Michael's Coding Spot

SEARCH

HOME C# WPF PROGRAMMING VSIX ABOUT RSS

7 Debugging Techniques you should know in C# .NET

Jan 9, 2018 | C# | 10

SUBSCRIBE

Join the mailing list and get updates on new articles

EMAIL

SUBSCRIBE

Hi and welcome to my blog! I'm a software developer and tech

I think I spend more time debugging code than writing code, designing software architecture, trying to reproduce bugs and even going to meetings! Debugging is the biggest time consumer we have as developers.

As time goes by, I keep learning new tricks that make my debugging more effective. I've gathered in this post 7 debugging techniques that I learned relatively late into my career and I consider advanced (though sometimes something advanced for some is trivial for others). Enjoy and I hope you learn something new!

1. Evaluate an expression with or without side effects

leader, working with and writing about .NET, WPF, C++, office life, developer philosophy and whatever comes to mind.

More about me →

On my day job, I work on **OzCode**, a VS extension that makes debugging C# easier.

See for yourself →

CATEGORIES

Azure (1)

C# (35)

Personal life (3)

Programming (17)

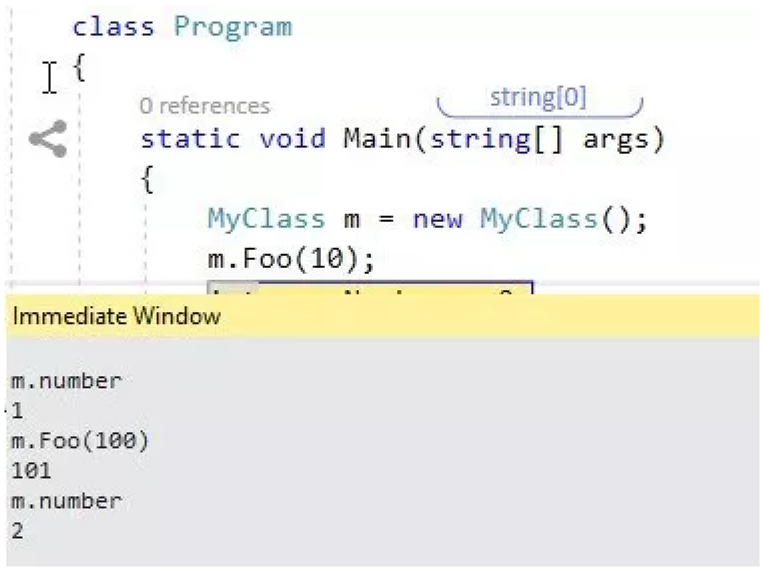
Sometimes, during debugging, we want to evaluate an expression that can change application state when executed. For example, a property's getter that modifies a static variable. This can corrupt our debugging session. There's a way however, to evaluate expressions without side effects.

Consider the following class:

```
class MyClass
{
    public int number = 0;

    public int Foo(int a)
    {
        number++;
        return a + 1;
    }
}
```

If I evaluate **Foo** in the Immediate Window, **number** will increase and my application state will be changed:



```
class Program
{
    0 references
    static void Main(string[] args)
    {
        MyClass m = new MyClass();
        m.Foo(10);
```

```
m.number
1
m.Foo(100)
101
m.number
2
```

Visual Studio (1)

VSIX (12)

RECENT POSTS

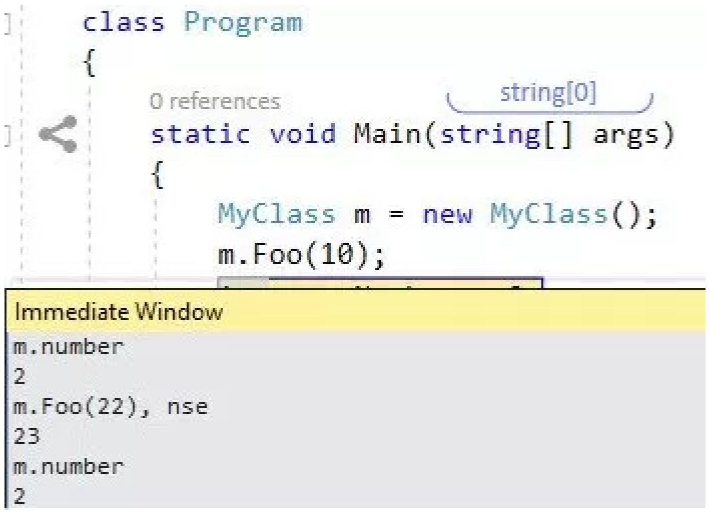
Use Performance Counters in .NET to measure Memory, CPU, and Everything – Full Guide
Sep 9, 2019 | C#

Pipeline Implementations in C#: TPL Dataflow Async steps and Disruptor-net
Aug 26, 2019 | C#

Logging in C# .NET Modern-day Practices: The Complete Guide
Aug 13, 2019 | C#

We can see here that **m.number** increased from 1 to 2.

By adding "**, nse**" (No Side Effects) to the expression will evaluate without changing the application state.



```
class Program
{
    0 references                    string[0]
    static void Main(string[] args)
    {
        MyClass m = new MyClass();
        m.Foo(10);
```

Immediate Window

```
m.number
2
m.Foo(22), nse
23
m.number
2
```

Now we can see that the evaluation returned correctly, and **m.number** stayed the same.

Adding **, nse** runs the code in a sandbox environment of sorts and application state will not be changed. Running with *No Side Effects* doesn't always work. For example, when the evaluated method calls Native code.

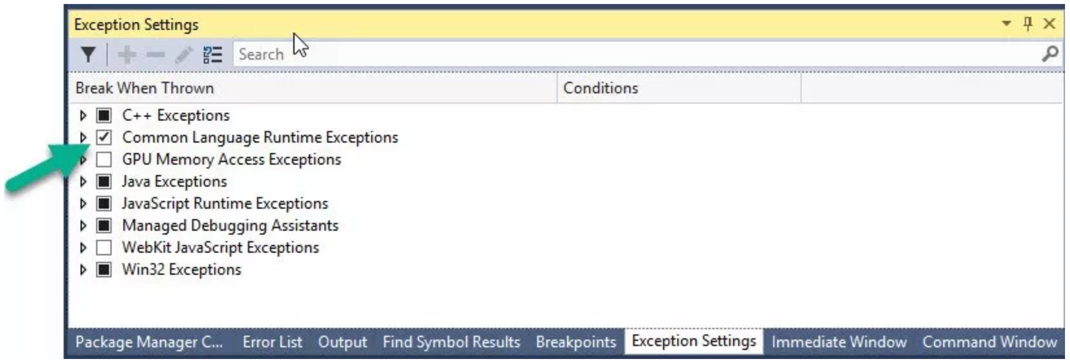
2. Breaking on Exceptions with or without source code

.NET has 2 types of Exceptions: **Handled Exceptions** that you swallow in a **catch** clause and **Unhandled Exceptions**. Unhandled Exceptions can crash your program or Thread.

Debug & Catch Exceptions in Visual Studio: The Complete Guide
Aug 5, 2019 | C#

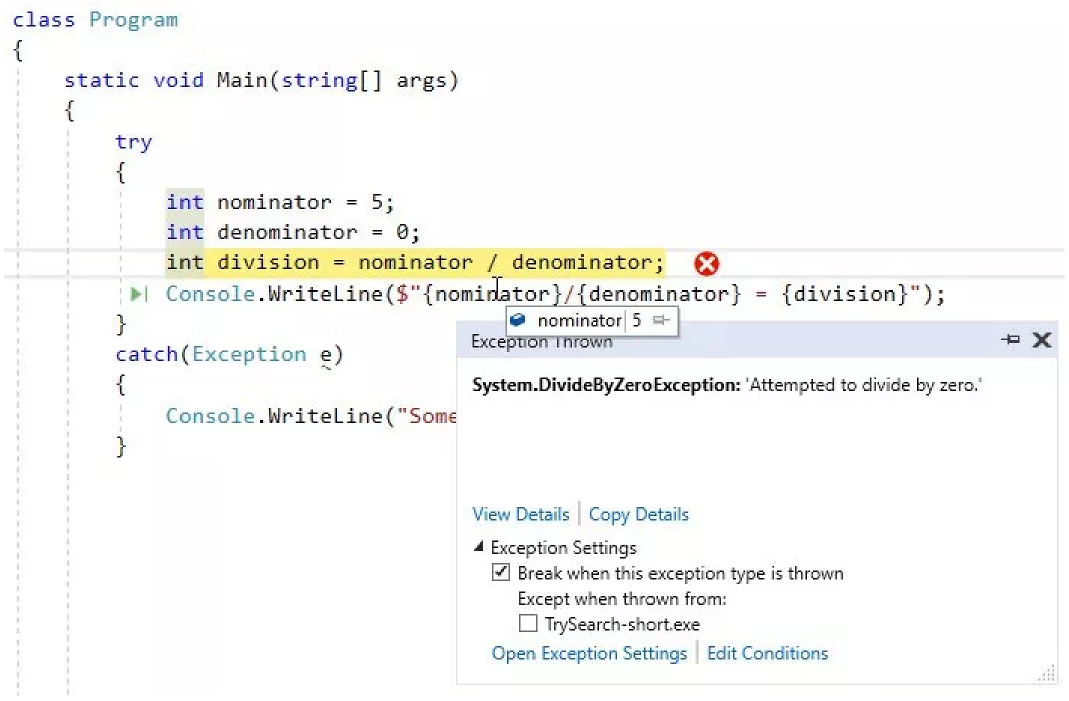
Pipeline Pattern in C# (part 2) with TPL Dataflow
Jul 22, 2019 | C#

While debugging, Visual Studio will always break on Unhandled Exceptions. As for Handled Exceptions, you can control whether to break on them in the Exceptions Settings Tool Window. In Menu, go to Debug -> Windows -> Exception Settings:



- ■ C++ Exceptions
- ✓ Common Language Runtime Exceptions
- ☐ GPU Memory Access Exceptions
- ■ Java Exceptions
- ■ JavaScript Runtime Exceptions
- ■ Managed Debugging Assistants
- ☐ WebKit JavaScript Exceptions
- ■ Win32 Exceptions

Toggling CLR exceptions controls whether to break on managed exceptions. You can even select specific exception types to ignore.



```
class Program
{
    static void Main(string[] args)
    {
        try
        {
            int nominator = 5;
            int denominator = 0;
            int division = nominator / denominator;
            Console.WriteLine($"{nominator}/{denominator} = {division}");
        }
        catch(Exception e)
        {
            Console.WriteLine("Some
        }
```

nominator | 5
Exception Thrown

System.DivideByZeroException: 'Attempted to divide by zero.'

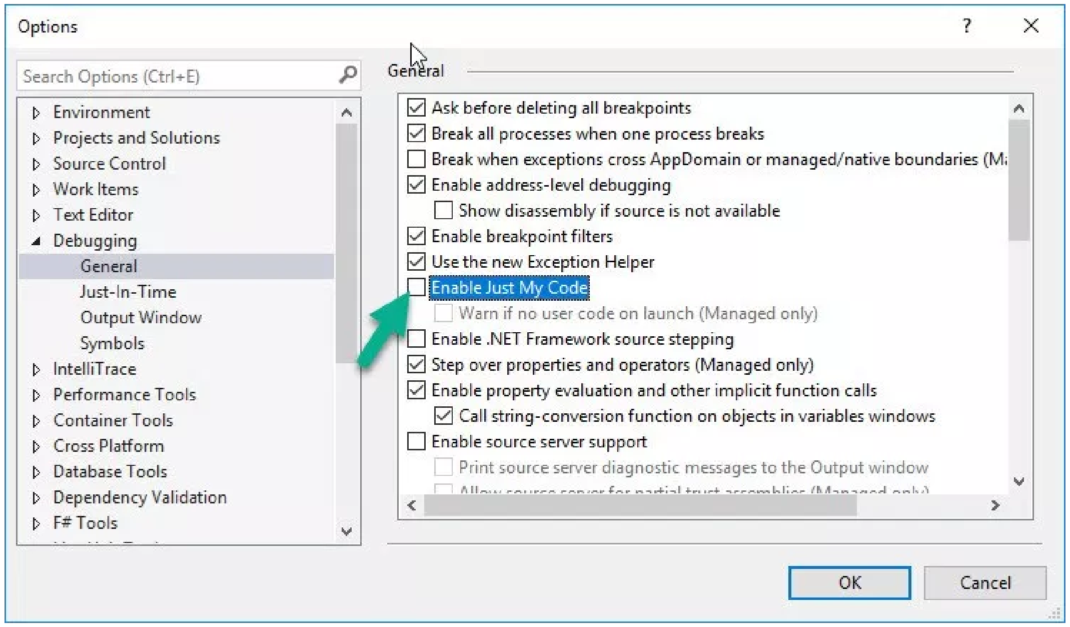
View Details | Copy Details
Exception Settings
☑ Break when this exception type is thrown
Except when thrown from:
☐ TrySearch-short.exe
Open Exception Settings | Edit Conditions

This is a very useful debugging feature to catch problems in the application. Whenever something doesn't work for an unexplained reason, there's a good chance an exception was thrown.

Enable and Disable Just My Code

By default, VS will break only on Exceptions thrown in your code. However, there are often problems that happen due to exceptions in one of your references. You might have used the API wrong or there's a bug in some NuGet package you are using.

To break on exceptions from all code, go to **Tools -> Options** and in **Debugging**, uncheck "Enable Just My Code".



- Environment
- Projects and Solutions
- Source Control
- Work Items
- Text Editor
- Debugging
  - General
  - Just-In-Time
  - Output Window
  - Symbols
- IntelliTrace
- Performance Tools
- Container Tools
- Cross Platform
- Database Tools
- Dependency Validation
- F# Tools

General

- ✓ Ask before deleting all breakpoints
- ✓ Break all processes when one process breaks
- ☐ Break when exceptions cross AppDomain or managed/native boundaries (M
- ✓ Enable address-level debugging
  - ☐ Show disassembly if source is not available
- ✓ Enable breakpoint filters
- ✓ Use the new Exception Helper
- ☐ Enable Just My Code
  - ☐ Warn if no user code on launch (Managed only)
- ☐ Enable .NET Framework source stepping
- ✓ Step over properties and operators (Managed only)
- ✓ Enable property evaluation and other implicit function calls
  - ✓ Call string-conversion function on objects in variables windows
- ☐ Enable source server support
  - ☐ Print source server diagnostic messages to the Output window
  - ☐ Allow source server for partial trust assemblies (Managed only)

Now you'll break on all exceptions, including in the methods of your references. You probably won't see the code since you don't have the symbols of that library, but you will see the Call Stack and Exception type.

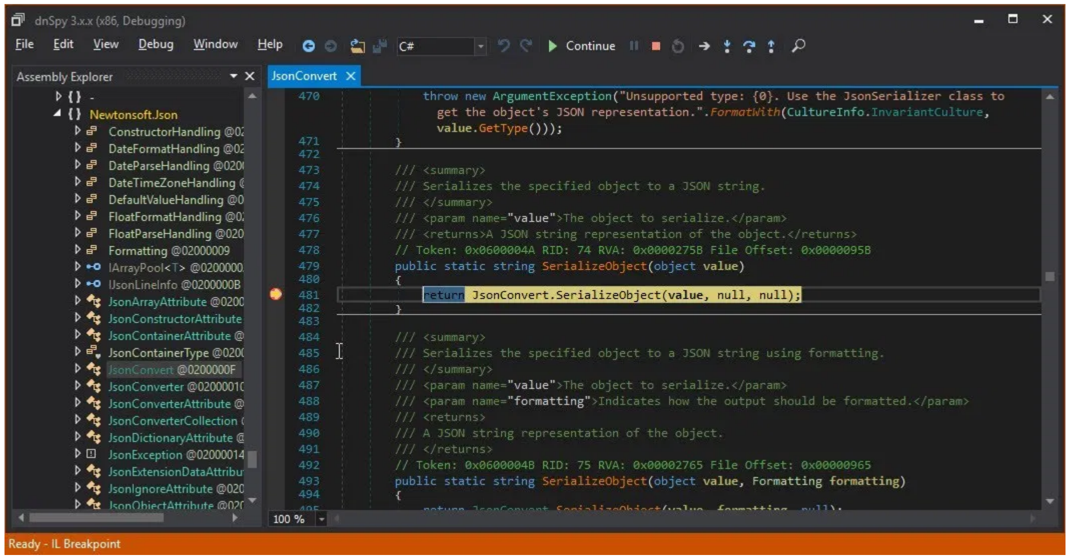
There's a way to debug inside the referenced library with a neat free tool called **DnSpy**.

You'll probably want to keep **Enable Just My Code** enabled most of the time. Otherwise, Visual Studio will keep loading symbols (which is time-consuming) and you might start breaking on an endless stream of exceptions that you don't care about.

3. Debug your references with DnSpy

Sometimes you'll encounter problems in your references that you'll need to investigate. With DnSpy you can debug any assembly, even without its source code.

DnSpy will decompile and reveal source code of any .dll or .exe file. You can then place breakpoints in the decompiled source code and debug. Here's me debugging **Newtonsoft.Json**'s "SerializeObject" method:



```csharp
        throw new ArgumentException("Unsupported type: {0}. Use the JsonSerializer class to
          get the object's JSON representation.".FormatWith(CultureInfo.InvariantCulture,
          value.GetType()));
    }

    /// <summary>
    /// Serializes the specified object to a JSON string.
    /// </summary>
    /// <param name="value">The object to serialize.</param>
    /// <returns>A JSON string representation of the object.</returns>
    // Token: 0x0600004A RID: 74 RVA: 0x0000275B File Offset: 0x0000095B
    public static string SerializeObject(object value)
    {
        return JsonConvert.SerializeObject(value, null, null);
    }

    /// <summary>
    /// Serializes the specified object to a JSON string using formatting.
    /// </summary>
    /// <param name="value">The object to serialize.</param>
    /// <param name="formatting">Indicates how the output should be formatted.</param>
    /// <returns>
    /// A JSON string representation of the object.
    /// </returns>
    // Token: 0x0600004B RID: 75 RVA: 0x00002765 File Offset: 0x00000965
    public static string SerializeObject(object value, Formatting formatting)
    {
```

Using DnSpy is pretty straightforward. First, copy the path of the referenced library you want to debug and open it in DnSpy. You will be able to see the decompiled source code. Then, search the method you want to debug and place a breakpoint. Finally, run your program from DnSpy with F5 (you can also Attach to Process). As the breakpoint hits, DnSpy will break and you can debug just like in VS with Step Over and Step Into (F10 and F11).

Another feature is the ability to **edit** the assembly. So you'll be able to modify your referenced class library.

> Much like with DnSpy, you can decompile and debug references directly in Visual Studio. There are two convenient ways to do this. One is to use dotPeek as a symbol server (dotPeek is a free tool by JetBrains). The second way, and I think the most convenient one, is to use ReSharper.

4. Edit and Continue

Edit and Continue has been around for awhile, and if you know it just skip ahead. Since it's so useful, I wasn't able to resist including it.

Edit and Continue allows you to make changes to the code while in break mode. This means that if you found a bug, you don't have to stop debugging and run the program again. You can change the code while in break mode and VS knows how to deal with that.

There are a lot of limitations to this. For example, you can't change inside a lambda function.

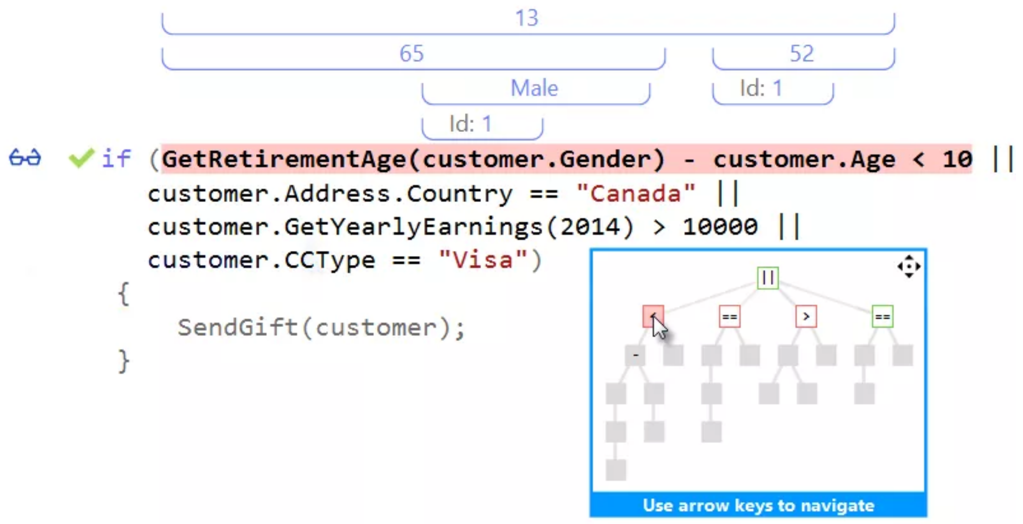
Edit and Continue in XAML

A very useful and relatively recent feature is XAML Edit and Continue. Since VS 2015, Update 2, you can edit XAML code while debugging and see the changes immediately. There's even no need to be in break mode to edit the XAML.

There are some issues and bugs with this from my experience. Sometimes it doesn't work, or doesn't work from first try. Specifically, I had issued with **Styles** and **Triggers**. Nevertheless, a very useful feature.

5. Use OzCode

OzCode is a unique VS extensions to help debugging. There are a lot of excellent features that will help debugging. For example, Head-Up Display;



```
if (GetRetirementAge(customer.Gender) - customer.Age < 10 ||
    customer.Address.Country == "Canada" ||
    customer.GetYearlyEarnings(2014) > 10000 ||
    customer.CCType == "Visa")
{
    SendGift(customer);
}
```

And Reveal:

```
var customers = CustomersRepository.LoadCustomers();
```

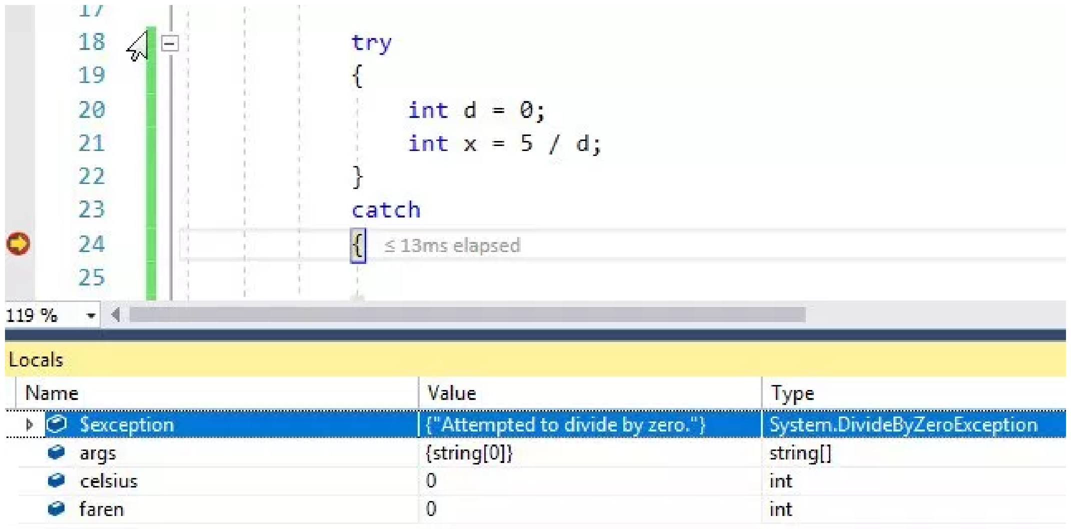
> **Disclaimer**: I am currently an OzCode employee. This is however, my personal blog and this post is not a promotion, just a professional recommendation.

6. Special $ variables

VS debugger exposes some special variables, all starting with the $ sign.

$exception

In a catch clause, **$exception** will give the exception instance value. You can see it in **Locals** tool window. You can also evaluate it in **Immediate Window** or any **Watch** tool window.



```
try
{
    int d = 0;
    int x = 5 / d;
}
catch
{ ≤ 13ms elapsed
```

119 %

Locals

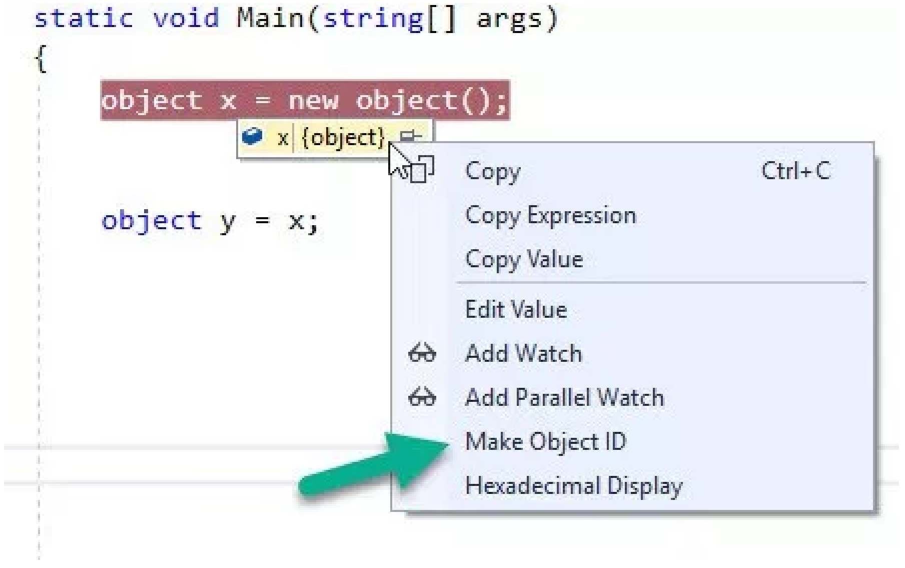
Name	Value	Type
$exception	{"Attempted to divide by zero."}	System.DivideByZeroException
args	{string[0]}	string[]
celsius	0	int
faren	0	int

$ReturnValue

You can see a method's returns value with the special **$ReturnValue** variable. You'll need to place a breakpoint in the method's last line, and then press F10 once more. Afterwards, evaluate $ReturnValue in the immediate window or the Watch window. The same value will appear in the **Autos** and **Local** windows.

Make Object ID – $1, $2, …

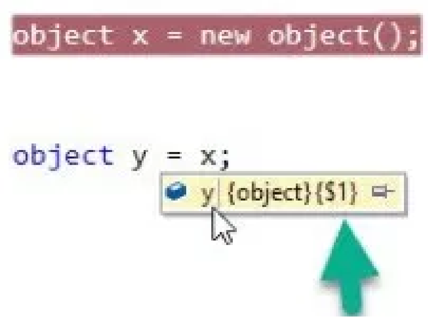
A useful feature in VS is to give ID number to a specific object instance. This is available from the hover DataTip:



```
static void Main(string[] args)
{
    object x = new object();

    object y = x;
```

This will give this instance the ID **$1**. Typing $1 in the immediate window or watch window will evaluate this instance (works globally, in any breakpoint scope). Any variable referencing $1 will show the $1 in the DataTip:

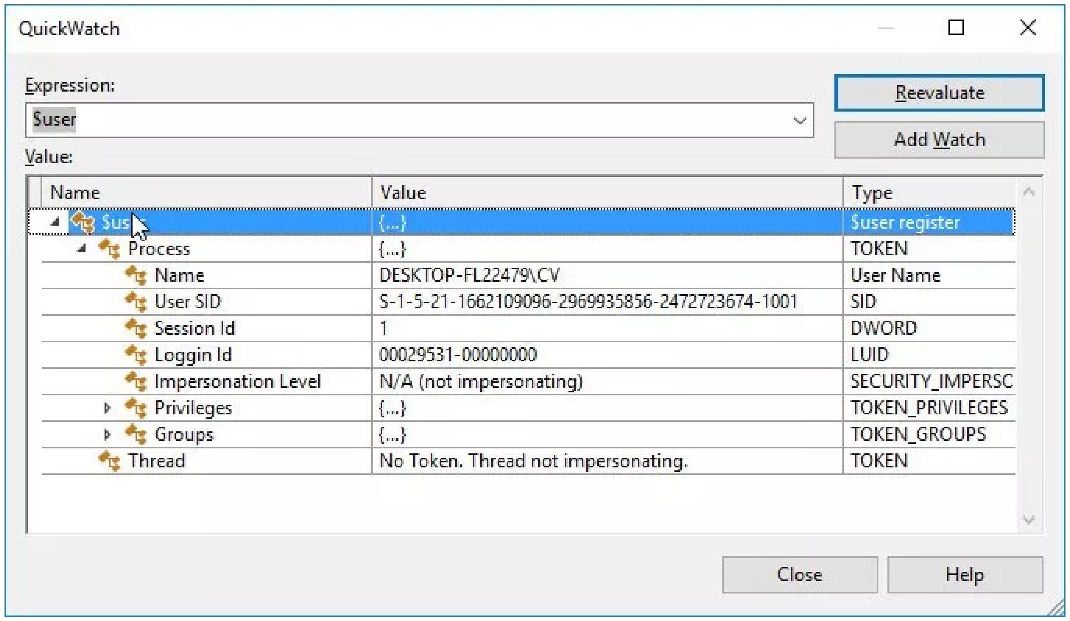


```
object x = new object();

object y = x;
```

$user

Holds information on the user that’s running the current process



Name	Value	Type
$us	{...}	$user register
Process	{...}	TOKEN
Name	DESKTOP-FL22479\CV	User Name
User SID	S-1-5-21-1662109096-2969935856-2472723674-1001	SID
Session Id	1	DWORD
Loggin Id	00029531-00000000	LUID
Impersonation Level	N/A (not impersonating)	SECURITY_IMPERSC
Privileges	{...}	TOKEN_PRIVILEGES
Groups	{...}	TOKEN_GROUPS
Thread	No Token. Thread not impersonating.	TOKEN

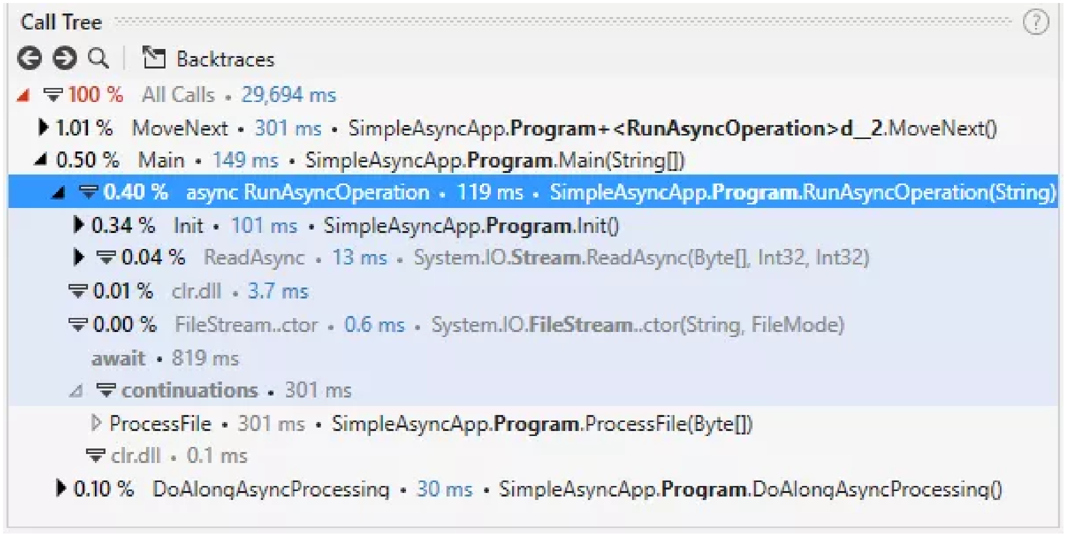
7. Make sense of a complicated scenario with dotTrace Performance Profiler

Sometimes, we have to work on new, unfamiliar projects. Suppose you have to fix a bug in such a new project, where you can reproduce the bug, but have no idea what happened in the code. How do you even know where to place a breakpoint and start debugging?

There are a lot of ways to go about it. You can try to find UI event handlers, guess class names according to functionality or ask someone for help. I'd like to suggest another alternative: Use a performance profiler!

dotTrace is able to attach to a process and “record” a runtime section. The recording is called a **Snapshot**.

Suppose I record the 10 seconds where my bug reproduces. Now, dotTrace will show a list of all the methods called during the recording.

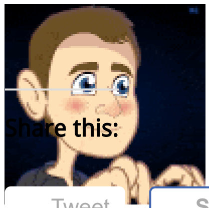


```
Call Tree
Backtraces
◢ 100 %  All Calls • 29,694 ms
  ▶ 1.01 %  MoveNext • 301 ms • SimpleAsyncApp.Program+<RunAsyncOperation>d__2.MoveNext()
  ◢ 0.50 %  Main • 149 ms • SimpleAsyncApp.Program.Main(String[])
    ◢ 0.40 %  async RunAsyncOperation • 119 ms • SimpleAsyncApp.Program.RunAsyncOperation(String)
      ▶ 0.34 %  Init • 101 ms • SimpleAsyncApp.Program.Init()
      ▶ 0.04 %  ReadAsync • 13 ms • System.IO.Stream.ReadAsync(Byte[], Int32, Int32)
      0.01 %  clr.dll • 3.7 ms
      0.00 %  FileStream..ctor • 0.6 ms • System.IO.FileStream..ctor(String, FileMode)
        await • 819 ms
      ◿ continuations • 301 ms
        ▷ ProcessFile • 301 ms • SimpleAsyncApp.Program.ProcessFile(Byte[])
        clr.dll • 0.1 ms
    ▶ 0.10 %  DoAlongAsyncProcessing • 30 ms • SimpleAsyncApp.Program.DoAlongAsyncProcessing()
```

There’s a bunch of useful information and features, much for performance profiling. In particular, you can see how much time was spent on each method. You can also view which methods were called by a specific method and the other way round – which method called the specific method.
So dotTrace is useful not just to solve performance issues. Whenever I’m working on unfamiliar territory, dotTrace will give me a picture of what happens in the code!

Subscribe to get
post updates and:

EMAIL



Performance Optimizations in C# .NET: 10 Best Practices

SUBSCRIBE

Share this:

Tweet Share 10 SHARE

‹ PREVIOUS

Visual Studio 2017 Extension development tutorial part 5: Highlight code in Editor

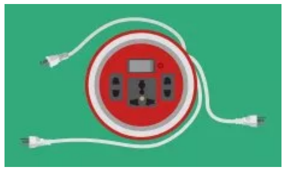
NEXT ›

Visual Studio 2017 Extension development tutorial part 6: Follow a span of code with TrackingSpan

RELATED POSTS

How to Debug LINQ queries in C#

June 13, 2019



Extension Methods Guidelines in C# .NET

June 5, 2019

How to resolve .NET reference and NuGet package version conflicts

April 24, 2018

C# Features: Innovations or Imitations? – Part 2

October 2, 2018

10 COMMENTS

Lidan on January 9, 2018 at 9:51 pm

Nice one Michael, I learned a few things…

REPLY

michaels9876@gmail.com on January 9, 2018 at 10:57 pm

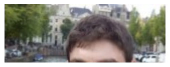
Glad to hear, and Thanks!

REPLY

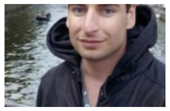
getcodetrack on January 10, 2018 at 12:05 am

Didn't know about the ,nse option. Have you tried out CodeTrack yet ? It gives you way more info than dottrace for this purpose (and it's free). You can get it for free at http://www.getcodetrack.com !

REPLY



michaels9876@gmail.com on January 10, 2018 at 11:12 am



Didn’t know about CodeTrack, but it looks interesting! I’ll give it a try.

REPLY

Mike Kellogg on January 17, 2018 at 5:55 pm

Nice post. I was just telling someone the other day that debugging is a way more important skill for a dev than just writing. I picked up a few things here; the $return would save me from always declaring a var prior to exiting a method just so I can take a look at it. And the Oz HUD looks really interesting. Thanks for the post!

REPLY

michaels9876@gmail.com on January 17, 2018 at 7:53 pm

You welcome Mike, glad you enjoyed it

REPLY

Vinay on February 6, 2019 at 12:50 pm

Awespme Blog, please keep writing 🙂

REPLY

Michael Shpilt on February 6, 2019 at 5:02 pm

Thanks, Will do!

REPLY

Zhiyuan Ma on June 14, 2019 at 5:32 am

could I translate your blog to chinese?

REPLY

Michael Shpilt on June 14, 2019 at 3:11 pm

Sure thing, just please link to the original article. And I'd love a link to the Chinese version as well 🙂

REPLY

TRACKBACKS/PINGBACKS

1. The Complete Archive of C# .NET Debugging Resources: Best Practices and Handy Tips | OzCode - [...] https://michaelscodingspot.com/7-debugging-techniques-know-c-net/ [...]

LEAVE A REPLY

Enter your comment here...

Privacy Policy

Debugging Tutorial