Interactively debug .NET apps with the Visual Studio Code debugger

Billy Vanegas


C# 9.0

Debugging with VSCode and C#

- Debugger
  - A debugger is a software tool used to observe and control the execution flow of your program with an analytical approach. Its design goal is to help find the root cause of a bug and help you resolve it. It works by either hosting your program in its own execution process or running as a separate process that's attached to your running program, like .NET.
  - Debuggers come in different flavors. Some work directly from the command line while others come with a graphical user interface. In this module, we'll use the integrated graphical debugger of Visual Studio Code.

Activate debug

File Edit Selection View Go Run Terminal Help • Program.cs - DotNetDebugging - Visual Studio Code

EXPLORER

OPEN EDITORS 1 UNSAVED

• Program.cs

DOTNETDEBUGGING

bin

obj

DotNetDebugging.csproj

Program.cs

Program.cs •

```
1  int result = Fibonacci(5);
2  Console.WriteLine(result);
3
4  static int Fibonacci(int n)
5  {
6      int n1 = 0;
7      int n2 = 1;
8      int sum;
9
10     for (int i = 2; i < n; i++)
11     {
12         sum = n1 + n2;
13         n1 = n2;
14         n2 = sum;
15     }
16
17     return n == 0 ? n1 : n2;
18 }
```

Activate debug and other assets

NOTIFICATIONS

Required assets to build and debug are missing from 'DotNetDebugging'. Add them?

Source: C# (Extension) Don't Ask Again Not Now Yes

OUTLINE

0 0 DotNetDebugging Ln 18, Col 2 Spaces: 4 UTF-8 with BOM CRLF C#

Visual Studio Code debugger overview

File Edit Selection View Go Run Terminal Help — Program.cs - DotNetDebugging - Visual Studio Code

1 RUN AND DEBUG .NET Core Launch (console)

2 VARIABLES
Locals
n [int]: 5
n1 [int]: 0
n2 [int]: 0
sum [int]: 0

3 WATCH

4 CALL STACK — PAUSED ON BREAKPOINT
DotNetDebugging.dll!Program.<Main>$.__Fibonacci|0_
DotNetDebugging.dll!Program.<Main>$(string[] args)
[External Code] Unknown Source 0

5 BREAKPOINTS
All Exceptions
User-Unhandled Exceptions
Program.cs 6

6 Program.cs

```
int result = Fibonacci(5);
Console.WriteLine(result);

static int Fibonacci(int n)
{
    int n1 = 0;
    int n2 = 1;
    int sum;

    for (int i = 2; i < n; i++)
    {
        sum = n1 + n2;
        n1 = n2;
        n2 = sum;
    }

    return n == 0 ? n1 : n2;
```

7

8 TERMINAL OUTPUT DEBUG CONSOLE — Filter (e.g. text, !exclude)

```
to help you develop and test your applications.
---------------------------------------------------------------
Loaded 'C:\Program Files\dotnet\shared\Microsoft.NETCore.App\6.0.5\System.Private.CoreLib.dll'. Skipped loading symbols. Module is optimized and the debugger option 'Just My Code' is enabled.
Loaded 'D:\00-CComputing\Doc\05-eCSharp\Ejercicios\DotNetDebugging\bin\Debug\net6.0\DotNetDebugging.dll'. Symbols loaded.
Loaded 'C:\Program Files\dotnet\shared\Microsoft.NETCore.App\6.0.5\System.Runtime.dll'. Skipped loading symbols. Module is optimized and the debugger option 'Just My Code' is enabled.
Loaded 'C:\Program Files\dotnet\shared\Microsoft.NETCore.App\6.0.5\System.Console.dll'. Skipped loading symbols. Module is optimized and the debugger option 'Just My Code' is enabled.
```

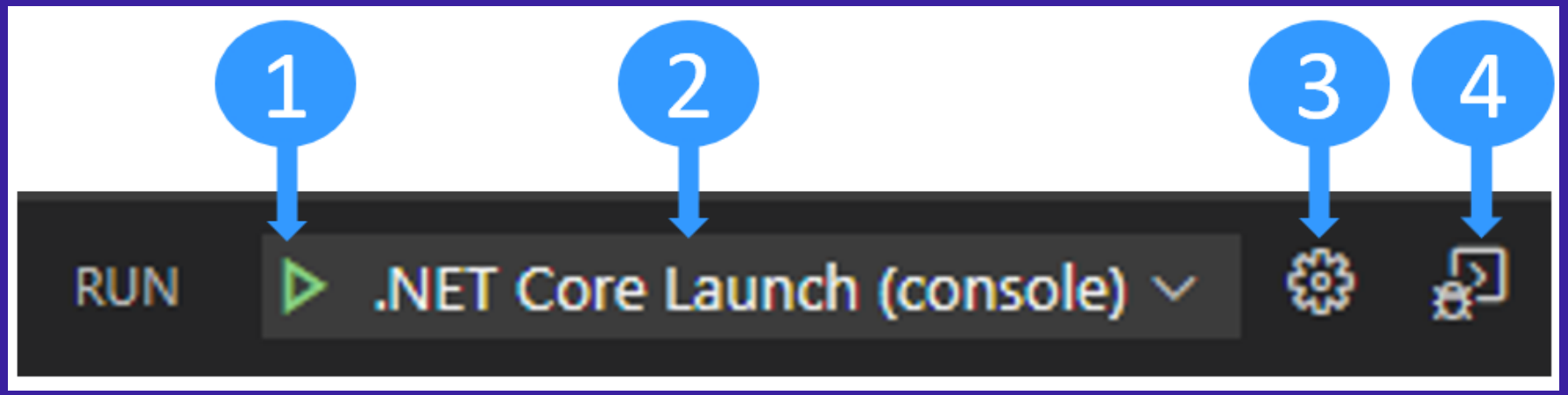
⊗ 0 ⚠ 0 .NET Core Launch (console) (DotNetDebugging) DotNetDebugging Ln 6, Col 16 (11 selected) Spaces: 4 UTF-8 with BOM CRLF C#

Debugging with VSCode and C# (Cont.)

- Configure Visual Studio Code for .NET debugging
  1. Debugger launch controls
  2. Variables state
  3. Watched variables state
  4. Current call stack
  5. Breakpoints
  6. Execution controls
  7. Current execution step
  8. Debug console

Debugging with VSCode and C# (Cont.)

1. Debugger launch controls



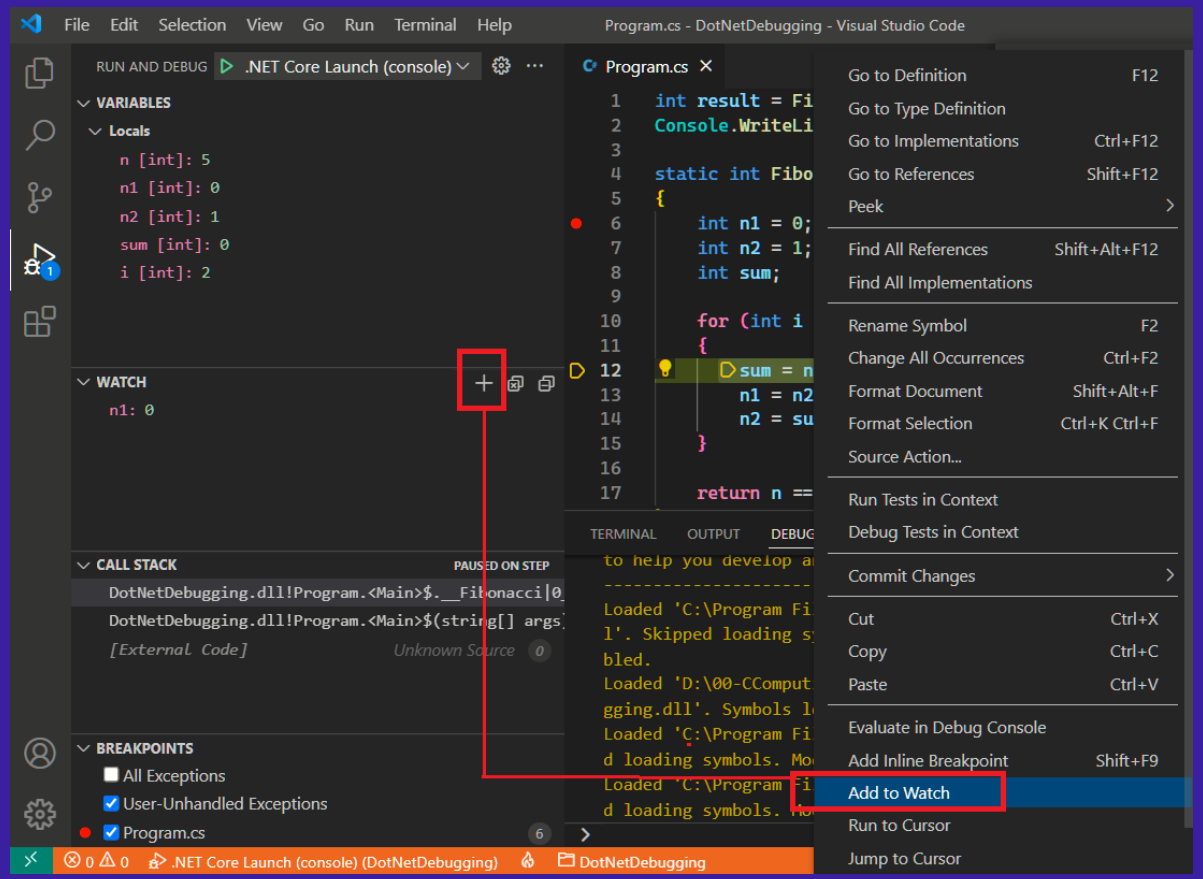
1. Start debugging.
2. Select the active launch configuration.
3. Edit the launch.json file. Create it if you need to.
4. Open the debug terminal.

Debugging with VSCode and C# (Cont.)

2. Variables state
   - **Local variables** are accessible in the current scope, usually the current function.
   - **Global variables** are accessible from everywhere in your program. System objects from the JavaScript runtime are also included, so don't be surprised if you see a lot of stuff in there.
   - **Closure variables** are accessible from the current closure, if any. A closure combines the local scope of a function with the scope from the outer function it belongs to.

Debugging with VSCode and C# (Cont.)

3. Watched variables state



- To track a variable state across time or different functions
- Select the plus button to enter a variable name or an expression to watch
- Right-click a variable in the Variables panel and select Add to watch

Debugging with VSCode and C# (Cont.)

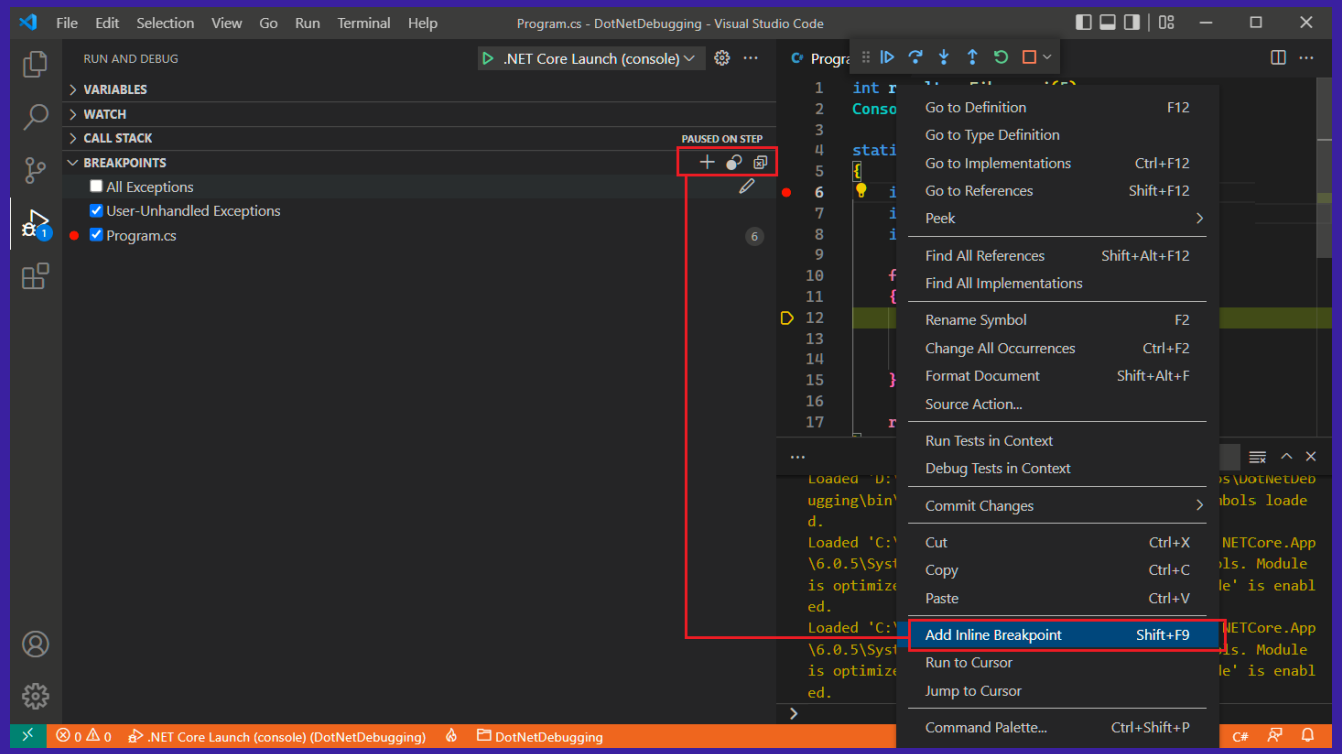
4. Current call stack

```
CALL STACK                                                          PAUSED ON STEP
  DotNetDebugging.dll!Program.<Main>$.__Fibonacci|0_0(int n) Line 12     Program.cs   12:9
  DotNetDebugging.dll!Program.<Main>$(string[] args) Line 1             Program.cs   1:1
  [External Code]                                                       Unknown Source   0
```

- To track an entrance of every time your methods are called.
- Very usefully to find an exception.

Debugging with VSCode and C# (Cont.)

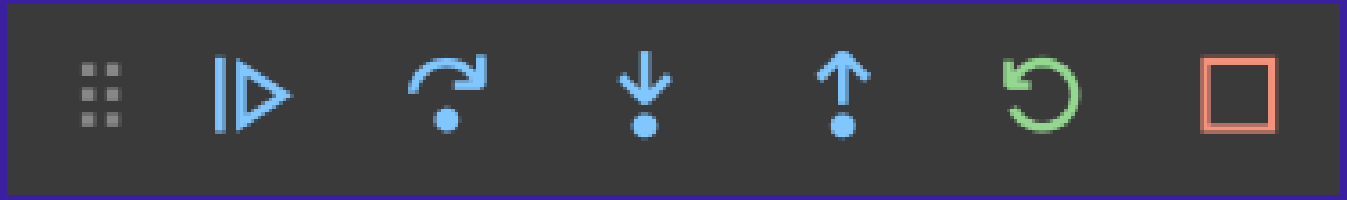
5. Breakpoints and 7. Current execution Steps



- See and toggle all the breakpoints you placed in your code.
- Toggle options to break on caught or uncaught exceptions
- Right-click a line to add or remove a breakpoint

Debugging with VSCode and C# (Cont.)

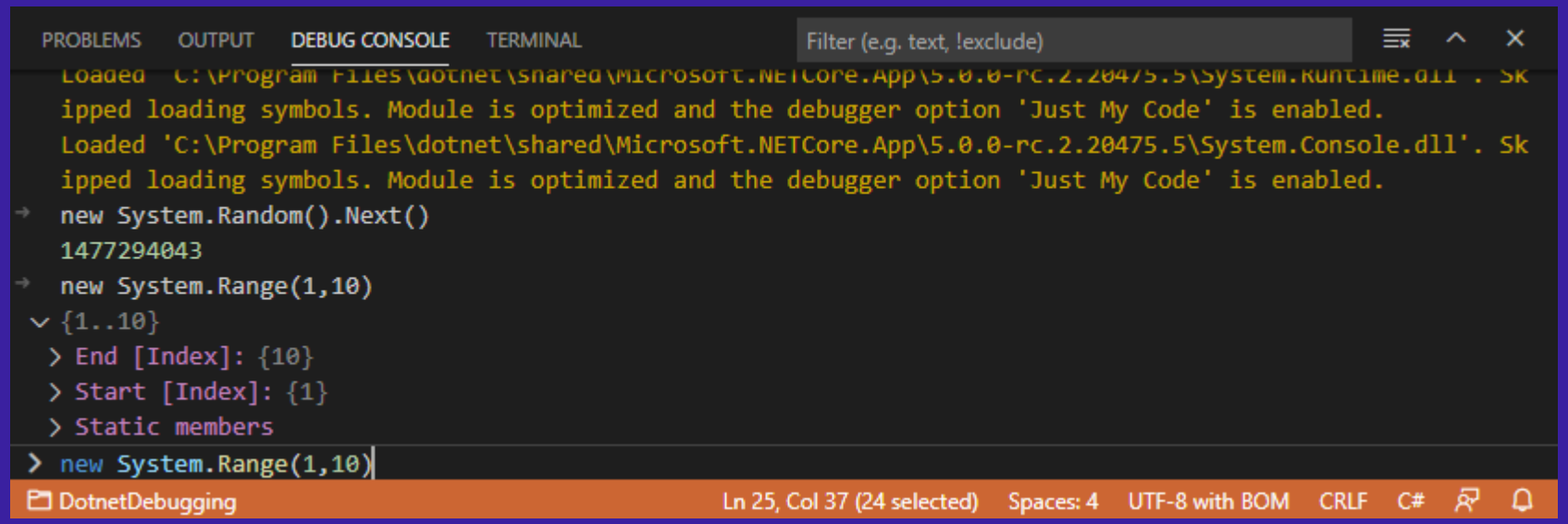
6. Execution controls



- Continue or pause execution.
- Step over.
- Step into.
- Step out.
- Restart.
- Stop.

Debugging with VSCode and C# (Cont.)

8. Debug console



- To visualize your application console logs
- To evaluate expressions or execute code in the current execution content, like commands or variable names in the built-in .NET debugger.

Logging and tracing .NET applications

- Tracing is a way for you to monitor the execution of your application while it's running.
- You can add tracing and debugging instrumentation to your .NET application when you develop it.
- You can use that instrumentation while you're developing the application and after you've deployed it.

Logging and tracing .NET applications (Cont.)

- System.Console
  - Always enabled and always writes to the console.
  - Useful for information that your customer might need to see in the release.
  - Because it's the simplest approach, it's often used for ad-hoc temporary debugging. This debug code is often never checked in to source control.
- System.Diagnostics.Trace
  - Only enabled when TRACE is defined.
  - Writes to attached Listeners, by default, the DefaultTraceListener.
  - Use this API when you create logs that will be enabled in most builds.
- System.Diagnostics.Debug
  - Only enabled when DEBUG is defined (when in debug mode).
  - Writes to an attached debugger.
  - Use this API when you create logs that will be enabled only in debug builds.

Define TRACE and DEBUG constants

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>…
  </PropertyGroup>

  <PropertyGroup Condition="'$(Configuration)|$(Platform)'=='Debug|AnyCPU'">
    <DefineConstants>DEBUG;TRACE</DefineConstants>
  </PropertyGroup>
  <PropertyGroup Condition="'$(Configuration)|$(Platform)'=='Release|AnyCPU'">
    <DefineConstants>TRACE</DefineConstants>
  </PropertyGroup>

</Project>
```

Conditional tracing

```
if (count == 0) {
        Debug.WriteLine( "The count is 0 and this may cause an exception.");

}

Debug.WriteLineIf(count==0,"The count is 0 and this may cause an exception.");

bool errorFlag = false;
System.Diagnostics.Trace.WriteIf(errorFlag, "Error in AppendData procedure.");
System.Diagnostics.Debug.WriteIf(errorFlag, "Transaction abandoned.");
System.Diagnostics.Trace.Write( "Invalid value for data request ");
```

Verify that certain conditions exist

```
int  IntegerDivide(int dividend, int divisor) {

    Debug.Assert(divisor != 0, $"{nameof(divisor)} is 0 and will cause an exception.");
    return dividend / divisor;

}
```

Exercise – Logging and tracing

- Write to the debug console
- Check for conditions with Assert

Thanks!