Mastering Debugging in Visual Studio 2015 - A Beginner's Guide

By **Abhijit Jana**

Describes all debugging features like Breakpoints, DataTips, Watch Windows, Multithreaded Debugging, Attaching to a process, Parallel Program Debugging and IntelliTrace Debugging

Table of Contents

Introduction

In the software development life cycle, testing and defect fixing take more time than actually code writing. In general, debugging is a process of finding out defects in the program and fixing them. Defect fixing comes after the debugging, or you can say they are co-related. When you have some defects in your code, first of all you need to identify the root cause of the defect, which is called the debugging. When you have the root cause, you can fix the defect to make the program behavior as expected.

Now how to debug the code? Visual Studio IDE gives us a lot of tools to debug our application. Sometimes debugging activity takes a very long time to identify the root cause. But VS IDE provides a lot of handy tools which help to debug code in a better way. Debugger features include error listing, adding breakpoints, visualize the program flow, control the flow of execution, data tips, watch variables and many more. Many of them are very common for many developers and many are not. In this article, I have discussed all the important features of VS IDE for debugging like Breakpoint, labeling and saving breakpoints, putting conditions and filter on breakpoints, DataTips, Watch windows, Multithreaded debugging, Thread window, overview of parallel debugging and overview of IntelliTrace Debugging. I hope this will be very helpful for beginners to start up with and for becoming an expert on debugging. Please note, targeted Visual Studio version is Visual Studio 2013. Many things are common in older versions, but many features such as Labeling breakpoint, Pinned DataTip, Multithreaded Debugging, Parallel debugging and IntelliTrace are added in VS 2013. Please provide your valuable suggestions and feedback to improve my article.

How to Start?

You can start debugging from the Debug menu of VS IDE. From the Debug Menu, you can select "Start Debugging" or just press F5 to start the program. If you have placed breakpoints in your code, then execution will begin automatically.

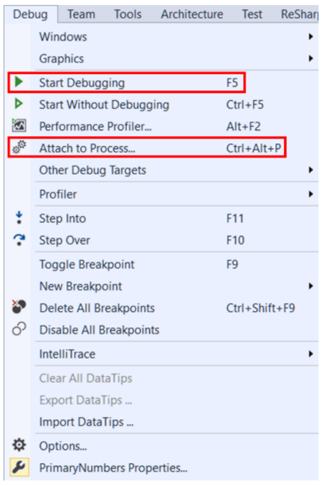


Figure: Start Debugging

There is another way to start the debugging by "Attach Process". Attach process will start a debug session for the application. Mainly we are very much familiar with the attach process debugging for ASP.NET Web Application. I have published two different articles on the same on CodeProject. You may have a look into this.

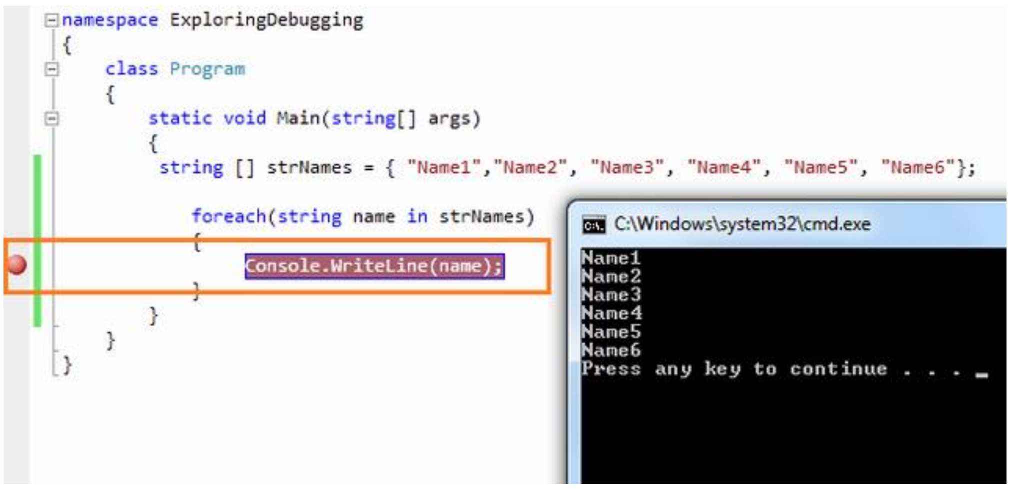
- Debug Your ASP.NET Application that Hosted on IIS

- Remote IIS Debugging: Debug your ASP.NET Application which is hosted on "Remote IIS Server"

We generally start debugging any application just by putting breakpoint on code where we think the problem may occur. So, let’s start with breakpoints.

Breakpoints

Breakpoint is used to notify debugger where and when to pause the execution of program. You can put a breakpoint in code by clicking on the side bar of code or by just pressing F9 at the front of the line. So before keeping a breakpoint, you should know what is going wrong in your code and where it has to be stopped. When the debugger reaches the breakpoint, you can check out what's going wrong within the code by using a different debugging tool.



```
namespace ExploringDebugging
{
    class Program
    {
        static void Main(string[] args)
        {
         string [] strNames = { "Name1","Name2", "Name3", "Name4", "Name5", "Name6"};

            foreach(string name in strNames)
            {
                Console.WriteLine(name);
            }
        }
    }
}
```

```
C:\Windows\system32\cmd.exe
Name1
Name2
Name3
Name4
Name5
Name6
Press any key to continue . . .
```

Figure: Set Breakpoint

Debugging with Breakpoints

You have already set a breakpoint in your code where you want to pause the execution. And now start the program by pressing **"F5"**. When the program reaches the breakpoint, execution will automatically pause. Now you have several options to check your code. After hitting the breakpoint, breakpoint line will show as yellow color which indicates that this is the line which will execute next.

Now you have several commands available in break mode, using which you can proceed for further debugging.

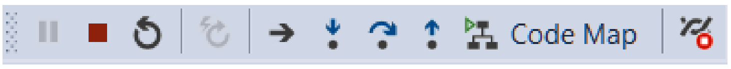
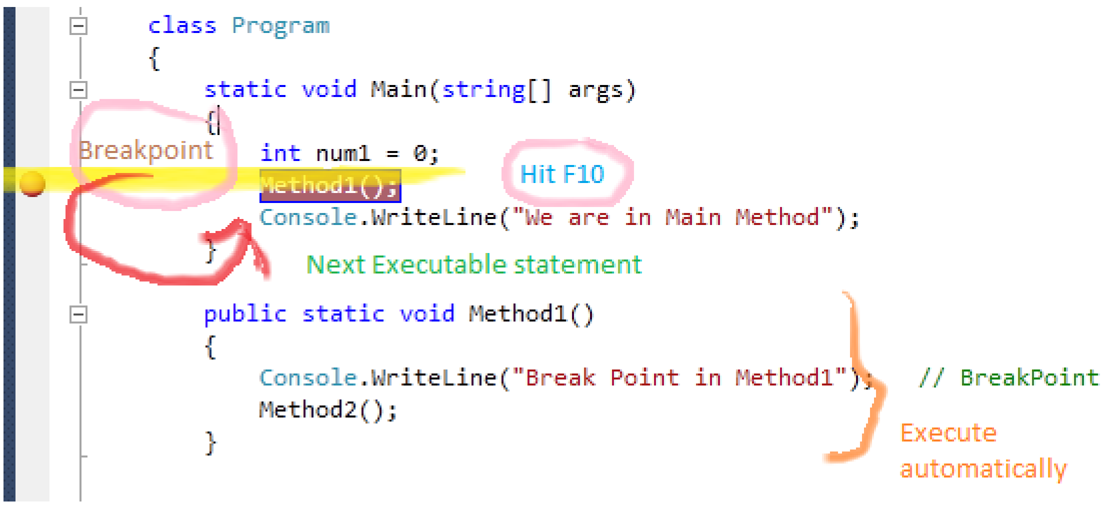


Figure: Breakpoint Toolbar

Step Over

After debugger hits the breakpoint, you may need to execute the code line by line. **"Step Over" [F10]** command is used to execute the code line by line. This will execute the currently highlighted line and then pause. If you select F10 while a method call statement is highlighted, the execution will stop after the next line of the calling statement. Step Over will execute the entire method at a time.

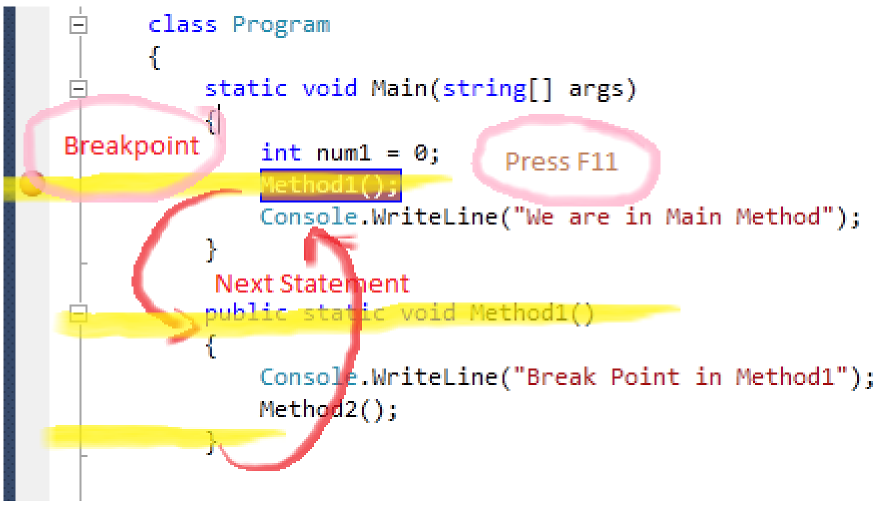


```
class Program
{
    static void Main(string[] args)
    {
        int num1 = 0;
        Method1();
        Console.WriteLine("We are in Main Method");
    }

    public static void Method1()
    {
        Console.WriteLine("Break Point in Method1");   // BreakPoint
        Method2();
    }
```

Figure: Step Over - F10

Step Into

This is similar to Step Over. The only difference is, if the current highlighted section is any methods call, the debugger will go inside the method. Shortcut key for Step Into is **"F11"**.



```
class Program
{
    static void Main(string[] args)
    {
        int num1 = 0;
        Method1();
        Console.WriteLine("We are in Main Method");
    }

    public static void Method1()
    {
        Console.WriteLine("Break Point in Method1");
        Method2();
    }
```

Figure: Step Into - F11

Step Out

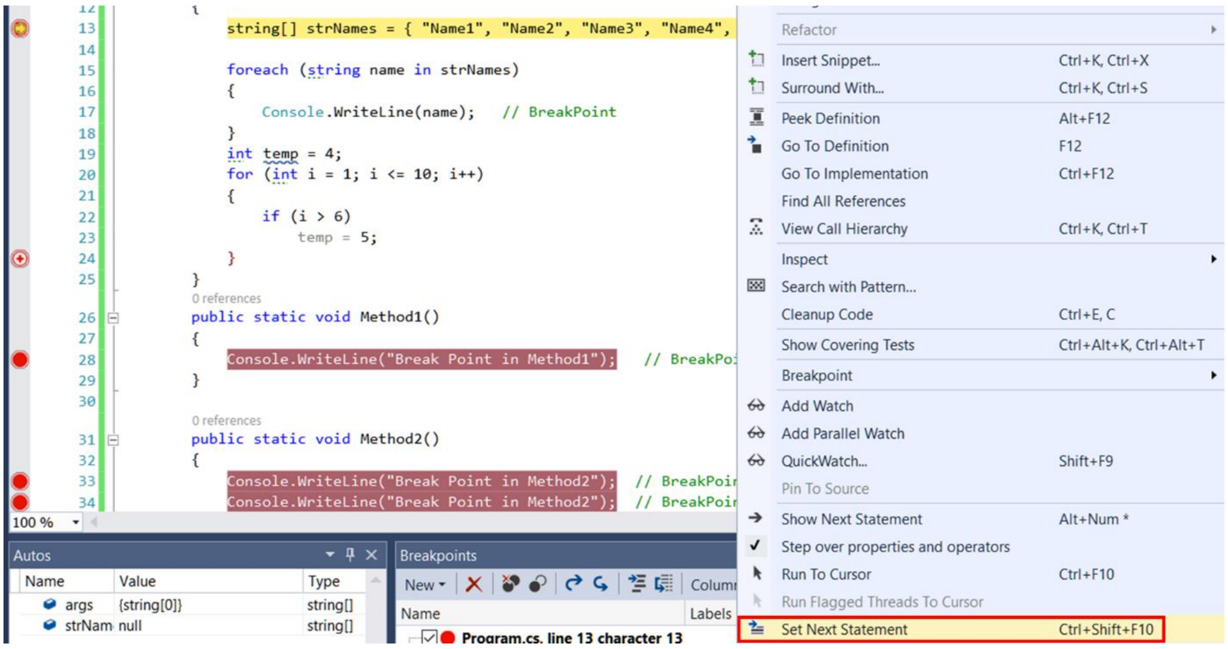
This is related when you are debugging inside a method. If you press the **Shift - F11** within the current method, then the execution will complete the execution of the method and will pause at the next statement from where it called.

Continue

It's like run your application again. It will continue the program flow unless it reaches the next breakpoint. The shortcut key for continue is **"F5"**.

Set Next Statement

This is quite an interesting feature. Set Next Statement allows you to **change the path of execution** of program while debugging. If your program paused in a particular line and you want to change the execution path, **go to the particular line**, **Right click** on the line and select **"Set Next Statement**" from the context menu. You will see, execution comes to that line without executing the previous lines of code. This is quite useful when you found some line of code may causing breaking your application and you don't want to break at that time. Shortcut key for Set Next Statement is **Ctrl + Shift + F10**.



```
{
    string[] strNames = { "Name1", "Name2", "Name3", "Name4",

    foreach (string name in strNames)
    {
        Console.WriteLine(name);   // BreakPoint
    }
    int temp = 4;
    for (int i = 1; i <= 10; i++)
    {
        if (i > 6)
            temp = 5;
    }
}
0 references
public static void Method1()
{
    Console.WriteLine("Break Point in Method1");    // BreakPo
}

0 references
public static void Method2()
{
    Console.WriteLine("Break Point in Method2");   // BreakPoir
    Console.WriteLine("Break Point in Method2");   // BreakPoir
```

Name	Value	Type
args	{string[0]}	string[]
strNam	null	string[]

Program.cs, line 13 character 13

100 %
Autos
Breakpoints
New
Column
Name
Labels
Refactor
Insert Snippet... Ctrl+K, Ctrl+X
Surround With... Ctrl+K, Ctrl+S
Peek Definition Alt+F12
Go To Definition F12
Go To Implementation Ctrl+F12
Find All References
View Call Hierarchy Ctrl+K, Ctrl+T
Inspect
Search with Pattern...
Cleanup Code Ctrl+E, C
Show Covering Tests Ctrl+Alt+K, Ctrl+Alt+T
Breakpoint
Add Watch
Add Parallel Watch
QuickWatch... Shift+F9
Pin To Source
Show Next Statement Alt+Num *
Step over properties and operators
Run To Cursor Ctrl+F10
Run Flagged Threads To Cursor
Set Next Statement Ctrl+Shift+F10

Figure: Set Next Statement

Show Next Statement [Ctrl+*]

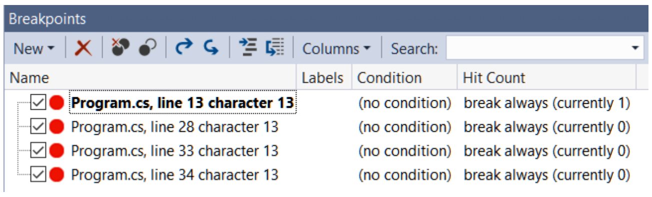
This line is marked as a yellow arrow. These lines indicate that it will be executed next when we continue the program.

Labeling in Break Point

This is the new feature in VS 2013. This is used for better managing breakpoints. It enables us to better group and filter breakpoints. It's kind of categorization of breakpoints. If you are having different types of breakpoints which are related with a particular functionality, you can give their name and can enable, disable, filter based on the requirements. To understand the whole functionality, let's assume that you have the below code block which you want to debug.

```
class Program
    {
        static void Main(string[] args)
        {
            string[] strNames = { "Name1", "Name2", "Name3", "Name4", "Name5", "Name6" };

            foreach (string name in strNames)
            {
                Console.WriteLine(name);   // BreakPoint
            }
            int temp = 4;
            for (int i = 1; i <= 10; i++)
            {
                if (i > 6)
                    temp = 5;
            }
        }

        public static void Method1()
        {
            Console.WriteLine("Break Point in Method1");    // BreakPoint
        }

        public static void Method2()
        {
            Console.WriteLine("Break Point in Method2");  // BreakPoint
            Console.WriteLine("Break Point in Method2");  // BreakPoint
        }

        public static void Method3()
        {
            Console.WriteLine("Break Point in Method3");  // Breakpoint
        }
    }
```

If you run the program, execution will pause on the first breakpoint. Now see the below picture, where you have the list of breakpoints.


Breakpoints

Name	Labels	Condition	Hit Count
☑ ● Program.cs, line 13 character 13		(no condition)	break always (currently 1)
☑ ● Program.cs, line 28 character 13		(no condition)	break always (currently 0)
☑ ● Program.cs, line 33 character 13		(no condition)	break always (currently 0)
☑ ● Program.cs, line 34 character 13		(no condition)	break always (currently 0)

Figure: Breakpoint List

In the given picture label column in blank. Now, see how you can set the label on break point and what is the use of it. To set label for any breakpoint, you just need to right click on the breakpoint symbol on the particular line or you can set it directly from breakpoint window.

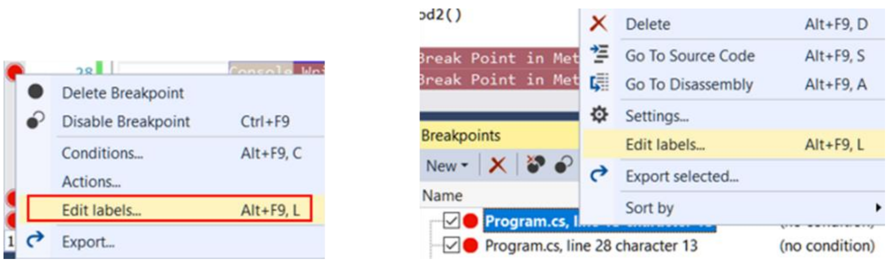
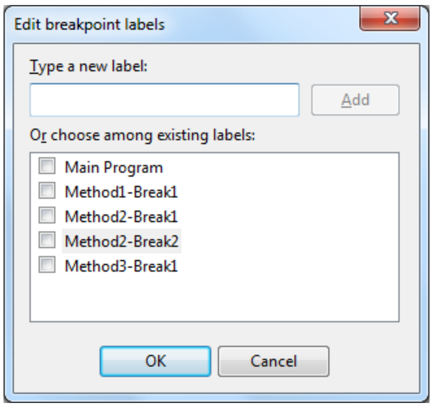


Figure: Setting Breakpoint Label

Right Click on Breakpoint, Click on the **Edit Labels** link, you can add the label for each and every breakpoints. As per the sample code, I have given very simple understandable names for all the breakpoints.


Edit breakpoint labels

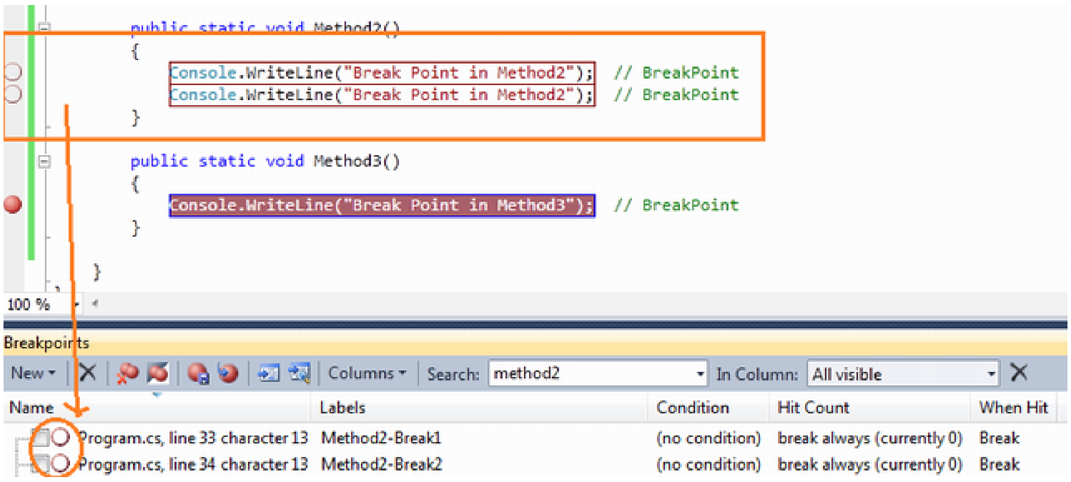
Type a new label:

Or choose among existing labels:

- ☐ Main Program
- ☐ Method1-Break1
- ☐ Method2-Break1
- ☐ Method2-Break2
- ☐ Method3-Break1

Figure: Adding Breakpoint Label

Let's have a look at how this labeling helps us during debugging. At this time, all the break points are enabled. Now if you don’t want to debug the `method2`, in a general case you need to go to the particular method and need to disable the breakpoints one by one, here you can filter/search them by label name and can disable easily by selecting them together.



```
public static void Method2()
{
    Console.WriteLine("Break Point in Method2");  // BreakPoint
    Console.WriteLine("Break Point in Method2");  // BreakPoint
}

public static void Method3()
{
    Console.WriteLine("Break Point in Method3");  // BreakPoint
}
}
```

Name	Labels	Condition	Hit Count	When Hit
Program.cs, line 33 character 13	Method2-Break1	(no condition)	break always (currently 0)	Break
Program.cs, line 34 character 13	Method2-Break2	(no condition)	break always (currently 0)	Break

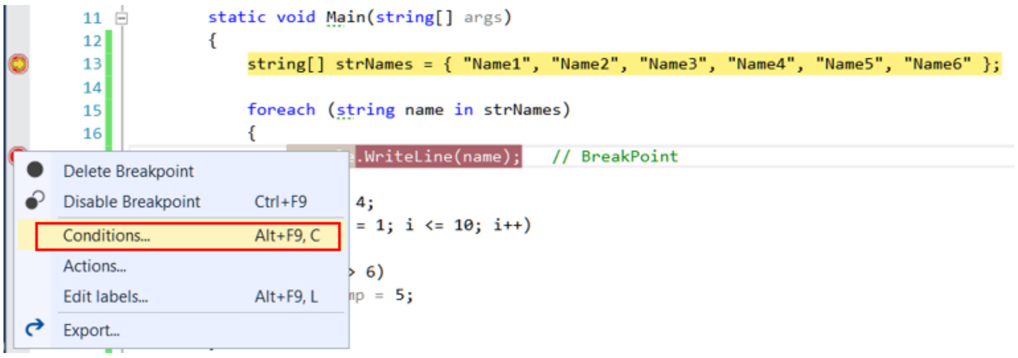
Figure: Filter Breakpoint Using Labels

This is all about the Breakpoint labeling. The example I have shown to you is very basic, but it is very much useful when you have huge lines of code, multiple projects, etc.

Conditional Breakpoint

Suppose you are iterating through a large amount of data and you want to debug a few of them. It means you want to pause your program on some specific condition. Visual Studio Breakpoints allow you to put conditional breakpoint. So if and only if that condition is satisfied, the debugger will pause the execution.

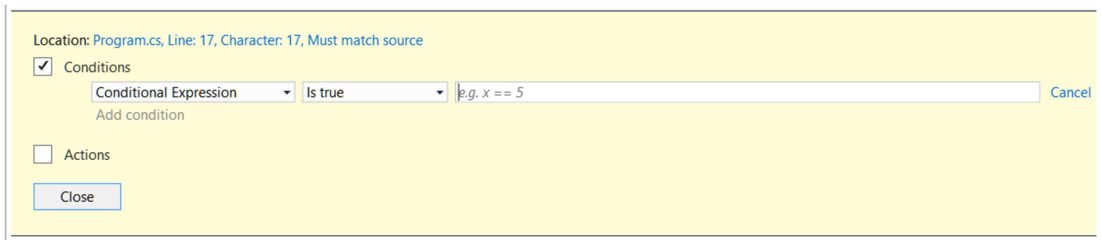
To do this, first of all you need to put the breakpoint on a particular line where you want to pause execution. Then just "Right Click" on the "Red" breakpoint icon. From there you just click on "Condition" .



```
static void Main(string[] args)
{
    string[] strNames = { "Name1", "Name2", "Name3", "Name4", "Name5", "Name6" };

    foreach (string name in strNames)
    {
        .WriteLine(name);   // BreakPoint

        4;
        = 1; i <= 10; i++)

        > 6)
        np = 5;
```

Figure: Set Breakpoint Condition

By clicking on the "Condition" link from context menu, the below screen will come where you can set the condition for breakpoints.



Location: Program.cs, Line: 17, Character: 17, Must match source

☑ Conditions

Conditional Expression | Is true | e.g. x == 5 | Cancel

Add condition

☐ Actions

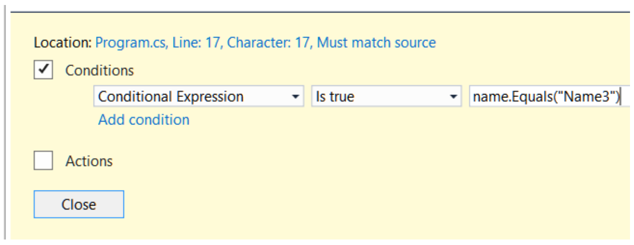
Close

Figure: Breakpoint Condition Settings

Let's assume that you have the following code block:

```
class Program
    {
        static void Main(string[] args)
        {
         string [] strNames = { "Name1","Name2", "Name3", "Name4", "Name5", "Name6"};

            foreach(string name in strNames)
            {
                 Console.WriteLine(name); // Breakpoint is here
            }
        }
    }
```

You have a breakpoint on `Console.WriteLine()` statement. On running of the program, execution will stop every time inside that `for-each` statement. Now if you want your code to break only when `name="Name3"`. What needs to be done? This is very simple, you need to give the condition like `name.Equals("Name3")`.



Location: Program.cs, Line: 17, Character: 17, Must match source

☑ Conditions

Conditional Expression | Is true | name.Equals("Name3")

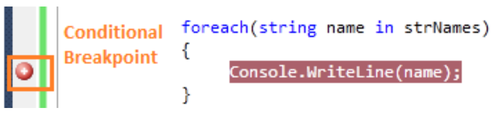
Add condition

☐ Actions

Close

Figure: Set Breakpoint Condition

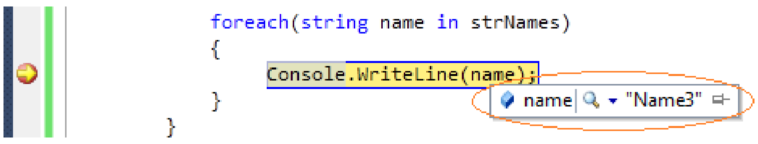
Check the Breakpoint Symbol. It should look like a plus (+) symbol inside the breakpoint circle which indicates the conditional breakpoints.



```
foreach(string name in strNames)
{
    Console.WriteLine(name);
}
```

Figure: Conditional Breakpoint Symbol

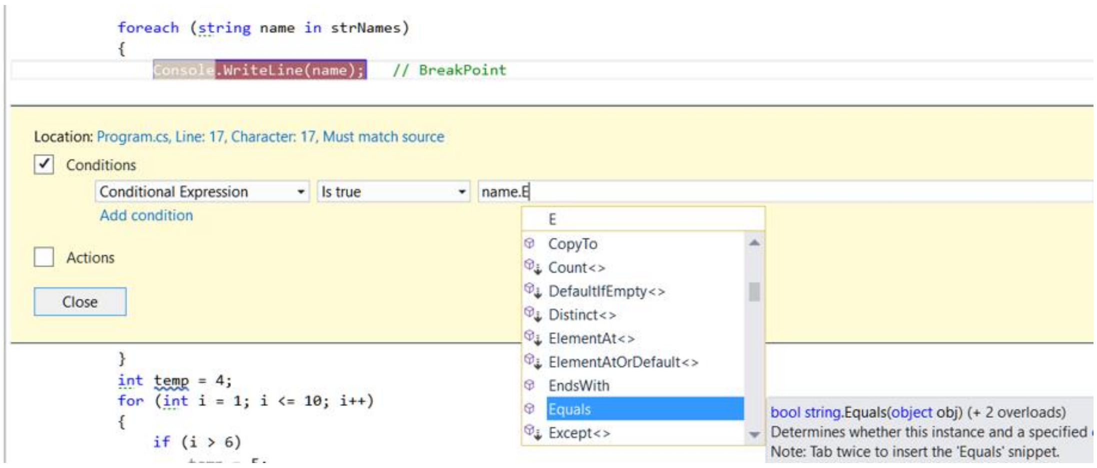
After setup of the condition of your breakpoint, if you run the application to debug it, you will see execution of program is only paused when it satisfied the given condition with breakpoint. In this case when `name="Name3"`.



```
foreach(string name in strNames)
{
    Console.WriteLine(name);
}
    }
```

Figure: Conditional Breakpoint hit

intellisense In Condition Text Box: The breakpoint condition which I have demonstrated here is very simple and can be written easily inside condition textbox. Sometimes, you may need to specify too big or complex conditions also. For that, you do not need to worry, VS IDE provide the intellisense within the condition textbox also. So whenever you are going to type anything inside the condition box, you will feel like typing inside the editor itself. Have a look into the below picture.



```
foreach (string name in strNames)
{
    Console.WriteLine(name);   // BreakPoint
```

Location: Program.cs, Line: 17, Character: 17, Must match source

✓ Conditions

Conditional Expression | Is true | name.E

Add condition

Actions

Close

```
}
int temp = 4;
for (int i = 1; i <= 10; i++)
{
    if (i > 6)
```

bool string.Equals(object obj) (+ 2 overloads)
Determines whether this instance and a specified
Note: Tab twice to insert the 'Equals' snippet.

Figure: intellisense in condition textbox

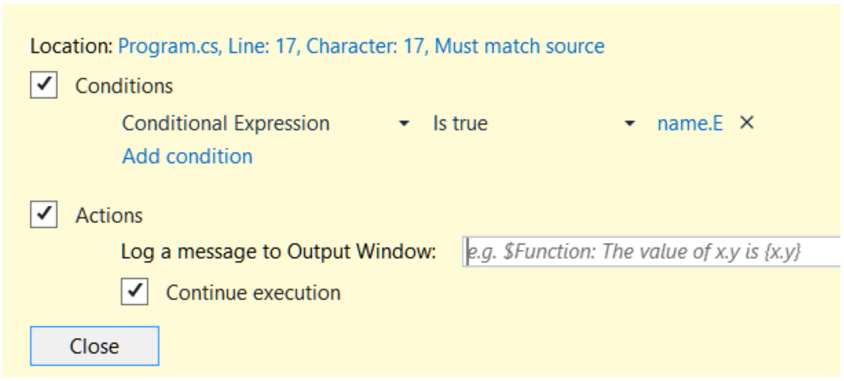
I have almost covered all about the conditional breakpoints except one thing. In condition window you have seen that there are two options available:

1. **Is True** and
2. **When Changed**

We have already seen what is the use of **"Is True"** option. **"When changed"** is used when you want to break the code if some value has changed for some particular value.

Action

It's possible also to define actions. For example, to define some message with be printed to Output Window.



Location: Program.cs, Line: 17, Character: 17, Must match source

✓ Conditions

Conditional Expression | Is true | name.E ×

Add condition

✓ Actions

Log a message to Output Window: e.g. $Function: The value of x.y is {x.y}

✓ Continue execution

Close

Import / Export Breakpoint

This is another interesting feature where you can save breakpoints and can use them in future. Visual Studio saves breakpoints in an XML Format. To save the breakpoints, you just need to click on the **"Export"** button in breakpoint window.

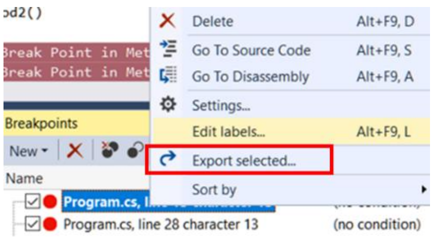
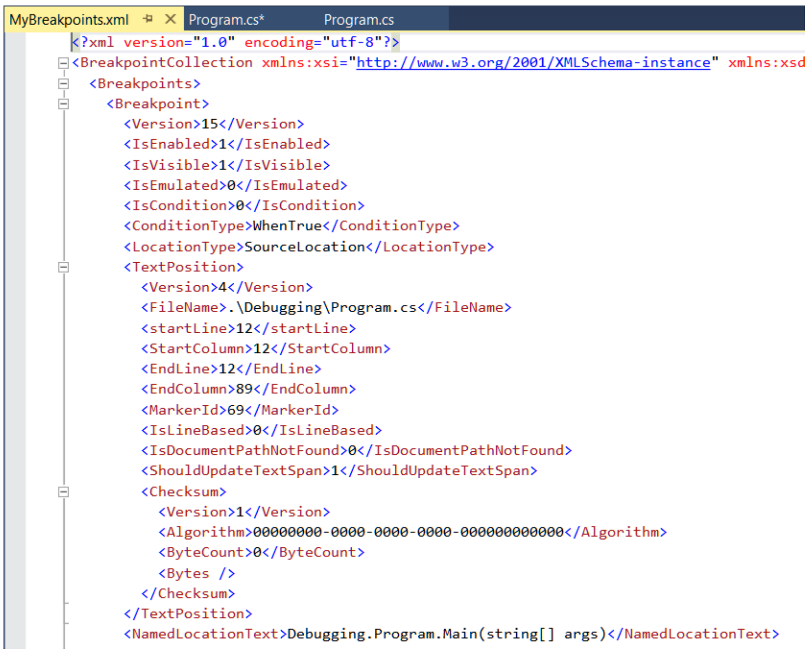


Figure: Save Breakpoints

You can use the saved XML file for the future and you can pass the same to other developers. You can also save breakpoints based on the search on labels. Let's have a quick look inside the content of the XML File. The XML file is collection of `BreakPoints` tag within `BreakpointCollection`. Each breakpoints tag contains information about the particular breakpoint like *line number*, *is enabled*, etc.



```
<?xml version="1.0" encoding="utf-8"?>
<BreakpointCollection xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd
  <Breakpoints>
    <Breakpoint>
      <Version>15</Version>
      <IsEnabled>1</IsEnabled>
      <IsVisible>1</IsVisible>
      <IsEmulated>0</IsEmulated>
      <IsCondition>0</IsCondition>
      <ConditionType>WhenTrue</ConditionType>
      <LocationType>SourceLocation</LocationType>
      <TextPosition>
        <Version>4</Version>
        <FileName>.\Debugging\Program.cs</FileName>
        <startLine>12</startLine>
        <StartColumn>12</StartColumn>
        <EndLine>12</EndLine>
        <EndColumn>89</EndColumn>
        <MarkerId>69</MarkerId>
        <IsLineBased>0</IsLineBased>
        <IsDocumentPathNotFound>0</IsDocumentPathNotFound>
        <ShouldUpdateTextSpan>1</ShouldUpdateTextSpan>
        <Checksum>
          <Version>1</Version>
          <Algorithm>00000000-0000-0000-0000-000000000000</Algorithm>
          <ByteCount>0</ByteCount>
          <Bytes />
        </Checksum>
      </TextPosition>
      <NamedLocationText>Debugging.Program.Main(string[] args)</NamedLocationText>
```

Figure: Breakpoint XML File

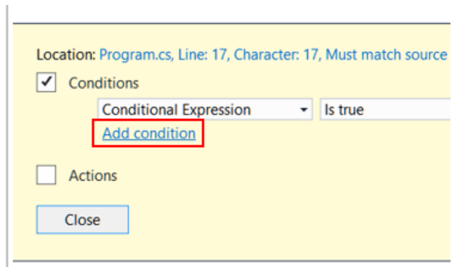
If you delete all the breakpoints from your code at any time, you can easily import them by just clicking on the "`Import`" breakpoints button. This will restore all of your saved breakpoints.

Note: Breakpoint Import depends on the line number where you have set your breakpoint earlier. If your line number changed, breakpoint will set on the previous line number only, so you will get breakpoints on unexpected lines.

Breakpoint Hit Count

Breakpoint Hit Count is used to keep track of how many times the debugger has paused at some particular breakpoint. You also have some option like to choose when the debugger will stop. **"Breakpoint Hit Count"** window having the following:

1. Break always
2. Break when the hit count is equal to a specified number
3. Break when the hit count is a multiple of a specified number
4. Break when the hit count is greater than or equal to a specified number.

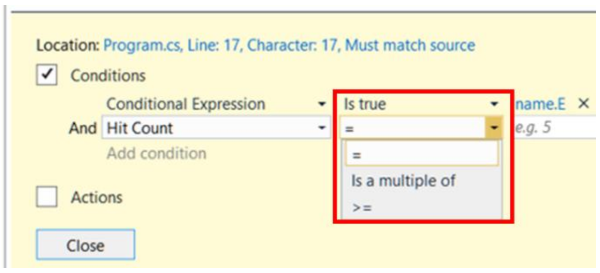

Location: Program.cs, Line: 17, Character: 17, Must match source

Conditions ✓

Conditional Expression | Is true

Figure: Breakpoint Hit Count

By default, it's set to always. So, whenever the breakpoints hits, hit count will increase automatically. Now we can set some condition as earlier mentioned. So based on that condition, breakpoint will hit and counter will be increased.


Location: Program.cs, Line: 17, Character: 17, Must match source

Conditions ✓

Conditional Expression | Is true | name.E

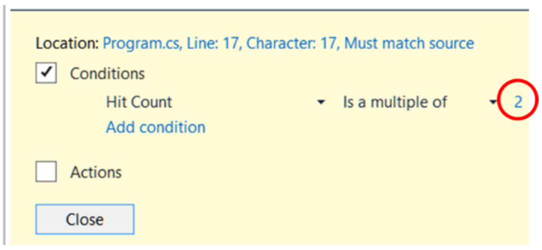
And Hit Count | = | e.g. 5

Figure: Breakpoint Hit Count Options

Let's explore it with the sample code block below:

```
for (int i = 0; i <= 10; i++)
{
    Console.WriteLine(i.ToString());  // Breakpoint
}
```

If you want your code to break when hit count is a multiple of 2, you need to select the third option from the dropdown list. After that, your code line should break only when the hit count is 2,4,6,... etc.


Location: Program.cs, Line: 17, Character: 17, Must match source

Conditions ✓

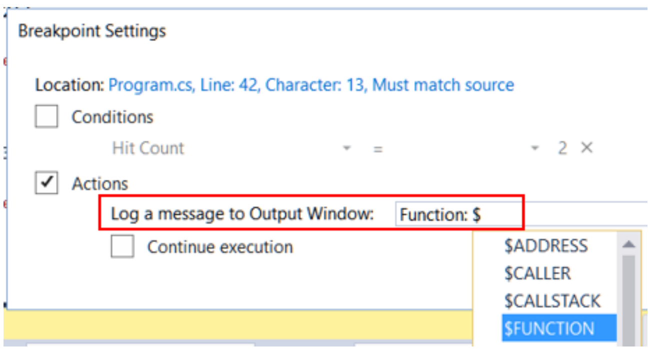
Hit Count | Is a multiple of | 2

Figure: Hit Count Condition

Till now, I hope you are very much clear about breakpoints, labeling, hit count, etc. Now have a look at what is **"Breakpoint When Hit"** option.

Breakpoint When Hit

This is apart from your normal debug steps, you want to do something else while breakpoint is hit, like print a message or run some macros. For those type of scenarios, you may go for this option. You can open this window by just right clicking on breakpoint.


Breakpoint Settings

Location: Program.cs, Line: 42, Character: 13, Must match source

☐ Conditions

Hit Count = 2

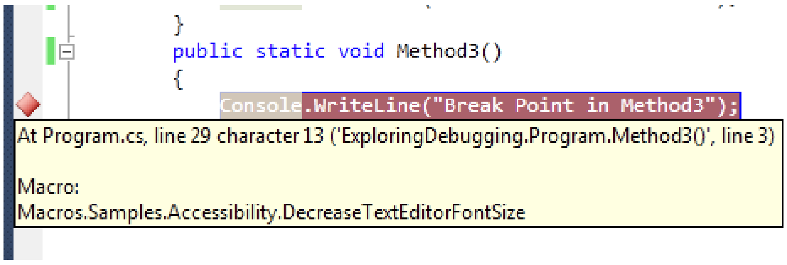
✓ Actions

Log a message to Output Window: Function: $

☐ Continue execution

Figure: When Breakpoint Is Hit

The first thing that you need to notice is the symbol of breakpoint. Breakpoint symbol has changed to a diamond and you can also check out the tool tip message which indicates what it is going to do when execution reaches here.



```
        }
        public static void Method3()
        {
            Console.WriteLine("Break Point in Method3");
```

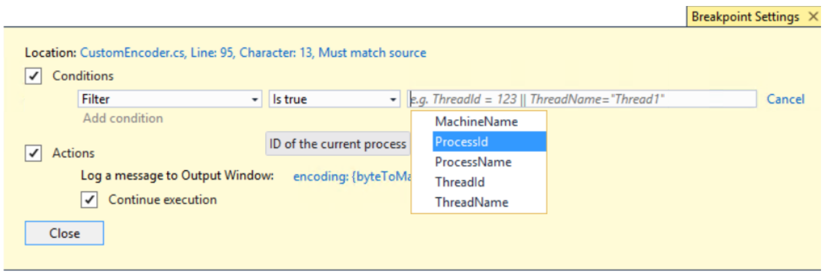
At Program.cs, line 29 character 13 ('ExploringDebugging.Program.Method3()', line 3)

Macro:
Macros.Samples.Accessibility.DecreaseTextEditorFontSize

Figure: When Breakpoint Is Hit

Breakpoint Filter

You can restrict the breakpoint to hit for certain processes or threads. This is extremely helpful while you are dealing with multithreading program. To open the filter window, you need to right click on the breakpoint and select **"Filter"**.


Breakpoint Settings

Location: CustomEncoder.cs, Line: 95, Character: 13, Must match source

✓ Conditions

Filter | Is true | e.g. ThreadId = 123 || ThreadName="Thread1"

Add condition

✓ Actions

Log a message to Output Window: encoding: {byteToMa

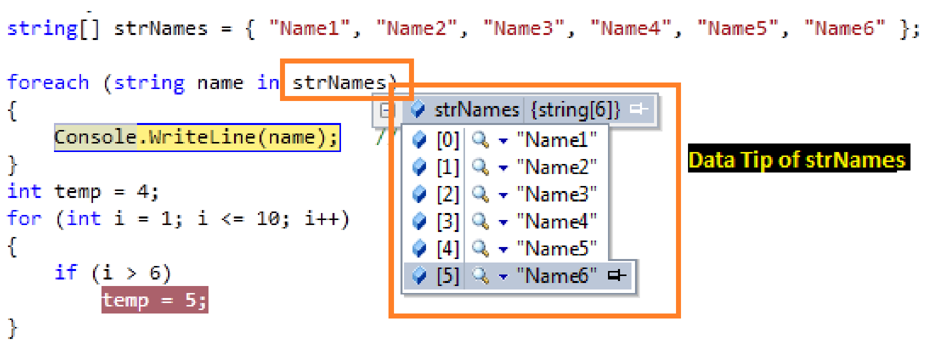
✓ Continue execution

Figure: Breakpoint Filter

In the filter criteria, you can set the process name, Id, Machine name, Thread ID, etc. I have described it in detailed uses in Multithreading debugging section.

Data Tip

Data tip is kind of an advanced tool tip message which is used to inspect the objects or variable during the debugging of the application. When debugger hits the breakpoint, if you mouse over to any of the objects or variables, you can see their current values. Even you can get the details of some complex object like `dataset`, `datatable`, etc. There is a "+" sign associated with the dataTip which is used to expand its child objects or variables.



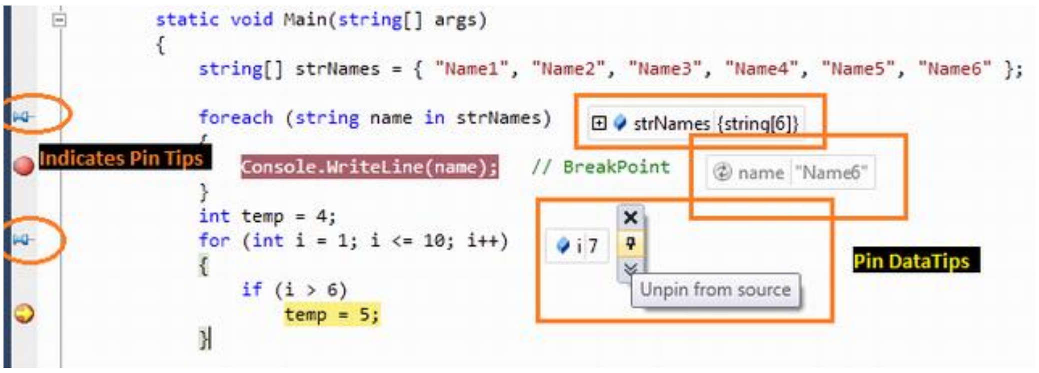
```
string[] strNames = { "Name1", "Name2", "Name3", "Name4", "Name5", "Name6" };

foreach (string name in strNames)
{
    Console.WriteLine(name);
}
int temp = 4;
for (int i = 1; i <= 10; i++)
{
    if (i > 6)
        temp = 5;
}
```

```
strNames {string[6]}
[0] "Name1"
[1] "Name2"
[2] "Name3"
[3] "Name4"
[4] "Name5"
[5] "Name6"
```

Figure: DataTips During Debugging

Pin Inspect Value During Debugging

While debugging in Visual Studio, we generally used mouse over on the object or variable to inspect the current value. This shows the current data items held by the inspected object. But this is for a limited time, as long as the mouse is pointed to that object those value will be available. But in Visual Studio there is a great feature to pin and unpin this inspected value. We can pin as many of any object and their sub object value also. Please have a look into the below picture:



```
static void Main(string[] args)
{
    string[] strNames = { "Name1", "Name2", "Name3", "Name4", "Name5", "Name6" };

    foreach (string name in strNames)
    {
        Console.WriteLine(name);   // BreakPoint
    }
    int temp = 4;
    for (int i = 1; i <= 10; i++)
    {
        if (i > 6)
            temp = 5;
    }
```

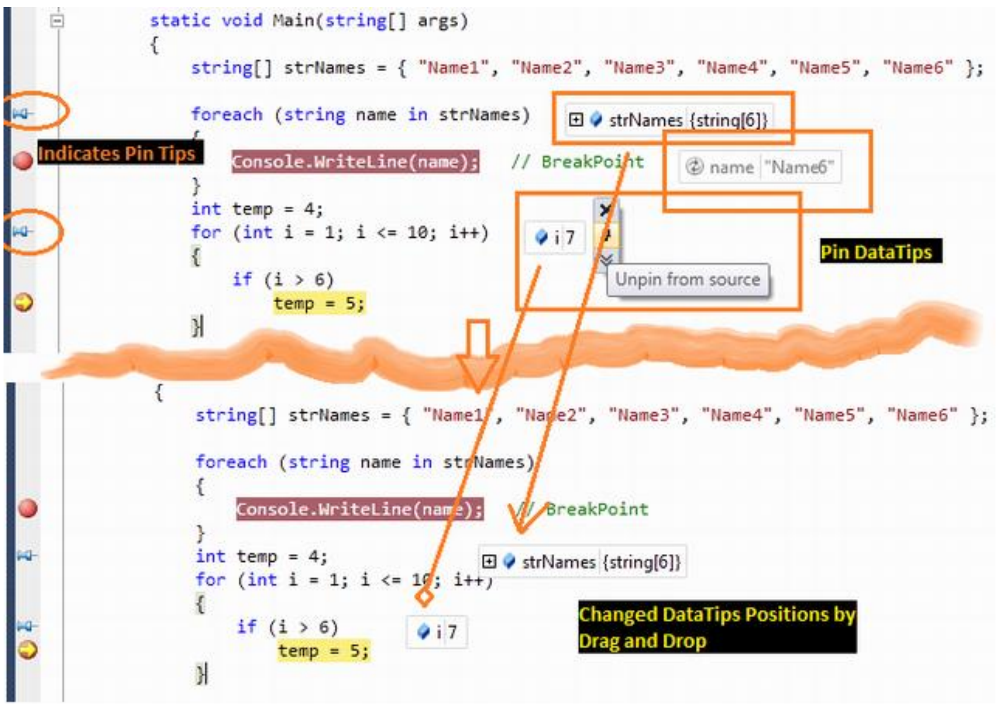
```
strNames {string[6]}
name "Name6"
i 7
```

Figure: Pin Inspect Value During Debugging

When you mouse over on the inspect object, you will get pin icon with each and every object's properties, variable. Click on that pin icon to make it pinned. Unless you manually close these pinned items, they will be visible in the IDE.

Drag-Drop Pin Data Tip

You can easily **Drag and Drop** the data tip inside the Visual Studio IDE. This is quite helpful when you need to see some object value list in the bottom section of code. You can easily drag those pinned data tips over there.



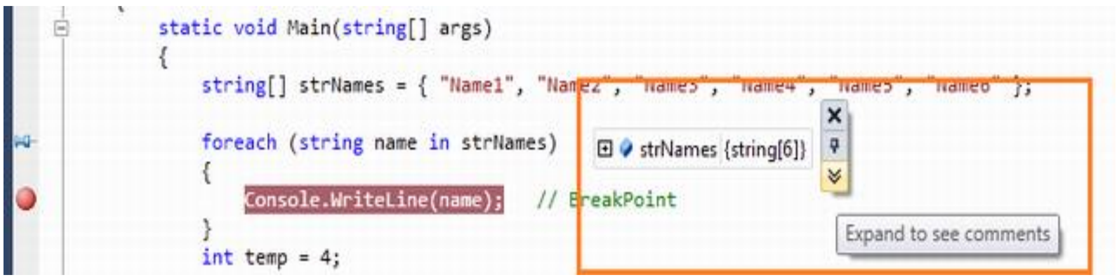
```
static void Main(string[] args)
{
    string[] strNames = { "Name1", "Name2", "Name3", "Name4", "Name5", "Name6" };

    foreach (string name in strNames)
    {
        Console.WriteLine(name);    // BreakPoint
    }
    int temp = 4;
    for (int i = 1; i <= 10; i++)
    {
        if (i > 6)
            temp = 5;
    }
```

```
{
    string[] strNames = { "Name1", "Name2", "Name3", "Name4", "Name5", "Name6" };

    foreach (string name in strNames)
    {
        Console.WriteLine(name);    // BreakPoint
    }
    int temp = 4;
    for (int i = 1; i <= 10; i++)
    {
        if (i > 6)
            temp = 5;
    }
```

Figure: Drag Drop Data Tips

Adding Comments

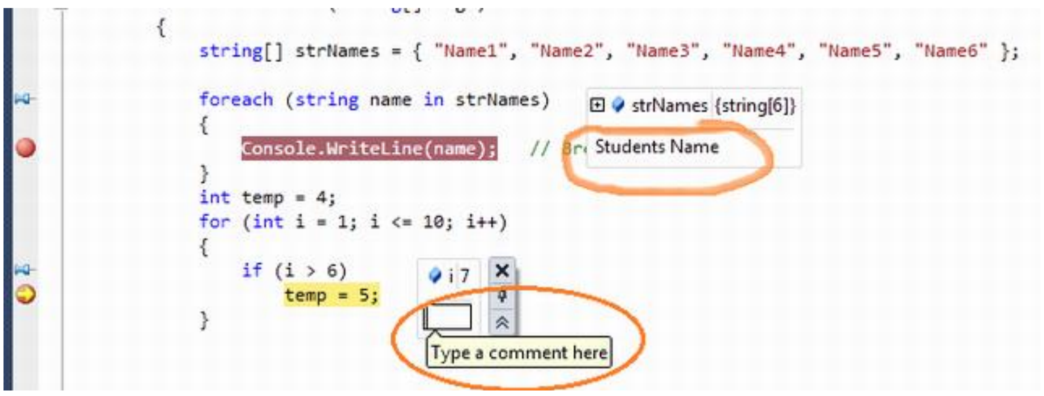
You can add comments on the pinned data tip. For providing comments, you need to click on "**Expand to see the comments**" button. This will brings up an additional textbox to add comments.



```
static void Main(string[] args)
{
    string[] strNames = { "Name1", "Name2", "Name3", "Name4", "Name5", "Name6" };

    foreach (string name in strNames)
    {
        Console.WriteLine(name);    // BreakPoint
    }
    int temp = 4;
```

Figure: Comments in DataTip

Below is some demonstration of Adding comments on pinned inspect value:

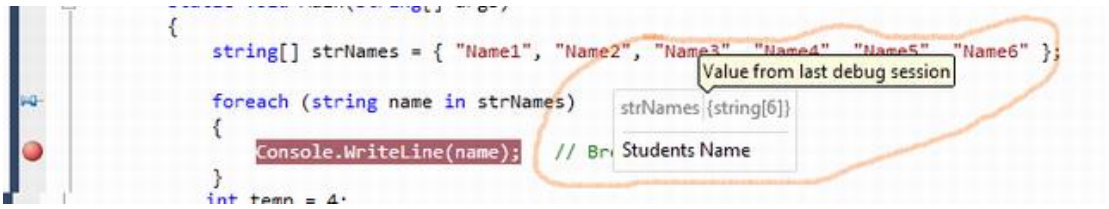


```
{
    string[] strNames = { "Name1", "Name2", "Name3", "Name4", "Name5", "Name6" };

    foreach (string name in strNames)
    {
        Console.WriteLine(name);    // Br
    }
    int temp = 4;
    for (int i = 1; i <= 10; i++)
    {
        if (i > 6)
            temp = 5;
    }
```

Figure: Adding Comments For Datatips

Last Session Debugging Value

This is another great feature of Visual Studio debugging. If you pinned some data tip during the debugging, the value of pinned item will remain stored in a session. In normal mode of coding, if you mouse over the pin icon, it will show the details of the last debugging session value as shown in the below picture:

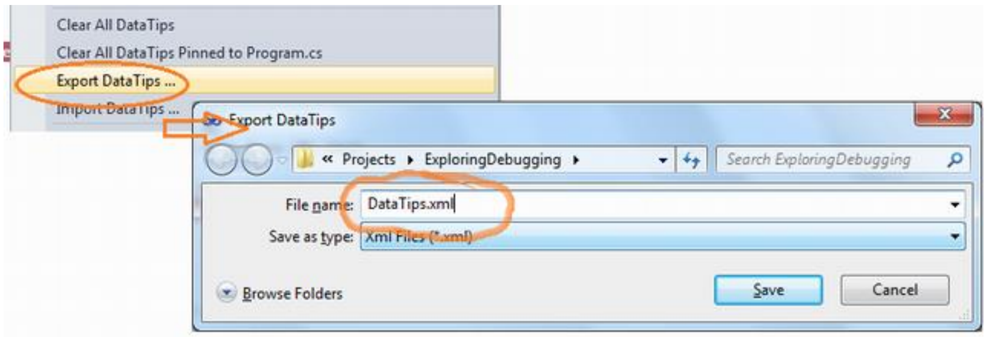


```
{
    string[] strNames = { "Name1", "Name2", "Name3", "Name4", "Name5", "Name6" };

    foreach (string name in strNames)
    {
        Console.WriteLine(name);    // Br
    }
    int temp = 4;
```

Figure: Last Session Debug Value

Import Export Data Tips

This feature is quite similar to the import/Export breakpoints. Like Breakpoints, you can import and export pinned data tips values in an XML file.

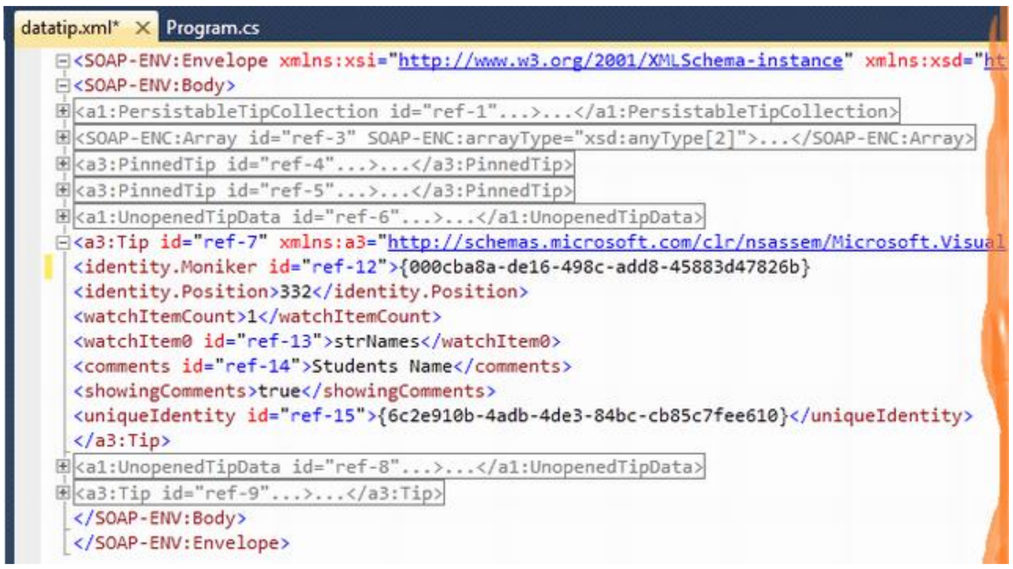


Clear All DataTips
Clear All DataTips Pinned to Program.cs
Export DataTips ...
Import DataTips ...
Export DataTips
« Projects ▸ ExploringDebugging ▸
Search ExploringDebugging
Browse Folders
Save
Cancel

File name: DataTips.xml
Save as type: Xml Files (*.xml)

Figure: Export Data Tips

These saved data tips can be imported during any point of time for further debugging. The XML file looks like below:



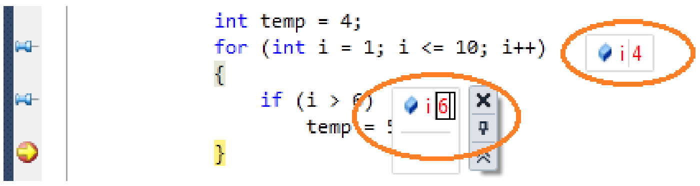
```
<SOAP-ENV:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="ht
<SOAP-ENV:Body>
<a1:PersistableTipCollection id="ref-1"...>...</a1:PersistableTipCollection>
<SOAP-ENC:Array id="ref-3" SOAP-ENC:arrayType="xsd:anyType[2]">...</SOAP-ENC:Array>
<a3:PinnedTip id="ref-4"...>...</a3:PinnedTip>
<a3:PinnedTip id="ref-5"...>...</a3:PinnedTip>
<a1:UnopenedTipData id="ref-6"...>...</a1:UnopenedTipData>
<a3:Tip id="ref-7" xmlns:a3="http://schemas.microsoft.com/clr/nsassem/Microsoft.Visual
<identity.Moniker id="ref-12">{000cba8a-de16-498c-add8-45883d47826b}
<identity.Position>332</identity.Position>
<watchItemCount>1</watchItemCount>
<watchItem0 id="ref-13">strNames</watchItem0>
<comments id="ref-14">Students Name</comments>
<showingComments>true</showingComments>
<uniqueIdentity id="ref-15">{6c2e910b-4adb-4de3-84bc-cb85c7fee610}</uniqueIdentity>
</a3:Tip>
<a1:UnopenedTipData id="ref-8"...>...</a1:UnopenedTipData>
<a3:Tip id="ref-9"...>...</a3:Tip>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Figure: XML Content of DataTips

You can further explore the XML file if you want to know more details about it. :)

Change Value Using Data Tips

DataTips is also used to changed the value while debugging. This means it can be used like a watch window. From the list of Pinned objects, you can change their value to see the impact on the program.



```
int temp = 4;
for (int i = 1; i <= 10; i++)
{
    if (i > 6)
        temp = 5
}
```

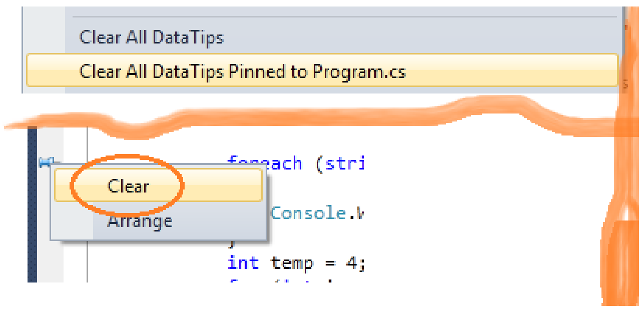
Figure: Change Value Within Data Tip

Clear Data Tips

You can clear the data Tips by selecting **"Clear Data Tips"** from Debug menu. There are two options:

1. Clear All Data Tips
2. Clear All Data Tips Pinned To [*File Name.Cs*]

So if you want to clear all the Data Tips from all over your project / solution, just select the first option. But if you want to delete the pinned data tips from a particular file, then you need to open that particular file, there you have the second option to select. You can even delete particular pinned data tips by just right clicking on it and clicking **"Clear"**.



```
foreach (stri
    Console.W
}
int temp = 4;
```

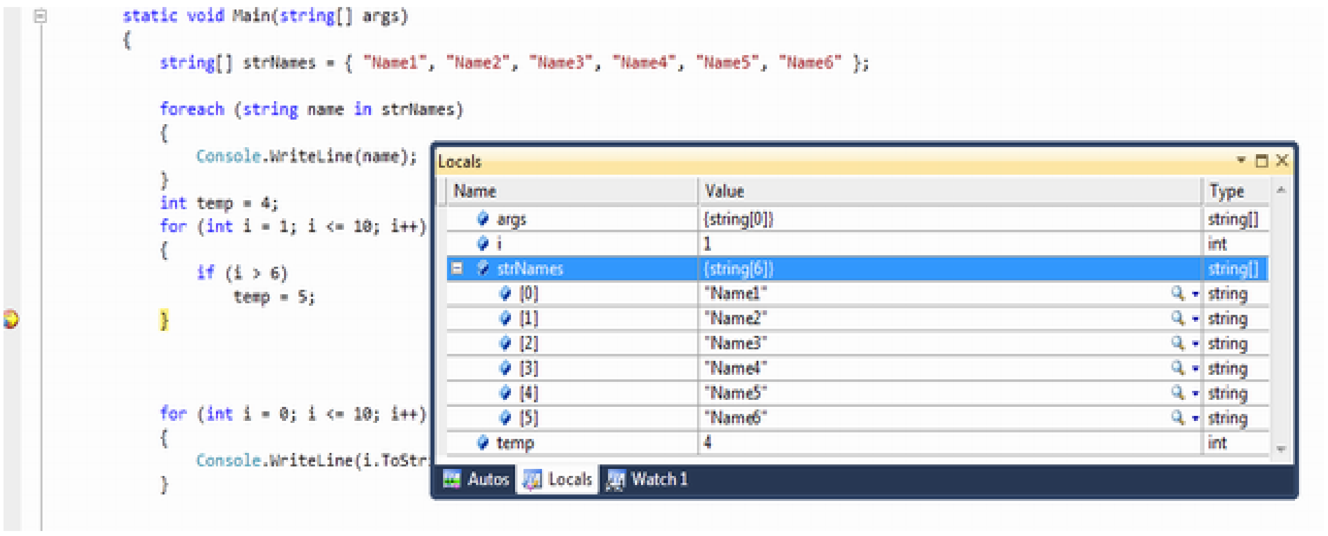
Figure: Clear Data Tips

Watch Windows

You can say it is an investigation window. After breakpoint has been hit, the next thing you want to do is to investigate the current object and variables values. When you mouse hover on the variable, it shows the information as a data tip which you can expand, pin, import which I have already explained. There are various types of watch windows like Autos, Local, etc. Let's have a look into their details.

Locals

It automatically displays the list of variables within the scope of current methods. If your debugger currently hits a particular breakpoint and if you open the "Autos" window, it will show you the current scope object variable along with the value.



```
static void Main(string[] args)
{
    string[] strNames = { "Name1", "Name2", "Name3", "Name4", "Name5", "Name6" };

    foreach (string name in strNames)
    {
        Console.WriteLine(name);
    }
    int temp = 4;
    for (int i = 1; i <= 10; i++)
    {
        if (i > 6)
            temp = 5;
    }


    for (int i = 0; i <= 10; i++)
    {
        Console.WriteLine(i.ToStr
    }
```

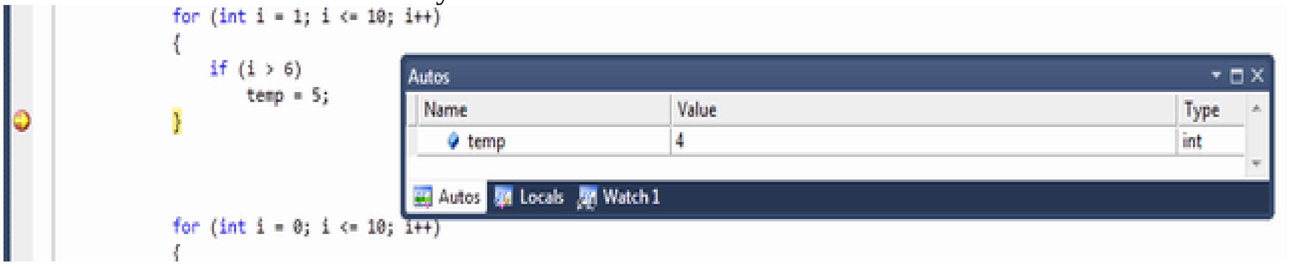
Locals

Name	Value	Type
args	{string[0]}	string[]
i	1	int
strNames	{string[6]}	string[]
[0]	"Name1"	string
[1]	"Name2"	string
[2]	"Name3"	string
[3]	"Name4"	string
[4]	"Name5"	string
[5]	"Name6"	string
temp	4	int

Figure: Local Variables

Autos

The VS debugger automatically detects these variables during the debugging. Visual Studio determines which objects or variables are important for the current code statement and based on that, it lists down the "**Autos**" variable. Shortcut key for the Autos Variable is **"Ctrl + D + A"**.



```
for (int i = 1; i <= 10; i++)
{
    if (i > 6)
        temp = 5;
}


for (int i = 0; i <= 10; i++)
{
```

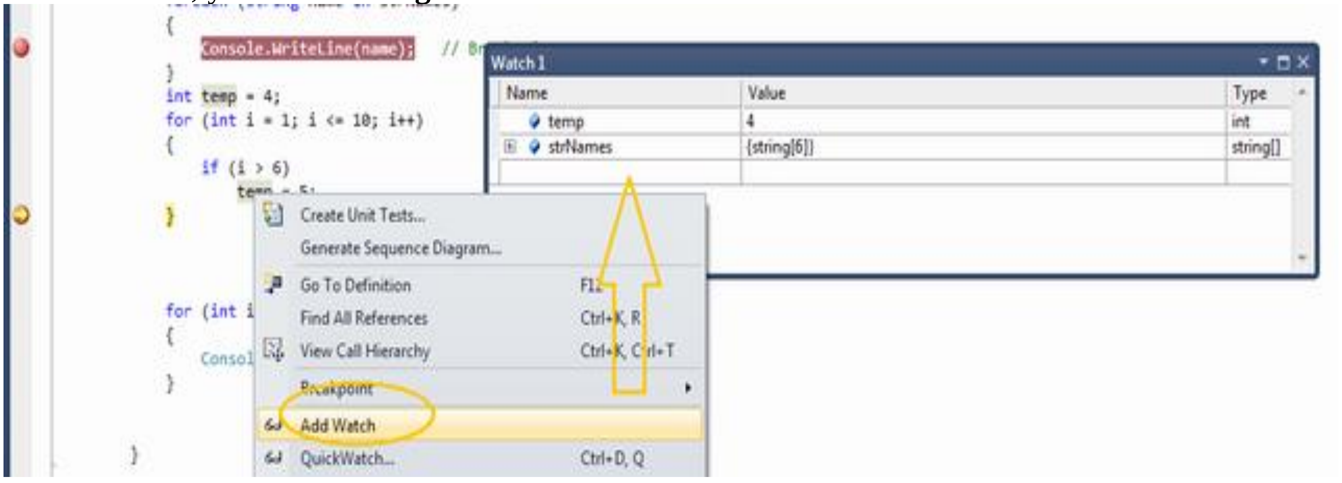
Autos

Name	Value	Type
temp	4	int

Figure: Autos - Ctrl+D, A

Watch

Watch windows are used for adding variables as per requirement. It displays variables that you have added. You can add as many variables as you want into the watch window. To add variables in the watch window, you need to **"Right Click"** on variable and then select **"Add To Watch"**.



```
{
    Console.WriteLine(name);    // Br
}
int temp = 4;
for (int i = 1; i <= 10; i++)
{
    if (i > 6)
        temp = 5;
}


for (int i
{
    Consol
}
```

Watch 1

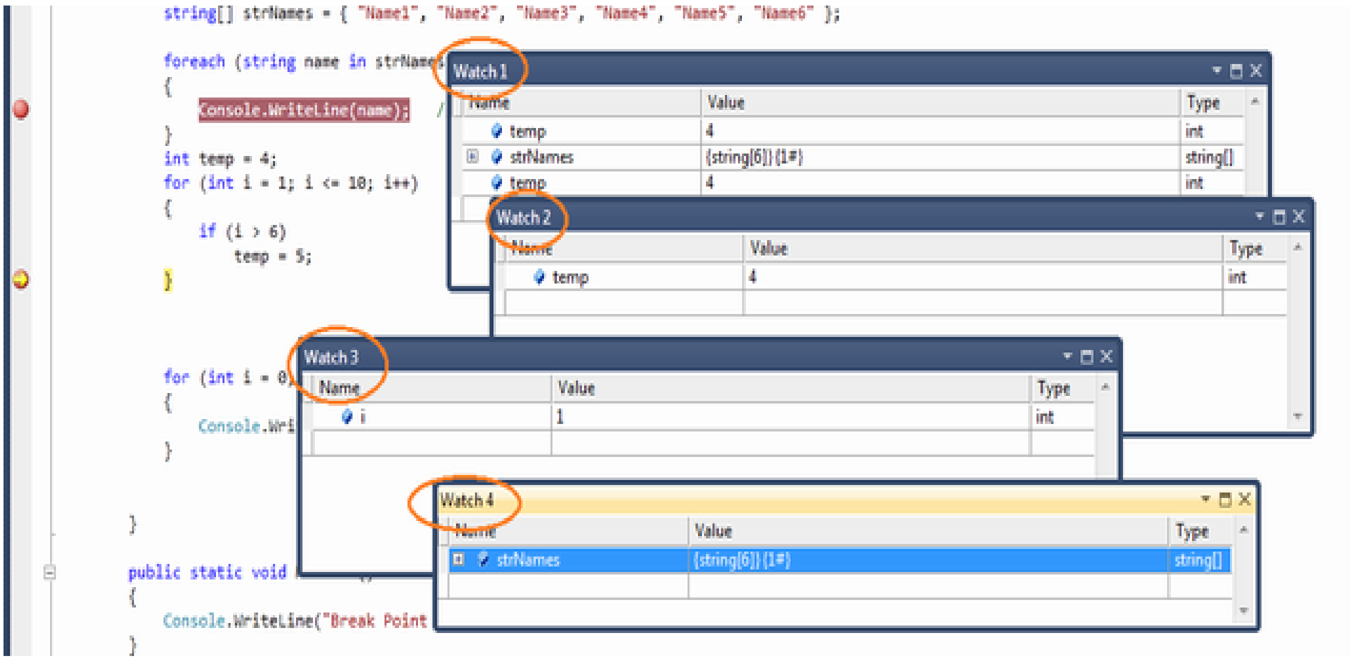
Name	Value	Type
temp	4	int
strNames	{string[6]}	string[]

Create Unit Tests...
Generate Sequence Diagram...
Go To Definition F12
Find All References Ctrl+K, R
View Call Hierarchy Ctrl+K, Ctrl+T
Breakpoint
Add Watch
QuickWatch... Ctrl+D, Q

Figure: Autos - Ctrl+D, W

You can also use **Drag and Drop** to add variables in watch windows. If you want to delete any variable from watch window, just right click on that variable and select **"Delete Watch"**. From the debug window, you can also edit the variable value at run time.

There are 4 different watch windows available which you can use parallelly.



```
string[] strNames = { "Name1", "Name2", "Name3", "Name4", "Name5", "Name6" };

foreach (string name in strName
{
    Console.WriteLine(name);
}
int temp = 4;
for (int i = 1; i <= 10; i++)
{
    if (i > 6)
        temp = 5;
}


for (int i = 0
{
    Console.Wri
}

}

public static void
{
Console.WriteLine("Break Point
}
```

Watch 1

Name	Value	Type
temp	4	int
strNames	{string[6]}{1#}	string[]
temp	4	int

Watch 2

Name	Value	Type
temp	4	int

Watch 3

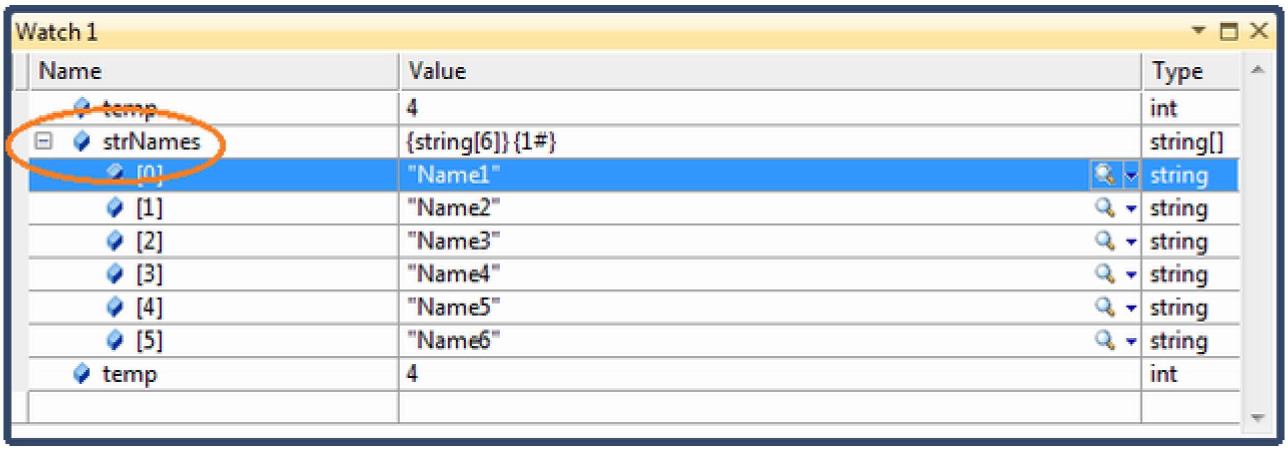
Name	Value	Type
i	1	int

Watch 4

Name	Value	Type
strNames	{string[6]}{1#}	string[]

Figure: Multiple Watch Window

If any of the row variables of the above window holds the object instance, you can have a **"+"** symbol with the variable to explore the properties and member of that object variable.

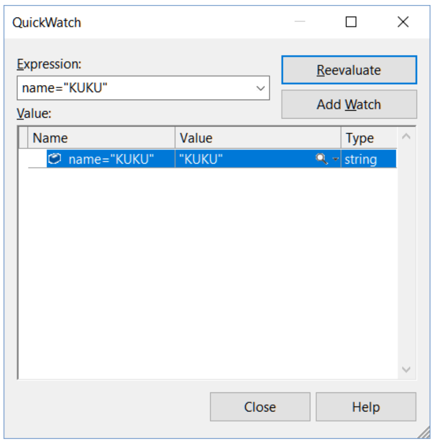


Watch 1

Name	Value	Type
temp	4	int
strNames	{string[6]}{1#}	string[]
[0]	"Name1"	string
[1]	"Name2"	string
[2]	"Name3"	string
[3]	"Name4"	string
[4]	"Name5"	string
[5]	"Name6"	string
temp	4	int

Figure: Expanding Watched Variable

Sometimes you don't need to add a watch but still wish to observe a variable or evaluate an expression. The QuickWatch dialog allows you to do that. To open the QuickWatch dialog you right click on a variable and then select QuickWatch from the shortcut menu. This is how a QuickWatch dialog looks:



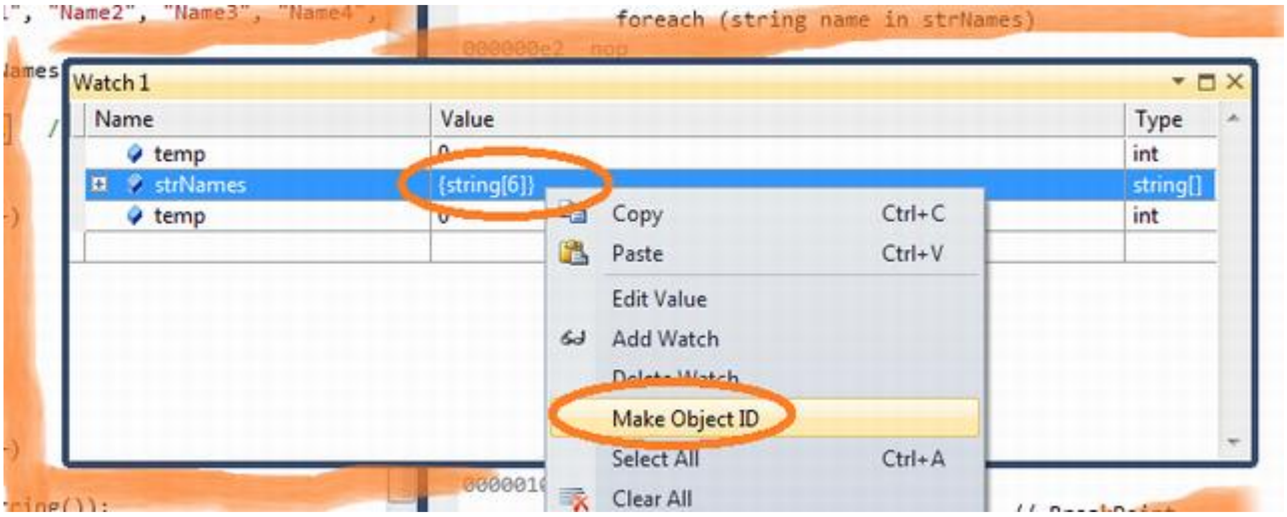
Expression: name="KUKU"

Name	Value	Type
name="KUKU"	"KUKU"	string

Figure: QuickWatch

Creating Object ID

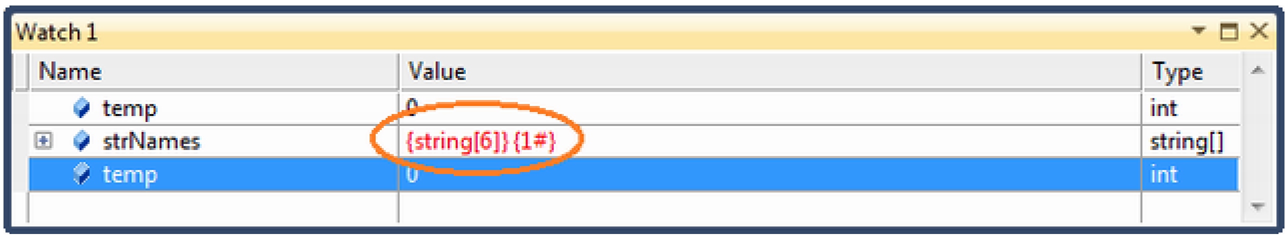
Visual Studio Debugger has another great functionality where you can create an object ID for any particular instance of object. This is very much helpful when you want to monitor any object at any point of time even if it goes out of scope. To create Object Id, from watch window you need to right click on a particular object variable and then need to click on **"Make Object ID"**.



Name	Value	Type
temp	0	int
strNames	{string[6]}	string[]
temp	0	int

Figure: Creating Object ID

After adding Object Id with a particular object variable, Visual Studio adds a numeric number with "#" with that object to indicate that one Object ID has been created.



Name	Value	Type
temp	0	int
strNames	{string[6]} {1#}	string[]
temp	0	int

Figure: Object ID Added

You can use the object ID to set a breakpoint on a method of a specific instance. For example, suppose you have an object that is an instance of class CMyType, and the instance has object ID 5#. Class

CMyType includes a method aMethod. You can set a function breakpoint on method aMethod of instance 5# as follows:

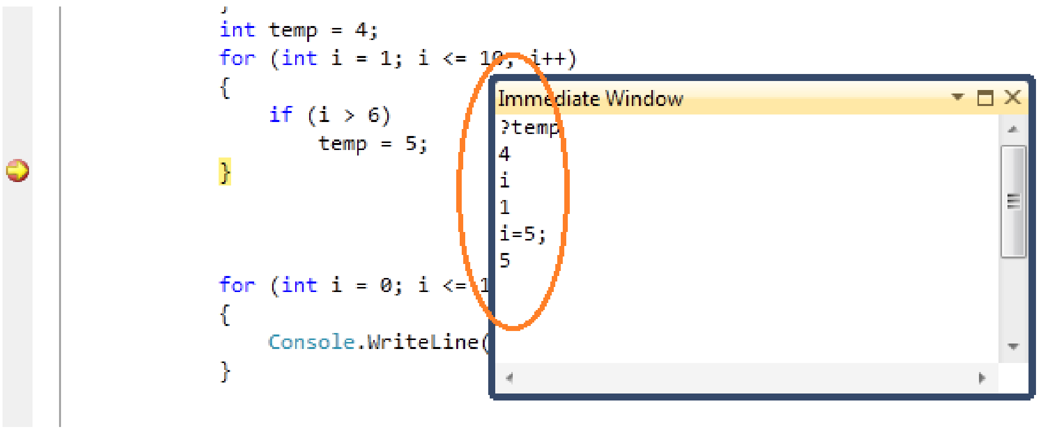
((CMyType) 5#).aMethod

You can also use the object ID in a breakpoint condition. The following example shows how you can test the object ID in a condition.

this == 5#

Immediate Window

Immediate window is very common and a favorite with all developers. It's very helpful in debug mode of the application if you want to change the variable values or execute some statement without impacting your current debugging steps. You can open the Immediate window from menu **Debug** > **Window** > **Immediate Window** { **Ctrl + D, I** / **Alt + Ctrl - I** }. Immediate window has a set of commands which can be executed any time during debugging. It also supports Intellisense. During Debug mode, you can execute any command or execute any code statement from here.



```
int temp = 4;
for (int i = 1; i <= 10; i++)
{
    if (i > 6)
        temp = 5;
}


for (int i = 0; i <= 1
{
    Console.WriteLine(
}
```

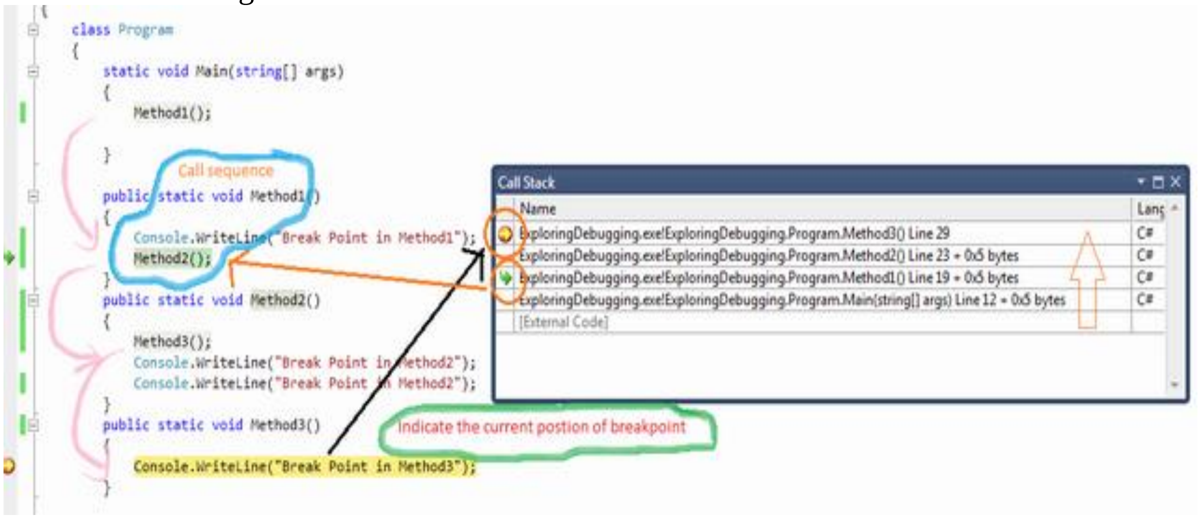
```
?temp
4
i
1
i=5;
5
```

Figure: Basic Immediate Window

These are very much common features for all the developers, so I am not going into details of each and every command of Immediate window.

Call Stack

These features also improve the productivity during debugging. If you have multiple method calling or nested calling all over your application and during debugging, you want to check from where this method has invoked, **"Call Stack"** comes into the picture. The Call Stack Window shows that current method call nesting.

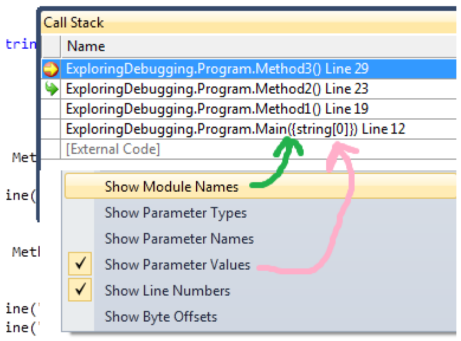


```
{
    class Program
    {
        static void Main(string[] args)
        {
            Method1();

        }
        public static void Method1()
        {
            Console.WriteLine("Break Point in Method1");
            Method2();
        }
        public static void Method2()
        {
            Method3();
            Console.WriteLine("Break Point in Method2");
            Console.WriteLine("Break Point in Method2");
        }
        public static void Method3()
        {
            Console.WriteLine("Break Point in Method3");
        }
```

Name	Lang
ExploringDebugging.exe!ExploringDebugging.Program.Method3() Line 29	C#
ExploringDebugging.exe!ExploringDebugging.Program.Method2() Line 23 + 0x5 bytes	C#
ExploringDebugging.exe!ExploringDebugging.Program.Method1() Line 19 + 0x5 bytes	C#
ExploringDebugging.exe!ExploringDebugging.Program.Main(string[] args) Line 12 + 0x5 bytes	C#
[External Code]	

Figure: Call Stack

In Call Stack window if you clicked on any of the rows, it will point you to the actual code of line of Visual Studio Code Editor. You can also customize the call stack row view by selecting different types of columns. To customize, Right Click on the **"Call Stack"** window, and from the context menu, you can select or deselect the option.

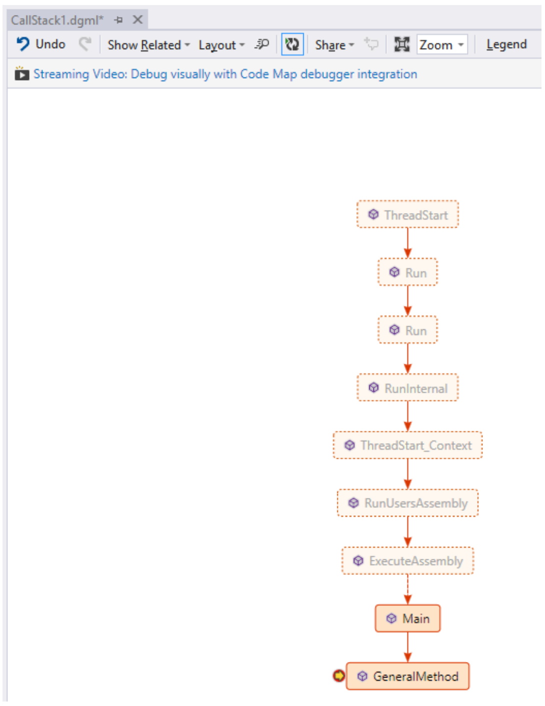


Name
ExploringDebugging.Program.Method3() Line 29
ExploringDebugging.Program.Method2() Line 23
ExploringDebugging.Program.Method1() Line 19
ExploringDebugging.Program.Main({string[0]}) Line 12
[External Code]

Figure: Call Stack Customization

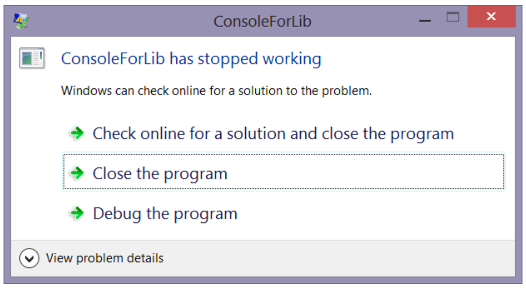
Call stack is very much important when you have multiple methods call all across the application and one particular method throwing an exception on some particular case. At that time, you can use call stack to see from where this method is getting invoked, based on that you can fix the defect.

It’s usefully also to view call stack if graphic form. For this purpose, use the option “Show Call Stack on the Code Map”:



Debugging command line running programs (Attaching to a process)

To debug a program running from command line (cmd.exe) you need to use a special method – "`Debugger.Break()`". Place this method into a point where you want to begin debugging.
After that run a program from command line of cmd.exe console.

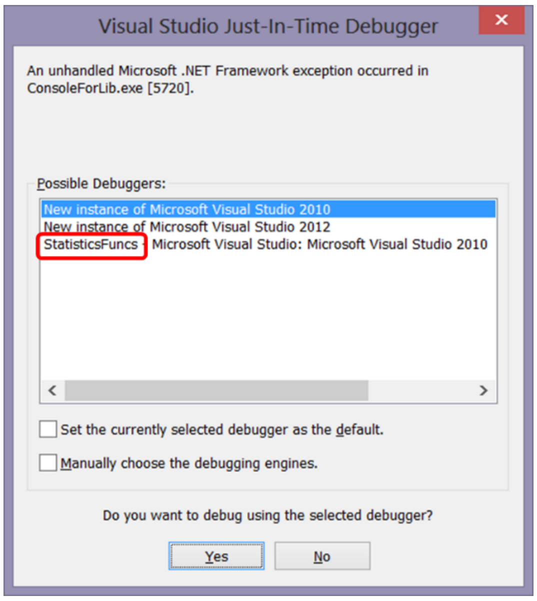


ConsoleForLib

ConsoleForLib has stopped working

Windows can check online for a solution to the problem.

Select "Debug the program":



Visual Studio Just-In-Time Debugger

An unhandled Microsoft .NET Framework exception occurred in ConsoleForLib.exe [5720].

Possible Debuggers:

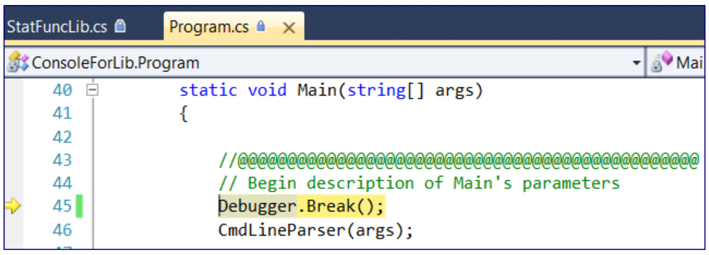
New instance of Microsoft Visual Studio 2010
New instance of Microsoft Visual Studio 2012
StatisticsFuncs - Microsoft Visual Studio: Microsoft Visual Studio 2010

Set the currently selected debugger as the default.

Manually choose the debugging engines.

Do you want to debug using the selected debugger?

Chose a wished option (usually it's worth to use already running instant of VS – in this case it is "StatisticsFuncs"). Finally, you have to receive something like this:



```
        static void Main(string[] args)
        {

            //@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
            // Begin description of Main's parameters
            Debugger.Break();
            CmdLineParser(args);
```

Appendix

Code listing:

```
class Program
    {
        static void Main(string[] args)
        {
            string[] strNames = { "Name1", "Name2", "Name3", "Name4", "Name5", "Name6" };

            foreach (string name in strNames)
            {
                Console.WriteLine(name);    // BreakPoint
            }
            int temp = 4;
            for (int i = 1; i <= 10; i++)
            {
                if (i > 6)
                    temp = 5;

                Method1();
                Method2();
                Method3();
            }
        }
        public static void Method1()
        {
            Console.WriteLine("Break Point in Method1");    // BreakPoint
        }

        public static void Method2()
        {
            Console.WriteLine("Break Point in Method2");  // BreakPoint
        }

        public static void Method3()
        {
            Console.WriteLine("Break Point in Method3");  // Breakpoint
        }
    }
```