known as holes, in the address space. When a virtual memory allocation is requested, the virtual memory manager has to find a single free block that is large enough to satisfy that allocation request. Even if you have 2 GB of free space, the allocation that requires 2 GB will be unsuccessful unless all of that space is in a single address block.

Conditions for a garbage collection

Garbage collection occurs when one of the following conditions is true:

- The system has low physical memory.
- The memory that is used by allocated objects on the managed heap surpasses an acceptable threshold. This threshold is continuously adjusted as the process runs.
- The [GC.Collect](#) method is called. In almost all cases, you do not have to call this method, because the garbage collector runs continuously. This method is primarily used for unique situations and testing.

The managed heap

After the garbage collector is initialized by the CLR, it allocates a segment of memory to store and manage objects. This memory is called the managed heap, as opposed to a native heap in the operating system.

There is a managed heap for each managed process. All threads in the process allocate memory for objects on the same heap.

To reserve memory, the garbage collector calls the Win32 [VirtualAlloc](#) function, and reserves one segment of memory at a time for managed applications. The garbage collector also reserves segments as needed, and releases segments back to the operating system (after clearing them of any objects) by calling the Win32 [VirtualFree](#) function.

The size of segments allocated by the garbage collector is implementation-specific and is subject to change at any time, including in periodic updates. Your app should never make assumptions about or depend on a particular segment size, nor should it attempt to configure the amount of memory available for segment allocations.

The fewer objects allocated on the heap, the less work the garbage collector has to do. When you allocate objects, do not use rounded-up values that exceed your needs, such as allocating an array of 32 bytes when you need only 15 bytes.

When a garbage collection is triggered, the garbage collector reclaims the memory that is occupied by dead objects. The reclaiming process compacts live objects so that they are

moved together, and the dead space is removed, thereby making the heap smaller. This ensures that objects that are allocated together stay together on the managed heap, to preserve their locality.

The intrusiveness (frequency and duration) of garbage collections is the result of the volume of allocations and the amount of survived memory on the managed heap.

The heap can be considered as the accumulation of two heaps: the large object heap and the small object heap.

The large object heap contains very large objects that are 85,000 bytes and larger. The objects on the large object heap are usually arrays. It is rare for an instance object to be extremely large.

Generations

The heap is organized into generations so it can handle long-lived and short-lived objects. Garbage collection primarily occurs with the reclamation of short-lived objects that typically occupy only a small part of the heap. There are three generations of objects on the heap:

- **Generation 0.** This is the youngest generation and contains short-lived objects. An example of a short-lived object is a temporary variable. Garbage collection occurs most frequently in this generation.

Newly allocated objects form a new generation of objects and are implicitly generation 0 collections, unless they are large objects, in which case they go on the large object heap in a generation 2 collection.

Most objects are reclaimed for garbage collection in generation 0 and do not survive to the next generation.

- **Generation 1.** This generation contains short-lived objects and serves as a buffer between short-lived objects and long-lived objects.
- **Generation 2.** This generation contains long-lived objects. An example of a long-lived object is an object in a server application that contains static data that is live for the duration of the process.

Garbage collections occur on specific generations as conditions warrant. Collecting a generation means collecting objects in that generation and all its younger generations. A generation 2 garbage collection is also known as a full garbage collection, because it reclaims all objects in all generations (that is, all objects in the managed heap).

What happens during a garbage collection

A garbage collection has the following phases:

- A marking phase that finds and creates a list of all live objects.
- A relocating phase that updates the references to the objects that will be compacted.
- A compacting phase that reclaims the space occupied by the dead objects and compacts the surviving objects. The compacting phase moves objects that have survived a garbage collection toward the older end of the segment.

Because generation 2 collections can occupy multiple segments, objects that are promoted into generation 2 can be moved into an older segment. Both generation 1 and generation 2 survivors can be moved to a different segment, because they are promoted to generation 2.

Ordinarily, the large object heap is not compacted, because copying large objects imposes a performance penalty. However, starting with the .NET Framework 4.5.1, you can use the [GCSettings.LargeObjectHeapCompactionMode](#) property to compact the large object heap on demand.

[https://msdn.microsoft.com/en-us/library/ee787088\(v=vs.110\).aspx](https://msdn.microsoft.com/en-us/library/ee787088(v=vs.110).aspx)

Static class can be inherited from another class?

Static classes are sealed and therefore cannot be **inherited**. They cannot **inherit** from any **class** except **Object**. **Static classes** cannot contain an instance constructor; however, they **can** have a **static** constructor. For more information, see **Static Constructors (C# Programming Guide)**

This is actually by design. There seems to be no good reason to inherit a static class. It has public static members that you can always access via the class name itself. **The only reasons I have seen for inheriting static stuff have been bad ones, such as saving a couple of characters of typing.**

There may be reason to consider mechanisms to bring static members directly into scope (and we will in fact consider this after the Orcas product cycle), but static class inheritance is not the way to go: It is the wrong mechanism to use, and works only for static members that happen to reside in a static class.

(Mads Torgersen, C# Language PM)

Other opinions from [channel9](#)

Inheritance in .NET works only on instance base. Static methods are defined on the type level not on the instance level. That is why overriding doesn't work with static methods/properties/events...

Static methods are only held once in memory. There is no virtual table etc. that is created for them.

If you invoke an instance method in .NET, you always give it the current instance. This is hidden by the .NET runtime, but it happens. **Each instance method has as first argument a pointer (reference) to the object that the method is run on. This doesn't happen with static methods** (as they are defined on type level). How should the compiler decide to select the method to invoke?

Extension Methods

Extension methods enable you to "add" methods to existing types without creating a new derived type, recompiling, or otherwise modifying the original type. Extension methods are a special kind of static method, but they are called as if they were instance methods on the extended type. For client code written in C# and Visual Basic, there is no apparent difference between calling an extension method and the methods that are actually defined in a type.

<https://msdn.microsoft.com/en-IN/library/bb383977.aspx>

```
namespace ExtensionMethods
{
    public static class MyExtensions
    {
        public static int WordCount(this String str)
        {
            return str.Split(new char[] { ' ', '.', '?' },
                             StringSplitOptions.RemoveEmptyEntries).Length;
        }
    }
}

string s = "Hello Extension Methods";
int i = s.WordCount();
```

Binding Extension Methods at Compile Time

You can use extension methods to extend a class or interface, but not to override them. An extension method with the same name and signature as an interface or class method will never be called. At compile time, extension methods always have lower priority than

instance methods defined in the type itself. In other words, if a type has a method named `Process(int i)`, and you have an extension method with the same signature, the compiler will always bind to the instance method. When the compiler encounters a method invocation, it first looks for a match in the type's instance methods. If no match is found, it will search for any extension methods that are defined for the type, and bind to the first extension method that it finds. The following example demonstrates how the compiler determines which extension method or instance method to bind to.

Example

The following example demonstrates the rules that the C# compiler follows in determining whether to bind a method call to an instance method on the type, or to an extension method. The static class `Extensions` contains extension methods defined for any type that implements `IMyInterface`. Classes `A`, `B`, and `C` all implement the interface.

The `MethodB` extension method is never called because its name and signature exactly match methods already implemented by the classes.

When the compiler cannot find an instance method with a matching signature, it will bind to a matching extension method if one exists.

C#

```
// Define an interface named IMyInterface.
namespace DefineIMyInterface
{
    using System;

    public interface IMyInterface
    {
        // Any class that implements IMyInterface must define a method
        // that matches the following signature.
        void MethodB();
    }
}

// Define extension methods for IMyInterface.
namespace Extensions
{
    using System;
    using DefineIMyInterface;

    // The following extension methods can be accessed by instances of any
    // class that implements IMyInterface.
    public static class Extension
    {
        public static void MethodA(this IMyInterface myInterface, int i)
        {
            Console.WriteLine
                ("Extension.MethodA(this IMyInterface myInterface, int i)");
        }
    }
}
```

```

        public static void MethodA(this IMyInterface myInterface, string s)
        {
            Console.WriteLine
                ("Extension.MethodA(this IMyInterface myInterface, string
s)");
        }

        // This method is never called in ExtensionMethodsDemo1, because each
        // of the three classes A, B, and C implements a method named MethodB
        // that has a matching signature.
        public static void MethodB(this IMyInterface myInterface)
        {
            Console.WriteLine
                ("Extension.MethodB(this IMyInterface myInterface)");
        }
    }
}

// Define three classes that implement IMyInterface, and then use them to test
// the extension methods.
namespace ExtensionMethodsDemo1
{
    using System;
    using Extensions;
    using DefineIMyInterface;

    class A : IMyInterface
    {
        public void MethodB() { Console.WriteLine("A.MethodB()"); }
    }

    class B : IMyInterface
    {
        public void MethodB() { Console.WriteLine("B.MethodB()"); }
        public void MethodA(int i) { Console.WriteLine("B.MethodA(int i)"); }
    }

    class C : IMyInterface
    {
        public void MethodB() { Console.WriteLine("C.MethodB()"); }
        public void MethodA(object obj)
        {
            Console.WriteLine("C.MethodA(object obj)");
        }
    }

    class ExtMethodDemo
    {
        static void Main(string[] args)
        {
            // Declare an instance of class A, class B, and class C.
            A a = new A();
            B b = new B();
            C c = new C();

```

```

// For a, b, and c, call the following methods:
//      -- MethodA with an int argument
//      -- MethodA with a string argument
//      -- MethodB with no argument.

// A contains no MethodA, so each call to MethodA resolves to
// the extension method that has a matching signature.
a.MethodA(1);           // Extension.MethodA(object, int)
a.MethodA("hello");     // Extension.MethodA(object, string)

// A has a method that matches the signature of the following call
// to MethodB.
a.MethodB();            // A.MethodB()

// B has methods that match the signatures of the following
// method calls.
b.MethodA(1);           // B.MethodA(int)
b.MethodB();            // B.MethodB()

// B has no matching method for the following call, but
// class Extension does.
b.MethodA("hello");     // Extension.MethodA(object, string)

// C contains an instance method that matches each of the
following
// method calls.
c.MethodA(1);           // C.MethodA(object)
c.MethodA("hello");     // C.MethodA(object)
c.MethodB();            // C.MethodB()
    }
}
}
/* Output:
    Extension.MethodA(this IMyInterface myInterface, int i)
    Extension.MethodA(this IMyInterface myInterface, string s)
    A.MethodB()
    B.MethodA(int i)
    B.MethodB()
    Extension.MethodA(this IMyInterface myInterface, string s)
    C.MethodA(object obj)
    C.MethodA(object obj)
    C.MethodB()
*/

```

Nested Types

Use a nested class when the class you are nesting is only useful to the enclosing class. For instance, nested classes allow you to write something like (simplified):

```

public class SortedMap {
    private class TreeNode {
        TreeNode left;
        TreeNode right;
    }
}

```


You can make a complete definition of your class in one place, you don't have to jump through any PIMPL hoops to define how your class works, and the outside world doesn't need to see anything of your implementation.

If the `TreeNode` class was external, you would either have to make all the fields `public` or make a bunch of get/set methods to use it. The outside world would have another class polluting their intellisense.

<http://stackoverflow.com/questions/48872/why-when-should-you-use-nested-classes-in-net-or-shouldnt-you>

Use of Using keyword

“**Using**” keyword takes the parameter of type `IDisposable`. Whenever you are **using** any `IDisposable` type object you should **use** the “**using**” keyword to handle automatically when it should close or dispose. Internally **using keyword** calls the `Dispose()` method to dispose the `IDisposable` object.

Dispose and Finalise

Finalize. 1) Used to free unmanaged resources like files, database connections, COM etc. held by an object before that object is destroyed. //At runtime **C#** destructor is automatically Converted to **Finalize method**. 2) Internally, it is called by Garbage Collector and cannot be called by user code

What happens to FReachable Objects if they are found to be reachable?

If suppose an object implements the `Finalize` method but inside it refers an alive static object of the application (bad design! but very possible).

Now when GC kicks in and finalises the object by putting it in Finalization queue and then move it to FReachable queue where it would call its `finalize` method.

But whoa! it finds its referring an alive object so it doesnt allow GC to reclaim the memory occupied by the object and marks the object alive again. A zombie object!

At this point where does this object reside?

1. Remains in freachable?
2. Remains in Finalization queue?
3. Remains on managed heap in an indeterminate state (removed from freachable and finalization queues)?

Also what can be the best place to `ReRegisterForFinalize()` for such object?

it finds its referring an alive object

That does not matter. An outgoing reference is irrelevant for GC.

Another scenario is where the finalizing objects makes itself reachable again, by registering itself into some rooted list.

This is called resurrection. It does not require much special attention from the GC: the finalizers are processed and the reference is removed from fReachable. Note that there is nothing special, objects in fReachable are *not* in an indeterminate state at any time. They have to be rescanned in the next GC collection. One of the costs of a finalizer is needing 2 rounds of the GC.

Usually the object would call `ReRegisterForFinalize(this)` when it resurrects.

But please note that resurrection is far from a common practice.

Dispose

Garbage collector (GC) plays the main and important role in .NET for memory management so programmer can focus on the application functionality. Garbage collector is responsible for releasing the memory (objects) that is not being used by the application. But GC has limitation that, it can reclaim or release only memory which is used by managed resources. There are a couple of resources which GC is not able to release as it doesn't have information that, how to claim memory from those resources like File handlers, window handlers, network sockets, database connections etc. If your application these resources than it's programs responsibility to release unmanaged resources. For example, if we open a file in our program and not closed it after processing than that file will not be available for other operation or it is being used by other application than they can not open or modify that file. For this purpose `FileStream` class provides `Dispose` method. We must call this method after file processing finished. Otherwise it will through exception `Access Denied` or file is being used by other program.

Close Vs Dispose

Some objects expose `Close` and `Dispose` two methods. For `Stream` classes both serve the same purpose. `Dispose` method calls `Close` method inside.

```
1. void Dispose()  
2. {  
3.     this.Close();  
4. }
```

Here question comes, why do we need `Dispose` method in `Stream`. Having `Dispose` method will enable you to write below code and implicitly call `dispose` method and ultimately will call `Close` method.

```
1. using (FileStream file = new FileStream("path", FileMode.Open, FileAccess.Read))
```

```
2. {  
3.     //Do something with file  
4. }
```

But for some classes both methods behave slightly different. For example Connection class. If Close method is called than it will disconnect with database and release all resources being used by the connection object and Open method will reconnect it again with database without reinitializing the connection object. However Dispose method completely release the connection object and cannot be reopen just calling Open method. We will have re-initialize the Connection object.

Creating Dispose

To implement Dispose method for your custom class, you need to implement IDisposable interface. IDisposable interface expose Dispose method where code to release unmanaged resource will be written.

Finalize

Finalize method also called destructor to the class. Finalize method can not be called explicitly in the code. Only Garbage collector can call the the Finalize when object become inaccessible. Finalize method cannot be implemented directly it can only be implement via declaring destructor. Following class illustrate, how to declare destructor. It is recommend that implement Finalize and Dispose method together if you need to implement Finalize method. After compilation destructor becomes Finalize method.

```
1. public class MyClass: IDisposable {  
2.  
3.     //Construcotr  
4.     public MyClass() {  
5.         //Initialization:  
6.     }  
7.  
8.     //Destrucor also called Finalize  
9.     ~MyClass() {  
10.         this.Dispose();  
11.     }  
12.  
13.     public void Dispose() {
```

```
14.    //write code to release unmanaged resource.  
15.    }  
16. }
```

Using Finalize

Now question is, When to implement Finalize? There may be any unmanaged resource for example file stream declared at class level. We may not be knowing what stage or which step should be appropriate to close the file. This object is being use at many places in the application. So in this scenario Finalize can be appropriate location where unmanaged resource can be released. It means, clean the memory acquired by the unmanaged resource as soon as object is inaccessible to application.

Finalize is bit expensive to use. It doesn't clean the memory immediately. When application runs, Garbage collector maintains a separate queue/array when it adds all object which has finalized implemented. Other term GC knows which object has Finalize implemented. When the object is ready to claim memory, Garbage Collector call finalize method for that object and remove from the collection. In this process it just clean the memory that used by unmanaged resource. Memory used by managed resource still in heap as inaccessible reference. That memory release, whenever Garbage Collector run next time. Due to finalize method GC will not clear entire memory associated with object in first attempt.

Conclusion

It is always recommended that, one should not implement the Finalize method until it is extremely necessary. First priority should always be to implement the Dispose method and clean unmanaged as soon as possible when processing finish with that.

<http://www.c-sharpcorner.com/UploadFile/nityaprakash/back-to-basics-dispose-vs-finalize/>

.Net Garbage Collection and Finalization Queue

Finalize is a special method that is automatically called by the garbage collector (GC) before the object is collected. This method is only called by the GC. Destructor in C# are automatically translated into Finalize. You can see the IL code using IDASM where you will see that destructor is renamed to finalize.

```
1.    public class A  
2.    {  
3.        ~A()  
4.    {  
5.        Console.WriteLine("Class A Destructor");  
6.        //Clean up unmanaged resources here  
7.    }  
8. }
```

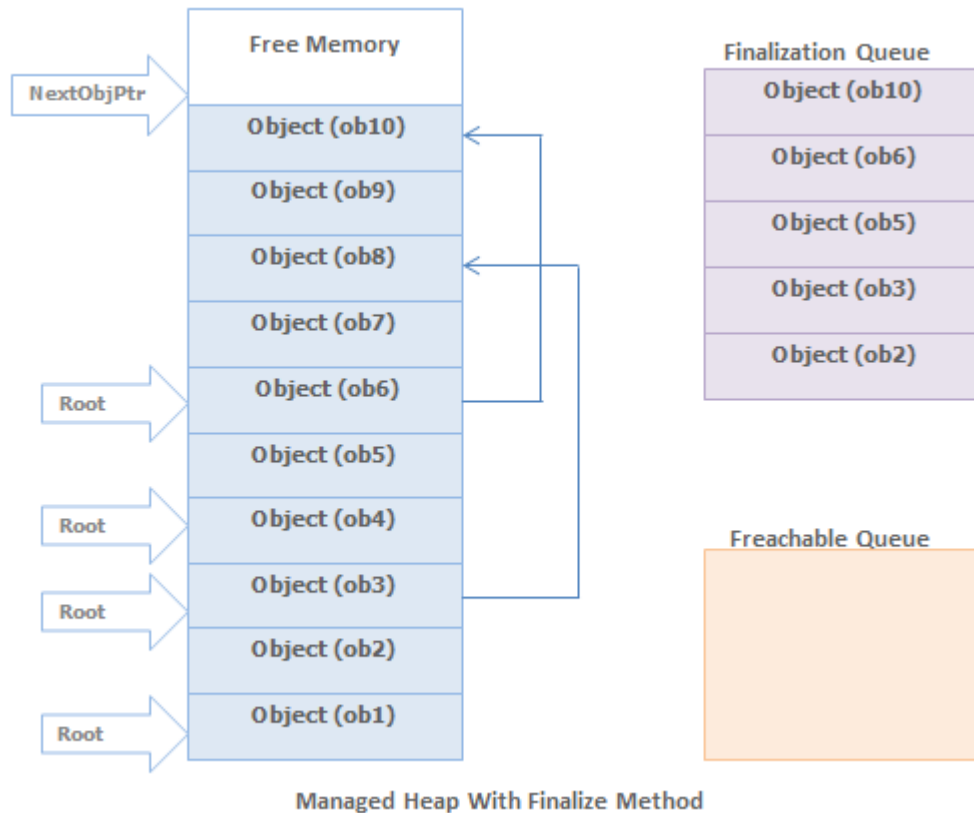
The CLR translate the above C# destructor to like this:

```
1.      public override void Finalize()  
2.      {  
3.      try  
4.      {  
5.      Console.WriteLine("Class A Destructor");  
6.      //Clean up unmanaged resources here  
7.      }  
8.      finally  
9.      {  
10.     base.Finalize();  
11.     }  
12.     }
```

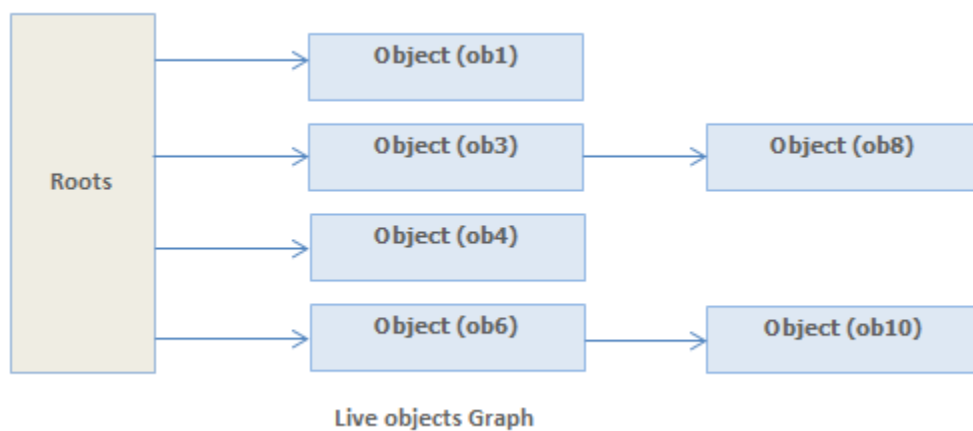
Fundamental of Finalization

When a new object is created, the memory is allocated in the managed heap. If newly created object have a Finalize() method or a destructor then a pointer pointing to that object is put into the finalization queue. Basically, finalization queue is an internal data structure that is controlled and managed by the GC. Hence each pointer in finalization queue points to an object that have its Finalize method call before the memory is reclaimed.

In the below fig. the managed heap contains 10 objects and objects 2,3,5,6,and 10 also contains the Finalize method. Hence pointers to these objects were added to the finalization queue.



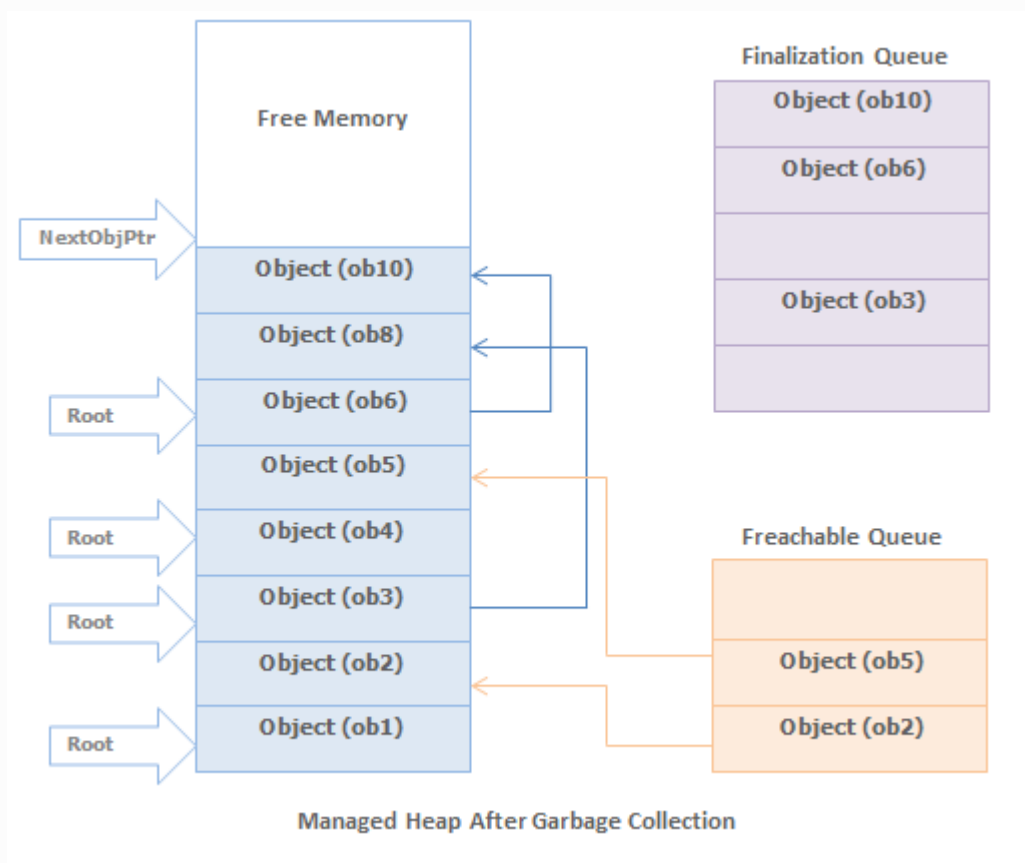
When the garbage collector starts go through the roots, it make a graph of all the objects reachable from the roots. The below fig. shows a heap with allocated objects. In this heap the application roots directly refer to the objects 1,3,4,6 and object 3 & 6 refers to the objects 8 & 10. Hence all these objects will become the part of the live objects graph.



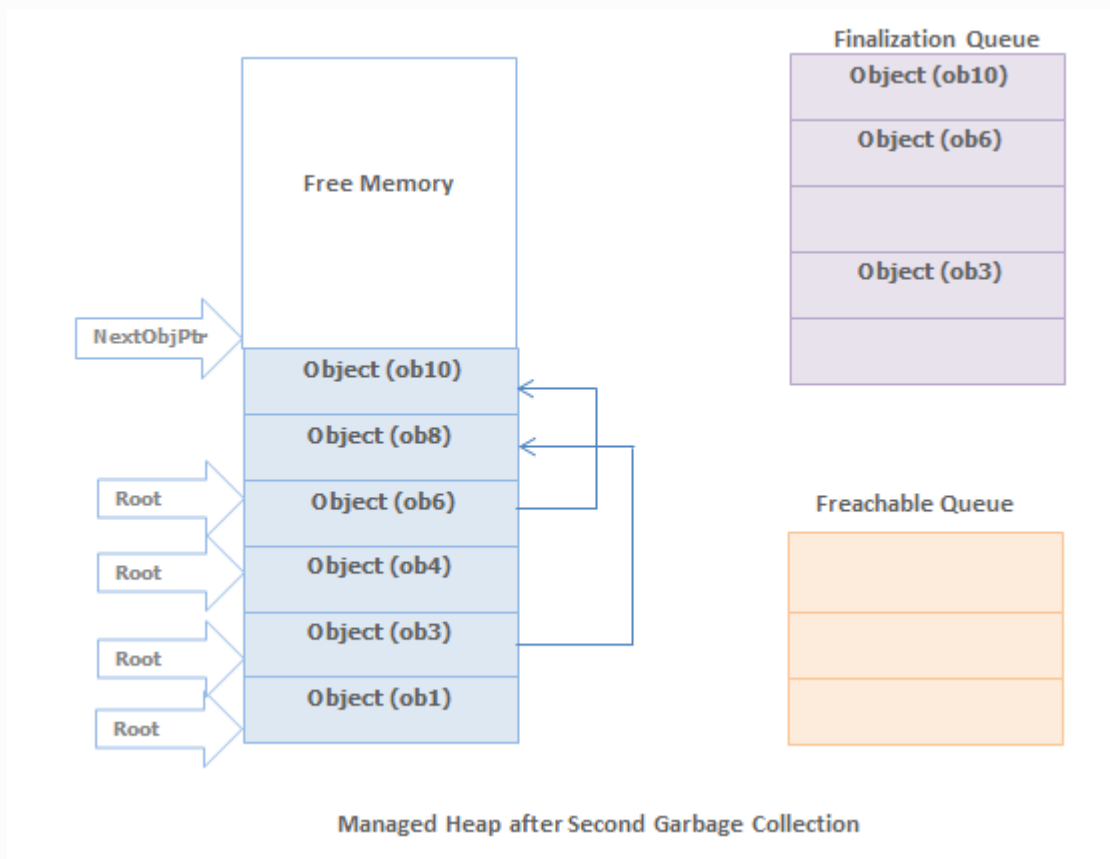
The objects which are not reachable from application's roots, are considered as garbage since these are not accessible by the application. In above heap objects 2,5,7,9 will be considered as dead objects.

Before the collections for dead objects, the garbage collector looks into the finalization queue for pointers identifies these objects. If the pointer found, then the pointer is flushed from the finalization queue and append to the freachable queue. The freachable queue is also an internal data structure and controlled by the garbage collector. Now each and every pointer with in the freachable queue will identify an object that is ready to have its Finalize method called.

After the collection, the managed heap looks like below Fig. Here, you see that the memory occupied by objects 7 and 9 has been reclaimed because these objects did not have a Finalize method. However, the memory occupied by objects 2 and 5 could not be reclaimed because their Finalize method has not been called yet.



When an object's pointer entry move from the finalization queue to the freachable queue, the object is not considered garbage and its memory is not reclaimed. There is a special run-time thread that is dedicated for calling Finalize methods. When there is no entry in the freachable queue then this thread sleeps. But when there is entry in the freachable queue, this thread wakes and removes each entry from the queue by calling each object's Finalize method.



The next time the garbage collector is invoked, it sees that the finalized objects are truly garbage, since the application's roots don't point to it and the freachable queue no longer points to it. Now the memory for the object is simply reclaimed.

Note

1. The two GCs are required to reclaim memory used by objects that have Finalize method.

Reference : <http://msdn.microsoft.com/en-us/magazine/bb985010.aspx>

<http://www.dotnettricks.com/learn/netframework/net-garbage-collection-and-finalization-queue>

What's the purpose of GC.SuppressFinalize(this) in Dispose() method?

When implementing the dispose pattern you might also add a finalizer to your class that calls `Dispose()`. This is to make sure that `Dispose()` *always* gets called, even if a client forgets to call it.

To prevent the dispose method from running twice (in case the object already has been disposed) you add `GC.SuppressFinalize(this);`

Freachable queue and Finalization queue

Freachable what? You might ask. Freachable (pronounced F-reachable) is one of CLR Garbage Collector internal structures that is used in a finalization part of garbage collection. You might have heard about the Finalization queue where every object that needs finalization lands initially. This is determined based on whether he has a `Finalize` method, or it's object type contains a `Finalize` method definition to speak more precisely. This seems like a good idea, GC wants to keep track of all objects that he needs to call `Finalize` on, so that when he collects he can find them easily. Why would he need another collection then?

Well apparently what GC does when he finds a garbage object that is on Finalizable queue, is a bit more complicated than you might expect. GC doesn't call the `Finalize` method directly, instead removes object reference from Finalizable queue and puts it on a (wait for it..) Freachable queue. Weird, huh? Well it turns out there is a specialized CLR thread that is only responsible for monitoring the Freachable queue and when GC adds new items there, he kicks in, takes objects one by one and calls it's `Finalize` method. One important point about it is that you shouldn't rely on `Finalize` method being called by the same thread as rest of you app, don't count on Thread Local Storage etc.

But what interest me more is why? Well the article doesn't give an answer to that, but there are two things that come to my mind. First is performance, you obviously want the garbage collection to be as fast as possible and a great deal of work was put into making it so. It seems only natural to keep side tasks like finalization handled by a background thread, so that main one can be as fast a possible. Second, but not less important is that `Finalize` is after all a client code from the GC perspective, CLR can't really trust your dear reader implementation. Maybe your `Finalize` will throw exception or will go into infinite loop? It's not something you want to be a part of GC process, it's much less dangerous if it can only affect a background thread.

Implementing Finalize and Dispose to Clean Up Unmanaged Resources

[https://msdn.microsoft.com/en-us/library/b1yfk5e\(v=vs.100\).aspx](https://msdn.microsoft.com/en-us/library/b1yfk5e(v=vs.100).aspx)

Class instances often encapsulate control over resources that are not managed by the runtime, such as window handles (HWND), database connections, and so on. Therefore, you should provide both an explicit and an implicit way to free those resources. Provide implicit control by implementing the protected [Finalize](#) on an object (destructor syntax in C# and C++). The garbage collector calls this method at some point after there are no longer any valid references to the object.

In some cases, you might want to provide programmers using an object with the ability to explicitly release these external resources before the garbage collector frees the object. If an external resource is scarce or expensive, better performance can be achieved if the programmer explicitly releases resources when they are no longer being used. To provide explicit control, implement the [Dispose](#) provided by the [IDisposable](#). The consumer of the object should call this method when it is finished using the object. **Dispose** can be called even if other references to the object are alive.

Note that even when you provide explicit control using **Dispose**, you should provide implicit cleanup using the **Finalize** method. **Finalize** provides a backup to prevent resources from permanently leaking if the programmer fails to call **Dispose**.

For more information about implementing **Finalize** and **Dispose** to clean up unmanaged resources, see [Garbage Collection](#). The following example illustrates the basic design pattern for implementing **Dispose**. This example requires the [System](#) namespace.

```
// Design pattern for a base class.
public class Base: IDisposable
{
    private bool disposed = false;

    //Implement IDisposable.
    public void Dispose()
    {
        Dispose(true);
        GC.SuppressFinalize(this);
    }

    protected virtual void Dispose(bool disposing)
    {
        if (!disposed)
        {
            if (disposing)
            {
                // Free other state (managed objects).
            }
            // Free your own state (unmanaged objects).
        }
    }
}
```

```

        // Set large fields to null.
        disposed = true;
    }
}

// Use C# destructor syntax for finalization code.
~Base()
{
    // Simply call Dispose(false).
    Dispose (false);
}

// Design pattern for a derived class.
public class Derived: Base
{
    private bool disposed = false;

    protected override void Dispose(bool disposing)
    {
        if (!disposed)
        {
            if (disposing)
            {
                // Release managed resources.
            }
            // Release unmanaged resources.
            // Set large fields to null.
            // Call Dispose on your base class.
            disposed = true;
        }
        base.Dispose(disposing);
    }
    // The derived class does not have a Finalize method
    // or a Dispose method without parameters because it inherits
    // them from the base class.
}

```

The following code expands the previous example to show the different ways **Dispose** is invoked and when **Finalize** is called. The stages of the disposing pattern are tracked with output to the console. The allocation and release of an unmanaged resource is handled in the derived class.

```

using System;
using System.Collections.Generic;
using System.Runtime.InteropServices;

// Design pattern for a base class.
public abstract class Base : IDisposable
{
    private bool disposed = false;
    private string instanceName;
    private List<object> trackingList;

```

```

public Base(string instanceName, List<object> tracking)
{
    this.instanceName = instanceName;
    trackingList = tracking;
    trackingList.Add(this);
}

public string InstanceName
{
    get
    {
        return instanceName;
    }
}

//Implement IDisposable.
public void Dispose()
{
    Console.WriteLine("\n[{0}].Base.Dispose()", instanceName);
    Dispose(true);
    GC.SuppressFinalize(this);
}

protected virtual void Dispose(bool disposing)
{
    if (!disposed)
    {
        if (disposing)
        {
            // Free other state (managed objects).
            Console.WriteLine("[{0}].Base.Dispose(true)", instanceName);
            trackingList.Remove(this);
            Console.WriteLine("[{0}] Removed from tracking list: {1:x16}",
                instanceName, this.GetHashCode());
        }
        else
        {
            Console.WriteLine("[{0}].Base.Dispose(false)", instanceName);
        }
        disposed = true;
    }
}

// Use C# destructor syntax for finalization code.
~Base()
{
    // Simply call Dispose(false).
    Console.WriteLine("\n[{0}].Base.Finalize()", instanceName);
    Dispose(false);
}

}

// Design pattern for a derived class.
public class Derived : Base
{
    private bool disposed = false;

```

```

private IntPtr umResource;

public Derived(string instanceName, List<object> tracking) :
    base(instanceName, tracking)
{
    // Save the instance name as an unmanaged resource
    umResource = Marshal.StringToCoTaskMemAuto(instanceName);
}

protected override void Dispose(bool disposing)
{
    if (!disposed)
    {
        if (disposing)
        {
            Console.WriteLine("[{0}].Derived.Dispose(true)",
InstanceName);
            // Release managed resources.
        }
        else
        {
            Console.WriteLine("[{0}].Derived.Dispose(false)",
InstanceName);
            // Release unmanaged resources.
            if (umResource != IntPtr.Zero)
            {
                Marshal.FreeCoTaskMem(umResource);
                Console.WriteLine("[{0}] Unmanaged memory freed at {1:x16}",
InstanceName, umResource.ToInt64());
                umResource = IntPtr.Zero;
            }
            disposed = true;
        }
        // Call Dispose in the base class.
        base.Dispose(disposing);
    }
    // The derived class does not have a Finalize method
    // or a Dispose method without parameters because it inherits
    // them from the base class.
}

public class TestDisposal
{
    public static void Main()
    {
        List<object> tracking = new List<object>();

        // Dispose is not called, Finalize will be called later.
        using (null)
        {
            Console.WriteLine("\nDisposal Scenario: #1\n");
            Derived d3 = new Derived("d1", tracking);
        }

        // Dispose is implicitly called in the scope of the using statement.
        using (Derived d1 = new Derived("d2", tracking))
    }
}

```

```

    {
        Console.WriteLine("\nDisposal Scenario: #2\n");
    }

    // Dispose is explicitly called.
    using (null)
    {
        Console.WriteLine("\nDisposal Scenario: #3\n");
        Derived d2 = new Derived("d3", tracking);
        d2.Dispose();
    }

    // Again, Dispose is not called, Finalize will be called later.
    using (null)
    {
        Console.WriteLine("\nDisposal Scenario: #4\n");
        Derived d4 = new Derived("d4", tracking);
    }

    // List the objects remaining to dispose.
    Console.WriteLine("\nObjects remaining to dispose = {0:d}",
tracking.Count);
    foreach (Derived dd in tracking)
    {
        Console.WriteLine("    Reference Object: {0:s}, {1:x16}",
            dd.InstanceName, dd.GetHashCode());
    }

    // Queued finalizers will be exeucted when Main() goes out of scope.
    Console.WriteLine("\nDequeueing finalizers...");
}
}

// The program will display output similar to the following:
//
// Disposal Scenario: #1
//
// Disposal Scenario: #2
//
// [d2].Base.Dispose()
// [d2].Derived.Dispose(true)
// [d2] Unmanaged memory freed at 000000000034e420
// [d2].Base.Dispose(true)
// [d2] Removed from tracking list: 0000000002bf8098
//
// Disposal Scenario: #3
//
// [d3].Base.Dispose()
// [d3].Derived.Dispose(true)
// [d3] Unmanaged memory freed at 000000000034e420
// [d3].Base.Dispose(true)
// [d3] Removed from tracking list: 0000000000bb8560
//
// Disposal Scenario: #4

```

```
//
//
// Objects remaining to dispose = 2
//   Reference Object: d1, 000000000297b065
//   Reference Object: d4, 0000000003553390
//
// Dequeueing finalizers...
//
// [d4].Base.Finalize()
// [d4].Derived.Dispose(false)
// [d4] Unmanaged memory freed at 000000000034e420
// [d4].Base.Dispose(false)
//
// [d1].Base.Finalize()
// [d1].Derived.Dispose(false)
// [d1] Unmanaged memory freed at 000000000034e3f0
// [d1].Base.Dispose(false)
```

For an additional code example illustrating the design pattern for implementing **Finalize** and **Dispose**, see [Implementing a Dispose Method](#).

Customizing a Dispose Method Name

Occasionally a domain-specific name is more appropriate than **Dispose**. For example, a file encapsulation might want to use the method name **Close**. In this case, implement **Dispose** privately and create a public **Close** method that calls **Dispose**. The following code example illustrates this pattern. You can replace **Close** with a method name appropriate to your domain. This example requires the [System](#) namespace.

C#

[VB](#)

```
// Do not make this method virtual.
// A derived class should not be allowed
// to override this method.
public void Close()
{
    // Call the Dispose method with no parameters.
    Dispose();
}
```

Finalize

The following rules outline the usage guidelines for the **Finalize** method:

- Implement **Finalize** only on objects that require finalization. There are performance costs associated with **Finalize** methods.

- If you require a **Finalize** method, consider implementing **IDisposable** to allow users of your class to avoid the cost of invoking the **Finalize** method.
- Do not make the **Finalize** method more visible. It should be **protected**, not **public**.
- An object's **Finalize** method should free any external resources that the object owns. Moreover, a **Finalize** method should release only resources that the object has held onto. The **Finalize** method should not reference any other objects.
- Do not directly call a **Finalize** method on an object other than the object's base class. This is not a valid operation in the C# programming language.
- Call the **base class's Finalize** method from an object's **Finalize** method.

Note

The base class's **Finalize** method is called automatically with the C# and C++ destructor syntax.

Dispose

The following rules outline the usage guidelines for the **Dispose** method:

- Implement the dispose design pattern on a type that encapsulates resources that explicitly need to be freed. Users can free external resources by calling the public **Dispose** method.
- Implement the dispose design pattern on a base type that commonly has derived types that hold onto resources, even if the base type does not. If the base type has a **Close** method, often this indicates the need to implement **Dispose**. In such cases, do not implement a **Finalize** method on the base type. **Finalize** should be implemented in any derived types that introduce resources that require cleanup.
- Free any disposable resources a type owns in its **Dispose** method.
- After **Dispose** has been called on an instance, prevent the **Finalize** method from running by calling the [GC.SuppressFinalize](#). The exception to this rule is the rare situation in which work must be done in **Finalize** that is not covered by **Dispose**.
- Call the base class's **Dispose** method if it implements **IDisposable**.

- Do not assume that **Dispose** will be called. Unmanaged resources owned by a type should also be released in a **Finalize** method in the event that **Dispose** is not called.
- Throw an **ObjectDisposedException** from instance methods on this type (other than **Dispose**) when resources are already disposed. This rule does not apply to the **Dispose** method because it should be callable multiple times without throwing an exception.
- Propagate the calls to **Dispose** through the hierarchy of base types.
The **Dispose** method should free all resources held by this object and any object owned by this object. For example, you can create an object such as a **TextReader** that holds onto a **Stream** and an **Encoding**, both of which are created by the **TextReader** without the user's knowledge. Furthermore, both the **Stream** and the **Encoding** can acquire external resources. When you call the **Dispose** method on the **TextReader**, it should in turn call **Dispose** on the **Stream** and the **Encoding**, causing them to release their external resources.
- Consider not allowing an object to be usable after its **Dispose** method has been called. Re-creating an object that has already been disposed is a difficult pattern to implement.
- Allow a **Dispose** method to be called more than once without throwing an exception. The method should do nothing after the first call.

The theory behind covariance and contravariance in C# 4

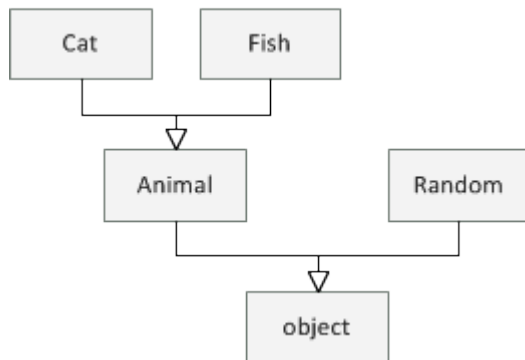


In C# 4.0, we can annotate generic type parameters with **out** and **in** annotations to specify whether they should behave *covariantly* or *contravariantly*. This is mainly useful when using already defined standard interfaces. Covariance means that you can use **IEnumerable<string>** in place where **IEnumerable<object>** is expected. Contravariance allows you to pass **Comparable<object>** as an argument of a method taking **Comparable<string>**.

So far, so good. If you already learned about covariance and contravariance in C# 4, then the above two examples are probably familiar. If you're new to the concepts, then the examples should make sense (after a bit of thinking, but I'll say more about them). However, there is still a number of questions. Is there some easy way to explain the two concepts? Why one option makes sense for some types and the other for different types? And why the hell is it called *covariance* and *contravariance* anyway?

In this blog post, I'll explain some of the mathematics that you can use to think about covariance and contravariance.

Covariance and contravariance in C#



Let's say that we have a hierarchy of classes as shown in the diagram on the right. All types are subclasses of the .NET **object** type. Then we have **Animal** which has two subclasses **Cat** and **Fish** and finally, there is also some other .NET type **Random**.

Covariance

Let's start by looking what *covariance* means. Assuming that every **Animal** has a **Name**, we can write the following function that prints names of all animals in some collection:

```
void PrintAnimals(IEnumerable<Animal> animals) {  
  
    for(var animal in animals)
```

```
Console.WriteLine(animal.Name);  
  
}
```

It is important to note that `IEnumerable<Animal>` can be only used to read `Animal` values from the collection (using the `Current` property of `IEnumerator<Animal>`). There is no way we could write `Animal` back to the collection - it is read-only. Now, if we create a collection of cats, can we use `PrintAnimals` to print their names?

```
IEnumerable<Cat> cats = new List<Cat> { new Cat("Troublemaker") };  
  
PrintAnimals(cats);
```

If you compile and run this sample, you'll see that the C# compiler accepts it and the program runs fine. When calling `PrintAnimals`, the compiler uses *covariance* to convert `IEnumerable<Cat>` to `IEnumerable<Animal>`. This is correct, because the `IEnumerable` interface is marked as *covariant* using the `out` annotation. When you run the program, the `PrintAnimals` method cannot cause anything wrong, because it can only *read* animals from the collection. Using a collection of cats as an argument is fine, because all cats are animals.

Contravariance

Contravariance works the other way than *covariance*. Let's say we have a method that creates some cats and compares them using a provided `IComparer<Cat>` object. In a more realistic example, the method might, for example, sort the cats:

```
void CompareCats(IComparer<Cat> comparer) {  
  
    var cat1 = new Cat("Otto");  
  
    var cat2 = new Cat("Troublemaker");  
  
    if (comparer.Compare(cat2, cat1) > 0)
```

```
Console.WriteLine("Troublemaker wins!");  
  
}
```

The `comparer` object takes cats as arguments, but it never returns a cat as the result. You could say that it is a write-only in the way in which it uses the generic type parameter. Now, thanks to *contravariance*, we can create a comparer that can compare animals and use it as an argument to `CompareCats`:

```
IComparer<Animal> compareAnimals = new AnimalSizeComparator();  
  
CompareCats(compareAnimals);
```

The compiler accepts this code because the `IComparer` interface is *contravariant* and its generic type parameter is marked with the `in` annotation. When you run the program, it also makes sense. The `compareAnimals` object that we created knows how to compare animals and so it can certainly also compare two cats. A problem would be if we could read a `Cat` from `IComparer<Cat>` (because we couldn't get `Cat` from `IComparer<Animal>`!), but that is not possible, because `IComparer` is write-only.

<http://tomasp.net/blog/variance-explained.aspx/>

Covariance and Contravariance in Generics

Covariance and contravariance are terms that refer to the ability to use a less derived (less specific) or more derived type (more specific) than originally specified. Generic type parameters support covariance and contravariance to provide greater flexibility in assigning and using generic types. When you are referring to a type system, covariance, contravariance, and invariance have the following definitions. The examples assume a base class named `Base` and a derived class named `Derived`.

- Covariance

Enables you to use a more derived type than originally specified.

You can assign an instance of `IEnumerable<Derived>` (`IEnumerable(Of Derived)` in Visual Basic) to a variable of type `IEnumerable<Base>`.

- Contravariance

Enables you to use a more generic (less derived) type than originally specified.

You can assign an instance of `IEnumerable<Base>` (`IEnumerable(Of Base)` in Visual Basic) to a variable of type `IEnumerable<Derived>`.

- Invariance

Means that you can use only the type originally specified; so an invariant generic type parameter is neither covariant nor contravariant.

You cannot assign an instance of `IEnumerable<Base>` (`IEnumerable(Of Base)` in Visual Basic) to a variable of type `IEnumerable<Derived>` or vice versa.

Covariant type parameters enable you to make assignments that look much like ordinary [Polymorphism](#), as shown in the following code.

C#

VB

```
IEnumerable<Derived> d = new List<Derived>();
IEnumerable<Base> b = d;
```

The [List<T>](#) class implements the [IEnumerable<T>](#) interface, so `List<Derived>` (`List(Of Derived)` in Visual Basic) implements `IEnumerable<Derived>`. The covariant type parameter does the rest.

Contravariance, on the other hand, seems counterintuitive. The following example creates a delegate of type `Action<Base>` (`Action(Of Base)` in Visual Basic), and then assigns that delegate to a variable of type `Action<Derived>`.

C#

VB

```
Action<Base> b = (target) =>
{ Console.WriteLine(target.GetType().Name); };
Action<Derived> d = b;
d(new Derived());
```

This seems backward, but it is type-safe code that compiles and runs. The lambda expression matches the delegate it is assigned to, so it defines a method that takes one parameter of type `Base` and that has no return value. The resulting delegate can be assigned to a variable of type `Action<Derived>` because the type parameter `T` of the [Action<T>](#) delegate is contravariant. The code is type-safe because `T` specifies a parameter type. When the delegate of type `Action<Base>` is invoked as if it were a delegate of type `Action<Derived>`, its argument

must be of type `Derived`. This argument can always be passed safely to the underlying method, because the method's parameter is of type `Base`.

In general, a covariant type parameter can be used as the return type of a delegate, and contravariant type parameters can be used as parameter types. For an interface, covariant type parameters can be used as the return types of the interface's methods, and contravariant type parameters can be used as the parameter types of the interface's methods.

Covariance and contravariance are collectively referred to as *variance*. A generic type parameter that is not marked covariant or contravariant is referred to as *invariant*. A brief summary of facts about variance in the common language runtime:

- In the .NET Framework 4, variant type parameters are restricted to generic interface and generic delegate types.
- A generic interface or generic delegate type can have both covariant and contravariant type parameters.
- Variance applies only to reference types; if you specify a value type for a variant type parameter, that type parameter is invariant for the resulting constructed type.
- Variance does not apply to delegate combination. That is, given two delegates of types `Action<Derived>` and `Action<Base>` (`Action(Of Derived)` and `Action(Of Base)` in Visual Basic), you cannot combine the second delegate with the first although the result would be type safe. Variance allows the second delegate to be assigned to a variable of type `Action<Derived>`, but delegates can combine only if their types match exactly.

The following subsections describe covariant and contravariant type parameters in detail:

- [Generic Interfaces with Covariant Type Parameters](#)
- [Generic Interfaces with Contravariant Generic Type Parameters](#)
- [Generic Delegates with Variant Type Parameters](#)
- [Defining Variant Generic Interfaces and Delegates](#)
- [List of Variant Generic Interface and Delegate Types](#)

Generic Interfaces with Covariant Type Parameters

Starting with the .NET Framework 4, several generic interfaces have covariant type parameters; for example: [IEnumerable<T>](#), [IEnumerator<T>](#), [IQueryable<T>](#), and [IGrouping<TKey,](#)

[TElement>](#). All the type parameters of these interfaces are covariant, so the type parameters are used only for the return types of the members.

The following example illustrates covariant type parameters. The example defines two types: `Base` has a static method named `PrintBases` that takes an `IEnumerable<Base>` (`IEnumerable(Of Base)` in Visual Basic) and prints the elements. `Derived` inherits from `Base`. The example creates an empty `List<Derived>` (`List(Of Derived)` in Visual Basic) and demonstrates that this type can be passed to `PrintBases` and assigned to a variable of type `IEnumerable<Base>` without casting. [List<T>](#) implements [IEnumerable<T>](#), which has a single covariant type parameter. The covariant type parameter is the reason why an instance of `IEnumerable<Derived>` can be used instead of `IEnumerable<Base>`.

C#

[VB](#)

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

class Base
{
    public static void PrintBases(IEnumerable<Base> bases)
    {
        foreach(Base b in bases)
        {
            Console.WriteLine(b);
        }
    }
}

class Derived : Base
{
    public static void Main()
    {
        List<Derived> dlist = new List<Derived>();

        Derived.PrintBases(dlist);
        IEnumerable<Base> bIEnum = dlist;
    }
}
```

[Back to top](#)

Generic Interfaces with Contravariant Generic Type Parameters

Starting with the .NET Framework 4, several generic interfaces have contravariant type parameters; for example: [IComparer<T>](#), [IComparable<T>](#), and [IEqualityComparer<T>](#). These

interfaces have only contravariant type parameters, so the type parameters are used only as parameter types in the members of the interfaces.

The following example illustrates contravariant type parameters. The example defines an abstract (MustInherit in Visual Basic) `Shape` class with an `Area` property. The example also defines a `ShapeAreaComparer` class that implements `IComparer<Shape>` (`IComparer(Of Shape)` in Visual Basic). The implementation of the [IComparer<T>.Compare](#) method is based on the value of the `Area` property, so `ShapeAreaComparer` can be used to sort `Shape` objects by area.

The `Circle` class inherits `Shape` and overrides `Area`. The example creates a [SortedSet<T>](#) of `Circle` objects, using a constructor that takes an `IComparer<Circle>` (`IComparer(Of Circle)` in Visual Basic). However, instead of passing an `IComparer<Circle>`, the example passes a `ShapeAreaComparer` object, which implements `IComparer<Shape>`. The example can pass a comparer of a less derived type (`Shape`) when the code calls for a comparer of a more derived type (`Circle`), because the type parameter of the [IComparer<T>](#) generic interface is contravariant.

When a new `Circle` object is added to the `SortedSet<Circle>`, the `IComparer<Shape>.Compare` method (`IComparer(Of Shape).Compare` method in Visual Basic) of the `ShapeAreaComparer` object is called each time the new element is compared to an existing element. The parameter type of the method (`Shape`) is less derived than the type that is being passed (`Circle`), so the call is type safe. Contravariance enables `ShapeAreaComparer` to sort a collection of any single type, as well as a mixed collection of types, that derive from `Shape`.

C#

VB

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

abstract class Shape
{
    public virtual double Area { get { return 0; }}
}

class Circle : Shape
{
    private double r;
    public Circle(double radius) { r = radius; }
    public double Radius { get { return r; }}
    public override double Area { get { return Math.PI * r * r; }}
}

class ShapeAreaComparer : System.Collections.Generic.IComparer<Shape>
{
    int IComparer<Shape>.Compare(Shape a, Shape b)
    {
        if (a == null) return b == null ? 0 : -1;
        return b == null ? 1 : a.Area.CompareTo(b.Area);
    }
}
```



```

    }
}

class Program
{
    static void Main()
    {
        // You can pass ShapeAreaComparer, which implements IComparer<Shape>,
        // even though the constructor for SortedSet<Circle> expects
        // IComparer<Circle>, because type parameter T of IComparer<T> is
        // contravariant.
        SortedSet<Circle> circlesByArea =
            new SortedSet<Circle>(new ShapeAreaComparer())
                { new Circle(7.2), new Circle(100), null, new Circle(.01) };

        foreach (Circle c in circlesByArea)
        {
            Console.WriteLine(c == null ? "null" : "Circle with area " +
c.Area);
        }
    }
}

/* This code example produces the following output:

null
Circle with area 0.000314159265358979
Circle with area 162.860163162095
Circle with area 31415.9265358979
*/

```

[Back to top](#)

Using Variance in Delegates (C#)

Visual Studio 2015

Updated: July 20, 2015

For the latest documentation on Visual Studio 2017 RC, see [Visual Studio 2017 RC Documentation](#).

When you assign a method to a delegate, *covariance* and *contravariance* provide flexibility for matching a delegate type with a method signature. Covariance permits a method to have

return type that is more derived than that defined in the delegate. Contravariance permits a method that has parameter types that are less derived than those in the delegate type.

Example 1: Covariance

Description

This example demonstrates how delegates can be used with methods that have return types that are derived from the return type in the delegate signature. The data type returned by `DogsHandler` is of type `Dogs`, which derives from the `Mammals` type that is defined in the delegate.

Code

C#

```
class Mammals{}
class Dogs : Mammals{}

class Program
{
    // Define the delegate.
    public delegate Mammals HandlerMethod();

    public static Mammals MammalsHandler()
    {
        return null;
    }

    public static Dogs DogsHandler()
    {
        return null;
    }

    static void Test()
    {
        HandlerMethod handlerMammals = MammalsHandler;

        // Covariance enables this assignment.
        HandlerMethod handlerDogs = DogsHandler;
    }
}
```

Example 2: Contravariance

Description

This example demonstrates how delegates can be used with methods that have parameters of a type that are base types of the delegate signature parameter type. With contravariance, you can use one event handler instead of separate handlers. For example, you can create an

event handler that accepts an `EventArgs` input parameter and use it with a `Button.MouseClick` event that sends a `MouseEventArgs` type as a parameter, and also with a `TextBox.KeyDown` event that sends a `KeyEventArgs` parameter.

Code

C#

```
// Event handler that accepts a parameter of the EventArgs type.
private void MultiHandler(object sender, System.EventArgs e)
{
    label1.Text = System.DateTime.Now.ToString();
}

public Form1()
{
    InitializeComponent();

    // You can use a method that has an EventArgs parameter,
    // although the event expects the KeyEventArgs parameter.
    this.button1.KeyDown += this.MultiHandler;

    // You can use the same method
    // for an event that expects the MouseEventArgs parameter.
    this.button1.MouseClick += this.MultiHandler;
}
```

<https://msdn.microsoft.com/en-in/library/mt654057.aspx>

Generic Delegates with Variant Type Parameters

In the .NET Framework 4, the `Func` generic delegates, such as [Func<T, TResult>](#), have covariant return types and contravariant parameter types. The `Action` generic delegates, such as [Action<T1, T2>](#), have contravariant parameter types. This means that the delegates can be assigned to variables that have more derived parameter types and (in the case of the `Func` generic delegates) less derived return types.

Note

The last generic type parameter of the `Func` generic delegates specifies the type of the return value in the delegate signature. It is covariant (`out` keyword), whereas the other generic type

parameters are contravariant (`in` keyword).

The following code illustrates this. The first piece of code defines a class named `Base`, a class named `Derived` that inherits `Base`, and another class with a static method (Shared in Visual Basic) named `MyMethod`. The method takes an instance of `Base` and returns an instance of `Derived`. (If the argument is an instance of `Derived`, `MyMethod` returns it; if the argument is an instance of `Base`, `MyMethod` returns a new instance of `Derived`.) In `Main()`, the example creates an instance of `Func<Base, Derived>` (`Func(Of Base, Derived)` in Visual Basic) that represents `MyMethod`, and stores it in the variable `f1`.

C#

VB

```
public class Base {}
public class Derived : Base {}

public class Program
{
    public static Derived MyMethod(Base b)
    {
        return b as Derived ?? new Derived();
    }

    static void Main()
    {
        Func<Base, Derived> f1 = MyMethod;
    }
}
```

The second piece of code shows that the delegate can be assigned to a variable of type `Func<Base, Base>` (`Func(Of Base, Base)` in Visual Basic), because the return type is covariant.

C#

VB

```
// Covariant return type.
Func<Base, Base> f2 = f1;
Base b2 = f2(new Base());
```

The third piece of code shows that the delegate can be assigned to a variable of type `Func<Derived, Derived>` (`Func(Of Derived, Derived)` in Visual Basic), because the parameter type is contravariant.

C#

VB

```
// Contravariant parameter type.
Func<Derived, Derived> f3 = f1;
Derived d3 = f3(new Derived());
```

The final piece of code shows that the delegate can be assigned to a variable of type `Func<Derived, Base>` (`Func(Of Derived, Base)` in Visual Basic), combining the effects of the contravariant parameter type and the covariant return type.

C#

VB

```
// Covariant return type and contravariant parameter type.
Func<Derived, Base> f4 = f1;
Base b4 = f4(new Derived());
```

Variance in Generic and Non-Generic Delegates

In the preceding code, the signature of `MyMethod` exactly matches the signature of the constructed generic delegate: `Func<Base, Derived>` (`Func(Of Base, Derived)` in Visual Basic). The example shows that this generic delegate can be stored in variables or method parameters that have more derived parameter types and less derived return types, as long as all the delegate types are constructed from the generic delegate type [Func<T, TResult>](#).

This is an important point. The effects of covariance and contravariance in the type parameters of generic delegates are similar to the effects of covariance and contravariance in ordinary delegate binding (see [Variance in Delegates](#)). However, variance in delegate binding works with all delegate types, not just with generic delegate types that have variant type parameters. Furthermore, variance in delegate binding enables a method to be bound to any delegate that has more restrictive parameter types and a less restrictive return type, whereas the assignment of generic delegates works only if both delegate types are constructed from the same generic type definition.

The following example shows the combined effects of variance in delegate binding and variance in generic type parameters. The example defines a type hierarchy that includes three types, from least derived (`Type1`) to most derived (`Type3`). Variance in ordinary delegate binding is used to bind a method with a parameter type of `Type1` and a return type of `Type3` to a generic delegate with a parameter type of `Type2` and a return type of `Type2`. The resulting generic delegate is then assigned to another variable whose generic delegate type has a parameter of type `Type3` and a return type of `Type1`, using the covariance and contravariance of generic type parameters. The second assignment requires both the variable type and the delegate type to be constructed from the same generic type definition, in this case, [Func<T, TResult>](#).

C#

VB

```
using System;
```

```

public class Type1 {}
public class Type2 : Type1 {}
public class Type3 : Type2 {}

public class Program
{
    public static Type3 MyMethod(Type1 t)
    {
        return t as Type3 ?? new Type3();
    }

    static void Main()
    {
        Func<Type2, Type2> f1 = MyMethod;

        // Covariant return type and contravariant parameter type.
        Func<Type3, Type1> f2 = f1;
        Type1 t1 = f2(new Type3());
    }
}

```

[Back to top](#)

Defining Variant Generic Interfaces and Delegates

Starting with the .NET Framework 4, Visual Basic and C# have keywords that enable you to mark the generic type parameters of interfaces and delegates as covariant or contravariant.

Note

Starting with the .NET Framework version 2.0, the common language runtime supports variance annotations on generic type parameters. Prior to the .NET Framework 4, the only way to define a generic class that has these annotations is to use Microsoft intermediate language (MSIL), either by compiling the class with [ilasm.exe \(IL Assembler\)](#) or by emitting it in a dynamic assembly.

A covariant type parameter is marked with the `out` keyword (`Out` keyword in Visual Basic, + for the [MSIL Assembler](#)). You can use a covariant type parameter as the return value of a method that belongs to an interface, or as the return type of a delegate. You cannot use a covariant type parameter as a generic type constraint for interface methods.

Note

If a method of an interface has a parameter that is a generic delegate type, a covariant type parameter of the interface type can be used to specify a contravariant type parameter of the delegate type.

A contravariant type parameter is marked with the `in` keyword (`In` keyword in Visual Basic, – for the [MSIL Assembler](#)). You can use a contravariant type parameter as the type of a parameter of a method that belongs to an interface, or as the type of a parameter of a delegate. You can use a contravariant type parameter as a generic type constraint for an interface method.

Only interface types and delegate types can have variant type parameters. An interface or delegate type can have both covariant and contravariant type parameters.

Visual Basic and C# do not allow you to violate the rules for using covariant and contravariant type parameters, or to add covariance and contravariance annotations to the type parameters of types other than interfaces and delegates. The [MSIL Assembler](#) does not perform such checks, but a [TypeLoadException](#) is thrown if you try to load a type that violates the rules.

For information and example code, see [Variance in Generic Interfaces](#).

[Back to top](#)

List of Variant Generic Interface and Delegate Types

In the .NET Framework 4, the following interface and delegate types have covariant and/or contravariant type parameters.

Type	Covariant type parameters	Contravariant type parameters
Action<T> to Action<T1, T2, T3, T4, T5, T6, T7, T8, T9, T10, T11, T12, T13, T14, T15, T16>		Yes

Type	Covariant type parameters	Contravariant type parameters
<u>Comparison<T></u>		Yes
<u>Converter<TInput, TOutput></u>	Yes	Yes
<u>Func<TResult></u>	Yes	
<u>Func<T, TResult> to Func<T1, T2, T3, T4, T5, T6, T7, T8, T9, T10, T11, T12, T13, T14, T15, T16, TResult></u>	Yes	Yes
<u>IComparable<T></u>		Yes
<u>Predicate<T></u>		Yes
<u>IComparer<T></u>		Yes
<u>IEnumerable<T></u>	Yes	

Type	Covariant type parameters	Contravariant type parameters
IEnumerator<T>	Yes	
IEqualityComparer<T>		Yes
IGrouping<TKey, TElement>	Yes	
IOrderedEnumerable<TElement>	Yes	
IOrderedQueryable<T>	Yes	
IQueryable<T>	Yes	

[https://msdn.microsoft.com/en-in/library/dd799517\(v=vs.110\).aspx](https://msdn.microsoft.com/en-in/library/dd799517(v=vs.110).aspx)

C# Versions

- **C# 4.0** released with .NET 4 and VS2010 (April 2010). Major new features: late binding (dynamic), delegate and interface generic variance, more COM support, named arguments, tuple data type and optional parameters
- **C# 5.0** released with .NET 4.5 and VS2012 (August 2012). [Major features](#): async programming, caller info attributes. Breaking change: [loop variable closure](#).
- **C# 6.0** released with .NET 4.6 and VS2015 (July 2015). Implemented by [Roslyn](#). Features: initializers for automatically implemented properties, using directives to import static members, exception filters, indexed members and element initializers, await in catch and finally, extension Add methods in collection initializers.

- **C# 7.0** Major new features: tuples, ref locals and ref return, pattern matching (including pattern-based switch statements), inline out parameter declarations, local functions, binary literals, digit separators, and arbitrary async returns.

New Features in .NET 4.5 and 5.0

<http://www.c-sharpcorner.com/UploadFile/princy.scorpin/new-features-in-net-4-5-and-5-0/>

This is my attempt at explaining some new features in .NET 4.5 and 5.0.

I am explaining the following features:

- Parallel foreach
- BigInteger
- Expando Objects
- Named and Optional Parameters
- Tuple

1. Parallel.ForEach

Parallel.ForEach is a feature introduced by the Task Parallel Library (TPL). This feature helps you run your loop in parallel. You need to have a multi-processor to use of this feature.

Simple foreach loop

```
foreach (string i in listStrings)
```

```
{
```

```
.....
```

```
}
```

Parallel foreach

```
Parallel.ForEach(listStrings, text=>
```

```
{
```

```
.....
```

```
});
```

2. BigInteger

BigInteger is added as a new feature in the System.Numerics DLL. It is an immutable type that represents a very large integer whose value has no upper or lower bounds.

```
BigInteger obj = new BigInteger("123456789123456789123456789");
```

3. ExpandoObject

The ExpandoObject is part of the Dynamic Language Runtime (DLR). One can add and remove members from this object at run time.

Create a dynamic instance.

```
dynamic Person = new ExpandoObject();
```

```
Person.ID = 1001;
```

```
Person.Name = "Princy";
```

```
Person.LastName = "Gupta";
```

4. Named and Optional Parameters

Optional Parameters

A developer can now have some optional parameters by providing default values for them. PFB how to create optional parameters.

```
Void PrintName(string a, string b, string c = "princy")
```

```
{
```

```
    Console.WriteLine(a, b, c)
```

```
}
```

We can now call the function PrintName() by passing either two or three parameters as in the following:

```
PrintName("Princy", "Gupta", "Jain");
```

```
PrintName("Princy", "Gupta");
```

Output

PrincyGuptaJain
PrincyGuptaprincy

Note: An optional parameter can only be at the end of the parameter list.

Named Parameters

With this new feature the developer can pass values to parameters by referring to parameters by names.

```
Void PrintName(string A, string B
```

```
{  
  
}
```

Function call

```
PrintName (B: "Gupta", A: "Princy");
```

With this feature we don't need to pass parameters in the same order to a function.

5. Tuple

A Tuple provides us the ability to store various types in a single object.

The following shows how to create a tuple.

```
Tuple<int, string, bool> tuple = new Tuple<int, string, bool>(1,"princy", true);
```

```
Var tupleobj = Tuple.Create(1, "Princy", true);
```

In order to access the data inside the tuple, use the following:

```
string name = tupleobj.Item2;
```

```
int age = tupleobj.Item1;
```

```
bool obj = tupleobj.Item3;
```

List of New Features in C# 6.0

We can discuss about each of the new features, but first, below is a list of few features in C# 6.0:

1. Auto Property Initializer
2. Primary Constructors
3. Dictionary Initializer
4. Declaration Expressions
5. **Static** Using
6. **await** inside **catch** block
7. Exception Filters
8. Conditional Access Operator to check **NULL** Values

1. Auto Property Initializer

Before

The only way to initialize an Auto Property is to implement an explicit constructor and set property values inside it.

[Hide](#) [Copy Code](#)

```
public class AutoPropertyBeforeCsharp6
{
    private string _postTitle = string.Empty;
    public AutoPropertyBeforeCsharp6()
    {
        //assign initial values
        PostID = 1;
        PostName = "Post 1";
    }

    public long PostID { get; set; }

    public string PostName { get; set; }

    public string PostTitle
    {
        get { return _postTitle; }
        protected set
        {
            _postTitle = value;
        }
    }
}
```

After

In C# 6, auto implemented property with initial value can be initialized without having to write the constructor. We can simplify the above example to the following:

[Hide](#) [Copy Code](#)

```
public class AutoPropertyInCsharp6
{
    public long PostID { get; } = 1;

    public string PostName { get; } = "Post 1";

    public string PostTitle { get; protected set; } = string.Empty;
}
```

2. Primary Constructors

We mainly use constructor to initialize the values inside it. (Accept parameter values and assign those parameters to instance properties).

Before

[Hide](#) [Copy Code](#)

```
public class PrimaryConstructorsBeforeCSharp6
{
    public PrimaryConstructorsBeforeCSharp6(long postId, string postName, string postTitle)
    {
        PostID = postId;
        PostName = postName;
        PostTitle = postTitle;
    }

    public long PostID { get; set; }
    public string PostName { get; set; }
    public string PostTitle { get; set; }
}
```

After

[Hide](#) [Copy Code](#)

```
public class PrimaryConstructorsInCSharp6(long postId, string postName, string postTitle)
{
    public long PostID { get; } = postId;
    public string PostName { get; } = postName;
    public string PostTitle { get; } = postTitle;
}
```

In C# 6, primary constructor gives us a shortcut syntax for defining constructor with parameters. Only one primary constructor per class is allowed.

If you look closely at the above example, we moved the parameters initialization beside the class name.

You may get the following error **“Feature ‘primary constructor’ is only available in ‘experimental’ language version.”** To solve this, we need to edit

the **SolutionName.csproj** file to get rid of this error. What you have to do is we need to add additional setting after **WarningTag**.

[Hide](#) [Copy Code](#)

```
<LangVersion>experimental</LangVersion>

<PropertyGroup Condition=" '$(Configuration)|$(Platform)' == 'Debug|AnyCPU' ">
  <DebugSymbols>>true</DebugSymbols>
  <DebugType>full</DebugType>
  <Optimize>>false</Optimize>
  <OutputPath>bin\Debug\</OutputPath>
  <DefineConstants>DEBUG;TRACE</DefineConstants>
  <ErrorReport>prompt</ErrorReport>
  <WarningLevel>4</WarningLevel>
  <LangVersion>experimental</LangVersion>
</PropertyGroup>
```

Feature 'primary constructor' is only available in 'experimental' language version

3. Dictionary_INITIALIZER

Before

The old way of writing a **dictionary** initializer is as follows:

[Hide](#) [Copy Code](#)

```
public class DictionaryInitializerBeforeCSharp6
{
    public Dictionary<string, string> _users = new Dictionary<string, string>()
    {
        { "users", "Venkat Baggu Blog" },
        { "Features", "Whats new in C# 6" }
    };
}
```

After

We can define **dictionary** initializer like an array using square brackets.

[Hide](#) [Copy Code](#)

```
public class DictionaryInitializerInCSharp6
{
    public Dictionary<string, string> _users { get; } = new Dictionary<string, string>()
    {
        ["users"] = "Venkat Baggu Blog",
        ["Features"] = "Whats new in C# 6"
    };
}
```

4. Declaration Expressions

Before

[Hide](#) [Copy Code](#)

```
public class DeclarationExpressionsBeforeCShapr6()
{
    public static int CheckUserExist(string userId)
    {
        //Example 1
        int id;
        if (!int.TryParse(userId, out id))
        {
            return id;
        }
        return id;
    }

    public static string GetUserRole(long userId)
    {
        ///Example 2
        var user = _userRepository.Users.FindById(x => x.UserID == userId);
        if (user!=null)
        {
            // work with address ...

            return user.City;
        }
    }
}
```

After

In C# 6, you can declare an local variable in the middle of the expression. With declaration expressions, we can also declare variables inside **if** statements and various loop statements.

[Hide](#) [Copy Code](#)

```
public class DeclarationExpressionsInCShapr6()
{
    public static int CheckUserExist(string userId)
    {
        if (!int.TryParse(userId, out var id))
        {
            return id;
        }
        return 0;
    }

    public static string GetUserRole(long userId)
    {
        ///Example 2
        if ((var user = _userRepository.Users.FindById(x => x.UserID == userId) != null)
        {
            // work with address ...

```



```

        return user.City;
    }
}

```

5. Using Statics

Before

To you **static** members, you don't need an instance of object to invoke a method. You use syntax as follows:

[Hide](#) [Copy Code](#)

```

TypeName.MethodName

```

[Hide](#) [Copy Code](#)

```

public class StaticUsingBeforeCSharp6
{
    public void TestMethod()
    {
        Console.WriteLine("Static Using Before C# 6");
    }
}

```

After

In C# 6, you have the ability to use the **Static Members** without using the type name. You can import the **static** classes in the namespaces.

If you look at the below example, we moved the **Static Console** class to the namespace:

[Hide](#) [Copy Code](#)

```

using System.Console;
namespace newfeatureincsharp6
{
    public class StaticUsingInCSharp6
    {
        public void TestMethod()
        {
            WriteLine("Static Using Before C# 6");
        }
    }
}

```

6. await Inside catch Block

Before C# 6, **await** keyword is not available inside the **catch** and **finally** blocks. In C# 6, we can finally use the **await** keyword inside **catch** and **finally** blocks.

[Hide](#) [Copy Code](#)

```
try
{
    //Do something
}
catch (Exception)
{
    await Logger.Error("exception logging")
}
```

7. Exception Filters

Exception filters allow you a feature to check an **if** condition before the **catch** block executes.

Consider an example that an exception occurred now we want to check if the **InnerException null**, then it will execute **catch** block.

[Hide](#) [Copy Code](#)

```
//Example 1
try
{
    //Some code
}
catch (Exception ex) if (ex.InnerException == null)
{
    //Do work
}
```

//Before C# 6 we write the above code as follows

```
//Example 1
try
{
    //Some code
}
catch (Exception ex)
{
    if(ex.InnerException != null)
    {
        //Do work;
    }
}
```

8. Conditional Access Operator to Check NULL Values?

Consider an example that we want to retrieve a **UserRanking** based on the **UserID** only if **UserID** is not **null**.

Before

Hide Copy Code

```
var userRank = "No Rank";  
if(UserID != null)  
{  
    userRank = Rank;  
}  
  
//or  
  
var userRank = UserID != null ? Rank : "No Rank"
```

After

Hide Copy Code

```
var userRank = UserID?.Rank ?? "No Rank";
```

The post [New features in C# 6.0](#) appeared first on [Venkat Baggu Blog](#).
<https://www.codeproject.com/articles/874205/new-features-in-csharp>

List of All New Features in C# 6.0: Part 1



- [Nitin Pandit](#)
- Dec 30 2014

- [Article](#)

-
- [38](#)
- [97](#)

- 176.3k

-

- -

- -

- -

- -

- -

-

-

-

-

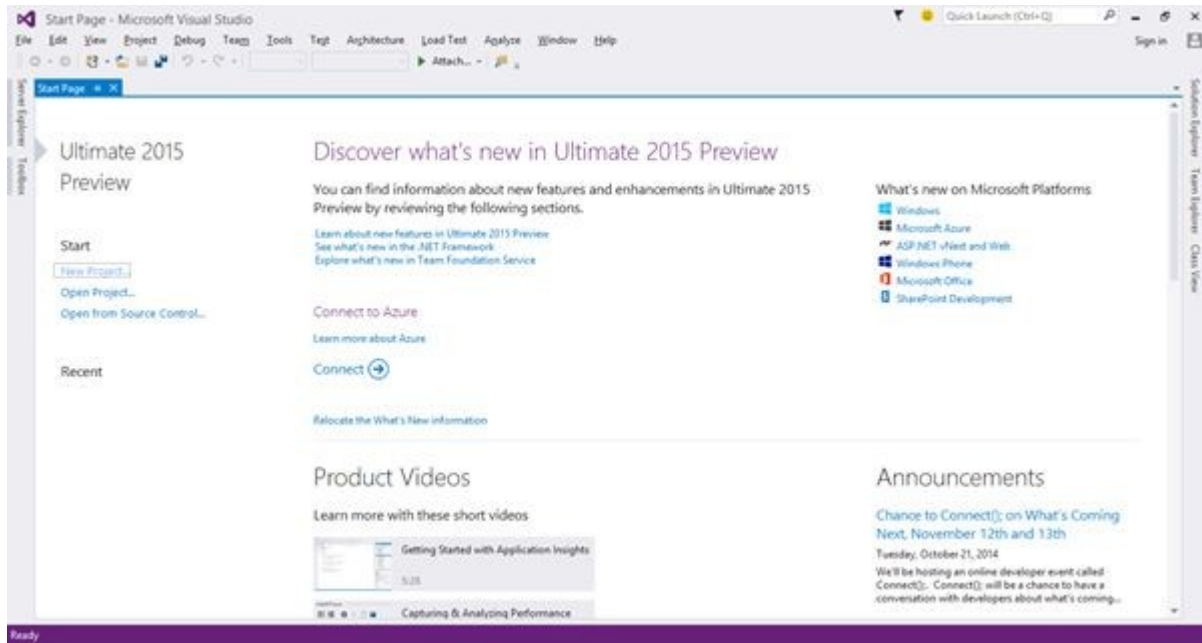
-

- -

[FullCSharp.zip](#)

[Download 100% FREE Spire Office APIs](#)

Microsoft has announced some new keywords and some new behavior of C# 6.0 in Visual Studio 2015.

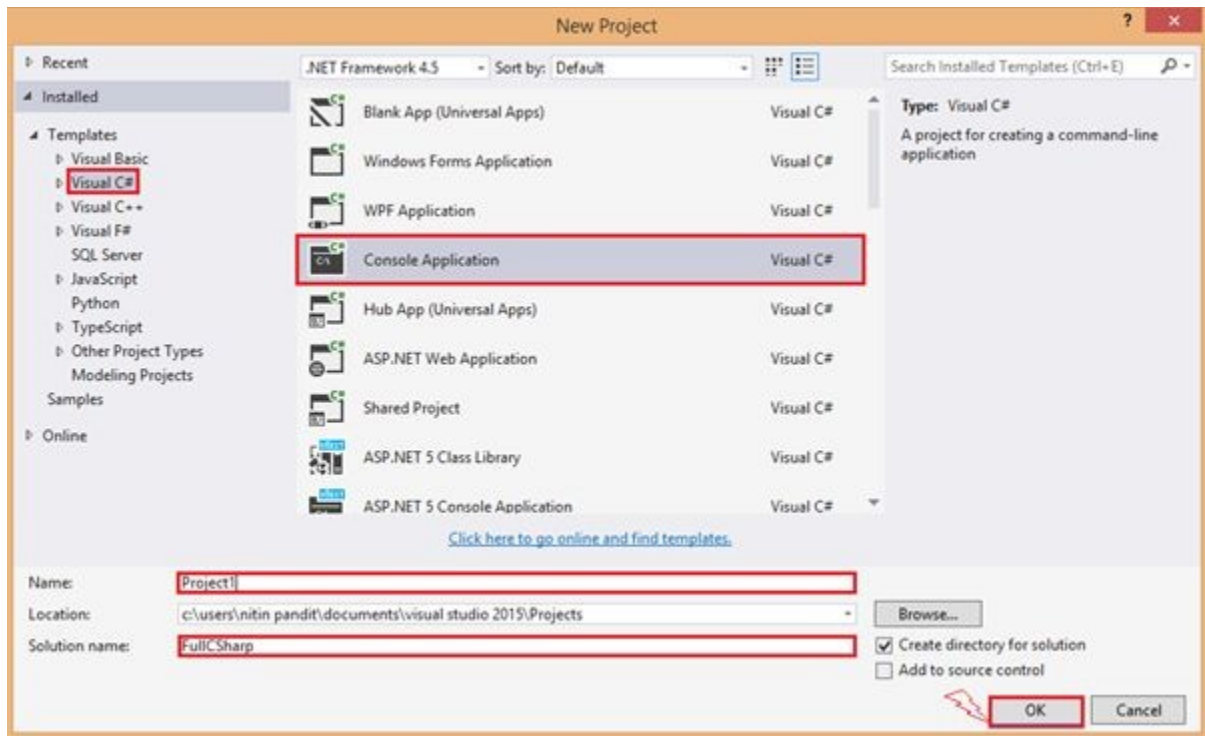


In this article we will learn the following topics.

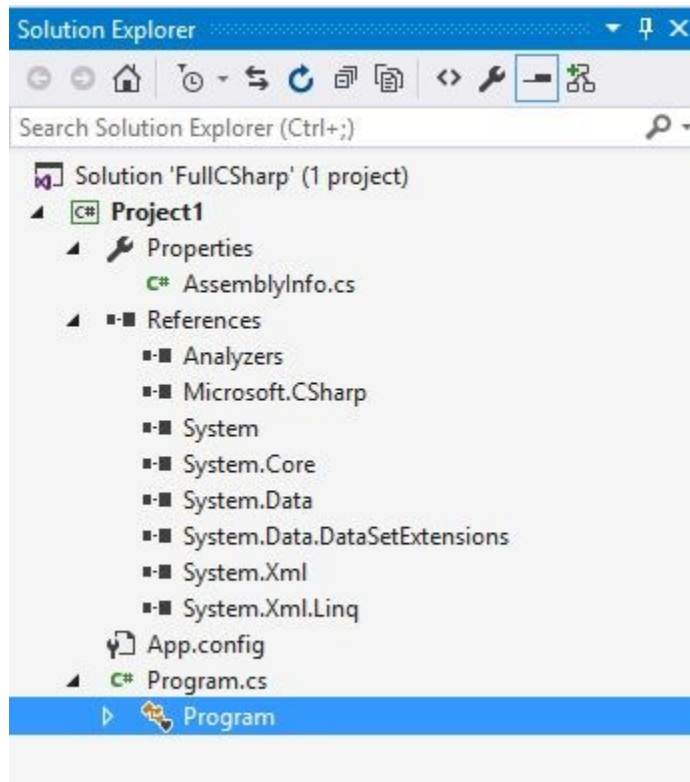
1. using Static.
2. Auto property initializer.
3. Dictionary Initializer.
4. nameof Expression.
5. New way for Exception filters.
6. await in catch and finally block.
7. Null – Conditional Operator.
8. Expression – Bodied Methods
9. Easily format strings – String interpolation

For testing all

Open **Visual Studio 2015** and select "File" -> "New" -> "Project...".



Click OK and then you will get a solution that you will see in the Solution Explorer.

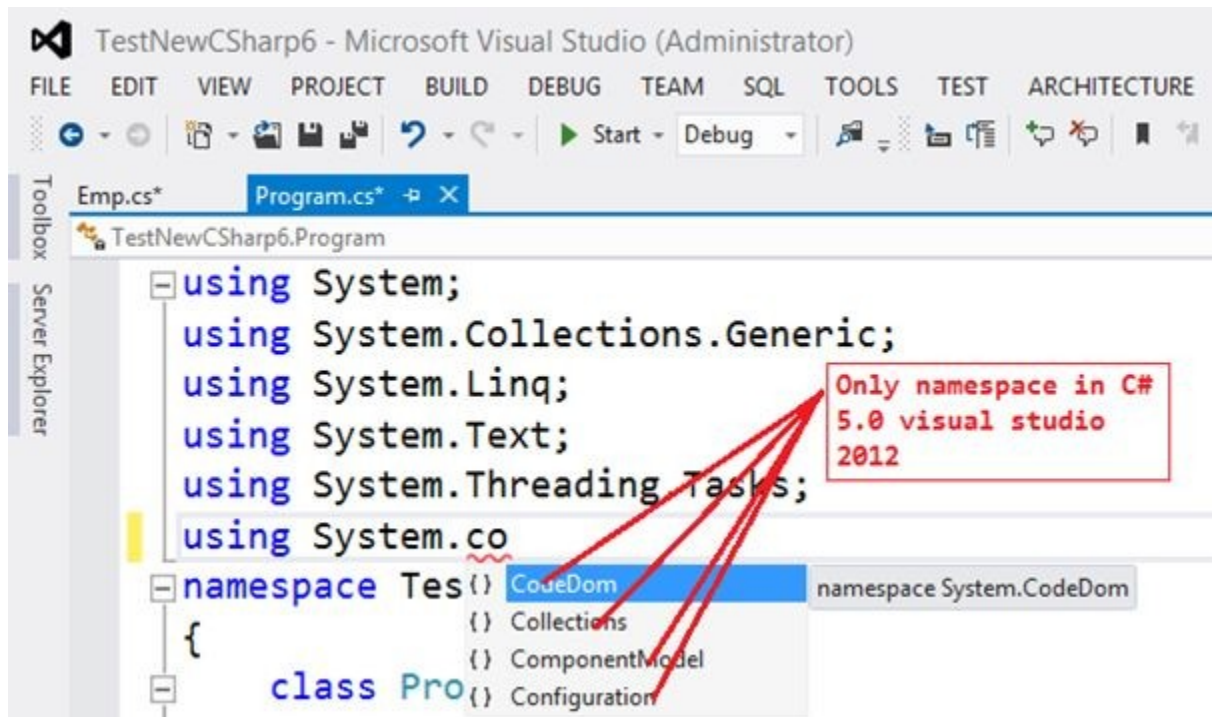


Now just do something with your **program.cs** file to test your compile time code.

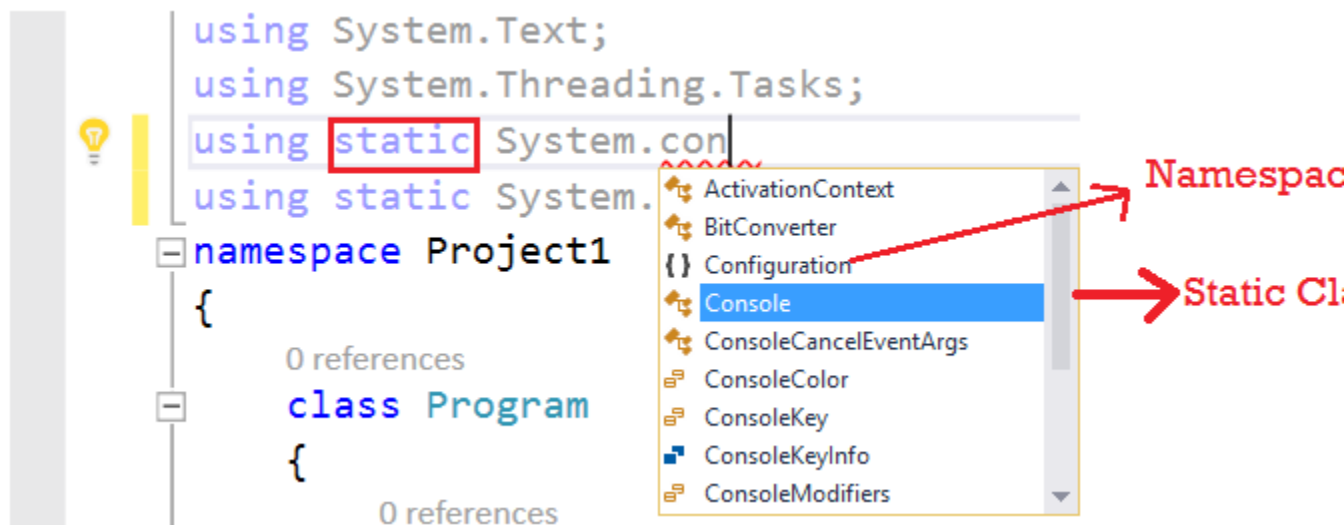
1. using Static

This is a new concept in C# 6.0 that allows us to use any class that is static as a namespace that is very useful for every developer in that code file where we need to call the static methods from a static class like in a number of times we need to call many methods from **Convert.ToInt32()** or **Console.Write()**, **Console.WriteLine()** so we need to write the class name first then the method name every time in C# 5.0. In C# 6.0 however Microsoft announced a new behavior to our cs compiler that allows me to call all the static methods from the static class without the name of the classes because now we need to first use our static class name in starting with all the namespaces.

In C# 5.0



In C# 6.0



Code in 5.0

1. `using System;`
2. `using System.Collections.Generic;`


```
3. using System.Linq;
4. using System.Text;
5. using System.Threading.Tasks;
6. namespace TestNewCSharp6
7. {
8.     class Program
9.     {
10.         static void Main(string[] args)
11.         {
12.             Console.WriteLine("Enter first value ");
13.             int val1 = Convert.ToInt32(Console.ReadLine());
14.             Console.WriteLine("Enter next value ");
15.             int val2 = Convert.ToInt32(Console.ReadLine());
16.             Console.WriteLine("sum : {0}", (val1 + val2));
17.             Console.ReadLine();
18.         }
19.     }
20. }
```

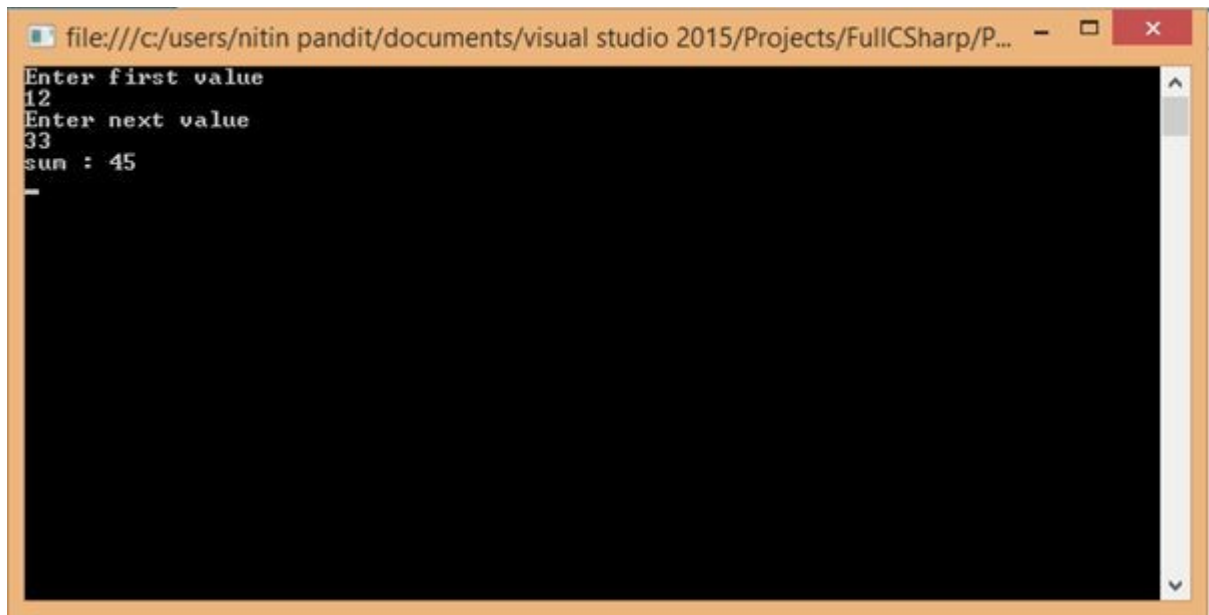
Code in C# 6.0

```
21. using System;
22. using System.Collections.Generic;
23. using System.Linq;
24. using System.Text;
25. using System.Threading.Tasks;
26. using static System.Convert;
27. using static System.Console;
```

```
28. namespace Project1
29. {
30.     class Program
31.     {
32.         static void Main(string[] args)
33.         {
34.             WriteLine("Enter first value ");
35.             int val1 =.ToInt32(ReadLine());
36.             WriteLine("Enter next value ");
37.             int val2 =.ToInt32(ReadLine());
38.             WriteLine("sum : {0}", (val1+val2));
39.             ReadLine();
40.         }
41.     }
42. }
```

Now you can see the difference for yourself, there is less code but the output will be the same.

Output

A screenshot of a console window from Visual Studio. The window title bar shows the file path: file:///c:/users/nitin pandit/documents/visual studio 2015/Projects/FullCSharp/P... The console output is as follows:

```
Enter first value
12
Enter next value
33
sum : 45
```

2. Auto property initializer

Auto property initializer is a new concept to set the value of a property during of property declaration. We can set the default value of a read-only property, it means a property that only has a {get;} attribute. In the previous version of C# 5.0 we can set the values of the property in the default constructor of the class. Let's have an example. Suppose we need to set some property's value of a class as in the following:

In C# 5.0

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace TestNewCSharp6
{
    class Emp
    {
        public Emp()
        {
            Name = "nitin";
            Age = 25;
            Salary = 999;
        }
        public string Name { get; set; }
        public int Age { get; set; }
        public int Salary { get; private set; }
    }
}

```

Code

1. **using** System;
2. **using** System.Collections.Generic;
3. **using** System.Linq;
4. **using** System.Text;
5. **namespace** TestNewCSharp6
6. {
7. **class** Emp
8. {
9. **public** Emp()
10. {

```

11.     Name = "nitin";
12.     Age = 25;
13.     Salary = 999;
14. }
15.     public string Name { get; set; }
16.     public int Age { get; set; }
17.     public int Salary { get; private set; }
18. }
19. }

```

Here we can set the default value of my property in only by constructors.

In C# 6.0

```

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

namespace Project2
{
    0 references
    class Emp
    {
        0 references
        public string Name { get; set; }="nitin";
        0 references
        public int Age { get; set; }=25;
        0 references
        public int Salary { get; }=999;
    }
}

```

Code

```
20. using System;
21. using System.Collections.Generic;
22. using System.Linq;
23. using System.Text;
24. using System.Threading.Tasks;
25.
26. namespace Project2
27. {
28.     class Emp
29.     {
30.         public string Name { get; set; }="nitin";
31.         public int Age { get; set; }=25;
32.         public int Salary { get; }=999;
33.     }
34. }
```

Here we can initialization the default value for the property in the same line.

3. Dictionary Initializer

Dictionary initializer is a new concept in C# 6.0. We can directly initialize a value of a key in a Dictionary Collection with it, either the key in the collection would be a string data type or any other data type. Let's see the declaration syntax in both versions like in C# 5.0 and also in C# 6.0 respectively.

C# 5.0

```

class Program
{
    static void Main(string[] args)
    {
        Dictionary<string, string> dicStr = new Dictionary<string, string>()
        {
            {"Nitin", "Noida"},
            {"Sonu", "Baraut"},
            {"Rahul", "Delhi"},
        };
        dicStr["Mohan"] = "Noida";
        foreach (var item in dicStr)
        {
            Console.WriteLine(item.Key + " " + item.Value);
        }
        Console.WriteLine("*****");
        Dictionary<int, string> dicInt = new Dictionary<int, string>()
        {
            {1, "Nitin"},
            {2, "Sonu"},
            {3, "Mohan"},
        };
        dicInt[4] = "Rahul";
        foreach (var item in dicInt)
        {
            Console.WriteLine(item.Key + " " + item.Value);
        }
        Console.Read();
    }
}

```

Code

1. **using** System;
2. **using** System.Collections.Generic;
3. **using** System.Data;
4. **using** System.Linq;
5. **using** System.Text;
6. **using** System.Threading.Tasks;
7. **namespace** TestNewCSharp6
8. {
9. **class** Program
10. {
11. **static void** Main(**string**[] args)
12. {

```
13. Dictionary<string, string> dicStr = new Dictionary<string, string>()
14. {
15.     {"Nitin", "Noida"},
16.     {"Sonu", "Baraut"},
17.     {"Rahul", "Delhi"},
18. };
19. dicStr["Mohan"] = "Noida";
20. foreach (var item in dicStr)
21. {
22.     Console.WriteLine(item.Key + " " + item.Value);
23. }
24. Console.WriteLine("*****");
25. Dictionary<int, string> dicInt = new Dictionary<int, string>()
26. {
27.     {1, "Nitin"},
28.     {2, "Sonu"},
29.     {3, "Mohan"},
30. };
31. dicInt[4] = "Rahul";
32. foreach (var item in dicInt)
33. {
34.     Console.WriteLine(item.Key + " " + item.Value);
35. }
36. Console.Read();
37. }
38. }
```


39. }

C# 6.0

```
class Program
{
    static void Main(string[] args)
    {
        Dictionary<string, string> dicStr = new Dictionary<string, string>()
        {
            ["Nitin"]="Noida",
            ["Sonu"]="Baraut",
            ["Rahul"]="Delhi",
        };
        dicStr["Mohan"] = "Noida";
        foreach (var item in dicStr)
        {
            Console.WriteLine(item.Key + " " + item.Value);
        }
        Console.WriteLine("*****");
        Dictionary<int, string> dicInt = new Dictionary<int, string>()
        {
            [1]="Nitin",
            [2]="Sonu",
            [3]="Mohan"
        };
        dicInt[4] = "Rahul";
        foreach (var item in dicInt)
        {
            Console.WriteLine(item.Key + " " + item.Value);
        }
        Console.Read();
    }
}
```

Here we can initialize the Dictionary values directly by "=" operator

Code

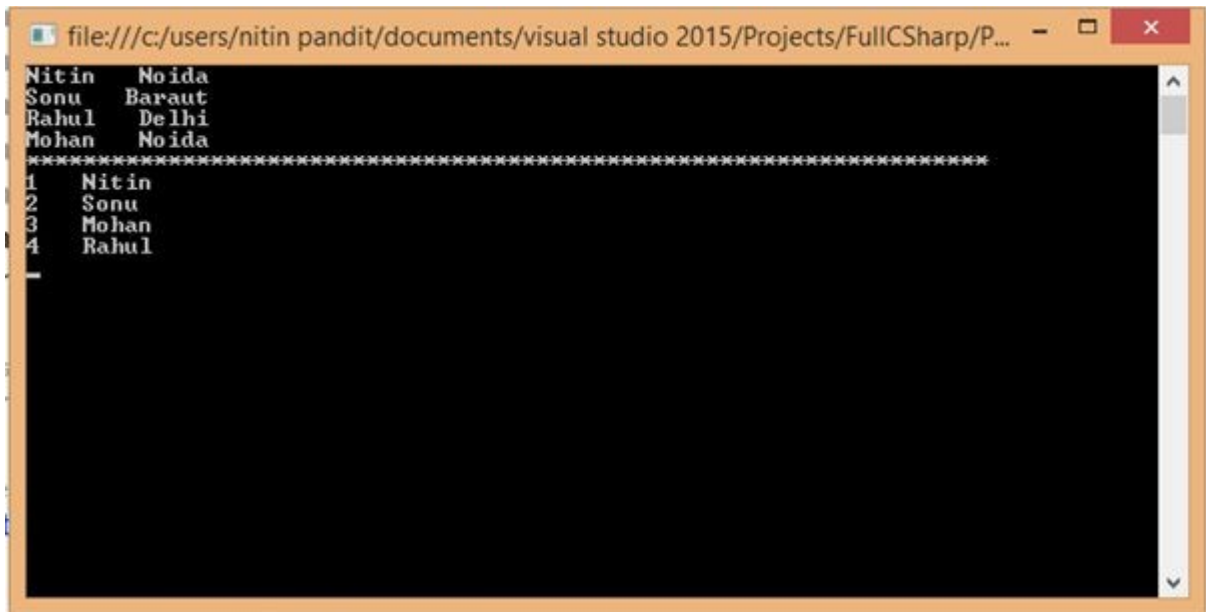
```
40. using System;
41. using System.Collections.Generic;
42. using System.Data;
43. using System.Linq;
44. using System.Text;
45. using System.Threading.Tasks;
46. namespace Project3
47. {
48.     class Program
49.     {
50.         static void Main(string[] args)
```

```
51.     {
52.         Dictionary<string, string> dicStr = new Dictionary<string, string>()
53.     {
54.         ["Nitin"]="Noida",
55.         ["Sonu"]="Baraut",
56.         ["Rahul"]="Delhi",
57.     };
58.     dicStr["Mohan"] = "Noida";
59.     foreach (var item in dicStr)
60.     {
61.         Console.WriteLine(item.Key + " " + item.Value);
62.     }
63.     Console.WriteLine("*****
*****");
64.     Dictionary<int, string> dicInt = new Dictionary<int, string>()
65.     {
66.         [1]="Nitin",
67.         [2]="Sonu",
68.         [3]="Mohan"
69.     };
70.     dicInt[4] = "Rahul";
71.     foreach (var item in dicInt)
72.     {
73.         Console.WriteLine(item.Key + " " + item.Value);
74.     }
75.     Console.Read();
76. }
```

```
77. }  
78. }
```

Here we can initialize the Dictionary values directly by the “=” operator and in C# 5.0 we need to create an object as a {key,value} pair and the output will be the same in both versions.

Output



```
file:///c:/users/nitin pandit/documents/visual studio 2015/Projects/FullCSharp/P...  
Nitin Noida  
Sonu Baraut  
Rahul Delhi  
Mohan Noida  
*****  
1 Nitin  
2 Sonu  
3 Mohan  
4 Rahul
```

4. nameof Expression

nameof is new keyword in C# 6.0 and it's very usefull from a developer's point of view because when we need to use a property, function or a data member name into a message as a string so we need to use the name as hard-coded in “name” in the string and in the future my property or method's name will be changed so it must change all the messages in every form or every page so it's very complicated to remember that how many number of times you already use the name of them in your project code files and this avoids having hardcoded strings to be specified in our code as well as avoids explicit use of reflection to get the names. Let's have an example.

We have a class:

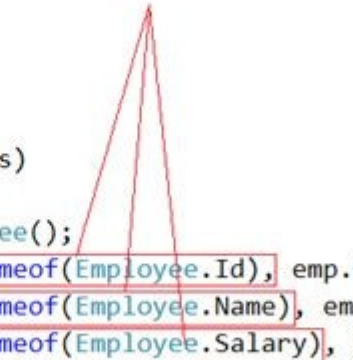
5 references

```
class Employee
{
    2 references
    public int Id { get; set; } = 101;
    2 references
    public string Name { get; set; } = "Nitin";
    2 references
    public int Salary { get; set; } = 9999;
}
```

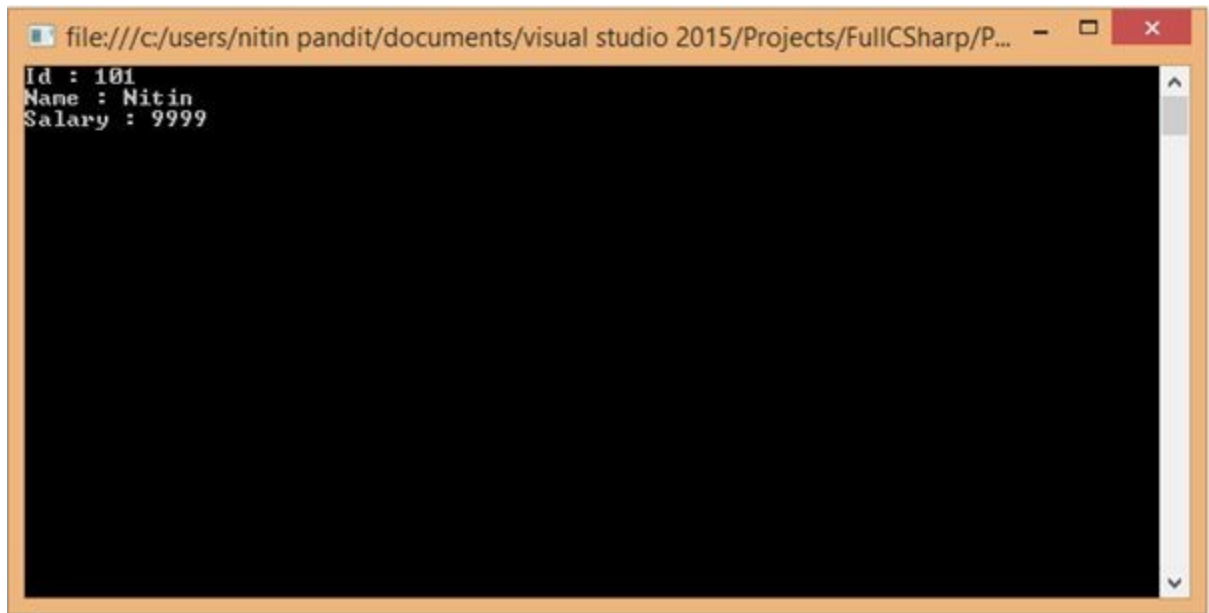
And we need to show the values of this class property to the console and also if we need to print the name of my property too with the message so in C# 6.0 we can use the nameof expression rather than hardcode the name of the property.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Console;
namespace Project4
{
    0 references
    class Program
    {
        0 references
        static void Main(string[] args)
        {
            Employee emp = new Employee();
            WriteLine("{0} : {1}", nameof(Employee.Id), emp.Id);
            WriteLine("{0} : {1}", nameof(Employee.Name), emp.Name);
            WriteLine("{0} : {1}", nameof(Employee.Salary), emp.Salary);
            ReadLine();
        }
    }
}
```

nameof Expression will return string literal of the name of property or a methods



Output



```
file:///c:/users/nitin pandit/documents/visual studio 2015/Projects/FullCSharp/P...
Id : 101
Name : Nitin
Salary : 9999
```

Code

```
1. using System;
2. using System.Collections.Generic;
3. using System.Linq;
4. using System.Text;
5. using System.Threading.Tasks;
6. using System.Console;
7. namespace Project4
8. {
9.     class Program
10.    {
11.        static void Main(string[] args)
12.        {
13.            Employee emp = new Employee();
14.            WriteLine("{0} : {1}", nameof(Employee.Id), emp.Id);
15.            WriteLine("{0} : {1}", nameof(Employee.Name), emp.Name);
```

```
16.     WriteLine("{0} : {1}", nameof(Employee.Salary), emp.Salary);
17.     ReadLine();
18. }
19. }
20. class Employee
21. {
22.     public int Id { get; set; } = 101;
23.     public string Name { get; set; } = "Nitin";
24.     public int Salary { get; set; } = 9999;
25. }
26. }
```

5. Exception filters

Exception filters are a new concept for C#. In C# 6.0 they are already supported by the VB compiler but now they are coming into C#. Exception filters allow us to specify a condition with a catch block so if the condition will return true then the catch block is executed only if the condition is satisfied. This is also a best attribute of new C# 6.0 that makes it easy to do exception filtrations in also that type of code contains a large amount of source code. Let's have an example.

```
using System.Console;
namespace project5
{
    0 references
    class Program
    {
        0 references
        static void Main(string[] args)
        {
            int val1 = 0;
            int val2 = 0;
            try
            {
                WriteLine("Enter first value :");
                val1 = int.Parse(ReadLine());
                WriteLine("Enter Next value :");
                val2 = int.Parse(ReadLine());
                WriteLine("Div : {0}", (val1 / val2));
            }
            catch (Exception ex) if (val2 == 0)
            {
                WriteLine("Can't be Division by zero 0");
            }
            catch (Exception ex)
            {
                WriteLine(ex.Message);
            }
            ReadLine();
        }
    }
}
```

If() condition with catch() block for exception filtration

rest exception will be handle by this catch{}

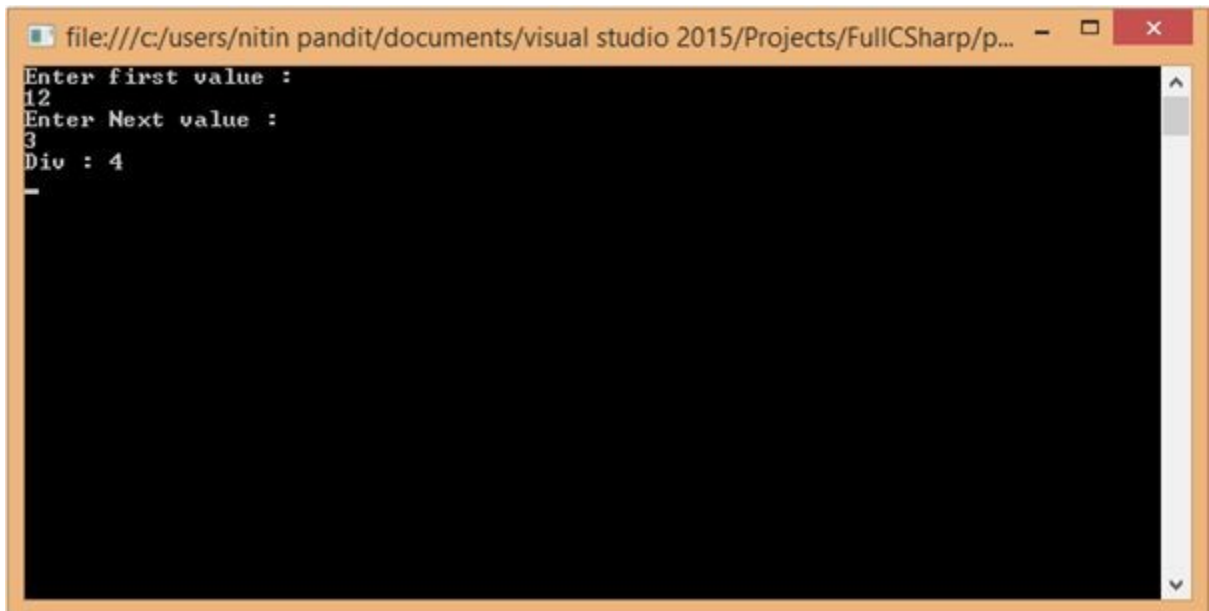
Code

1. `using` System;
2. `using` System.Collections.Generic;
3. `using` System.Linq;
4. `using` System.Text;
5. `using` System.Threading.Tasks;
6. `using` System.Console;
7. `namespace` project5
8. {

```
9.  class Program
10. {
11.     static void Main(string[] args)
12.     {
13.         int val1 = 0;
14.         int val2 = 0;
15.         try
16.         {
17.             WriteLine("Enter first value :");
18.             val1 = int.Parse(ReadLine());
19.             WriteLine("Enter Next value :");
20.             val2 = int.Parse(ReadLine());
21.             WriteLine("Div : {0}", (val1 / val2));
22.         }
23.         catch (Exception ex) if (val2 == 0)
24.         {
25.             WriteLine("Can't be Division by zero ☹");
26.         }
27.         catch (Exception ex)
28.         {
29.             WriteLine(ex.Message);
30.         }
31.         ReadLine();
32.     }
33. }
34. }
```


Output

If all the values are entered by user id correctly then the output will be:



```
file:///c:/users/nitin pandit/documents/visual studio 2015/Projects/FullCSharp/p...
Enter first value :
12
Enter Next value :
3
Div : 4
_
```

If the user enters an invalid value for division, like 0, then it will throw the exception that will be handled by Exception filtration where you mentioned an **if()** with **catch{}** block and the output will be something.

6. Await in catch and finally block

This is a new behavior of C# 6.0 that now we are able to call **async** methods from catch and also from finally. Using async methods are very usefull because we can call then asynchronously and while working with async and **await**, you may have experienced that you want to put some of the result awaiting either in a catch or finally block or in both. Let's suppose we need to call a async method and there is a try and a catch{ } block so when the exception occurs it is thrown in the catch{ } bock. We need to write log information into a file or send a service call to send exception details to the server so call the asynchronous method, so use await in catch{ }, this is only possible in C# 6.0. Let's have an example.

We have a class and there is a method that is async and we need to call this with two parameters and if there is an exception then that will we return an exception and will go to the **catch{}** block and then we will call an async method using await and finally we have called the same in the finally.

```

public class MyMath
{
    1 reference
    public async void Div(int value1, int value2)
    {
        try
        {
            int res = value1 / value2;
            WriteLine("Div : {0}", res);
        }
        catch (Exception ex)
        {
            await asyncMethodForCatch();
        }
        finally
        {
            await asyncMethodForFinally();
        }
    }
    1 reference
    private async Task asyncMethodForFinally()
    {
        WriteLine("Method from async finally Method !!");
    }

    1 reference
    private async Task asyncMethodForCatch()
    {
        WriteLine("Method from async Catch Method !!");
    }
}

```

Call the async div() in Main().

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Console;
namespace project6
{
    0 references
    class Program
    {
        0 references
        static void Main(string[] args)
        {
            MyMath obj = new MyMath();
            obj.Div(12, 0);
            ReadLine();
        }
    }
}

```

Code

1. **using** System;
2. **using** System.Collections.Generic;
3. **using** System.Linq;
4. **using** System.Text;
5. **using** System.Threading.Tasks;
6. **using** System.Console;
7. **namespace** project6
8. {
9. **class** Program

```
10. {
11.     static void Main(string[] args)
12.     {
13.         MyMath obj = new MyMath();
14.         obj.Div(12, 0);
15.         ReadLine();
16.     }
17. }
18. public class MyMath
19. {
20.     public async void Div(int value1, int value2)
21.     {
22.         try
23.         {
24.             int res = value1 / value2;
25.             WriteLine("Div : {0}", res);
26.         }
27.         catch (Exception ex)
28.         {
29.             await asyncMethodForCatch();
30.         }
31.         finally
32.         {
33.             await asyncMethodForFinally();
34.         }
35.     }
36.     private async Task asyncMethodForFinally()
```

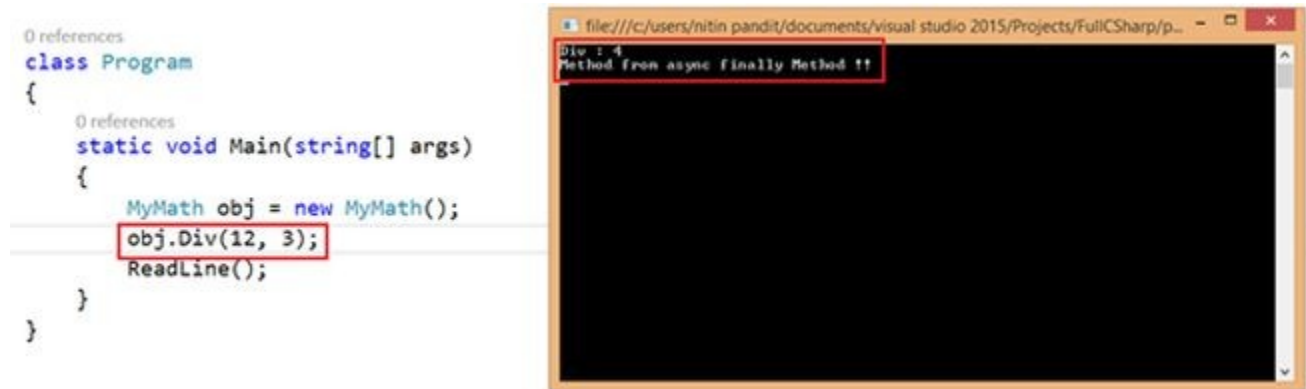
```

37.     {
38.         WriteLine("Method from async finally Method !!");
39.     }
40.
41.     private async Task asyncMethodForCatch()
42.     {
43.         WriteLine("Method from async Catch Method !!");
44.     }
45. }
46. }

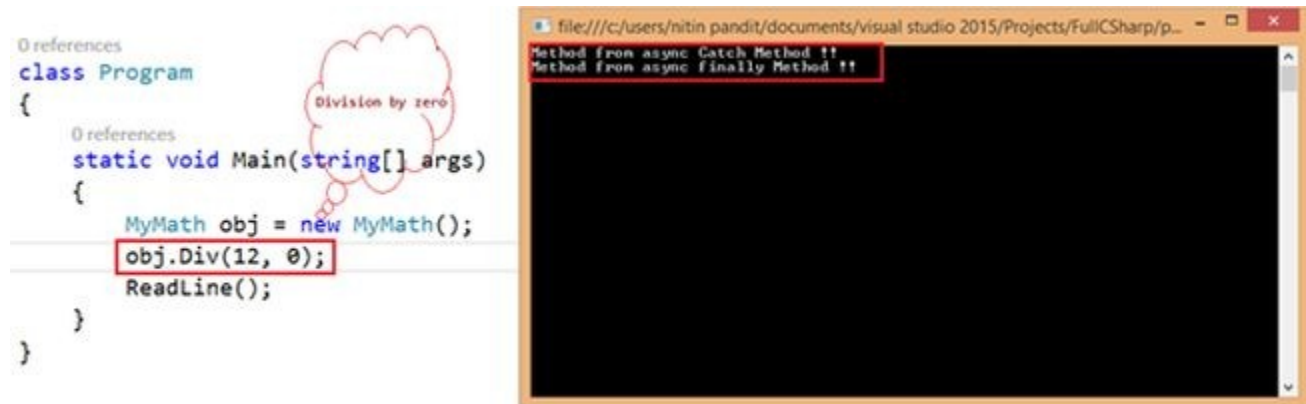
```

Output

If there is no exception then:



And when the exception occurs:



7. Null-Conditional Operator

The Null-Conditional operator is a new concept in C# 6.0 that is very beneficial for a developer in a source code file that we want to compare an object or a reference data type with **null**. So we need to write multiple lines of code to compare the objects in previous versions of C# 5.0 but in C# 6.0 we can write an **in-line null-conditional** with the **?** and **??** operators, so let's have an example and compare both versions, **C# 5.0** vs **C# 6.0**. We will write the code for both version.

Let's suppose we have two classes:

```
class Employee
{
    0 references
    public string Name { get; set; }
    0 references
    public Address EmployeeAddress { get; set; }
}
1 reference
class Address
{
    0 references
    public string HomeAddress { get; set; }
    0 references
    public string OfficeAddress { get; set; }
}
```

Now we need to write the code to compare the objects of the employee class with null in C# 5.0. So we need to write **if()** and **else** multiple lines.

```

class Program
{
    static void Main(string[] args)
    {
        Employee emp = new Employee();
        emp.Name = "Nitin Pandit";
        emp.EmployeeAddress = new Address()
        {
            HomeAddress = "Noida Sec 15",
            OfficeAddress = "Noida Sec 16"
        };
        if (emp != null && emp.EmployeeAddress != null)
        {
            Console.WriteLine(emp.Name + " " + emp.EmployeeAddress.HomeAddress + " " + emp.EmployeeAddress.OfficeAddress);
        }
        else
        {
            Console.WriteLine("No Address !!");
        }
        Console.ReadLine();
    }
}

```

Check first then write

If we want to write the same code in C# 6.0 then we can use ? and ?? to check the null value of an object, as in the following:

Condition ? code that use in case not null ?? in case of null

```

static void Main()
{
    Employee emp = new Employee();
    emp.Name = "Nitin Pandit";
    emp.EmployeeAddress = new Address()
    {
        HomeAddress = "Noida Sec 15",
        OfficeAddress = "Noida Sec 16"
    };
    WriteLine((emp?.Name) + " " + (emp?.EmployeeAddress?.HomeAddress ?? "No Address"));
    ReadLine();
}

```

For inline condition checking
condition ? if true ?? if false

if emp != null then it return value of name

it will check employeeAddress ref if not null then return true and print HomeAddress else "No Address"

if true if false

Output

The output when no object is null:

```

class Program
{
    0 references
    static void Main()
    {
        Employee emp = new Employee();
        emp.Name = "Nitin Pandit";
        emp.EmployeeAddress = new Address()
        {
            HomeAddress = "Noida Sec 15",
            OfficeAddress = "Noida Sec 16"
        };
        WriteLine((emp?.Name) + " " + (emp?.EmployeeAddress?.HomeAddress??"No Address"));
        ReadLine();
    }
}

```



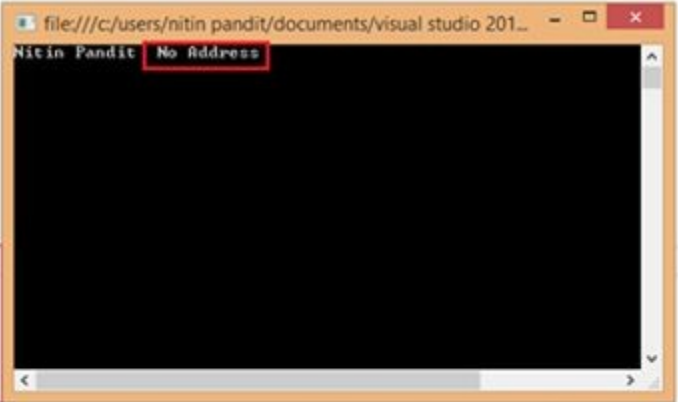
The screenshot shows a console window with the output "Nitin Pandit Noida Sec 15". The window title is "file:///c:/users/nitin pandit/documents/visual studi...".

The output when an object is null:

```

0 references
class Program
{
    0 references
    static void Main()
    {
        Employee emp = new Employee();
        emp.Name = "Nitin Pandit";
        //emp.EmployeeAddress = new Address()
        //{
        //    HomeAddress = "Noida Sec 15",
        //    OfficeAddress = "Noida Sec 16"
        //};
        WriteLine((emp?.Name) + " " + (emp?.EmployeeAddress?.HomeAddress??"No Address"));
        ReadLine();
    }
}

```



The screenshot shows a console window with the output "Nitin Pandit No Address". The window title is "file:///c:/users/nitin pandit/documents/visual studio 201...". A red box highlights the output "No Address". A red cloud-shaped annotation points to the commented-out code, stating "If EmployeeAddress is null".

Code

1. `using System;`
2. `using System.Collections.Generic;`
3. `using System.Linq;`
4. `using System.Text;`
5. `using System.Threading.Tasks;`
6. `using System.Console;`

7. **namespace** project7

```
8. {  
9.     class Program  
10. {  
11.     static void Main()  
12.     {  
13.         Employee emp = new Employee();  
14.         emp.Name = "Nitin Pandit";  
15.         emp.EmployeeAddress = new Address()  
16.         {  
17.             HomeAddress = "Noida Sec 15",  
18.             OfficeAddress = "Noida Sec 16"  
19.         };  
20.         WriteLine((emp?.Name) + " " + (emp?.EmployeeAddress?.HomeAddress??"No  
    Address"));  
21.         ReadLine();  
22.     }  
23. }  
24. class Employee  
25. {  
26.     public string Name { get; set; }  
27.     public Address EmployeeAddress { get; set; }  
28. }  
29. class Address  
30. {  
31.     public string HomeAddress { get; set; }  
32.     public string OfficeAddress { get; set; }
```

```
33.  }  
34. }
```

8. Expression–Bodied Methods

A Expression–Bodied Method is a very usefull way to write a function in a new way and also very usefull for those methods that can return their value by a single line so we can write those methods by using the “=>” lamda Operator in C# 6.0 so let's have an example.

We will just write a method in both versions of C# 5.0 vs C# 6.0 with a method that only returns a message of a string value.

In C# 5.0:

```
class Program  
{  
    static void Main(string[] args)  
    {  
        Console.WriteLine(GetTime());  
        Console.ReadLine();  
    }  
    public static string GetTime()  
    {  
        return "Current Time - " + DateTime.Now.ToString("hh:mm:ss");  
    }  
}
```

In C# 6.0

```

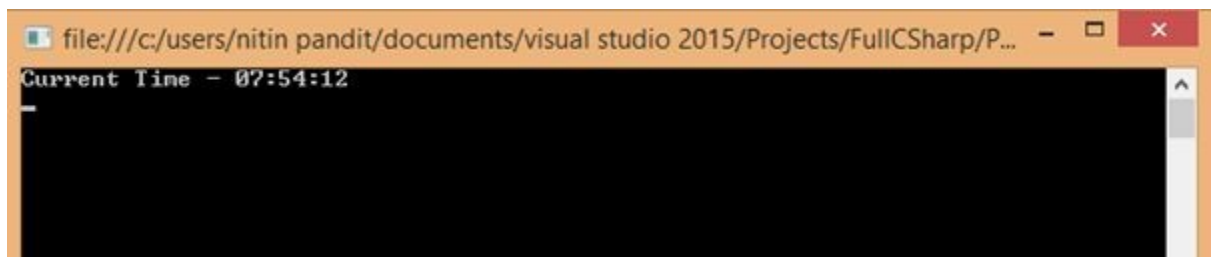
using System.Console;
namespace Project8
{
    0 references
    class Program
    {
        0 references
        static void Main(string[] args)
        {
            WriteLine(GetTime());
            ReadLine();
        }
        1 reference
        public static string GetTime()=> "Current Time - " + DateTime.Now.ToString("hh:mm:ss");
    }
}

```

=> using for body

Output

The output will be the same in both.



Code

1. **using** System;
2. **using** System.Collections.Generic;
3. **using** System.Linq;
4. **using** System.Text;
5. **using** System.Threading.Tasks;
6. **using** System.Console;
7. **namespace** Project8
8. {
9. **class** Program
10. {

```

11.     static void Main(string[] args)
12.     {
13.         WriteLine(GetTime());
14.         ReadLine();
15.     }

16.     public static string GetTime()=> "Current Time - " + DateTime.Now.ToString("h
    h:mm:ss");

17.
18. }
19. }

```

9. Easily format strings using String interpolation

To easily format a string value in **C# 6.0** without any **string.Format()** method we can write a format for a string. It's a very usefull and time consuming process to define multiple string values by "**{ variable }**". So let's have an example on **String interpolation**. First we are writing the source code by **string.Format()**.

```

class Program
{
    static void Main(string[] args)
    {
        string FirstName = "Nitin";
        string LastName = "Pandit";

        // With String.Format
        string output = string.Format("{0}- {1}", FirstName, LastName);
        Console.WriteLine(output);
        Console.ReadLine();
    }
}

```

Now by "{variable}".

```

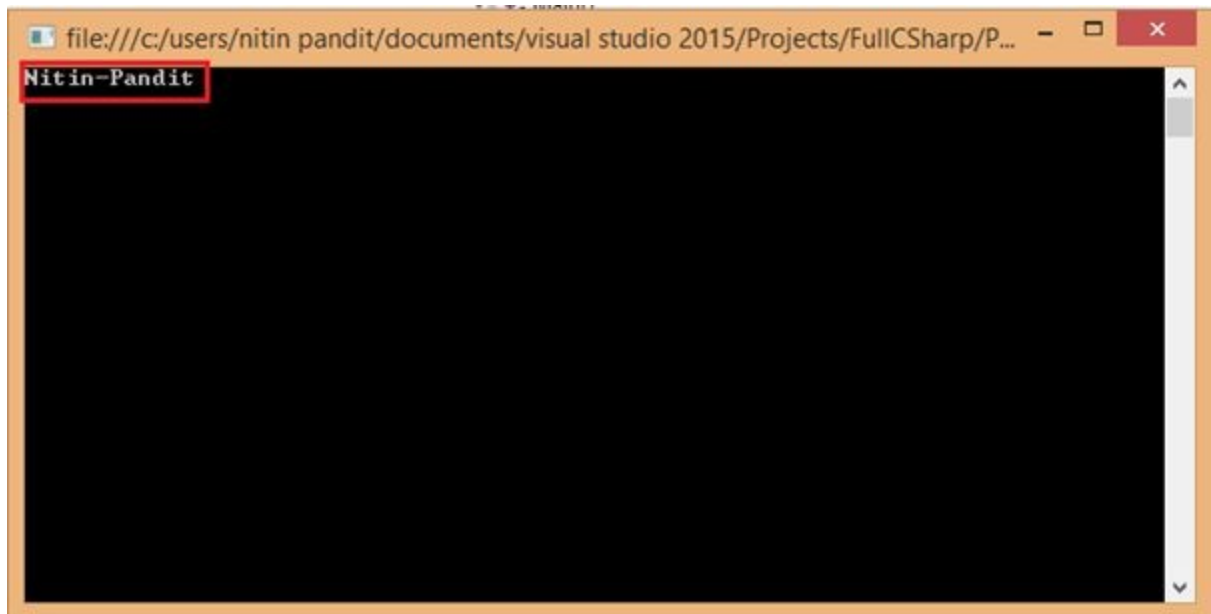
class Program
{
    0 references
    static void Main()
    {
        string FirstName = "Nitin";
        string LastName = "Pandit";

        // With String Interpolation in C# 6.0
        string output= $"{FirstName}-{LastName}";
        WriteLine(output);
        ReadLine();
    }
}

```

Output

The output will be the same in both but by “{**variable**}” is a very short way to write the same code.



Code

```
1. using System;
2. using System.Collections.Generic;
3. using System.Linq;
4. using System.Text;
5. using System.Threading.Tasks;
6. using System.Console;
7. namespace Project9
8. {
9.     class Program
10.    {
11.        static void Main()
12.        {
13.            string FirstName = "Nitin";
14.            string LastName = "Pandit";
15.
16.            // With String Interpolation in C# 6.0
17.            string output= $"{FirstName}-{LastName}";
18.            WriteLine(output);
19.            ReadLine();
20.        }
21.    }
22. }
```

Thank you for reading this article.

<http://www.c-sharpcorner.com/UploadFile/8ef97c/full-C-Sharp-6-0-in-single-article-on-visual-studio-2015-preview/>

List of All New Features in C# 6.0: Part 2



-

- [Nitin Pandit](#)

- Jan 13 2015

- [Article](#)

-

- [10](#)

- [36](#)

- 38.5k

-

-

- [-](#)

-

- [-](#)

-

- [-](#)

-

- [-](#)

-

- [-](#)

-

-

-

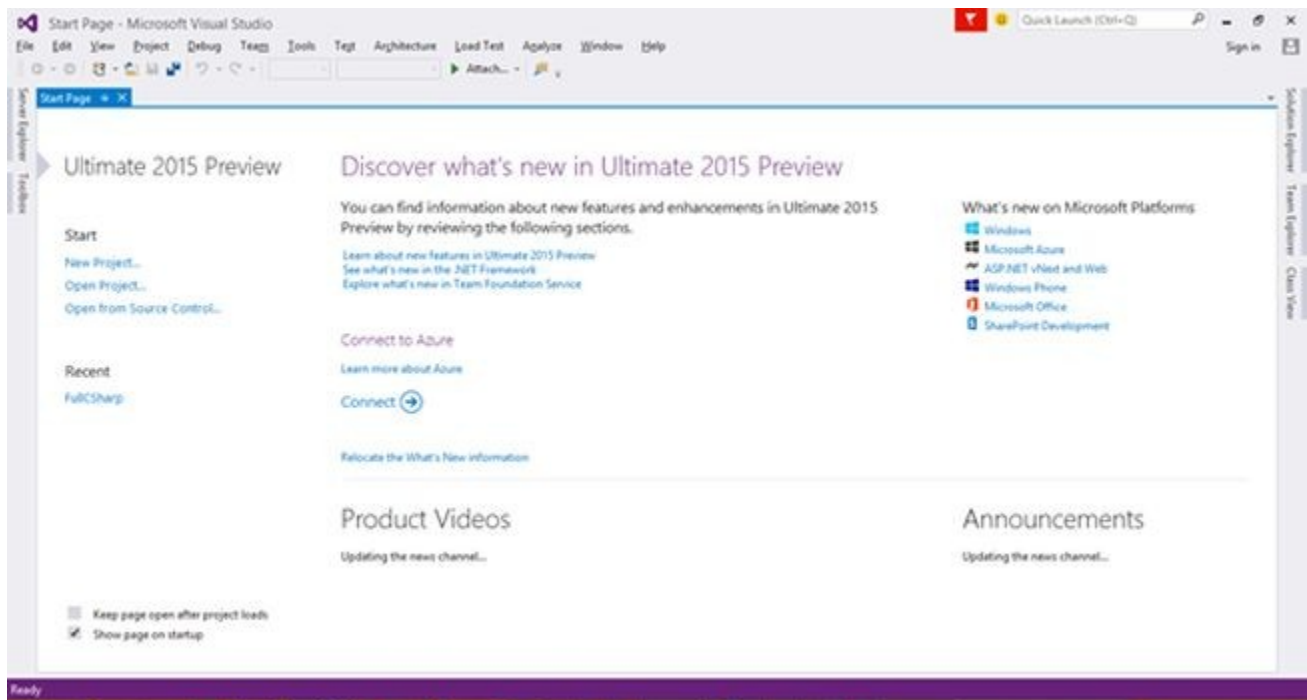


[NewCSharpPart2.zip](#)

[Download 100% FREE Spire Office APIs](#)

Hi guys. I am back with two new features of C# 6.0 in Visual Studio 2015 Preview. Before reading this article, I highly recommend reading the previous part:

- [List of All New Features in C# 6.0: Part 1](#)



This article explains parameterless constructors in structs and how it is now easier to create a custom exception class by shortcuts in Visual Studio 2015 and we have many new features in C# 6.0 that have already been posted in my last article. For more you can go through with that.

List of All New Features in C# 6.0

1. Parameterless constructors in structs.
2. Creating a custom exceptions class.

Parameterless constructors in Structs

Parameterless constructors in structs in C# 6.0 is now possible, we can create a parameterless

constructor in a struct explicitly in Visual Studio 2015. Let's see a property of structs for C# 5.0 and prior to understand structs first.

Structs Overview

Structs have the following properties:

- Structs are value types whereas classes are reference types.
- Unlike classes, structs can be instantiated without using a new operator.
- Structs can declare constructors, but they must take parameters until C# 5.0.
- A struct cannot inherit from another struct or class and it cannot be the base of a class. All structs inherit directly from **System.ValueType**, that inherits from **System.Object**.
- A struct can implement interfaces.

We use structs for small code, or in other words we can say we can write a structs definition anywhere where we have a class only for a few properties or methods so there is no need to use the memory in the heap, we can use stack memory by using structs and the object of structs can call the default constructor. That means that with parameterless constructors there is no need to use a **new** keyword because until C# 5.0 we are unable to create a parameterless constructor in structs like **Int32** is a struct and we can use that without using the **new** keyword but in the case of a class we must use new so that we can create a memory block in the heap and at the time of new we must call the constructor of the class without a parameter or with a parameter explicitly.


Structs have a default constructor (parameterless constructor) and there is no need to call that with the new keyword because structs are value types so we can call their constructors implicitly. Let's have an example in C# 5.0.

I just wrote a struct of Student Type to store the records and also tried to write a default or parameterless constructor but at compile time it will generate an error.

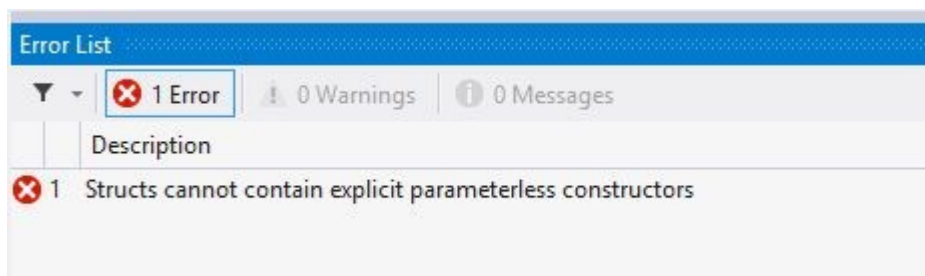
```

public struct Stu
{
    public Stu()
    {
    }
    int rollno;
    public int Rollno
    {
        get { return rollno; }
        set { rollno = value; }
    }
    string name;
    public string Name
    {
        get { return name; }
        set { name = value; }
    }
}

```



I just wrote a struct of Student Type to store the records and also attempted to write a default or parameterless constructor but at compile time it will generate an error.

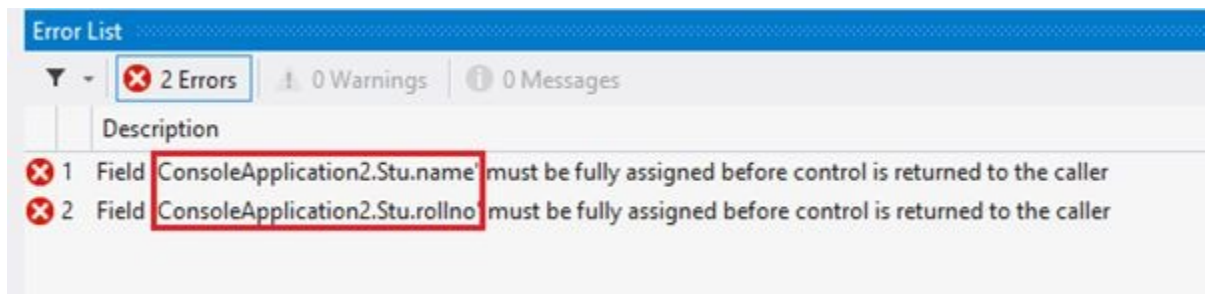


Structs can't contain explicit Parameterless Constructors because the object of a structs type is a **Value Type** stored in stack memory. That's why there is no need to call a parameterless constructor with the **new** keyword explicitly. So let's see, when I write a constructor with parameters it's again returning an error. Let's see that.

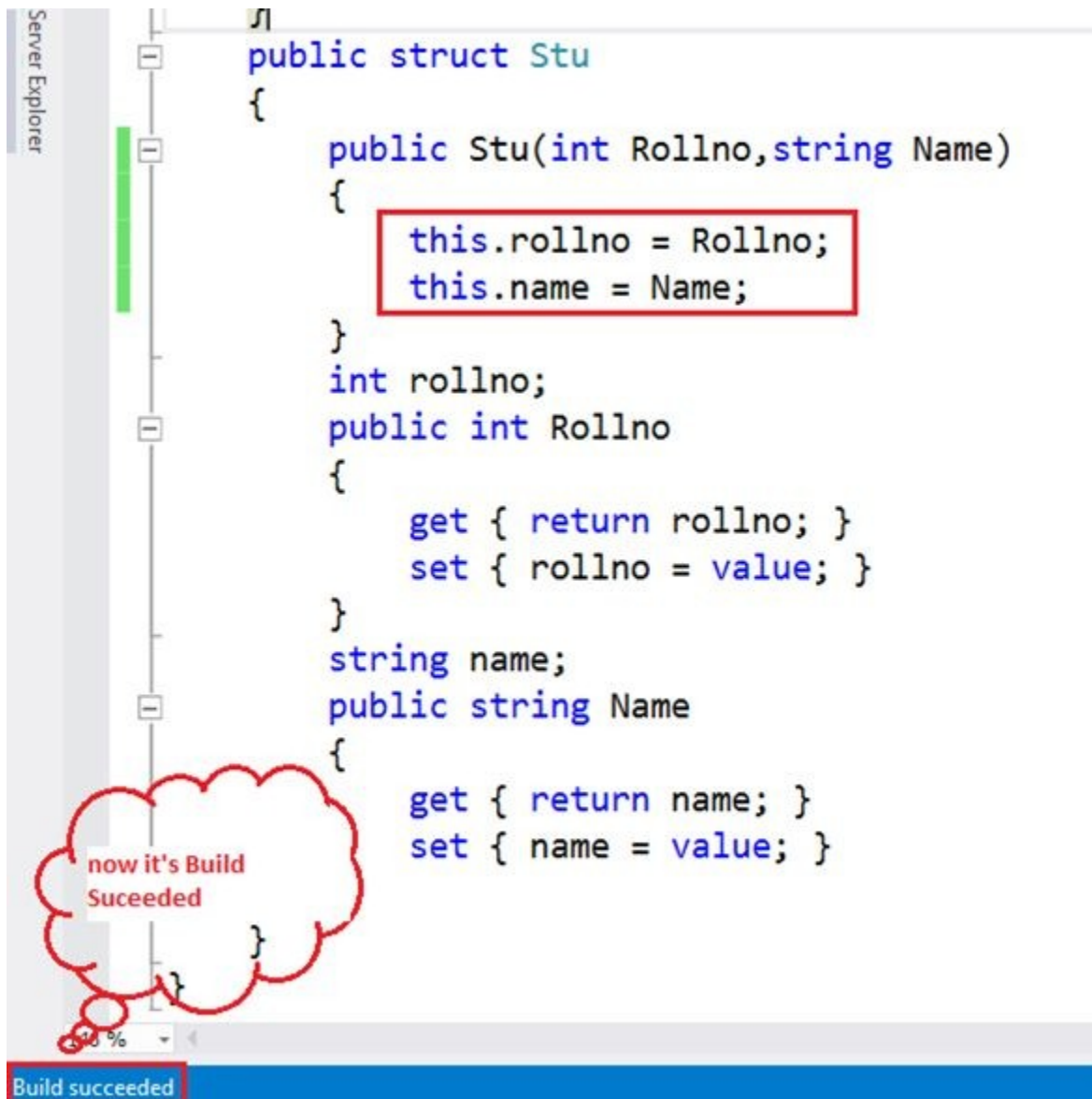
```

public struct Stu
{
    public Stu(int Rollno, string Name)
    {
    }
    int rollno;
    public int Rollno
    {
        get { return rollno; }
        set { rollno = value; }
    }
    string name;
    public string Name
    {
        get { return name; }
        set { name = value; }
    }
}

```



In that example code that you have already see above we have still 2 errors because we don't assign the default value for all those Data Members. Those are declared in my struct so it's necessary to assign all the data members in my constructor of structs and then my source code will be built successfully.



```
public struct Stu
{
    public Stu(int Rollno, string Name)
    {
        this.rollno = Rollno;
        this.name = Name;
    }
    int rollno;
    public int Rollno
    {
        get { return rollno; }
        set { rollno = value; }
    }
    string name;
    public string Name
    {
        get { return name; }
        set { name = value; }
    }
}
```

now it's Build Succeeded

Build succeeded

Code

1. **public struct** Stu
2. {
3. **public** Stu(**int** Rollno,**string** Name)
4. {
5. **this**.rollno = Rollno;
6. **this**.name = Name;
7. }

```

8.  int rollno;

9.  public int Rollno

10. {

11.     get { return rollno; }

12.     set { rollno = value; }

13. }

14. string name;

15. public string Name

16. {

17.     get { return name; }

18.     set { name = value; }

19. }

20. }


```

So I think it's enough to understand how to use a parameter constructor in C# 5.0 of structs. Let's see how to use all the properties of structs by the object of **structs** without using the new keyword because it's able to call their parameterless constructor that is declared by the compiler implicitly we can't write that manually in C# 5.0.

```

class Program
{
    static Stu obj;
    static void Main(string[] args)
    {
        obj.Rollno = 101;
        obj.Name = "Nitin Pandit";
        Console.WriteLine("Roll No : {0} , Name : {1} ", obj.Rollno, obj.Name);
        Console.ReadLine();
    }
}

```



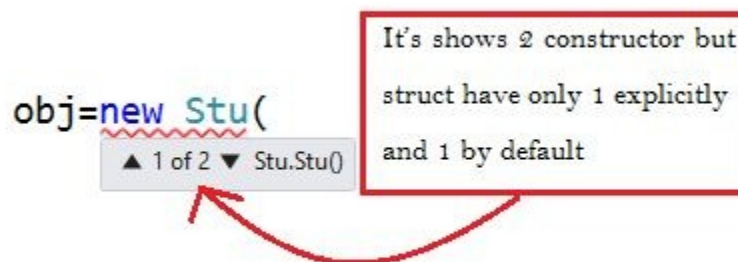
we can use the object of struct without new keyword

See the output. When we are using a property by an object without assigning the object with the new keyword we call the parameterless constructor.

Output

```
file:///F:/MyClassCode/ConsoleApplication2/Con
Roll No : 101 , Name : Nitin Pandit
```

Now we need to assign the object with the new keyword so it shows 2 constructors but a struct has only 1 explicitly and 1 by default created by the compiler. See the image below.



So you can use any of them, if we are not assigned the object the parameterless constructor will be called implicitly.

In C# 6.0

In C# 6.0 with Visual Studio 2015 we are able to create a parameterless constructor in structs too. In previous versions of C# we are not able to create a parameterless constructor so all the value type data members must be assigned with their default values by the parameterless constructor created by the compiler. For example if we have a variable of `Int32` Type then to that will be assigned 0 by default but now you can assign any default value to every data member of your structs like a class but still we have a condition such as if we are going to create a parameterless constructor in my struct so every member must be assigned in to that constructor but that condition is not applicable for class types so now enjoy the new behaviour of structs with a parameterless constructor. Let's see the example:

```

public struct Stu
{
    0 references
    public Stu()
    {
        rollno = 0;
        name = string.Empty;
    }
    int rollno;
    0 references
    public int Rollno
    {
        get { return rollno; }
        set { rollno = value; }
    }
    string name;
    0 references
    public string Name
    {
        get { return name; }
        set { name = value; }
    }
}


```

That code gets Build successful because we assigned all data members of the struct but if we assigned only one and one or more and the rest remains to be assigned then that will generate an error as in the following:

```

public Stu()
{
    rollno
}

```



So assign all the data members in your default or parameterless constructor and use that without the new keyword.

Creating custom exceptions Class

Creating a Custom Exception class is a new benefit of Visual Studio 2015. In Visual Studio 2012 or

in any Visual Studio if we need to handle my exception by a catch block by a custom exception class so we need to write a class file first and then we must inherit a System.Exception class and for using the Message property to be overridden and define the custom Exception Message as in the following:

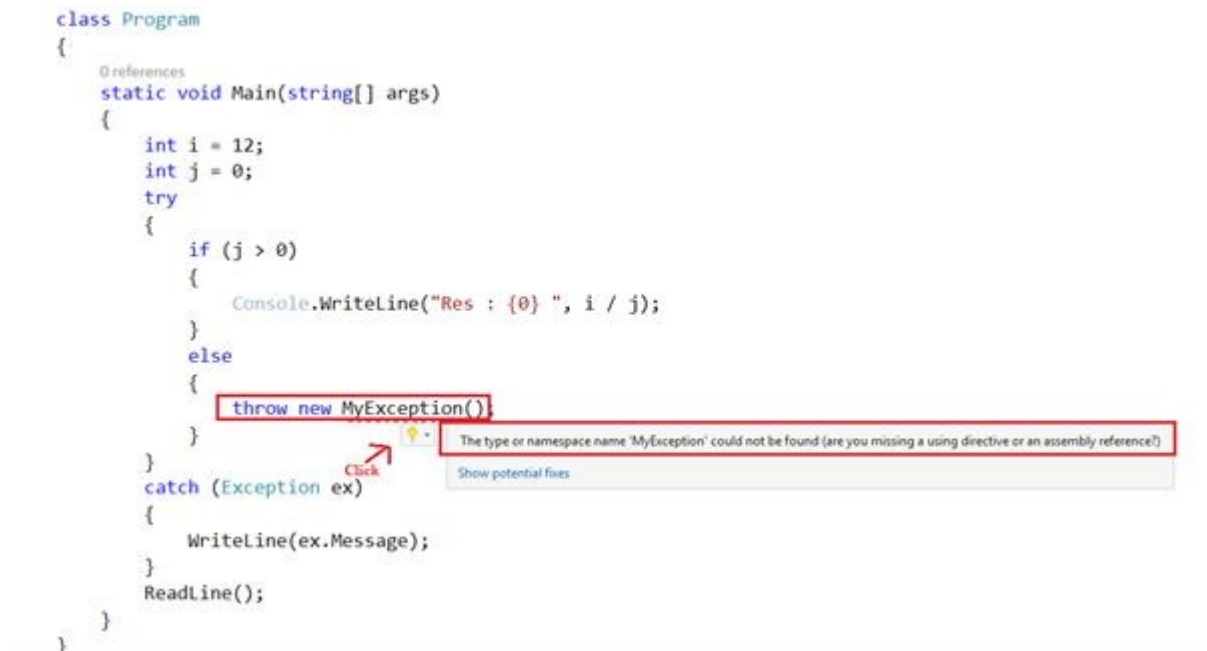
```
public class MyException: Exception
{
    public override string Message
    {
        get
        {
            return "Exception by My Class !!";
        }
    }
}
```

And we can use that class as an **Exception** class by the **throw** keyword and handle in a **catch** block directly in a **System.Exception** type object.

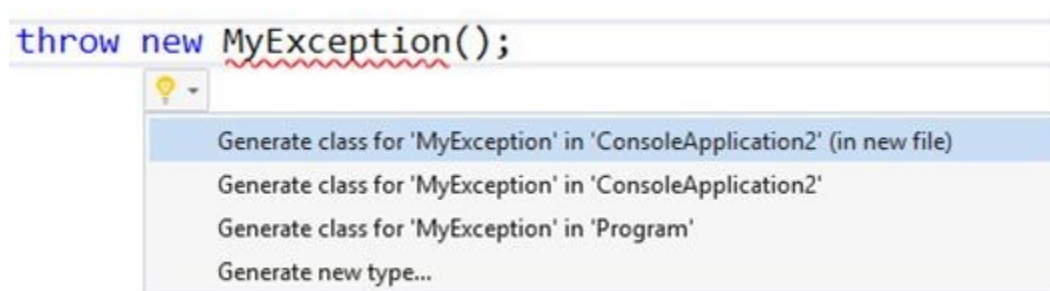

```
public static void Main()
{
    int i = 12;
    int j = 0;
    try
    {
        if (j > 0)
        {
            Console.WriteLine("Res : {0} ", i / j);
        }
        else
        {
            throw new MyException();
        }
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex.Message);
    }
    Console.ReadLine();
}
```

But in Visual Studio 2015 it's too easy to create a custom exception class


In Visual Studio 2015 it's too easy to create that custom exception class. There is no need to create that class manually, we need to use the **throw** keyword and create an object by the new keyword.



When you click then it will show you all the maximum options.



And also it shows the preview of that code that can be written by Visual Studio.

 **CS0246** The type or namespace name 'MyException' could not be found (are you missing a using directive or an assembly reference?)

Adding 'MyException' to 'ConsoleApplication2' with content:

```
using System;
using System.Runtime.Serialization;

namespace ConsoleApplication2
{
    [Serializable]
    internal class MyException : Exception
    {
        public MyException()
        {
        }

        public MyException(string message) : base(message)
        {
        }

        public MyException(string message, Exception innerException) : base(message, innerException)
        {
        }

        protected MyException(SerializationInfo info, StreamingContext context) : base(info, context)
        {
        }
    }
}
```

Preview changes

Now just press Enter and that will create a serializable Exception Class in a new file like the following:

```

using System;
using System.Runtime.Serialization;

namespace ConsoleApplication2
{
    [Serializable]
    5 references
    internal class MyException : Exception
    {
        1 reference
        public MyException()
        {
        }

        0 references
        public MyException(string message) : base(message)
        {
        }

        0 references
        public MyException(string message, Exception innerException) : base(message, innerException)
        {
        }

        0 references
        protected MyException(SerializationInfo info, StreamingContext context) : base(info, context)
        {
        }
    }
}

```

Now override any property in the **System.Exception** class and use them as a normal Exception class.

I hope you enjoy this article.

Thanks.

<http://www.c-sharpcorner.com/UploadFile/8ef97c/list-of-all-new-features-in-C-Sharp-6-0-part-2/>

What's new in .NET 2015

.NET 2015 introduces the .NET Framework 4.6 and .NET Core. Some new features apply to both, and other features are specific to .NET Framework 4.6 or .NET Core.

- **ASP.NET 5**

.NET 2015 includes ASP.NET 5, which is a lean .NET platform for building modern cloud-based apps. The platform is modular so you can include only those features that are needed in your application. It can be hosted on IIS or self-hosted in a custom process, and you can run apps with different versions of the .NET Framework on the same server. It includes a new environment configuration system that is designed for cloud deployment.

MVC, Web API, and Web Pages are unified into a single framework called MVC 6. You build ASP.NET 5 apps through the new tools in Visual Studio 2015. Your existing applications will work on the new .NET Framework; however to build an app that uses MVC 6 or SignalR 3, you must use the project system in Visual Studio 2015.

For information, see [ASP.NET 5](#).

- **ASP.NET Updates**

- **Task-based API for Asynchronous Response Flushing**

ASP.NET now provides a simple task-based API for asynchronous response flushing, [HttpResponse.FlushAsync](#), that allows responses to be flushed asynchronously by using your language's `async/await` support.

- Model binding supports task-returning methods

In the .NET Framework 4.5, ASP.NET added the Model Binding feature that enabled an extensible, code-focused approach to CRUD-based data operations in Web Forms pages and user controls. The Model Binding system now supports [Task](#)-returning model binding methods. This feature allows Web Forms developers to get the scalability benefits of `async` with the ease of the data-binding system when using newer versions of ORMs, including the Entity Framework.

Async model binding is controlled by the `aspnet:EnableAsyncModelBinding` configuration setting.

```
<appSettings>
  <add key=" aspnet:EnableAsyncModelBinding" value="true|false" />
</appSettings>
```

On apps that target the .NET Framework 4.6, it defaults to `true`. On apps running on the .NET Framework 4.6 that target an earlier version of the .NET Framework, it is `false` by default. It can be enabled by setting the configuration setting to `true`.

- **HTTP/2 Support (Windows 10)**

[HTTP/2](#) is a new version of the HTTP protocol that provides much better connection utilization (fewer round-trips between client and server), resulting in lower latency web page loading for users. Web pages (as opposed to services) benefit the most from HTTP/2, since the protocol optimizes for multiple artifacts being requested as part of a single experience. HTTP/2

support has been added to ASP.NET in the .NET Framework 4.6. Because networking functionality exists at multiple layers, new features were required in Windows, in IIS, and in ASP.NET to enable HTTP/2. You must be running on Windows 10 to use HTTP/2 with ASP.NET.

HTTP/2 is also supported and on by default for Windows 10 Universal Windows Platform (UWP) apps that use the [System.Net.Http.HttpClient](#) API.

In order to provide a way to use the [PUSH_PROMISE](#) feature in ASP.NET applications, a new method with two overloads, [PushPromise\(String\)](#) and [PushPromise\(String, String, NameValueCollection\)](#), has been added to the [HttpResponse](#) class.

Note

While ASP.NET 5 supports HTTP/2, support for the PUSH PROMISE feature has not yet been added.

The browser and the web server (IIS on Windows) do all the work. You don't have to do any heavy-lifting for your users.

Most of the [major browsers support HTTP/2](#), so it's likely that your users will benefit from HTTP/2 support if your server supports it.

- **Support for the Token Binding Protocol**

Microsoft and Google have been collaborating on a new approach to authentication, called the [Token Binding Protocol](#). The premise is that authentication tokens (in your browser cache) can be stolen and used by criminals to access otherwise secure resources (e.g. your bank account) without requiring your password or any other privileged knowledge. The new protocol aims to mitigate this problem.

The Token Binding Protocol will be implemented in Windows 10 as a browser feature. ASP.NET apps will participate in the protocol, so that authentication tokens are validated to be legitimate. The client and the server implementations establish the end-to-end protection specified by the protocol.

- **Randomized string hash algorithms**

The .NET Framework 4.5 introduced a [randomized string hash algorithm](#). However, it was not supported by ASP.NET because of some ASP.NET features depended on a stable hash code. In .NET Framework 4.6, randomized string hash algorithms are now supported. To enable this feature, use the `aspnet:UseRandomizedStringHashAlgorithm` config setting.

```
<appSettings>
  <add key="aspnet:UseRandomizedStringHashAlgorithm" value="true|false" />
</appSettings>
```

- **ADO.NET**

ADO .NET now supports the Always Encrypted feature available in SQL Server 2016 Community Technology Preview 2 (CTP2). With Always Encrypted, SQL Server can perform operations on encrypted data, and best of all the encryption key resides with the application inside the customer's trusted environment and not on the server. Always Encrypted secures customer data so DBAs do not have access to plain text data. Encryption and decryption of data happens transparently at the driver level, minimizing changes that have to be made to existing applications. For details, see [Always Encrypted \(Database Engine\)](#) and [Always Encrypted \(client development\)](#).

- **64-bit JIT Compiler for managed code**

The .NET Framework 4.6 features a new version of the 64-bit JIT compiler (originally code-named RyuJIT). The new 64-bit compiler provides significant performance improvements over the older 64-bit JIT compiler. The new 64-bit compiler is enabled for 64-bit processes running on top of the .NET Framework 4.6. Your app will run in a 64-bit process if it is compiled as 64-bit or AnyCPU and is running on a 64-bit operating system. While care has been taken to make the transition to the new compiler as transparent as possible, changes in behavior are possible. We would like to hear directly about any issues encountered when using the new JIT compiler. Please contact us through [Microsoft Connect](#) if you encounter an issue that may be related to the new 64-bit JIT compiler.

The new 64-bit JIT compiler also includes hardware SIMD acceleration features when coupled with SIMD-enabled types in the [System.Numerics](#) namespace, which can yield good performance improvements.

- **Assembly loader improvements**

The assembly loader now uses memory more efficiently by unloading IL assemblies after a corresponding NGEN image is loaded. This change decreases virtual memory, which is particularly beneficial for large 32-bit apps (such as Visual Studio), and also saves physical memory.

- **Base class library changes**

Many new APIs have been added around to .NET Framework 4.6 to enable key scenarios. These include the following changes and additions:

- **`ICollection<T>` implementations**

Additional collections implement [ICollection<T>](#) such as [Queue<T>](#) and [Stack<T>](#).

- **`CultureInfo.CurrentCulture` and `CultureInfo.CurrentUICulture`**

The [CultureInfo.CurrentCulture](#) and [CultureInfo.CurrentUICulture](#) properties are now read-write rather than read-only. If you assign a new [CultureInfo](#) object to these properties, the current thread culture defined by the `Thread.CurrentThread.CurrentCulture` property and the current UI thread culture defined by the `Thread.CurrentThread.CurrentUICulture` properties also change.

- **Enhancements to garbage collection (GC)**

The [GC](#) class now includes [TryStartNoGCRegion](#) and [EndNoGCRegion](#) methods that allow you to disallow garbage collection during the execution of a critical path.

A new overload of the [GC.Collect\(Int32, GCCollectionMode, Boolean, Boolean\)](#) method allows you to control whether both the small object heap and the large object heap are swept and compacted or swept only.

- **SIMD-enabled types**

The [System.Numerics](#) namespace now includes a number of SIMD-enabled types, such as [Matrix3x2](#), [Matrix4x4](#), [Plane](#), [Quaternion](#), [Vector2](#), [Vector3](#), and [Vector4](#).

Because the new 64-bit JIT compiler also includes hardware SIMD acceleration features, there are especially significant performance improvements when using the SIMD-enabled types with the new 64-bit JIT compiler.

- **Cryptography updates**

The [System.Security.Cryptography](#) API is being updated to support the [Windows CNG cryptography APIs](#). Previous versions of the .NET Framework have relied entirely on an [earlier version of the Windows Cryptography APIs](#) as the basis for the [System.Security.Cryptography](#) implementation. We have had requests to support the CNG API, since it supports [modern cryptography algorithms](#), which are important for certain categories of apps.

The .NET Framework 4.6 includes the following new enhancements to support the Windows CNG cryptography APIs:

- A set of extension methods for X509 Certificates, `System.Security.Cryptography.X509Certificates.RSACertificateExtensions.GetRSAPublicKey(System.Security.Cryptography.X509Certificates.X509Certificate2)` and `System.Security.Cryptography.X509Certificates.RSACertificateExtensions.GetRSAPrivateKey(System.Security.Cryptography.X509Certificates.X509Certificate2)`, that return a CNG-based implementation rather than a CAPI-based implementation when possible. (Some smartcards, etc., still require CAPI, and the APIs handle the fallback).
- The [System.Security.Cryptography.RSACng](#) class, which provides a CNG implementation of the RSA algorithm.
- Enhancements to the RSA API so that common actions no longer require casting. For example, encrypting data using an [X509Certificate2](#) object requires code like the following in previous versions of the .NET Framework.

C#

VB

```
RSACryptoServiceProvider rsa =  
(RSACryptoServiceProvider)cert.PrivateKey;  
byte[] oaepEncrypted = rsa.Encrypt(data, true);  
byte[] pkcs1Encrypted = rsa.Encrypt(data,  
false);
```

Code that uses the new cryptography APIs in the .NET Framework 4.6 can be rewritten as follows to avoid the cast.

C#

VB

```
RSA rsa = cert.GetRSAPrivateKey();  
if (rsa == null)  
    throw new InvalidOperationException("An RSA  
certificate was expected");  
  
byte[] oaepEncrypted = rsa.Encrypt(data,  
RSAEncryptionPadding.OaepSHA1);  
byte[] pkcs1Encrypted = rsa.Encrypt(data,  
RSAEncryptionPadding.Pkcs1);
```

- **Support for converting dates and times to or from Unix time**

The following new methods have been added to the [DateTimeOffset](#) structure to support converting date and time values to or from Unix time:

- [DateTimeOffset.FromUnixTimeSeconds](#)
- [DateTimeOffset.FromUnixTimeMilliseconds](#)
- [DateTimeOffset.ToUnixTimeSeconds](#)
- [DateTimeOffset.ToUnixTimeMilliseconds](#)

○ **Compatibility switches**

The new [AppContext](#) class adds a new compatibility feature that enables library writers to provide a uniform opt-out mechanism for new functionality for their users. It establishes a loosely-coupled contract between components in order to communicate an opt-out request. This capability is typically important when a change is made to existing functionality. Conversely, there is already an implicit opt-in for new functionality.

With [AppContext](#), libraries define and expose compatibility switches, while code that depends on them can set those switches to affect the library behavior. By default, libraries provide the new functionality, and they only alter it (that is, they provide the previous functionality) if the switch is set.

An application (or a library) can declare the value of a switch (which is always a [Boolean](#) value) that a dependent library defines. The switch is always implicitly `false`. Setting the switch to `true` enables it. Explicitly setting the switch to `false` provides the new behavior.

C#

```
AppContext.SetSwitch("Switch.AmazingLib.ThrowOnException", true);
```

The library must check if a consumer has declared the value of the switch and then appropriately act on it.

```
if (!AppContext.TryGetSwitch("Switch.AmazingLib.ThrowOnException",  
    out shouldThrow))  
{  
    // This is the case where the switch value was not set by the  
    application.  
    // The library can choose to get the value of shouldThrow by  
    other means.  
    // If no overrides nor default values are specified, the value  
    should be 'false'.  
    // A false value implies the latest behavior.
```

```

}

// The library can use the value of shouldThrow to throw
exceptions or not.
if (shouldThrow)
{
    // old code
}
else {
    // new code
}
}

```

It's beneficial to use a consistent format for switches, since they are a formal contract exposed by a library. The following are two obvious formats.

- *Switch.namespace.switchname*
- *Switch.library.switchname*

○ **Changes to the task-based asynchronous pattern (TAP)**

For apps that target the .NET Framework 4.6, [Task](#) and [Task<TResult>](#) objects inherit the culture and UI culture of the calling thread. The behavior of apps that target previous versions of the .NET Framework, or that do not target a specific version of the .NET Framework, is unaffected. For more information, see the "Culture and task-based asynchronous operations" section of the [CultureInfo](#) class topic.

The [System.Threading.AsyncLocal<T>](#) class allows you to represent ambient data that is local to a given asynchronous control flow, such as an `async` method. It can be used to persist data across threads. You can also define a callback method that is notified whenever the ambient data changes either because the [AsyncLocal<T>.Value](#) property was explicitly changed, or because the thread encountered a context transition.

Three convenience methods, [Task.CompletedTask](#), [Task.FromCanceled](#), and [Task.FromException](#), have been added to the task-based asynchronous pattern (TAP) to return completed tasks in a particular state.

The [NamedPipeClientStream](#) class now supports asynchronous communication with its new [ConnectAsync](#) method.

○ **EventSource now supports writing to the Event log**

You now can use the [EventSource](#) class to log administrative or operational messages to the event log, in addition to any existing ETW sessions created

on the machine. In the past, you had to use the `Microsoft.Diagnostics.Tracing.EventSource` NuGet package for this functionality. This functionality is now built-into the .NET Framework 4.6.

Both the NuGet package and the .NET Framework 4.6 have been updated with the following features:

- **Dynamic events**

Allows events defined "on the fly" without creating event methods.

- **Rich payloads**

Allows specially attributed classes and arrays as well as primitive types to be passed as a payload

- **Activity tracking**

Causes Start and Stop events to tag events between them with an ID that represents all currently active activities.

To support these features, the overloaded [Write](#) method has been added to the [EventSource](#) class.

- **Windows Presentation Foundation (WPF)**

- **HDPI improvements**

HDPI support in WPF is now better in the .NET Framework 4.6. Changes have been made to layout rounding to reduce instances of clipping in controls with borders. By default, this feature is enabled only if your [TargetFrameworkAttribute](#) is set to .NET 4.6. Applications that target earlier versions of the framework but are running on the .NET Framework 4.6 can opt in to the new behavior by adding the following line to the [<runtime>](#) section of the app.config file:

```
<AppContextSwitchOverrides
value="Switch.MS.Internal.DoNotApplyLayoutRoundingToMarginsAndBorderThickness=false"
/>
```

WPF windows straddling multiple monitors with different DPI settings (Multi-DPI setup) are now completely rendered without blacked-out regions. You can opt out of this behavior by adding the following line to the `<appSettings>` section of the app.config file to disable this new behavior:

```
<add key="EnableMultiMonitorDisplayClipping" value="true"/>
```

Support for automatically loading the right cursor based on DPI setting has been added to [System.Windows.Input.Cursor](#).

- **Touch is better**

Customer reports on [Connect](#) that touch produces unpredictable behavior have been addressed in the .NET Framework 4.6. The double tap threshold for Windows Store applications and WPF applications is now the same in Windows 8.1 and above.

- **Transparent child window support**

WPF in the .NET Framework 4.6 supports transparent child windows in Windows 8.1 and above. This allows you to create non-rectangular and transparent child windows in your top-level windows. You can enable this feature by setting the [HwndSourceParameters.UsesPerPixelTransparency](#) property to `true`.

- **Windows Communication Foundation (WCF)**

- **SSL support**

WCF now supports SSL version TLS 1.1 and TLS 1.2, in addition to SSL 3.0 and TLS 1.0, when using NetTcp with transport security and client authentication. It is now possible to select which protocol to use, or to disable old lesser secure protocols. This can be done either by setting the [SslProtocols](#) property or by adding the following to a configuration file.

```
<netTcpBinding>
  <binding>
    <security mode= "None|Transport|Message|
TransportWithMessageCredential" >
      <transport clientCredentialType="None|Windows|
Certificate"
                                protectionLevel="None|Sign|EncryptAndSign"
                                sslProtocols="Ssl3|Tls1|Tls11|Tls12">
      </transport>
    </security>
  </binding>
</netTcpBinding>
```

- **Sending messages using different HTTP connections**

WCF now allows users to ensure certain messages are sent using different underlying HTTP connections. There are two ways to do this:

- **Using a connection group name prefix**

Users can specify a string that WCF will use as a prefix achieve for the connection group name. Two messages with different prefixes are sent using different underlying HTTP connections. You set the prefix by adding a key/value pair to the message's [Message.Properties](#) property. The key is "HttpTransportConnectionGroupNamePrefix"; the value is the desired prefix.

- **Using different channel factories**

Users can also enable a feature that ensures that messages sent using channels created by different channel factories will use different underlying HTTP connections. To enable this feature, users must set the following `appSetting` to `true`:

```
<appSettings>
  <add
    key="wcf:httpTransportBinding:useUniqueConnectionPoolPerFactory" value="true" />
</appSettings>
```

- **Windows Workflow Foundation (WWF)**

You can now specify the number of seconds a workflow service will hold on to an out-of-order operation request when there is an outstanding “non-protocol” bookmark before timing out the request. A “non-protocol” bookmark is a bookmark that is not related to outstanding Receive activities. Some activities create non-protocol bookmarks within their implementation, so it may not be obvious that a non-protocol bookmark exists. These include State and Pick. So if you have a workflow service implemented with a state machine or containing a Pick activity, you will most likely have non-protocol bookmarks. You specify the interval by adding a line like the following to the `appSettings` section of your `app.config` file:

```
<add key="microsoft:WorkflowServices:FilterResumeTimeoutInSeconds"
value="60"/>
```

The default value is 60 seconds. If `value` is set to 0, out-of-order requests are immediately rejected with a fault with text that looks like this:

Output

Operation 'Request3|{http://tempuri.org/}IService' on service instance with identifier '2b0667b6-09c8-4093-9d02-f6c67d534292' cannot be performed at this time. Please ensure that the operations are performed in the correct order and that the binding in use provides ordered delivery guarantees.

This is the same message that you receive if an out-of-order operation message is received and there are no non-protocol bookmarks.

If the value of the `FilterResumeTimeoutInSeconds` element is non-zero, there are non-protocol bookmarks, and the timeout interval expires, the operation fails with a timeout message.

- **Transactions**

You can now include the distributed transaction identifier for the transaction that has caused an exception derived from [TransactionException](#) to be thrown. You do this by adding the following key to the `appSettings` section of your `app.config` file:

```
<add
key="Transactions:IncludeDistributedTransactionIdInExceptionMessage"
value="true"/>
```

The default value is `false`.

- **Networking**

- **Socket reuse**

Windows 10 includes a new high-scalability networking algorithm that makes better use of machine resources by reusing local ports for outbound TCP connections. The .NET Framework 4.6 supports the new algorithm, enabling .NET apps to take advantage of the new behavior. In previous versions of Windows, there was an artificial concurrent connection limit (typically 16,384, the default size of the dynamic port range), which could limit the scalability of a service by causing port exhaustion when under load.

In the .NET Framework 4.6, two new APIs have been added to enable port reuse, which effectively removes the 64K limit on concurrent connections:

- The [SocketOptionName.ReuseUnicastPort](#) enumeration value.
- The [ServicePointManager.ReusePort](#) property.

By default, the [ServicePointManager.ReusePort](#) property is `false` unless the `HWRPortReuseOnSocketBind` value of the `HKLM\SOFTWARE\Microsoft\.NETFramework\v4.0.30319` registry key is

set to 0x1. To enable local port reuse on HTTP connections, set the [ServicePointManager.ReusePort](#) property to `true`. This causes all outgoing TCP socket connections from [HttpClient](#) and [HttpWebRequest](#) to use a new Windows 10 socket option, [SO_REUSE_UNICASTPORT](#), that enables local port reuse.

Developers writing a sockets-only application can specify the [SocketOptionName.ReuseUnicastPort](#) option when calling a method such as [Socket.SetSocketOption](#) so that outbound sockets reuse local ports during binding.

- **Support for international domain names and PunyCode**

A new property, [IdnHost](#), has been added to the [Uri](#) class to better support international domain names and PunyCode.

- **Resizing in Windows Forms controls.**

This feature has been expanded in .NET Framework 4.6 to include the [DomainUpDown](#), [NumericUpDown](#), [DataGridViewComboBoxColumn](#), [DataGridViewwColumn](#) and [ToolStripSplitButton](#) types and the rectangle specified by the [Bounds](#) property used when drawing a [UITypeEditor](#).

This is an opt-in feature. To enable it, set the `EnableWindowsFormsHighDpiAutoResizing` element to `true` in the application configuration (app.config) file:

```
<appSettings>
  <add key="EnableWindowsFormsHighDpiAutoResizing" value="true" />
</appSettings>
```

- **Support for code page encodings**

.NET Core primarily supports the Unicode encodings and by default provides limited support for code page encodings. You can add support for code page encodings available in the .NET Framework but unsupported in .NET Core by registering code page encodings with the [Encoding.RegisterProvider](#) method. For more information, see [System.Text.CodePagesEncodingProvider](#).

- **.NET Native**

Windows apps for Windows 10 that target .NET Core and are written in C# or Visual Basic can take advantage of a new technology that compiles apps to native code rather than IL. They produce apps characterized by faster startup and execution times. For more information, see [Compiling Apps with .NET Native](#). For an overview of

.NET Native that examines how it differs from both JIT compilation and NGEN and what that means for your code, see [.NET Native and Compilation](#).

Your apps are compiled to native code by default when you compile them with Visual Studio 2015. For more information, see [Getting Started with .NET Native](#).

To support debugging .NET Native apps, a number of new interfaces and enumerations have been added to the unmanaged debugging API. For more information, see the [Debugging \(Unmanaged API Reference\)](#) topic.

- **Open-source .NET Framework packages**

.NET Core packages such as the immutable collections, [SIMD APIs](#), and networking APIs such as those found in the [System.Net.Http](#) namespace are now available as open source packages on [GitHub](#). To access the code, see [NetFx on GitHub](#). For more information and how to contribute to these packages, see [.NET Core and Open-Source](#), [.NET Home Page on GitHub](#).

[Back to top](#)

What's new in the .NET Framework 4.5.2

- **New APIs for ASP.NET apps.** The new [HttpResponse.AddOnSendingHeaders](#) and [HttpResponseBase.AddOnSendingHeaders](#) methods let you inspect and modify response headers and status code as the response is being flushed to the client app. Consider using these methods instead of the [PreSendRequestHeaders](#) and [PreSendRequestContent](#) events; they are more efficient and reliable.

The [HostingEnvironment.QueueBackgroundWorkItem](#) method lets you schedule small background work items. ASP.NET tracks these items and prevents IIS from abruptly terminating the worker process until all background work items have completed. This method can't be called outside an ASP.NET managed app domain.

The new [HttpResponse.HeadersWritten](#) and [HttpResponseBase.HeadersWritten](#) properties return Boolean values that indicate whether the response headers have been written. You can use these properties to make sure that calls to APIs such as [HttpResponse.StatusCode](#) (which throw exceptions if the headers have been written) will succeed.

- **Resizing in Windows Forms controls.** This feature has been expanded. You can now use the system DPI setting to resize components of the following additional controls (for example, the drop-down arrow in combo boxes):

[ComboBox](#)
[ToolStripComboBox](#)

[ToolStripMenuItem](#)
[Cursor](#)
[DataGridView](#)
[DataGridViewComboBoxColumn](#)

This is an opt-in feature. To enable it, set the `EnableWindowsFormsHighDpiAutoResizing` element to `true` in the application configuration (`app.config`) file:

```
<appSettings>  
  <add key="EnableWindowsFormsHighDpiAutoResizing" value="true" />  
</appSettings>
```

- **New workflow feature.** A resource manager that's using the [EnlistPromotableSinglePhase](#) method (and therefore implementing the [IPromotableSinglePhaseNotification](#) interface) can use the new [Transaction.PromoteAndEnlistDurable](#) method to request the following:
 - Promote the transaction to a Microsoft Distributed Transaction Coordinator (MSDTC) transaction.
 - Replace [IPromotableSinglePhaseNotification](#) with an [ISinglePhaseNotification](#), which is a durable enlistment that supports single phase commits.

This can be done within the same app domain, and doesn't require any extra unmanaged code to interact with MSDTC to perform the promotion. The new method can be called only when there's an outstanding call from [System.Transactions](#) to the [IPromotableSinglePhaseNotification.Promote](#) method that's implemented by the promotable enlistment.

- **Profiling improvements.** The following new unmanaged profiling APIs provide more robust profiling:

[COR_PRF_ASSEMBLY_REFERENCE_INFO Structure](#)
[COR_PRF_HIGH_MONITOR Enumeration](#)
[GetAssemblyReferences Method](#)
[GetEventMask2 Method](#)
[SetEventMask2 Method](#)
[AddAssemblyReference Method](#)

Previous `ICorProfiler` implementations supported lazy loading of dependent assemblies. The new profiling APIs require dependent assemblies that are injected by the profiler to be loadable immediately, instead of being loaded after the app is fully initialized. This change doesn't affect users of the existing `ICorProfiler` APIs.

- **Debugging improvements.** The following new unmanaged debugging APIs provide better integration with a profiler. You can now access metadata inserted by the

profiler as well as local variables and code produced by compiler ReJIT requests when dump debugging.

[SetWriteableMetadataUpdateMode Method](#)

[EnumerateLocalVariablesEx Method](#)

[GetLocalVariableEx Method](#)

[GetCodeEx Method](#)

[GetActiveRejitRequestILCode Method](#)

[GetInstrumentedILMap Method](#)

- **Event tracing changes.** The .NET Framework 4.5.2 enables out-of-process, Event Tracing for Windows (ETW)-based activity tracing for a larger surface area. This enables Advanced Power Management (APM) vendors to provide lightweight tools that accurately track the costs of individual requests and activities that cross threads. These events are raised only when ETW controllers enable them; therefore, the changes don't affect previously written ETW code or code that runs with ETW disabled.
- **Promoting a transaction and converting it to a durable enlistment**

[Transaction.PromoteAndEnlistDurable](#) is a new API added to the .NET Framework 4.5.2 and 4.6:

```
[System.Security.Permissions.PermissionSetAttribute(System.Security.Permissions.SecurityAction.LinkDemand, Name = "FullTrust")]  
public Enlistment PromoteAndEnlistDurable(Guid  
resourceManagerIdentifier,
```

```
IPromotableSinglePhaseNotification promotableNotification,  
                                     ISinglePhaseNotification  
enlistmentNotification,  
                                     EnlistmentOptions  
enlistmentOptions)
```

The method may be used by an enlistment that was previously created by [Transaction.EnlistPromotableSinglePhase](#) in response to the [ITransactionPromoter.Promote](#) method. It asks `System.Transactions` to promote the transaction to an MSDTC transaction and to “convert” the promotable enlistment to a durable enlistment. After this method completes successfully, the [IPromotableSinglePhaseNotification](#) interface will no longer be referenced by `System.Transactions`, and any future notifications will arrive on the provided [ISinglePhaseNotification](#) interface. The enlistment in question must act as a durable enlistment, supporting transaction logging and recovery. Refer to [Transaction.EnlistDurable](#) for details. In addition, the enlistment must support [ISinglePhaseNotification](#). This method can *only* be called while processing

an [ITransactionPromoter.Promote](#) call. If that is not the case, a [TransactionException](#) exception is thrown.

[Back to top](#)

What's new in the .NET Framework 4.5.1

April 2014 updates:

- [Visual Studio 2013 Update 2](#) includes updates to the Portable Class Library templates to support these scenarios:
 - You can use Windows Runtime APIs in portable libraries that target Windows 8.1, Windows Phone 8.1, and Windows Phone Silverlight 8.1.
 - You can include XAML (Windows.UI.Xaml types) in portable libraries when you target Windows 8.1 or Windows Phone 8.1. The following XAML templates are supported: Blank Page, Resource Dictionary, Templated Control, and User Control.
 - You can create a portable Windows Runtime component (.winmd file) for use in Store apps that target Windows 8.1 and Windows Phone 8.1.
 - You can retarget a Windows Store or Windows Phone Store class library like a Portable Class Library.

For more information about these changes, see [Portable Class Library](#).

- The .NET Framework content set now includes documentation for .NET Native, which is a precompilation technology for building and deploying Windows apps. .NET Native compiles your apps directly to native code, rather than to intermediate language (IL), for better performance. For details, see [Compiling Apps with .NET Native](#).
- The [.NET Framework Reference Source](#) provides a new browsing experience and enhanced functionality. You can now browse through the .NET Framework source code online, [download the reference](#) for offline viewing, and step through the sources (including patches and updates) during debugging. For more information, see the blog entry [A new look for .NET Reference Source](#).

Core new features and enhancements in the .NET Framework 4.5.1 include:

- Automatic binding redirection for assemblies. Starting with Visual Studio 2013, when you compile an app that targets the .NET Framework 4.5.1, binding redirects may be added to the app configuration file if your app or its components reference multiple versions of the same assembly. You can also enable this feature for projects that target older versions of the .NET Framework. For more information, see [How to: Enable and Disable Automatic Binding Redirection](#).

- Ability to collect diagnostics information to help developers improve the performance of server and cloud applications. For more information, see the [WriteEventWithRelatedActivityId](#) and [WriteEventWithRelatedActivityIdCore](#) methods in the [EventSource](#) class.
- Ability to explicitly compact the large object heap (LOH) during garbage collection. For more information, see the [GCSettings.LargeObjectHeapCompactionMode](#) property.
- Additional performance improvements such as ASP.NET app suspension, multi-core JIT improvements, and faster app startup after a .NET Framework update. For details, see the [.NET Framework 4.5.1 announcement](#) and the [ASP.NET app suspend](#) blog post.

Improvements to Windows Forms include:

- Resizing in Windows Forms controls. You can use the system DPI setting to resize components of controls (for example, the icons that appear in a property grid) by opting in with an entry in the application configuration file (app.config) for your app. This feature is currently supported in the following Windows Forms controls:

[PropertyGrid](#)

[TreeView](#)

Some aspects of the [DataGridView](#) (see [new features in 4.5.2](#) for additional controls supported)

To enable this feature, add a new <appSettings> element to the configuration file (app.config) and set the `EnableWindowsFormsHighDpiAutoResizing` element to `true`:

```
<appSettings>
  <add key="EnableWindowsFormsHighDpiAutoResizing" value="true" />
</appSettings>
```

Improvements when debugging your .NET Framework apps in Visual Studio 2013 include:

- Return values in the Visual Studio debugger. When you debug a managed app in Visual Studio 2013, the Autos window displays return types and values for methods. This information is available for desktop, Windows Store, and Windows Phone apps. For more information, see [Examine return values of method calls](#) in the MSDN Library.
- Edit and Continue for 64-bit apps. Visual Studio 2013 supports the Edit and Continue feature for 64-bit managed apps for desktop, Windows Store, and Windows Phone. The existing limitations remain in effect for both 32-bit and 64-bit apps (see the last section of the [Supported Code Changes \(C#\)](#) article).

- Async-aware debugging. To make it easier to debug asynchronous apps in Visual Studio 2013, the call stack hides the infrastructure code provided by compilers to support asynchronous programming, and also chains in logical parent frames so you can follow logical program execution more clearly. A Tasks window replaces the Parallel Tasks window and displays tasks that relate to a particular breakpoint, and also displays any other tasks that are currently active or scheduled in the app. You can read about this feature in the "Async-aware debugging" section of the [.NET Framework 4.5.1 announcement](#).
- Better exception support for Windows Runtime components. In Windows 8.1, exceptions that arise from Windows Store apps preserve information about the error that caused the exception, even across language boundaries. You can read about this feature in the "Windows Store app development" section of the [.NET Framework 4.5.1 announcement](#).

Starting with Visual Studio 2013, you can use the [Managed Profile Guided Optimization Tool \(Mpgo.exe\)](#) to optimize Windows 8.x Store apps as well as desktop apps.

For new features in ASP.NET 4.5.1, see [ASP.NET 4.5.1 and Visual Studio 2013](#) on the ASP.NET site.

[Back to top](#)

What's new in the .NET Framework 4.5

Core new features and improvements

- Ability to reduce system restarts by detecting and closing .NET Framework 4 applications during deployment. See [Reducing System Restarts During .NET Framework 4.5 Installations](#).
- Support for arrays that are larger than 2 gigabytes (GB) on 64-bit platforms. This feature can be enabled in the application configuration file. See the [<gcAllowVeryLargeObjects> element](#), which also lists other restrictions on object size and array size.
- Better performance through background garbage collection for servers. When you use server garbage collection in the .NET Framework 4.5, background garbage collection is automatically enabled. See the Background Server Garbage Collection section of the [Fundamentals of Garbage Collection](#) topic.
- Background just-in-time (JIT) compilation, which is optionally available on multi-core processors to improve application performance. See [ProfileOptimization](#).
- Ability to limit how long the regular expression engine will attempt to resolve a regular expression before it times out. See the [Regex.MatchTimeout](#) property.

- Ability to define the default culture for an application domain. See the [CultureInfo](#) class.
- Console support for Unicode (UTF-16) encoding. See the [Console](#) class.
- Support for versioning of cultural string ordering and comparison data. See the [SortVersion](#) class.
- Better performance when retrieving resources. See [Packaging and Deploying Resources](#).
- Zip compression improvements to reduce the size of a compressed file. See the [System.IO.Compression](#) namespace.
- Ability to customize a reflection context to override default reflection behavior through the [CustomReflectionContext](#) class.
- Support for the 2008 version of the Internationalized Domain Names in Applications (IDNA) standard when the [System.Globalization.IdnMapping](#) class is used on Windows 8.
- Delegation of string comparison to the operating system, which implements Unicode 6.0, when the .NET Framework is used on Windows 8. When running on other platforms, the .NET Framework includes its own string comparison data, which implements Unicode 5.x. See the [String](#) class and the Remarks section of the [SortVersion](#) class.
- Ability to compute the hash codes for strings on a per application domain basis. See [<UseRandomizedStringHashAlgorithm> Element](#).
- Type reflection support split between [Type](#) and [TypeInfo](#) classes. See [Reflection in the .NET Framework for Windows Store Apps](#).

Managed Extensibility Framework (MEF)

In the .NET Framework 4.5, the Managed Extensibility Framework (MEF) provides the following new features:

- Support for generic types.
- Convention-based programming model that enables you to create parts based on naming conventions rather than attributes.
- Multiple scopes.

- A subset of MEF that you can use when you create Windows 8.x Store apps. This subset is available as a [downloadable package](#) from the NuGet Gallery. To install the package, open your project in Visual Studio, choose **Manage NuGet Packages** from the **Project** menu, and search online for the `Microsoft.Composition` package.

For more information, see [Managed Extensibility Framework \(MEF\)](#).

Asynchronous file operations

In the .NET Framework 4.5, new asynchronous features were added to the C# and Visual Basic languages. These features add a task-based model for performing asynchronous operations. To use this new model, use the asynchronous methods in the I/O classes. See [Asynchronous File I/O](#).

Tools

In the .NET Framework 4.5, Resource File Generator (Resgen.exe) enables you to create a .resw file for use in Windows 8.x Store apps from a .resources file embedded in a .NET Framework assembly. For more information, see [Resgen.exe \(Resource File Generator\)](#).

Managed Profile Guided Optimization (Mpggo.exe) enables you to improve application startup time, memory utilization (working set size), and throughput by optimizing native image assemblies. The command-line tool generates profile data for native image application assemblies. See [Mpggo.exe \(Managed Profile Guided Optimization Tool\)](#). Starting with Visual Studio 2013, you can use Mpggo.exe to optimize Windows 8.x Store apps as well as desktop apps.

Parallel computing

The .NET Framework 4.5 provides several new features and improvements for parallel computing. These include improved performance, increased control, improved support for asynchronous programming, a new dataflow library, and improved support for parallel debugging and performance analysis. See the entry [What's New for Parallelism in .NET 4.5](#) in the Parallel Programming with .NET blog.

Web

ASP.NET 4.5 and 4.5.1 add model binding for Web Forms, WebSocket support, asynchronous handlers, performance enhancements, and many other features. For more information, see the following resources:

- [ASP.NET 4.5 and Visual Studio 2012](#) in the MSDN Library.
- [ASP.NET 4.5.1 and Visual Studio 2013](#) on the ASP.NET site.

Networking

The .NET Framework 4.5 provides a new programming interface for HTTP applications. For more information, see the new [System.Net.Http](#) and [System.Net.Http.Headers](#) namespaces.

Support is also included for a new programming interface for accepting and interacting with a WebSocket connection by using the existing [HttpListener](#) and related classes. For more information, see the new [System.Net.WebSockets](#) namespace and the [HttpListener](#) class.

In addition, the .NET Framework 4.5 includes the following networking improvements:

- RFC-compliant URI support. For more information, see [Uri](#) and related classes.
- Support for Internationalized Domain Name (IDN) parsing. For more information, see [Uri](#) and related classes.
- Support for Email Address Internationalization (EAI). For more information, see the [System.Net.Mail](#) namespace.
- Improved IPv6 support. For more information, see the [System.Net.NetworkInformation](#) namespace.
- Dual-mode socket support. For more information, see the [Socket](#) and [TcpListener](#) classes.

Windows Presentation Foundation (WPF)

In the .NET Framework 4.5, Windows Presentation Foundation (WPF) contains changes and improvements in the following areas:

- The new [Ribbon](#) control, which enables you to implement a ribbon user interface that hosts a Quick Access Toolbar, Application Menu, and tabs.
- The new [INotifyDataErrorInfo](#) interface, which supports synchronous and asynchronous data validation.
- New features for the [VirtualizingPanel](#) and [Dispatcher](#) classes.
- Improved performance when displaying large sets of grouped data, and by accessing collections on non-UI threads.
- Data binding to static properties, data binding to custom types that implement the [ICustomTypeProvider](#) interface, and retrieval of data binding information from a binding expression.
- Repositioning of data as the values change (live shaping).

- Ability to check whether the data context for an item container is disconnected.
- Ability to set the amount of time that should elapse between property changes and data source updates.
- Improved support for implementing weak event patterns. Also, events can now accept markup extensions.

Windows Communication Foundation (WCF)

In the .NET Framework 4.5, the following features have been added to make it simpler to write and maintain Windows Communication Foundation (WCF) applications:

- Simplification of generated configuration files.
- Support for contract-first development.
- Ability to configure ASP.NET compatibility mode more easily.
- Changes in default transport property values to reduce the likelihood that you will have to set them.
- Updates to the [XmlDictionaryReaderQuotas](#) class to reduce the likelihood that you will have to manually configure quotas for XML dictionary readers.
- Validation of WCF configuration files by Visual Studio as part of the build process, so you can detect configuration errors before you run your application.
- New asynchronous streaming support.
- New HTTPS protocol mapping to make it easier to expose an endpoint over HTTPS with Internet Information Services (IIS).
- Ability to generate metadata in a single WSDL document by appending `?singleWSDL` to the service URL.
- Websockets support to enable true bidirectional communication over ports 80 and 443 with performance characteristics similar to the TCP transport.
- Support for configuring services in code.
- XML Editor tooltips.
- [ChannelFactory](#) caching support.
- Binary encoder compression support.

- Support for a UDP transport that enables developers to write services that use "fire and forget" messaging. A client sends a message to a service and expects no response from the service.
- Ability to support multiple authentication modes on a single WCF endpoint when using the HTTP transport and transport security.
- Support for WCF services that use internationalized domain names (IDNs).

For more information, see [What's New in Windows Communication Foundation](#).

Windows Workflow Foundation (WF)

In the .NET Framework 4.5, several new features were added to Windows Workflow Foundation (WF), including:

- State machine workflows, which were first introduced as part of the .NET Framework 4.0.1 ([.NET Framework 4 Platform Update 1](#)). This update included several new classes and activities that enabled developers to create state machine workflows. These classes and activities were updated for the .NET Framework 4.5 to include:
 - The ability to set breakpoints on states.
 - The ability to copy and paste transitions in the workflow designer.
 - Designer support for shared trigger transition creation.
 - Activities for creating state machine workflows, including: [StateMachine](#), [State](#), and [Transition](#).
- Enhanced Workflow Designer features such as the following:
 - Enhanced workflow search capabilities in Visual Studio, including **Quick Find** and **Find in Files**.
 - Ability to automatically create a Sequence activity when a second child activity is added to a container activity, and to include both activities in the Sequence activity.
 - Panning support, which enables the visible portion of a workflow to be changed without using the scroll bars.
 - A new **Document Outline** view that shows the components of a workflow in a tree-style outline view and lets you select a component in the **Document Outline** view.

- Ability to add annotations to activities.
- Ability to define and consume activity delegates by using the workflow designer.
- Auto-connect and auto-insert for activities and transitions in state machine and flowchart workflows.
- Storage of the view state information for a workflow in a single element in the XAML file, so you can easily locate and edit the view state information.
- A NoPersistScope container activity to prevent child activities from persisting.
- Support for C# expressions:
 - Workflow projects that use Visual Basic will use Visual Basic expressions, and C# workflow projects will use C# expressions.
 - C# workflow projects that were created in Visual Studio 2010 and that have Visual Basic expressions are compatible with C# workflow projects that use C# expressions.
- Versioning enhancements:
 - The new [WorkflowIdentity](#) class, which provides a mapping between a persisted workflow instance and its workflow definition.
 - Side-by-side execution of multiple workflow versions in the same host, including [WorkflowServiceHost](#).
 - In Dynamic Update, the ability to modify the definition of a persisted workflow instance.
- Contract-first workflow service development, which provides support for automatically generating activities to match an existing service contract.

For more information, see [What's New in Windows Workflow Foundation](#).

.NET for Windows 8.x Store apps

Windows 8.x Store apps are designed for specific form factors and leverage the power of the Windows operating system. A subset of the .NET Framework 4.5 or 4.5.1 is available for building Windows 8.x Store apps for Windows by using C# or Visual Basic. This subset is called .NET for Windows 8.x Store apps and is discussed in an [overview](#) in the Windows Dev Center.

Portable Class Libraries

The Portable Class Library project in Visual Studio 2012 (and later versions) enables you to write and build managed assemblies that work on multiple .NET Framework platforms. Using a Portable Class Library project, you choose the platforms (such as Windows Phone and .NET for Windows 8.x Store apps) to target. The available types and members in your project are automatically restricted to the common types and members across these platforms. For more information, see [Portable Class Library](https://msdn.microsoft.com/en-us/library/ms171868(v=vs.110).aspx).

[https://msdn.microsoft.com/en-us/library/ms171868\(v=vs.110\).aspx](https://msdn.microsoft.com/en-us/library/ms171868(v=vs.110).aspx)

params (C# Reference)

For the latest documentation on Visual Studio 2017 RC, see [Visual Studio 2017 RC Documentation](#).

By using the `params` keyword, you can specify a [method parameter](#) that takes a variable number of arguments.

You can send a comma-separated list of arguments of the type specified in the parameter declaration or an array of arguments of the specified type. You also can send no arguments. If you send no arguments, the length of the `params` list is zero.

No additional parameters are permitted after the `params` keyword in a method declaration, and only one `params` keyword is permitted in a method declaration.

Example

The following example demonstrates various ways in which arguments can be sent to a `params` parameter.

C#

```
public class MyClass
{
    public static void UseParams(params int[] list)
    {
        for (int i = 0; i < list.Length; i++)
```

```

        {
            Console.Write(list[i] + " ");
        }
        Console.WriteLine();
    }

    public static void UseParams2(params object[] list)
    {
        for (int i = 0; i < list.Length; i++)
        {
            Console.Write(list[i] + " ");
        }
        Console.WriteLine();
    }

    static void Main()
    {
        // You can send a comma-separated list of arguments of the
        // specified type.
        UseParams(1, 2, 3, 4);
        UseParams2(1, 'a', "test");

        // A params parameter accepts zero or more arguments.
        // The following calling statement displays only a blank line.
        UseParams2();

        // An array argument can be passed, as long as the array
        // type matches the parameter type of the method being called.
        int[] myIntArray = { 5, 6, 7, 8, 9 };
        UseParams(myIntArray);

        object[] myObjArray = { 2, 'b', "test", "again" };
        UseParams2(myObjArray);

        // The following call causes a compiler error because the object
        // array cannot be converted into an integer array.
        //UseParams(myObjArray);

        // The following call does not cause an error, but the entire
        // integer array becomes the first element of the params array.
        UseParams2(myIntArray);
    }
}
/*
Output:
1 2 3 4
1 a test

5 6 7 8 9
2 b test again
System.Int32[]
*/

```

<https://msdn.microsoft.com/en-us/library/w5zay9db.aspx>

LinkedList Algorithm

Define a Node with content and next node details

Have required node objects, head, current, size

Add Node to LinkedList

- check for head is null or not

- if Null then make current node as head

- else make current.next as the adding node

- make the added node as current node

ListNodes

- assign head to tempNode

- continue the loop till tempNode gets to null

- assign tempNode.next to tempNode

Retrieve Node

- assign head to tempNode

- assign null to returnNode

intialise 0 to count

continue the loop till gets the count equal to required position

then return the tempNode as returnNode

break

increment the count

assign tempNode.next to tempNode so that the iteration will go on

return returnNode

[A simple and pure C# implementation of the good old linked list](#)

Back to Forum: [Tech Off](#)



• [Mauricio Feijo](#)

While interviewing for a gig I was asked to complete an assignment, as a test. The request was to implement a C# linked list, without using Collections. They also required me to write Add, Delete and Retrieve methods with the signatures as they are in the code below.

This is an ages old exercise but every time a do it I feel it gets a bit cleaner and better.

With no more delay, here is the code:

```
using System;
```



```

namespace LinkedListExample
{
    public class List
    {
        public class Node
        {
            public object NodeContent;
            public Node Next;
        }

        private int size;
        public int Count
        {
            get
            {
                return size;
            }
        }

        /// <summary>
        /// The head of the list.
        /// </summary>
        private Node head;

        /// <summary>
        /// The current node, used to avoid
adding nodes before the head
        /// </summary>
        private Node current;

        public List()
        {
            size = 0;
            head = null;

```

```

    }

    /// <summary>
    /// Add a new Node to the list.
    /// </summary>
    public void Add(object content)
    {
        size++;

        // This is a more verbose
implementation to avoid adding nodes to the head
of the list
        var node = new Node()
        {
            NodeContent = content
        };

        if (head == null)
        {
            // This is the first node. Make
it the head
            head = node;
        }
        else
        {
            // This is not the head. Make it
current's next node.
            current.Next = node;
        }

        // Makes newly added node the
current node
        current = node;
    }

```

// This implementation is simpler
but adds nodes in reverse order. It adds nodes
to the head of the list

```
        //head = new Node()
        //{
        //    Next = head,
        //    NodeContent = content
        //};

    }

    /// <summary>
    /// Throwing this in to help test the
list
    /// </summary>
    public void ListNodes()
    {
        Node tempNode = head;

        while (tempNode != null)
        {
            Console.WriteLine(tempNode.NodeContent);
            tempNode = tempNode.Next;
        }
    }

    /// <summary>
    /// Returns the Node in the specified
position or null if inexistent
    /// </summary>
```

```
    /// <param name="Position">One based  
position of the node to retrieve</param>  
    /// <returns>The desired node or null if  
inexistent</returns>
```

```
    public Node Retrieve(int Position)  
    {
```

```
        Node tempNode = head;  
        Node retNode = null;  
        int count = 0;
```

```
        while (tempNode != null)  
        {  
            if (count == Position - 1)  
            {  
                retNode = tempNode;  
                break;  
            }  
            count++;  
            tempNode = tempNode.Next;  
        }
```

```
        return retNode;  
    }
```

```
    /// <summary>  
    /// Delete a Node in the specified  
position  
    /// </summary>  
    /// <param name="Position">Position of  
node to be deleted</param>
```

```
    /// <returns>Successful</returns>  
    public bool Delete(int position)  
    {  
        LNode curNode = head;  
        LNode nextNode = head;  
        int count = 0;  
  
        if (position == 1)
```

```

{
    if (position == size)
    {
        head = null;
        current = null;
        size--;
        return true;
    }
    else if (position > size)
    {
        return false;
    }
    else
    {
        head = head.next;
        size--;
        return true;
    }
}
if (position > 1 && position <= size)
{
    while (nextNode != null)
    {
        if (count == position - 1)
        {
            curNode.next = nextNode.next;
            size--;
            return true;
        }
        count++;
        curNode = nextNode;
        nextNode = nextNode.next;
    }
}
return false;
}
}
}

```

I added a client console app to test the linked list. Here is how it turned out:

```

using LinkedListExample;
using System;

namespace LinkedListExampleClient

```

```

{
    class Program
    {
        static void Main(string[] args)
        {
            List list = new List();

            list.Add("A");
            list.Add("B");
            list.Add("C");
            list.Add("D");
            list.Add("E");
            list.Add("F");
            list.Add("G");
            list.Add("H");

            list.ListNodes();
            Console.WriteLine();

            Console.WriteLine();
            Console.WriteLine("Deleting node
8");

            list.Delete(8);
            list.ListNodes();

            Console.WriteLine();
            Console.WriteLine("Position 5: " +
list.Retrieve(5).NodeContent);

            Console.WriteLine();
            Console.WriteLine("Deleting node
5");

            list.Delete(5);

            Console.WriteLine();

```

```

        Console.WriteLine("Position 5: " +
list.Retrieve(5).NodeContent);

        Console.WriteLine();
        list.ListNodes();

        Console.ReadLine();

    }
}
}

```

<https://channel9.msdn.com/Forums/TechOff/A-simple-and-pure-C-implementation-of-the-good-old-linked-list>

Where can I find difficult algorithm/data structure problems?

Sites that contain algorithmic question. These sites are useful if one has little preparation time (a few days to one or two weeks).

- [List of Problems - Techie Delight](#)
- [Coderust 2.0: Faster Coding Interview Preparation using Visualizations](#)
- <http://courses.csail.mit.edu/iap...>
- [InterviewBit](#)
- [GeeksforGeeks - A computer science portal for geeks](#)
- [CS Academy](#)
- <http://www.qubeet.com/>
- <http://everything2.com/title/har...>
- <http://www.intertechtion.com/ans...>
- <http://algomuse.appspot.com/archive>
- [Interview Questions | CareerCup](#)
- <http://www.dsalgo.com/>
- [Coding Interview Questions](#)

- [We Help Coders Get Hired](#)
- [Data Structure - Assessment](#)

Sites that have online judges, and are useful for longer term growth (think 2 months or more):

- [HackerRank](#) - by Interviewstreet
- <http://www.topcoder.com>
- [CS Academy...](#)

[\(more\)](#)