Visual Studio 2015 Enhancements

- Akhil Mittal

Visual Studio 2015 Enhancements

- Akhil Mittal

Visual Studio 2015 Enhancements

(A deep dive into new improved Visual Studio 2015 features)

Akhil Mittal

Sr. Analyst (Magic Software)

https://www.facebook.com/csharppulse

https://in.linkedin.com/in/akhilmittal

https://twitter.com/AkhilMittal20

google.com/+AkhilMittal

https://www.codeteddy.com

https://github.com/akhilmittal

Table of Contents

About Author

*Akhil Mittal is a Microsoft MVP, C# Corner MVP, a Code project MVP, blogger, programmer by heart and currently working as a Sr. Analyst in **Magic Software** and have an experience of more than 9 years in C#.Net. He is a B.Tech in Computer Science and holds a diploma in Information Security and Application Development. His work experience includes Development of Enterprise Applications using C#, .Net and SQL Server, Analysis as well as Research and Development. He is a MCP in Web Applications (MCTS-70-528, MCTS-70-515) and .Net Framework 2.0 (MCTS-70-536). Visit his personal blog **CodeTeddy** for more informative articles.*

Akhil Mittal

Sr. Analyst (Magic Software)

Preface

I have always been a great admirer of Visual Studio IDE (Interactive Development Environment) . Visual Studio has proved to be the best IDE for me and I use it for almost all my coding as well as debugging work. My love for the IDE has forced me to write this book to explain what more Visual Studio 2015 now offers to a developer in terms of cross platform development, cloud based development, code assistance, refactoring, debugging and a lot more. The power of Visual Studio is now not only limited to development and coding but it offers a one stop solution to all the requirements needed while coding, development, code analysis or deployment. I'll use Visual Studio Enterprise 2015 throughout all the chapters and explain how one can leverage Visual Studio to be more productive.

Code Assistance in visual Studio 2015

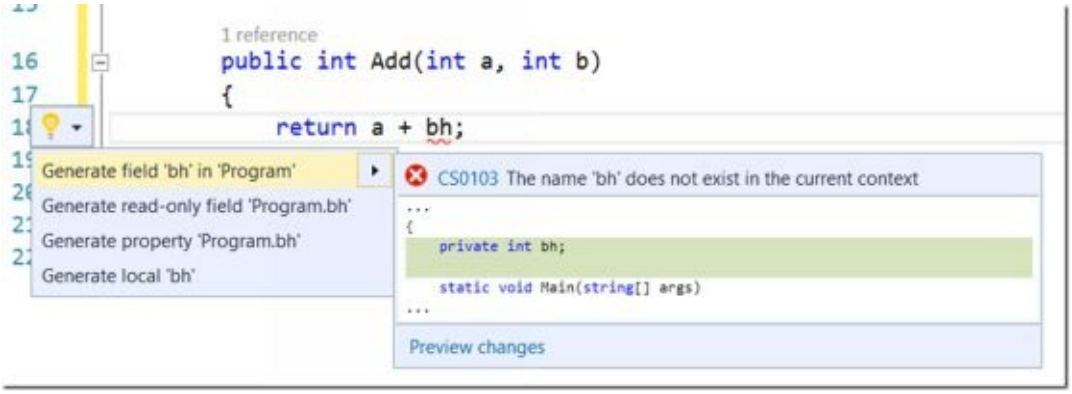
In this part of the book, I’ll cover how development with Visual Studio 2015 can increase your productivity to n times and enables you to write more cleaner and optimized code.

Code Assistance

In earlier versions of Visual studio, you must have seen that whenever you write a buggy code, the code editor provides suggestion with the help of a tool tip. This feature have improved a lot and is shown as a light bulb icon in Visual Studio code editor. This option provides you the real time suggestions while coding in Visual Studio code editor to improve code quality or fix the coding issues. It helps you in identifying syntax errors, provides useful code hints and assist you with static code analysis. I am using a sample code to explain the enhancements, for that I have created a console application in my Visual Studio and named it VisualStudio2015ConsoleApplication.

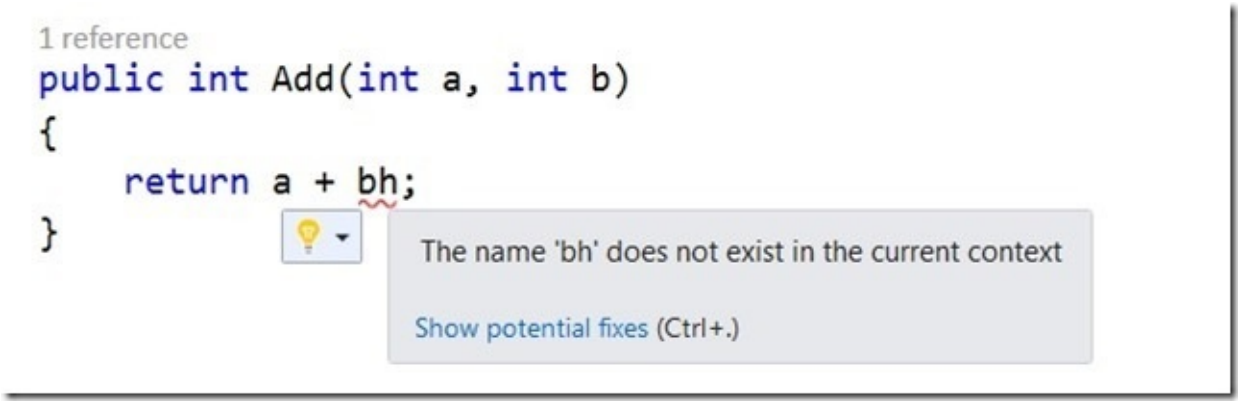
Syntax Error Suggestions

Suppose there is syntax error in your code like I purposely did in the below image,

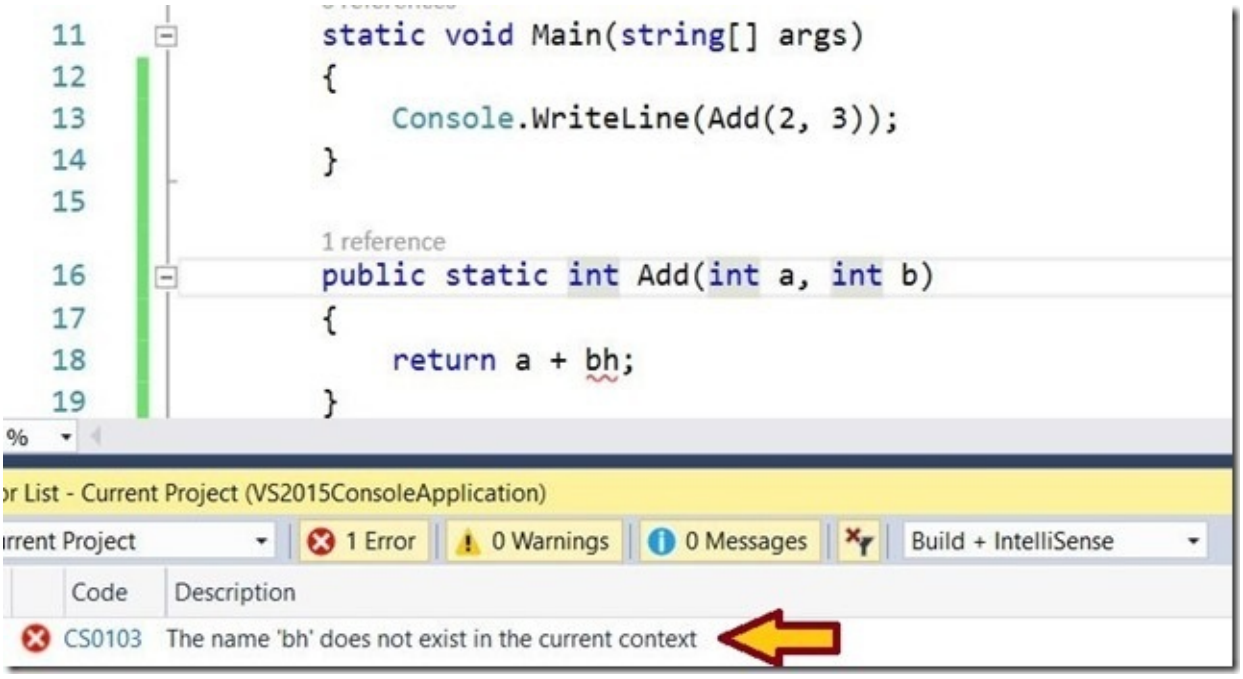


The light bulb icon immediately shows up when you click your mouse over the erroneous variable having red line and displays an issue summary, an error code with a link to documentation. It also displays a possible list of code fixes and refactoring's. In above mentioned example I am writing an add method taking two parameters a and b, but I am trying to return a result as a+bh. Now since "bh" is not declared anywhere in the method or passed as a parameter, the light bulb icon shows up and provides certain possible options or suggestions about how this variable can be taken care of. It suggests to generate a variable named "bh" , create a field or property as well.

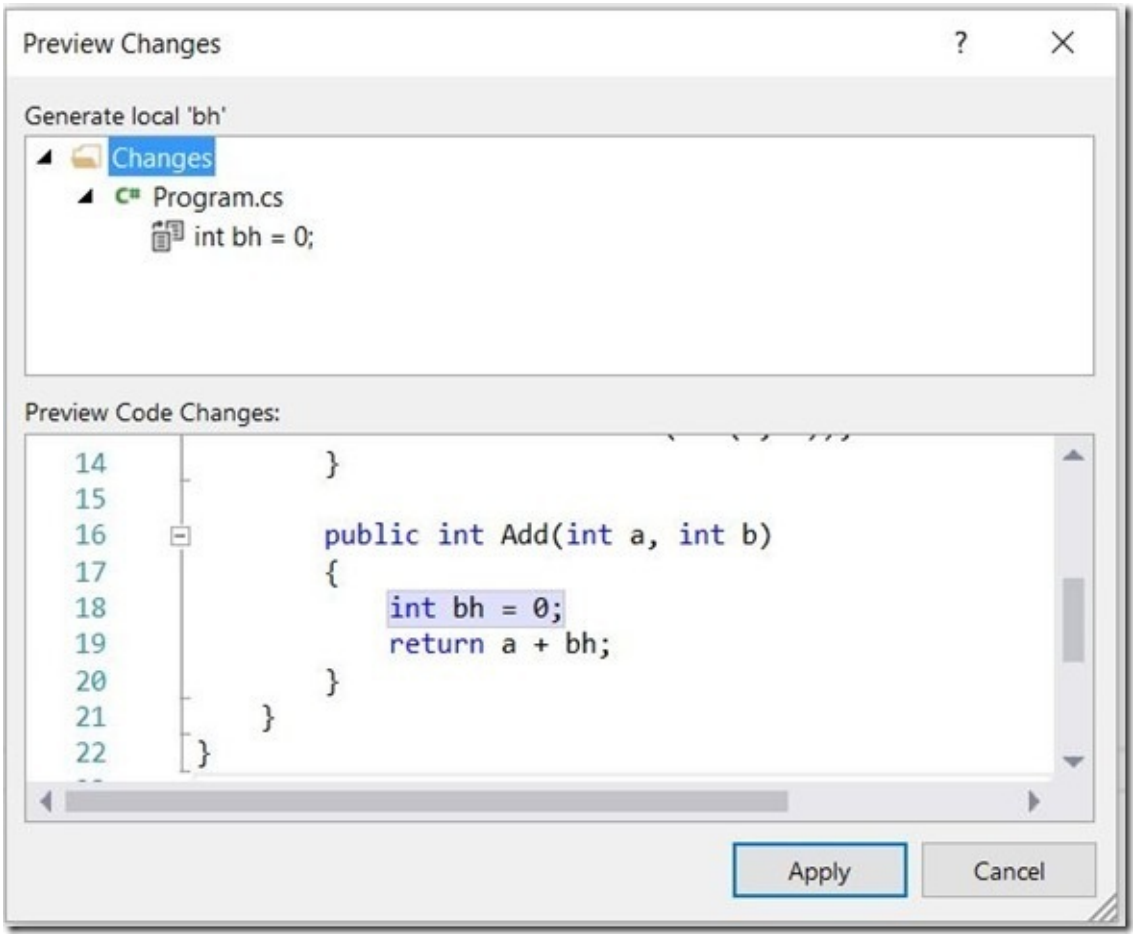
If you hover on the error line you'll be shown a light bulb icon showing error and potential fixes.



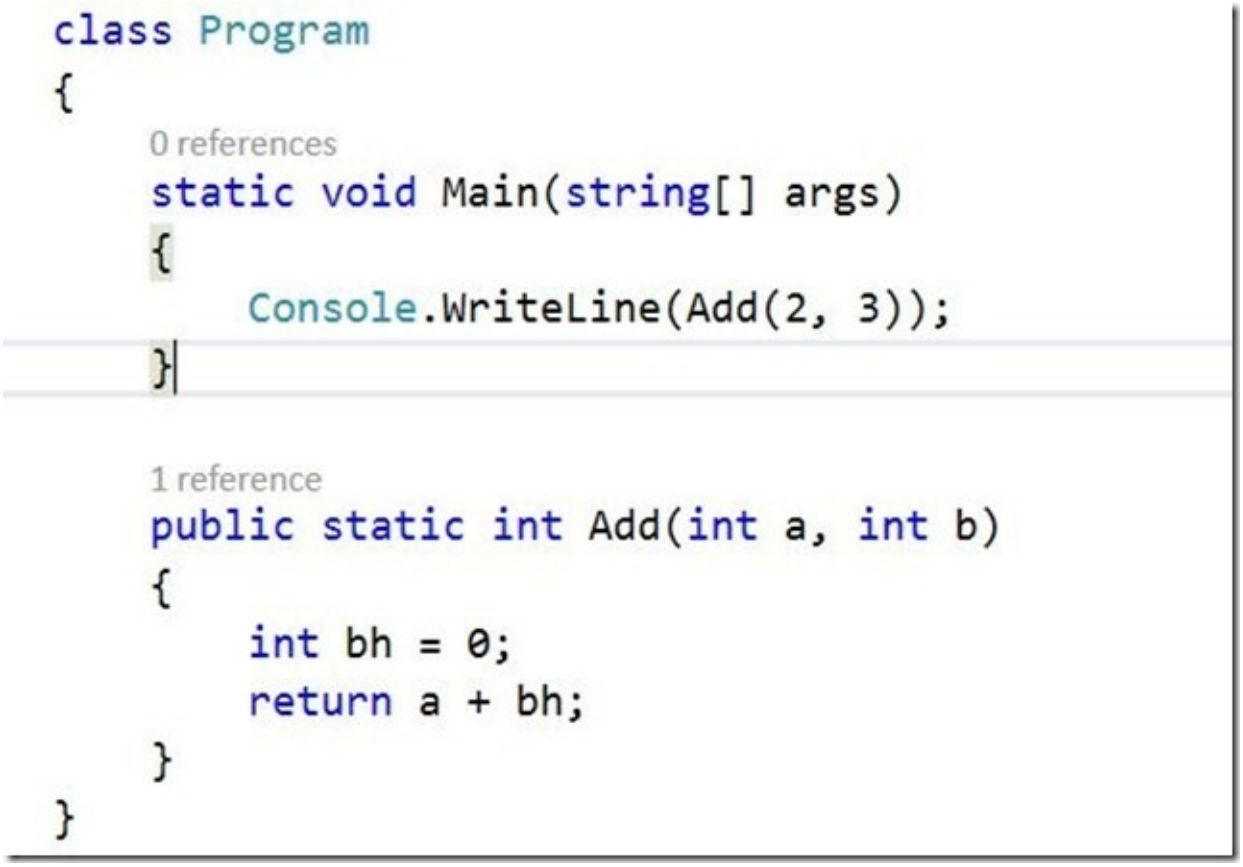
Note that you can also use Ctrl+. to see the error using your keyboard. If you click on Show potential fixes, you'll get the same options as shown in the first image. Alternatively if by any chance you doubt light bulb icon and build your console application, you'll again be shown the same error as follow.



The syntax error assistance displays a light bulb icon, a description of the error, and a link to show possible fixes as well. When you click on the error code i.e. CS0103 you'll be redirected to the documentation of the error code. It also offers to preview changes once you go for any of the suggestions provided by light bulb icon. So if I click on Preview changes it shows me the preview of the option that I have chosen as shown below.



Now we don't have to go to code and explicitly define that variable. Just click on Apply button and Visual Studio takes care of everything. Therefore the first option that I chose is now reflected in my code.



I remember that I used to do these on the fly modifications to improve productivity using ReSharper. We saw that we got the same error on compiling the application which proves that light bulb icon's code suggestion can help us write error free code before even compiling the application and getting to know about the actual error after a compile. Therefore we don't have to wait to compile the application to know about compile time error. You can test different scenarios to explore syntax error suggestions given by light bulb icon in our day to day programming.

Code Suggestions

Let’s take another scenario. Suppose I define an interface named ICalculator and add a class named Calculator and inherit the calculator.cs class from that interface.

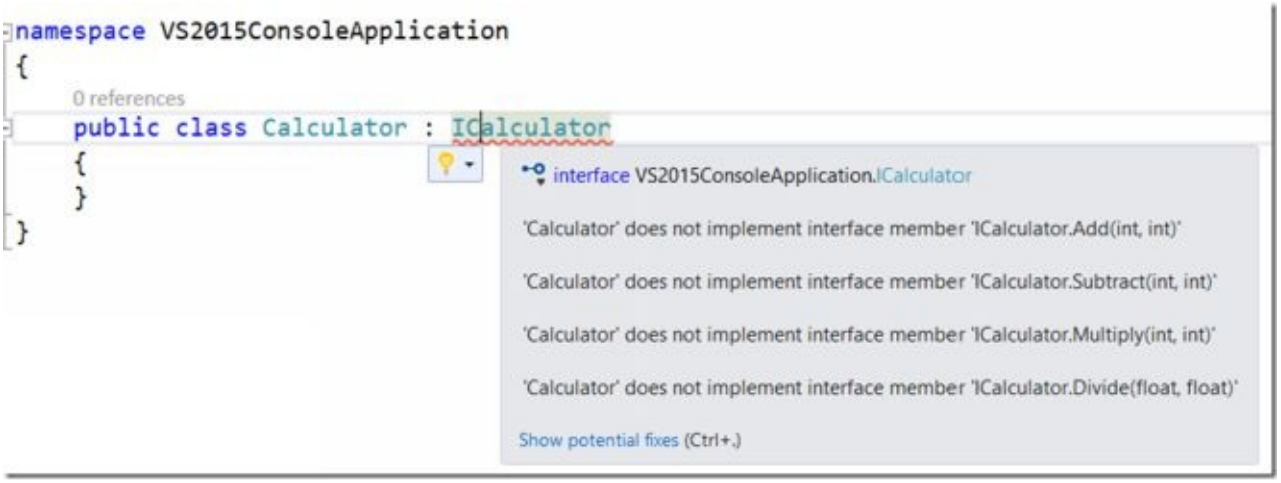
Interface

```
1: interface ICalculator
2:  {
3:      int Add(int a, int b);
4:      int Subtract(int a, int b);
5:      int Multiply(int a, int b);
6:      float Divide(float a, float b);
7:  }
```

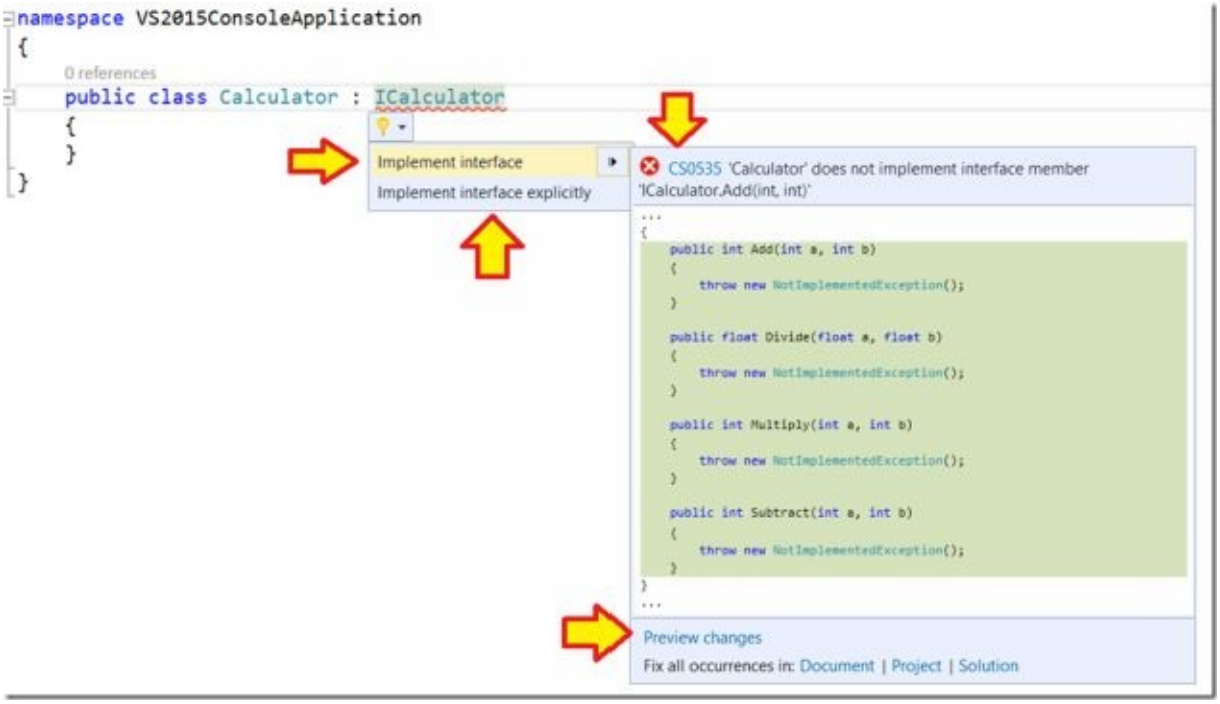
Class

```
1: public class Calculator : ICalculator
2:  {
3:  }
```

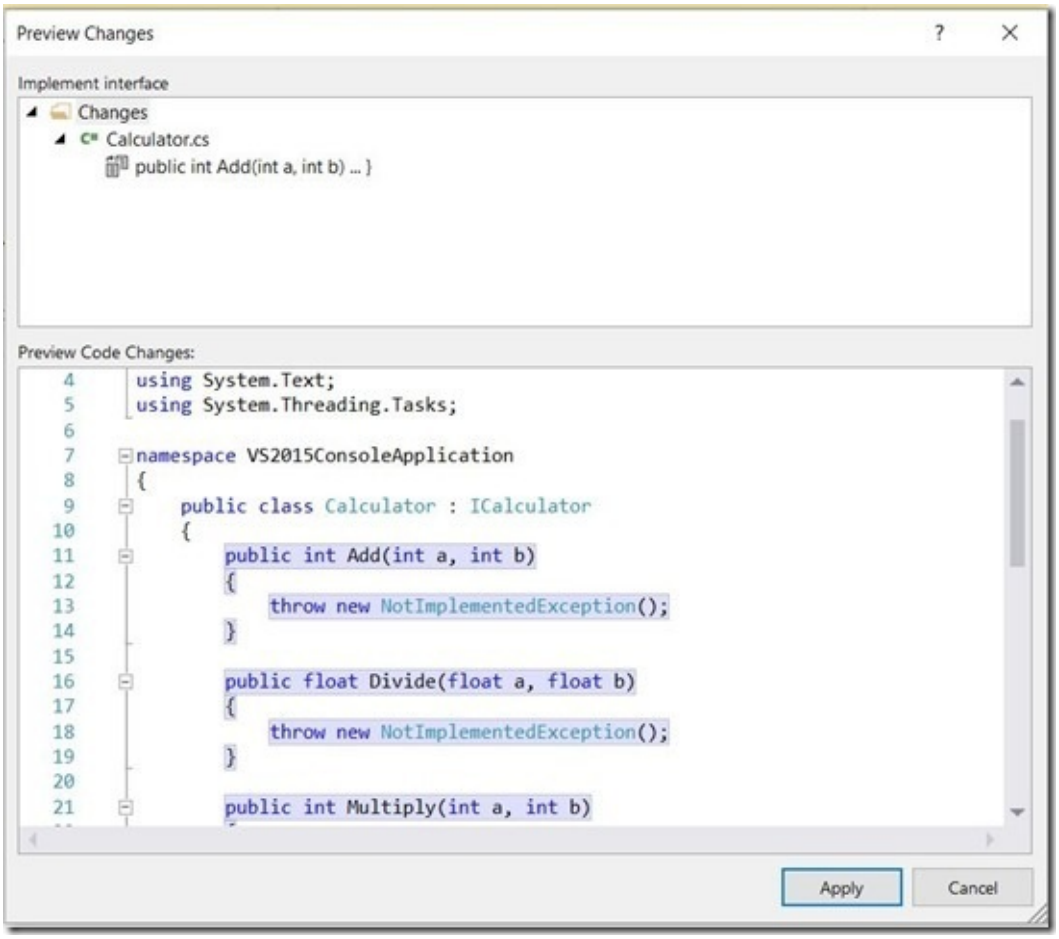
You’ll see that there will be a red error line under the ICalculator interface named in calculator class. You can get to see the light bulb icon in the same way as shown in previous example i.e. hover or click on error. Here you’ll see that light bulb icon is assisting us with some additional conceptual information that the interface that we are using is contains several methods that needs to be implemented in Calculator class.



Therefore we see that light bulb not only assists us in finding syntax error but also suggests conceptual or logical resolution of mistakes we do in programming. When you show on “Show potential fixes link” it will show all the possible fixes for this error in a detailed user friendly manner with an option to resolve and fix it with preview as well.



In the above image you can see that the code assistance is providing option to either implicitly and explicitly implement interface ICalculator, and if we analyze we can clearly say that these are the only possible options that a developer may opt for in this scenario. Moreover it shows the error doe link referring to its description. If you choose first option and choose preview changes link, you'll see following preview and you can choose to apply that if it is what you need.



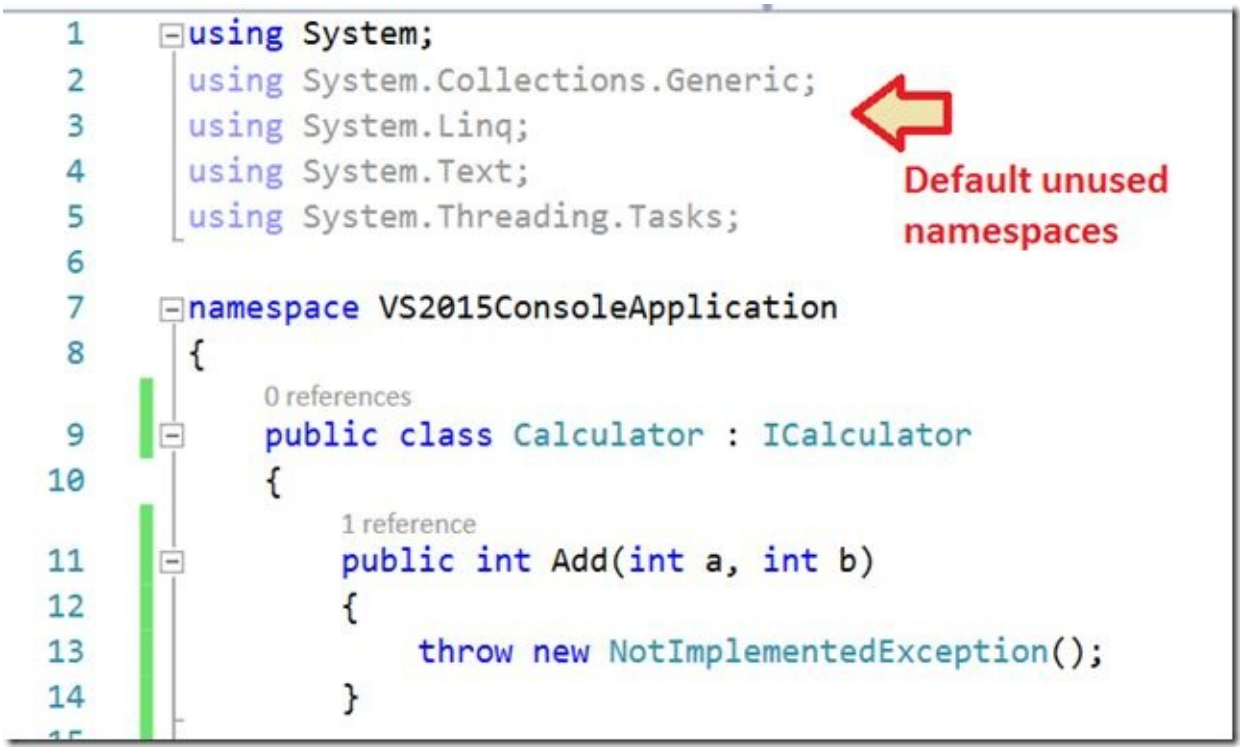
So click apply and we get following class with all the interface methods having default implementations.

```
1: public class Calculator : ICalculator
2: {
3:     public int Add(int a, int b)
4:     {
5:         throw new NotImplementedException();
6:     }
7:
8:     public float Divide(float a, float b)
9:     {
10:        throw new NotImplementedException();
11:    }
12:
13:    public int Multiply(int a, int b)
14:    {
15:        throw new NotImplementedException();
16:    }
17:
18:    public int Subtract(int a, int b)
19:    {
20:        throw new NotImplementedException();
21:    }
22: }
```

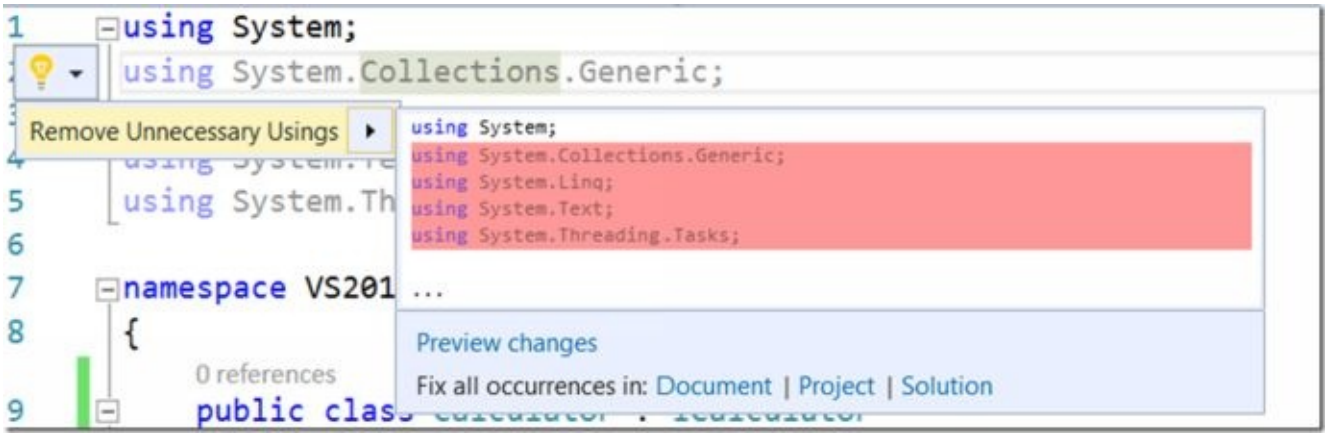
Likewise light bulb icon provides numerous code suggestion options while development and coding following which we can increase the productivity of writing code without unnecessarily compiling the application and writing the code manually.

Refactoring Suggestions

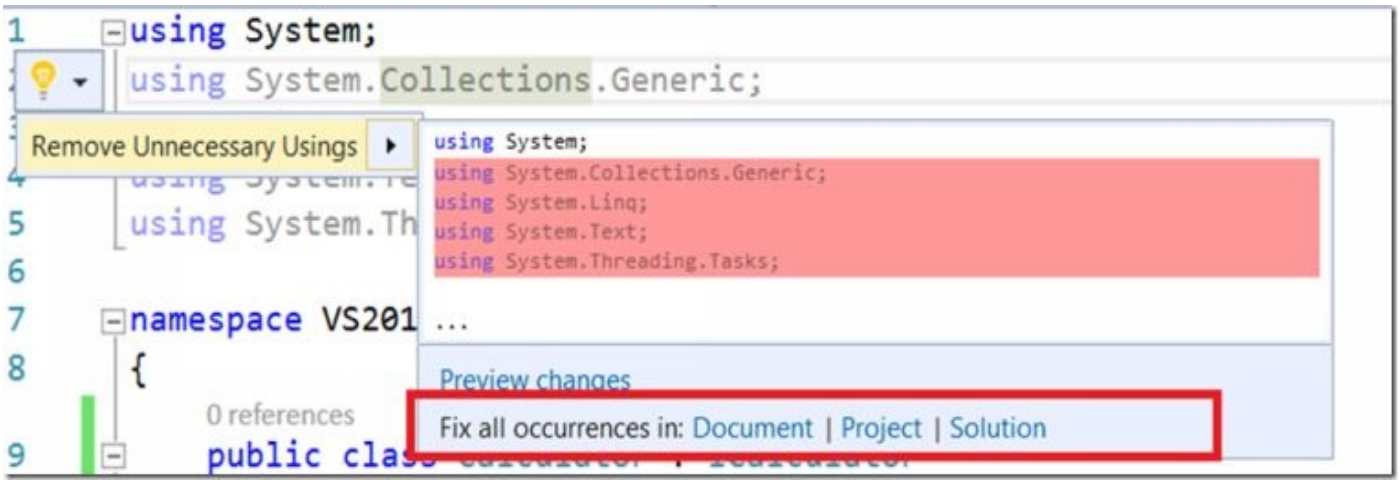
Light bulb icon is not only limited to code suggestions and syntax error suggestions but also comes with a great capability of refactoring techniques. When I added the calculator class, the class was added with few default namespaces as shown below.



In the above scenario as we can see there are few namespaces added by default in the class that are currently not used. When you hover the mouse over those namespaces, light bulb icon shows up with some refactoring suggestions as shown below.



The above image shows the suggestion of Light bulb icon asking to remove “unnecessary usings”. We see here that Visual Studio is smart enough to know what refactoring is required in the code and accordingly can suggest a developer to optimize the code. If you apply the suggestion it will remove the unnecessary “usings” from your code. You can also select to fix all occurrences of this issue in the current document, the project, or the solution. If we just want to make this local change, we can select Remove Unnecessary Usings here, and the unused usings are removed.

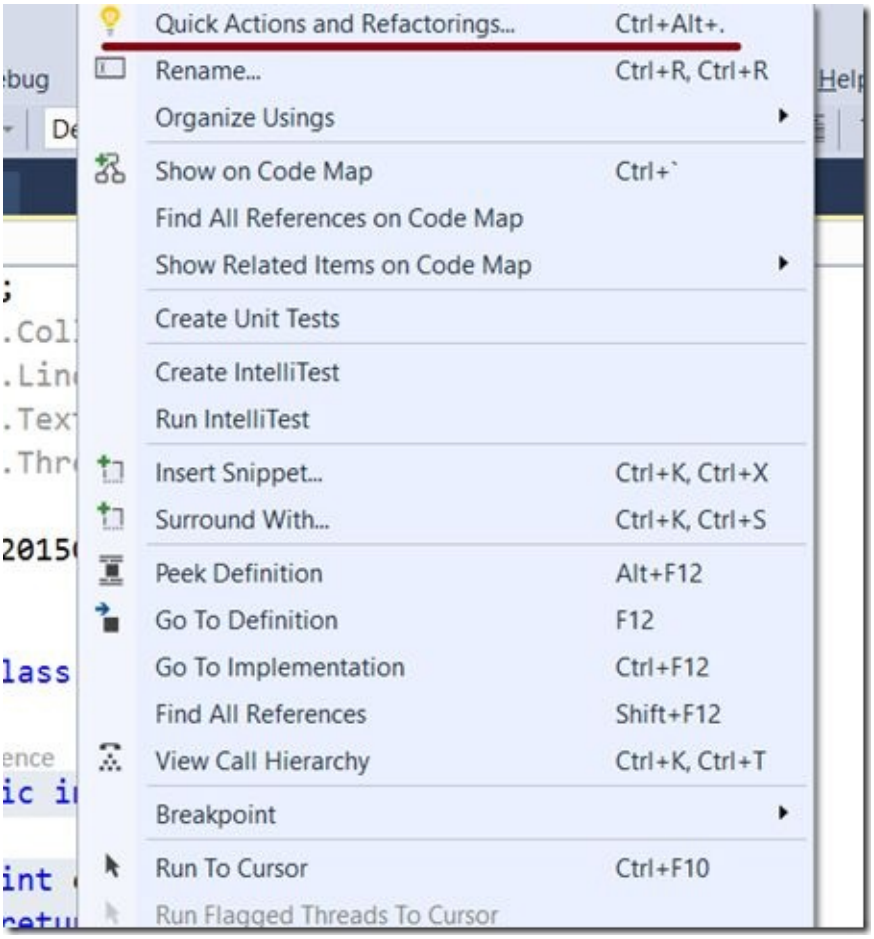

using System;
using System.Collections.Generic;
Remove Unnecessary Usings
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
...
Preview changes
Fix all occurrences in: Document | Project | Solution
using System.Th
namespace VS201
{
0 references
public clas

Quick Suggestions and Refactoring

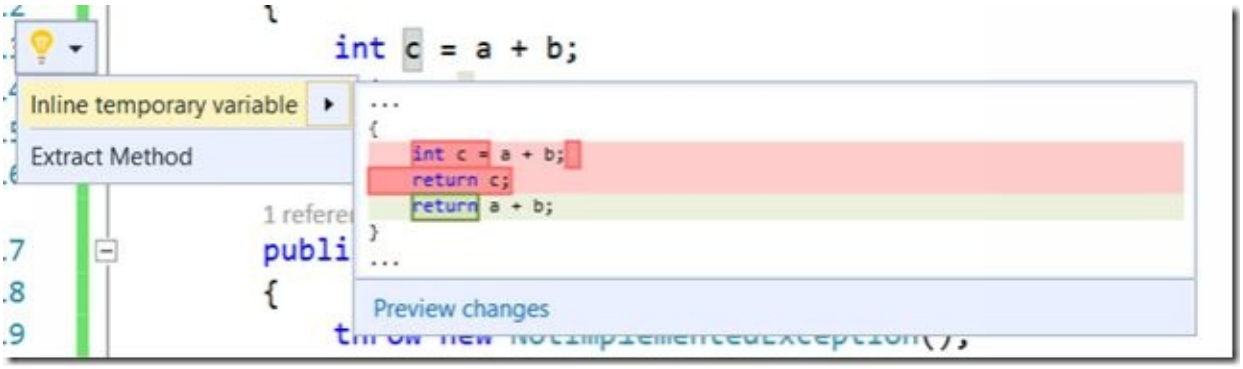
Now when I go to my Calculator.cs class and define the Add method as follows.

```
1: public int Add(int a, int b)
2:     {
3:         int c = a + b;
4:         return c;
5:     }
```

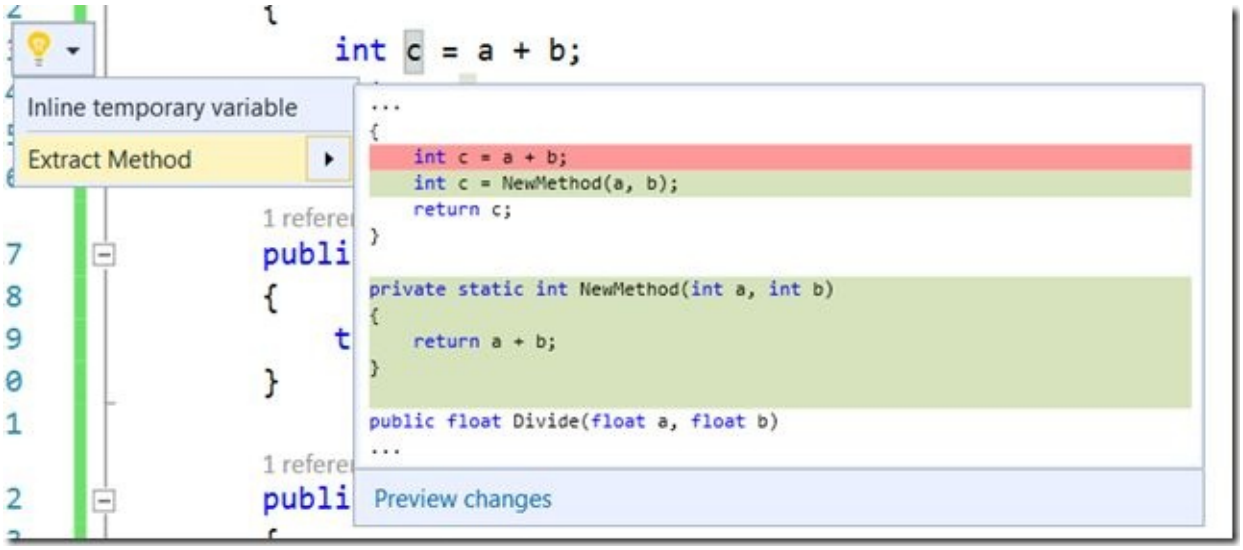
It is the correct way of defining an add method, but on the second thought what if I want to refactor or optimize this method. Visual Studio 2015 provides us facility to do quick refactoring of code with suggestions using an option enabled in context menu of the editor. Just right click on “c” and you’ll get to see an option at the top of the context menu saying “Quick Actions and Refactorings…”.



Note that in above code block, Visual Studio didn’t suggest any thing implicitly and due to some syntax error but we have an option to choose and ask for Visual Studio’s help explicitly to know if a particular written code could be enhanced, optimized, refactored more or not. There could be cases that choosing this option too does not show any suggestion to improve code which means your code is already refactored and optimized. But in the above mentioned scenario if we select the “Quick Actions and Refactorings…” option, VS gives us two options to further optimize the code.



or,



If we have a glance over both the options, the first says to skip using temporary variable and just return (a+b) which is a good suggestion by the way and second option says to extract a method out of the code and put (a+b) in any other method and return from there. Now in these situations its choice of developer on what option he chooses.I choose first option and apply the changes that it showed me in preview and I got following code which looks better that the earlier one.

```
1: public int Add(int a, int b)
2:     {
3:         return a + b;
4:     }
```

Note that these are the small examples that I am taking to just explain the power of Visual Studio 2015. There could be complex and tricky scenarios where you may actually need a lot of help through these features.

Code Analyzers in Visual Studio 2015

In first part of the book we learnt about Visual Studio's capability of code suggestions and code optimizations. In this chapter we'll cover another interesting and very useful feature of Visual Studio 2015 i.e. Live Static Code Analysis. The feature's name in itself is self-explanatory. Visual Studio provides a power to a developer to know about the optimization techniques as well as compile time errors while coding itself i.e. a developer is not supposed to compile the code again and again to know about compile time errors or code optimizations, he can view all of these on the fly while coding and fix it then and there. Let's cover the topic in detail with practical examples.

Live Static Code Analysis

Live Static Code Analysis is not a sentence or a phrase, each and every word has a unique meaning that adds value to the overall feature. Live means you can get the code analysis on the fly while you type your code. You get a real time code check facility without even compiling the code explicitly. Static means analyzing the code when it is not in running state. This feature is intended to save a lot of development time while coding. The code is checked through pre-defined analyzers while you type and you don’t have to build/compile the application again and again. Moreover these analyzers apart from just showing errors also provide certain warning or suggestive messages while coding that improves a developer’s coding skills and enable him to follow best coding practices. These in built analyzers work on specific set of rules, and the code when written is validated while typing against these rules.

Analyzers can be availed via Nuget Packages installed in Visual Studio. There are numerous analyzers available in Visual Studio 2015. In earlier versions of Visual Studio there were a very few of them available, but now you can get analyzers for each and every approach that you use in Visual Studio for development. There are analyzers available for Azure, Unit tests, Exception handling, Mocks, ORM etc. You can choose analyzers as per your need.

The best thing that this feature provides is that one can build our own analyzer against the type of code we want. This means we can define our own set of rules for coding, and these rules will be invoked against the live code that you or any other developer is writing. The rules may include warnings, errors, suggestions, automated code fixes etc. Let us see by some examples on how we can leverage this capability.

Code Analyzers

Create a console application that contains simple arithmetic operations as methods. The name of the console application used in this chapter is VS2015ConsoleApplication. It contains an interface named ICalculator and a class named Calculator that inherits from mentioned interface as shown below.

ICalculator:

```
 1: using System;
 2: using System.Collections.Generic;
 3: using System.Linq;
 4: using System.Text;
 5: using System.Threading.Tasks;
 6:
 7: namespace VS2015ConsoleApplication
 8: {
 9:     interface ICalculator
10:     {
11:         int Add(int a, int b);
12:         int Subtract(int a, int b);
13:         int Multiply(int a, int b);
14:         float Divide(float a, float b);
15:     }
16: }
```

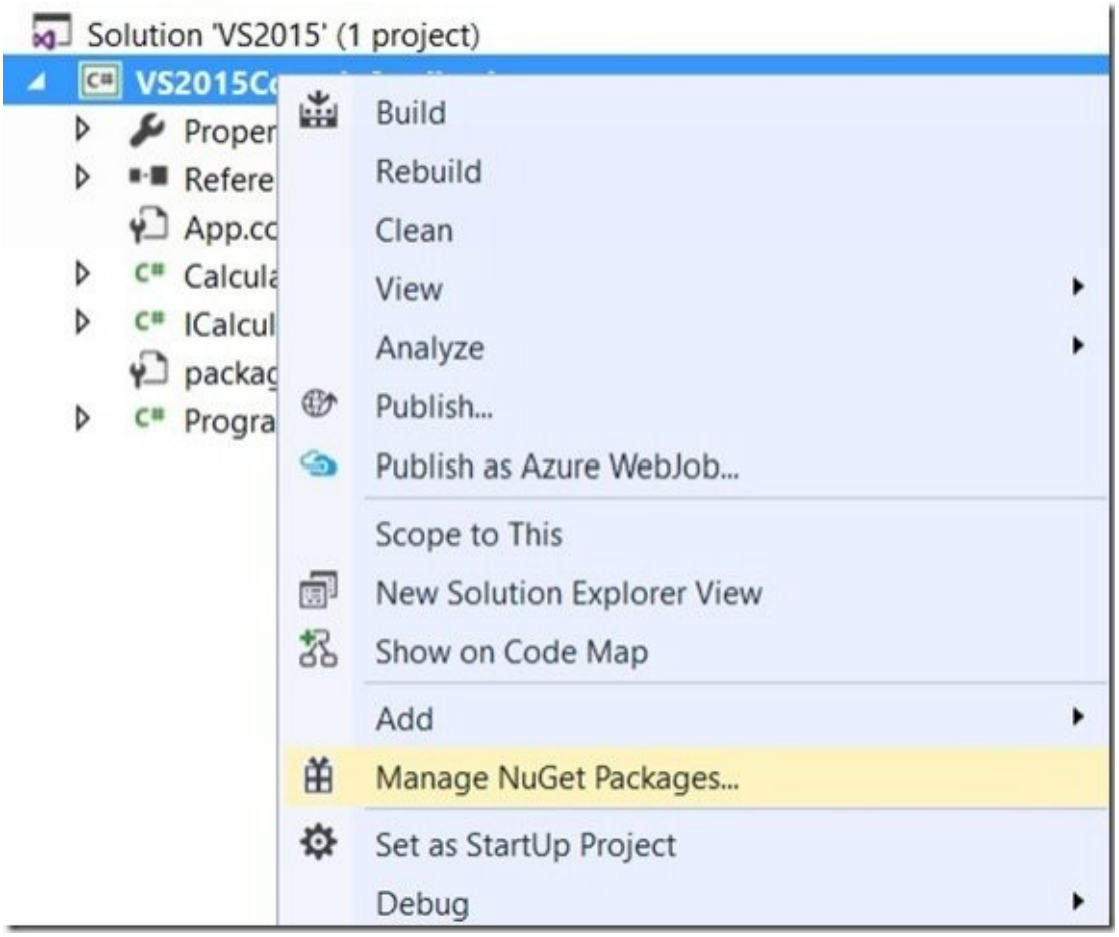
Calculator:

```
 1: using System;
 2: using System.Collections.Generic;
 3: using System.Linq;
 4: using System.Text;
 5: using System.Threading.Tasks;
 6:
 7: namespace VS2015ConsoleApplication
 8: {
 9:     public class Calculator : ICalculator
10:     {
11:         public int Add(int a, int b)
```

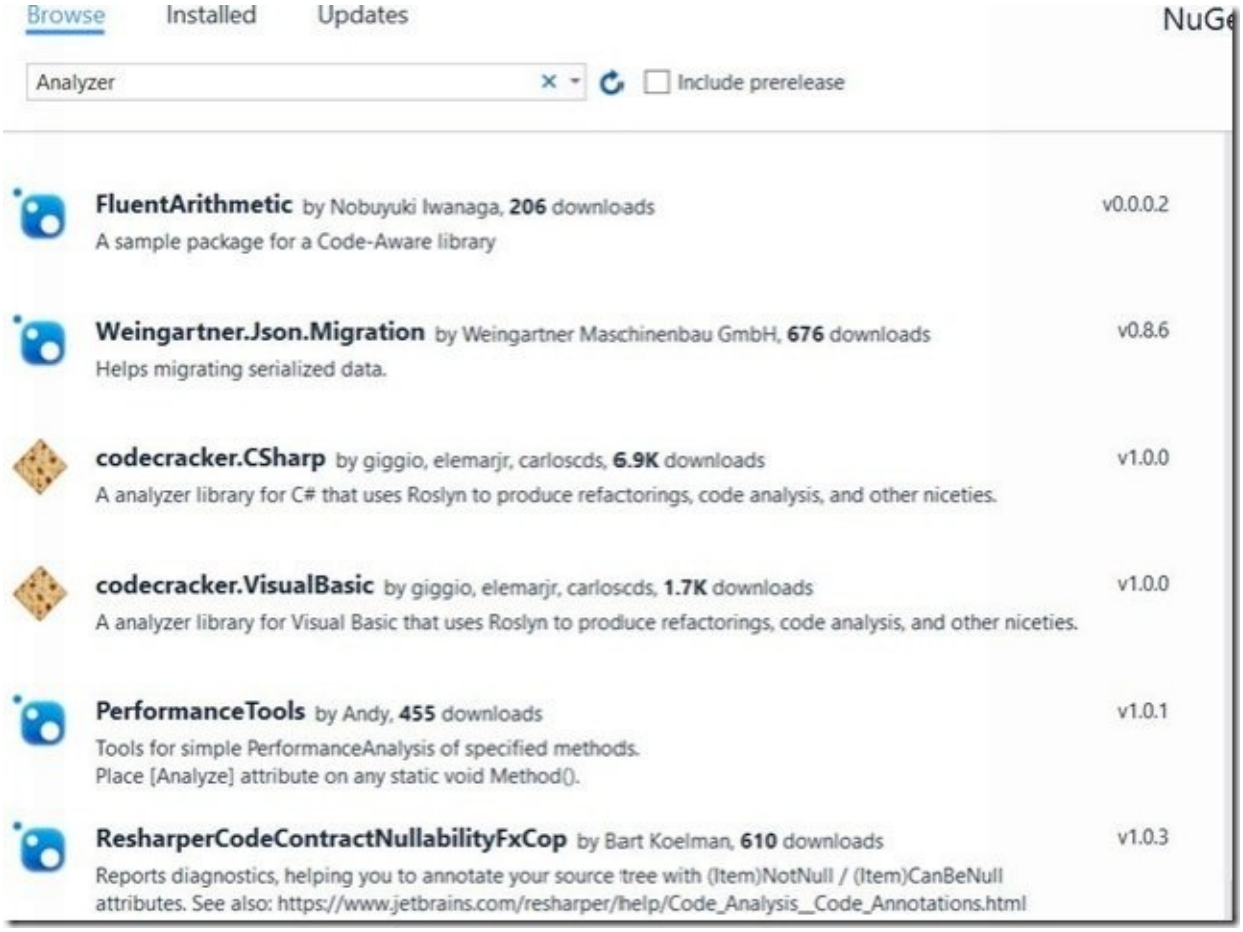
```
12:         {
13:             throw new NotImplementedException();
14:         }
15:
16:         public float Divide(float a, float b)
17:         {
18:             throw new NotImplementedException();
19:         }
20:
21:         public int Multiply(int a, int b)
22:         {
23:             throw new NotImplementedException();
24:
25:         }
26:
27:         public int Subtract(int a, int b)
28:         {
29:             throw new NotImplementedException();
30:         }
31:     }
32: }
33:
```

As one can see there is not enough code. We'll use this code to just check and understand the feature and concept behind Live Static Code Analysis.

Right click on the project file in Visual Studio and click on Manage Nuget Packages… option from the context menu.

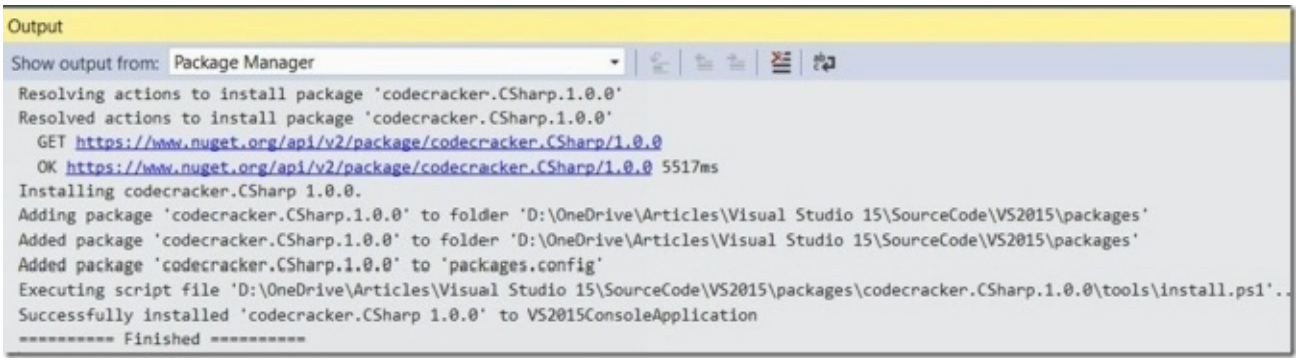


When you click on this option a window will be launched in the IDE where you get an option to check for installed packages, updates for existing packages or browse for new Nuget Packages. Use search box to search for a keyword named “Analyzer” as show below.

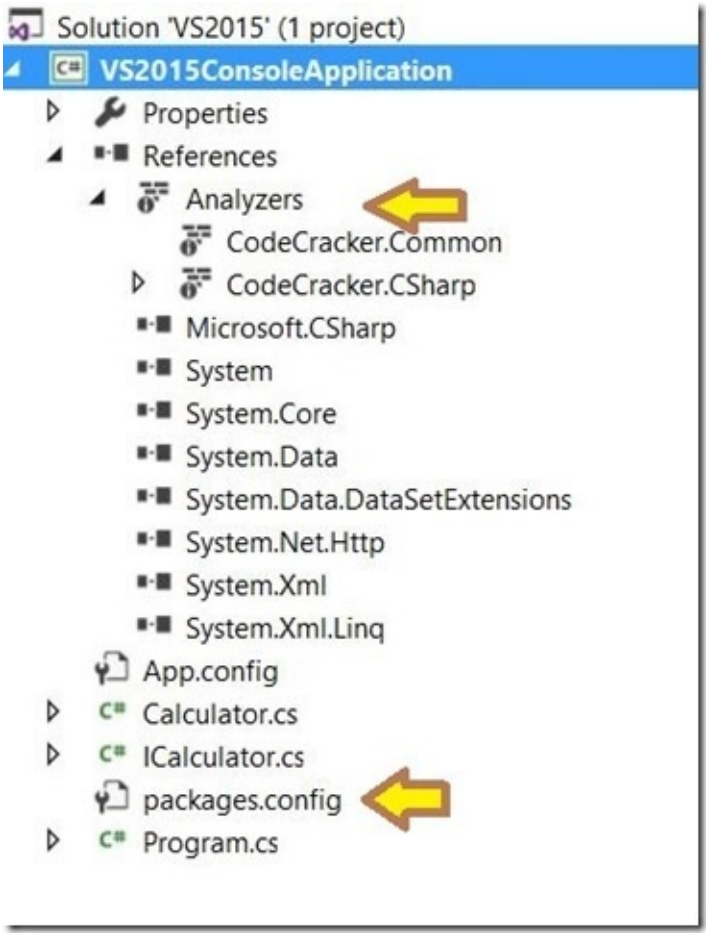


You'll see that a number of code analyzers will appear on the window. When you click them, they show the description in the right side window. Here we get a choice to use whatever analyzer we need to use with our existing project. We'll use one of the analyzers to check its capability. I have chosen "codecracker.CSharp" analyzer for further explanation and detailing. Scroll down to this analyzer and click on codecracker.CSharp and install it by clicking install button as shown in below image.

Once installed, the package manager console in visual studio will show the message for successful installation as shown below.



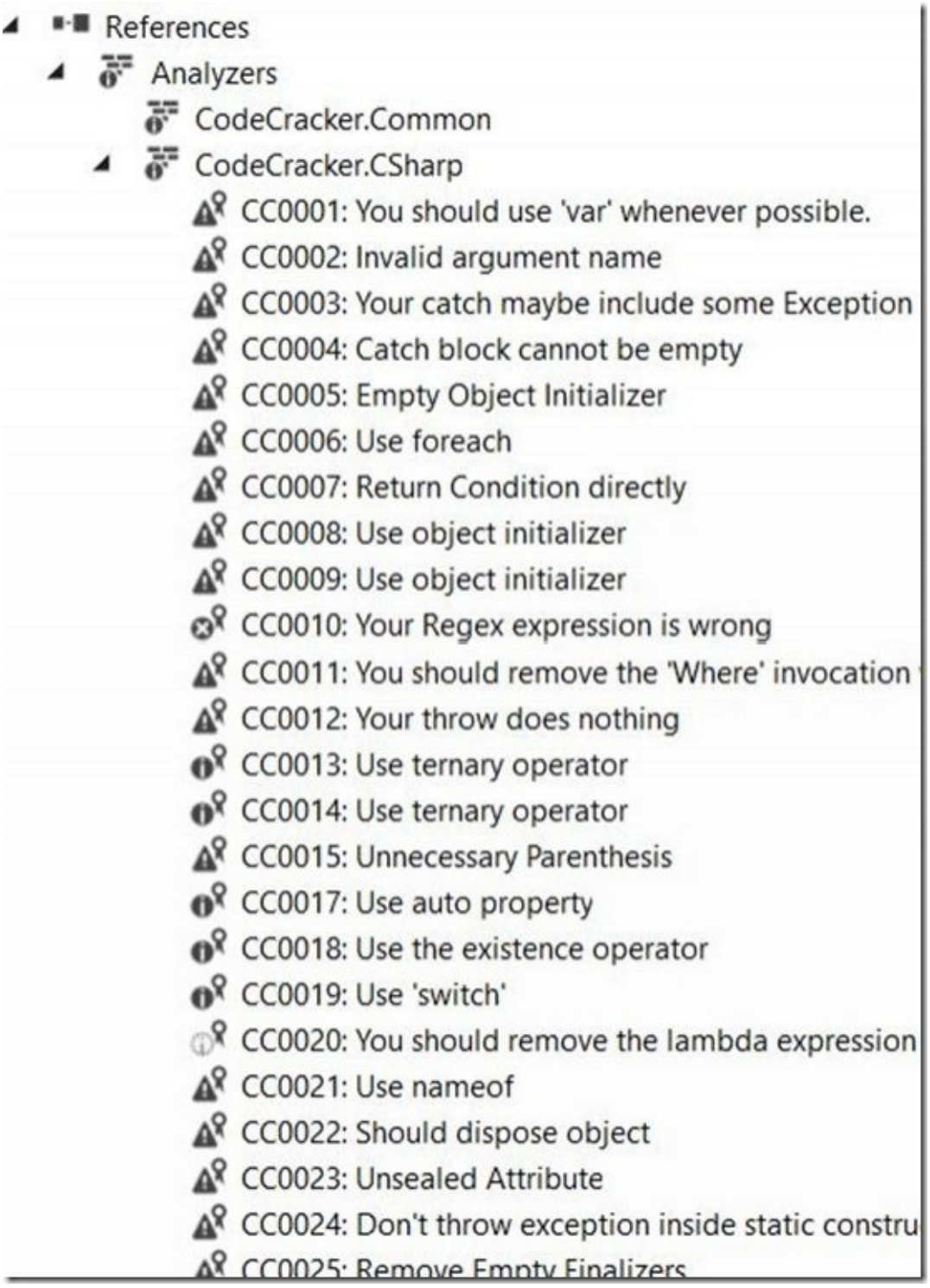
And in Visual Studio , under References a new reference will be named saying "Analyzers->CodeCracker.Common,CodeCracker.CSharp". Analyzers is the root reference where multiple analyzers can lie.



In addition to that a new “packages.config” file will be added having following code that contains package’s information.

```
1: <?xml version=“1.0” encoding=“utf-8”?>
2: <packages>
3:   <package id=“codecracker.CSharp” version=“1.0.0” targetFramework=“net45” />
4: </packages>
```

Since we can see the packages.config and a reference added in the solution that means the analyzer we downloaded from Nuget is successfully installed for the console application project. You can check all set of rules of the chosen analyzer by clicking on the arrow of analyzer under reference in visual Studio. Click on “CodeCracker.CSharp” and get a list of all the messages, info, suggestions, warnings and errors that this set contains.



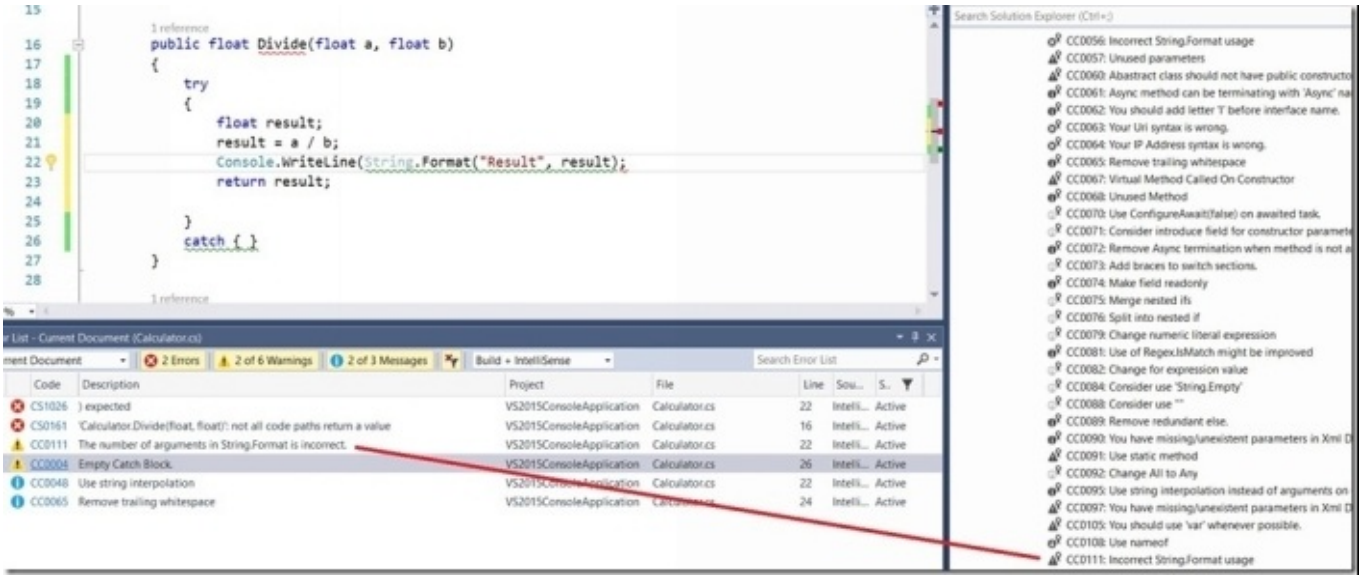
Each rule has a particular syntax like a unique code associated to it, with a descriptive symbol like warning, error, information, suggestion and a descriptive text. Like for an example if we check the first rule, it signifies that it is a sort of warning/suggestion with unique code CC0001 having a description as “You should use ‘var’ whenever possible”. This means that while coding if any such situation occurs where this rule violates, we’ll be shown a message like that with a warning symbol.

Code Analysis

Let us write some code to check how analyzers work. Following is some code I have written to implement Divide method in Calculator class.

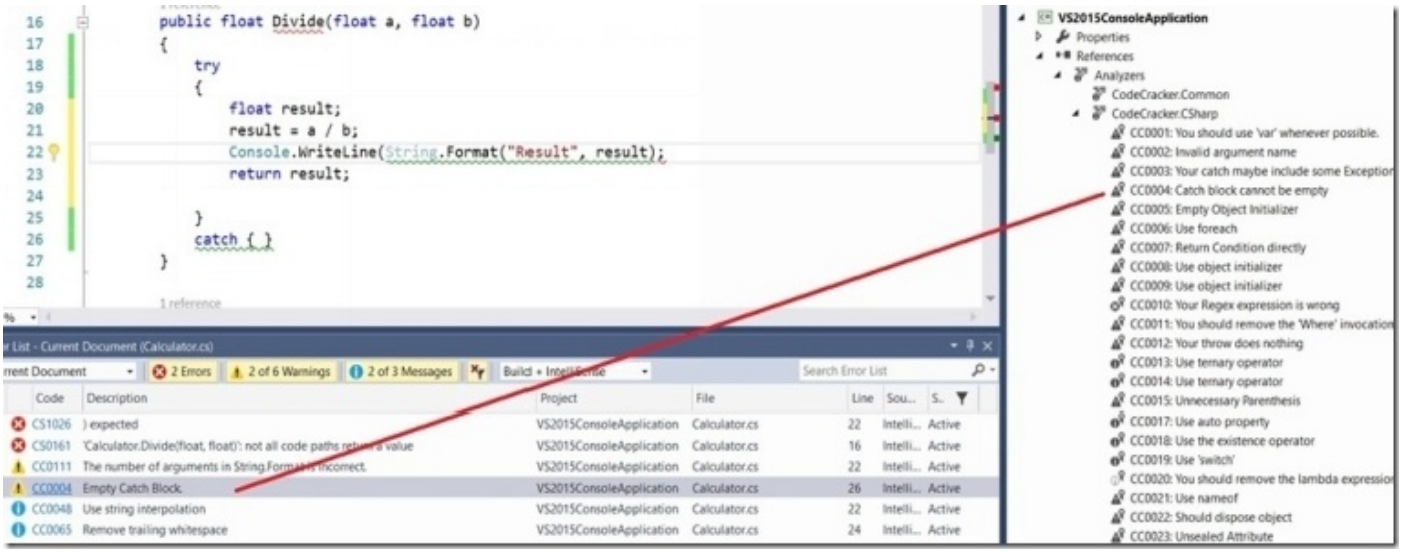
```
1: public float Divide(float a, float b)
2:     {
3:         try
4:         {
5:             float result;
6:             result = a / b;
7:             Console.WriteLine(String.Format("Results", result));
8:             return result;
9:
10:         }
11:
12:         catch { }
13:
14:     }
```

The above mentioned code block is purposely written in a way that may result in some compile time error. While writing the code, keep the Error List window open. You'll notice that while writing the code, whenever you violate any of the Code Analyzer rule, it immediately flashes in Error List window. It shows that you are getting the feedback and optimizations options on the fly while coding, and you do not have to depend upon compiling the code. For example, check the following image. I have written the code and till now not compiled the project.



We see that two warnings are shown that means while writing the code we broke two rules of installed code analyzer. The first one says it is warning with code CC0111 with a description that number of arguments for "String.Format" method is incorrect. I purposely wrote the code in this way to explain how code analyzers work. As we can see and compare that the rule shown to us in Error List window matches exactly to that of shown in Code Analyzer's set of rules list at right side. This proves the code analyzer is working

as desired. Here we see that we need not have to compile the application and as soon as we fix the code, the error/warning of code analyzer is gone. I purposely left catch block empty so that the code analyzer may validate the code against its defined rule for catch block and suggest us. It clearly says CC0004 Empty catch Block rule is violated.

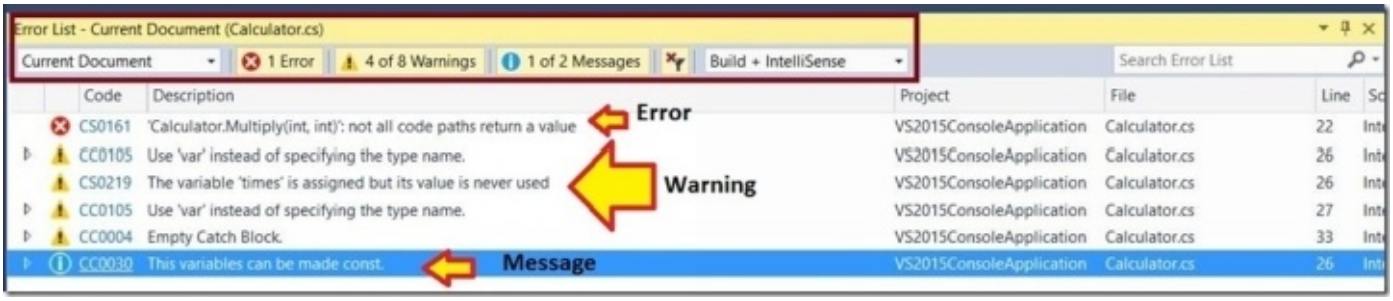


Case Study

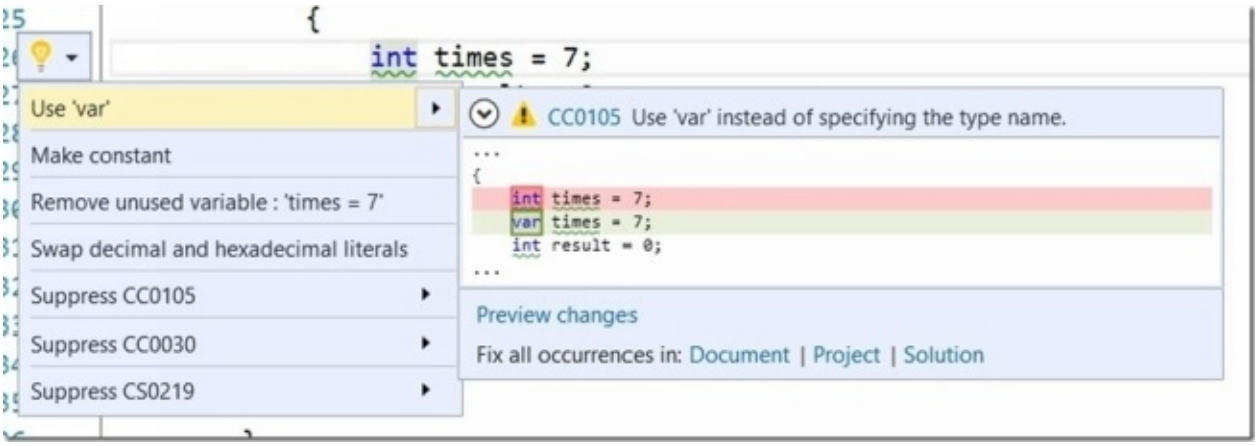
Let us take another scenario and see how analyzers help in providing constructive suggestions w.r.t. code fix or optimization. The following code implements “Multiply” method, there are few errors, some un-optimized code in this method.

```
1: public int Multiply(int a, int b)
2: {
3:     try
4:     {
5:         int times = 7;
6:         int result = 0;
7:         result = a * b;
8:         return result;
9:
10:    }
11:
12:    catch { }
13:
14:
15: }
```

While writing this code I got following messages, warnings and errors in Error List window.

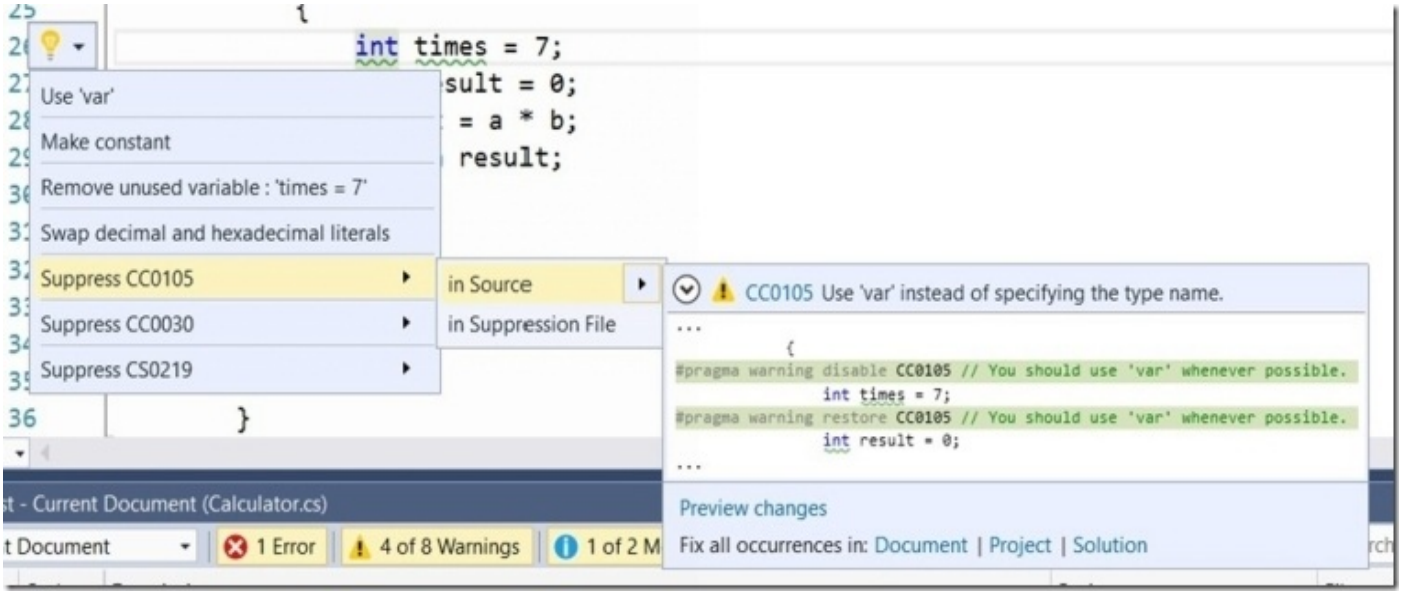


Note that the Error List window here is configured for Current Document. You can change that to Current Project or all open documents as per your requirement. It shows 1 error, 4 warnings and 1 message. The error is displayed by compiler but the warnings and message is displayed through analyzer that we have used in this project. The warnings are self-explanatory. if you don’t know how to fix them, the light bulb icon helps you to fix those. For CC0105 i.e. use var instead of specifying a type name, if you click on **int times=7;** the light bulb icon will be shown with few suggestions as shown below.



We see here that Light bulb icon is smart enough to take care of analyzer's suggestions and provide constructive workarounds to fix the code. The first suggestion is use 'var'. You can have a preview before you actually apply it, and it's your choice whether you want to make this fix in the opened document, project or solution. Another suggestion that Light bulb icon gives is for making the variable as constant. It suggests so because it is smart enough to analyze that the variable is having a fixed value throughout the method. Since the variable is not used anywhere in the method, it suggests to remove that unused variable therefore optimizing the code.

There are other suggestions like Suppress CC0105, CC0030, CS 0219. Light bulb icon provides you an option for what to do with similar kind of warnings and errors in future. In case you do not want a suggestion to be shown for a particular analyzer warning or message, then you can opt to suppress that message which means it will not be shown again.



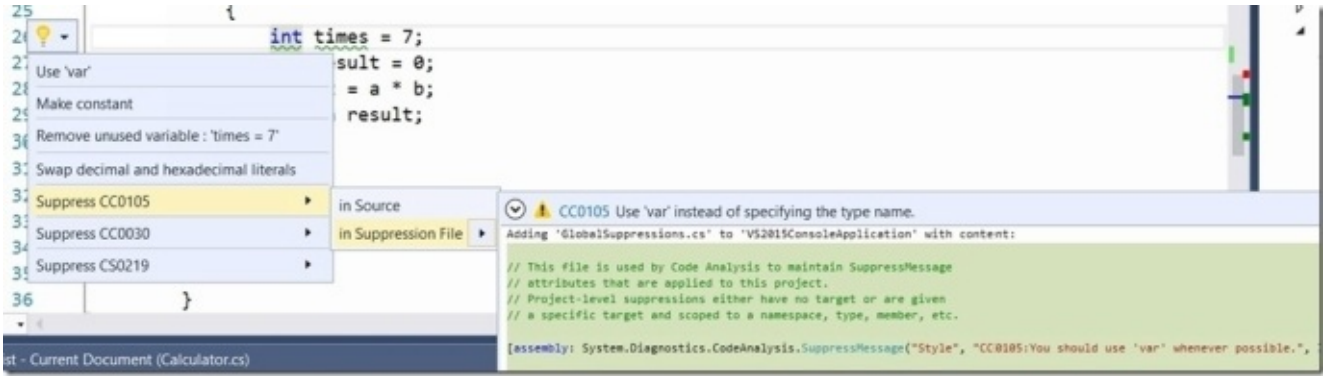
There are two ways in which you can perform suppression and Code suggestion gives you option to choose any of two with a preview.

1. **In Source.** You can suppress the warning/message/error for that particular line of code as shown in above image. In this case a #pragma statement will encapsulate the underlying code and you'll not be shown the message again. The code becomes like shown below.

```
1: public int Multiply(int a, int b)
2:     {
3:         try
```

```
4:         {
5: pragma warning disable CC0105 // You should use ‘var’ whenever possible.
6:             int times = 7;
7: pragma warning restore CC0105 // You should use ‘var’ whenever possible.
8:             int result = 0;
9:             result = a * b;
10:            return result;
11:
12:        }
```

2. **In suppression file**. Another option is to put that suppression in a separate file, which means the suppression is at project level and you’ll not be shown the message/warning for this rule throughout the code in this particular project. This option creates a separate class file and adds to your project.



The file is named GlobalSuppressions.cs and contains following format for the suppressed messages.

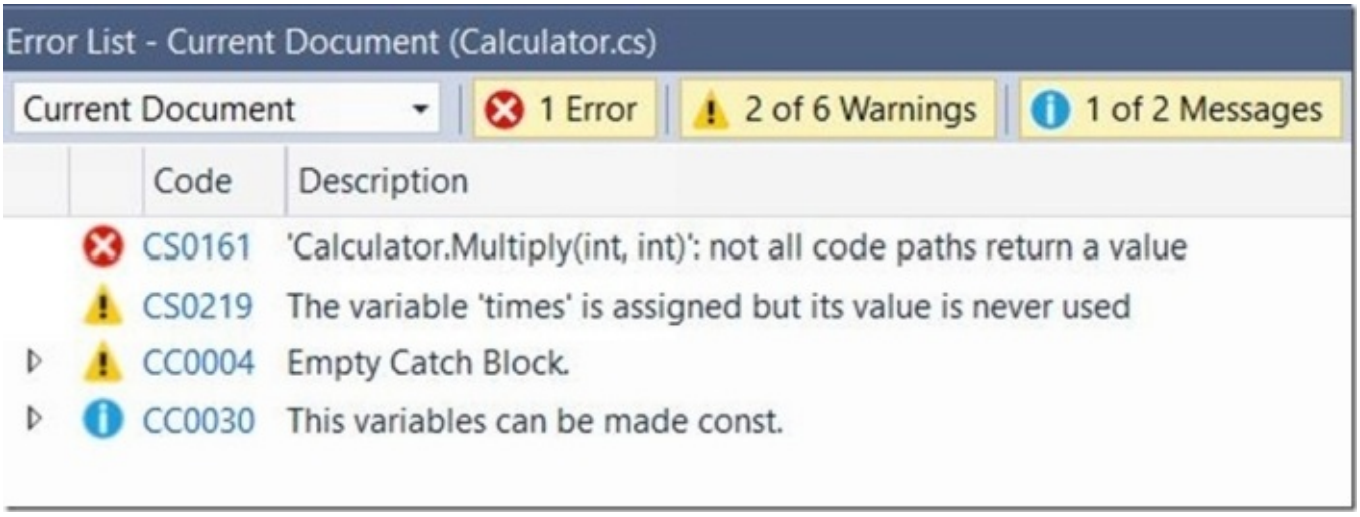
```
1:
2: // This file is used by Code Analysis to maintain SuppressMessage
3: // attributes that are applied to this project.
4: // Project-level suppressions either have no target or are given
5: // a specific target and scoped to a namespace, type, member, etc.
6:
7: [assembly: System.Diagnostics.CodeAnalysis.SuppressMessage(“Style”, “CC0105:You should use ‘var’ whenever possible.”, Justification = “<Pending>”, Scope = “member”, Target = “~M:VS2015ConsoleApplication.Calculator.Multiply(System.Int32,System.Int32)~System.Int32”)]
8:
```

In this particular example I am not opting to suppress the message, instead I’ll go for fixing as per code analyzer’s recommendations. Let us make a fix for CC0105 first.

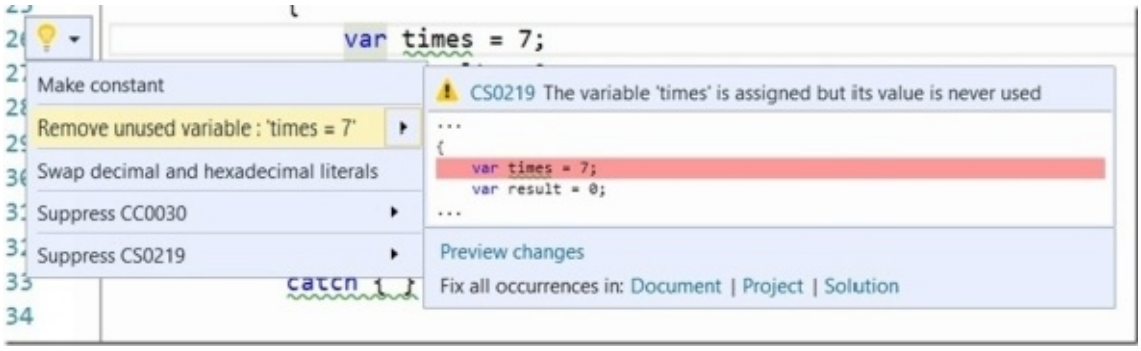
```
1: public int Multiply(int a, int b)
2:     {
3:         try
4:         {
5:             var times = 7;
6:             var result = 0;
7:             result = a * b;
8:             return result;
9:
10:        }
```

```
11:
12:         catch { }
13:
14:
15:     }
```

Therefore Error List window becomes as shown below.



We see here that the first warning has been taken care of and now not shown here. Now fix the CS0219.

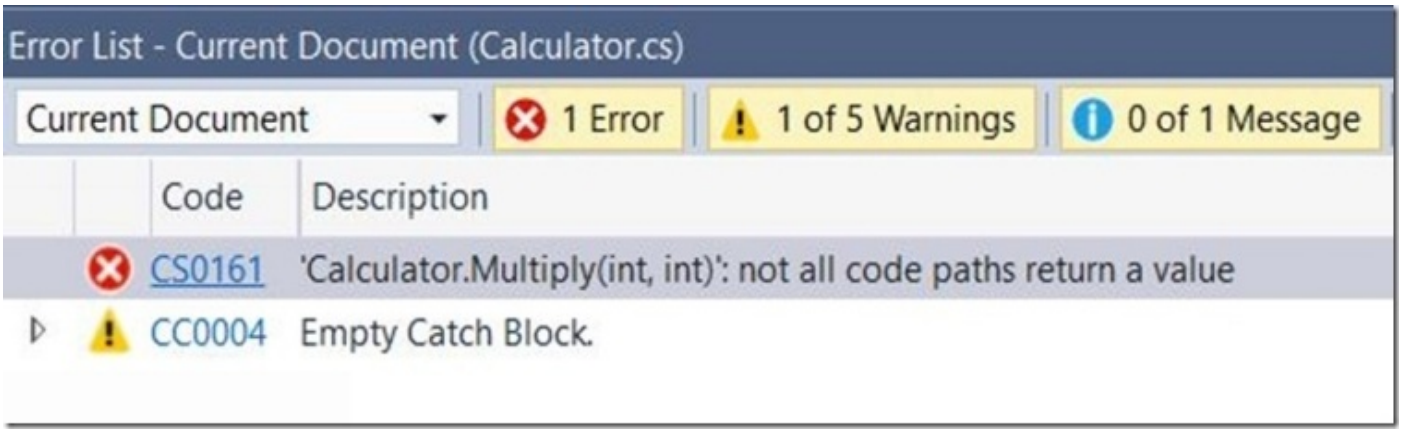


Since we are not using the variable I'll go for removing the unused variable as suggested by Light Bulb Icon. Preview the changes and apply it to code. This will remove the times variable from the code. now we are left with following code.

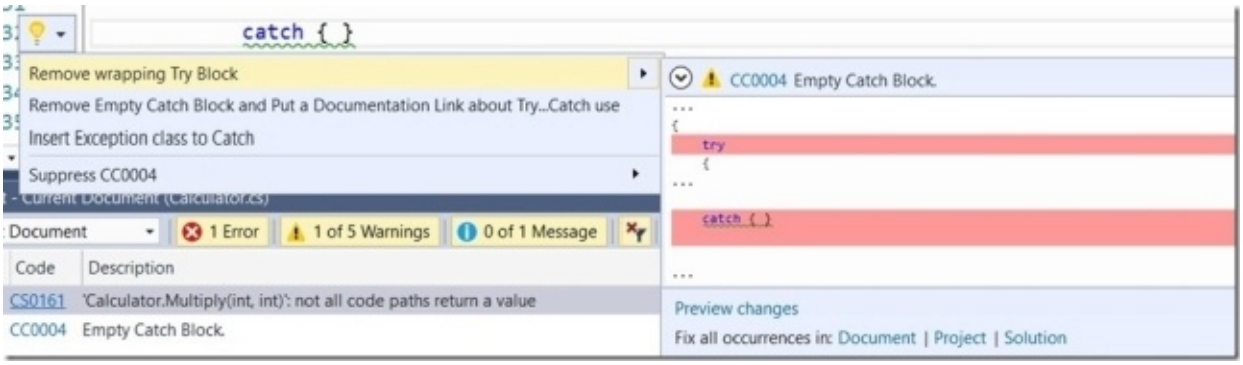
```
1: public int Multiply(int a, int b)
2:     {
3:         try
4:         {
5:             var result = 0;
6:             result = a * b;
7:             return result;
8:
9:         }
10:
11:         catch { }
12:
13:
```

```
14:     }
```

And the following Error List which shows one error and one is warning. The warning is regarding empty catch block that we are using i.e. CC0004 marked in analyzer's rule list.



Again when you click on Light bulb icon that shows while you click on catch block code , you'll see multiple code suggestion options to fix this code as shown below.



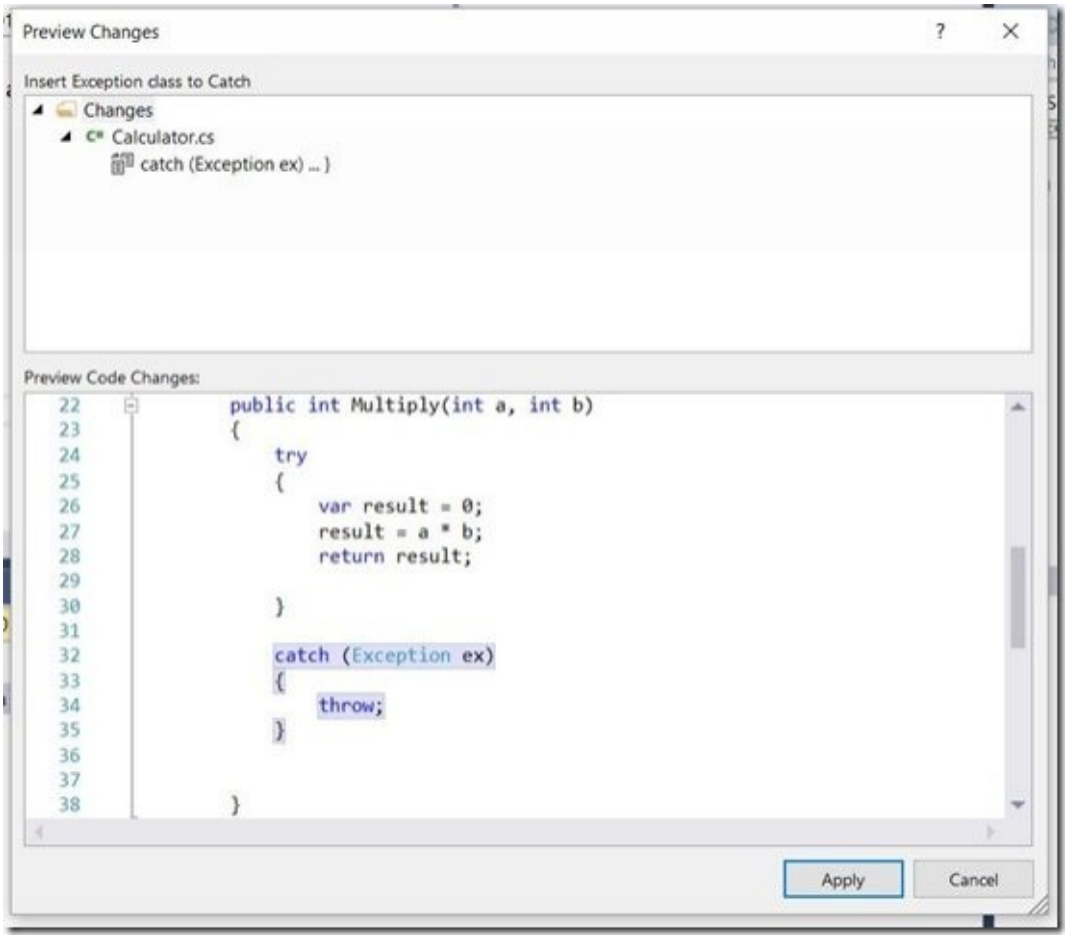
There are various options you can choose for your empty catch block code like 1. Removing the try catch block. You can choose this option if you have nothing to do with exceptions or maybe you accidently applied try catch block. If you go for option 2 i.e. Remove Empty catch Block and put a Documentation…, this option will remove the try-catch block and mark a TODO item there. If you preview the option and apply the fix, you'll get following code for this method.

```
1: public int Multiply(int a, int b)
2:     {
3:         {
4:             var result = 0;
5:             result = a * b;
6:             return result;
7:
8:         }
9:         //TODO: Consider reading MSDN Documentation about how to use Try…Catch => http://msdn.microsoft.com/en-us/library/0yd65esw.aspx
```

The method puts a commented TODO item stating to read MSDN documentation to read about how to use try-catch.

We have already seen how to suppress a message and warning, so let us try the third option i.e. Insert Exception class to catch. Note that you don't have to explicitly write catch block unless you have some custom requirement in the case. Just choose the third

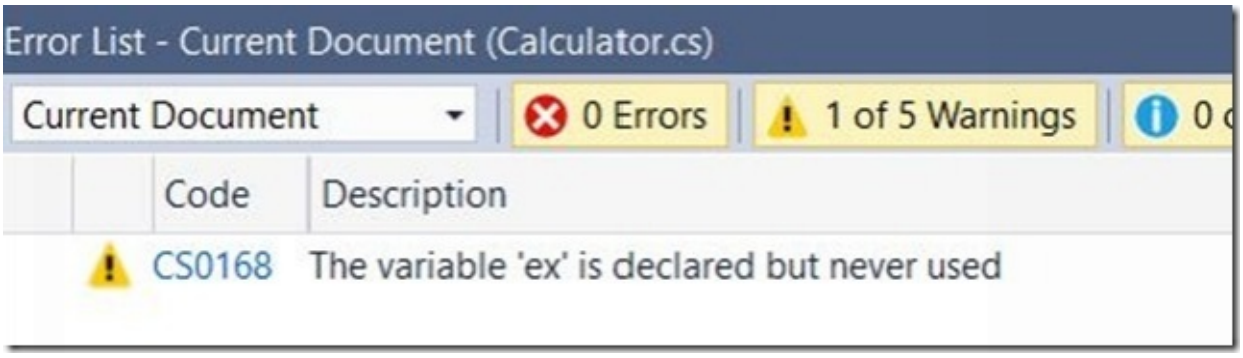
option and preview changes.



You can see the option will insert implementation for catch block with Exception class object as a parameter and throw as exception in case catch block gets invoked. Therefore code becomes as follows.

```
1: public int Multiply(int a, int b)
2:     {
3:         try
4:         {
5:             var result = 0;
6:             result = a * b;
7:             return result;
8:
9:         }
10:
11:        catch (Exception ex)
12:        {
13:            throw;
14:        }
```

But as soon as you make this change, you'll see that error that was shown earlier has been taken care of because now there is return at every exit point of method but a new warning is introduced as shown below.



This image shows that we are trying to declare a variable in catch block that we are not using. Again you can take help from light bulb icon.

There are two options, either remove the variable or suppress the message. Alternatively you can also apply your coding skills and does something with that variable in the code block itself, like displaying error message, logging etc. I'll choose first option i.e. remove the unused variable. Immediately after choosing that option the variable gets removed and our code becomes cleaner as shown below with absolutely no error, warnings or messages.

```
1: public int Multiply(int a, int b)
2: {
3:     try
4:     {
5:         var result = 0;
6:         result = a * b;
7:         return result;
8:     }
9:     catch (Exception)
10:    {
11:        throw;
12:    }
13: }
```

Conclusion

In this part we covered another great feature of Visual Studio 2015 i.e. Live Static Code Analysis. I explained a case study in detail but you may explore other rules and code scenarios while you play around with this feature. I again want to mention that you can try this feature and explore its capability of helping in code analysis and optimization and that too on the fly while you write without even compiling the code. You can try out with more code analyzers available on Nuget. In the next part I'll cover new enhanced renaming and refactoring features that Visual Studio 2015 provides.

Renaming Assistance in Visual Studio 2015

In this part of the book on learning Visual Studio 2015 we'll cover topic named renaming in Visual Studio 2015. Yes Visual Studio 2015 provides a great capability of refactoring/renaming the code. It helps developer to optimize and refactor the code as per the development or best practices need. This feature enables developer to follow best practices by giving refactoring suggestions as well as helping in fixing and optimizing the code. Renaming the variables, methods, properties, classes or even projects has always been a challenge to a developer when working on large code base. Another challenge that most of the developers face is w.r.t. code comments and writing an optimized method. Visual Studio 2015 helps in code refactoring and renaming as well in a very easy and friendly way.

Code Renaming

There are a lot of features that this capability of renaming covers in visual Studio 2015.Change preview, inline and modeless rename windows, renaming on the fly, detecting and resolving conflicts, renaming code comments are few of them. Let us discuss each one in detail via practical examples. I am using Visual Studio 2015 Enterprise edition and have created a console application named VS2015ConsoleApplication having an interface named IProducts and a class named MyProducts that implements the interface. IProducts contains two methods, one to return product based on product code and another to return complete list of products. Product.cs is another class containing Product entity. This class acts as a transfer object or entity. We'll use Main method of Program.cs class to invoke methods of MyProducts class.

IProducts

```
 1: using System;
 2: using System.Collections.Generic;
 3: using System.Linq;
 4: using System.Text;
 5: using System.Threading.Tasks;
 6:
 7: namespace VS2015ConsoleApplication
 8: {
 9:     interface IProducts
10:     {
11:         Product GetProduct(string productCode);
12:         List<Product> GetProductList();
13:     }
14: }
```

Product

```
 1: using System;
 2: using System.Collections.Generic;
 3: using System.Linq;
 4: using System.Text;
 5: using System.Threading.Tasks;
 6:
 7: namespace VS2015ConsoleApplication
 8: {
 9:     public class Product
10:     {
11:         public string ProductName { get; set; }
12:         public string ProductCode { get; set; }
13:         public string ProductPrice { get; set; }
14:         public string ProductType { get; set; }
15:         public string ProductDescription { get; set; }
```

```
16:
17:     }
18: }
```

MyProducts

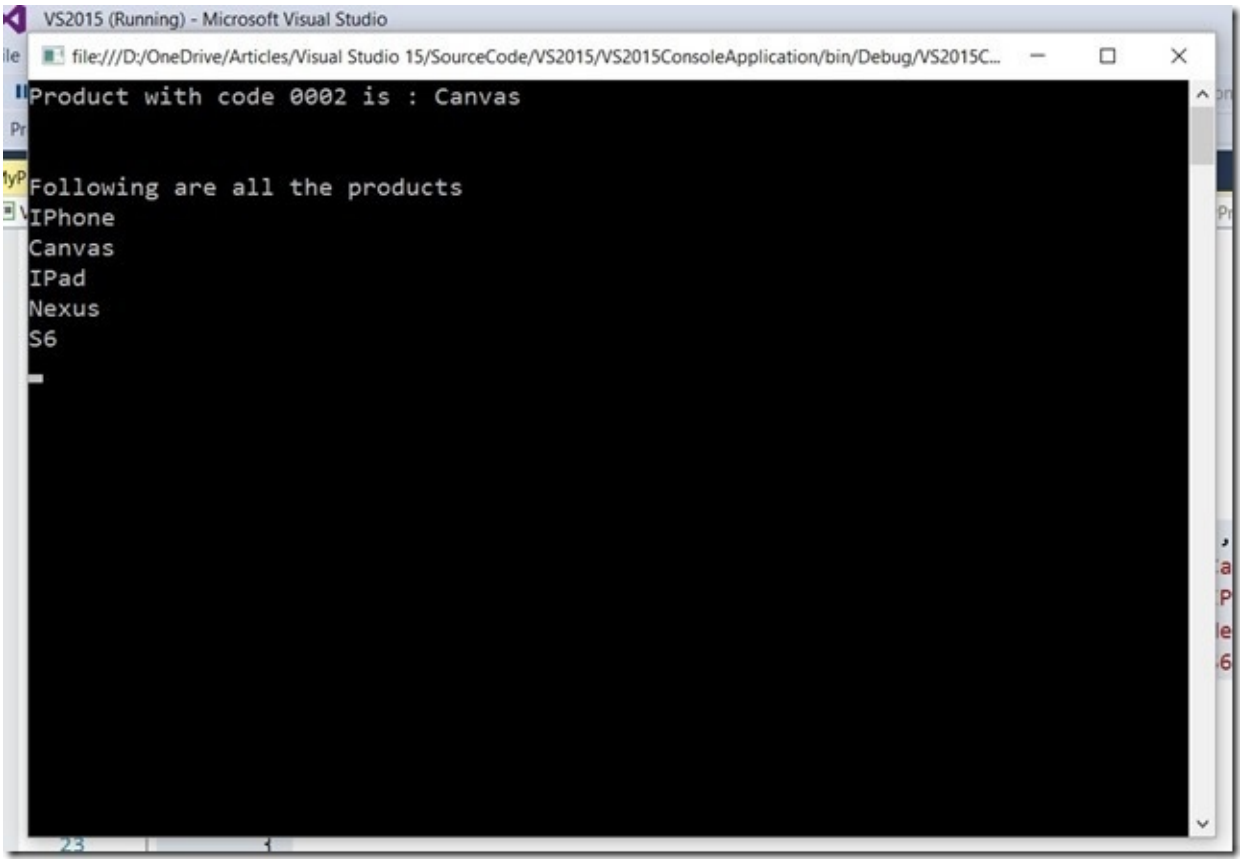
```
1: using System;
2: using System.Collections.Generic;
3: using System.Linq;
4: using System.Text;
5: using System.Threading.Tasks;
6:
7: namespace VS2015ConsoleApplication
8: {
9:     public class MyProducts :IProducts
10:     {
11:         List<Product> _productList = new List<Product>();
12:         public MyProducts()
13:         {
14:             _productList.Add(new Product {ProductCode="0001",ProductName="IPhone",ProductPrice="60000",ProductType="Phone",ProductDescription="Apple IPhone" } );
15:             _productList.Add(new Product { ProductCode = "0002", ProductName = "Canvas", ProductPrice = "20000", ProductType = "Phone", ProductDescription = "Micromax phone" });
16:             _productList.Add(new Product { ProductCode = "0003", ProductName = "IPad", ProductPrice = "30000", ProductType = "Tab", ProductDescription = "Apple IPad" });
17:             _productList.Add(new Product { ProductCode = "0004", ProductName = "Nexus", ProductPrice = "30000", ProductType = "Phone", ProductDescription = "Google Phone" });
18:             _productList.Add(new Product { ProductCode = "0005", ProductName = "S6", ProductPrice = "40000", ProductType = "Phone", ProductDescription = "Samsung phone" });
19:
20:         }
21:
22:         public Product GetProduct(string productCode)
23:         {
24:             return _productList.Find(p => p.ProductCode == productCode);
25:         }
26:
27:         public List<Product> GetProductList()
28:         {
29:             return _productList;
30:         }
31:     }
32: }
```

Program.cs

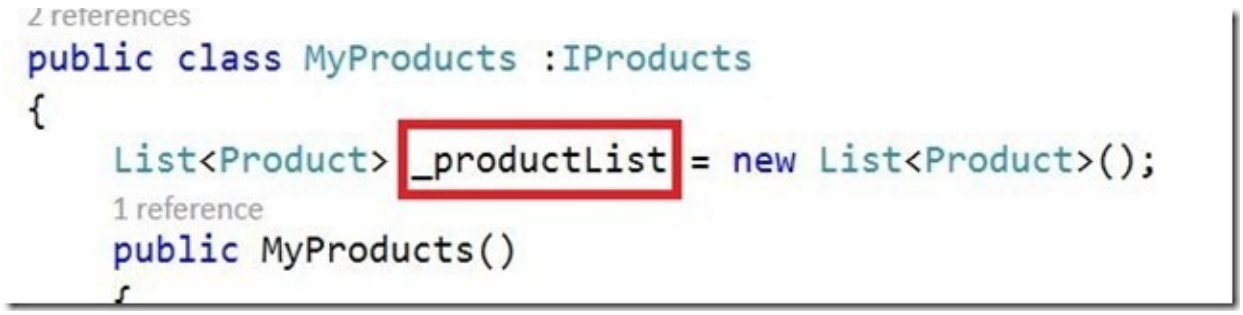
```
1: using System;
2: using System.Collections.Generic;
3: using System.Linq;
```

```
4: using System.Text;
5: using System.Threading.Tasks;
6:
7: namespace VS2015ConsoleApplication
8: {
9:     class Program
10:    {
11:        static void Main()
12:        {
13:            var myProducts = new MyProducts();
14:            Console.WriteLine( String.Format("Product with code 0002 is : {0}", myProducts.GetProduct("0002").ProductName));
15:            Console.WriteLine(Environment.NewLine);
16:            var productList = myProducts.GetProductList();
17:            Console.WriteLine("Following are all the products");
18:
19:            foreach (var product in productList)
20:            {
21:                Console.WriteLine(product.ProductName);
22:            }
23:            Console.ReadLine();
24:        }
25:    }
26: }
```

Our code is ready. Program file's main method calls both the interface methods from MyProducts class to test the functionality.



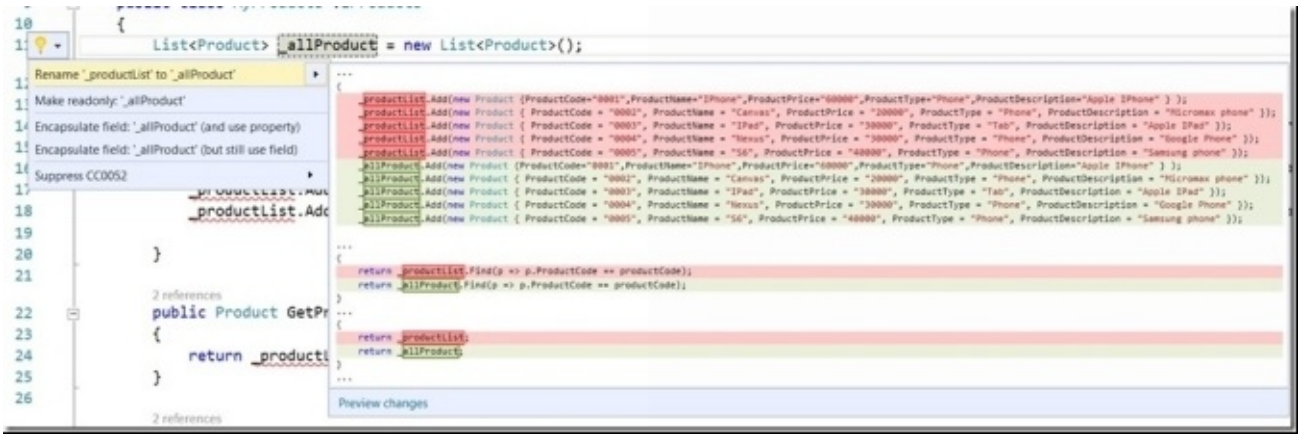
We'll use this code base to learn code renaming features. The renaming experience may vary w.r.t. what actual operation is being performed. For example in MyProducts class we are using _productList as a variable containing list of all products, and this variable is widely used throughout the class.



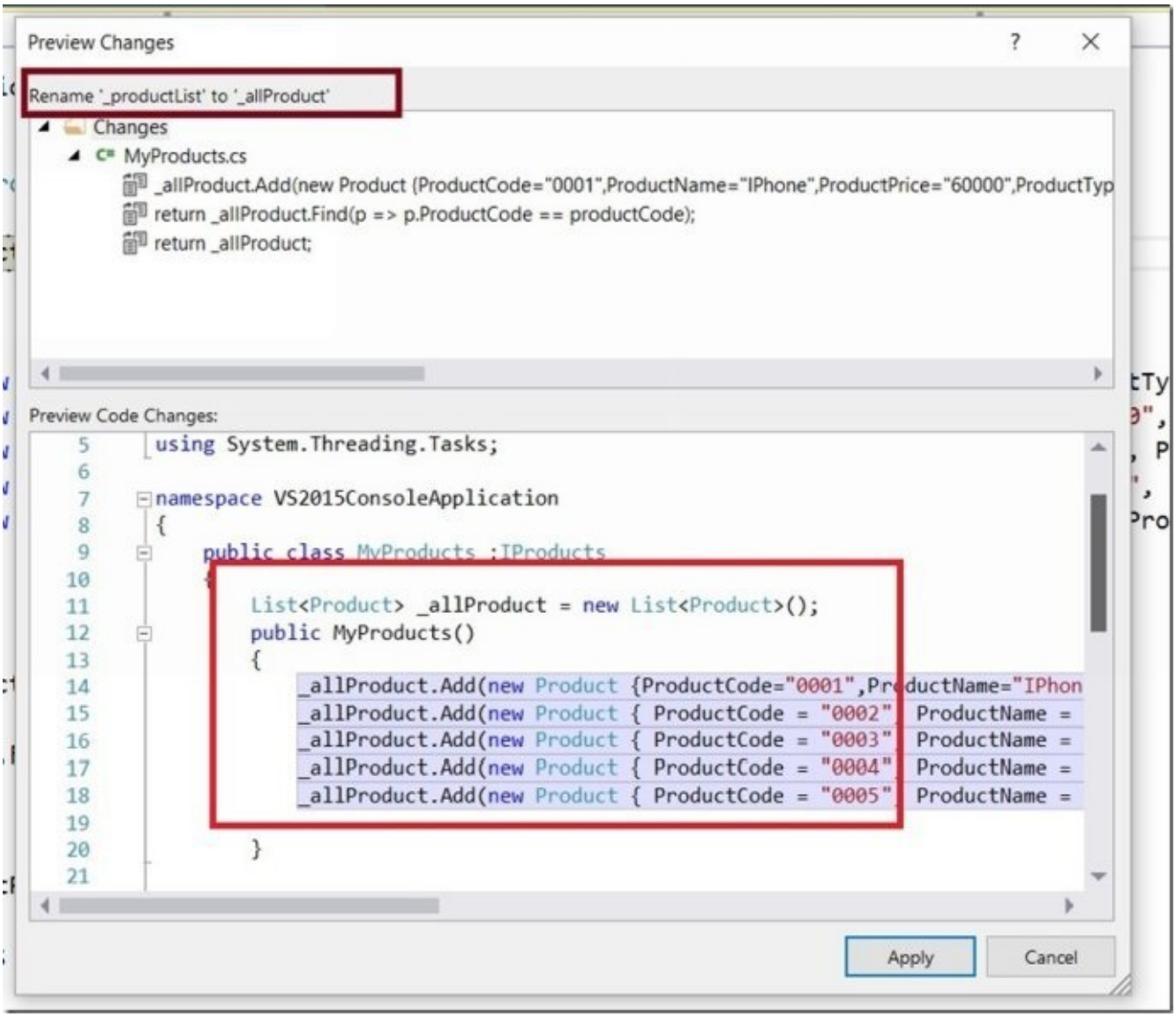
Let us try to rename this variable to a new name called _allProducts.

```
2 references
public class MyProducts :IProducts
{
    List<Product> _allProduct = new List<Product>();
    1 reference
    public MyProducts()
    {
        _productList.Add(new Product {ProductCode="0001",Pro
        _productList.Add(new Product { ProductCode = "0002",
        _productList.Add(new Product { ProductCode = "0003",
        _productList.Add(new Product { ProductCode = "0004",
        _productList.Add(new Product { ProductCode = "0005",

    }

    2 references
    public Product GetProduct(string productCode)
    {
        return _productList.Find(p => p.ProductCode == produ
    }

    2 references
    public List<Product> GetProductList()
    {
        return _productList;
    }
}
}
```

Notice that as soon as _productsList is changed to _allProduct, a dotted box is shown around the changed variable name and a light bulb icon appears at the left. One can see one more change, that all the instances where _productList was used has a red line below them. This means that all the instances should be changed accordingly. You can see here the power of Visual Studio. Visual studio is smart enough to understand and communicate that the change that a developer is making has to be reflected to other places too. Let us see what Light bulb icon says. Let us click on light bulb icon appeared at the left margin.



There is one new suggestion that Light bulb icon is showing now and that is regarding “Rename”. In the suggestion window Light bulb icon shows all the _productList instances highlighted and says to rename all the instances to new name i.e _allProducts. When you preview the changes by clicking “Preview Changes ” link shown at the bottom of suggestion window, it shows the resultant changes that you’ll get once variable is renamed.

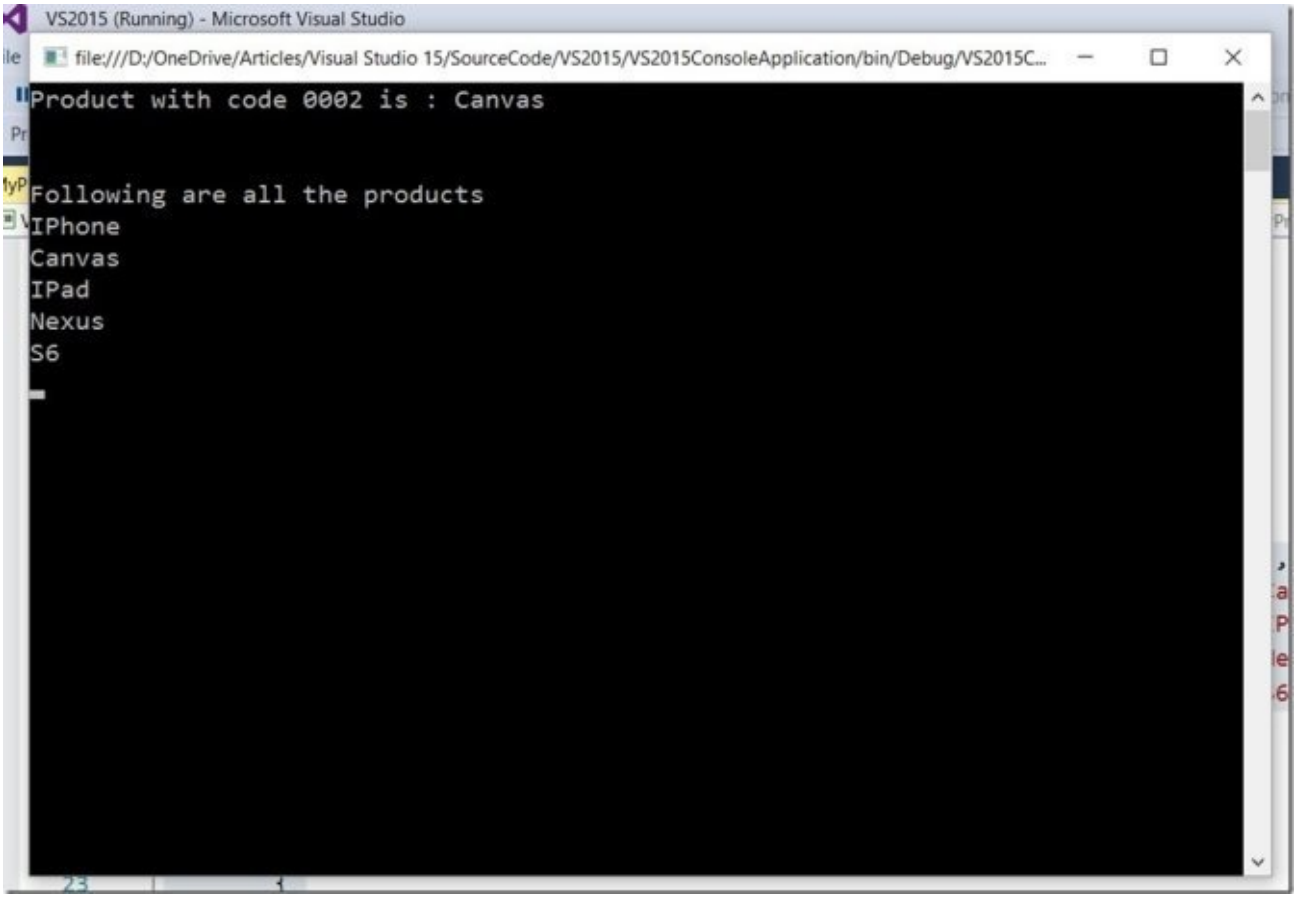


In preview you can clearly see the changes that will take place once the variable is renamed. Let us click on apply changes, and you'll see the method has a new changed variable now at all the instances.

```
1: using System;
2: using System.Collections.Generic;
3: using System.Linq;
4: using System.Text;
5: using System.Threading.Tasks;
6:
7: namespace VS2015ConsoleApplication
8: {
9:     public class MyProducts :IProducts
10:    {
11:        List<Product> _allProduct = new List<Product>();
12:        public MyProducts()
13:        {
14:            _allProduct.Add(new Product {ProductCode="0001",ProductName="IPhone",ProductPrice="60000",ProductType="Phone",ProductDescription="Apple IPhone" } );
15:            _allProduct.Add(new Product { ProductCode = "0002", ProductName = "Canvas", ProductPrice = "20000", ProductType = "Phone", ProductDescription = "Micromax phone" });
```

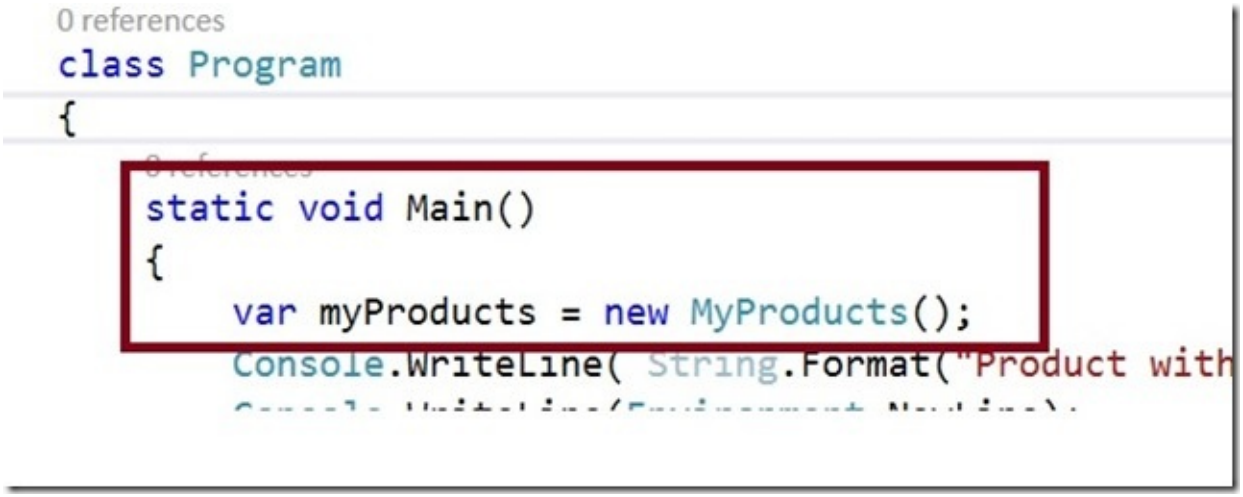
```
16:             _allProduct.Add(new Product { ProductCode = “0003”, ProductName = “IPad”, ProductPrice = “30000”, ProductType = “Tab”, ProductDescription = “Apple IPad” });
17:             _allProduct.Add(new Product { ProductCode = “0004”, ProductName = “Nexus”, ProductPrice = “30000”, ProductType = “Phone”, ProductDescription = “Google Phone” });
18:             _allProduct.Add(new Product { ProductCode = “0005”, ProductName = “S6”, ProductPrice = “40000”, ProductType = “Phone”, ProductDescription = “Samsung phone” });
19:
20:         }
21:
22:         public Product GetProduct(string productCode)
23:         {
24:             return _allProduct.Find(p => p.ProductCode == productCode);
25:         }
26:
27:         public List<Product> GetProductList()
28:         {
29:             return _allProduct;
30:         }
31:     }
32: }
```

Now let’s run the application to check if the changes had an impact on the functionality of the application. Press F5.

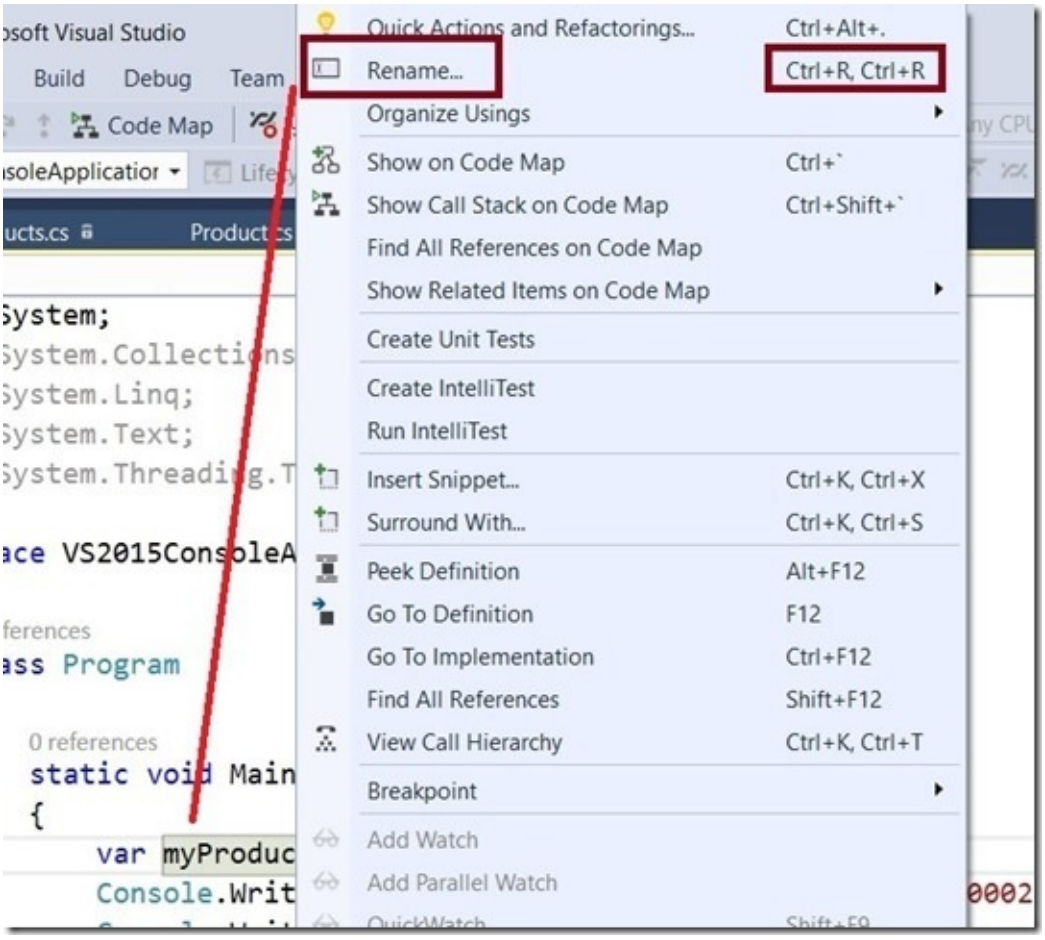


It is clearly seen that application has no build error and produces the same result as earlier. There is one thing that a developer needs to take care of, sometimes renaming may be risky and tricky in large code base, so while renaming it is suggested to have a glance over preview i.e. given by suggestion window.

Let us take another scenario.Open the Program.cs file and perform rename on myProducts object.

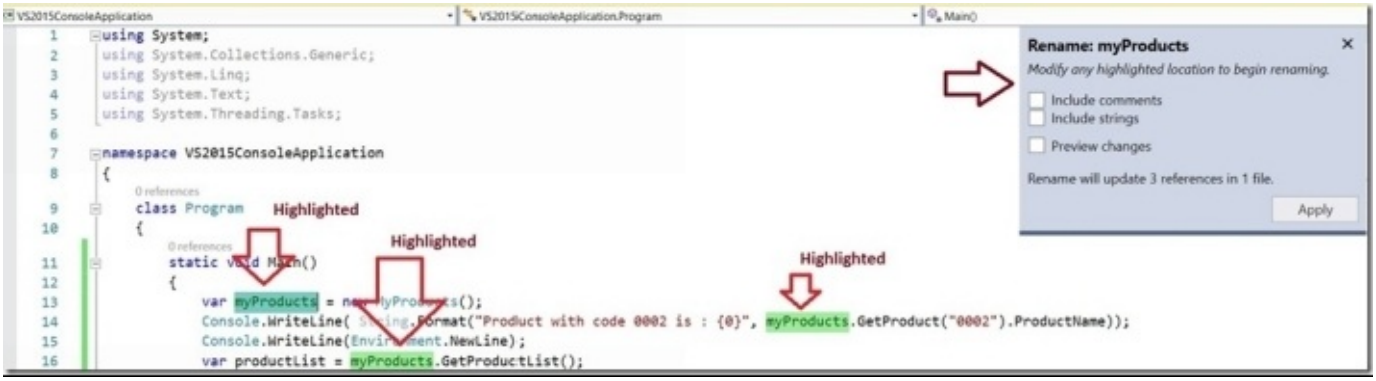


This time we'll do renaming through context menu. Right click on myProducts object name and you'll see the context menu open. There is an option of rename on the opened context menu.



In front of Rename option, there is a shortcut available too Ctrl+R, Ctrl+R. When you select Rename, you'll notice that the renaming experience here is different , all the instances of the renaming gets highlighted. There is a new window that appears at the right corner of code window having few other options. Now if you rename the object name, the other highlighted variables get renamed on the fly while typing. To close the Rename window you can click cross button on that window. Notice that this option of renaming through context menu is much faster than the earlier one, and enables you to

rename on the fly in a single step with live preview while typing. Once the variable is renamed and you close the window, all the highlighting will be gone and you get a clean renamed variable at all the occurrences.

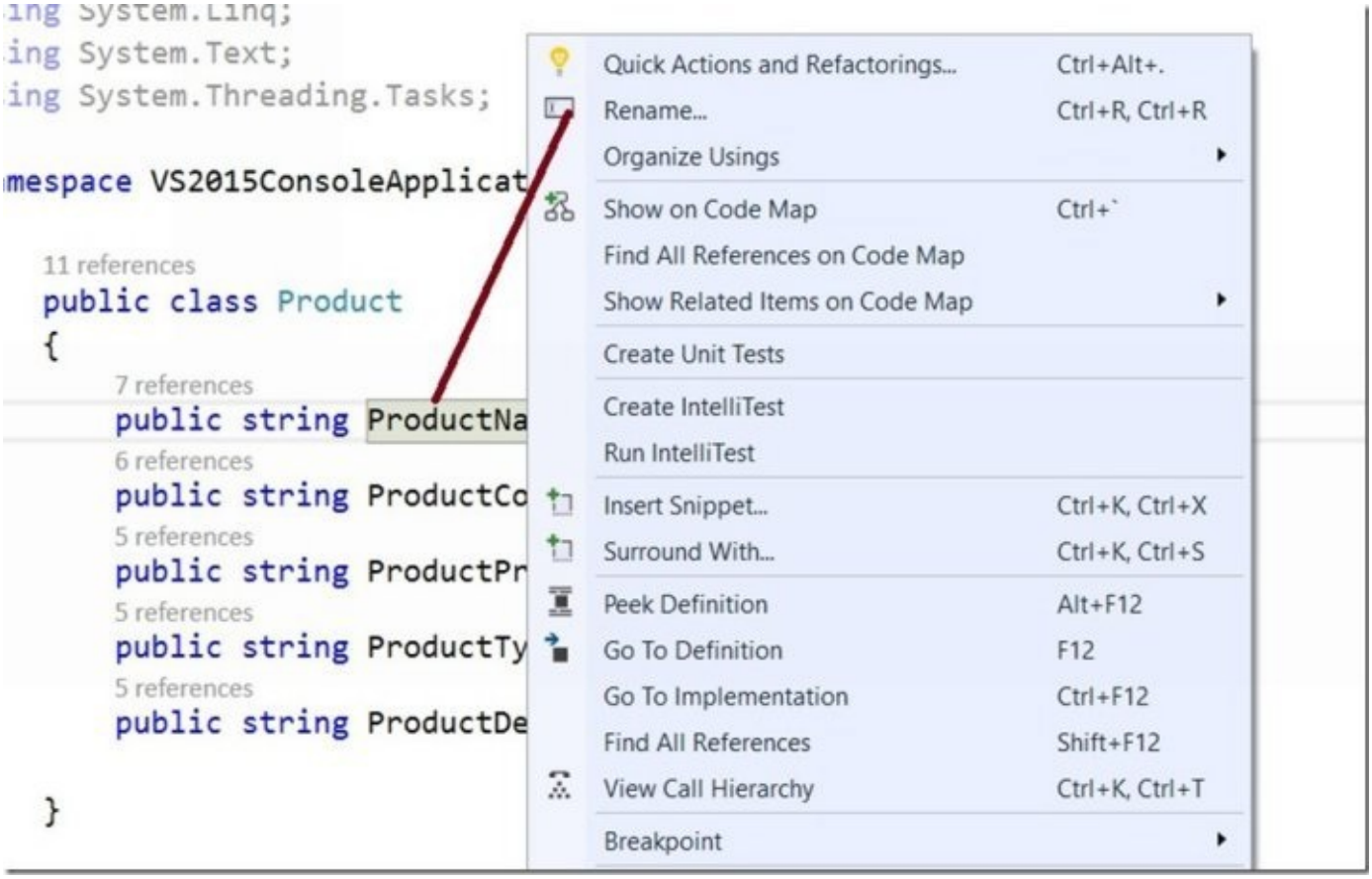


I changed the object name from myProducts to products.

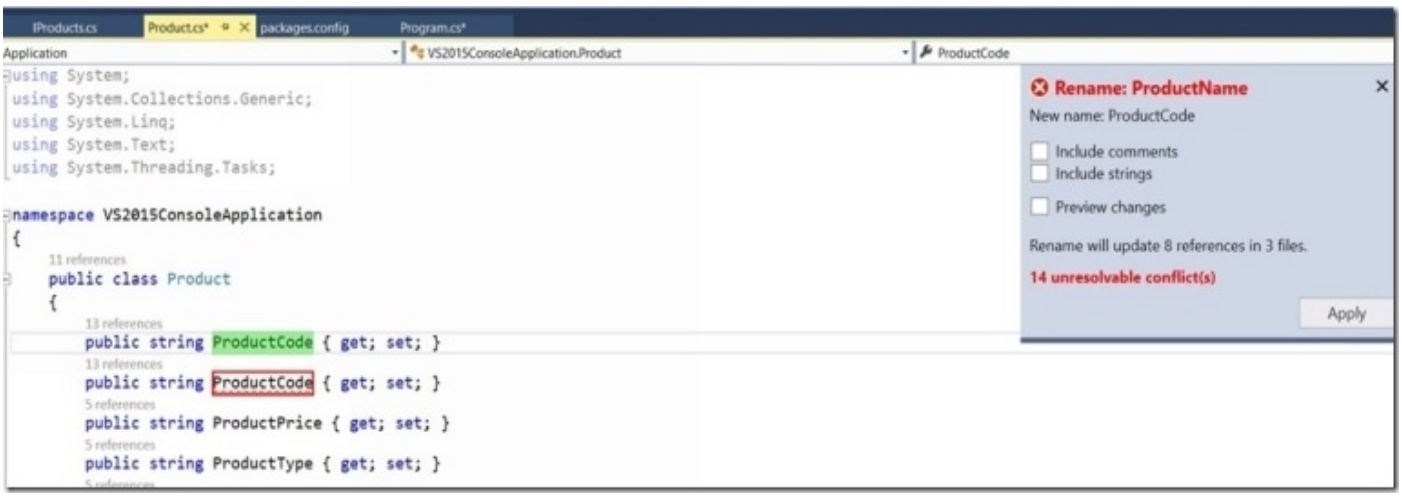
```
1: using System;
2: using System.Collections.Generic;
3: using System.Linq;
4: using System.Text;
5: using System.Threading.Tasks;
6:
7: namespace VS2015ConsoleApplication
8: {
9:     class Program
10:    {
11:        static void Main()
12:        {
13:            var myProducts = new MyProducts();
14:            Console.WriteLine( String.Format("Product with code 0002 is : {0}", myProducts.GetProduct("0002").ProductName));
15:            Console.WriteLine(Environment.NewLine);
16:            var productList = myProducts.GetProductList();
17:            Console.WriteLine("Following are all the products");
18:
19:            foreach (var product in productList)
20:            {
21:                Console.WriteLine(product.ProductName);
22:            }
23:            Console.ReadLine();
24:        }
25:    }
26: }
```

Rename Conflicts

Let us take another scenario where we try to rename a variable to another name that is already assigned to any other variable in the same method or try to rename a property of a class having another property having same name as of new name. Let us open Product class for example. Product class contains the Product properties, let us try to rename **ProductName** property. Right click on **ProductName** property and open Rename window by clicking Rename from context menu.

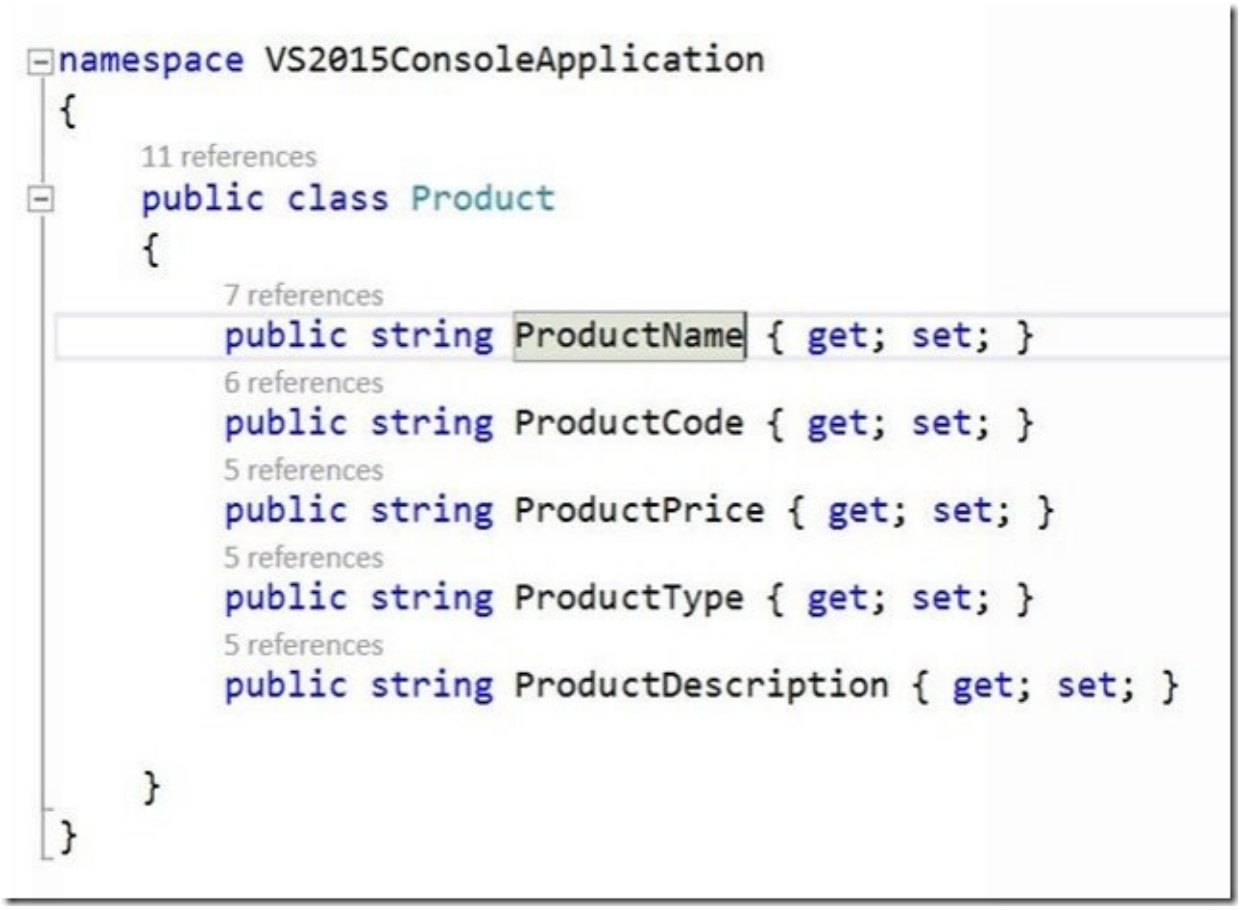


We'll change the name of **ProductName** property to **ProductCode**. We already have **ProductCode** property available in this class. Let us see what happens.Notice that when you start typing the new name of **ProductName** property to **ProductCode**, the rename window shows error with Red color, and the already exiting **ProductCode** property also shows a red box around it.



Here Visual Studio 2015 smartly tells that the new property name already exists in the

class and this change may result in having conflicts. Now you know that this change may cause your application to break, so just press escape and the rename window will be gone with your new name reverted to old one.



Therefore Visual Studio helps to detect any renaming conflicts and suggest to resolve them.

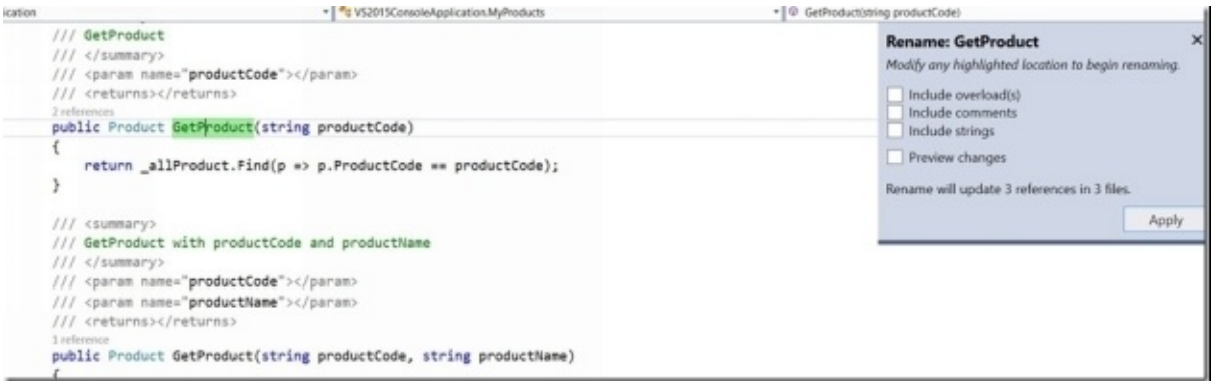
Rename Overloads, Strings, Code Comments

Let us take one more scenario. Add an overload method named GetProduct to IProduct interface and implement that method in MyProducts class.

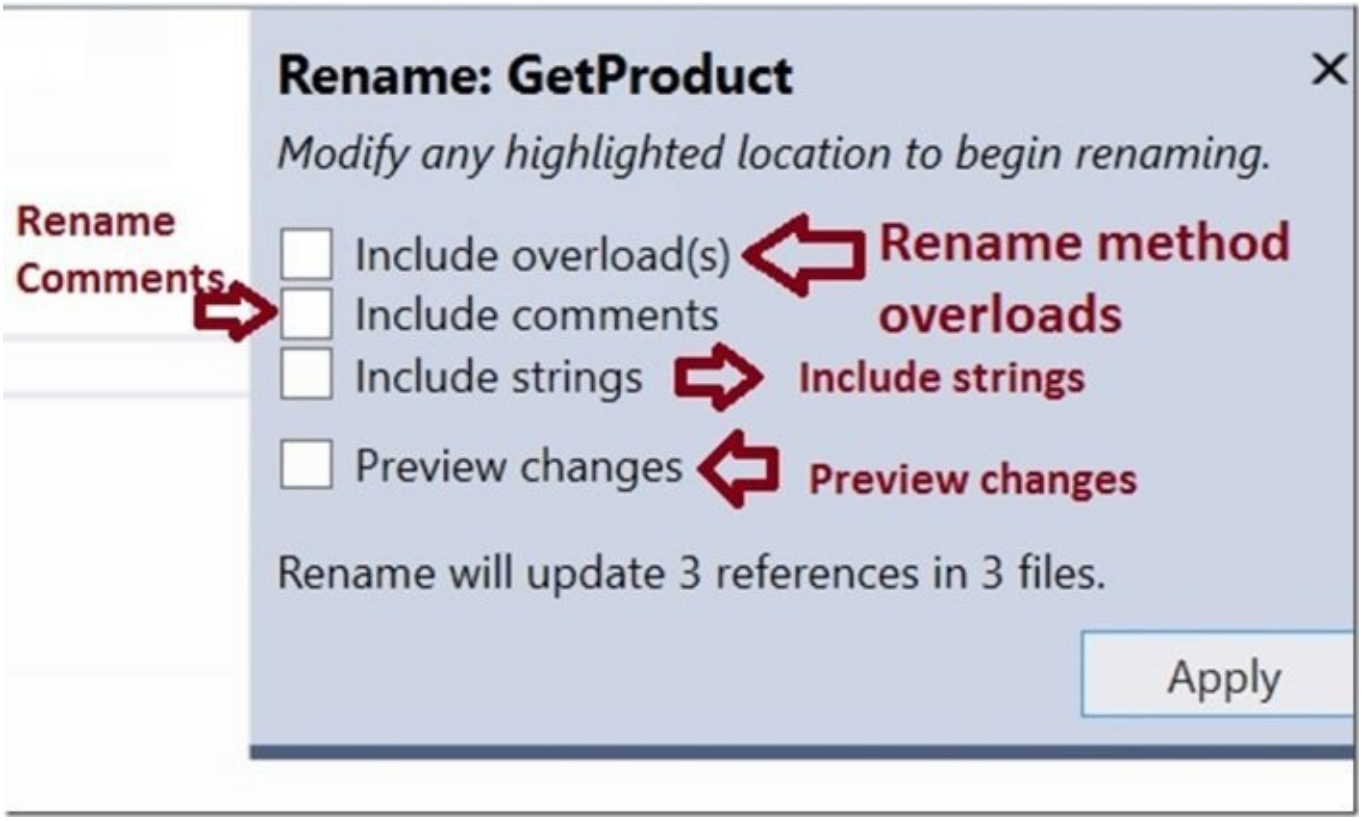
```
1: interface IProducts
2: {
3:     Product GetProduct(string productCode);
4:     Product GetProduct(string productCode,string productName);
5:     List<Product> GetProductList();
6: }
```

```
1: /// <summary>
2:     /// GetProduct
3:     /// </summary>
4:     /// <param name="productCode"></param>
5:     /// <returns></returns>
6:     public Product GetProduct(string productCode)
7:     {
8:         return _allProduct.Find(p => p.ProductCode == productCode);
9:     }
10:
11:    /// <summary>
12:    /// GetProduct with productCode and productName
13:    /// </summary>
14:    /// <param name="productCode"></param>
15:    /// <param name="productName"></param>
16:    /// <returns></returns>
17:    public Product GetProduct(string productCode, string productName)
18:    {
19:        return _allProduct.Find(p => p.ProductCode == productCode && p.ProductName==productName);
20:    }
21:
```

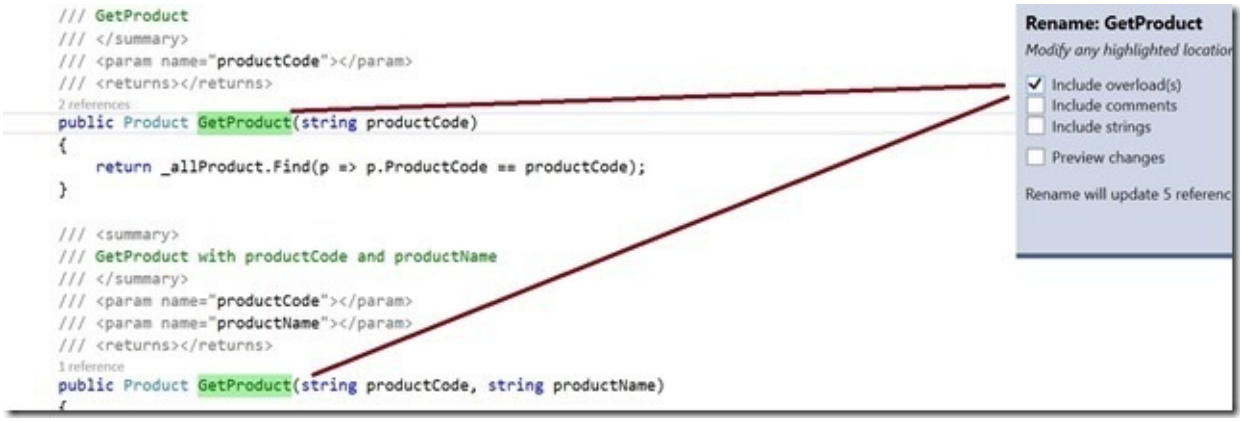
Now try to rename the GetProduct method in MyProducts class. Put the cursor in between the GetProduct method name and press Ctrl+R, Ctrl+R. The rename window will be opened as shown below.



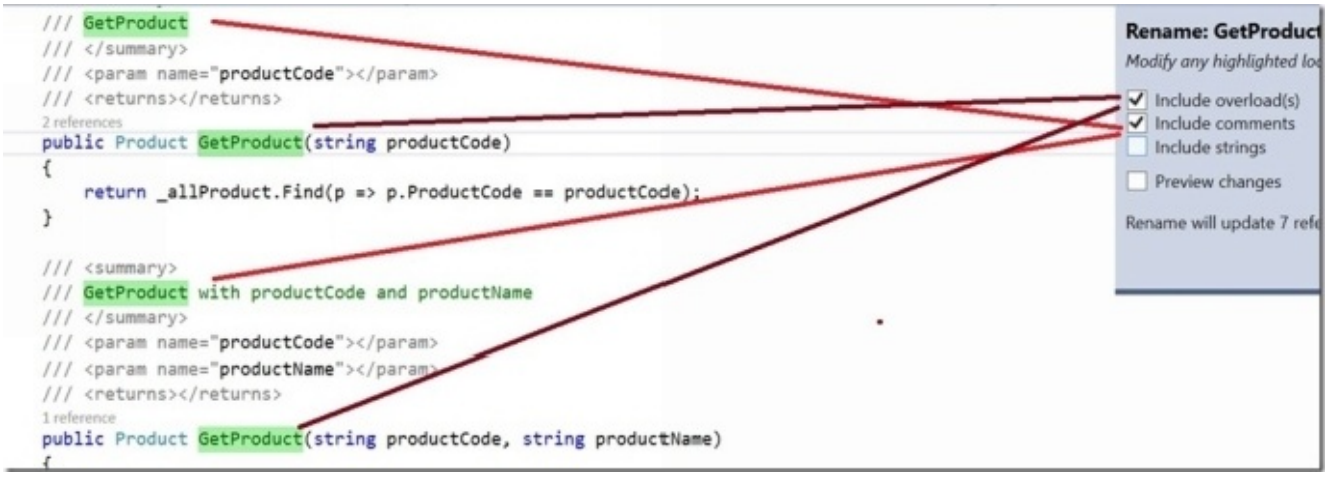
You see here the Rename window opens having few more options as shown below.



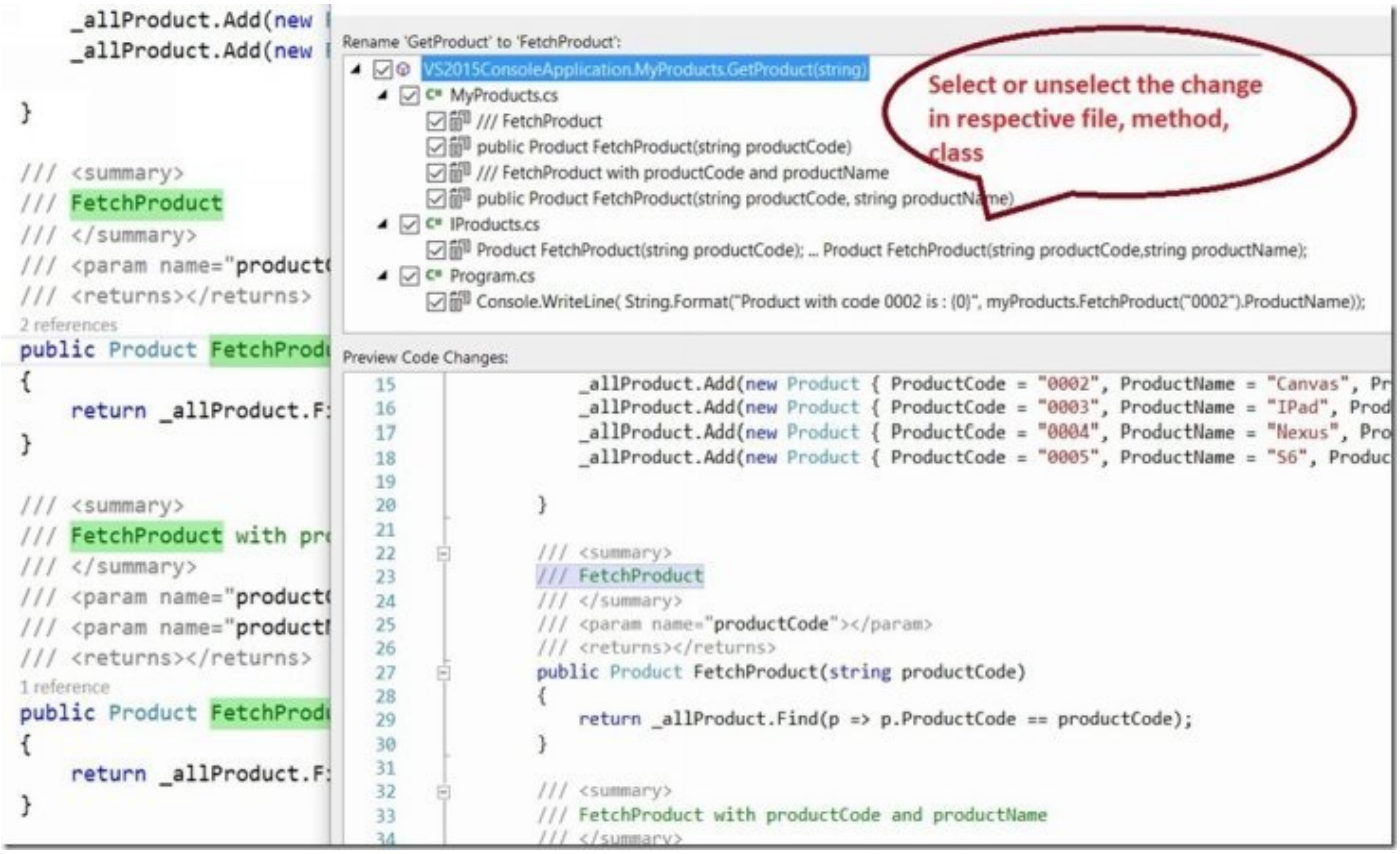
If you select the checkbox to include overloads, the overloads of that method which we are renaming also gets renamed, in this case if we choose this option the overloaded method of GetProduct() also gets renamed.



If we also choose the second option then the code comments also get renamed when we rename the method to new name. The name if exists in code comments also gets renamed to new name.



The renaming option also allows to rename related strings for that name, i.e. the third option. You can also preview the changes before renaming. preview changes window shows you all the occurrences that will remain, and provides you an option again to review and select un-select the change you want to take place at particular instance. Here I am trying to change GetProduct name to FetchProduct . Let us check Preview changes check box and press Apply button.

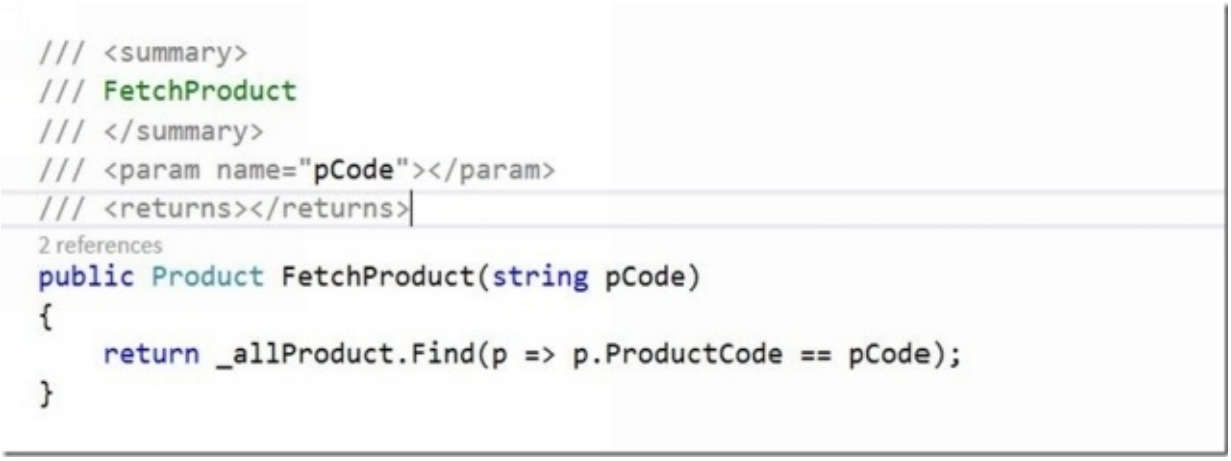


A preview window will get opened. Notice that not only for particular file but this window accumulates all the possible areas where this change may take place and affect code like in MyProduct.cs file, IProducts.cs interface and program.cs file. it says that when method name is changed, it will reflect in these files as well. Now it is the choice of a developer, whether to let that change happen or not. So preview window gives an option to developer to check or un-check the checkbox for corresponding affected file for that change to take place or not. The code suggestion and code assistance mechanism of Visual Studio is so strict that it takes care of all these modifications and automations very smartly.

In this case I didn’t un-select any of the file and let that change happen. it automatically renamed GetProduct to Fetch Product to all the areas shown including Comments, Interface and Program file as well.

In a similar way you can also rename the parameters passed in a method, this will also include all the possible renaming options like rename in comments and methods. Like shown in the following image, if I try to rename the parameter of FetchProduct method, it highlights all the possible renaming changes.

Put the cursor between productCode and press Ctrl+R, Ctrl+R, the rename window will appear and all instances of parameter gets highlighted, even the comment too. Change the name to pCode and click on apply.



We see the parameter name is changed along with the name in comment too.

Conclusion

In this chapter we covered Renaming assistance provided by Visual Studio 2015. We learnt about the renaming experience in various ways that includes following bullet points.

- Renaming assistance through light bulb actions
- Change preview with smart suggestions.
- Rename window and its renaming options.
- On the fly , live renaming experience. Rename as you type.
- Conflict detection and resolution while renaming.
- Renaming code comments.

In the next section of this book, I’ll cover code refactoring in Visual Studio 2015.

Code Refactoring in Visual Studio 2015

Introduction

Code Refactoring has always been a challenge for developers. This is one of the major skill that a developer should have to write an optimized , clean and fast code. There were third party tools available to help you achieve this, but none have shown that capability that Visual Studio 2015 has come up with. Visual Studio has always offered code refactoring techniques in tit-bit, but the latest version of Visual Studio i.e. 2015 provides a unique experience altogether to achieve refactoring. There are many features that refactoring of code in Visual Studio provides. We'll cover few of them like inline temporary variable and introduce local. Refactoring w.r.t. inline temporary variable and introduce local is not only limited to C# but VB developers can also leverage this feature. We'll cover the topic with one small method as an example and try to optimize it as far as we can. One thing is worth taking care of that code refactoring software and techniques are only meant for sharp developers. If you don't have an idea what new code will do and it looks strange to you, you should never try it.

Case Study

I am taking an example of MyProducts class that we created in earlier section.

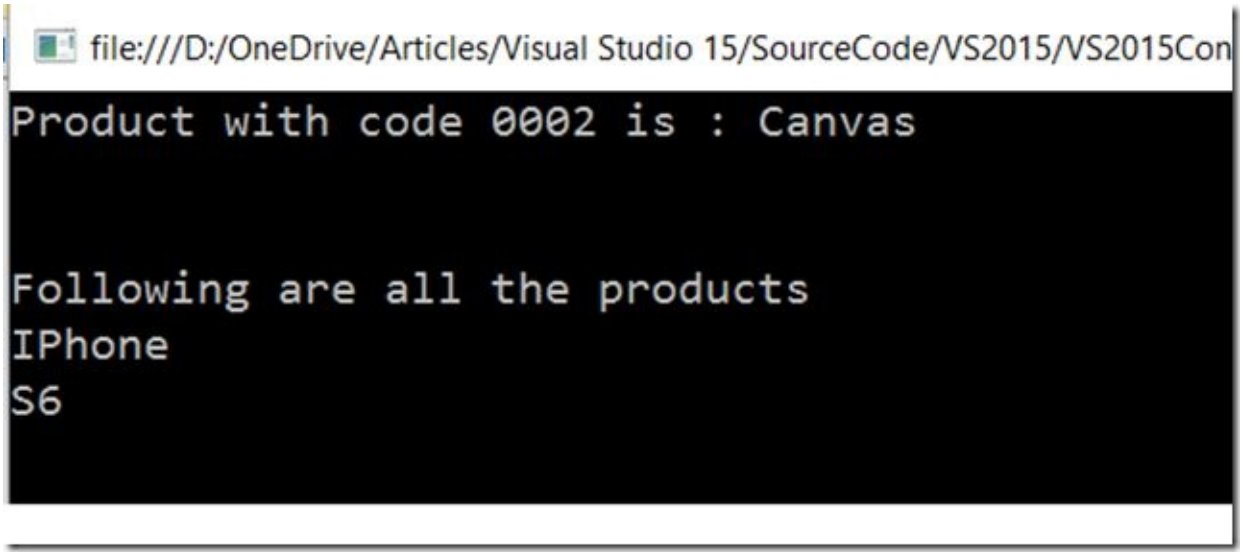
```
1: using System;
2: using System.Collections.Generic;
3: using System.Linq;
4: using System.Text;
5: using System.Threading.Tasks;
6:
7: namespace VS2015ConsoleApplication
8: {
9:     public class MyProducts :IProducts
10:    {
11:        List<Product> _allProduct = new List<Product>();
12:        public MyProducts()
13:        {
14:            _allProduct.Add(new Product {ProductCode="0001",ProductName="IPhone",ProductPrice="60000",ProductType="Phone",ProductDescription="Apple IPhone" } );
15:            _allProduct.Add(new Product { ProductCode = "0002", ProductName = "Canvas", ProductPrice = "20000", ProductType = "Phone", ProductDescription = "Micromax phone" });
16:            _allProduct.Add(new Product { ProductCode = "0003", ProductName = "IPad", ProductPrice = "30000", ProductType = "Tab", ProductDescription = "Apple IPad" });
17:            _allProduct.Add(new Product { ProductCode = "0004", ProductName = "Nexus", ProductPrice = "30000", ProductType = "Phone", ProductDescription = "Google Phone" });
18:            _allProduct.Add(new Product { ProductCode = "0005", ProductName = "S6", ProductPrice = "40000", ProductType = "Phone", ProductDescription = "Samsung phone" });
19:
20:        }
21:
22:        /// <summary>
23:        /// FetchProduct
24:        /// </summary>
25:        /// <param name="pCode"></param>
26:        /// <returns></returns>
27:        public Product FetchProduct(string pCode)
28:        {
29:            return _allProduct.Find(p => p.ProductCode == pCode);
30:        }
31:
32:        /// <summary>
33:        /// FetchProduct with productCode and productName
34:        /// </summary>
35:        /// <param name="productCode"></param>
36:        /// <param name="productName"></param>
37:        /// <returns></returns>
38:        public Product FetchProduct(string productCode, string productName)
```

```
39:     {
40:         return _allProduct.Find(p => p.ProductCode == productCode && p.ProductName==productName);
41:     }
42:
43:     public List<Product> GetProductList()
44:     {
45:         return _allProduct;
46:     }
47: }
48: }
```

We'll add one more method to this class. The objective of that method will be to return all products from the product list whose price is greater than 30000. I am trying to keep the logic and method very simple for the sake of understanding. I have kept the name of this method is FetchProduct(). Notice that we already had two methods with the same name. Now this will also work as an overload to FetchProduct() method.

```
/// <summary>
/// FetchProduct having price greater that 3000
/// </summary>
/// <returns></returns>
1 reference
public List<Product> FetchProduct()
{
    var products = from p in _allProduct where Convert.ToInt32(p.ProductPrice) > 30000 select p;
    var productList = products.ToList();
    return productList;
}
```

The above mentioned method is very simple in nature and contains LINQ query that fetches products having price greater than 30000. When you call this method from Program.cs and iterate over the elements we get following result.



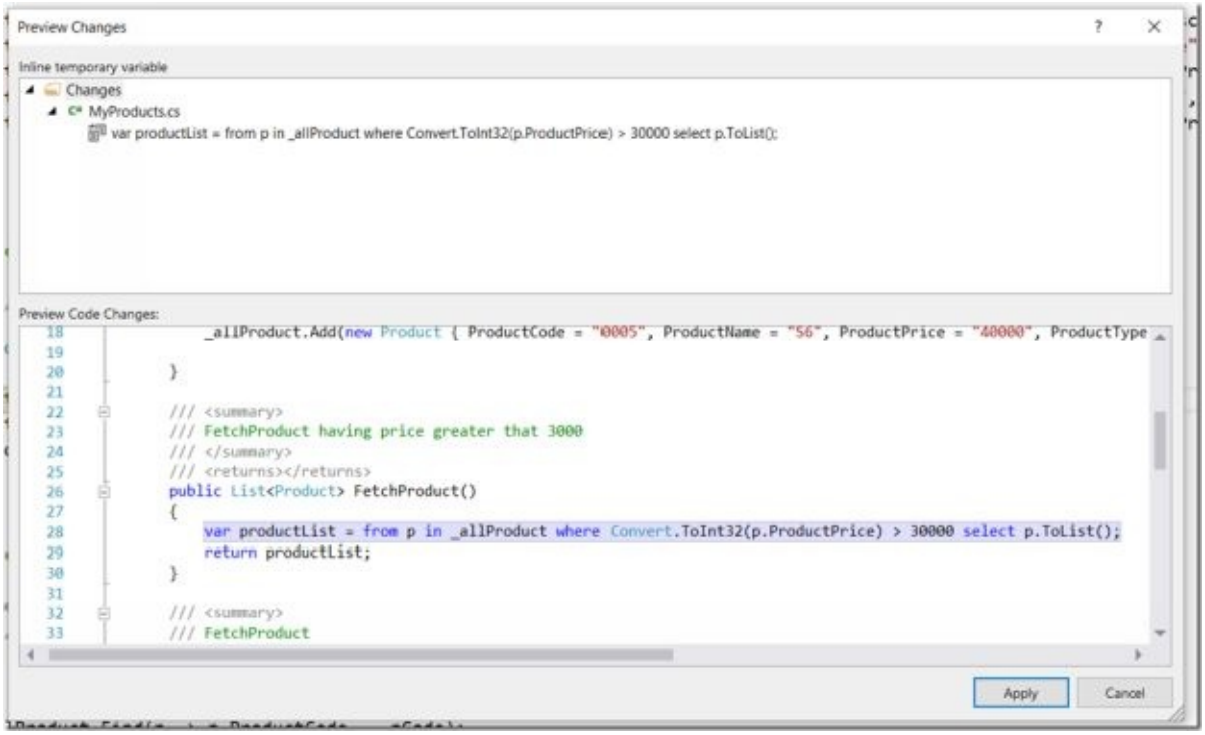
Program.cs code is as follows.

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

```
 7: namespace VS2015ConsoleApplication
 8: {
 9:     class Program
10:     {
11:         static void Main()
12:         {
13:             var myProducts = new MyProducts();
14:             Console.WriteLine( String.Format("Product with code 0002 is : {0}", myProducts.FetchProduct("0002").ProductName));
15:             Console.WriteLine(Environment.NewLine);
16:             var productList = myProducts.FetchProduct();
17:             Console.WriteLine("Following are all the products");
18:
19:             foreach (var product in productList)
20:             {
21:                 Console.WriteLine(product.ProductName);
22:             }
23:             Console.ReadLine();
24:         }
25:     }
26: }
```

So, our method is working fine. the question is how much we can optimize this method further. When you click in between products variable, the light bulb icon will show up with some suggestions that you can apply to this line to optimize.

Here come inline temporary variable in picture. The light bulb icon says to remove this variable and bring that code to a single line. When you preview the changes that this code assistance is suggesting you get following preview window.



It shows that the temporary products variable will be replaced with productList itself therefore helping us to save number of lines as well as memory allocation for a variable. Click on apply changes and you'll get the refactored code.

```
/// <summary>
/// FetchProduct having price greater that 3000
/// </summary>
/// <returns></returns>
1 reference
public List<Product> FetchProduct()
{
    var productList = from p in _allProduct where Convert.ToInt32(p.ProductPrice) > 30000 select p.ToList();
    return productList;
}
```

Now can this code be further refactored. you can take help from light bulb icon. Either click in between productList variable or right click on productList variable and open Quick Actions from context menu.

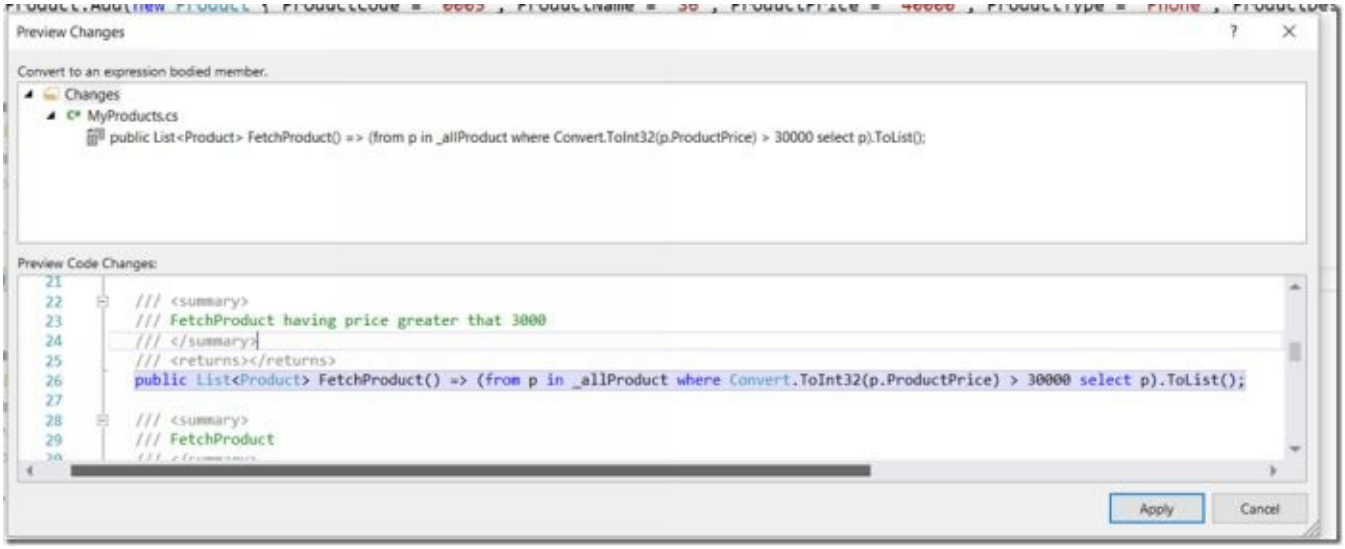
We see here, productList is also a temporary variable and it is suggested to remove that too. Let us preview changes and apply them to get more optimized code. Doing this the productList variable will be replaced by a single return statement, but you'll notice an error here.if you remember, I said that code refactoring is for intelligent and sharp developers. We see here that while refactoring the code, LINQ query is not encapsulated in the bracket and ToList() method is directly applied to "p" variable. We have to rectify this by putting brackets around LINQ query. This was one of the scenario, and you may face many of such. So you have to be sure about the change that you are about to do. visual Studio only suggests, it does not code for you.

```
/// <summary>
/// FetchProduct having price greater that 3000
/// </summary>
/// <returns></returns>
1 reference
public List<Product> FetchProduct()
{
    return (from p in _allProduct where Convert.ToInt32(p.ProductPrice) > 30000 select p).ToList();
}
```

Brackets to remove ToList error.

Now we have a single return statement. Our code has reduced to just one line and we saved few memory too by ignoring temporary variables. This certainly makes the code faster as well. But can we further optimize this method? Let's take a shot and click on return statement. You'll see the light bulb icon again shows up with some suggestions.

It says that the method could be converted to an expression bodied member. Let us preview the change.



The preview says that the method is converted into an expression. Looking at this I do not find any issue. So we can certainly opt this option for the sake of refactoring. Press apply changes and we get following code.

```
/// <summary>
/// FetchProduct having price greater that 3000
/// </summary>
/// <returns></returns>
1 reference
public List<Product> FetchProduct() => (from p in _allProduct where Convert.ToInt32(p.ProductPrice) > 30000 select p).ToList();
```

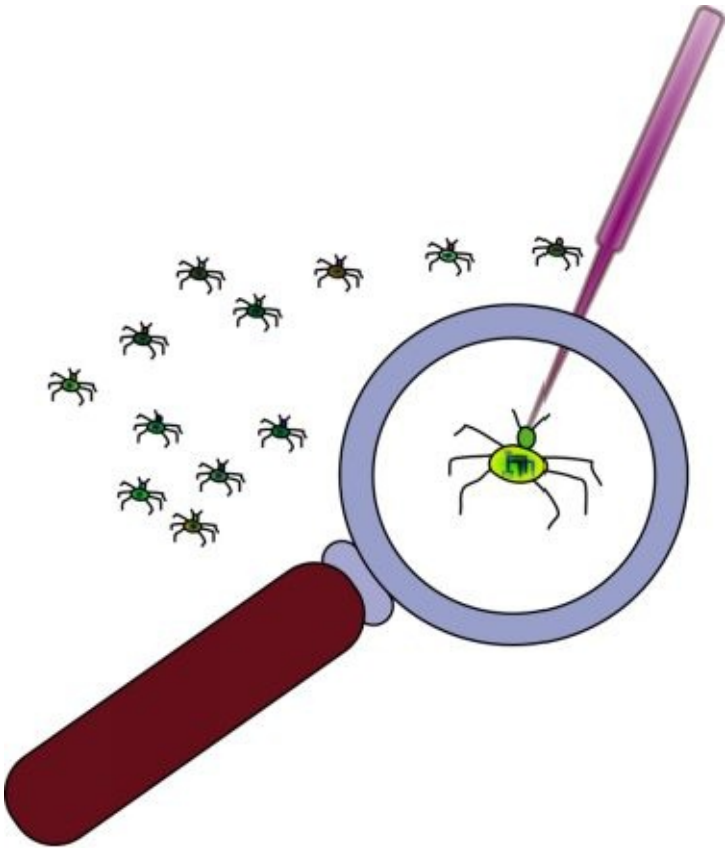
Now if you try to further optimize this method, you’ll not find much scope.

Conclusion

This was just a small example that I showed on how you can leverage the capability of Visual Studio 2015 in optimizing your code. In the next section of this book I’ll be covering topics like debugging features in Visual Studio 2015.

Breakpoint Configuration Improvements and New improved Error List

Visual Studio has always been a great IDE for code debugging. It provides a numerous features for debugging and configuring the code. Being a developer we always spend a lot of time spend time in running and debugging the code, therefore improvements to debugging features can have a big impact on our productivity. This chapter covers the debugging improvements that Visual Studio 2015 has come up with.



Breakpoint configuration improvements

The earlier versions of Visual Studio already provided the feature of breakpoint configuration , so it’s not new to developer. The only thing new in Visual Studio 2015 is the user experience and ease of using the configurations. Breakpoint configuration is now more easier to use and reachable. Visual Studio 2015 introduces a new inline toolbar. With this toolbar you can easily open the Breakpoint Configuration Settings or enable/disable the breakpoint. Secondly, the Context menu for break point configuration in Visual Studio 2015 is simplified. The few options of the Context menu have been moved to the Breakpoint Configuration Settings window. The Settings window now comes in a peek window, so you can easily check and change the settings as there will be no modal window. The whole breakpoint configuration is now divided into two parts, Actions and Conditions. Let us understand the topic in detail with practical implementation. I am using Visual Studio 2015 enterprise edition for this chapter and have added a console application named VS2015ConsoleApplication in my Visual Studio.Let’s say we have a MyProduct class containing product as an entity specific basic operations like fetching the product, returning the list of products as shown below.

```
1: using System;
2: using System.Collections.Generic;
3: using System.Linq;
4: using System.Text;
5: using System.Threading.Tasks;
6:
7: namespace VS2015ConsoleApplication
8: {
9:     public class MyProducts :IProducts
10:    {
11:        List<Product> _allProduct = new List<Product>();
12:        public MyProducts()
13:        {
14:            _allProduct.Add(new Product {ProductCode=“0001”,ProductName=“IPhone”,ProductPrice=“60000”,ProductType=“Phone”,ProductDescription=“Apple IPhone” } );
15:            _allProduct.Add(new Product { ProductCode = “0002”, ProductName = “Canvas”, ProductPrice = “20000”, ProductType = “Phone”, ProductDescription = “Micromax phone” });
16:            _allProduct.Add(new Product { ProductCode = “0003”, ProductName = “IPad”, ProductPrice = “30000”, ProductType = “Tab”, ProductDescription = “Apple IPad” });
17:            _allProduct.Add(new Product { ProductCode = “0004”, ProductName = “Nexus”, ProductPrice = “30000”, ProductType = “Phone”, ProductDescription = “Google Phone” });
18:            _allProduct.Add(new Product { ProductCode = “0005”, ProductName = “S6”, ProductPrice = “40000”, ProductType = “Phone”, ProductDescription = “Samsung phone” });
19:
20:        }
21:
22:        /// <summary>
23:        /// FetchProduct having price greater that 3000
24:        /// </summary>
25:        /// <returns></returns>
```

```
26:         public List<Product> FetchProduct() => (from p in _allProduct where Convert.ToInt32(p.ProductPrice) > 30000 select p).ToList();
27:
28:         /// <summary>
29:         /// FetchProduct
30:         /// </summary>
31:         /// <param name="pCode"></param>
32:         /// <returns></returns>
33:         public Product FetchProduct(string pCode)
34:         {
35:             return _allProduct.Find(p => p.ProductCode == pCode);
36:         }
37:
38:         /// <summary>
39:         /// FetchProduct with productCode and productName
40:         /// </summary>
41:         /// <param name="productCode"></param>
42:         /// <param name="productName"></param>
43:         /// <returns></returns>
44:         public Product FetchProduct(string productCode, string productName)
45:         {
46:             return _allProduct.Find(p => p.ProductCode == productCode && p.ProductName==productName);
47:         }
48:
49:         public List<Product> GetProductList()
50:         {
51:             return _allProduct;
52:         }
53:     }
54: }
```

where IProducts is a simple interface.

```
1: using System;
2: using System.Collections.Generic;
3: using System.Linq;
4: using System.Text;
5: using System.Threading.Tasks;
6:
7: namespace VS2015ConsoleApplication
8: {
9:     interface IProducts
10:     {
11:         Product FetchProduct(string productCode);
12:         Product FetchProduct(string productCode,string productName);
13:         List<Product> GetProductList();
14:     }
```

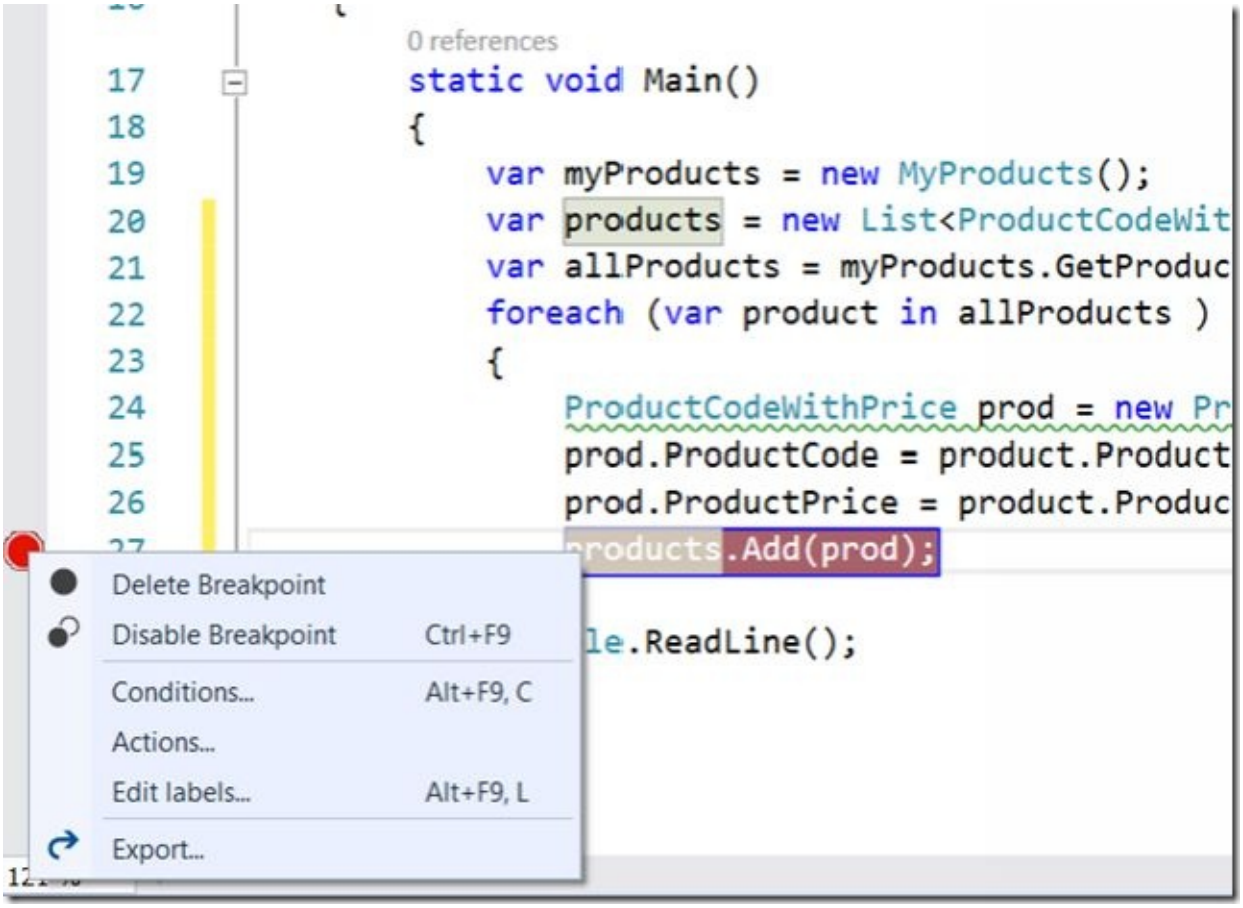
```
15: }
```

In the following Program class, we are just fetching all the products and creating a new list of products for a new entity named ProductCodeWithPrice, where we list only the product code and price of products.

```
 1: using System;
 2: using System.Collections.Generic;
 3: using System.Linq;
 4: using System.Text;
 5: using System.Threading.Tasks;
 6:
 7: namespace VS2015ConsoleApplication
 8: {
 9:     public class ProductCodeWithPrice
10:     {
11:         public string ProductCode { get; set; }
12:         public string ProductPrice { get; set; }
13:
14:     }
15:     class Program
16:     {
17:         static void Main()
18:         {
19:             var myProducts = new MyProducts();
20:             var products = new List<ProductCodeWithPrice>();
21:             var allProducts = myProducts.GetProductList();
22:             foreach (var product in allProducts )
23:             {
24:                 ProductCodeWithPrice prod = new ProductCodeWithPrice();
25:                 prod.ProductCode = product.ProductCode;
26:                 prod.ProductPrice = product.ProductPrice;
27:                 products.Add(prod);
28:             }
29:             Console.ReadLine();
30:         }
31:     }
32: }
```

Now let us say we are debugging the code while a new product list is created and we want to place a breakpoint after a new ProductCodePrice instance is created in foreach loop.

When a breakpoint is put at line 27, notice the new inline toolbar. From here I can open the Settings or Enable and Disable the breakpoint. When we right click on the breakpoint to open the context menu, we see a new simplified context menu with most of the options that use to be presenting there now moved to settings option.



Let’s again check the inline toolbar. Let’s pick the Settings option. Notice that the settings now appear in a peek window instead of a modal dialog window. This helps a developer to easily modify the settings while debugging.

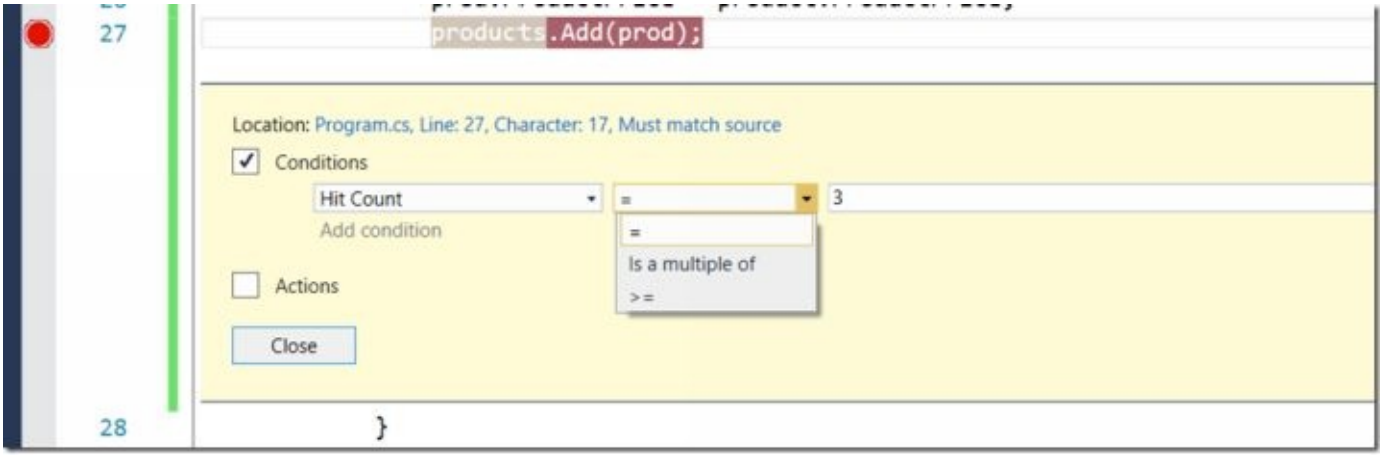

24 ProductCodeWithPrice prod = new ProductCodeWithPrice();
25 prod.ProductCode = product.ProductCode;
26 prod.ProductPrice = product.ProductPrice;
27 products.Add(prod);
Breakpoint Settings
Location: Program.cs, Line: 27, Character: 17, Must match source
Conditions
Actions
Close
28 }

Conditions

Let’s try to explore how conditions work. When we place a breakpoint and open the settings window, it shows options for Conditions and Actions and also mentions the location of breakpoint with the details like file name, line number and character position. Clicking on conditions checkbox shows some other options on how a condition can be configured.

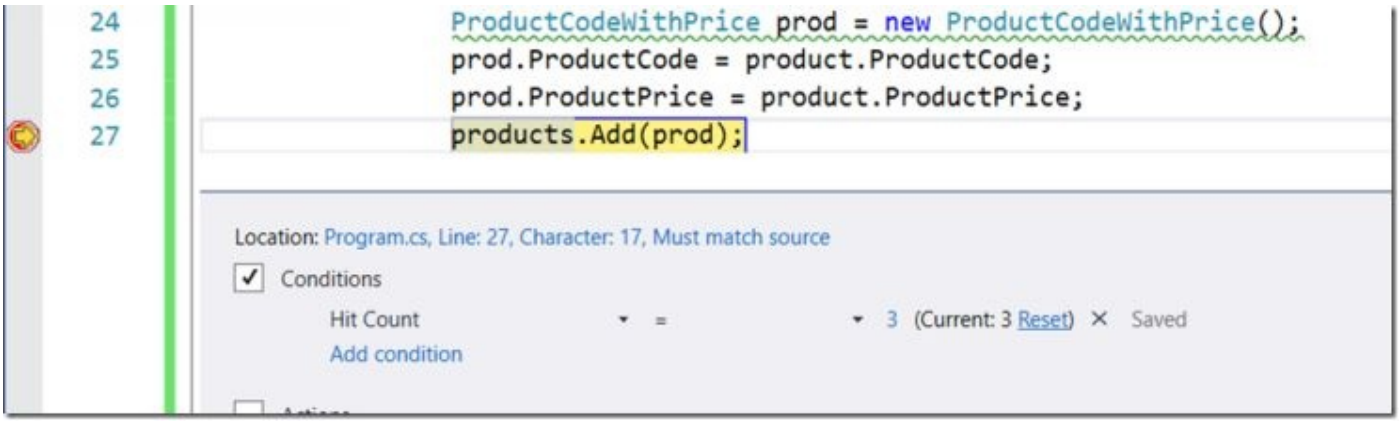
The default is Conditional Expression, but there are two other options as well i.e. Hit Count and Filter. Hit Count option is used when there is a need that an execution pause is required at a particular iteration in the loop.

The second drop down list is used to validate the condition. In this case we have placed a breakpoint after prod object is created in each iteration.

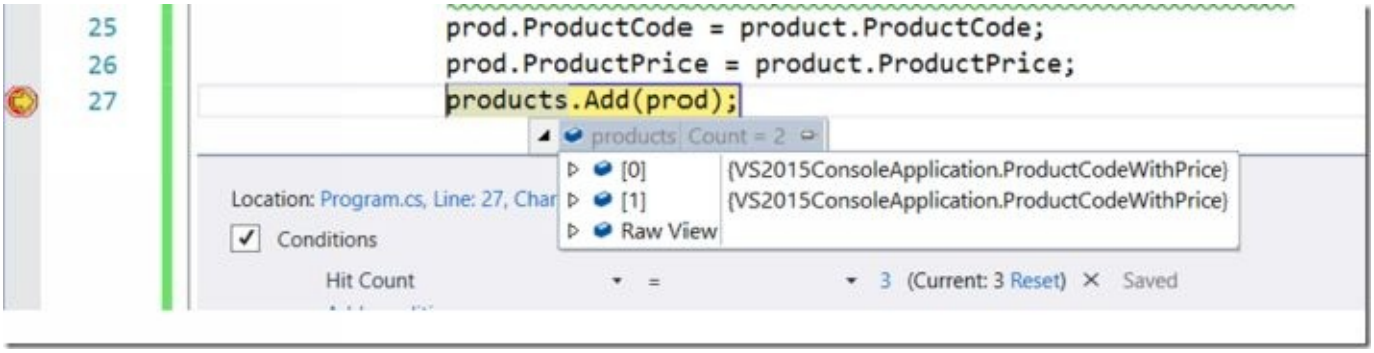


Notice that we could pick Is a multiple of, or greater than or equal to to validate the Hit Count.

Let’s suppose there is a scenario where we need to pause the execution and check the products list values after 3 iterations. So we choose Hit Count option as condition and “Is equal” to option in second dropdown and in the text box near to it, type 3. This means that when the loop will be running third time the execution is paused at line number 27 therefore hitting the breakpoint. Run the application and wait for the breakpoint to get hit.

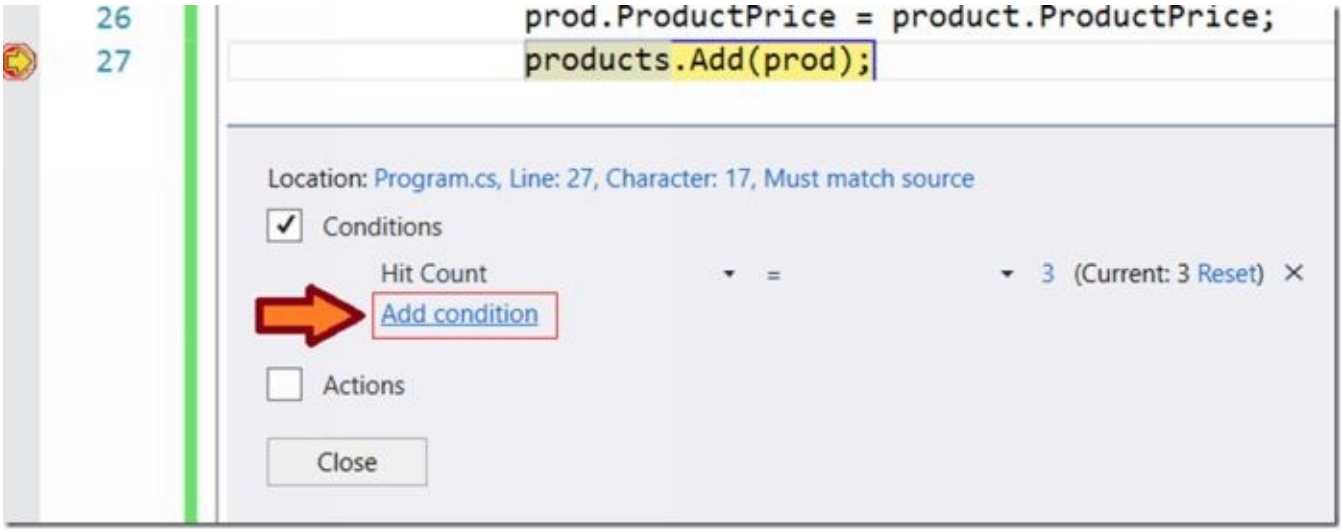


Notice that the conditions information is live. It shows me the current Hit Count. The application stopped at debug point when the hit count was 3. At this point the count can also be changed, let's change it to 4, or it could simply be reset, and data tooltips can still be used to view the variables. If we hover over the products list we can see it already has two products (prod) in it, so we must be in the third iteration of the loop because we're breaking before we're adding to the list.

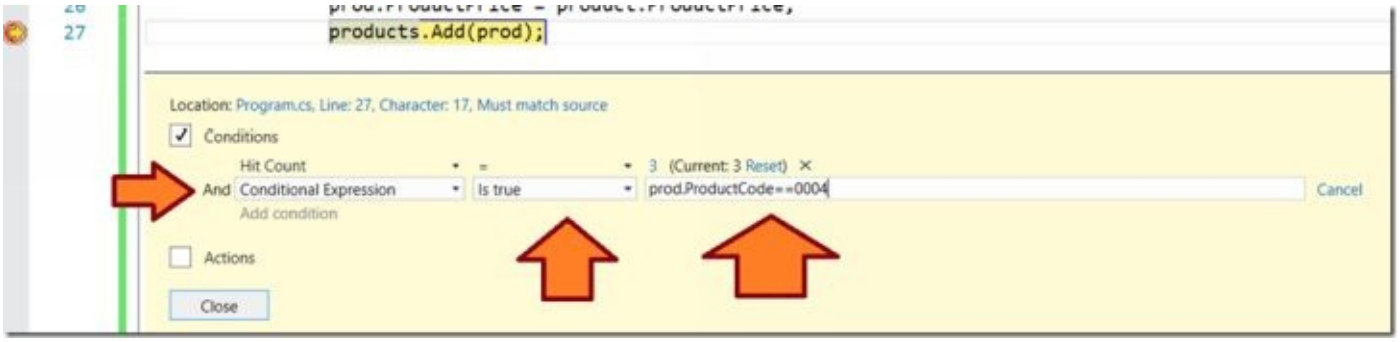


One of the interesting feature w.r.t. Visual Studio 2015 break point configuration is that if a breakpoint is accidentally deleted , it could again be applied by using Ctrl+Z.

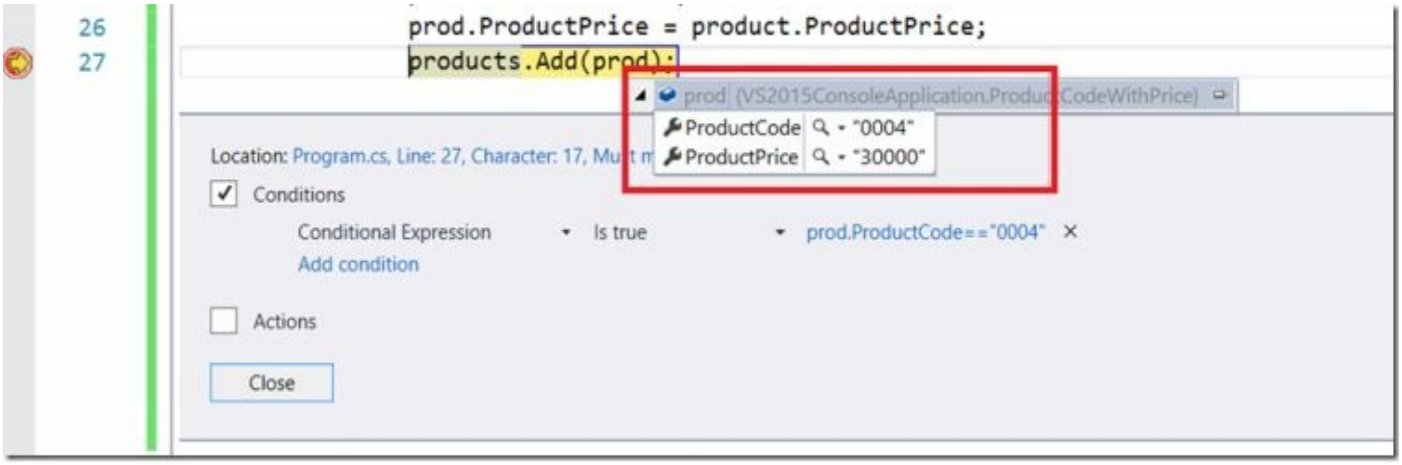
A breakpoint condition with the Hit Count can be used any time if we need to hit the breakpoint at specific hit count or at some particular interval of hits. This is normally useful while processing lists of items and in recursive methods. Even though the application is still running, another condition can also be selected to be added, let's add it through the conditional expression. We'll check this by adding a Conditional Expression here. Let's say we want the breakpoint to be hit when the product code of prod instance is "0004" . So click on Add condition option while the application is stopped at the breakpoint and add a conditional expression.



You can add multiple conditions and configure your breakpoint for desired debugging to improve productivity. When Add condition option is clicked a new row is added with all available options as shown earlier while applying Hit Count breakpoint. Choose conditional expression option and validate it to be true when prod.ProductCode==”0004”. Notice that you can write any expression in the expression textbox. The expression could be simple or complex with multiple && and || conditions too. Moreover while typing, the intellisense also works and helps to create expressions.



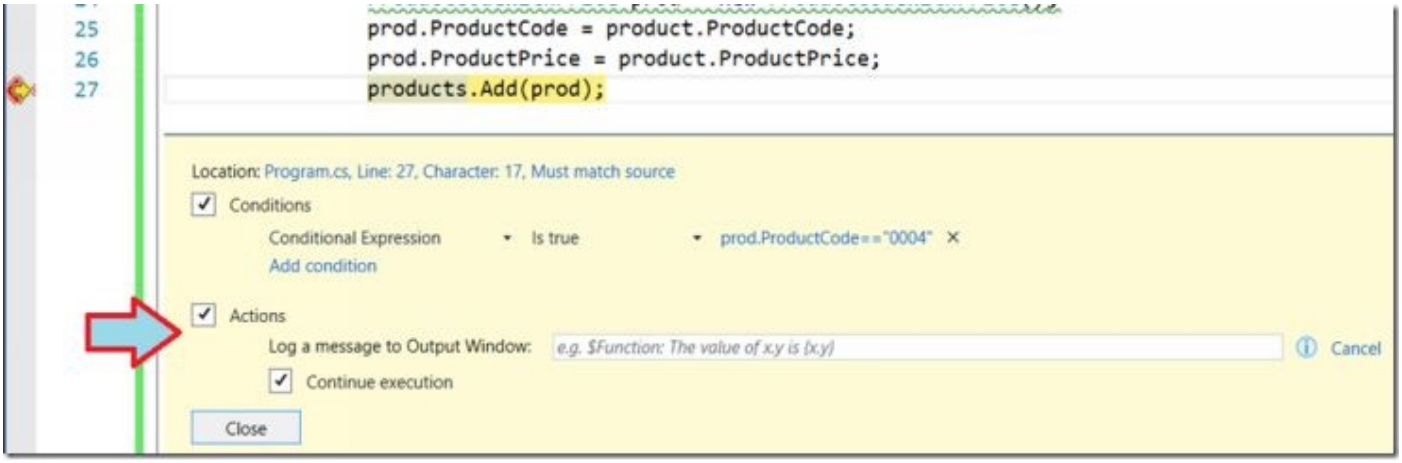
If you want you can delete the prior condition of hit count , else the debug point will be hit multiple times. I am removing the prior condition here. Run the application and you’ll see that the break point is hit when the condition that was mentioned at breakpoint becomes true.



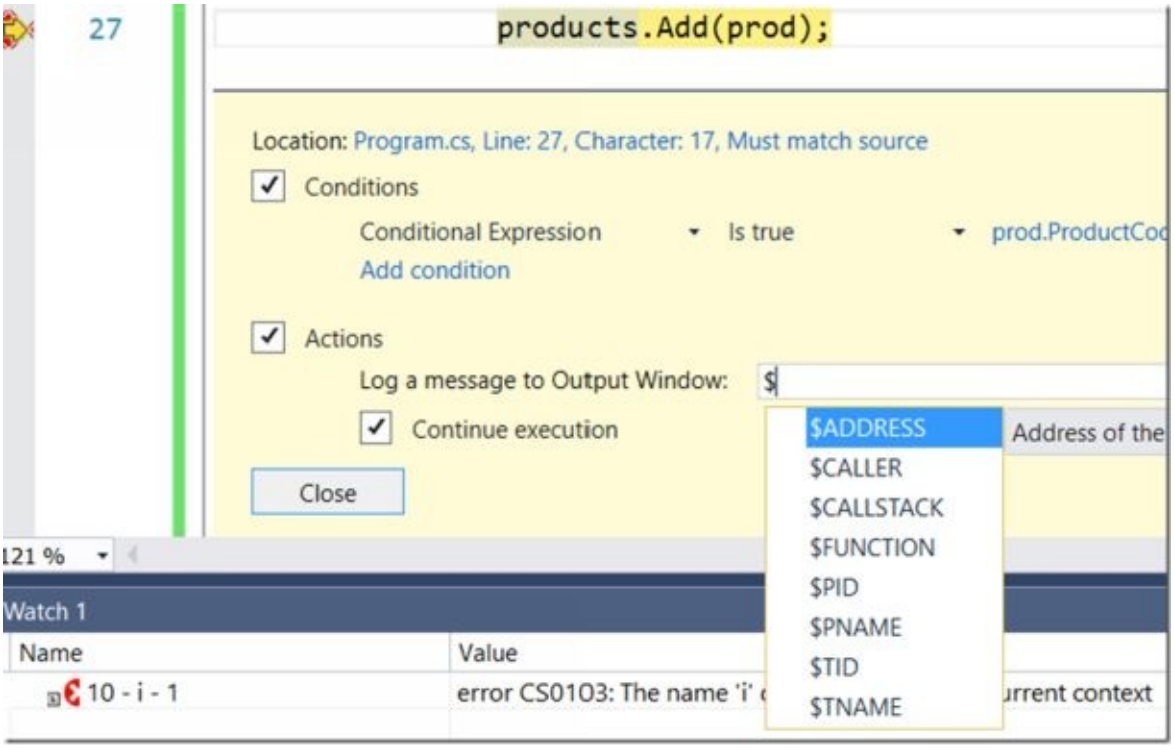
We see here the execution stops as soon as the condition of product code being “0004” is met.

Actions

Let us see how Actions work. By default, when the conditions are true, the debugger will pause at the particular breakpoint. This behavior can also be configured by checking the actions. One can select to log the message, enter the desired message in the Message field provided.

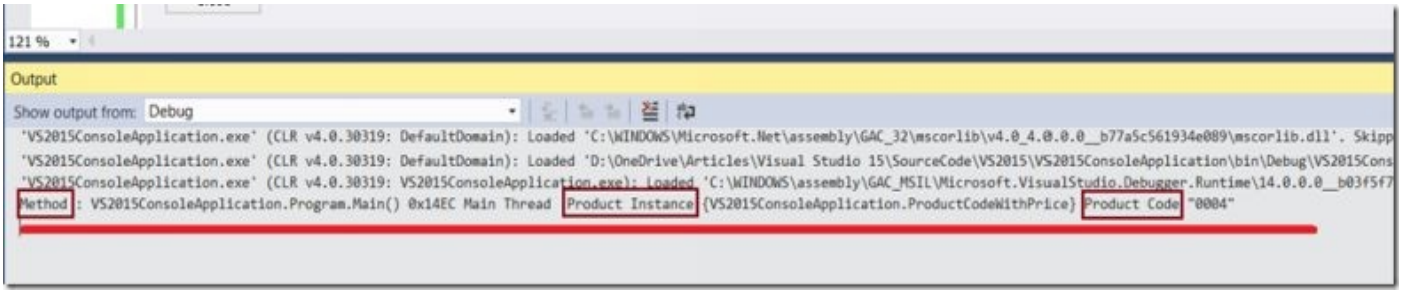


We can also enter desired plain text of our choice and customize the message for better readability and understanding. Dollar ($) can be used to display system values here, when you type dollar in the message field , you get the list of all the pseudo variables that can be used to log the message.



Curly braces {} are used to display the output or variables from the application or code base and you get the intellisense support as well in the message fields. You can log the message in output window. let's give it a try and try to log something at this breakpoint condition. You also have the option to Continue execution. This option refrains the debugger from pausing each time a breakpoint is hit. This option could be selected if you want to log the message without stopping at the breakpoint.

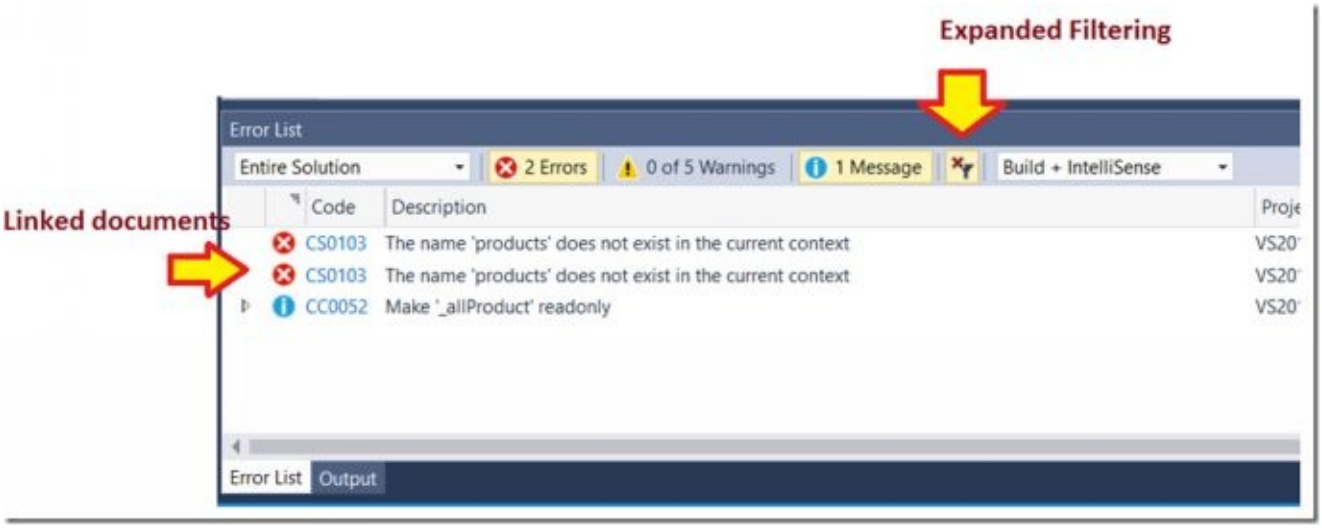
In actions message field, I am trying to log a message when the condition of prod having product code == “0004” is true. I have configured the message field to log $Function , $TID, $TNAME along with {prod} i.e. product instance and prod.ProductCode. notice that I have also used plain text like “Method : ”, “Product Instance”, “Product Code” to make my message more readable. I have chosen to continue the execution without stopping at breakpoint. Let’s run the application and see what happens.



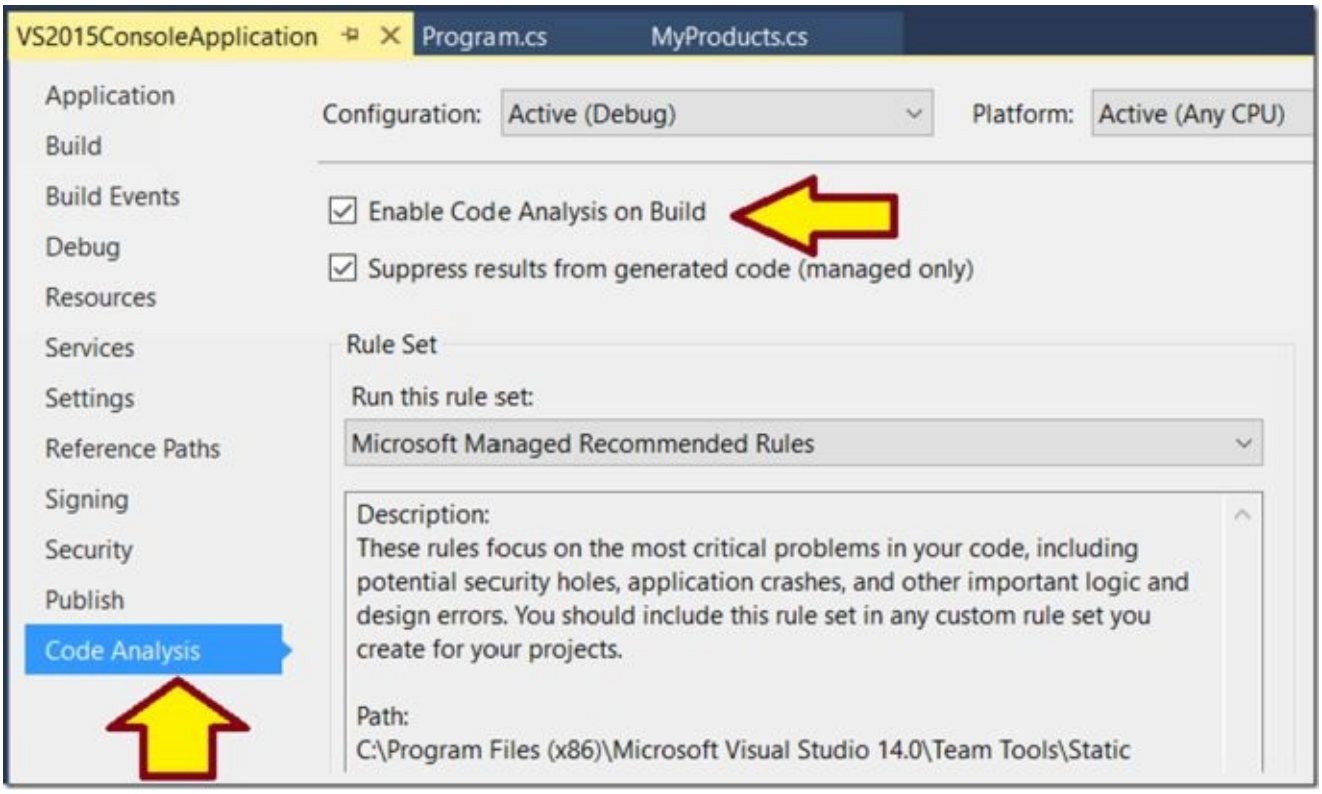
All the information that we defined in Message field is logged into output window as desired. All the details along with the plain text that I used is logged in the same sequence as defined. You can use the Log a message action anytime when you want to display information each time the breakpoint is hit.

New improved Error List

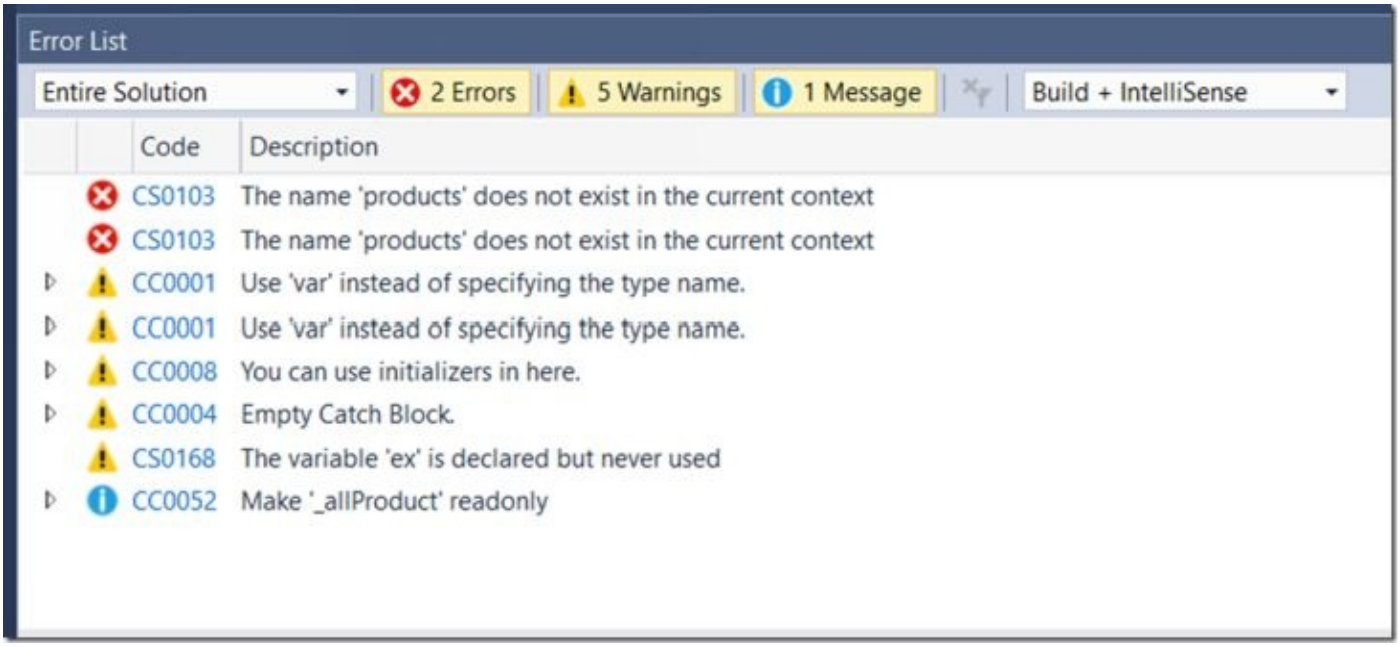
The new Error List in Visual Studio 2015 is now much more improved where you can get your live list of compiler and code analysis errors and warnings. The major improvements in the Error List include display of the error code, linked to a document on that issue. You can click that link to view the document online. Filtering has been expanded much more. One can still filter on current project or document, but now filtering can also be done on error severity, the error code, a set of projects or on a set of files.



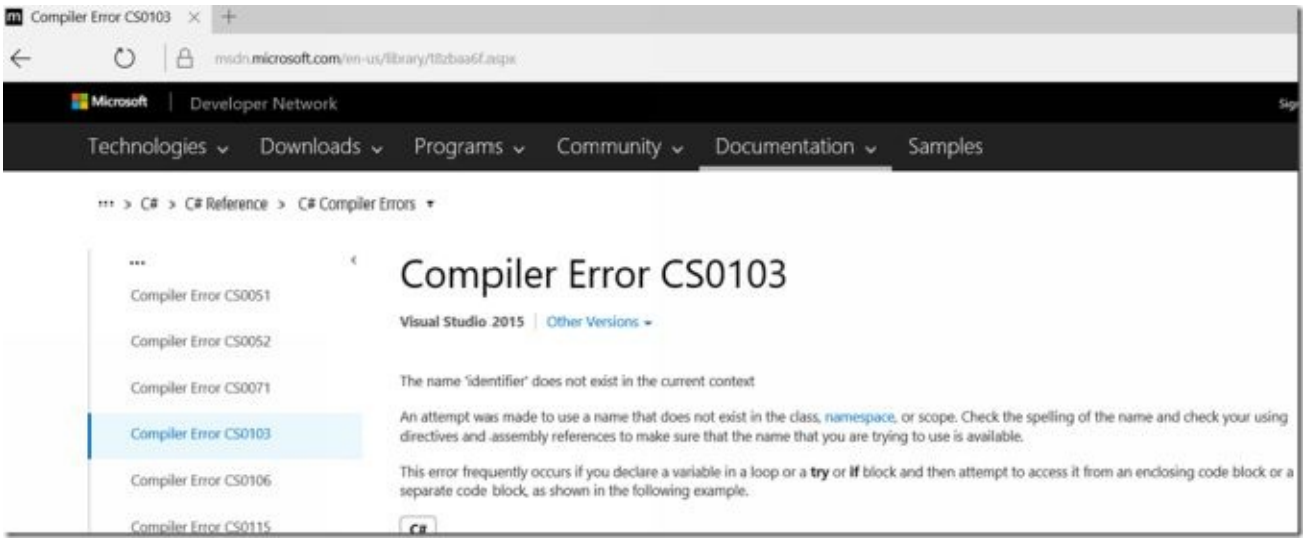
The maximum error limit in Visual Studio 2015 has also been removed. Earlier there was no way to really tell how many errors we had in one go when the error number was too high. Each time we fix certain numbers of errors, we were shown more errors on compilation Now, all of your errors and warnings will appear in the Error List in one go. Let’s practically try to see the error list improvements. I have intentionally made few changes in the Main method of program.cs class to get some errors and warnings. I have removed var from products declaration, added an empty catch block with an empty finally block. Before compiling the code, I have also enabled the Enable Code Analysis on Build option. You can find this option by right clicking on your project, open properties and in properties window select Code Analysis option , normally appears at last as shown in the image.



Now when we compile that code we get few errors and warning as expected.

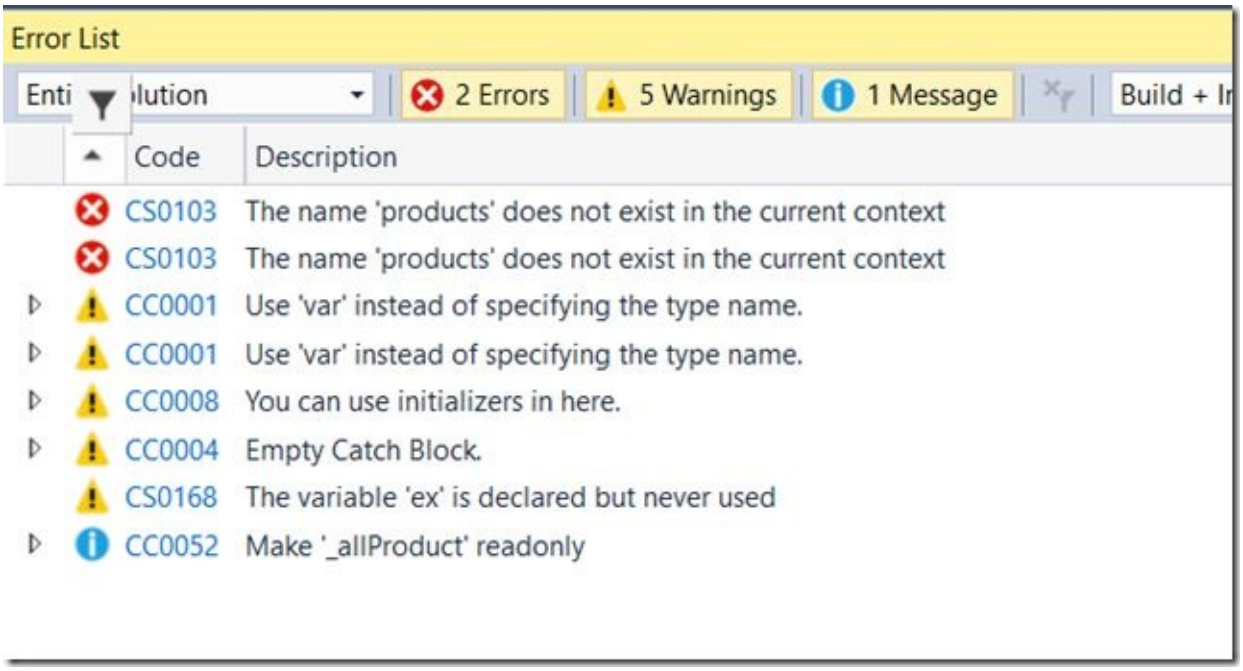


we see here that we get errors and warnings from the compiler and as well from the code analyzer. CS as a prefix to the error/warning code represents that it is through compiler and CC represents code analyzers here. We got all the expected warnings and errors. Notice that errors and warnings have their respective symbols. The tabs at the top shows 2 Errors, 5 Warnings and 1 Message. You can choose these options to filter and see what you need. Let's say you don't want to see Warnings and Messages, then you can click on the respective tabs above to see only Error list. Notice that every error code is in the form of a link when you click on any error code, it redirects you to its documentation page. Let's click on CS 0103 i.e. the first error saying "The name 'products' does not exist in the current context".

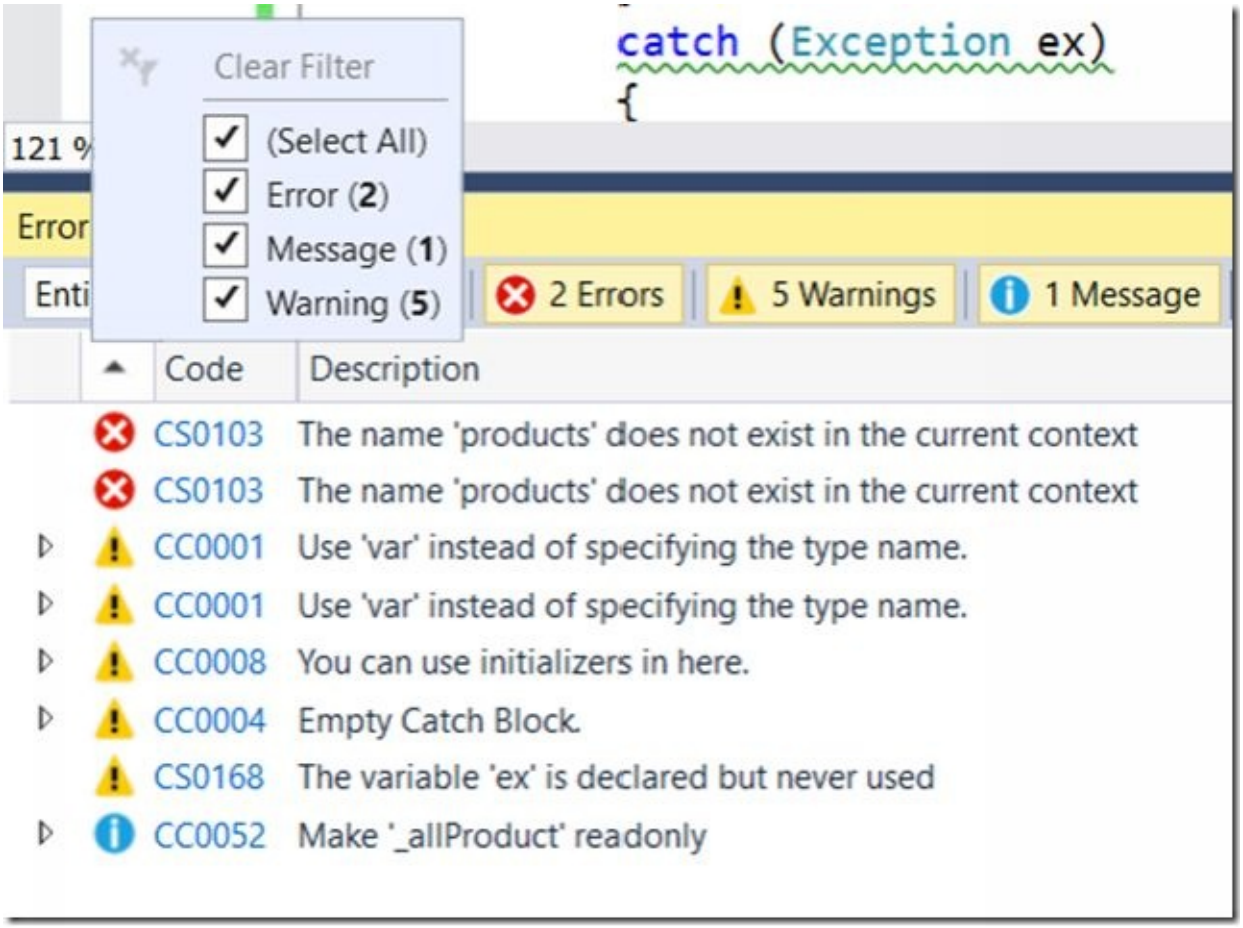


We see that the ink has redirected to MSDN link having detailed document of that error.

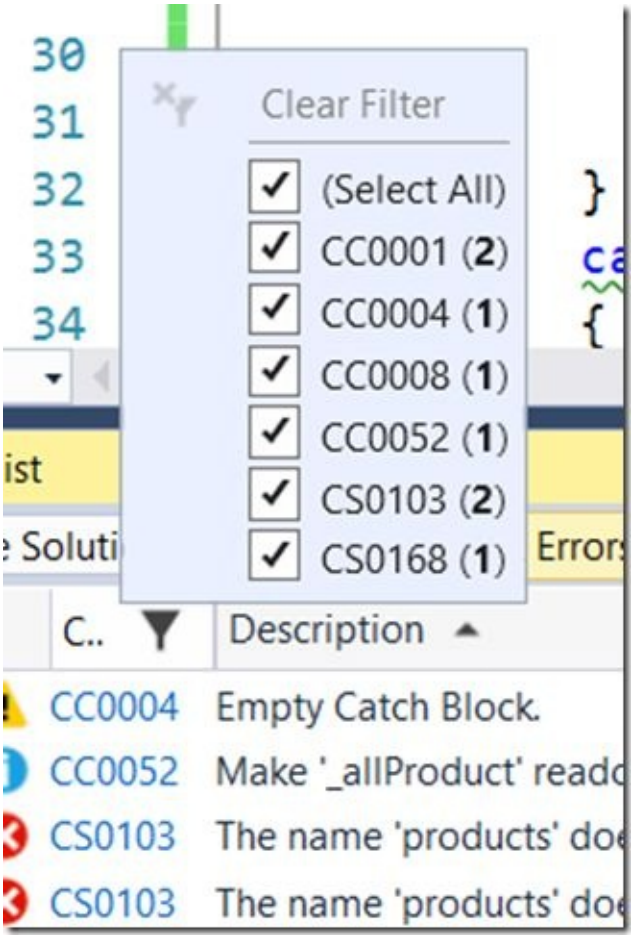
Filtering has been expanded more to filter on errors, warning and severity as well. To check that just click on top of the columns of error list where the error/warning symbol is displayed.



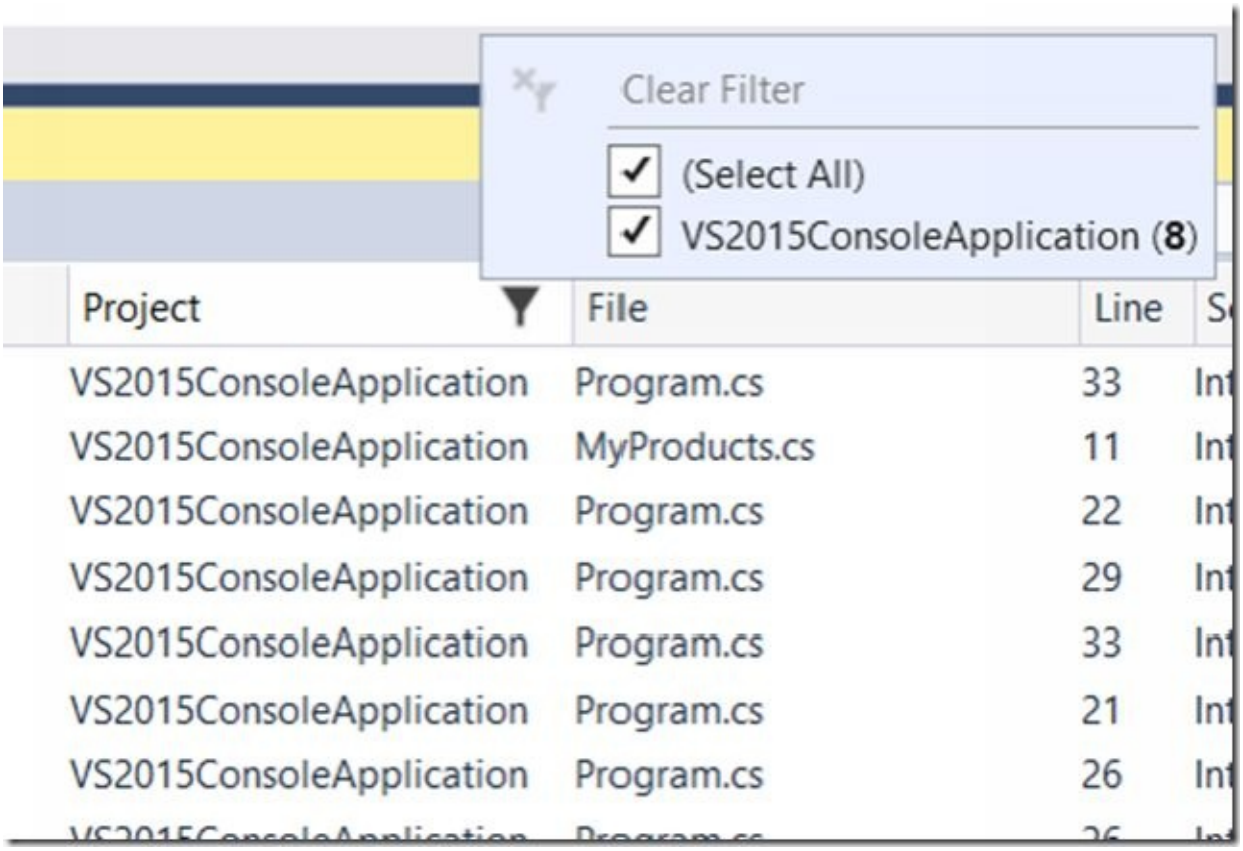
As soon as you click on the top as shown in the above image, the filter option will appear there itself and you can click that filter icon to see the types of more available filters.



You can choose to filter the list based on your selection by checking or un checking the check box. Filter option is widely available for code as well as for Projects. You can particularly select which code to include as shown below in the image,



Or which project or files to select as a filter.



So you can see that filtering option has been expanded to take care of multiple options therefore improving control, configurations and productivity of a developer.

Conclusion

In this chapter we covered the new improved debugging techniques that Visual Studio 2015 has come up with. We covered the break point configurations with several practical scenarios and sneak peeked in new improved Error list. These options can also be found in prior versions of Visual Studio, but VS 2015 has an improved and more discoverable version of them.

Tool Window Support for LINQ and Lambda Expressions

This chapter will cover another debugging improvement of Visual Studio 2015 i.e. tool window support for LINQ and Lambda expressions.

Prerequisites

Visual Studio 2015 Express has been used in this tutorial to explain the concepts and features. For samples and practice, a Visual Studio solution is created having a console application named VS2015ConsoleApplication.The console application contains a MyProduct class containing product as an entity specific basic operations like fetching the product, returning the list of products as shown below.

```
using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;


namespace VS2015ConsoleApplication

{

    public class MyProducts :IProducts

    {

    List<Product> _allProduct = new List<Product>();

    public MyProducts()

    {

    _allProduct.Add(new Product
{ProductCode="0001",ProductName="IPhone",ProductPrice="60000",ProductType="Phone",ProductDescription="App
IPhone" } );

    _allProduct.Add(new Product { ProductCode = "0002", ProductName = "Canvas", ProductPrice = "20000",
ProductType = "Phone", ProductDescription = "Micromax phone" });

    _allProduct.Add(new Product { ProductCode = "0003", ProductName = "IPad", ProductPrice = "30000",
ProductType = "Tab", ProductDescription = "Apple IPad" });

    _allProduct.Add(new Product { ProductCode = "0004", ProductName = "Nexus", ProductPrice = "30000",
ProductType = "Phone", ProductDescription = "Google Phone" });

    _allProduct.Add(new Product { ProductCode = "0005", ProductName = "S6", ProductPrice = "40000", ProductType
= "Phone", ProductDescription = "Samsung phone" });


    }


    /// <summary>

    /// FetchProduct having price greater that 3000

    /// </summary>

    /// <returns></returns>

    public List<Product> FetchProduct() => (from p in _allProduct where Convert.ToInt32(p.ProductPrice) > 30000
select p).ToList();


    /// <summary>
```

```
    /// FetchProduct
    /// </summary>
    /// <param name="pCode"></param>
    /// <returns></returns>
    public Product FetchProduct(string pCode)
    {
    return _allProduct.Find(p => p.ProductCode == pCode);
    }

    /// <summary>
    /// FetchProduct with productCode and productName
    /// </summary>
    /// <param name="productCode"></param>
    /// <param name="productName"></param>
    /// <returns></returns>
    public Product FetchProduct(string productCode, string productName)
    {
    return _allProduct.Find(p => p.ProductCode == productCode && p.ProductName==productName);
    }

    public List<Product> GetProductList()
    {
    return _allProduct;
    }
    }
}
```

where IProducts is a simple interface.

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

namespace VS2015ConsoleApplication
{
    interface IProducts
    {
    Product FetchProduct(string productCode);
```

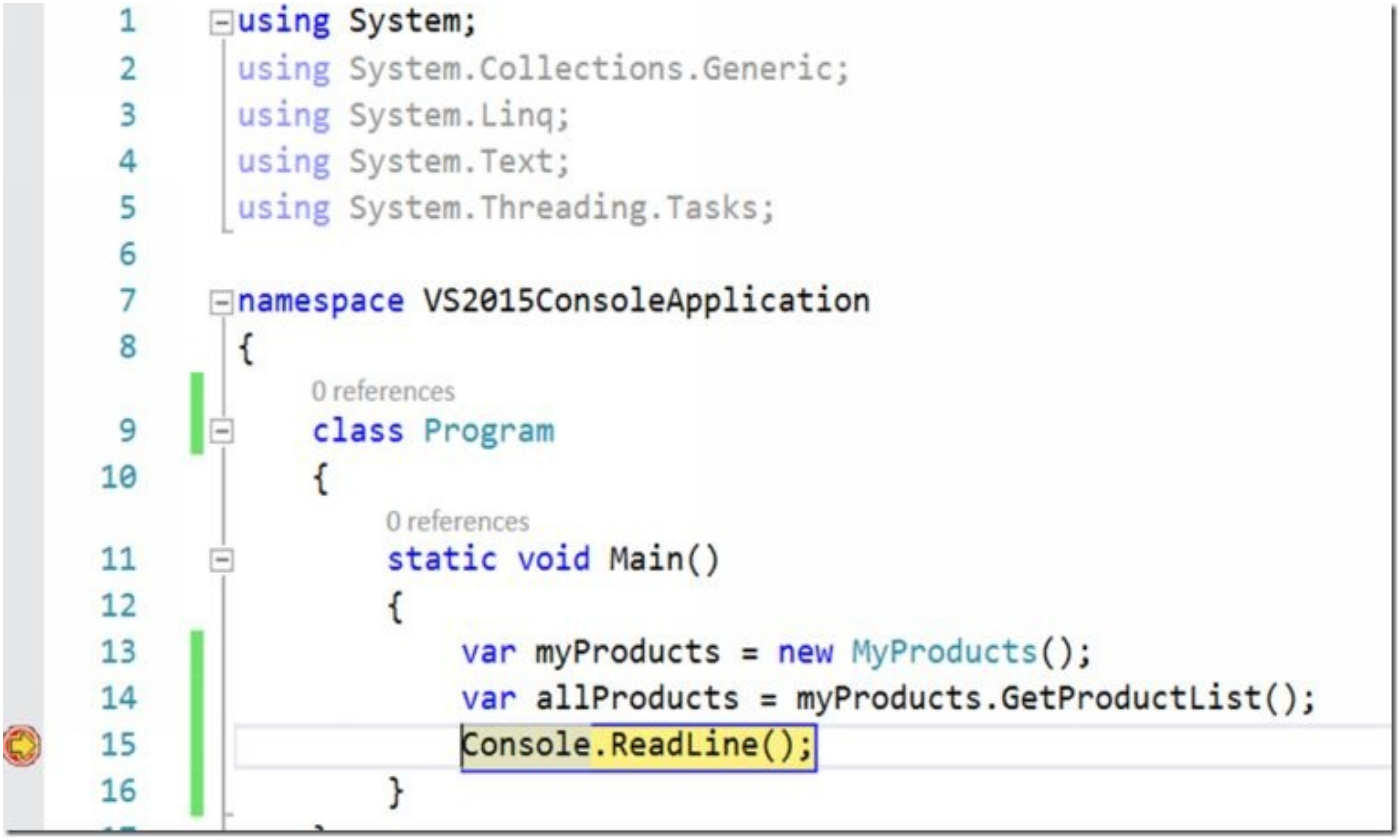
```
    Product FetchProduct(string productCode,string productName);
    List<Product> GetProductList();
    }
}
```

In the following Program.cs file the FetchProduct() method is called to get the list of all the products.

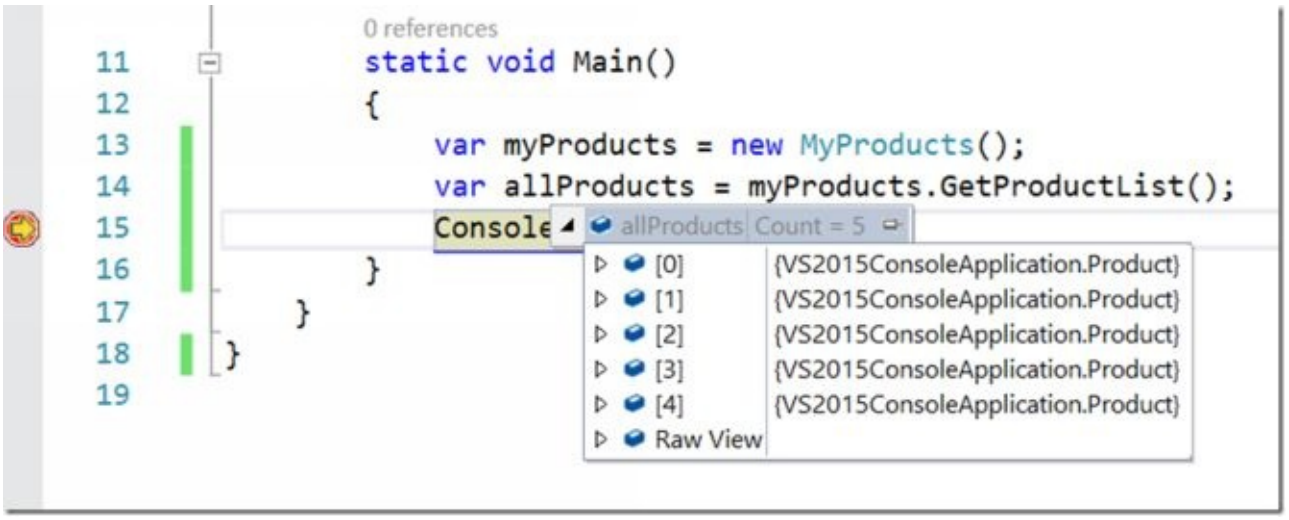
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace VS2015ConsoleApplication
{
    class Program
    {
    static void Main()
    {
    var myProducts = new MyProducts();
    var allProducts = myProducts.GetProductList();
    Console.ReadLine();
    }
    }
}
```

Tool window support for LINQ and lambda expressions

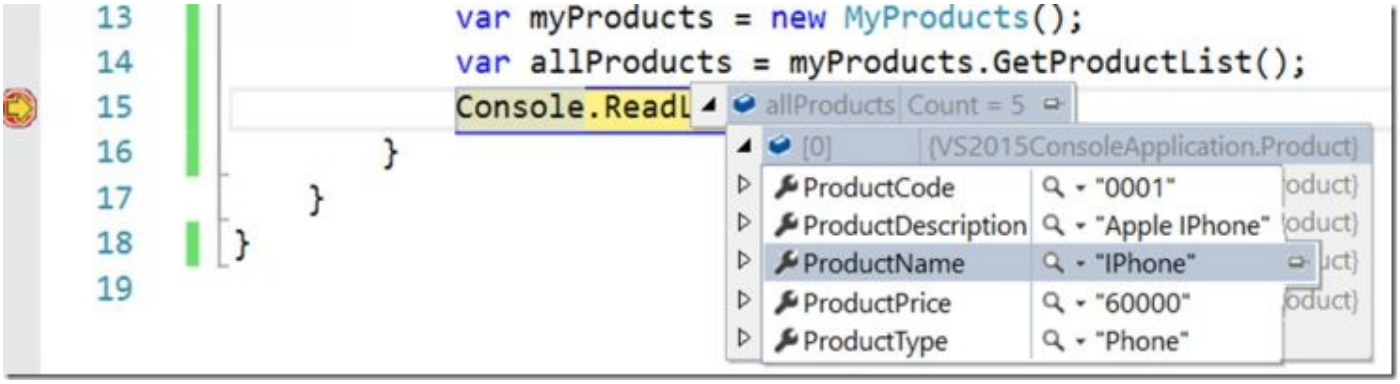
This is one of the best improvements that Visual Studio 2015 has come up with. In earlier versions of Visual Studio when a developer tries to execute/write LINQ or Lambda expressions in immediate window or watch window, the code was not supported and a message used to appear that LINQ and lambda expressions are not allowed in immediate or watch window. With the newly released product i.e. Visual Studio 2015, this limitation has been taken care of. Now a developer can take liberty to execute LINQ and Lambda expressions in immediate window. This feature proves to be very helpful in run time debugging the code, one can write LINQ queries in immediate windows at run time to select or filter the lists or objects. Let's cover the topic through practical examples. We have already a solution with a console application that fetches a list of Products. To practically check this tool support feature, place a breakpoint at Console.ReadLine() in program.cs i.e. when the list is fetched.



When we hover and try to see all products we get list of products as shown in below image.

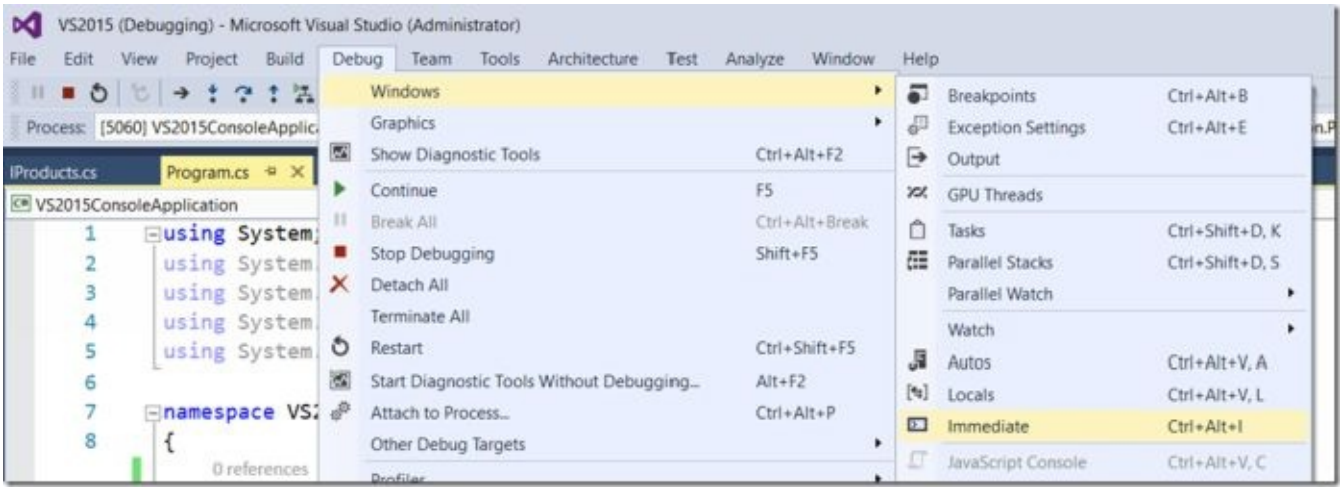


Now suppose there is a situation where developer wants to perform certain operations for debugging at this breakpoint like checking the product name, product price of each product entity or needs to see products only having price greater than 30000, then a developer has to explicitly navigate through each list item in the window opened as shown below.

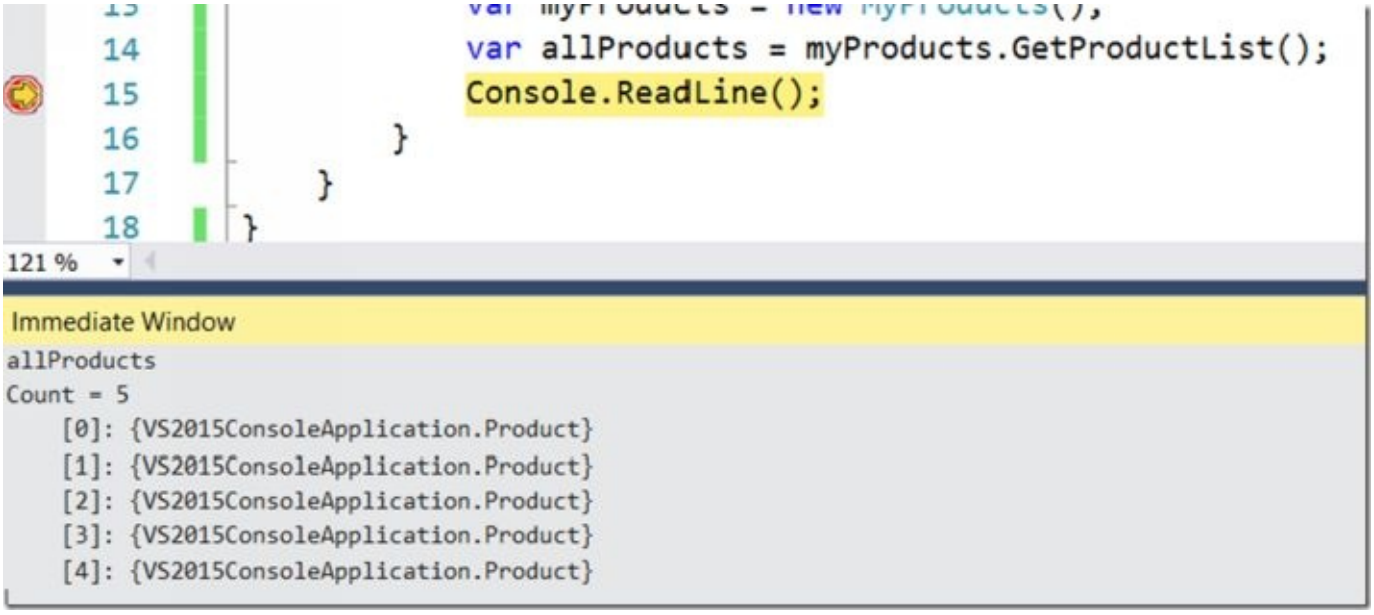


The above mentioned methodology for debugging and navigating is quite time consuming. Think about list having 1000's of records. One can leverage the support of LINQ and Lambda expressions in immediate window to expedite the debugging requirements now. Suppose one has a need to get list of all the ProductName in the list. let's see how we do this in immediate window.

Open the immediate window. You can open it through Visual Studio Debug menu as shown below. I.e. Debug->Windows->Immediate or you can use the keyboard default shortcut to open the window i.e. Ctrl+Alt+I.

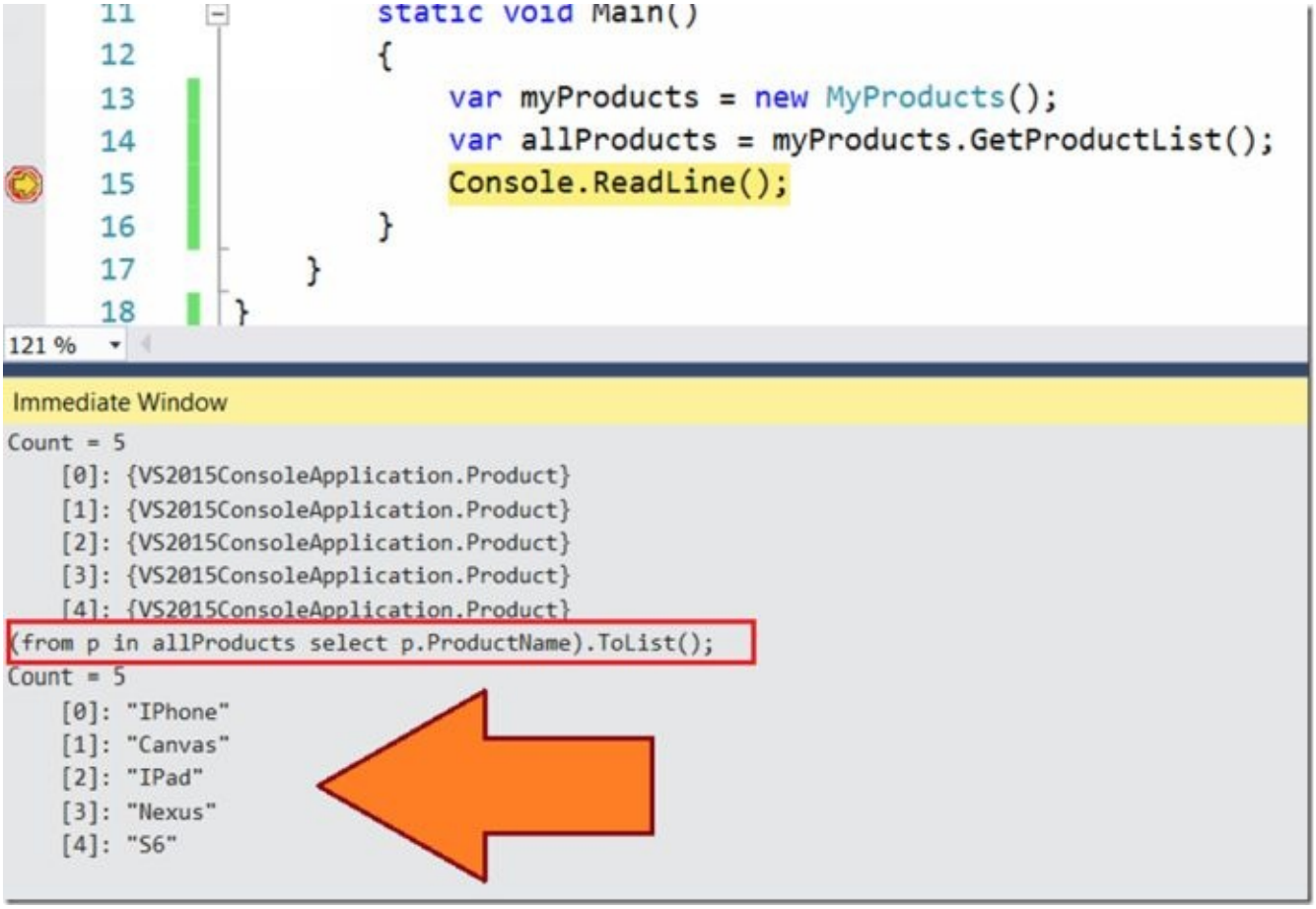


The immediate window will open at bottom of the class file and you can view the products list by typing allProducts in the window while the code execution is paused at breakpoint. You'll get the list of products as we saw while hovering the mouse over allProducts variable.



Now let's write a LINQ query to get all product names from the list of products and press enter. Notice that a list of ProductName is fetched through this LINQ query immediately.

(from p in allProducts select p.ProductName).ToList();



Likewise you can also perform filtering over the list or execute any LINQ query that you need to execute for the sake of debugging.

In earlier versions LINQ was not supported in immediate window. Now let’s try this with lambda expression.

allProducts.Select(p=>p.ProductName).ToList();

We get the same result.

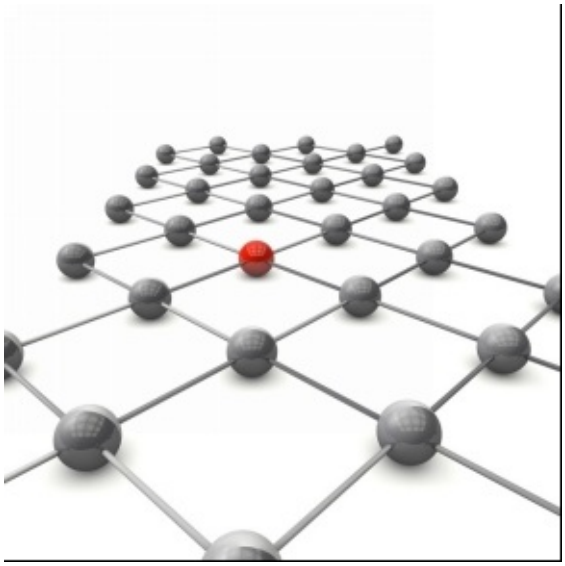
Conclusion

In this part of the Visual Studio 2015 book, we covered immediate window and Watch window support for LINQ and lambda expressions in Visual Studio. This is an extremely useful feature for debugging the collections or objects. We'll cover more debugging improvement features in upcoming parts. You can also use LINQ and lambda in the watch window.

PerfTip Feature in Visual Studio 2015

Introduction

In the earlier part of the book covered topics like breakpoint configuration improvements and new improved error list, tool window support for lambda and LINQ queries in Visual Studio 2015.This chapter will cover another debugging improvements of Visual Studio 2015 i.e. new PerfTip feature.



Prerequisites

Visual Studio 2015 Express has been used in this tutorial to explain the concepts and features. For samples and practice, a Visual Studio solution is created having a console application named VS2015ConsoleApplication.The console application contains a MyProduct class containing product as an entity specific basic operations like fetching the product, returning the list of products as shown below.

```
using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;


namespace VS2015ConsoleApplication

{

    public class MyProducts :IProducts

    {

    List<Product> _allProduct = new List<Product>();

    public MyProducts()

    {

    _allProduct.Add(new Product
{ProductCode="0001",ProductName="IPhone",ProductPrice="60000",ProductType="Phone",ProductDescription="App
IPhone" } );

    _allProduct.Add(new Product { ProductCode = "0002", ProductName = "Canvas", ProductPrice = "20000",
ProductType = "Phone", ProductDescription = "Micromax phone" });

    _allProduct.Add(new Product { ProductCode = "0003", ProductName = "IPad", ProductPrice = "30000",
ProductType = "Tab", ProductDescription = "Apple IPad" });

    _allProduct.Add(new Product { ProductCode = "0004", ProductName = "Nexus", ProductPrice = "30000",
ProductType = "Phone", ProductDescription = "Google Phone" });

    _allProduct.Add(new Product { ProductCode = "0005", ProductName = "S6", ProductPrice = "40000", ProductType
= "Phone", ProductDescription = "Samsung phone" });


    }


    /// <summary>

    /// FetchProduct having price greater that 3000

    /// </summary>

    /// <returns></returns>

    public List<Product> FetchProduct() => (from p in _allProduct where Convert.ToInt32(p.ProductPrice) > 30000
select p).ToList();


    /// <summary>
```

```
    /// FetchProduct
    /// </summary>
    /// <param name="pCode"></param>
    /// <returns></returns>
    public Product FetchProduct(string pCode)
    {
    return _allProduct.Find(p => p.ProductCode == pCode);
    }

    /// <summary>
    /// FetchProduct with productCode and productName
    /// </summary>
    /// <param name="productCode"></param>
    /// <param name="productName"></param>
    /// <returns></returns>
    public Product FetchProduct(string productCode, string productName)
    {
    return _allProduct.Find(p => p.ProductCode == productCode && p.ProductName==productName);
    }

    public List<Product> GetProductList()
    {
    return _allProduct;
    }
    }
}
```

where IProducts is a simple interface.

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

namespace VS2015ConsoleApplication
{
    interface IProducts
    {
    Product FetchProduct(string productCode);
```

```
    Product FetchProduct(string productCode,string productName);
    List<Product> GetProductList();
    }
}
```

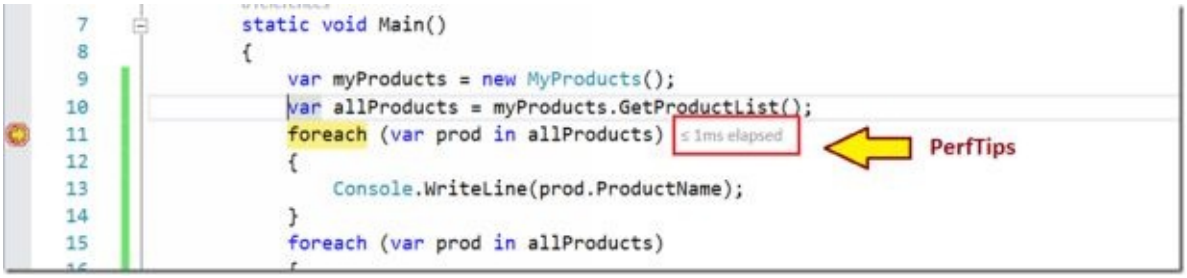
In the following Program.cs file the FetchProduct() method is called to get the list of all the products.

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

namespace VS2015ConsoleApplication
{
    class Program
    {
    static void Main()
    {
    var myProducts = new MyProducts();
    var allProducts = myProducts.GetProductList();
    foreach (var prod in allProducts)
    {
    Console.WriteLine(prod.ProductName);
    }
    foreach (var prod in allProducts)
    {
    if(Convert.ToInt32(prod.ProductPrice)>30000)
    Console.WriteLine(String.Format("ProductName : {0} and ProductPrice :
{1}",prod.ProductName,prod.ProductPrice));
    }
    Console.ReadLine();
    }
    }
}
```

New PerfTip Feature

The new PerfTip feature in Visual Studio 2015 provides performance centric information. A developer can get performance specific information without depending upon any tool, code or stopwatch implementation. While debugging the code, the PerfTip feature displays performance information in the form of tooltip. The tooltip shows the span of time for which the code was running. While stepping over the code, it displays the span of time that the code was running during that step. Even if we run from breakpoint to breakpoint, the PerfTip feature displays the span of time the code was running between the two breakpoints. Let's cover this feature through practical examples.

Let's place the breakpoint at the start of first foreach loop in Program.cs class, let's say we want to get the length of time elapsed while reaching to this line of code. Run the application.



You can see the PerTip appeared saying time elapsed is less than 1 millisecond. We did not used any stopwatch or code or any third party tool to get this time elapsed. It is the new perfTip feature of Visual Studio 2015 that shows the time elapsed during program execution till this breakpoint. Now if I again run the application but with one change. I have put Thread.Sleep for 10 milliseconds after both the foreach loops and place the breakpoint there. Now when you run the application and step over the Thread.Sleep(10) breakpoint, the PerfTip feature will add that elapsed 10 millisecond too while displaying the time elapsed in previous step.

It clearly says the time elapsed till previous step was less than or equal to 14 milliseconds. You can also get the time elapsed between two breakpoints. Let's try to do that. Let's put

the first breakpoint at the start of the method and another at last line of the method. Now when you run the application the code execution stops at first breakpoint. Simply press F5 or continue with the execution, the execution will halt at second breakpoint, therefore showing the total time elapsed for complete method to execute.

In our case it is less than 4 millisecond. You can leverage the capabilities of this feature for troubleshooting performance bottle necks in your application.

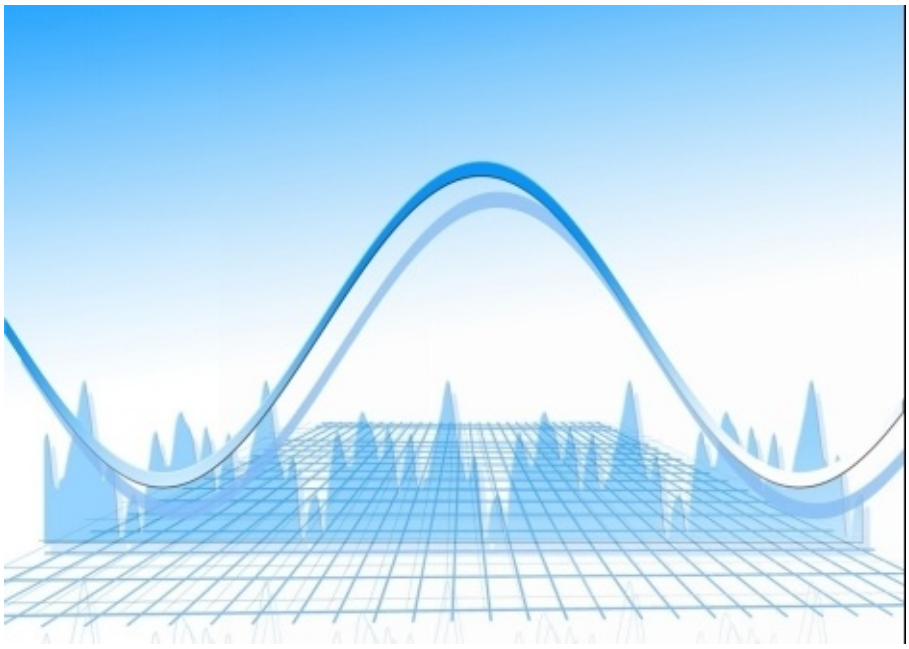
Conclusion

In this chapter we covered one of the critical feature of trouble shooting performance bottle necks i.e. PerfTips. In my next chapter I’ll introduce you to new Diagnostic Tool Window of Visual Studio 2015.We’ll learn how to use the new Diagnostic tool window and how to leverage its capability.

Diagnostic Tool Window in Visual Studio 2015

Introduction

The earlier parts of this book talked about Visual Studio 2015 improvements and enhancements. This chapter on Visual Studio 2015 will cover another interesting feature of Visual Studio 2015 i.e. Diagnostic Tool Window. We’ll have a glance over the new diagnostic window and try to cover the topic with practical examples, thus exploring hos this diagnostic tool window helps in debugging and monitoring.

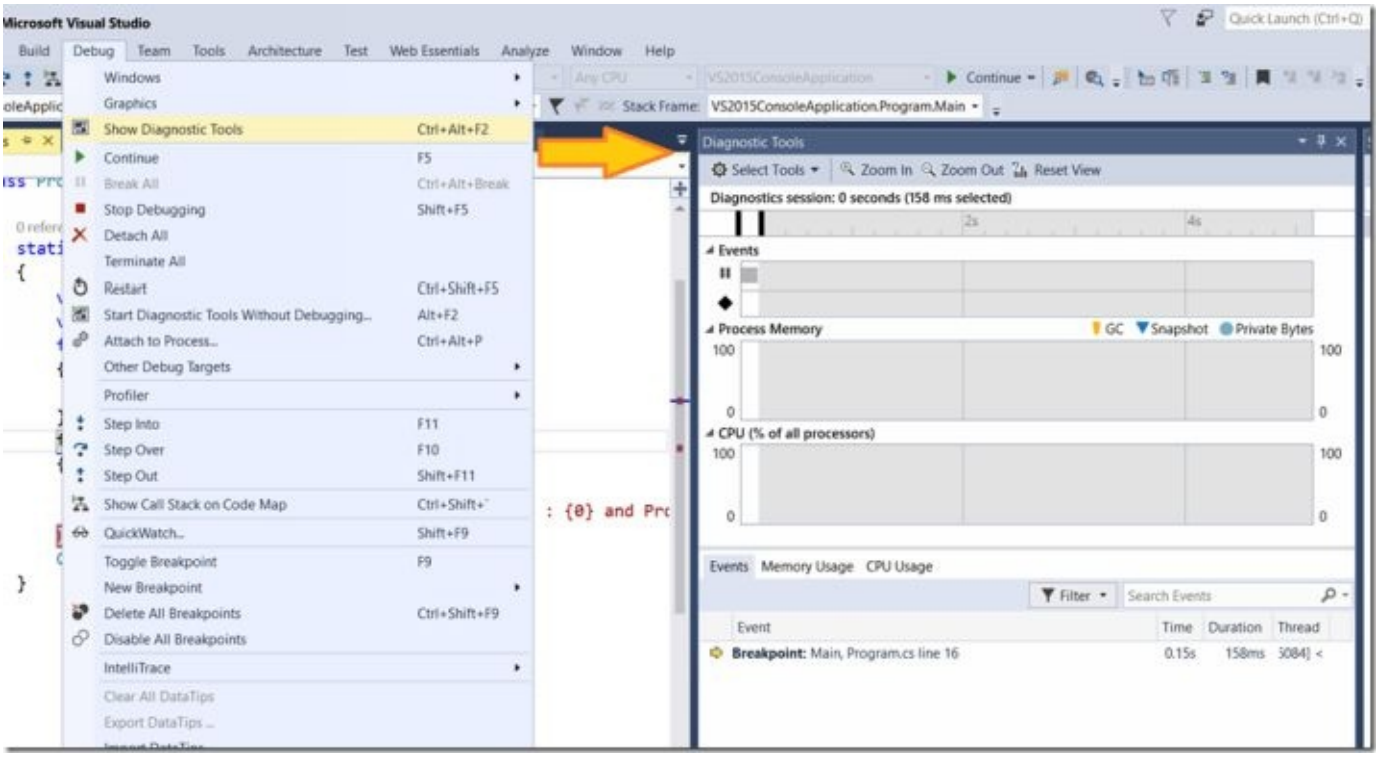


Diagnostic Tool Window

The new Diagnostic Tools window in Visual Studio appears whenever you run the application with debugging.

This window provides an ample amount of information that is very helpful when debugging the application. It contains a timeline across the top that provides a temporal display of application's debugging session.

The diagnostic tool window can be launched via Debug-> Show Diagnostic Tools option as shown in the below image.



Under the timeline are three important views.

The Debugger Events view displays breakpoint, output, and IntelliTrace events.

The Memory view displays the processor or memory usage in bytes.

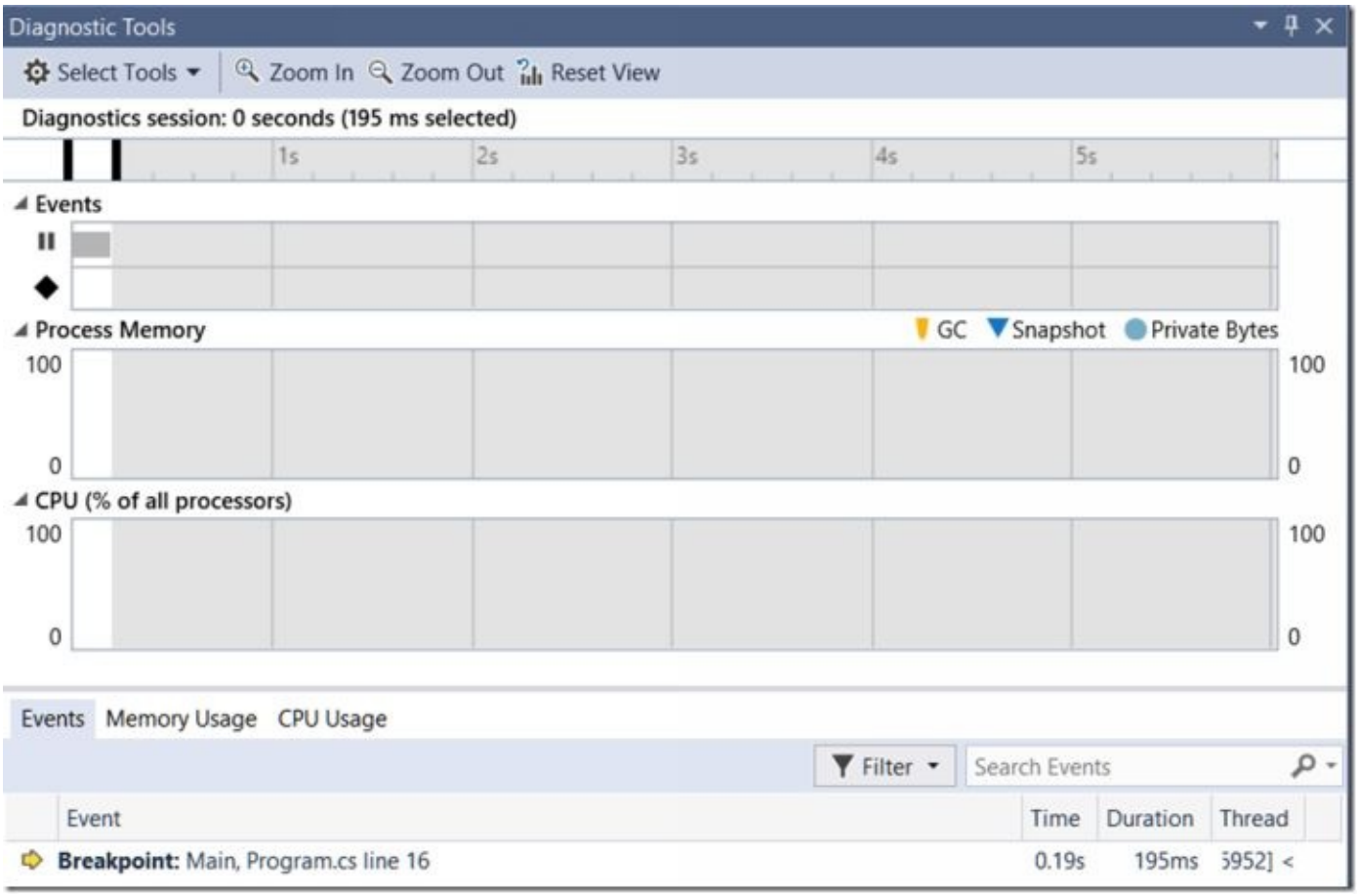
The CPU utilization view displays the percent of processor CPU usage.

At the bottom of the Diagnostic Tools window are tabs with additional detail and options. The yellow arrow at the bottom shows that code being debugged is currently at a breakpoint.

This chapter will cover the different views of Diagnostic Tool Window, one by one. We'll use the existing code base as a sample to explore the features. Place the breakpoints at two locations as shown below in the existing Program.cs file we have been using throughout the prior chapters.

```
0 references
class Program
{
    0 references
    static void Main()
    {
        var myProducts = new MyProducts();
        var allProducts = myProducts.GetProductList();
        foreach (var prod in allProducts)
        {
            Console.WriteLine(prod.ProductName);
        }
        foreach (var prod in allProducts)
        {
            if(Convert.ToInt32(prod.ProductPrice)>30000)
            Console.WriteLine(String.Format("ProductName : {0} and ProductPrice : {1}",prod.ProductName,prod.ProductPrice));
        }
        Console.ReadLine();
    }
}
```

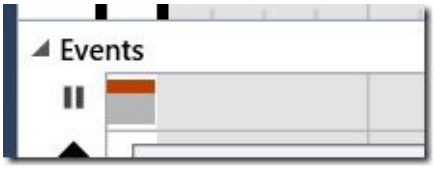
Now run the application and wait for execution to pause at break point. Now open the Diagnostic tools or press Ctrl+Alt+F2. You get following view of Diagnostic tool window.


Diagnostic Tools
Select Tools
Zoom In
Zoom Out
Reset View
Diagnostics session: 0 seconds (195 ms selected)
1s
2s
3s
4s
5s
Events
Process Memory
GC
Snapshot
Private Bytes
100
0
100
0
CPU (% of all processors)
100
0
100
0
Events
Memory Usage
CPU Usage
Filter
Search Events
Event
Time
Duration
Thread
Breakpoint: Main, Program.cs line 16
0.19s
195ms
5952] <

TimeLine

This example debugging session was 0.19 seconds in length from the time of application start until the current breakpoint, and we are looking at about 5.5 seconds of that time here in the timeline. The black bars here and here allow you to select a portion of the timeline.

The selected time interval, shown in white, is the area of the timeline that you want to focus on. Drag either black bar to adjust the selection area. The other sections of this window adjust based on the location of the black bars. Note that since our selected area includes part of a red line in the breakpoint track, the list also displays the end of that breakpoint line.



Debugger Events

The Debugger Events view in the Diagnostic Tools window displays events that occurred during the debugging session. This view is broken into tracks. The Break Events track shows events which stopped execution of the application, such as breakpoints that were hit, stepping through the code or breaking exceptions. The bars indicate the amount of time the application was running in between break events. The bars in this track are color coded, red indicates a breakpoint or breaking exception, yellow indicates a step through the code, and blue indicates a break all operation.

The blue bar indicates the time from the last breakpoint until the time that the debugging session was paused, using the Break All feature.

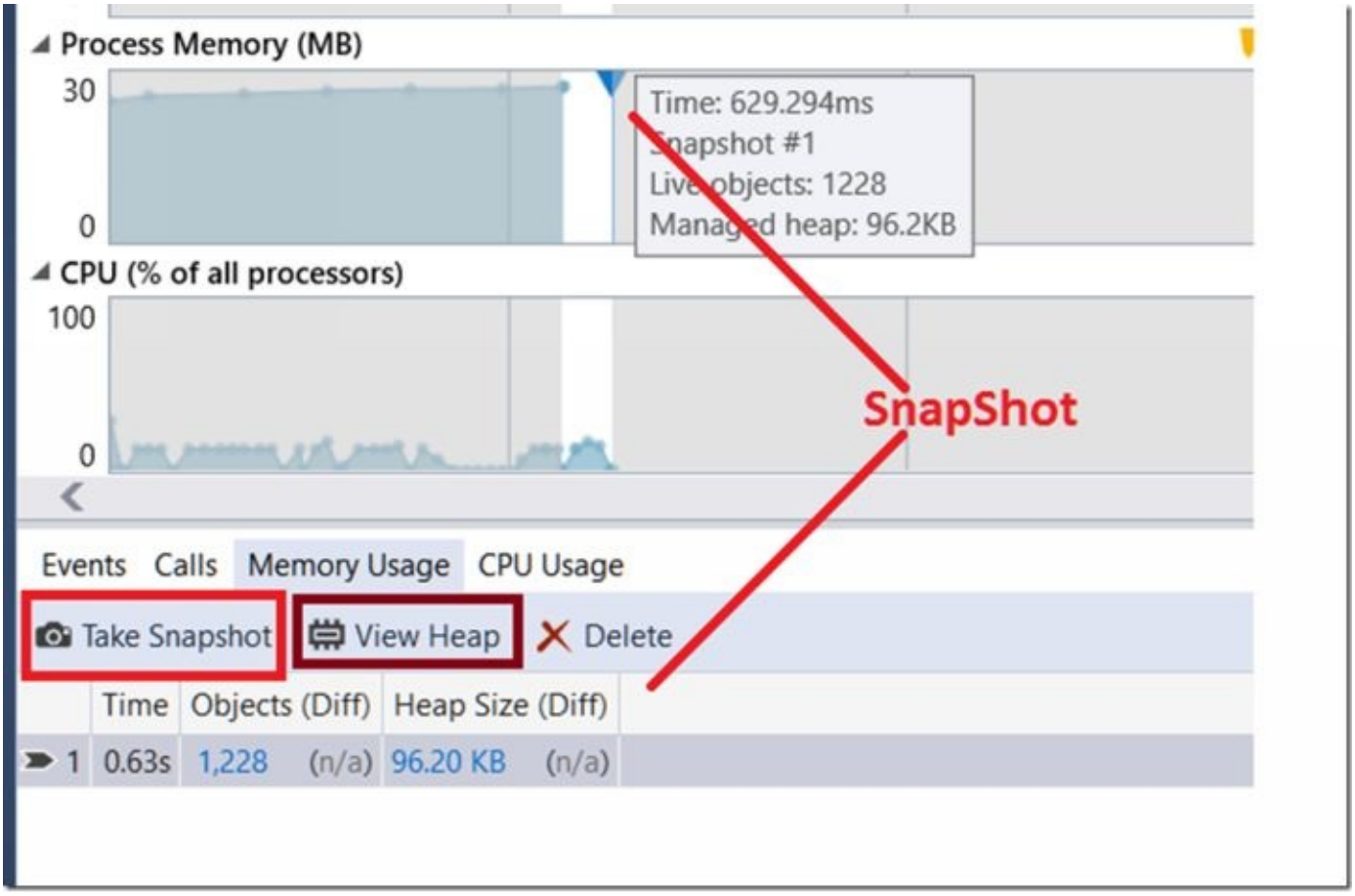
Hover over a bar to view the elapsed time.

The IntelliTrace track shows all of the other events collected by IntelliTrace, such as ADO.NET calls, file access, and so on. It is also color coded, black for important events, gray for less important events, and purple for custom events.

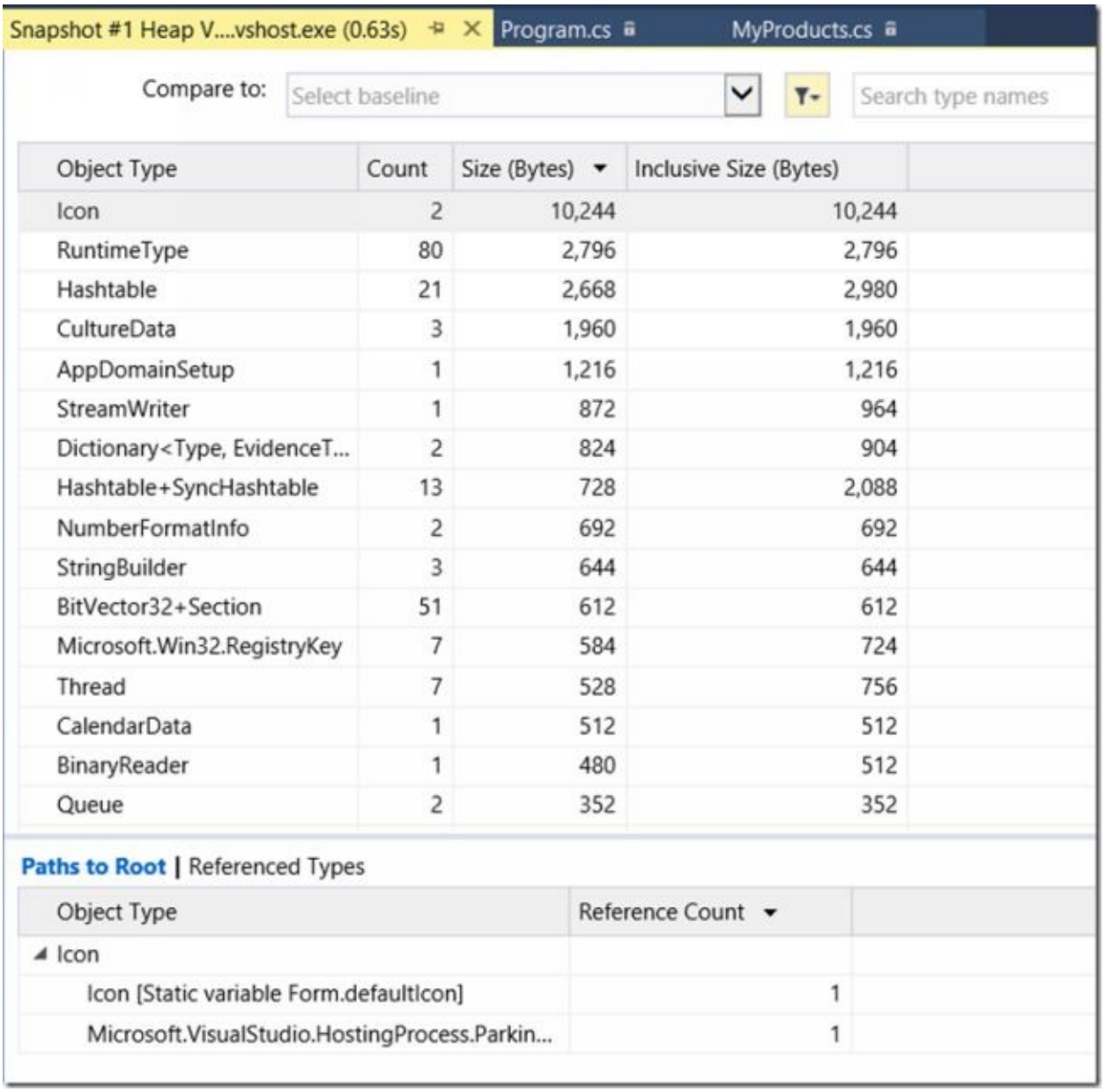
Process Memory

The Process Memory section of the Diagnostic Tools window helps you monitor the memory usage of the app while you are debugging. A memory usage tool was introduced in Visual Studio 2013 as part of the Performance and Diagnostics hub. In Visual Studio 2015, this tool was merged into the Diagnostic Tools window and enhanced. As with the current memory usage tool, you can take a memory snapshot at any point in time and compare it against another snapshot.

In this sample debugging session you can see there is one snapshot.

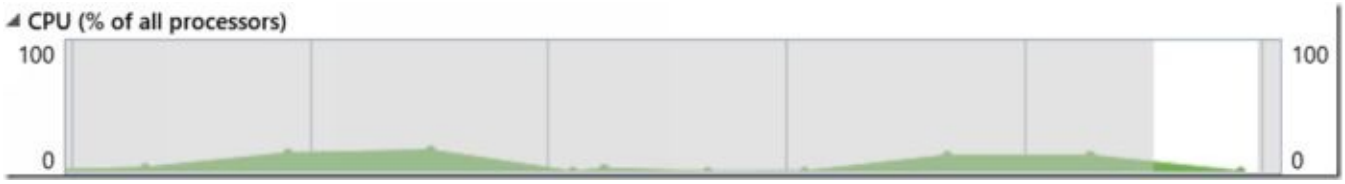


Using the memory usage tool helps you analyze the cause of memory spikes or leaks. You can also View the Heap information through Memory Usage window.


Snapshot #1 Heap V....vshost.exe (0.63s)
Program.cs
MyProducts.cs
Compare to:
Select baseline
Search type names
Object Type
Count
Size (Bytes)
Inclusive Size (Bytes)
Icon 2 10,244 10,244
RuntimeType 80 2,796 2,796
Hashtable 21 2,668 2,980
CultureData 3 1,960 1,960
AppDomainSetup 1 1,216 1,216
StreamWriter 1 872 964
Dictionary<Type, EvidenceT... 2 824 904
Hashtable+SyncHashtable 13 728 2,088
NumberFormatInfo 2 692 692
StringBuilder 3 644 644
BitVector32+Section 51 612 612
Microsoft.Win32.RegistryKey 7 584 724
Thread 7 528 756
CalendarData 1 512 512
BinaryReader 1 480 512
Queue 2 352 352
Paths to Root | Referenced Types
Object Type
Reference Count
Icon
Icon [Static variable Form.defaultIcon] 1
Microsoft.VisualStudio.HostingProcess.Parkin... 1

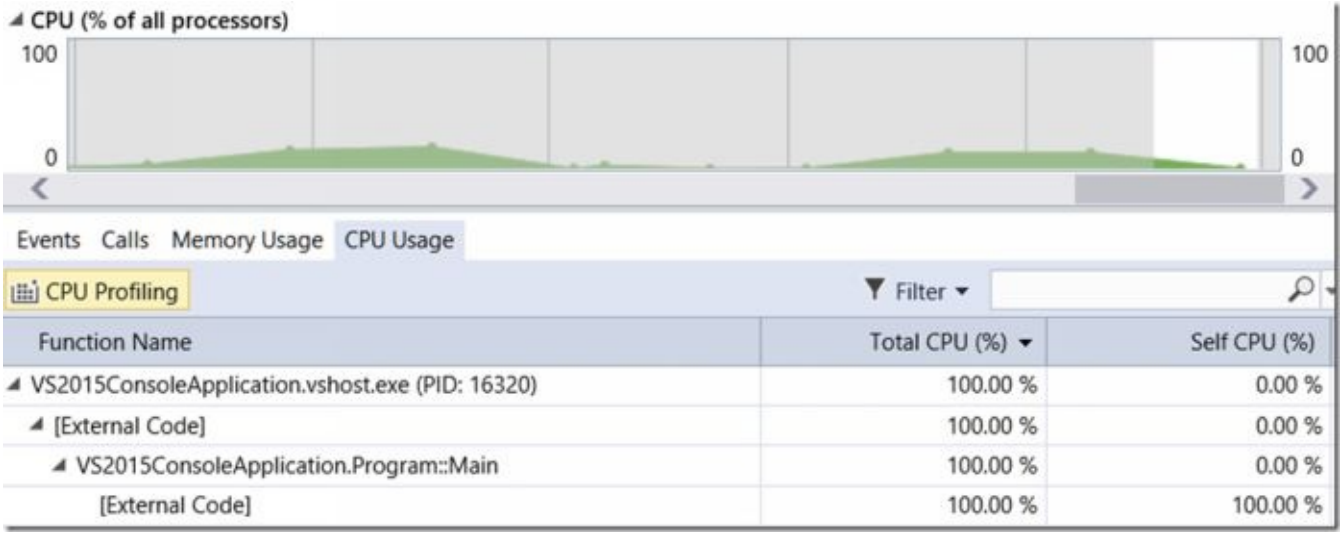
CPU

The CPU utilization section of the Diagnostic Tools window helps you monitor the CPU usage of your application by viewing a live CPU graph.



This graph shows you how much CPU resources are used by the code. Looking at the CPU usage with the debugger events helps you correlate spikes in CPU with the execution of the code, making it easier to debug CPU-related issues.

You can switch on CPU profiling for more detailed CPU usage statistics.



You can use the Diagnostic Tools window to aid in your debugging process and improve productivity and performance.

Conclusion

In this chapter we covered a very useful and critical feature of Visual Studio 2015, named Diagnostic Tools Window. We saw how Diagnostic tool window can help you debug your code and monitor CPU and memory usage too.

Index

P

Q

R

S

T

V