How to dig into the CLR

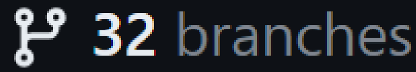
Christophe Nasarre November 12, 2023

Introduction

When I started to work on the second edition of *Pro .NET Memory Management : For Better Code, Performance, and Scalability* by Konrad Kokosa, I already spent some time in the CLR code for a couple of pull requests related to the garbage collector. However, updating the book to cover 5 new versions of .NET requires looking at new APIs but also digging deep inside the CLR (and especially the GC) hundreds of thousand lines of code!

The first step is to install Visual Studio 2022 Preview that allows you to compile and run projects targeting .NET 8. Then, goto https://github.com/dotnet/runtime and git clone the tag of the .NET 8 preview version you have installed.

runtime Public



Switch branches/tags

Find a tag

Branches Tags

v8.0.0-rc.1.23419.4

v8.0.0-preview.7.23375.6

v8.0.0-preview.6.23329.7

v8.0.0-preview.5.23280.8

v8.0.0-preview.4.23259.5

v8.0.0-preview.3.23174.8

That way, you will be able to directly run the same version that you will debug.

And now, what are the next steps?

The goal of this post is to share with you the tips and tricks I used to navigate into the CLR implementation so you could better understand how things are working.

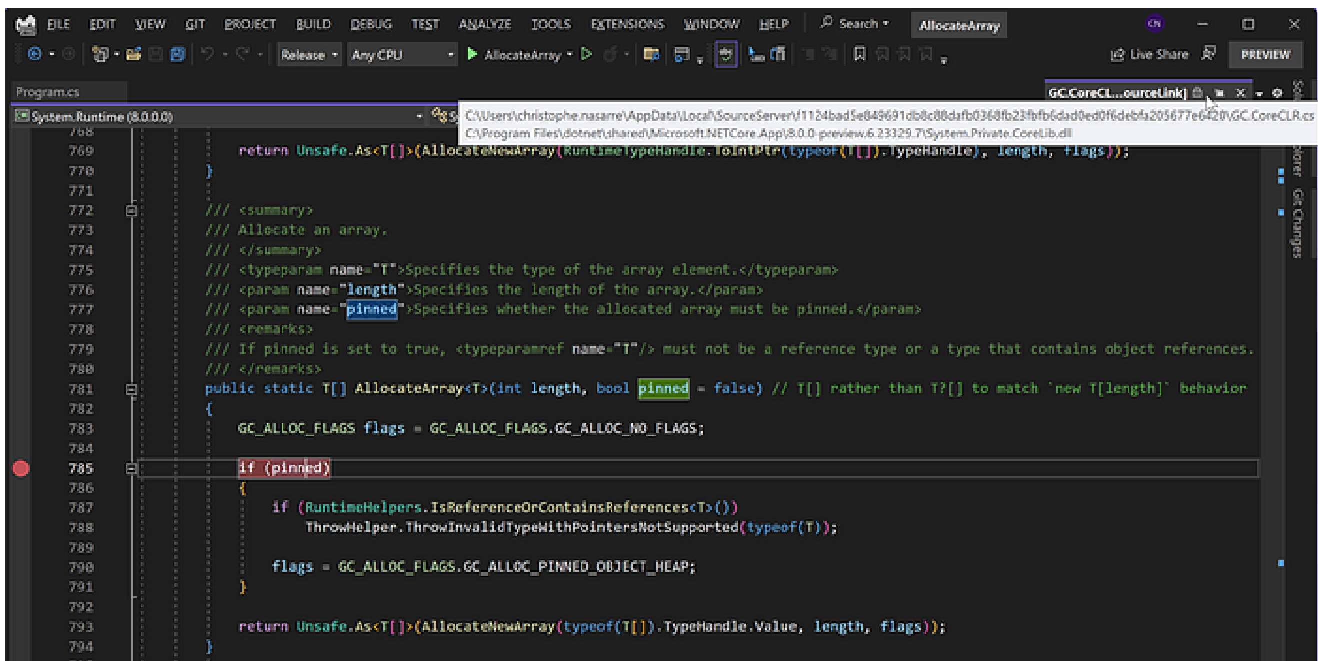
From C# to C++

As a .NET developer, I'm used to the APIs provided by the Base Class Library built on top of the CLR. Let's take as an example the following code that is using the GC.AllocateArray method that allows you to allocate a pinned in memory array and available since .NET 5.

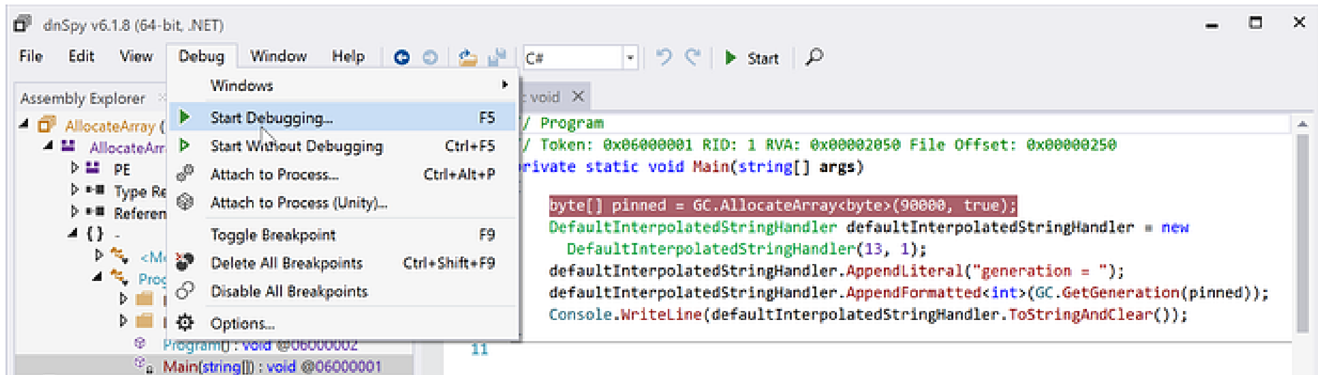
```
using System;

  {        {        [] pinned = GC.AllocateArray<>(, );        Console.WriteLine();
}}
```

When you Ctrl+click the method name (or use F12), thanks to Source Link integration, you go to its implementation where you can even set breakpoint:



If you don't use Visual Studio, you could open the generated assembly into a decompiler such as ILSpy or DnSpy. The latter even allows you to set breakpoints and debug the disassembly IL without any source.



In both cases, only the managed implementation will be available: you soon end up to an “internal call” corresponding to a native function implemented by the CLR. The managed methods are decorated with the MethodImplOptions.InternalCall attribute.

```
// Token: 0x0600053C RID: 1340
[MethodImpl(MethodImplOptions.InternalCall)]
internal static extern Array AllocateNewArray(IntPtr typeHandle, int length, GC.GC_ALLOC_FLAGS flags);
```

```
// Token: 0x0600053D RID: 1341
[MethodImpl(MethodImplOptions.InternalCall)]
private static extern int GetGenerationWR(IntPtr handle);
```

```
// Token: 0x0600053E RID: 1342
[LibraryImport("QCall", EntryPoint = "GCInterface_GetTotalMemory")]
[DllImport("QCall", EntryPoint = "GCInterface_GetTotalMemory", ExactSpelling = true)]
private static extern long GetTotalMemory();
```

```
// Token: 0x0600053F RID: 1343
[LibraryImport("QCall", EntryPoint = "GCInterface_Collect")]
[DllImport("QCall", EntryPoint = "GCInterface_Collect", ExactSpelling = true)]
private static extern void _Collect(int generation, int mode);
```

For the garbage collector code, you can look into the GC.CoreCLR.cs file where these methods are defined. You can note some methods decorated with the DllImport attribute to bind to native functions exported by a “QCall” library. There is an optimized path in P/Invoke done by the JIT to transform these calls not like a usual LoadLibrary/GetProcAddress as you could expect. Instead, they will be routed to the exported methods by coredll.dll and defined in the **s_QCall** array in qcallentrypoints.cpp. But where to look further for the native implementation?

Instead of searching among the thousands of files, focus on comutilnative.h that defines the signature of most exported functions. The implementation of the exported native functions is found in comutilnative.cpp. This is where you should start your journey in the native implementation of the CLR. For the list of **all** functions called by the libraries in the runtime, look at the ecalllist.h file (around **gGCInterfaceFuncs** and **gGCSettingsFuncs** specifically for the GC).

Note that you might also find some implementations under the folder like in the file for .

CLR Source code debugging

It is nice to know that the implementation of most CLR exported native functions used by the BCL is in comutilnative.cpp. For the GC, the functions are either statics from the GCInterface class or static functions prefixed by **GCInterface_**; I don't know why all are not part of **GCInterface**…

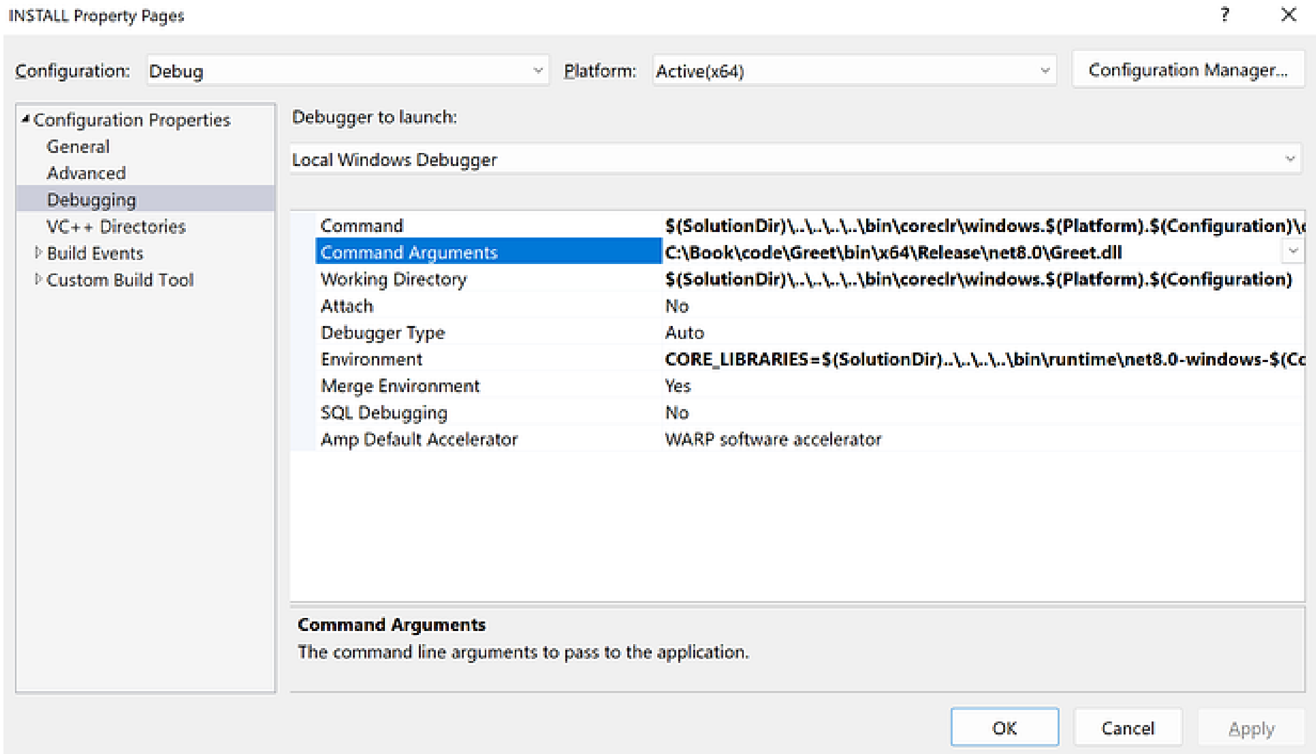
When you look at the GC-related methods implementation, a lot are calling methods from the instance returned by GCHeapUtilities::GetGCHeap() that corresponds to the static g_pGCHeap global variable. It is interesting to follow the threads of calls like that, but I have to admit that, after a few hops, I'm starting to get lost. So, I'm drawing boxes for types on a piece of paper and arrays from their fields to other types as boxes.

However, with a code base that big, I definitively prefer to set breakpoints and write a small C# application to call the methods I'm interested in and see what data structures are used in the different layers of implementation. Don't be scared: WinDBG is not required to achieve this goal. As this page explains, you need to type the following commands in a shell at the root of the repo:

.\build.cmd -s clr -c Debug.\build.cmd clr.nativeprereqs -a x64 -c debug.\build.cmd -msbuild

The last command generates a CoreCLR.sln solution file in artifacts\obj\coreclr\windows.x64.Debug\ide) that you can open in Visual Studio 2022 Preview.

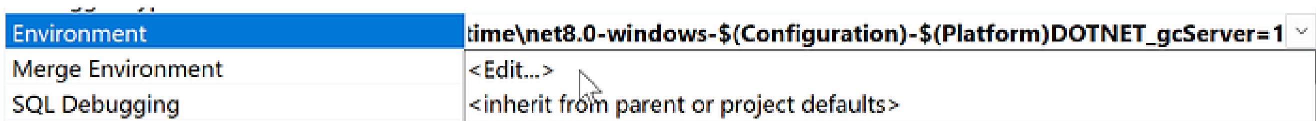
In VS, right-click the **INSTALL** project, select Properties and setup the Debugging properties



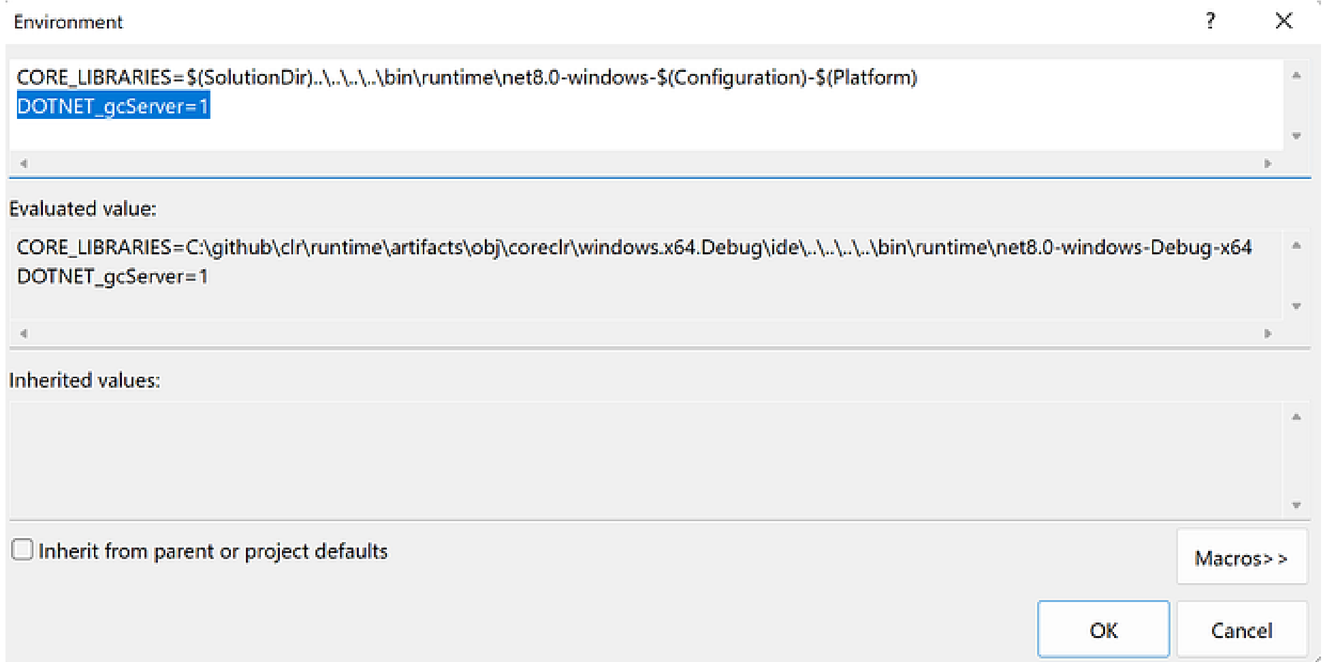
Here are the details of each property:

Command	$(SolutionDir)\..\..\..\..\bin\coreclr\windows.$(Platform).$(Configuration)\corerun.exe
Command Arguments	Full path to your compiled dll you want to debug
Working Directory	$(SolutionDir)\..\..\..\..\bin\coreclr\windows.$(Platform).$(Configuration)
Environment	CORE_LIBRARIES=$(SolutionDir)..\..\..\..\bin\runtime\net8.0-windows-$(Configuration)-$(Platform)

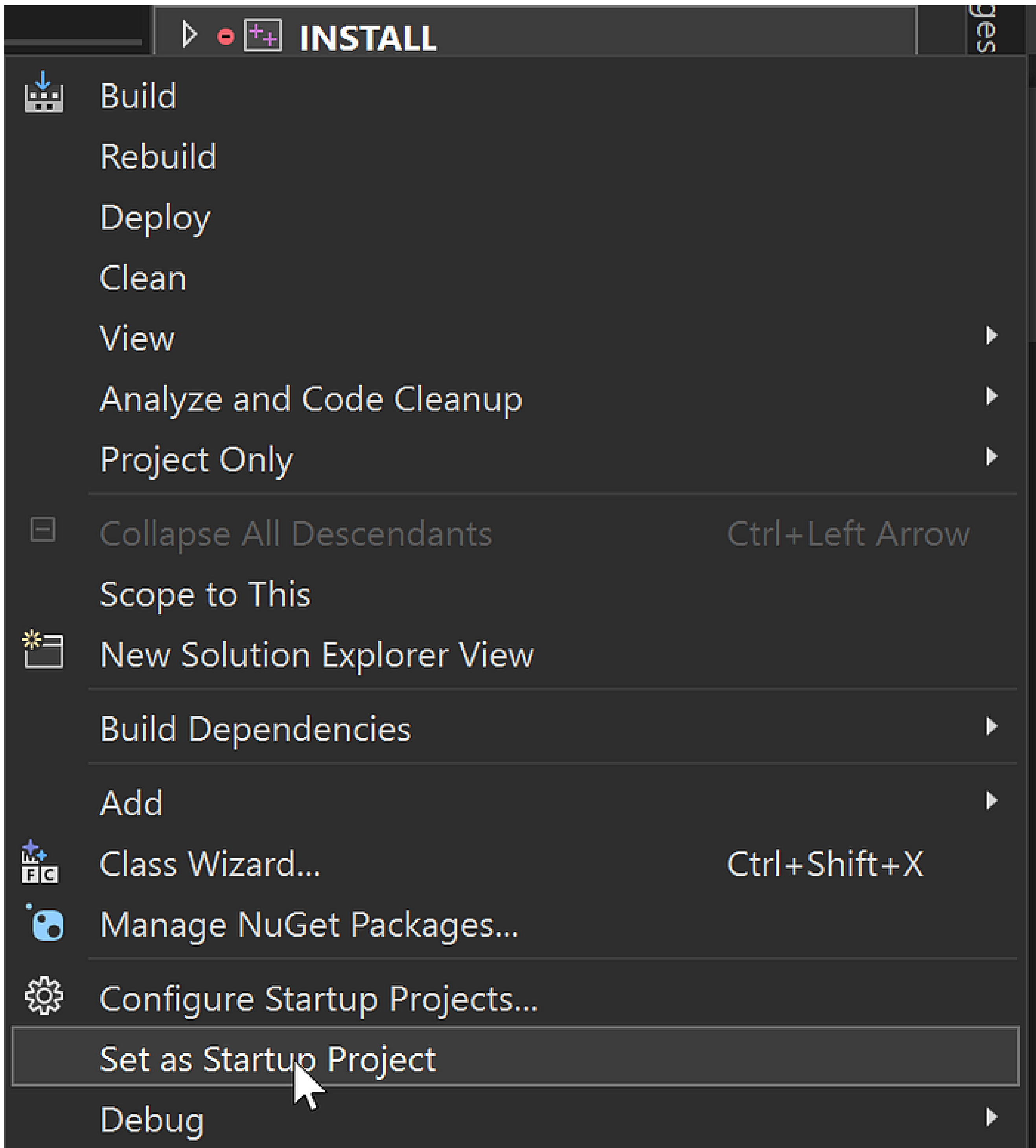
It could be interesting to set some environment variables such as **DOTNET_gcServer** to 1 for a GC Server configuration instead of workstation. In that case, click the <Edit..> choice in the combo-box:



And update the textbox at the top:



The final step is to set this project as the startup project:

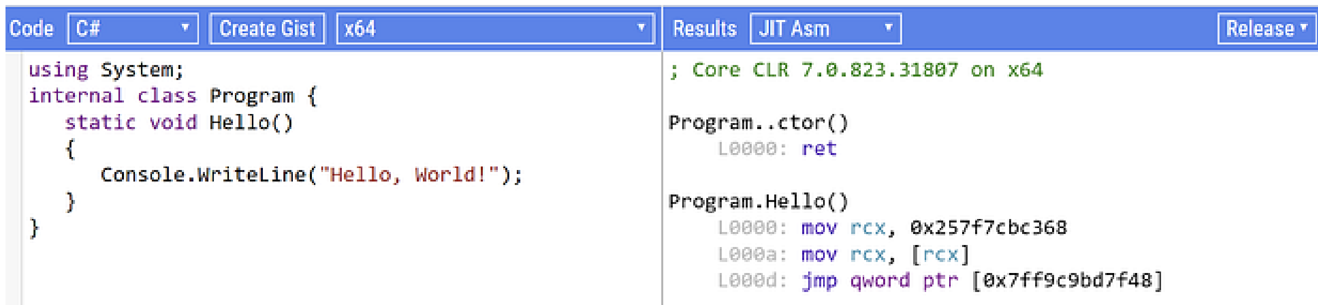


You are now able to set the breakpoint you want in the native code of the CLR and type F5/Debug in Visual Studio to step into the code!

And what about the assembly code?

Some specific data structures, such as the NonGC Heap, are used by the JIT compiler when generating the assembly code from the IL compiled from your C# code. It means that you need to look at that JITted code to fully understand what is going on.

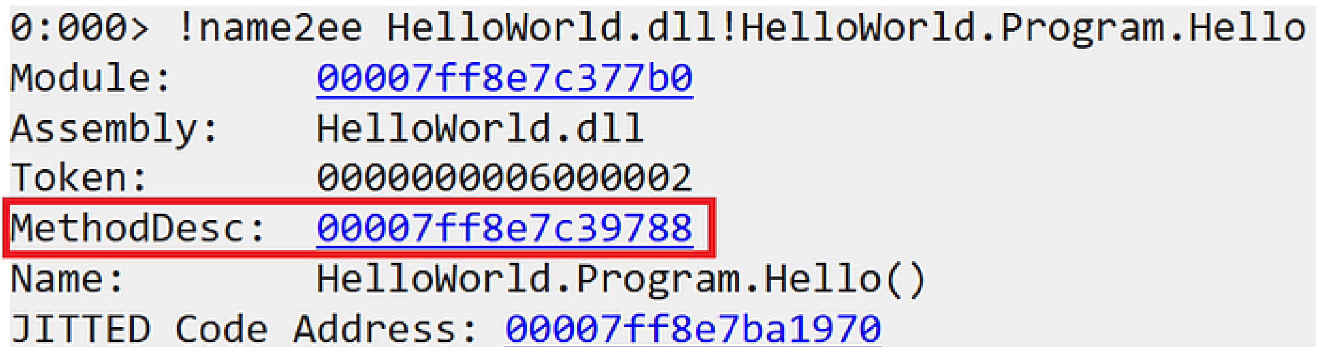
A first way to get it is to use https://sharplab.io/, type your C# code and select x64 for Core of x86/x64 for Framework:



But as you can see from this screenshot, it is using the .NET 7 compiler. What if you would like to see the .NET 8 compilation result just in case something changed?

The solution I'm using is to generate a memory dump with procdump -ma <pid> of a test application. Before opening the dump in WinDBG, there is something you should be aware of: with the tiered compilation, you will need to call a method several times before the final optimized assembly code gets JITed. Or… decorate the method you are interested in with the [MethodImpl(MethodImplOptions.AggressiveOptimization)] attribute to instruct the JIT to directly generate the most optimized tier.

Once the dump loaded in WinDBG, the first step is to get the MethodTable pointer corresponding to the method you are interested in. For that, use the **name2ee** SOS command:



Click the link corresponding to MethodDesc to run the **dumpmd** SOS command:

```
0:000> !DumpMD /d 00007ff8e7c39788
Method Name:          HelloWorld.Program.Hello()
Class:                00007ff8e7c4b4e0
MethodTable:          00007ff8e7c397b0
mdToken:              0000000006000002
Module:               00007ff8e7c377b0
IsJitted:             yes
Current CodeAddr:     00007ff8e7ba1970
Version History:
  ILCodeVersion:      0000000000000000
  ReJIT ID:           0
  IL Addr:            00000265d2d9205d
     CodeAddr:           00007ff8e7ba1970  (QuickJitted)
     NativeCodeVersion:  0000000000000000
```

The last step is to click the link corresponding to CodeAddr to run the **U** command and see the JITted assembly code:

```
0:000> !U /d 00007ff8e7ba1970
Normal JIT generated code
HelloWorld.Program.Hello()
ilAddr is 00000265D2D9205D pImport is 00000286F28FEB80
Begin 00007FF8E7BA1970, size 21

C:\Book\Pro .NET Memory V2\blogs\How to dig into the CLR\Code\HelloWorld\Program.cs @ 16:
>>> 00007ff8`e7ba1970 55              push    rbp
00007ff8`e7ba1971 4883ec20        sub     rsp,20h
00007ff8`e7ba1975 488d6c2420      lea     rbp,[rsp+20h]
00007ff8`e7ba197a 48b9c804d867a6020000 mov rcx,2A667D804C8h ("Hello, World!")
00007ff8`e7ba1984 ff15cec00b00    call    qword ptr [00007ff8`e7c5da58]

C:\Book\Pro .NET Memory V2\blogs\How to dig into the CLR\Code\HelloWorld\Program.cs @ 17:
00007ff8`e7ba198a 90              nop
00007ff8`e7ba198b 4883c420        add     rsp,20h
00007ff8`e7ba198f 5d              pop     rbp
00007ff8`e7ba1990 c3              ret
```

If you compare this code to get the “Hello, World!” string, with the one shown by sharplab,

```
[]        []
```

you might notice a tiny difference: there is one less indirection in .NET 8!
But this is another story that will be told in the second edition of the “Pro .NET Memory Management: For Better Code, Performance, and Scalability” book ;^)