Intro to Windows Debugger (WinDBG) for .NET Developers
And
Concepts in C#

Vijay Rodrigues (VijayRod)

Last Updated: 2011-11-30

I've written this content as part of driving .Net Academy. Most of content has been written at home in my spare time with a real slow internet connection.

A lot of acknowledgements to Ranjan Bhattacharjee for this idea, to Xavier S Raj for valuable mentorship and for the logo, and to Rishi Maini for his guidance whenever required.

We've also discussed this with multiple resources and got ideas from them. They've made a difference - Kane Conway, Satya Madhuri Krovvidi, Dhiraj H N, Durai Murugan, Varun Dhawan, Balmukund Lakhani, Anmol Bhasin, Tejas Shah, Ajith Krishnan, Shasank Prasad, Chandresh Hegde.

If you have any new inputs or suggestion that will help this effort please let me know.

Table of Contents

21. Intro to WinDBG for .NET Developers - part XV (still more CLR internals - Conditionals)
22. Intro to WinDBG for .NET Developers - part XVI (still more CLR internals - Loops)
23. Intro to WinDBG for .NET Developers - part XVII (still more CLR internals - Methods)
24. Intro to WinDBG for .NET Developers - part XVIII (still more CLR internals - Classes)
25. Intro to WinDBG for .NET Developers - part XIX (still more CLR internals - Inheritance)
26. Intro to WinDBG for .NET Developers - part XX (still more CLR internals - Method overriding)
27. Intro to WinDBG for .NET Developers - part XXI (still more CLR internals - Access Modifiers)
28. Intro to WinDBG for .NET Developers - part XXII (still more CLR internals - Static)
29. Intro to WinDBG for .NET Developers - part XXIII (still more CLR internals - Properties)
30. Intro to WinDBG for .NET Developers - part XXIV (still more CLR internals - Indexers)
31. Intro to WinDBG for .NET Developers - part XXV (still more CLR internals - Interface)

To give a start

Thanks to Tejas Shah for this sample!

Have fun with this sample. And then try and figure out why the differences.

---------- Sample code----------

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Globalization;
using System.Threading;


namespace stringdctest
{
    class Program
    {
        static void spanishTest()
        {
            CultureInfo myCItradES = new CultureInfo(0x040A, false); //Thats es-
ES culture - same rules as Czech
            CultureInfo myCItradUS = new CultureInfo(0x0409, false); //Thats en-
US culture - same rules as Czech

            string s1 = "cHa";
            string s2 = "cha";
            int result1 = String.Compare(s1, s2, true, myCItradUS);
            Console.WriteLine("\nWhen the CurrentCulture is \"en-
US\",\nthe result of comparing {0} with {1} is: {2}", s1, s2, result1);

            result1 = String.Compare(s1, s2, true, myCItradES);
            Console.WriteLine("\nWhen the CurrentCulture is \"es-
ES\",\nthe result of comparing {0} with {1} is: {2}", s1, s2, result1);

        }

        static void danishTest()
        {
            CultureInfo myCItraddn = new CultureInfo(0x0406, false); //Thats da-
DK culture - same rules as Czech
            CultureInfo myCItradUS = new CultureInfo(0x0409, false); //Thats es-
ES culture - same rules as Czech

            string s1 = "aab";
            string s2 = "aAb";
            int result1 = String.Compare(s1, s2, true, myCItradUS);
            Console.WriteLine("\nWhen the CurrentCulture is \"en-
US\",\nthe result of comparing {0} with {1} is: {2}", s1, s2, result1);

            result1 = String.Compare(s1, s2, true, myCItraddn);
            Console.WriteLine("\nWhen the CurrentCulture is \"da-
DK\",\nthe result of comparing {0} with {1} is: {2}", s1, s2, result1);
```

```
        }
        static void Main(string[] args)
        {
            spanishTest();
            danishTest();
            Console.Read();

        }
    }
}
```

```
When the CurrentCulture is "en-US",
the result of comparing cHa with cha is: 0

When the CurrentCulture is "es-ES",
the result of comparing cHa with cha is: -1

When the CurrentCulture is "en-US",
the result of comparing aab with aAb is: 0

When the CurrentCulture is "da-DK",
the result of comparing aab with aAb is: 1
```

Answer

Interesting sample. String class represents text as a series of *Unicode* characters so I did not have to check on code pages hence the difference appears to be due to Cultures. This is also a case insensitive comparison.

Differences are due to using culture-sensitive operations when results should be independent of culture which can cause application code, like below sample, to return different results on cultures with Custom Case Mappings and Sorting Rules. For example, in the Danish language (da-DK in below sample), a case-insensitive comparison of the two-letter pairs "aA" and "AA" is not considered equal.

Most .NET Framework methods that perform culture-sensitive string operations by default provide method overloads that allow you to explicitly specify the culture to use by passing a CultureInfo parameter or a System.Globalization.CompareOptions parameter. These overloads allow you to eliminate cultural variations in case mappings and sorting rules and guarantee culture-insensitive results. For example and *depending on* the desired behavior of the application, String.Compare can be used with the System.Globalization.CompareOptions parameter to avoid customer mappings and to avoid custom sorting rules, if required.

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Globalization;
using System.Threading;
```

```
namespace stringdctest
{
   class Program
   {
      static void spanishTest()
      {
         CultureInfo myCItradES = new CultureInfo(0x040A, false); //Thats es-ES culture - same rules as
Czech
         CultureInfo myCItradUS = new CultureInfo(0x0409, false); //Thats en-US culture - same rules as
Czech

         string s1 = "cHa";
         string s2 = "cha";

         int result1 = String.Compare(s1, s2, true, myCItradUS);
         Console.WriteLine("\nWhen the CurrentCulture is \"en-US\",\nthe result of comparing {0} with
{1} is: {2}", s1, s2, result1);
         result1 = String.Compare(s1, s2, true, myCItradES);
         Console.WriteLine("\nWhen the CurrentCulture is \"es-ES\",\nthe result of comparing {0} with
{1} is: {2}", s1, s2, result1);
         result1 = String.Compare(s1, s2, myCItradES, CompareOptions.OrdinalIgnoreCase);
         Console.WriteLine("\nWhen the CurrentCulture is \"es-ES\" AND using OrdinalIgnoreCase,\nthe
result of comparing {0} with {1} is: {2}", s1, s2, result1);
      }

      static void danishTest()
      {
         CultureInfo myCItraddn = new CultureInfo(0x0406, false); //Thats da-DK culture - same rules as
Czech
         CultureInfo myCItradUS = new CultureInfo(0x0409, false); //Thats en-US culture - same rules as
Czech

         string s1 = "aab";
         string s2 = "aAb";

         int result1 = String.Compare(s1, s2, true, myCItradUS);
         Console.WriteLine("\nWhen the CurrentCulture is \"en-US\",\nthe result of comparing {0} with
{1} is: {2}", s1, s2, result1);
         result1 = String.Compare(s1, s2, true, myCItraddn);
         Console.WriteLine("\nWhen the CurrentCulture is \"da-DK\",\nthe result of comparing {0} with
{1} is: {2}", s1, s2, result1);
         result1 = String.Compare(s1, s2, myCItraddn, CompareOptions.OrdinalIgnoreCase);
         Console.WriteLine("\nWhen the CurrentCulture is \"da-DK\" AND using OrdinalIgnoreCase,\nthe
result of comparing {0} with {1} is: {2}", s1, s2, result1);
      }

      static void Main(string[] args)
      {
         spanishTest();
         danishTest();
         Console.Read();
      }
   }
}
```

When the CurrentCulture is "en-US",
the result of comparing cHa with cha is: 0
When the CurrentCulture is "es-ES",
the result of comparing cHa with cha is: -1
When the CurrentCulture is "es-ES" AND using OrdinalIgnoreCase,
the result of comparing cHa with cha is: 0
When the CurrentCulture is "en-US",
the result of comparing aab with aAb is: 0
When the CurrentCulture is "da-DK",
the result of comparing aab with aAb is: 1
When the CurrentCulture is "da-DK" AND using OrdinalIgnoreCase,
the result of comparing aab with aAb is: 0

More info:
http://msdn.microsoft.com/en-us/xk2wykcz
Custom Case Mappings and Sorting Rules

http://msdn.microsoft.com/en-us/x15ca6w0
Performing Culture-Insensitive String Operations

Learning - videos (and books)

Some of us were discussing on quick and easy ways to learn C#/.NET. We wanted to list this so as to help go through this data during any free time. Our discussions seemed to be on "easy" or "videos" (or "books") (I myself like reading but prefer quick videos). So we found some interesting videos (and books) which hopefully will help everyone get to know C#/.NET better. These also cover basics.

.NET ACADEMY

Are there videos to start self-learning?
(Some videos do not play online however may play if downloaded)

Although below is a 3rd party link, i really liked it for its simplicity and would probably be a great way for basic/intermediate learning. It's a practical link:
http://www.youtube.com/user/ProgrammingVideos#grid/user/37FF167549C26150
Programming Video Tutorials
C# Tutorial
A C# video tutorial series.

http://msdn.microsoft.com/en-us/library/bb820889(VS.90).aspx
Guided Tour Videos (Visual C#)

http://msdn.microsoft.com/en-us/library/360kwx3z.aspx
How to: Create a C# Windows Forms Application

http://msdn.microsoft.com/en-us/library/bb383877(v=VS.90).aspx
Creating Your First Visual C# Application

http://msdn.microsoft.com/en-us/library/bb383962(v=VS.90).aspx
Visual C# Guided Tour

http://channel9.msdn.com/Blogs/TheChannel9Team/Daniel-Fernandez-Your-first-C-application
Daniel Fernandez - Your first C# application
-- Dan was Product Manager

http://channel9.msdn.com/Blogs/TheChannel9Team/Anders-Hejlsberg-What-influenced-the-development-of-C
Anders Hejlsberg, the distinguished engineer who is one of the key designers on the C# team, talks about the languages and other things that inspired the development of C#.
-- One of my favorites

http://msdn.microsoft.com/en-us/vcsharp/bb798022.aspx
"How Do I?" Videos for Visual C#

http://msdn.microsoft.com/en-us/vcsharp/bb466180.aspx

Visual C# Videos

http://www.learnerstv.com/Free-Computers-Video-lectures-ltv033-Page1.htm
Operating Systems and System Programming
-- Video on fundamentals

http://www.learnerstv.com/Free-Computers-Video-lectures-ltv061-Page1.htm
Principles of Programming Languages
-- Video on fundamentals

http://blogs.msdn.com/b/tess/archive/2010/03/30/new-debugger-extension-for-net-psscor2.aspx

http://blogs.msdn.com/b/dougste/archive/2009/02/18/failed-to-load-data-access-dll-0x80004005-or-what-is-mscordacwks-dll.aspx
-- 32bit and 64bit

Learning - Advantages of C# over VB.NET and vice versa

http://blogs.msdn.com/b/csharpfaq/archive/2004/03/11/87816.aspx
What are the advantages of C# over VB.NET and vice versa?

{

The choice between C# and VB.NET is largely one of subjective preference. Some people like C#'s terse syntax, others like VB.NET's natural language, case-insensitive approach. Both have access to the same framework libraries. Both will perform largely equivalently (with a few small differences which are unlikely to affect most people, assuming VB.NET is used with `Option Strict` on). Learning the .NET framework itself is a much bigger issue than learning either of the languages, and it's perfectly possible to become fluent in both - so don't worry too much about which to plump for. There are, however, a few actual differences which may affect your decision:

VB.NET Advantages

- Support for optional parameters - very handy for some COM interoperability
- Support for late binding with `Option Strict` off - type safety at compile time goes out of the window, but legacy libraries which don't have strongly typed interfaces become easier to use.
- Support for named indexers (aka properties with parameters).
- Various legacy VB functions (provided in the `Microsoft.VisualBasic` namespace, and can be used by other languages with a reference to the `Microsoft.VisualBasic.dll`). Many of these can be harmful to performance if used unwisely, however, and many people believe they should be avoided for the most part.
- The `with` construct: it's a matter of debate as to whether this is an advantage or not, but it's certainly a difference.
- Simpler (in expression - perhaps more complicated in understanding) event handling, where a method can declare that it handles an event, rather than the handler having to be set up in code.
- The ability to implement interfaces with methods of different names. (Arguably this makes it harder to find the implementation of an interface, however.)
- `Catch ... When ...` clauses, which allow exceptions to be filtered based on runtime expressions rather than just by type.
- The VB.NET part of Visual Studio .NET compiles your code in the background. While this is considered an advantage for small projects, people creating very large projects have found that the IDE slows down considerably as the project gets larger.

C# Advantages

- XML documentation generated from source code comments. (This is coming in VB.NET with Whidbey (the code name for the next version of Visual Studio and .NET), and there are tools which will do it with existing VB.NET code already.)
- Operator overloading - again, coming to VB.NET in Whidbey.
- Language support for unsigned types (you can use them from VB.NET, but they aren't in the language itself). Again, support for these is coming to VB.NET in Whidbey.
- The `using` statement, which makes unmanaged resource disposal simple.
- Explicit interface implementation, where an interface which is already implemented in a base class can be reimplemented separately in a derived class. Arguably this makes the class

harder to understand, in the same way that member hiding normally does.

- Unsafe code. This allows pointer arithmetic etc, and can improve performance in some situations. However, it is not to be used lightly, as a lot of the normal safety of C# is lost (as the name implies). Note that unsafe code is still managed code, i.e. it is compiled to IL, JITted, and run within the CLR.

Despite the fact that the above list appears to favour VB.NET (if you don't mind waiting for Whidbey), many people prefer C#'s terse syntax enough to make them use C# instead.

}

Practical guide to .Net debugging

I do feel you should start with http://blogs.msdn.com/b/tess/archive/2006/10/16/net-hang-debugging-walkthrough.aspx or with below mentioned labs since this site has multiple step-by-step .Net labs than can guide through different .Net application scenarios. This link was given to me by one of the .Net engineers quite some time back and still is an useful link.

Additional information
==========

http://blogs.msdn.com/b/tess/archive/2008/02/04/net-debugging-demos-information-and-setup-instructions.aspx

-- Above site has below labs

.NET Debugging Demos Lab 1: Hang

.NET Debugging Demos Lab 2: Crash

.NET Debugging Demos Lab 3: Memory

.NET Debugging Demos Lab 4: High CPU hang

.NET Debugging Demos Lab 5: Crash

.NET Debugging Demos Lab 6: Memory Leak

.NET Debugging Demos Lab 7: Memory Leak

Useful websites

Below and many more:

http://msdn.microsoft.com/en-us/

http://blogs.msdn.com/b/csharpfaq/

http://stackoverflow.com/questions/tagged/c%23

http://www.blackwasp.co.uk/Tutorials.aspx

Introduction

The thing I love about Managed (.NET) code is that customers can pretty much do the same thing as us/MS/CSS and don't necessarily need internal symbols for debugging. The Debugging Tools ship with an extension called SOS/PSSCor2 which you can load to look at things like the Managed Call Stack, and different objects. To get the dump, you can go about it different ways. One way would be to use a tool called DebugDiag (available from the Microsoft Download Center) and setup a rule for getting First Chance Exceptions on CLR exceptions. I actually just do a live debug within WinDBG as it is quicker for me to go through that.

.NET ACADEMY

Q & A:

How to debug .NET application?

.

.

.

.

.

.

.

Hope you've tried to answer the question before scrolling.

.

.

.

.

.

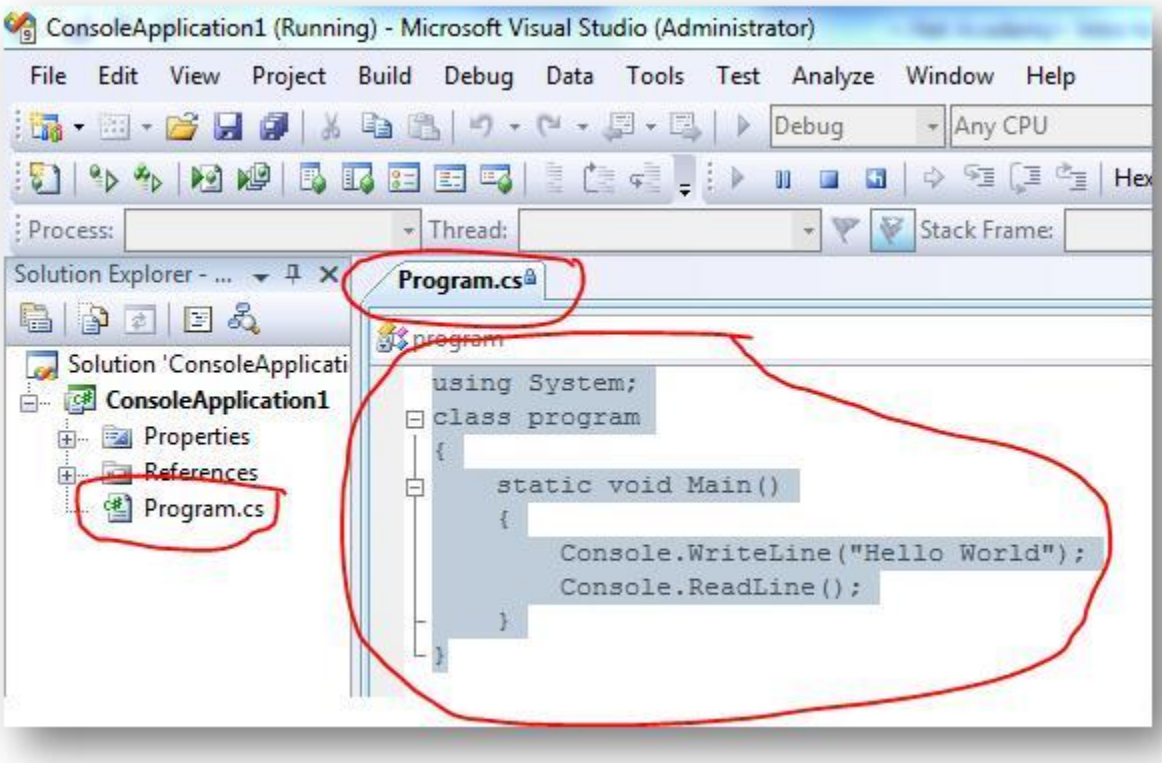
.

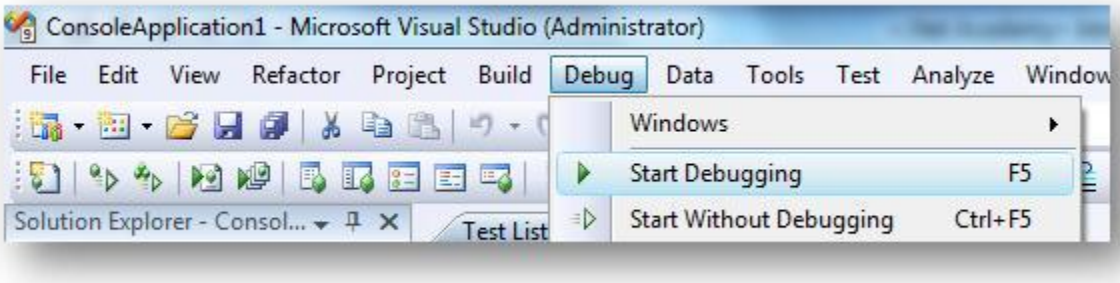
1. First let's write a simple .Net application. Create an **empty** console application.

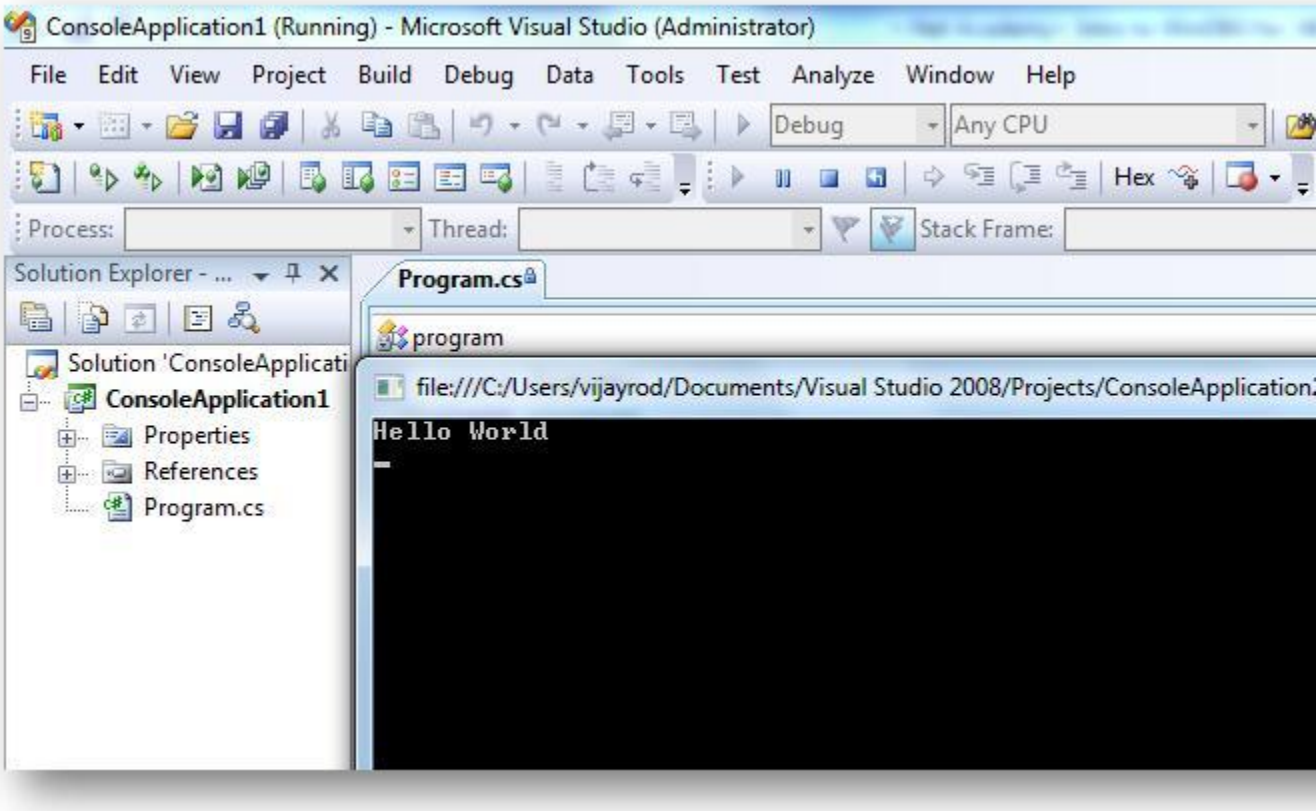
2. Double-click Program.cs in Solution Explorer of Visual Studio. Then paste below **code** in Program.cs:

```
using System;
class program
{
    static void Main()
    {
        Console.WriteLine("Hello World");
        Console.ReadLine();
    }
}
```

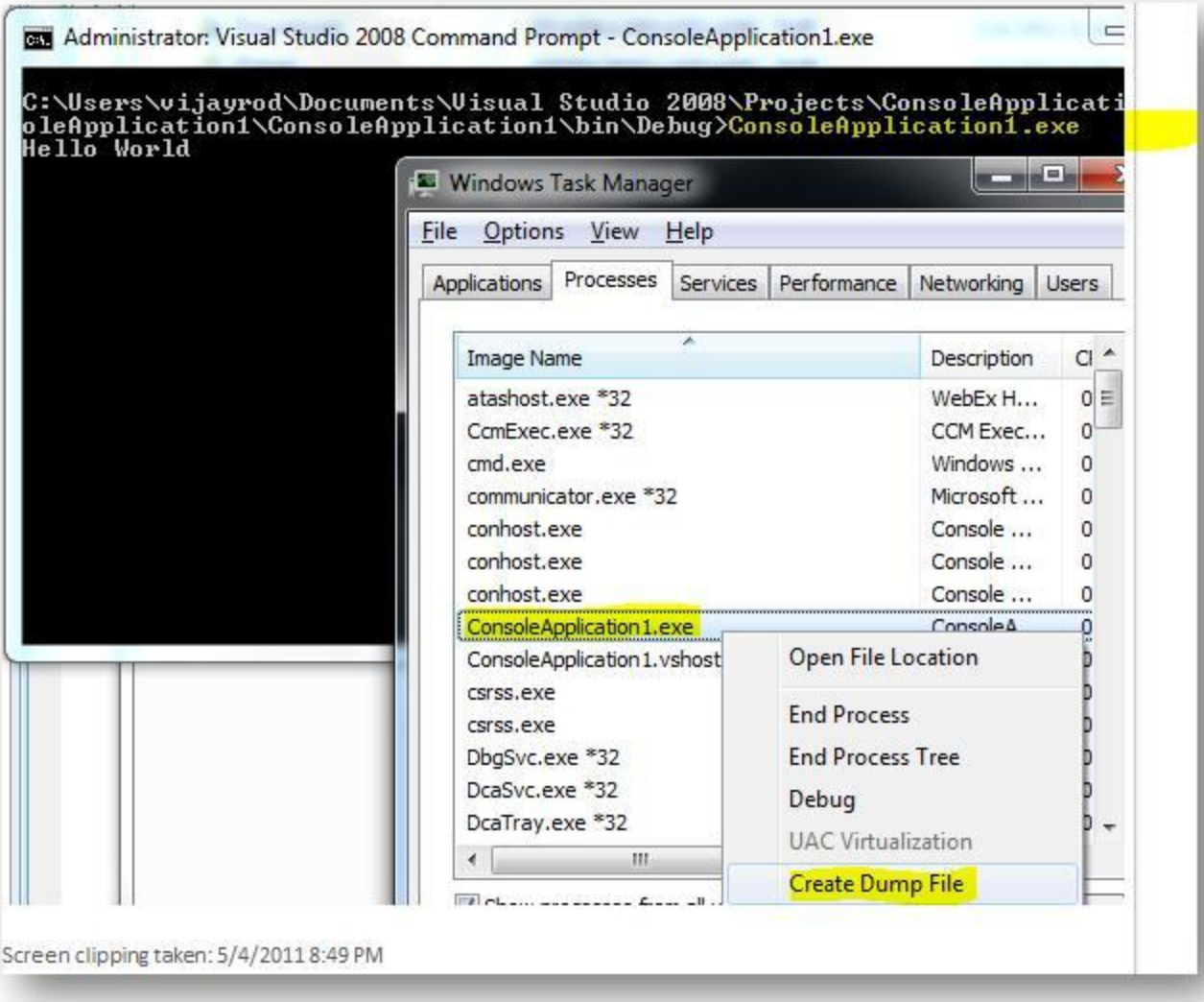
3. Visual Studio should look like below. I'm using Visual Studio 2008 so any other version may show a different layout though it should still have the below mentioned sections.
   Hit **F5** (or Ctrl+F5) or click on 'Start Debugging' as indicated below to execute the program. This will also generate the exe.
   Visual Studio should now display a command prompt saying 'Hello world' as indicated in screenshot further below:

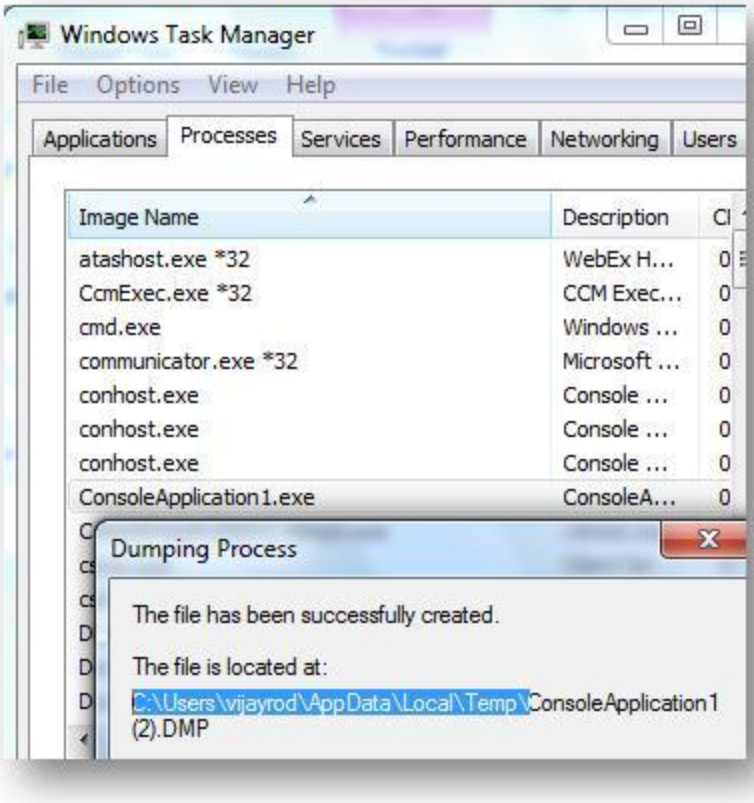






4. Browse to the location of the project using command prompt and run the exe. Then **capture a memory dump** of exe using Task Manager or any of the debugging tools:


Administrator: Visual Studio 2008 Command Prompt - ConsoleApplication1.exe
C:\Users\vijayrod\Documents\Visual Studio 2008\Projects\ConsoleApplicati
oleApplication1\ConsoleApplication1\bin\Debug>ConsoleApplication1.exe
Hello World
Windows Task Manager
File Options View Help
Applications Processes Services Performance Networking Users
Image Name
Description
atashost.exe *32
WebEx H...
CcmExec.exe *32
CCM Exec...
cmd.exe
Windows ...
communicator.exe *32
Microsoft ...
conhost.exe
Console ...
conhost.exe
Console ...
conhost.exe
Console ...
ConsoleApplication1.exe
ConsoleApplication1.vshost
csrss.exe
csrss.exe
DbgSvc.exe *32
DcaSvc.exe *32
DcaTray.exe *32
Open File Location
End Process
End Process Tree
Debug
UAC Virtualization
Create Dump File
Screen clipping taken: 5/4/2011 8:49 PM



5. Open the memory dump in WinDBG. Please refer below if you don't have WinDBG already installed (also need to align symbols with public symbol site which is mentioned in a later step):

http://msdn.microsoft.com/en-us/windows/hardware/gg463009.aspx
Download and Install Debugging Tools for Windows

6. Verify this is **managed**/.Net memory dump. A managed memory dump will have "mscorwks" which is a dll module that belongs to Microsoft .Net Framework:

```
0:000> lm
start            end              module name
00000000`00d70000 00000000`00d78000   ConsoleApplication1 C (private pdb
symbols)  C:\Users\vijayrod\Documents\Visual Studio
2008\Projects\ConsoleApplication2\ConsoleApplication1\ConsoleApplication1\obj\Debug\ConsoleApplicat
ion1.pdb
00000000`73a10000 00000000`73ad9000   msvcr80    (deferred)
00000000`770b0000 00000000`771cf000   kernel32   (private pdb
symbols)  e:\symcache\kernel32.pdb\D5E268B5DD1048A1BFB011C744DD3DFA2\kernel32.pdb
00000000`77530000 00000000`7762a000   user32     (deferred)
00000000`77630000 00000000`777dc000   ntdll      (private pdb
symbols)  e:\symcache\ntdll.pdb\30976E0BFBF745EF860899932F2B791F2\ntdll.pdb
000007fe`f4430000 000007fe`f45b4000   mscorjit   (deferred)
```

000007fe`f7570000 000007fe`f844b000   mscorlib_ni C (private pdb
symbols) e:\symcache\mscorlib.pdb\AC8693DFA4544A0D9413669A012D34061\mscorlib.pdb
000007fe`f8490000 000007fe`f8e3e000   mscorwks   (private pdb
symbols) e:\symcache\mscorwks.pdb\4EA1577D2DFE48FFBC7C2F7EA30A1DD21\mscorwks.pdb
000007fe`f8e40000 000007fe`f8ed0000   mscoreei   (private pdb
symbols) e:\symcache\mscoreei.pdb\29F342C6D83D4ADC95275E72E86B830D2\mscoreei.pdb
…

7. **Load .Net debugging extension** sos.dll (or can load Psscor2.dll) with the appropriate command mentioned in below link. If you're using VS 2010, then you have to **load sos.dll and have fun!!**

Microsoft (R) Windows Debugger Version 6.13.0007.1090 AMD64
Copyright (c) Microsoft Corporation. All rights reserved.

Loading Dump File [C:\Users\vijayrod\AppData\Local\Temp\ConsoleApplication1.DMP]
User Mini Dump File with Full Memory: Only application data is available

Executable search path is:
Windows 7 Version 7600 MP (2 procs) Free x64
Product: WinNt, suite: SingleUserTS
Machine Name:
Debug session time: Wed May 4 20:52:29.000 2011 (UTC + 5:30)
System Uptime: 1 days 1:37:26.550
Process Uptime: 0 days 0:05:32.000
.....
ntdll!ZwRequestWaitReplyPort+0xa:
00000000`7767f8da c3   ret

0:000> .sympath srv*http://msdl.microsoft.com/download/symbols
Symbol search path is: srv*http://msdl.microsoft.com/download/symbols
Expanded Symbol search path is: srv*http://msdl.microsoft.com/download/symbols

0:000> !sym noisy
noisy mode - symbol prompts on
0:000> .reload /f

0:000> .loadby sos mscorwks
0:000> kpn
Child-SP   RetAddr   Call Site
00 00000000`002de678 00000000`770d2b08 ntdll!NtRequestWaitReplyPort+0xa
01 00000000`002de680 00000000`77105601 kernel32!ConsoleClientCallServer+0x54
02 00000000`002de6b0 00000000`7711a922 kernel32!ReadConsoleInternal+0x1f1
03 00000000`002de800 00000000`770e9934 kernel32!ReadConsoleA+0xb2
04 00000000`002de8e0 000007fe`f875ccc7 kernel32!zzz_AsmCodeRange_End+0x8bea
05 00000000`002de920 000007fe`f7fd0fd3 mscorwks!DoNDirectCall__PatchGetThreadCall+0x7b
06 00000000`002de9d0 000007fe`f7fd10e9 mscorlib_ni!DomainNeutralILStubClass.IL_STUB(<HRESULT
0x80004001>)+0x93
07 00000000`002deaf0 000007fe`f7fd1202
mscorlib_ni!System.IO.__ConsoleStream.ReadFileNative(<HRESULT 0x80004001>)+0xa9

```
08 00000000`002deb50 000007fe`f78c065a mscorlib_ni!System.IO.__ConsoleStream.Read(<HRESULT
0x80004001>)+0x62
09 00000000`002debb0 000007fe`f78e28ca mscorlib_ni!System.IO.StreamReader.ReadBuffer(<HRESULT
0x80004001>)+0x5a
0a 00000000`002dec00 000007fe`f7fd435f mscorlib_ni!System.IO.StreamReader.ReadLine(<HRESULT
0x80004001>)+0x4a
0b 00000000`002dec50 000007ff`00160153
mscorlib_ni!System.IO.TextReader+SyncTextReader.ReadLine(<HRESULT 0x80004001>)+0x3f
0c 00000000`002decb0 000007fe`f875d4a2 ConsoleApplication1!program.Main(<HRESULT
0x80004001>)+0x33 [C:\Users\vijayrod\Documents\Visual Studio
2008\Projects\ConsoleApplication2\ConsoleApplication1\ConsoleApplication1\Program.cs @ 7]
0d 00000000`002decf0 000007fe`f8636ef3 mscorwks!CallDescrWorker+0x82
0e 00000000`002ded30 000007fe`f8626d1f mscorwks!CallDescrWorkerWithHandler+0xd3
0f 00000000`002dedd0 000007fe`f86b35b7 mscorwks!MethodDesc::CallDescr+0x24f
10 00000000`002df020 000007fe`f86cf358 mscorwks!ClassLoader::RunMain+0x22b
11 00000000`002df280 000007fe`f8697835 mscorwks!Assembly::ExecuteMainMethod+0xbc
12 00000000`002df570 000007fe`f858349f mscorwks!SystemDomain::ExecuteMainMethod+0x491
13 00000000`002dfb40 000007fe`f86c8ab0 mscorwks!ExecuteEXE+0x47
14 00000000`002dfb90 000007fe`f8e43309 mscorwks!CorExeMain+0xac
15 00000000`002dfbf0 000007fe`f8ed5b21 mscoreei!CorExeMain+0x41
16 00000000`002dfc20 00000000`770cf56d mscoree!CorExeMain_Exported+0x57
17 00000000`002dfc50 00000000`77662cc1 kernel32!BaseThreadInitThunk+0xd
18 00000000`002dfc80 00000000`00000000 ntdll!RtlUserThreadStart+0x1d

0:000> !clrstack
OS Thread Id: 0xfc8 (0)
Child-SP         RetAddr          Call Site
00000000002de9d0 000007fef7fd10e9
DomainNeutralILStubClass.IL_STUB(Microsoft.Win32.SafeHandles.SafeFileHandle, Byte*, Int32, Int32 ByRef,
IntPtr)
00000000002deaf0 000007fef7fd1202
System.IO.__ConsoleStream.ReadFileNative(Microsoft.Win32.SafeHandles.SafeFileHandle, Byte[], Int32, Int32,
Int32, Int32 ByRef)
00000000002deb50 000007fef78c065a System.IO.__ConsoleStream.Read(Byte[], Int32, Int32)
00000000002debb0 000007fef78e28ca System.IO.StreamReader.ReadBuffer()
00000000002dec00 000007fef7fd435f System.IO.StreamReader.ReadLine()
00000000002dec50 000007ff00160153 System.IO.TextReader+SyncTextReader.ReadLine()
00000000002decb0 000007fef875d4a2 program.Main()
```

8. The above managed stack (with !clrstack) clearly indicates that the application is waiting on a Read.

For the more curious:

- Below "!threads" shows .NET/managed stacks (equivalent to "~" for normal exe):

```
0:000> !threads
ThreadCount: 2
UnstartedThread: 0
BackgroundThread: 1
PendingThread: 0
DeadThread: 0
```

Hosted Runtime: no

				PreEmptive			Lock			
	ID	OSID	ThreadOBJ	State	GC	GC Alloc Context		Domain	Count	APT Exception
0	1	fc8	000000000051dcd0	a020	Enabled	0000000002660670:0000000002661fd0		0000000000514ce0	1	MTA
2	2	1e44	0000000000525cc0	b220	Enabled	0000000000000000:0000000000000000		0000000000514ce0	0	MTA (Finalizer)

- Below "~" shows 3 threads:

```
0:002> ~
#  0  Id: 19dc.fc8 Suspend: 0 Teb: 000007ff`fffde000 Unfrozen
   1  Id: 19dc.143c Suspend: 0 Teb: 000007ff`fffdc000 Unfrozen
.  2  Id: 19dc.1e44 Suspend: 0 Teb: 000007ff`fffd8000 Unfrozen
```

- Below shows the 3 threads – one is the application thread, second is the debugger thread (for capturing memory dump), third is the .NET Garbage collector:

```
0:002> ~*kL

#  0  Id: 19dc.fc8 Suspend: 0 Teb: 000007ff`fffde000 Unfrozen
Child-SP          RetAddr           Call Site
00000000`002de678 00000000`770d2b08 ntdll!NtRequestWaitReplyPort+0xa
00000000`002de680 00000000`77105601 kernel32!ConsoleClientCallServer+0x54
00000000`002de6b0 00000000`7711a922 kernel32!ReadConsoleInternal+0x1f1
00000000`002de800 00000000`770e9934 kernel32!ReadConsoleA+0xb2
00000000`002de8e0 000007fe`f875ccc7 kernel32!zzz_AsmCodeRange_End+0x8bea
00000000`002de920 000007fe`f7fd0fd3 mscorwks!DoNDirectCall__PatchGetThreadCall+0x7b
00000000`002de9d0 000007fe`f7fd10e9
mscorlib_ni!DomainNeutralILStubClass.IL_STUB(Microsoft.Win32.SafeHandles.SafeFileHandle, Byte*, Int32, Int32 ByRef,
IntPtr)+0x93
00000000`002deaf0 000007fe`f7fd1202
mscorlib_ni!System.IO.__ConsoleStream.ReadFileNative(Microsoft.Win32.SafeHandles.SafeFileHandle, Byte[], Int32, Int32,
Int32, Int32 ByRef)+0xa9
00000000`002deb50 000007fe`f78c065a mscorlib_ni!System.IO.__ConsoleStream.Read(Byte[], Int32, Int32)+0x62
00000000`002debb0 000007fe`f78e28ca mscorlib_ni!System.IO.StreamReader.ReadBuffer()+0x5a
00000000`002dec00 000007fe`f7fd435f mscorlib_ni!System.IO.StreamReader.ReadLine()+0x4a
00000000`002dec50 000007ff`00160153 mscorlib_ni!System.IO.TextReader+SyncTextReader.ReadLine()+0x3f
00000000`002decb0 000007fe`f875d4a2 ConsoleApplication1!program.Main()+0x33
00000000`002decf0 000007fe`f8636ef3 mscorwks!CallDescrWorker+0x82
00000000`002ded30 000007fe`f8626d1f mscorwks!CallDescrWorkerWithHandler+0xd3
00000000`002dedd0 000007fe`f86b35b7 mscorwks!MethodDesc::CallDescr+0x24f
00000000`002df020 000007fe`f86cf358 mscorwks!ClassLoader::RunMain+0x22b
00000000`002df280 000007fe`f8697835 mscorwks!Assembly::ExecuteMainMethod+0xbc
00000000`002df570 000007fe`f858349f mscorwks!SystemDomain::ExecuteMainMethod+0x491
00000000`002dfb40 000007fe`f86c8ab0 mscorwks!ExecuteEXE+0x47
00000000`002dfb90 000007fe`f8e43309 mscorwks!CorExeMain+0xac
00000000`002dfbf0 000007fe`f8ed5b21 mscoreei!CorExeMain+0x41
00000000`002dfc20 00000000`770cf56d mscoree!CorExeMain_Exported+0x57
00000000`002dfc50 00000000`77662cc1 kernel32!BaseThreadInitThunk+0xd
00000000`002dfc80 00000000`00000000 ntdll!RtlUserThreadStart+0x1d

   1  Id: 19dc.143c Suspend: 0 Teb: 000007ff`fffdc000 Unfrozen
Child-SP          RetAddr           Call Site
00000000`00cbf7a8 000007fe`fdd213a6 ntdll!ZwWaitForMultipleObjects+0xa
```

00000000`00cbf7b0 00000000`770bf190 KERNELBASE!WaitForMultipleObjectsEx+0xe8
00000000`00cbf8b0 000007fe`f86eb525 kernel32!WaitForMultipleObjects+0xb0
00000000`00cbf940 000007fe`f86ce331 mscorwks!DebuggerRCThread::MainLoop+0xbd
00000000`00cbf9f0 000007fe`f8585caa mscorwks!DebuggerRCThread::ThreadProc+0xf9
00000000`00cbfa40 00000000`770cf56d mscorwks!DebuggerRCThread::ThreadProcStatic+0x56
00000000`00cbfa90 00000000`77662cc1 kernel32!BaseThreadInitThunk+0xd
00000000`00cbfac0 00000000`00000000 ntdll!RtlUserThreadStart+0x1d

2 Id: 19dc.1e44 Suspend: 0 Teb: 000007ff`fffd8000 Unfrozen
Child-SP RetAddr Call Site
00000000`011ffa78 000007fe`fdd213a6 ntdll!ZwWaitForMultipleObjects+0xa
00000000`011ffa80 00000000`770bf190 KERNELBASE!WaitForMultipleObjectsEx+0xe8
00000000`011ffb80 000007fe`f86df0f3 kernel32!WaitForMultipleObjects+0xb0
00000000`011ffc10 000007fe`f86c922a mscorwks!WKS::WaitForFinalizerEvent+0x93
00000000`011ffc40 000007fe`f86b4ffc mscorwks!WKS::GCHeap::FinalizerThreadWorker+0x4a
00000000`011ffc80 000007fe`f86785ad mscorwks!MethodTableBuilder::AllocateFromHighFrequencyHeap+0x70
00000000`011ffcd0 000007fe`f86c0f2d mscorwks!List<CCodebaseEntry * __ptr64>::AddTail+0x11d
00000000`011ffda0 000007fe`f8574d1e mscorwks!ZapStubPrecode::GetType+0x39
00000000`011ffde0 000007fe`f86cf984 mscorwks!ManagedThreadBase_NoADTransition+0x42
00000000`011ffe40 000007fe`f8580718 mscorwks!WKS::GCHeap::FinalizerThreadStart+0x74
00000000`011ffe80 00000000`770cf56d mscorwks!Thread::intermediateThreadProc+0x78
00000000`011fff50 00000000`77662cc1 kernel32!BaseThreadInitThunk+0xd
00000000`011fff80 00000000`00000000 ntdll!RtlUserThreadStart+0x1d

Intro to WinDBG for .NET Developers - part II

We may see a shift of DBA role from operational database administration to database 'application' administration, due to the emergence of the cloud/Azure. Every developer or service engineer debugs applications at some stage in his or her career, yet debugging is often viewed as an arcane and difficult topic. While it is often difficult to determine why an application is hanging, leaking memory, or crashing, there are techniques you can apply to make the debugging process more productive and efficient.

.NET ACADEMY

Q & A:
Let's continue on the managed/.NET memory dump obtained in previous email.

Why symbols in debugging?

A debug symbol is information that expresses which programming-language constructs generated a specific piece of machine code in a given executable module. This information enables a person using a symbolic debugger to gain additional information about the binary, such as the names of variables and routines from the original source code. This information can be extremely helpful while trying to investigate and fix a crashing application or any other fault.

If debug symbols are embedded in the binary (exe) itself, the file can then grow significantly larger (sometimes by several megabytes). To avoid this extra size, modern compilers and early mainframe debugging systems output the symbolic information into a separate file; for Microsoft compilers, this file is called a PDB file. Microsoft has special online servers from which it's possible to download the debug symbols separately. Microsoft's WinDBG debugger can be configured to automatically download debug symbols for Windows DLLs on demand as they are needed (using .sympath). The PDB debug symbols that Microsoft distributes are only partial (they include only public functions, global variables and their data types.

How to load symbols?

.sympath
The .sympath command changes the default path of the host debugger for (pdb) symbol search.
The debugger's default behavior is to use lazy symbol loading (also known as deferred symbol loading). This kind of loading means that symbols are not loaded until they are required. However, if the symbol path is changed, all already-loaded symbols are immediately reloaded.
You can also force symbol loading by using the .reload (Reload Module) command with the /f option.
The local path is automatically created if it does not exist.
The .loadby command loads a new extension DLL into the debugger (.loadby DLLName ModuleName).

0:000> **.sympath** srv*c:\mysymbols*http://msdl.microsoft.com/download/symbols
Symbol search path is: srv*c:\mysymbols*http://msdl.microsoft.com/download/symbols
Expanded Symbol search path is: srv*c:\mysymbols*http://msdl.microsoft.com/download/symbols

-- <optional>List loaded modules
0:000> lm
start end module name
77c90000 77dcd000 ntdll (pdb symbols)
c:\mysymbols\ntdll.pdb\2D612400BDA64ECCA13827B91848B8FB2\ntdll.pdb

-- <optional>
Since there are several DLLs of CLR referenced by the SOS extension, you should load the SOS.dll by using its originally full path (load command with dll path).
In addition, .loadby locates a DLL by looking in the same directory as where another DLL was loaded from. So, ".loadby sos mscorwks" tells WinDbg to look up which directory mscorwks.dll was loaded from and load sos.dll from there. If you're running an unmanaged application (i. e., a non .NET application) then mscorwks.dll won't be loaded and you get the message "Unable to find module 'mscorwks'". You also get this message if you break into a managed application very early, before the .NET CLR is loaded.
-- Load SOS for debugging framework 1.0 / 1.1
0:000> .load clr10\sos
-- Load SOS for debugging framework 2.0 to 3.5
0:000> .loadby sos mscorwks
-- Load SOS for debugging framework 4.0
0:000> .loadby sos clr

-- <optional> can run below to load symbols for all modules to save time on symbol download later during debugging
0:000> !sym noisy
noisy mode - symbol prompts on
0:000> .reload /f

How to check if the memory dump is managed/.NET?

Lm will display 'mscorwks' (as one of the modules) for .Net 2.0 to 3.5 (will display 'clr' for .Net 4.0)

0:000> **lm**
start end module name
65f80000 66511000 mscorwks

How to check if managed symbols are loaded?

Execute any .Net command, like !threads (will not display any data if managed symbols are not loaded or may give an error). For example below shows 2 rows indicating the number of managed threads in this memory dump:

-- Good run
0:000> !threads
ThreadCount: 2
UnstartedThread: 0
BackgroundThread: 1
PendingThread: 0
DeadThread: 0
Hosted Runtime: no
PreEmptive GC Alloc Lock
ID OSID ThreadOBJ State GC Context Domain Count APT Exception
0 1 b94 0045d5f0 a020 Enabled 01d14638:01d15fe8 00459250 1 MTA
2 2 798 0046a1c8 b220 Enabled 00000000:00000000 00459250 0 MTA (Finalizer)

-- Bad run
0:000> !threads
Failed to find runtime DLL (mscorwks.dll),
Unable to find module 'mscorwks'

What to do if managed symbols are not loaded?

My preferred command (compared to .cordll) is .chain command which lists All loaded debugger extensions in their default search order.
Alternatively, the .cordll command controls managed code debugging and the Microsoft .NET common language runtime (CLR).

-- Below may could be false since it can only mean that a managed command was not issued so there was no need for any load attempt
0:000> .cordll
CLR DLL status: No load attempts

-- Turns on verbose mode for CLR module loading. Unloads and Loads the CLR debugging modules.
0:000> .cordll -ve -u -l
CLRDLL: Unknown processor type in image
C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727\mscorwks.dll
CLR DLL status: No load attempts

-- Below to get Image path for mscorwks (for .Net 4.0, Image path for clr)
0:000> lmv m mscorwks
start end module name
65f80000 66511000 mscorwks (deferred)
Image path: C:\Windows\Microsoft.NET\Framework\v2.0.50727\mscorwks.dll
0:000> **.exepath**+ C:\Windows\Microsoft.NET\Framework\v2.0.50727\
0:000> !threads
...

-- Good run
0:000> **.cordll**
CLR DLL status: **Loaded** DLL C:\Windows\Microsoft.NET\Framework\v2.0.50727\mscordacwks.dll

-- .chain command lists All loaded debugger extensions in their default search order.
-- Good run
0:000> **.chain**
Extension DLL search Path: ...
Extension DLL chain:
C:\Windows\Microsoft.NET\Framework\v2.0.50727\sos: image 2.0.50727.4959, API 1.0.0, built Sat Feb 05 09:38:51 2011
[path: C:\Windows\Microsoft.NET**Framework**\v2.0.50727\sos.dll]
dbghelp: image 6.11.0001.404, API 6.1.6, built Thu Feb 26 07:25:30 2009
[path: C:\Program Files\Debugging Tools for Windows (x86)\dbghelp.dll]
ext: image 6.11.0001.404, API 1.0.0, built Thu Feb 26 07:25:30 2009
[path: C:\Program Files\Debugging Tools for Windows (x86)\winext\ext.dll]
exts: image 6.11.0001.404, API 1.0.0, built Thu Feb 26 07:25:24 2009
[path: C:\Program Files\Debugging Tools for Windows (x86)\WINXP\exts.dll]
uext: image 6.11.0001.404, API 1.0.0, built Thu Feb 26 07:25:26 2009
[path: C:\Program Files\Debugging Tools for Windows (x86)\winext\uext.dll]

ntsdexts: image 6.1.7015.0, API 1.0.0, built Thu Feb 26 07:24:43 2009
[path: C:\Program Files\Debugging Tools for Windows (x86)\WINXP\ntsdexts.dll]

Intro to WinDBG for .NET Developers - part III (Examining code and stacks)

Debugging is a methodical process of finding and reducing the number of bugs, or defects, in a computer program or a piece of electronic hardware, thus making it behave as expected. Debugging tends to be harder when various subsystems are tightly coupled, as changes in one may cause bugs to emerge in another.

.NET ACADEMY

Q & A:
Let's continue on the managed/.NET memory dump obtained in previous email.

How exactly is a memory dump captured (using WinDbg or using MS SQL Dumper or Debug Diag or Task Manager or ProcDump or any other dump capture tool)?

Any time you debug a running application, the debug API inject's a remote thread into the target application and calls KERNEL32!DebugBreak, which is an int3 (0xcc) instruction on x86 processors. DebugBreak function causes a breakpoint exception to occur in the current process. This allows the calling thread to signal the debugger to handle the exception.
At this point the process brakes and stops executing, however, you now have at least one additional debug thread running.

Is there a quick list of managed/.Net/SOS commands for debugging?

http://geekswithblogs.net/.NETonMyMind/archive/2006/03/14/72262.aspx
WinDbg / SOS Cheat Sheet

Is there a quick list of native commands for debugging?

http://windbg.info/doc/1-common-cmds.html
Common WinDbg Commands (Thematically Grouped)

What is SOS.dll (SOS Debugging Extension)?

SOS is a debugger extension DLL designed to aid in the debugging of managed/.Net programs, by providing information about the internal common language runtime (CLR) environment.
"DLL" (Dynamic Link library), mentioned above, is Microsoft's implementation of the shared library concept in the Microsoft operating systems. These libraries usually have the file extension DLL, OCX (for libraries containing ActiveX controls), or DRV (for legacy system drivers).

How to quickly know the commands available for managed debugging?

The !help extension (in WinDBG) displays help text that describes the extension commands exported from the extension DLL.

0:000> **!help**

SOS is a debugger extension DLL designed to aid in the debugging of managed

programs. Functions are listed by category, then roughly in order of
importance. Shortcut names for popular functions are listed in parenthesis.
Type "!help <functionname>" for detailed info on that function.

```
Object Inspection               Examining code and stacks
------------------------------  ------------------------------
DumpObj (do)                    Threads
DumpArray (da)                  CLRStack
DumpStackObjects (dso)          IP2MD
DumpHeap                        U
DumpVC                          DumpStack
GCRoot                          EEStack
ObjSize                         GCInfo
FinalizeQueue                   EHInfo
PrintException (pe)             COMState
TraverseHeap                    BPMD

Examining CLR data structures   Diagnostic Utilities
------------------------------  ------------------------------
DumpDomain                      VerifyHeap
EEHeap                          DumpLog
Name2EE                         FindAppDomain
SyncBlk                         SaveModule
DumpMT                          GCHandles
DumpClass                       GCHandleLeaks
DumpMD                          VMMap
Token2EE                        VMStat
EEVersion                       ProcInfo
DumpModule                      StopOnException (soe)
ThreadPool                      MinidumpMode
DumpAssembly
DumpMethodSig                   Other
DumpRuntimeTypes                ------------------------------
DumpSig                         FAQ
RCWCleanupList
DumpIL
```

How to get detailed info for a specific function?

Type "!help <functionname>" for detailed info on that function. For example, either of below commands for help on !threads:

```
0:000> !help !threads

0:000> !help threads
```

Is there any other useful command to know more about sos?

Another useful command is !help faq:

```
0:000> !help faq
--------------------------------------------------------------------------------
>> Where can I get the right version of SOS for my build?
```

If you are running version 1.1 or 2.0 of the CLR, SOS.DLL is installed in the same directory as the main CLR dll (MSCORWKS.DLL). Newer versions of the Windows Debugger provide a command to make it easy to load the right copy of SOS.DLL:

".loadby sos mscorwks"

That will load the SOS extension DLL from the same place that MSCORWKS.DLL is loaded in the process. You shouldn't attempt to use a version of SOS.DLL that doesn't match the version of MSCORWKS.DLL. You can find the version of MSCORWKS.DLL by running

"lm v m mscorwks"

in the debugger.

If you are using a dump file created on another machine, it is a little bit more complex. You need to make sure the mscordacwks.dll file that came with that install is on your symbol path, and you need to load the corresponding version of sos.dll (typing .load <full path to sos.dll> rather than using the .loadby shortcut). Within the Microsoft corpnet, we keep tagged versions of mscordacwks.dll, with names like mscordacwks_<architecture>_<version>.dll that the Windows Debugger can load. If you have the correct symbol path to the binaries for that version of the Runtime, the Windows Debugger will load the correct mscordacwks.dll file.

>> I have a chicken and egg problem. I want to use SOS commands, but the CLR isn't loaded yet. What can I do?

In the debugger at startup you can type:

"sxe clrn"

Let the program run, and it will stop with the notice

"CLR notification: module 'mscorlib' loaded"

At this time you can use SOS commands. To turn off spurious notifications, type:

"sxd clrn"

>> I got the following error message. Now what?

```
0:000> .loadby sos mscorwks
0:000> !DumpStackObjects
Failed to find runtime DLL (mscorwks.dll), 0x80004005
Extension commands need mscorwks.dll in order to have something to do.
0:000>
```

This means that the CLR is not loaded yet, or has been unloaded. You need to wait until your managed program is running in order to use these commands. If you have just started the program a good way to do this is to type

bp mscorwks!EEStartup "g @$ra"

in the debugger, and let it run. After the function EEStartup is finished,
there will be a minimal managed environment for executing SOS commands.

>> I have a partial memory minidump, and !DumpObj doesn't work. Why?

In order to run SOS commands, many CLR data structures need to be traversed.
When creating a minidump without full memory, special functions are called at
dump creation time to bring those structures into the minidump, and allow a
minimum set of SOS debugging commands to work. At this time, those commands
that can provide full or partial output are:

CLRStack
Threads
Help
PrintException
EEVersion

For a minidump created with this minimal set of functionality in mind, you
will get an error message when running any other commands. A full memory dump
(obtained with ".dump /ma <filename>" in the Windows Debugger) is often the
best way to debug a managed program at this level.

Additional information (for .Net 4.0)
==========

http://msdn.microsoft.com/en-us/library/bb190764(v=VS.100).aspx
SOS.dll (SOS Debugging Extension)

http://msdn.microsoft.com/en-us/library/d21c150d(v=VS.100).aspx
Diagnosing Errors with Managed Debugging Assistants
"Turn on Managed Debugging Assistants. These enable additional runtime diagnostics,
particularly in the area of PInvoke/Interop."

What are threads?

In computer science, a thread of execution is the smallest unit of processing that can be scheduled by an operating system.

What is the difference between threads and processes?

The implementation of threads and processes differs from one operating system to another, but in most cases, a thread is contained inside a process.
In computing, a process is an instance of a computer program that is being executed. It contains the program code and its current activity. Depending on the operating system (OS), a process may be made up of multiple threads of execution that execute instructions concurrently.
Multiple threads can exist within the same process and share resources such as memory, while different processes do not share these resources. In particular, the threads of a process share the latter's instructions (its code) and its context (the values that its variables reference at any given moment).
To give an analogy, multiple threads in a process are like multiple cooks reading off the same cook book and following its instructions, not necessarily from the same page.

What is the equivalent managed/.Net command for dumping all threads (~ for native code during user-mode debugging)?

!threads
There is a difference:

- !Threads lists all the "managed"/.Net threads in the process
- ~ lists "All" threads in the process during user-mode debugging.

Why does the number of threads reported by WinDbg (!threads, ~) and Task Manager differ?

Task manager (Processes tab -> Threads column) reports the total number of threads for your process while !threads reports the number of managed threads. If you use the ~ command in WinDbg you will see all thread threads for the process (equal to number displayed thro Task Manager, or PerfMon using the Process\ThreadCount counter).

What are the different types of managed/.Net threads?

A managed thread is either a background thread or a foreground thread. Background threads are identical to foreground threads with one exception: a background thread does not keep the managed execution environment running. Once all foreground threads have been stopped in a managed process (where the .exe file is a managed assembly), the system stops all background threads and shuts down.

What does the output of !threads indicate?

!threads command will show us all .NET threads in the current process.

Debugger shorthand ID: The first column (without a column header) is the debugger shorthand ID. In fiber mode, if the thread is a fiber which is switched out by the host, this column will be shown as "<<<<"

The CLR Thread ID: This is the second column with header as "ID".

The OS thread ID: This is the column with header as "OSID".

The 'Debugger shorthand ID' and the OSID can also be obtained using "~".

The threads listed with XXXX for id are threads that have terminated but the corresponding thread objects have not yet been collected.
The finalizer thread is marked at the end of the line by the text (finalizer) (surprisingly enough). At the beginning of the line that has the (finalizer) text at the end of it you will see the thread's index. We will need it to run the next command.

A lot more details on !threads is given below from
http://blogs.msdn.com/b/yunjin/archive/2005/08/30/457756.aspx (Thread, System.Threading.Thread, and !Threads (III))
{

First *!Threads* gives some statistics about *Thread Store*.
ThreadCount: number of total *C++ Thread* objects in *Thread Store*.

UnstartedThread: number of *C++ Thread* objects marked as unstarted. If a user creates a *C# Thread* object, CLR will create an "unstarted" *C++ Thread* object. When *Thread.Start* is called on the C# object, CLR will create an *OS thread* and remove "unstarted" flag from the *C++ Thread* object.
BackgroundThread: number of *C++ Threads* (and the corresponding *OS threads*) considered as background. Being background simply means CLR won't wait the thread for shutting down. Threads created explicitly by using *System.Threading.Thread.Start* are by default foreground threads; whereas threads wandering into CLR from unmanaged world are by default background threads (*SetupThread* in *vm/threads.cpp* calls *SetBackground(TRUE)*). However, whether a thread is background could be changed by using *IsBackground* property in *C# Thread* object.
PendingThreads: If an *OS thread* is created but its *ThreadProc* hasn't be executed to the place to decrement unstarted counter in *Thread Store*, the thread is considered to be pending. Number of this type of threads should be quite low.
DeadThreads: Number of *C++ Thread* objects whose *OS threads* are already dead but the *C++ objects* themselves are not deleted yet.
In Rotor, all the five numbers are actually stored in *ThreadStore (vm/threads.h)* object as its fields.

Then it comes a table of all *C++ Thread* objects in Thread Store. Let me explain each field.
The first column doesn't have a header. It is the *OS thread* ID given by debugger just for debugging readability. Because the numbers only exist in debugger process, not the debuggee process, you may see the number being different when you look at a live session than when you debug a dump taken from the same live session. For a "dead" or "unstarted" thread, this column is "XXX".
ID: this is the thread ID assigned by OS, it remains consistent during debugger sessions, but OS could recycle it.
ThreadOBJ: address of *C++ Thread* object. You could see contents of the object by *"dt mscorwks!Thread <address>"* if you have symbols for mscorwks.dll.
State: one of the most important fields of the table. For Rotor, it is the *C++ Thread's m_State* field. It is combination of bit masks to indicate what the status the *Thread* currently is. All possible states (bit masks) are defined as enum *ThreadState* in *vm/Threads*. We already covered several states like *TS_Background*, *TS_Unstarted*, and *TS_Dead*. More states include *TS_AbortRequested* (this thread is requested to be aborted), *TS_AbortInitiated* (abort process is already started for this thread), *TS_GCSuspendPending* (GC is trying to suspend this thread), and etc.
Preemptive GC: also very important. In Rotor, this is *m_fPreemptiveGCDisabled* field of *C++ Thread* class. It indicates what GC mode the thread is in: "enabled" in the table means the thread is in *preemptive mode* where GC could preempt this thread at any time; "disabled" means the thread is in cooperative mode where GC has to wait the thread to give up its current work (the work is related to GC objects so it can't allow GC to move the objects around). When the thread is executing managed code (the current IP is in managed code), it is always in cooperative mode; when the thread is in *Execution Engine* (unmanaged code), *EE* code could choose to stay in either mode and could switch mode at any time; when a thread are outside of CLR (e.g, calling into native code using interop), it is always in preemptive mode.
GC Alloc context: allocate context GC might use when it tries to allocate object for this thread. In Rotor, it is *m_alloc_context* in *C++ Thread* object.

Domain: which *AppDomain* the thread is currently in (Rotor: *m_pDomain* field of *C++ Thread* class). You could use *!DumpDomain* or "dt mscorwks!AppDomain" to dump details of the domain. A thread can only be in one domain at a time, but it could switch into different domains. Speical marks will be put on thread's stack to when it transit to another domain.
Lock count: how many locks this thread has taken (Rotor: *m_dwLockCount* field of *C++ Thread* class). The locks it tracks include the managed monitors (taken by *lock(obj)* in C#), BCL's *ReaderWriterLock*, and certain locks inside CLR's unmanaged code.
APT: *COM* apartment for the thread, whether the thread is in a single-threaded apartment (*STA*), multithreaded apartment(*MTA*) or unknown.
Exception: the last managed exception thrown from this thread. It is saved in a GC handle in the *C++ Thread* object (Rotor: *m_LastThrownObjectHandle*).
The last column also indicates which special thread this thread is. However, *!Threads* only recognize a limited type of special threads for this field, including *Finalizer thread*, *GC thread*, *Threadpool Worker thread*, and *Threadpool Completion Port thread*. And for special threads which doesn't have a *C++ Thread* object (a special thread doesn't need to run managed code like debugger helper thread and server GC thread), they can not be displayed here. In Whidbey, a "*-special*" option is added to *!Threads* command which will show all special threads in the process as a separate list.

}

```
0:000> !threads
ThreadCount: 2
UnstartedThread: 0
BackgroundThread: 1
PendingThread: 0
DeadThread: 0
Hosted Runtime: no
                      PreEmptive   GC Alloc         Lock
    ID OSID ThreadOBJ   State    GC      Context      Domain   Count APT Exception
  0   1  b94 0045d5f0    a020 Enabled  01d14638:01d15fe8 00459250     1 MTA
  2   2  798 0046a1c8    b220 Enabled  00000000:00000000 00459250     0 MTA (Finalizer)

--note the 'Debugger shorthand ID' and the OSID in below which is same as in
output of !threads (e.g. first line says "0" and "b94"
0:000> ~
.  0  Id: b0.b94 Suspend: 0 Teb: 7ffdf000 Unfrozen
   1  Id: b0.444 Suspend: 0 Teb: 7ffde000 Unfrozen
   2  Id: b0.798 Suspend: 0 Teb: 7ffdd000 Unfrozen
```

How to view "special" threads?

```
0:000> !threads -special
ThreadCount: 2
UnstartedThread: 0
BackgroundThread: 1
PendingThread: 0
DeadThread: 0
Hosted Runtime: no
                      PreEmptive   GC Alloc         Lock
    ID OSID ThreadOBJ   State    GC      Context      Domain   Count APT Exception
```

```
  0    1  b94 0045d5f0     a020 Enabled  01d14638:01d15fe8 00459250     1 MTA
  2    2  798 0046a1c8     b220 Enabled  00000000:00000000 00459250     0 MTA (Finalizer)

    OSID    Special thread type
 1   444    DbgHelper
 2   798    Finalizer
```

What are some examples of "special" threads?

DbgHelper, Finalizer, GC, ADUnloadHelper, Timer.

What does DeadThread mean (in output of !threads)?

My understanding is that a dead thread refers to a C++ thread which no longer has an active OS thread, but still has references and thus cannot be destroyed (C++ threads use ref counting).
A C# thread holds a reference to a C++ thread, and if your managed code keeps a reference to a C# thread, then that could be your problem.

How to correlate between .Net/managed and native thread in WinDBG?

The managed thread has a member variable m_ManagedThreadId.
The m_ManagedThreadID corresponds to the "ID" in the !threads output:

```
--3 threads out of which 1 is the debugger thread (that captured the memory
dump)
0:000> ~
. 0  Id: b0.b94 Suspend: 0 Teb: 7ffdf000 Unfrozen
  1  Id: b0.444 Suspend: 0 Teb: 7ffde000 Unfrozen
  2  Id: b0.798 Suspend: 0 Teb: 7ffdd000 Unfrozen

--2 managed/.Net threads out of which 1 is in Finalizer state i.e. 1 active
.Net thread
--first column in below !threads matches output of ~ (excluding the native
debugger thread)
--OSID column in !threads matches Id in ~
0:000> !threads
    ID OSID ThreadOBJ Exception
  0    1  b94 0045d5f0
  2    2  798 0046a1c8  (Finalizer)

-- !DumpHeap is a powerful command that traverses the garbage collected heap,
collection statistics about objects.
-- below indicates total 3 objects/threads
-- Statistics indicate the two managed/.Net threads, out of which the second
is in ThreadAbortException/Finalizer state
0:000> !dumpheap -type System.Threading.Thread
Address      MT    Size
01d110fc 656f0ebc      72
01d11144 656f0ebc      72
01d11e78 656f10bc      56
total 3 objects
Statistics:
    MT    Count    TotalSize Class Name
```

```
656f10bc       1         56 System.Threading.Thread
656f0ebc       2        144 System.Threading.ThreadAbortException
Total 3 objects

--below confirms only 1 active managed/.Net thread
-- -mt lists only those objects with the MethodTable given
0:000> !dumpheap -mt 656f10bc
Address      MT     Size
01d11e78 656f10bc      56
total 1 objects
Statistics:
    MT    Count    TotalSize Class Name
656f10bc       1         56 System.Threading.Thread
Total 1 objects

-- -short limits output to just the address of each object
0:000> !dumpheap -mt 656f10bc -short
01d11e78

-- Can !do for 01d110fc, 01d11144 or 01d11e78 (however for this example, only
the last one is active managed thread; the others are either in Finalizer
state or an a native debugger thread)
-- !do is to dumpObject
-- Below gives address of MethodTable
-- Below also gives the Offset and Values of DONT_USE_InternalThread,
m_ManagedThreadId
-- The DONT_USE_InternalThread is pointer to the native thread. Its given
Value matches the ThreadOBJ in !threads for this thread
-- The m_ManagedThreadID corresponds to the ID column in the !threads output

0:000> !do 01d11e78
Name: System.Threading.Thread
MethodTable: 656f10bc
Fields:
    MT    Field   Offset           Type VT     Attr    Value Name
656f33b0  4000648      28       System.IntPtr  1 instance   45d5f0 DONT_USE_InternalThread
656f2d34  400064a      30        System.Int32  1 instance        1 m_ManagedThreadId

-- Below displays Value of DONT_USE_InternalThread (which is same as
ThreadOBJ in !threads for this thread)
-- dd displays data as Double-word values (4 bytes). The default count is 32
DWORDs (128 bytes). i.e. displays address + 4 bytes, 8 lines @ bytes each =
32
-- poi is the best operator to use if you want pointer-sized data. It returns
pointer-sized data from the specified address. The pointer size is 32 bits or
64 bits. In kernel debugging, this size is based on the processor of the
target computer. In user-mode debugging on an Itanium-based computer, this
size is 32 bits or 64 bits, depending on the target application.
-- dd poi(MethodTable_address + Offset)

0:000> dd poi(01d11e78+28)
0045d5f0  660c1f68 0000a020 00000000 0028f0b8
0045d600  00000000 00000000 00000001 00000001
0045d610  00000000 0045d618 0045d618 0045d618
0045d620  00000000 00000000 00000000 0044ee58
```

```
0045d630  01d14638 01d15fe8 00004fd0 00000000
0045d640  00000000 00000000 00000000 00000000
0045d650  0045c138 00476e18 00476ed8 00000188
0045d660  00475e70 00000000 00000000 00000100

-- And here is a simple script to dump the managed thread object and its id,
for all managed threads

0:000> .foreach ($t {!dumpheap -mt 656f10bc -short}) {  .printf " Thread Obj ${$t} and the Thread Id is %N
\n",poi(${$t}+28) }
 Thread Obj 01d11e78 and the Thread Id is 0045D5F0

-- Below displays the information in the current thread environment block
(TEB)
-- Below TEB address is also displayed in ~
0:000> !teb
TEB at 7ffdf000

-- Below displays the TEB using the managed pointer
-- The DONT_USE_InternalThread is pointer to the native thread. Dumping the
raw memory of the pointer should give us more information we are looking for.

0:000> dd poi(01d11e78+28)
0045d5f0  660c1f68 0000a020 00000000 0028f0b8
0045d600  00000000 00000000 00000001 00000001
0045d610  00000000 0045d618 0045d618 0045d618
0045d620  00000000 00000000 00000000 0044ee58
0045d630  01d14638 01d15fe8 00004fd0 00000000
0045d640  00000000 00000000 00000000 00000000
0045d650  0045c138 00476e18 00476ed8 00000188
0045d660  00475e70 00000000 00000000 00000100
0:000> dd
0045d670  00000000 00000000 00000000 00000000
0045d680  00000000 00000000 00000000 dfca504a
0045d690  00000000 00000000 00000000 00000000
0045d6a0  00000000 00000000 ffffffff ffffffff
0045d6b0  ffffffff ffffffff 00000000 00000000
0045d6c0  00000000 00000000 0045d9c0 00000000
0045d6d0  0046a2a8 00000000 00290000 00190000
0045d6e0  00000000 00000000 00000000 00000000
0:000> dd
0045d6f0  00000000 00000000 00000000 00000000
0045d700  000000c0 00000000 000000c4 00000000
0045d710  000000c8 00000000 000000cc 00000000
0045d720  00000000 00000000 00000000 00000000
0045d730  00000000 00000000 000000bc ffffffff
0045d740  ffffffff 00000001 00000b94 001312fc
0045d750  001311f8 80000000 00000002 00000000
0045d760  00000000 00000000 00000000 00000001
0:000> dd
0045d770  00000000 00000000 cccccccc 00000000
0045d780  00000000 00000000 00000000 00000000
0045d790  00000000 00000000 00000000 00000000
0045d7a0  00000000 0045d770 00000000 ffffffff
0045d7b0  00000000 00000000 00000000 00000000
```

```
0045d7c0  00000000 00000000 00000000 00000000
0045d7d0  00000000 00000000 00000000 00000000
0045d7e0  00000000 00193000 00193000 00459250
0:000> dd
0045d7f0  00000000 00000000 00000000 00000000
0045d800  00000000 00000000 00000000 00000000
0045d810  00000001 00000001 00000000 00000002
0045d820  ffffffff ffffffff 00000000 00000000
0045d830  00000000 ffffffff 00000000 00000000
0045d840  00000000 ffffffff 00000000 00000000
0045d850  00000000 ffffffff 00000000 00000000
0045d860  00000000 ffffffff 00000000 00000000
0:000> dd
0045d870  00000000 ffffffff 00000000 00000000
0045d880  00000000 ffffffff 00000000 00000000
0045d890  00000000 ffffffff 00000000 00000000
0045d8a0  00000000 ffffffff 00000000 00000000
0045d8b0  00000000 ffffffff 00000000 00000000
0045d8c0  00000000 ffffffff 00000000 00000000
0045d8d0  00000000 ffffffff 00000000 00000000
0045d8e0  00000000 ffffffff 00000000 00000000
0:000> dd
0045d8f0  00000000 ffffffff 00000000 00000000
0045d900  00000000 ffffffff 00000000 00000000
0045d910  00000000 00000000 00000000 00000000
0045d920  00000002 ffffffff 00000000 00000000
0045d930  00000000 00000000 00000000 00000000
0045d940  00000000 00000000 00000000 00000000
0045d950  00000000 00000000 7ffdf000 00000000
0045d960  00000000 00000000 00000000 00000000
-- 7ffdf000 is the TEB

-- The pointer to teb is in the 368th (0045d950 - 0045d5f0 = 360 + 8th DWORD
= 368) offset of the  DONT_USE_InternalThread and here is the script that
would get teb for each managed thread.
-- dwo Double-word from the specified address.

0:000> .foreach ($thread {!dumpheap -mt 656f10bc -short}) { .if ( poi(${$thread}+28) != 0) {.printf "%p
\n",dwo(poi(${$thread}+28)+368)}}
7ffdf000
```

Intro to WinDBG for .NET Developers - part IV (Examining code and stacks contd.)

Whether you are developing for computers with one processor or several, you want your application to provide the most responsive interaction with the user, even if the application is currently doing other work. Using multiple threads of execution is one of the most powerful ways to keep your application responsive to the user and at the same time make use of the processor in between or even during user events.

.NET ACADEMY

Q & A:
Let's continue on the managed/.NET memory dump obtained in previous email.

What is the difference between Native vs Managed threads?

Managed thread (or .Net thread) is a thread that executes managed code (i.e. .Net code).
In .Net, managed thread is assumed to be a logical thread running within the parent process. Today a managed thread maps to a native thread (in the default host) but this is an implementation detail that can be changed in the future.

How to view only the managed/.Net stack for a thread (equivalent to kL for native call stack)?

!CLRStack attempts to provide a true stack trace for managed code only. It is handy for clean, simple traces when debugging straightforward managed programs.
When you see methods with the name "[Frame:...", that indicates a transition between managed and unmanaged code:

```
0:000> !clrstack
OS Thread Id: 0xb94 (0)
ESP      EIP
0028f0b8 77cd6344 [NDirectMethodFrameStandaloneCleanup: 0028f0b8]
System.IO.__ConsoleStream.ReadFile(Microsoft.Win32.SafeHandles.SafeFileHandle, Byte*, Int32, Int32 ByRef, IntPtr)
0028f0d4 65c0ad87
System.IO.__ConsoleStream.ReadFileNative(Microsoft.Win32.SafeHandles.SafeFileHandle, Byte[], Int32, Int32, Int32, Int32 ByRef)
0028f100 65c0aca6 System.IO.__ConsoleStream.Read(Byte[], Int32, Int32)
0028f120 65679fbb System.IO.StreamReader.ReadBuffer()
0028f134 65679e0c System.IO.StreamReader.ReadLine()
0028f154 65c0dd3d System.IO.TextReader+SyncTextReader.ReadLine()
0028f160 00390094 program.Main()
0028f384 65f81b6c [GCFrame: 0028f384]
```

How to view parameters for managed/.Net function (equivalent to kp for native stack)?

The -p parameter will show arguments to the managed function. The -l parameter can be used to show information on local variables in a frame.
SOS can't retrieve local names at this time, so the output for locals is in
the format <local address> = <value>.

!CLRStack -l (shows stack with locals)
!CLRStack -p (shows stack with parameters)

How to view both locals and parameters for managed/.Net stack?

The -a (all) parameter is a short-cut for -l and -p combined.

!CLRStack -l -p (shows both parameters and locals)
OR
!CLRStack -a (shows both parameters and locals)
To get a better idea of what happened here, you can get more detailed output from the stack by running the **!clrstack -a** command. This gives you the full stack. Here is some of that output. I have edited it here because the output is quite verbose.

0:000> **!clrstack -a**
OS Thread Id: 0xb94 (0)
ESP EIP
0028f0b8 77cd6344 [NDirectMethodFrameStandaloneCleanup: 0028f0b8]
System.IO.__ConsoleStream.ReadFile(Microsoft.Win32.SafeHandles.SafeFileHandle, Byte*, Int32, Int32 ByRef, IntPtr)
0028f0d4 65c0ad87
System.IO.__ConsoleStream.ReadFileNative(Microsoft.Win32.SafeHandles.SafeFileHandle, Byte[], Int32, Int32, Int32, Int32 ByRef)
PARAMETERS:
hFile = <no data>
bytes = <no data>
offset = <no data>
count = <no data>
mustBeZero = <no data>
errorCode = 0x0028f100
LOCALS:
<CLR reg> = 0x0028efcc
<no data>
0x0028f0d4 = 0x01d14318
<no data>

How to view the entire (native + managed/.Net) stack of a thread in single command?

!DumpStack command provides a verbose stack trace. Therefore the output is very noisy and potentially confusing. The command is good for viewing the complete call stack when "kb" gets confused (due to the managed stack in between).
SOS allows the Windows Debugger to unwind the frames, and supplies managed code information where possible. :

0:000> **!dumpstack**
OS Thread Id: 0xb94 (0)
Current frame: ntdll!KiFastSystemCallRet

ChildEBP RetAddr Caller,Callee
0028ee78 77cd570c ntdll!NtRequestWaitReplyPort+0xc
0028ee7c 76fb1351 kernel32!ConsoleClientCallServer+0x88, calling ntdll!ZwRequestWaitReplyPort
.--removed to reduce stack size
0028f0a0 65c0ad87 (MethodDesc 0x6556ab24 +0x83
System.IO.__ConsoleStream.ReadFileNative(Microsoft.Win32.SafeHandles.SafeFileHandle, Byte[], Int32, Int32, Int32, Int32 ByRef)), calling 65617114
0028f0c0 65c0ad87 (MethodDesc 0x6556ab24 +0x83
System.IO.__ConsoleStream.ReadFileNative(Microsoft.Win32.SafeHandles.SafeFileHandle, Byte[], Int32, Int32, Int32, Int32 ByRef)), calling 65617114
0028f0e8 65c0aca6 (MethodDesc 0x6556ab0c +0x5e System.IO.__ConsoleStream.Read(Byte[], Int32, Int32)), calling (MethodDesc 0x6556ab24 +0
System.IO.__ConsoleStream.ReadFileNative(Microsoft.Win32.SafeHandles.SafeFileHandle, Byte[], Int32, Int32, Int32, Int32 ByRef))
0028f110 65679fbb (MethodDesc 0x655541e0 +0x3b System.IO.StreamReader.ReadBuffer())
0028f12c 65679e0c (MethodDesc 0x655541f8 +0x13c System.IO.StreamReader.ReadLine()), calling (MethodDesc 0x655541e0 +0 System.IO.StreamReader.ReadBuffer())
0028f14c 65c0dd3d (MethodDesc 0x65592864 +0x15 System.IO.TextReader+SyncTextReader.ReadLine())
0028f158 00390094 (MethodDesc 0x152ff4 +0x24 program.Main())
--removed to reduce stack size
0028fd08 77ceb3fc ntdll!_RtlUserThreadStart+0x1b, calling ntdll!__RtlUserThreadStart

Is there a command to run something on all threads?

"~*" runs (native) command on all threads.
"~*e" runs (native or managed/.Net) command on all threads.

Example1:

~*kpn
OR
~*ekpn
OR
--below with a space
~* kpn
OR
--below with a space
~*e kpn

Example2:

~*e!dumpstack
OR
--below with a space
~*e !dumpstack

How to get !DumpStack for all threads?

!EEStack command runs !DumpStack on all threads in the process. The -EE option is passed directly to !DumpStack. This command with a -short option tries to narrow down the output to "interesting" threads only, which is defined by:

1) The thread has taken a lock.
2) The thread has been "hijacked" in order to allow a garbage collection.
3) The thread is currently in managed code.

!EEStack
OR
~*e !dumpstack

How to find .Net method associated with an address ?
Is there a manual method to find the .Net stack instead of using the automated command !clrstack?

!IP2MD attempts to find the MethodDesc associated with a given address in managed JITTED code. You could run !U, !DumpMT, !DumpClass, !DumpMD, or !DumpModule on the fields listed to learn more.
For each frame that has managed code, we can use !IP2MD to eventually find the stack returned by !clrstack.
For example, this output from K. We have taken a return address into mscorlib_ni, and discovered information about that method:

0:000> **k**
ChildEBP RetAddr
0028ee78 77cd570c ntdll!KiFastSystemCallRet
0028ee7c 76fb1351 ntdll!NtRequestWaitReplyPort+0xc
0028ee9c 76fc4ad7 kernel32!ConsoleClientCallServer+0x88
0028ef98 7700bf78 kernel32!ReadConsoleInternal+0x1ac
0028f020 76fcb2d7 kernel32!ReadConsoleA+0x40
0028f068 0014a3eb kernel32!ReadFileImplementation+0x75
WARNING: Frame IP not in any known module. Following frames may be wrong.
0028f0a0 65c0ad87 0x14a3eb
0028f0e8 **65c0aca6** mscorlib_ni+0x78ad87
0028f110 65679fbb mscorlib_ni+0x78aca6
0028f12c 65679e0c mscorlib_ni+0x1f9fbb
0028f14c 65c0dd3d mscorlib_ni+0x1f9e0c

0:000> **!ip2md 65c0aca6**
MethodDesc: 6556ab0c
Method Name: System.IO.__ConsoleStream.Read(Byte[], Int32, Int32)
Class: 65513cac
MethodTable: 656f44e0
mdToken: 06003438
Module: 65481000
IsJitted: yes
CodeAddr: 65c0ac48

-- !U presents an annotated disassembly of a managed method when given a MethodDesc pointer for the method, or a code address within the method body.

0:000> **!U 6556ab0c**
preJIT generated code
System.IO.__ConsoleStream.Read(Byte[], Int32, Int32)
Begin 65c0ac48, size 89. Cold region begin 65e08af8, size a7

Hot region:
65c0ac48 55 push ebp
65c0ac49 8bec mov ebp,esp
65c0ac4b 57 push edi
65c0ac4c 56 push esi
65c0ac4d 53 push ebx
65c0ac4e 50 push eax
65c0ac4f 33c0 xor eax,eax
65c0ac51 8945f0 mov dword ptr [ebp-10h],eax
65c0ac54 8bf9 mov edi,ecx
65c0ac56 8bf2 mov esi,edx
65c0ac58 8b5d0c mov ebx,dword ptr [ebp+0Ch]
65c0ac5b 85f6 test esi,esi
65c0ac5d 0f8495de1f00 je mscorlib_ni+0x988af8 (65e08af8)
65c0ac63 85db test ebx,ebx
65c0ac65 0f8cb8de1f00 jl mscorlib_ni+0x988b23 (65e08b23)
65c0ac6b 837d0800 cmp dword ptr [ebp+8],0
65c0ac6f 0f8caede1f00 jl mscorlib_ni+0x988b23 (65e08b23)
65c0ac75 8b4604 mov eax,dword ptr [esi+4]
65c0ac78 2bc3 sub eax,ebx
65c0ac7a 3b4508 cmp eax,dword ptr [ebp+8]
65c0ac7d 0f8ce9de1f00 jl mscorlib_ni+0x988b6c (65e08b6c)
65c0ac83 807f1c00 cmp byte ptr [edi+1Ch],0
65c0ac87 7505 jne mscorlib_ni+0x78ac8e (65c0ac8e)
65c0ac89 e8f6c2f2ff call mscorlib_ni+0x6b6f84 (65b36f84) (System.IO.__Error.ReadNotSupported(), mdToken: 06003444)
65c0ac8e 33d2 xor edx,edx
65c0ac90 8955f0 mov dword ptr [ebp-10h],edx
65c0ac93 8b4f18 mov ecx,dword ptr [edi+18h]
65c0ac96 53 push ebx
65c0ac97 ff7508 push dword ptr [ebp+8]
65c0ac9a 52 push edx
65c0ac9b 8d45f0 lea eax,[ebp-10h]
65c0ac9e 50 push eax
65c0ac9f 8bd6 mov edx,esi
65c0aca1 e85e000000 call mscorlib_ni+0x78ad04 (65c0ad04) (System.IO.__ConsoleStream.ReadFileNative(Microsoft.Win32.SafeHandles.SafeFileHandle, Byte[], Int32, Int32, Int32, Int32 ByRef), mdToken: 0600343b)

-- !DumpMD command lists information about a MethodDesc. You can use !IP2MD to turn a code address in a managed function into a MethodDesc

0:000> **!dumpmd 6556ab0c**
Method Name: System.IO.__ConsoleStream.Read(Byte[], Int32, Int32)
Class: 65513cac
MethodTable: 656f44e0
mdToken: 06003438
Module: 65481000
IsJitted: yes
CodeAddr: 65c0ac48

-- !DumpClass will show attributes, as well as list the fields of the type. The output is similar to !DumpObj. Although static field values will be displayed, non-static values won't because you need an instance of an object for that.

0:000> **!dumpclass 65513cac**
Class Name: System.IO.__ConsoleStream
mdToken: 0200058a (C:\Windows\assembly\GAC_32\mscorlib\2.0.0.0__b77a5c561934e089\mscorlib.dll)
Parent Class: 654ac760
Module: 65481000
Method Table: 656f44e0
Vtable Slots: 21
Total Method Slots: 22
Class Attributes: 100100
NumInstanceFields: 8
NumStaticFields: 0
```
     MT    Field   Offset             Type VT     Attr    Value Name
656f0704  400018a       4       System.Object  0 instance          __identity
65ced25c  4001b6d       8 ...ream+ReadDelegate  0 instance          _readDelegate
65ced2e8  4001b6e       c ...eam+WriteDelegate  0 instance          _writeDelegate
656dbe84  4001b6f      10 ...ng.AutoResetEvent  0 instance          _asyncActiveEvent
656f2d34  4001b70      14        System.Int32  1 instance          _asyncActiveCount
656ee7e4  4001b6c     574     System.IO.Stream  0   shared   static Null
   >> Domain:Value  00459250:01d12658 <<
656eeab4  4001b75      18 ...es.SafeFileHandle  0 instance          _handle
656c45b4  4001b76      1c      System.Boolean  1 instance          _canRead
656c45b4  4001b77      1d      System.Boolean  1 instance          _canWrite
```

-- !DumpMT examines a MethodTable. **Each managed object has a MethodTable pointer at the start.** If you pass the "-MD" flag, you'll also see a list of all the methods
defined on the object

0:000> **!dumpmt 656f44e0**
EEClass: 65513cac
Module: 65481000
Name: System.IO.__ConsoleStream
mdToken: 0200058a (C:\Windows\assembly\GAC_32\mscorlib\2.0.0.0__b77a5c561934e089\mscorlib.dll)
BaseSize: 0x24
ComponentSize: 0x0
Number of IFaces in IFaceMap: 1
Slots in VTable: 38

-- !DumpModule

0:000> **!dumpmodule 65481000**
Name: C:\Windows\assembly\GAC_32\mscorlib\2.0.0.0__b77a5c561934e089\mscorlib.dll
Attributes: PEFile
Assembly: 0046de00
LoaderHeap: 00000000
TypeDefToMethodTableMap: 655a5434
TypeRefToMethodTableMap: 655a4c70
MethodDefToDescMap: 655a78d8
FieldDefToDescMap: 655bc864
MemberRefToDescMap: 655a4c74
FileReferencesMap: 65485b8c
AssemblyReferencesMap: 65485bc4
MetaData start address: 6581a3b0 (1857984 bytes)

-- An assembly can consist of multiple modules, and those will be listed.

0:000> **!dumpassembly 0046de00**
Parent Domain: 664cdb48
Name: C:\Windows\assembly\GAC_32\mscorlib\2.0.0.0__b77a5c561934e089\mscorlib.dll
ClassLoader: 0046de70
SecurityDescriptor: 013192a0
  Module Name
65481000 C:\Windows\assembly\GAC_32\mscorlib\2.0.0.0__b77a5c561934e089\mscorlib.dll

Intro to WinDBG for .NET Developers - part V (still more on Examining code and stacks)

.NET Framework is designed for cross-language compatibility.
Cross-language compatibility means .NET components can interact with each other irrespective of the languages they are written in. An application written in VB .NET can reference a DLL file written in C# or a C# application can refer to a resource written in VC++, etc. This language interoperability extends to Object-Oriented inheritance.

.NET ACADEMY

Q & A:
Let's continue on the managed/.NET memory dump obtained in previous email.

What is .Net?

The .NET Framework (pronounced dot net) is a software framework that runs primarily on Microsoft Windows. It includes a large library and supports several programming languages which allows language interoperability (each language can use code written in other languages). The .NET library is available to all the programming languages that .NET supports. Programs written for the .NET Framework execute in a software environment (as contrasted to hardware environment), known as the Common Language Runtime (CLR), an application virtual machine that provides important services such as security, memory management, and exception handling. The class library and the CLR together constitute the .NET Framework.

What's Visual Studio?

Microsoft also produces a popular integrated development environment largely for .NET software called Visual Studio.

I know when you compile a .NET app c# or vb the code is converted to MSIL. I was just wondering is it 100% reversible (to get original source code from MSIL) ?

It will never be 100% reversible since each compiler may perform different optimizations and therefore the original high language source cannot be
determined, but in general there are tool which will decompile MSIL code. There are tools that will allow you to obfuscate the compiled code to make it more difficult to know what you code is doing. It won't be reversible to the point where you can see the variable names that you used, but it can be reversed to a great degree. Check out Reflector by Lutz Roeder (or the later Reflector versions by Red Hat) for a good example of what is possible.

```
--original C# code snippet
static void Main(){
        int i;
        i = 10;
        Console.WriteLine(i);
}

--reversed code (obtained from the JIT compiled exe of above code)
--note the change in variable name, etc.
private static void Main()
{
```

```
    int num = 10;
    Console.WriteLine(num);
}
```

Is there some way to compile a .NET application directly to native code?

You can Compile MSIL to Native Code. You can use Ngen.
The Native Image Generator (Ngen.exe) is a tool that improves the performance of managed applications. Ngen.exe creates native images, which are files containing compiled processor-specific machine code, and installs them into the native image cache on the local computer. The runtime can use native images from the cache instead using the just-in-time (JIT) compiler to compile the original assembly.
Unfortunately, you still need the libraries from the framework in order to run your program. There's no feature that I know of with the MS .Net framework SDK that allows you to compile all the required files into a single executable.

http://msdn.microsoft.com/en-us/library/ht8ecch6(v=VS.90).aspx
Compiling MSIL to Native Code

Are there limitations to using the Ngen approach?

There are a number of things you need to be aware of:
•You still need the CLR to run your executable.
•The CLR will not dynamically optimize your assemblies based on the environment it's run in (e.g. 486 vs. 586 vs. 686, etc.)
All in all, it's only worth using Ngen if you need to reduce the startup time of your application.

What are the steps involves for executing managed code?

The managed execution process includes the following steps:

1. Choosing a compiler.
To obtain the benefits provided by the common language runtime, you must use one or more language (C#, VB.NET, etc.) compilers that target the runtime.

2. Compiling your code to Microsoft intermediate language (MSIL).
Compiling translates your source code into MSIL and generates the required metadata.

3. Compiling MSIL to native code.
At execution time, a just-in-time (JIT) compiler translates the MSIL into native code. During this compilation, code must pass a verification process that examines the MSIL and metadata to find out whether the code can be determined to be type safe.

4. Running code.
The common language runtime (CLR) provides the infrastructure that enables execution to take place as well as a variety of services that can be used during execution.

What's MSIL?

Microsoft Intermediate Language, also called Common Intermediate Language (CIL).

Microsoft Intermediate Language (MSIL) is a platform independent language that gets compiled into platform dependent executable file or dynamic link library. It means .NET compiler can generate code written using any supported languages and finally convert it to the required machine code depending on the target machine.

.Net source code is compiled into MSIL. MSIL is an object-oriented assembly language, and is entirely stack-based. Its byte code is translated into native code or executed by a virtual machine.

What is disassembly?

Disassembly, the output of a disassembler, is often formatted for human-readability rather than suitability for input to an assembler, making it principally a reverse-engineering tool.

A disassembler is a computer program that translates machine language into assembly language—the inverse operation to that of an assembler. A disassembler differs from a de-compiler, which targets a high-level language rather than an assembly language.

Assembly language source code generally permits the use of constants and programmer comments. These are usually removed from the assembled machine code by the assembler. If so, a disassembler operating on the machine code would produce disassembly lacking these constants and comments; the disassembled output becomes more difficult for a human to interpret than the original annotated source code.

Does .Net code work on multiple platforms (like Windows, Linux, etc.)?

While the standards that make up .NET are inherently cross platform, Microsoft's full implementation of .NET is only supported on Windows. Microsoft does provide limited .NET subsets for other platforms such as XNA for Windows, XBOX 360 and Windows Phone 7, Silverlight for Windows and Mac OS X. Alternative implementations of the CLR, base class libraries, and compilers also exist (sometimes from other vendors).

What is Native Compiling?

Strictly speaking it means converting the MSIL code of a .NET assembly to native machine code and then removing the MSIL code from that assembly, making it impossible to decompile it in a straightforward way.

Does .Net code get compiled to Native Code?

The idea of .Net is to compile into native code at Runtime using a JIT compiler (CLR). There is no IDE option/compiler switch to do this. You can optimize the IL output for an architecture (any cpu vs. 32bit / 64 bit).
If you still want to compile into native code, you can use ngen.exe. But this isn't a fool proof way of guaranteeing that the JIT compile will never happen.

When you execute an assembly, the runtime looks for the native image generated (using ngen) with options and settings that match the computer's current environment. The runtime reverts to JIT compilation of an assembly, if it cannot find a matching native image. The following changes to a computer's settings and environment cause native images to become invalid:

1)The version of the .NET Framework.
If you apply a patch, QFE, or update to the .NET Framework, all native images that you have created manually using Ngen.exe become invalid. These assemblies will still run, but the runtime will not load the assembly's corresponding native image. You must manually recreate new native images for these assemblies.

2) The CPU type.
If you upgrade a computer's processor to a new processor family, all native images stored in the native image cache become invalid.

3) The version of the operating system.
If the version of the operating system running on a computer changes, all native images stored in the native image cache become invalid.

4) The exact identity of the assembly.
If you recompile an assembly, the assembly's corresponding native image becomes invalid.

The exact identity of any assemblies the assembly references.
If you recompile any of the assemblies that an assembly references, the assembly's corresponding native image becomes invalid.

5) Security factors.

Because there are many factors that affect the startup time of an application, you should carefully determine which applications would benefit from the use of Ngen.exe. Experiment by running both a JIT-compiled and a pre-compiled version of a candidate assembly in the environment in which it will be used. This will allow you to compare the startup times for the same assembly executing under different compilation schemes.

How to view the disassembly of a managed method?
Is there an equivalent command to "u" (for native)?

!U presents an annotated disassembly of a managed method when given a MethodDesc pointer for the method, or a code address within the method body.

0:000> **!clrstack**
OS Thread Id: 0x5ec (0)
ESP EIP
0015f378 77696344 [NDirectMethodFrameStandaloneCleanup: 0015f378]
System.IO.__ConsoleStream.ReadFile(Microsoft.Win32.SafeHandles.SafeFileHandle, Byte*, Int32, Int32 ByRef, IntPtr)
0015f394 **656dad87**
System.IO.__ConsoleStream.ReadFileNative(Microsoft.Win32.SafeHandles.SafeFileHandle, Byte[], Int32, Int32, Int32, Int32 ByRef)
0015f3c0 656daca6 System.IO.__ConsoleStream.Read(Byte[], Int32, Int32)
0015f3e0 65149fbb System.IO.StreamReader.ReadBuffer()
0015f3f4 65149e0c System.IO.StreamReader.ReadLine()
0015f414 656ddd3d System.IO.TextReader+SyncTextReader.ReadLine()
0015f420 72064c46 program.Main()
0015f648 68e21b6c [GCFrame: 0015f648]

```
--taking an example address 656dad87
0:000> !U 656dad87
preJIT generated code
System.IO.__ConsoleStream.ReadFileNative(Microsoft.Win32.SafeHandles.SafeFileHandle, Byte[], Int32, Int32,
Int32, Int32 ByRef)
Begin 656dad04, size c2. Cold region begin 658d8ba8, size 38
...
656dad82 e88dc3a0ff     call    mscorlib_ni+0x197114 (650e7114)
(System.IO.__ConsoleStream.ReadFile(Microsoft.Win32.SafeHandles.SafeFileHandle, Byte*, Int32, Int32 ByRef,
IntPtr), mdToken: 0600343d)
>>> 656dad87 33d2          xor     edx,edx
656dad89 8955ec        mov     dword ptr [ebp-14h],edx
656dad8c 85c0          test    eax,eax
656dad8e 7525          jne     mscorlib_ni+0x78adb5 (656dadb5)
656dad90 e843719fff     call    mscorlib_ni+0x181ed8 (650d1ed8)
(System.Runtime.InteropServices.Marshal.GetLastWin32Error(), mdToken: 06003196)
656dad95 8903          mov     dword ptr [ebx],eax
656dad97 833b6d        cmp     dword ptr [ebx],6Dh
656dad9a 750c          jne     mscorlib_ni+0x78ada8 (656dada8)

0:000> u 656dad87
mscorlib_ni+0x78ad87:
656dad87 33d2          xor     edx,edx
656dad89 8955ec        mov     dword ptr [ebp-14h],edx
656dad8c 85c0          test    eax,eax
656dad8e 7525          jne     mscorlib_ni+0x78adb5 (656dadb5)
656dad90 e843719fff     call    mscorlib_ni+0x181ed8 (650d1ed8)
656dad95 8903          mov     dword ptr [ebx],eax
656dad97 833b6d        cmp     dword ptr [ebx],6Dh
656dad9a 750c          jne     mscorlib_ni+0x78ada8 (656dada8)
```

What's GC (in .Net)?

GC = Garbage Collection
In the common language runtime (CLR), the garbage collector (GC) serves as an automatic memory manager.
The .NET Framework's garbage collector manages the allocation and release of memory for your application. Each time you create a new object, the common language runtime allocates memory for the object from the managed heap. As long as address space is available in the managed heap, the runtime continues to allocate space for new objects. However, memory is not infinite. Eventually the garbage collector must perform a collection in order to free some memory. The garbage collector's optimizing engine determines the best time to perform a collection, based upon the allocations being made. When the garbage collector performs a collection, it checks for objects in the managed heap that are no longer being used by the application and performs the necessary operations to reclaim their memory.

How to know more information about Garbage Collector (GC) from a memory dump?

!GCInfo function is important for CLR Devs, but very difficult for anyone else to make sense of it. You would usually come to use it if you suspect a gc heap corruption bug caused by invalid GCEncoding for a particular method.
!GCInfo is especially useful for CLR Devs who are trying to determine if there is a bug in the JIT Compiler. It parses the GCEncoding for a method, which is a compressed stream of data indicating

when registers or stack locations contain managed objects. It is important to keep track of this information, because if a garbage collection occurs, the collector needs to know where roots are so it can update them with new object pointer values.
Normally you would print this output out and read it alongside a disassembly of the method.

```
0:000> !clrstack
OS Thread Id: 0x5ec (0)
ESP      EIP
0015f378 77696344 [NDirectMethodFrameStandaloneCleanup: 0015f378]
System.IO.__ConsoleStream.ReadFile(Microsoft.Win32.SafeHandles.SafeFileHandle, Byte*, Int32, Int32 ByRef,
IntPtr)
0015f394 656dad87
System.IO.__ConsoleStream.ReadFileNative(Microsoft.Win32.SafeHandles.SafeFileHandle, Byte[], Int32, Int32,
Int32, Int32 ByRef)
0015f3c0 656daca6 System.IO.__ConsoleStream.Read(Byte[], Int32, Int32)
0015f3e0 65149fbb System.IO.StreamReader.ReadBuffer()
0015f3f4 65149e0c System.IO.StreamReader.ReadLine()
0015f414 656ddd3d System.IO.TextReader+SyncTextReader.ReadLine()
0015f420 72064c46 program.Main()
0015f648 68e21b6c [GCFrame: 0015f648]

0:000> !u 656dad87
preJIT generated code
System.IO.__ConsoleStream.ReadFileNative(Microsoft.Win32.SafeHandles.SafeFileHandle, Byte[], Int32, Int32,
Int32, Int32 ByRef)
--note the begin address 656dad04 of this method
Begin 656dad04, size c2. Cold region begin 658d8ba8, size 38
...
--note the usage of [EBP-14H] in the code, you'll first find the
initialization to 0
--XOR operation is a short-cut to set the value of a register to zero.
Performing XOR on a value against itself always yields zero, and on many
architectures, this operation requires fewer CPU clock cycles than the
sequence of operations that may be required to load a zero value and save it
to the register.
656dad0d 33c0        xor    eax,eax
656dad0f 8945f0      mov    dword ptr [ebp-10h],eax
656dad12 8945ec      mov    dword ptr [ebp-14h],eax
...
--then its population
656dad56 8d5208      lea    edx,[edx+8]
656dad59 8955ec      mov    dword ptr [ebp-14h],edx
656dad5c 8b7dec      mov    edi,dword ptr [ebp-14h]
...
-- The last reference is interesting. The pointer is held live until the end
of the scope and is now de-referenced
656dad87 33d2        xor    edx,edx
656dad89 8955ec      mov    dword ptr [ebp-14h],edx

--[EBP-14H] is our pointer so now we read the mentioned !u output with
reference to this pointer
0:000> !gcinfo 656dad87
--entry point is the begin address 656dad04 of this method
entry point 656dad04
preJIT generated code
```

GC info 65825b53
Method info block:
  method   size = 00FA
  prolog   size = 17
  epilog   size = 10
  epilog   count = 4
  epilog   end   = no
  callee-saved regs = EDI ESI EBX EBP
  ebp frame   = yes
  fully interruptible= no
  double align   = no
  arguments size   = 4 DWORDs
  stack frame size = 2 DWORDs
  untracked count   = 1
  var ptr tab count = 0
  epilog # 0   at 0031
  epilog # 1   at 009A
  epilog # 2   at 00A7
  epilog # 3   at 00B8
81 7A DE 8C B5 |
B8 93 F1 3F 31 |
69 0D 11 |

Pointer table:
11 | [**EBP-14H**] an untracked pinned byref local
F9 74 C0 80 | 0074   call [ESI EBX'] argMask=00
F9 0F 80 80 | 0083   call [EBX'] argMask=00
F9 0E 80 80 | 0091   call [EBX'] argMask=00
CC 40 | 00DD   call [ESI] argMask=00
27 | 00E4   call [ESI] argMask=00
29 | 00ED   call [ESI] argMask=00
FF |

--can relate it to the other output we've covered so far
0:000> **k**
ChildEBP RetAddr
0015f138 7769570c ntdll!KiFastSystemCallRet
0015f13c 75c41351 ntdll!NtRequestWaitReplyPort+0xc
0015f15c 75c54ad7 kernel32!ConsoleClientCallServer+0x88
0015f258 75c9bf78 kernel32!ReadConsoleInternal+0x1ac
0015f2e0 75c5b2d7 kernel32!ReadConsoleA+0x40
0015f328 0029a3eb kernel32!ReadFileImplementation+0x75
WARNING: Frame IP not in any known module. Following frames may be wrong.
0015f360 **656dad87** 0x29a3eb
0015f3a8 656daca6 mscorlib_ni+0x78ad87
0015f3d0 65149fbb mscorlib_ni+0x78aca6
0015f3ec 65149e0c mscorlib_ni+0x1f9fbb
0015f40c 656ddd3d mscorlib_ni+0x1f9e0c
0015f418 72064c46 mscorlib_ni+0x78dd3d
0015f420 68e21b6c test_ni+0x4c46
0015f430 68e32209 mscorwks!CallDescrWorker+0x33
0015f4b0 68e46511 mscorwks!CallDescrWorkerWithHandler+0xa3
0015f5f4 68e46544 mscorwks!MethodDesc::CallDescr+0x19c
0015f610 68e46562 mscorwks!MethodDesc::CallTargetWorker+0x1f
0015f628 68eb0c45 mscorwks!MethodDescCallSite::Call_RetArgSlot+0x1a

0015f78c 68eb0b65 mscorwks!ClassLoader::RunMain+0x223
0015f9f4 68eb10b5 mscorwks!Assembly::ExecuteMainMethod+0xa6

```
--code address is the begin address 656dad04 of this method
```
0:000> **!ip2md 656dad87**
MethodDesc: 6503ab24
Method Name: System.IO.__ConsoleStream.ReadFileNative(Microsoft.Win32.SafeHandles.SafeFileHandle, Byte[], Int32, Int32, Int32, Int32 ByRef)
Class: 64fe3cac
MethodTable: 651c44e0
mdToken: 0600343b
Module: 64f51000
IsJitted: yes
CodeAddr: **656dad04**

What does "IsJitted" mean?

IsJitted means that it has been compiled:

0:000> **!ip2md 656dad87**
MethodDesc: 6503ab24
Method Name: System.IO.__ConsoleStream.ReadFileNative(Microsoft.Win32.SafeHandles.SafeFileHandle, Byte[], Int32, Int32, Int32, Int32 ByRef)
Class: 64fe3cac
MethodTable: 651c44e0
mdToken: 0600343b
Module: 64f51000
IsJitted: yes
CodeAddr: **656dad04**

How to know the exception handling blocks in a jitted method?

!EHInfo shows the exception handling blocks in a jitted method. For each handler, it shows the type, including code addresses and offsets for the clause block and the handler block. For a TYPED handler, this would be the "try" and "catch" blocks respectively.

```
--below indicates that no exception handling has been defined (since no
mention of TYPED, etc.)
```
0:000> **!ehinfo 656dad04**
MethodDesc: 6503ab24
Method Name: System.IO.__ConsoleStream.ReadFileNative(Microsoft.Win32.SafeHandles.SafeFileHandle, Byte[], Int32, Int32, Int32, Int32 ByRef)
Class: 64fe3cac
MethodTable: 651c44e0
mdToken: 0600343b
Module: 64f51000
IsJitted: yes
CodeAddr: 656dad04

For example, a different memory dump with a try-catch-finally block in Main() would indicate below:

```
--this is a different memory dump
```
0:000> **!ehinfo 006100ac**
MethodDesc: 00322ff4
Method Name: test1.Main()
Class: 00321260
MethodTable: 00323008
mdToken: 06000001
Module: 00322c5c
IsJitted: yes
CodeAddr: 00610070

```
--below indicates a try-catch block
```
EHHandler 0: TYPED
Clause: [0061008c, 006100ae] [1c, 3e]
Handler: [006100ae, 006100c1] [3e, 51]

```
--below indicates that the try-catch block also has a finally block
```
EHHandler 1: FINALLY
Clause: [0061008c, 006100d6] [1c, 66]
Handler: [006100d6, 006100ee] [66, 7e]

What is COM?

Component Object Model (COM) is a binary-interface standard for software componentry introduced by Microsoft in 1993. It is used to enable interprocess communication and dynamic object creation in a large range of programming languages. The term COM is often used in the Microsoft software development industry as an umbrella term that encompasses the OLE, OLE Automation, ActiveX, COM+ and DCOM technologies.

COM programmers build their software using COM-aware components. Different component types are identified by class IDs (CLSIDs), which are Globally Unique Identifiers (GUIDs). Each COM component exposes its functionality through one or more interfaces. The different interfaces supported by a component are distinguished from each other using interface IDs (IIDs), which are GUIDs too.

For example, one COM related SQL Server 2008 R2 error:
TITLE: Maintenance Plan Wizard Progress

Create maintenance plan failed.

ADDITIONAL INFORMATION:
Creating an instance of the COM component with CLSID {17BCA6E8-A95D-497E-B2F9-AF6AA475916F} from the IClassFactory failed due to the following error: c001f011. (Microsoft.SqlServer.ManagedDTS)

Is there any relation between COM and .Net?

COM development has largely been superseded by the Microsoft .NET, with .NET providing wrappers to the most commonly used COM controls.
Several of the services that COM+ provides have been largely replaced by recent releases of .NET. For example, the System.Transactions namespace in .NET provides the TransactionScope class, which provides transaction management without resorting to COM+. Similarly, queued components can be replaced by Windows Communication Foundation with an MSMQ transport.

WCF (Window Communication Foundation) solves a number of COM's remote execution shortcomings, allowing objects to be transparently marshalled by value across process or machine boundaries.

Despite this, COM remains a viable technology with an important software base. It is also ideal for script control of applications such as Office or Internet Explorer since it provides an interface for calling COM object methods from a script rather than requiring knowing the API at compile time. The GUID system used by COM has wide uses any time a unique ID is needed.

What are apartment models in COM?

In COM, threading issues are addressed by a concept known as "apartment models". Here the term "apartment" refers to an execution context wherein a single thread or a group of threads is associated with one or more COM objects.

What are the types of apartment models in COM?

There are three types of Apartment Models in the COM world: **Single-Threaded Apartment (STA), Multi-Threaded Apartment (MTA), and Neutral Apartment**. Each apartment represents one mechanism whereby an object's internal state may be synchronized across multiple threads.
The Single-Threaded Apartment (STA) model is a very commonly used model. Here, a COM object stands in a position similar to a desktop application's user interface. In an STA model, a single thread is dedicated to drive an object's methods, i.e. a single thread is always used to execute the methods of the object. In such an arrangement, method calls from threads outside of the apartment are marshalled and automatically queued by the system (via a standard Windows message queue). Thus, there is no worry about race conditions or lack of synchronicity because each method call of an object is always executed to completion before another is invoked.
If the COM object's methods perform their own synchronization, multiple threads dedicated to calling methods on the COM object are permitted. This is termed the Multiple Threaded Apartment (MTA). Calls to an MTA object from a thread in an STA are also marshaled. A process can consist of multiple COM objects, some of which may use STA and others of which may use MTA.

The Thread Neutral Apartment allows different threads, none of which is necessarily dedicated to calling methods on the object, to make such calls. The only provision is that all methods on the object must be serially reentrant.

How to know the COM apartment model of a thread?

!COMState lists the com apartment model for each thread, as well as a Context pointer if provided:

```
0:000> !comstate
   ID    TEB  APT    APTId CallerTID Context
 0  5ec 7ffdf000 MTA      0       0 00401da8
 1  f34 7ffde000 Ukn
 2  fe0 7ffdc000 MTA      0       0 00401da8
--thread "1" is the debugger thread (which not a .Net thread hence will not
show in !threads output)

--!comstate has output as !threads
0:000> !threads
ThreadCount: 2
```

UnstartedThread: 0
BackgroundThread: 1
PendingThread: 0
DeadThread: 0
Hosted Runtime: no

	ID	OSID	ThreadOBJ	State	PreEmptive GC	Alloc Context	Lock Domain	Count	**APT**	Exception
0	1	5ec	003cd5d0	a020	Enabled	01864638:01865fe8	003c9230	1	**MTA**	
2	2	fe0	003da1a8	b220	Enabled	00000000:00000000	003c9230	0	**MTA**	(Finalizer)

What is the command to set a managed breakpoint (equivalent to bp for normal breakpoint)?

!BPMD provides managed breakpoint support. If it can resolve the method name to a loaded, jitted or ngen'd function it will create a breakpoint with "bp". If not then either the module that contains the method hasn't been loaded yet or the module is loaded, but the function is not jitted yet. In these cases, !bpmd asks the Windows Debugger to receive CLR Notifications, and waits to receive news of module loads and JITs, at which time it will try to resolve the function to a breakpoint.

I want to set a breakpoint on the main method of my application, but SOS doesn't work until the runtime is loaded. How can I do this?

1) Start the debugger and type:
   sxe -c "" clrn
2) g
3) You'll get the following notification from the debugger:
   "CLR notification: module 'mscorlib' loaded"
4) Now you can load SOS and use commands. Type
   .loadby sos mscorwks
   then
   !bpmd myapp.exe MyApp.Main
5) g
6) You will stop at the start of MyApp.Main. If you type "bl" you will see the breakpoint listed.

```
--below links for more information to !bpmd
```
http://blogs.msdn.com/b/kristoffer/archive/2007/01/02/setting-a-breakpoint-in-managed-code-using-windbg.aspx
Setting a breakpoint in managed code using Windbg

http://blogs.msdn.com/b/tess/archive/2008/04/28/setting-breakpoints-in-net-code-using-bpmd.aspx
Setting breakpoints in .net code using !bpmd

```
-- CLR is using clrn/CLRNotificationException to notify sos/sosex on JIT
0:000> .if (dwo(mscorwks!g_dacNotificationFlags) = 0) {.echo bp not set } .else {.echo bp set}
bp not set
-- Here is the code to set the value to "1" (this is a hack and is not
required. If the !bpmd is set after load of mscorjit/clrjit it would work as
expected.)
0:000> ed mscorwks!g_dacNotificationFlags 00000001
```

Intro to WinDBG for .NET Developers - part VI (Object Inspection)

When developing a .Net application, one of the least visible sources of memory consumption is the overhead required by an object simply to exist. In applications that create a lot of small objects, this overhead can be a major or even a dominant factor in the total memory requirements for the application.

.NET ACADEMY

Q & A:
Let's continue on the managed/.NET memory dump obtained in previous email.

What are objects (in programming)?

Objects are the building blocks of OOP (Object Oriented Programming) and are commonly defined as variables or data structures that encapsulate behavior and data in a programmed unit. Objects are items that can be individually created, manipulated, and represent real world things in an abstract way.

What are the commands available for Object Inspection in SOS.dll?

- DumpObj (do) - allows to examine fields and important properties of an object.
- DumpArray (da) - allows you to examine elements of an array object.
- DumpStackObjects (dso) - display any managed objects it finds within the bounds of the current stack. Combined with the stack tracing commands like K and !CLRStack, it is a good aid to determining the values of locals and parameters.
- DumpHeap - powerful command that traverses the garbage collected heap, collection statistics about objects. With its various options, it can look for particular types, restrict to a range, or look for ThinLocks (see !SyncBlk documentation). Finally, it will provide a warning if it detects excessive fragmentation in the GC heap.
- DumpVC - allows you to examine the fields of a value class. In C#, this is a struct, and lives on the stack or within an Object on the GC heap.
- GCRoot - looks for references (or roots) to an object. These can exist in four places:

  1. On the stack
  2. Within a GC Handle
  3. In an object ready for finalization
  4. As a member of an object found in 1, 2 or 3 above.

- ObjSize - lists the size of all objects found on managed threads. In calculating object size, !ObjSize includes the size of all child objects in addition to the parent.
- FinalizeQueue - lists the objects registered for finalization.

- PrintException (pe) - will notify you if there are any nested exceptions on the current managed thread. (A nested exception occurs when you throw another exception within a catch handler already being called for another exception).

  If there are nested exceptions, you can re-run !PrintException with the "-nested" option to get full details on the nested exception objects. The !Threads command will also tell you which threads have nested exceptions.

  This will format fields of any object derived from System.Exception. One of themore useful aspects is that it will format the _stackTrace field, which is a binary array. If _stackTraceString field is not filled in, that can be helpful for debugging. You can of course use !DumpObj on the same exception object to explore more fields.

  PS !PrintException for a managed thread is equivalent to **!gle** for a native thread.

- TraverseHeap - writes out a file in a format understood by the CLR Profiler. You can download the CLR Profiler from this link.
  You can break into your process, load SOS, take a snapshot of your heap with this function, then continue:

  http://www.microsoft.com/downloads/en/details.aspx?FamilyID=A362781C-3870-43BE-8926-862B40AA0CD0
  CLR Profiler for the .NET Framework 2.0

```
-- You might find an object pointer by running !DumpStackObjects and choosing
from the resultant list.
```

0:000> **!dumpstackobjects**
OS Thread Id: 0x5ec (0)
ESP/REG Object Name
0015f35c 018640ac Microsoft.Win32.SafeHandles.SafeFileHandle
0015f36c 018640ac Microsoft.Win32.SafeHandles.SafeFileHandle
0015f3a0 01864310 System.Byte[]
0015f3a4 018640c0 System.IO.__ConsoleStream
0015f3c4 **018640f0** System.IO.StreamReader
0015f3c8 018640f0 System.IO.StreamReader
0015f3e0 018640f0 System.IO.StreamReader
0015f3e4 01864628 System.IO.TextReader+SyncTextReader
0015f404 01864628 System.IO.TextReader+SyncTextReader

```
-- Fields of type System.IO.TextReader, System.Text.Decoder, etc. are
themselves objects and you can run !dumpobj on them too
-- The column VT contains the value 1 if the field is a valuetype structure,
and 0 if the field contains a pointer to another object.
```

0:000> **!do 018640f0**
Name: System.IO.StreamReader
MethodTable: 651a8a68
EEClass: 64fda1c0
Size: 60(0x3c) bytes
(C:\Windows\assembly\GAC_32\mscorlib\2.0.0.0__b77a5c561934e089\mscorlib.dll)
Fields:

```
    MT    Field   Offset            Type VT     Attr    Value Name
651c0704  400018a      4       System.Object  0 instance 00000000 __identity
651b51a8  4001c37     58c System.IO.TextReader  0   shared   static Null
  >> Domain:Value  003c9230:NotInit  <<
651945b4  4001c3d     34      System.Boolean  1 instance       0 _closable
651be7e4  4001c3e      8    System.IO.Stream  0 instance 018640c0 stream
651c3458  4001c3f      c System.Text.Encoding  0 instance 018627e0 encoding
651ab0d4  4001c40     10  System.Text.Decoder  0 instance 018642f4 decoder
651c3558  4001c41     14       System.Byte[]  0 instance 01864310 byteBuffer
651c1718  4001c42     18       System.Char[]  0 instance 0186441c charBuffer
651c3558  4001c43     1c       System.Byte[]  0 instance 01862674 _preamble
651c2d34  4001c44     20        System.Int32  1 instance       0 charPos
651c2d34  4001c45     24        System.Int32  1 instance       0 charLen
651c2d34  4001c46     28        System.Int32  1 instance       0 byteLen
651c2d34  4001c47     2c        System.Int32  1 instance       0 bytePos
651c2d34  4001c48     30        System.Int32  1 instance     256 _maxCharsPerBuffer
651945b4  4001c49     35      System.Boolean  1 instance       0 _detectEncoding
651945b4  4001c4a     36      System.Boolean  1 instance       0 _checkPreamble
651945b4  4001c4b     37      System.Boolean  1 instance       0 _isBlocked
651a8a68  4001c3c    590 ...m.IO.StreamReader  0   shared   static Null
  >> Domain:Value  003c9230:0186412c <<

-- For valuetypes, you can take the MethodTable pointer in the MT column, and
the Value and pass them to the command !DumpVC.
--  The Value column provides the start address

0:000> !dumpvc 651945b4 0
Name: System.Boolean
MethodTable 651945b4
EEClass: 64f6edf4
Size: 12(0xc) bytes
(C:\Windows\assembly\GAC_32\mscorlib\2.0.0.0__b77a5c561934e089\mscorlib.dll)
Fields:
    MT    Field   Offset            Type VT     Attr    Value Name
651945b4  4000210      0      System.Boolean  1 instance        m_value
651c1718  4000211     58       System.Char[]  0   shared   static m_trimmableChars
  >> Domain:Value  003c9230:NotInit  <<
651c0ae8  4000212     5c       System.String  0   shared   static TrueString
  >> Domain:Value  003c9230:NotInit  <<
651c0ae8  4000213     60       System.String  0   shared   static FalseString
  >> Domain:Value  003c9230:NotInit  <<

-- You could look at the field directly in memory using the offset given.
-- dd displays Double-word values (4 bytes).
-- To specify an address range by an address and object count, specify an
address argument, the letter L (uppercase or lowercase), and a value
argument. The address specifies the starting address. The value specifies the
number of objects to be examined or displayed. The size of the object depends
on the command. For example, if the object size is 1 byte, the following
example is a range of 8 bytes (l1), beginning at the address 0x1864100
(018640f0+10).

0:000> dd 018640f0+10 l1
01864100  018642f4
```

0:000> **dd 018640f0+10 l8**
01864100 018642f4 01864310 0186441c 01862674
01864110 00000000 00000000 00000000 00000000

```
-- What else can you do with an object? You might run !GCRoot, to determine
what roots are keeping it alive.
```

0:000> **k**
ChildEBP RetAddr
0015f138 7769570c ntdll!KiFastSystemCallRet
0015f13c 75c41351 ntdll!NtRequestWaitReplyPort+0xc
0015f15c 75c54ad7 kernel32!ConsoleClientCallServer+0x88
0015f258 75c9bf78 kernel32!ReadConsoleInternal+0x1ac
0015f2e0 75c5b2d7 kernel32!ReadConsoleA+0x40
0015f328 0029a3eb kernel32!ReadFileImplementation+0x75
WARNING: Frame IP not in any known module. Following frames may be wrong.
0015f360 **656dad87** 0x29a3eb
0015f3a8 656daca6 mscorlib_ni+0x78ad87
0015f3d0 65149fbb mscorlib_ni+0x78aca6
0015f3ec 65149e0c mscorlib_ni+0x1f9fbb
0015f40c 656ddd3d mscorlib_ni+0x1f9e0c
0015f418 72064c46 mscorlib_ni+0x78dd3d
0015f420 68e21b6c test_ni+0x4c46
0015f430 68e32209 mscorwks!CallDescrWorker+0x33
0015f4b0 68e46511 mscorwks!CallDescrWorkerWithHandler+0xa3
0015f5f4 68e46544 mscorwks!MethodDesc::CallDescr+0x19c
0015f610 68e46562 mscorwks!MethodDesc::CallTargetWorker+0x1f
0015f628 68eb0c45 mscorwks!MethodDescCallSite::Call_RetArgSlot+0x1a
0015f78c 68eb0b65 mscorwks!ClassLoader::RunMain+0x223
0015f9f4 68eb10b5 mscorwks!Assembly::ExecuteMainMethod+0xa6

0:000> **!ip2md 656dad87**
MethodDesc: 6503ab24
Method Name: System.IO.__ConsoleStream.ReadFileNative(Microsoft.Win32.SafeHandles.SafeFileHandle, Byte[], Int32, Int32, Int32, Int32 ByRef)
Class: 64fe3cac
MethodTable: 651c44e0
mdToken: 0600343b
Module: 64f51000
IsJitted: yes
CodeAddr: 656dad04

0:000> **!gcinfo 6503ab24**
entry point 656dad04
preJIT generated code
GC info 65825b53

```
-- Or you can find all objects of that type with "!DumpHeap -type".
```

0:000> **!dumpheap -type System.IO.StreamReader**
Address   MT   Size
018640f0 651a8a68   60
0186412c 651a8b54   60

total 2 objects
Statistics:
    MT   Count   TotalSize Class Name
651a8b54   1   60 System.IO.StreamReader+NullStreamReader
651a8a68   1   60 System.IO.StreamReader
Total 2 objects

```
-- When called without options, dumpheap output is first a list of objects in
the heap, followed by a report listing all the types found, their size and
number
-- "Free" objects are simply regions of space the garbage collector can use
later.
```

0:000> **!dumpheap**
Address   MT   Size
01861000 003d0c00   12 Free
0186100c 003d0c00   12 Free
01861018 003d0c00   12 Free
01861024 651c0d0c   72
0186106c 651c0d9c   72
018610b4 651c0e2c   72
...
total 325 objects
Statistics:
    MT   Count   TotalSize Class Name
651c3fbc   1   12 System.Text.DecoderExceptionFallback
651c3f78   1   12 System.Text.EncoderExceptionFallback
657ec044   1   16 System.IO.TextReader+SyncTextReader
...
003d0c00   7   100   Free
651942b8   24   9412 System.Object[]
Total 325 objects

```
-- If 30% or more of the heap contains "Free" objects, the process may suffer
from heap fragmentation. This is usually caused by pinning objects for a long
time combined with a high rate of allocation. Here is example output where
!DumpHeap provides a warning about fragmentation:
```

<After the Statistics section>
Fragmented blocks larger than 1MB:
   Addr   Size Followed by
00a780c0   1.5MB   00bec800 System.Byte[]
00da4e38   1.2MB   00ed2c00 System.Byte[]
00f16df0   1.2MB   01044338 System.Byte[]

```
-- dumpheap -strings  restricts the output to a statistical string value
summary
```

0:000> **!dumpheap -strings**
total 125 objects
Statistics:
  Count   TotalSize String Value
      1   20 "\"
      1   20 ";"

```
    1        24 "\."
...
    1        40 "Hello World"
..
    5       140 "en-US"
    3       144 "C:\cs\test.exe"
    1       148 "C:\Windows\Globalization\en-us.nlp"
    1       204 "C:\Windows\Microsoft.NET\Framework\v2.0.50727\config\machine.co"
    2       224 "C:\Windows\Microsoft.NET\Framework\v2.0.50727\"
    2       232 "NLS_CodePage_437_3_2_0_0"
Total 125 objects

-- dumpheap -min ignores objects less than the size given in bytes
-- dumpheap -max ignores objects larger than the size given in bytes

0:000> !dumpheap -min 500
Address      MT     Size
01863c60 651c1718     524
0186441c 651c1718     524
02861010 651942b8    4096
02862020 651942b8     528
02862240 651942b8    4096
total 5 objects
Statistics:
    MT    Count    TotalSize Class Name
651c1718       2        1048 System.Char[]
651942b8       3        8720 System.Object[]
Total 5 objects

0:000> !dumpheap -min 500 -max 4000
Address      MT     Size
01863c60 651c1718     524
0186441c 651c1718     524
02862020 651942b8     528
total 3 objects
Statistics:
    MT    Count    TotalSize Class Name
651942b8       1         528 System.Object[]
651c1718       2        1048 System.Char[]
Total 3 objects

-- dumpheap -startAtLowerBound forces heap walk to begin at lower bound of a
supplied address range.
          (During plan phase, the heap is often not walkable because objects
          are being moved. In this case, DumpHeap may report spurious errors,
          in particular bad objects. It may be possible to traverse more of
          the heap after the reported bad object. Even if you specify an
          address range, !DumpHeap will start its walk from the beginning of
          the heap by default. If it finds a bad object before the specified
          range, it will stop before displaying the part of the heap in which
          you are interested. This switch will force !DumpHeap to begin its
          walk at the specified lower bound. You must supply the address of a
          good object as the lower bound for this to work. Display memory at
          the address of the bad object to manually find the next method
          table (use !dumpmt to verify). If the GC is currently in a call to
```

```
          memcopy, You may also be able to find the next object's address by
          adding the size to the start address given as parameters.)

-- dumpheap -mt lists only those objects with the given MethodTable
```

0:000> **!dumpheap -mt 651942b8**
Address MT Size
018612ac 651942b8 80
01861c6c 651942b8 16
...<lists total 24 objects>
total 24 objects
Statistics:
MT Count TotalSize Class Name
651942b8 24 9412 System.Object[]
Total 24 objects

```
-- dumpheap -type lists only those objects whose type name is a substring
match of the string provided.
```

0:000> **!dumpheap -type System**
Address MT Size
01861024 651c0d0c 72
...
651c313c 7 392 System.Collections.Hashtable
651c3238 7 1008 System.Collections.Hashtable+bucket[]
651c3558 6 1108 System.Byte[]
651c1718 25 2124 System.Char[]
651c0ae8 125 4780 System.String
651942b8 24 9412 System.Object[]
Total 311 objects

```
-- dumpheap - A special note about -type: Often, you'd like to find not only
Strings, but
System.Object arrays that are constrained to contain Strings. ("new
String[100]" actually creates a System.Object array, but it can only hold
System.String object pointers). You can use -type in a special way to find
these arrays. Just pass "-type System.String[]" and those Object arrays will
be returned. More generally, "-type <Substring of interesting type>[]".
```

0:000> **!dumpheap -type System.String[]**
Address MT Size
018612ac 651942b8 80
01861cd0 651942b8 20
01861d38 651942b8 20
01861d6c 651942b8 20
01861ed4 651942b8 20
01862314 651942b8 20
0186235c 651942b8 20
01862390 651942b8 20
018634fc 651942b8 56
018635c0 651942b8 56
total 10 objects
Statistics:
MT Count TotalSize Class Name

651942b8 10 332 System.Object[]
Total 10 objects

0:000> **!do 018612ac**
Name: System.Object[]
MethodTable: 651942b8
EEClass: 64f7da64
Size: 80(0x50) bytes
Array: Rank 1, Number of elements 16, Type CLASS
Element Type: System.String
Fields:
None

```
-- !DumpObj lists a size of 80 bytes for this System.Object[] object
-- but !ObjSize lists 164 bytes
-- This is probably because System.Object[] has 16 elements (and 0 fields)
```

0:000> **!do 018612ac**
Name: System.Object[]
MethodTable: 651942b8
EEClass: 64f7da64
Size: 80(0x50) bytes
Array: Rank 1, Number of elements 16, Type CLASS
Element Type: System.String
Fields:
None

0:000> **!objsize 018612ac**
sizeof(018612ac) = 164 (0xa4) bytes (System.Object[])

```
-- You can use !ObjSize to identify any particularly large objects, such as a
managed cache in a web server.
```

0:000> **!objsize**
Scan Thread 0 OSTHread 5ec
ESP:15f35c: sizeof(018640ac) = 20 (0x14) bytes (Microsoft.Win32.SafeHandles.SafeFileHandle)
ESP:15f36c: sizeof(018640ac) = 20 (0x14) bytes (Microsoft.Win32.SafeHandles.SafeFileHandle)
ESP:15f3a0: sizeof(01864310) = 268 (0x10c) bytes (System.Byte[])
ESP:15f3a4: sizeof(018640c0) = 56 (0x38) bytes (System.IO.__ConsoleStream)
ESP:15f3c4: sizeof(018640f0) = 1108 (0x454) bytes (System.IO.StreamReader)
ESP:15f3c8: sizeof(018640f0) = 1108 (0x454) bytes (System.IO.StreamReader)
ESP:15f3e0: sizeof(018640f0) = 1108 (0x454) bytes (System.IO.StreamReader)
ESP:15f3e4: sizeof(01864628) = 1124 (0x464) bytes (System.IO.TextReader+SyncTextReader)
ESP:15f404: sizeof(01864628) = 1124 (0x464) bytes (System.IO.TextReader+SyncTextReader)
Scan Thread 2 OSTHread fe0
DOMAIN(003C9230):HANDLE(Strong):2811ac: sizeof(018619c0) = 24 (0x18) bytes (System.Reflection.Assembly)
DOMAIN(003C9230):HANDLE(Strong):2811b4: sizeof(01861918) = 36 (0x24) bytes (System.Security.PermissionSet)
DOMAIN(003C9230):HANDLE(Strong):2811bc: sizeof(01861918) = 36 (0x24) bytes (System.Security.PermissionSet)
DOMAIN(003C9230):HANDLE(Strong):2811c0: sizeof(0186199c) = 24 (0x18) bytes (System.Reflection.Assembly)

DOMAIN(003C9230):HANDLE(Strong):2811cc: sizeof(01861918) = 36 (0x24) bytes (System.Security.PermissionSet)
DOMAIN(003C9230):HANDLE(Strong):2811d0: sizeof(01861918) = 36 (0x24) bytes (System.Security.PermissionSet)
DOMAIN(003C9230):HANDLE(Strong):2811d4: sizeof(018611ac) = 28 (0x1c) bytes (System.SharedStatics)
DOMAIN(003C9230):HANDLE(Strong):2811d8: sizeof(01861144) = 72 (0x48) bytes (System.Threading.ThreadAbortException)
DOMAIN(003C9230):HANDLE(Strong):2811dc: sizeof(018610fc) = 72 (0x48) bytes (System.Threading.ThreadAbortException)
DOMAIN(003C9230):HANDLE(Strong):2811e0: sizeof(018610b4) = 72 (0x48) bytes (System.ExecutionEngineException)
DOMAIN(003C9230):HANDLE(Strong):2811e4: sizeof(0186106c) = 72 (0x48) bytes (System.StackOverflowException)
DOMAIN(003C9230):HANDLE(Strong):2811e8: sizeof(01861024) = 72 (0x48) bytes (System.OutOfMemoryException)
DOMAIN(003C9230):HANDLE(Strong):2811f8: sizeof(01861e78) = 56 (0x38) bytes (System.Threading.Thread)
DOMAIN(003C9230):HANDLE(Strong):2811fc: sizeof(0186121c) = 308 (0x134) bytes (System.AppDomain)
DOMAIN(003C9230):HANDLE(WeakSh):2812fc: sizeof(01861e78) = 56 (0x38) bytes (System.Threading.Thread)
DOMAIN(003C9230):HANDLE(Pinned):2813f0: sizeof(02862240) = 6888 (0x1ae8) bytes (System.Object[])
DOMAIN(003C9230):HANDLE(Pinned):2813f4: sizeof(02862020) = 548 (0x224) bytes (System.Object[])
DOMAIN(003C9230):HANDLE(Pinned):2813f8: sizeof(0186118c) = 12 (0xc) bytes (System.Object)
DOMAIN(003C9230):HANDLE(Pinned):2813fc: sizeof(02861010) = 11220 (0x2bd4) bytes (System.Object[])

```
-- dumpheap -details asks the command to print out details of the element
using !DumpObj and !DumpVC format.
```

0:000> **!dumparray -details 018612ac**
Name: System.String[]
MethodTable: 651942b8
EEClass: 64f7da64
Size: 80(0x50) bytes
Array: Rank 1, Number of elements 16, Type CLASS
Element Methodtable: 651c0ae8
[0] 018611c8
  Name: System.String
  MethodTable: 651c0ae8
  EEClass: 64f7d65c
  Size: 30(0x1e) bytes
  (C:\Windows\assembly\GAC_32\mscorlib\2.0.0.0__b77a5c561934e089\mscorlib.dll)
  String: C:\cs\
  Fields:
      MT Field Offset Type VT Attr Value Name
  651c2d34 4000096 4 System.Int32 1 instance 7 m_arrayLength
  651c2d34 4000097 8 System.Int32 1 instance 6 m_stringLength
  651c17c8 4000098 c System.Char 1 instance 43 m_firstChar
  651c0ae8 4000099 10 System.String 0 shared static Empty
  >> Domain:Value 003c9230:01861198 <<
  651c1718 400009a 14 System.Char[] 0 shared static WhitespaceChars

```
   >> Domain:Value  003c9230:018616d4 <<
[1] 018611e8
   Name: System.String
   MethodTable: 651c0ae8
   EEClass: 64f7d65c
   Size: 52(0x34) bytes
    (C:\Windows\assembly\GAC_32\mscorlib\2.0.0.0__b77a5c561934e089\mscorlib.dll)
   String:    C:\cs\test.config
   Fields:
      MT    Field   Offset            Type VT     Attr    Value Name
   651c2d34  4000096      4       System.Int32  1 instance       18 m_arrayLength
   651c2d34  4000097      8       System.Int32  1 instance       17 m_stringLength
   651c17c8  4000098      c        System.Char  1 instance       43 m_firstChar
   651c0ae8  4000099     10       System.String  0   shared   static Empty
   >> Domain:Value  003c9230:01861198 <<
   651c1718  400009a     14       System.Char[]  0   shared   static WhitespaceChars
   >> Domain:Value  003c9230:018616d4 <<
[2] null
[3] null

-- dumpheap -start specifies from which index the command shows the elements.
Only supported for single dimension array.
-- -length specifies how many elements to show. Only supported for single
dimension array.

0:000> !dumparray -start 0 -length 2 -details 018612ac
Name: System.String[]
MethodTable: 651942b8
EEClass: 64f7da64
Size: 80(0x50) bytes
Array: Rank 1, Number of elements 16, Type CLASS
Element Methodtable: 651c0ae8
[0] 018611c8
   Name: System.String
   MethodTable: 651c0ae8
   EEClass: 64f7d65c
   Size: 30(0x1e) bytes
    (C:\Windows\assembly\GAC_32\mscorlib\2.0.0.0__b77a5c561934e089\mscorlib.dll)
   String:    C:\cs\
   Fields:
      MT    Field   Offset            Type VT     Attr    Value Name
   651c2d34  4000096      4       System.Int32  1 instance        7 m_arrayLength
   651c2d34  4000097      8       System.Int32  1 instance        6 m_stringLength
   651c17c8  4000098      c        System.Char  1 instance       43 m_firstChar
   651c0ae8  4000099     10       System.String  0   shared   static Empty
   >> Domain:Value  003c9230:01861198 <<
   651c1718  400009a     14       System.Char[]  0   shared   static WhitespaceChars
   >> Domain:Value  003c9230:018616d4 <<
[1] 018611e8
   Name: System.String
   MethodTable: 651c0ae8
   EEClass: 64f7d65c
   Size: 52(0x34) bytes
    (C:\Windows\assembly\GAC_32\mscorlib\2.0.0.0__b77a5c561934e089\mscorlib.dll)
   String:    C:\cs\test.config
```

```
  Fields:
      MT    Field   Offset                 Type VT     Attr    Value Name
  651c2d34  4000096      4       System.Int32  1 instance       18 m_arrayLength
  651c2d34  4000097      8       System.Int32  1 instance       17 m_stringLength
  651c17c8  4000098      c       System.Char  1 instance       43 m_firstChar
  651c0ae8  4000099     10       System.String  0   shared   static Empty
  >> Domain:Value  003c9230:01861198 <<
  651c1718  400009a     14       System.Char[]  0   shared   static WhitespaceChars
  >> Domain:Value  003c9230:018616d4 <<

-- dumpheap start end
-- start      Begin listing from this address
-- end        Stop listing at this address
-- The start/end parameters can be obtained from the output of !EEHeap -gc.
For
example, if you only want to list objects in the large heap segment:
-- !EEHeap enumerates process memory consumed by internal CLR data
structures.

0:000> !eeheap -gc
Number of GC Heaps: 1
generation 0 starts at 0x01861018
generation 1 starts at 0x0186100c
generation 2 starts at 0x01861000
ephemeral segment allocation context: none
segment    begin allocated     size
01860000 01861000  01865ff4 0x00004ff4(20468)
Large object heap starts at 0x02861000
segment    begin allocated     size
02860000 02861000  02863250 0x00002250(8784)
Total Size    0x7244(29252)
------------------------------
GC Heap Size    0x7244(29252)

0:000> !dumpheap 02861000 02863250
Address       MT     Size
02861000 003d0c00      16 Free
02861010 651942b8    4096
02862010 003d0c00      16 Free
02862020 651942b8     528
02862230 003d0c00      16 Free
02862240 651942b8    4096
02863240 003d0c00      16 Free
total 7 objects
Statistics:
    MT    Count    TotalSize Class Name
003d0c00       4         64     Free
651942b8       3       8720 System.Object[]
Total 7 objects

-- dumpheap -stat restricts the output to the statistical type summary

-- Good run
0:000> !dumpheap -stat
```

total 325 objects
Statistics:
    MT   Count   TotalSize Class Name
651c3fbc   1   12 System.Text.DecoderExceptionFallback
...
651c0ae8   125   4780 System.String
651942b8   24   9412 System.Object[]
Total 325 objects

```
-- Bad run
-- If gc heap corruption is present, you may see an error like this:
```
0:000> **!dumpheap -stat**
object 00a73d22: does not have valid MT
curr_object : 00a73d22
Last good object: 00a73d12

```
-- If called with no parameters, PrintException will look for the last
outstanding exception on the current thread and print it. This will be the
same exception that shows up in a run of !Threads.
```

0:000> **!pe**
There is no current managed exception on this thread

0:000> **!gle**
LastErrorValue: (Win32) 0 (0) - The operation completed successfully.
LastStatusValue: (NTSTATUS) 0xc0000034 - Object Name not found.

0:000> **~1s**

0:001> **!pe**
The current thread is unmanaged

```
-- Sample output
```

0:000> **!finalizequeue**
SyncBlocks to be cleaned up: 0
MTA Interfaces to be released: 0
STA Interfaces to be released: 0
---
generation 0 has 5 finalizable objects (**003d98e0->003d98f4**)
generation 1 has 0 finalizable objects (003d98e0->003d98e0)
generation 2 has 0 finalizable objects (003d98e0->003d98e0)
Ready for finalization 0 objects (003d98f4->003d98f4)
Statistics:
    MT   Count   TotalSize Class Name
651c4a04   1   20 Microsoft.Win32.SafeHandles.SafeFileMappingHandle
651c49ac   1   20 Microsoft.Win32.SafeHandles.SafeViewOfFileHandle
651beab4   2   40 Microsoft.Win32.SafeHandles.SafeFileHandle
651c10bc   1   56 System.Threading.Thread
Total 5 objects

```
-- The GC heap is divided into generations, and objects are listed
accordingly. We see that only generation 0 (the youngest generation) has any
objects registered for finalization. The notation "(003d98e0->003d98f4)"
```

means that if you look at memory in that range, you'll see the object pointers that are registered

-- You could run !DumpObj on any of those pointers to learn more. In this example, there are no objects ready for finalization, presumably because they still have roots (You can use !GCRoot to find out). The statistics section provides a higher-level summary of the objects registered for finalization. Note that objects ready for finalization are also included in the statistics (if any).

-- If you pass -detail then you get extra information on any SyncBlocks that need to be cleaned up, and on any RuntimeCallableWrappers (RCWs) that await cleanup.
Both of these data structures are cached and cleaned up by the finalizer thread when it gets a chance to run.

0:000> **dd 003d98e0 003d98f4**
003d98e0 01861e78 01862620 01863800 01863814
003d98f0 018640ac 00000000

0:000> **dd 003d98e0 003d98f4-4**
003d98e0 01861e78 01862620 01863800 01863814
003d98f0 018640ac

0:000> **dd 003d98e0 003d98f4+4**
003d98e0 01861e78 01862620 01863800 01863814
003d98f0 018640ac 00000000 00000000

-- Sample output
-- Some caution: !GCRoot doesn't attempt to determine if a stack root it encountered is valid or is old (discarded) data. You would have to use !CLRStack and !U to disassemble the frame that the local or argument value belongs to in order to determine if it is still in use.

0:000> **!gcroot 01861e78**
Note: Roots found on stacks may be false positives. Run "!help gcroot" for
more info.
Scan Thread 0 OSTHread 5ec
Scan Thread 2 OSTHread fe0
DOMAIN(003C9230):HANDLE(Strong):2811f8:Root:01861e78(System.Threading.Thread)
DOMAIN(003C9230):HANDLE(WeakSh):2812fc:Root:01861e78(System.Threading.Thread)

-- Sample output

-- View output in CLR Profiler

0:000> **!traverseheap -xml C:\th.xml**
Writing Xml format to file C:\th.xml
Gathering types...
tracing roots...
Scan Thread 0 OSTHread 5ec
Scan Thread 2 OSTHread fe0
Walking heap...
.....

file C:\th.xml saved

```
-- C:\th.xml

<gcheap>
        <types>
                <type id="1" name="System.Object[]"/>
                <type id="2" name="System.Text.DecoderNLS"/>
                ...
        </types>
        <roots>
                <root kind="handle" address="0x018619C0"/>
                <root kind="handle" address="0x01861918"/>
                ...
        </roots>
        <objects>
                <object address="0x01861024" typeid="12" size="72">
                </object>
                <object address="0x0186106C" typeid="13" size="72">
                </object>
                <object address="0x0186121C" typeid="18" size="100">
                  <member address="0x01861280"/>
                </object>
                ...
        </objects>
</gcheap>

-- Advanced dumpheap (automation)
```

0:000> **!dumpheap -mt 651942b8 -short**
018612ac
01861c6c
01861cd0
01861d38
01861d4c
01861d6c
01861e3c
01861ed4
01861f3c
01861f9c
01861fb0
01862014
01862314
0186235c
01862370
01862390
01862404
0186309c
018630d0
018634fc
018635c0
02861010
02862020
02862240

```
-- Can objsize each address
```

0:000> **!objsize 018612ac**
sizeof(018612ac) = 164 (0xa4) bytes (System.Object[])

```
-- OR better can automate objsize for all addresses returned by dumpheap
```

0:000> **.foreach(x { !dumpheap -mt 651942b8 -short }) { !objsize x}**
sizeof(018612ac) = 164 (0xa4) bytes (System.Object[])
sizeof(01861c6c) = 16 (0x10) bytes (System.Object[])
sizeof(01861cd0) = 68 (0x44) bytes (System.Object[])
sizeof(01861d38) = 68 (0x44) bytes (System.Object[])
sizeof(01861d4c) = 80 (0x50) bytes (System.Object[])
sizeof(01861d6c) = 68 (0x44) bytes (System.Object[])
sizeof(01861e3c) = 80 (0x50) bytes (System.Object[])
sizeof(01861ed4) = 52 (0x34) bytes (System.Object[])
sizeof(01861f3c) = 64 (0x40) bytes (System.Object[])
sizeof(01861f9c) = 52 (0x34) bytes (System.Object[])
sizeof(01861fb0) = 52 (0x34) bytes (System.Object[])
sizeof(01862014) = 104 (0x68) bytes (System.Object[])
sizeof(01862314) = 52 (0x34) bytes (System.Object[])
sizeof(0186235c) = 52 (0x34) bytes (System.Object[])
sizeof(01862370) = 64 (0x40) bytes (System.Object[])
sizeof(01862390) = 52 (0x34) bytes (System.Object[])
sizeof(01862404) = 64 (0x40) bytes (System.Object[])
sizeof(0186309c) = 1036 (0x40c) bytes (System.Object[])
sizeof(018630d0) = 96 (0x60) bytes (System.Object[])
sizeof(018634fc) = 256 (0x100) bytes (System.Object[])
sizeof(018635c0) = 256 (0x100) bytes (System.Object[])
sizeof(02861010) = 11220 (0x2bd4) bytes (System.Object[])
sizeof(02862020) = 548 (0x224) bytes (System.Object[])
sizeof(02862240) = 6888 (0x1ae8) bytes (System.Object[])

```
-- Advanced.
```

How can I determine what the source of those 2 StreamReaders are?

Try to find instances of StreamReader (for System.IO.StreamReader).

Then dump the callback properties (where callback address is the "Address" from the dumpheap output):
!do <callback address>

Then dump the Value address of the _target property:
!do <_target address>
That should spit out the object that holds a reference to the StreamReader, which should lead you to where the timer was created.

```
-- In below example, we get to know that the StreamReader is a
System.IO.__ConsoleStream i.e. that it redirects output to Console (and not
to a Windows Form, etc.)
```

0:000> **!dumpheap -type System.IO.StreamReader**
Address MT Size
018640f0 651a8a68 60
0186412c 651a8b54 60
total 2 objects
Statistics:
MT Count TotalSize Class Name
651a8b54 1 60 System.IO.StreamReader+NullStreamReader
651a8a68 1 60 System.IO.StreamReader
Total 2 objects

0:000> **!do 018640f0**
Name: System.IO.StreamReader
MethodTable: 651a8a68
EEClass: 64fda1c0
Size: 60(0x3c) bytes
(C:\Windows\assembly\GAC_32\mscorlib\2.0.0.0__b77a5c561934e089\mscorlib.dll)
Fields:
MT Field Offset Type VT Attr **Value** Name
651c0704 400018a 4 System.Object 0 instance 00000000 __identity
651b51a8 4001c37 58c System.IO.TextReader 0 shared static Null
>> Domain:Value 003c9230:NotInit <<
651945b4 4001c3d 34 System.Boolean 1 instance 0 _closable
651be7e4 4001c3e 8 System.IO.Stream 0 instance **018640c0** stream
651c3458 4001c3f c System.Text.Encoding 0 instance 018627e0 encoding
651ab0d4 4001c40 10 System.Text.Decoder 0 instance 018642f4 decoder
651c3558 4001c41 14 System.Byte[] 0 instance 01864310 byteBuffer
651c1718 4001c42 18 System.Char[] 0 instance 0186441c charBuffer
651c3558 4001c43 1c System.Byte[] 0 instance 01862674 _preamble
651c2d34 4001c44 20 System.Int32 1 instance 0 charPos
651c2d34 4001c45 24 System.Int32 1 instance 0 charLen
651c2d34 4001c46 28 System.Int32 1 instance 0 byteLen
651c2d34 4001c47 2c System.Int32 1 instance 0 bytePos
651c2d34 4001c48 30 System.Int32 1 instance 256 _maxCharsPerBuffer
651945b4 4001c49 35 System.Boolean 1 instance 0 _detectEncoding
651945b4 4001c4a 36 System.Boolean 1 instance 0 _checkPreamble
651945b4 4001c4b 37 System.Boolean 1 instance 0 _isBlocked
651a8a68 4001c3c 590 ...m.IO.StreamReader 0 shared static Null
>> Domain:Value 003c9230:0186412c <<

0:000> **!do 018640c0**
Name: **System.IO.__ConsoleStream**
MethodTable: 651c44e0
EEClass: 64fe3cac
Size: 36(0x24) bytes
(C:\Windows\assembly\GAC_32\mscorlib\2.0.0.0__b77a5c561934e089\mscorlib.dll)
Fields:
MT Field Offset Type VT Attr Value Name
651c0704 400018a 4 System.Object 0 instance 00000000 __identity
657bd25c 4001b6d 8 ...ream+ReadDelegate 0 instance 00000000 _readDelegate
657bd2e8 4001b6e c ...eam+WriteDelegate 0 instance 00000000 _writeDelegate
651abe84 4001b6f 10 ...ng.AutoResetEvent 0 instance 00000000 _asyncActiveEvent
651c2d34 4001b70 14 System.Int32 1 instance 1 _asyncActiveCount

```
651be7e4  4001b6c     574    System.IO.Stream  0   shared   static Null
   >> Domain:Value  003c9230:01862658 <<
651beab4  4001b75      18 ...es.SafeFileHandle  0 instance 018640ac _handle
651945b4  4001b76      1c      System.Boolean  1 instance        1 _canRead
651945b4  4001b77      1d      System.Boolean  1 instance        0 _canWrite
```

Intro to WinDBG for .NET Developers - part VII (CLR internals)

Since the common language runtime (CLR) (the SQL Server equivalent is called SQLCLR) will be the premiere infrastructure for building applications in Windows® for some time to come, gaining a deep understanding of it will help you build efficient, industrial-strength applications.

.NET ACADEMY

Q & A:
Let's continue on the managed/.NET memory dump obtained in previous email.

What are some of the files in the .Net framework?

The mscorrc.dll file is a Microsoft .NET Runtime resources library. The mscorrc.dll module contains resources such as icons and text for natural language translations. The .NET framework's functionality is contained in several Dynamic Link Libraries that are loaded and executed through the Windows 32 bit subsystem.

The component that runs the mscorrc.dll module is the mscoree.dll module or the Component Object Runtime Execution Engine. The mscoree.dll file is termed the "startup shim" and is unique on the machine (located in folder %SystemRoot%\System32\), regardless of the number of side by side installations or the version of the .NET Framework. After loading the mscorrc.dll module, it's the task of the mscoree.dll file to hand over execution to a specific version of the virtual machine component of Microsoft's .NET program or the Common Language Runtime depending on a number of factors. An installation of the CLR contains a bunch of files that help with the runtime execution of the .NET framework and with the prefix mscor, such as:

* mscorrc.dll - the resource library of the runtime component

* mscor**wks**.dll - the workstation version of the CLR

* mscor**svr**.dll - the server version of the CLR

* mscorlib.dll - contains a part of the System namespace of managed classes and contains low-level functionality that has a close relationship with the CLR itself (e.g., code to support concepts such as application domains)

* mscorjit.dll - the just in time compiler of the CLR to compile to native code at runtime

The existence of these files conveys that you have installed the .NET framework in your operating system. Dynamic Link Libraries such as the mscorrc.dll module are responsible for initializing the Common Language Runtime.

What is mscordacwks.dll and why is it required (considering we already have mscorwks.dll and sos.dll)?

mscorwks is the main MS CLR implementation.

mscordacwks provides a data-access abstraction over the CLR details, so that debuggers (such as SOS) don't need to know too much about the internals of mscorwks. The dll is used by Windows Debugger to enumerate CLR data structures. Mscordacwks.dll is part of .Net framework redist.

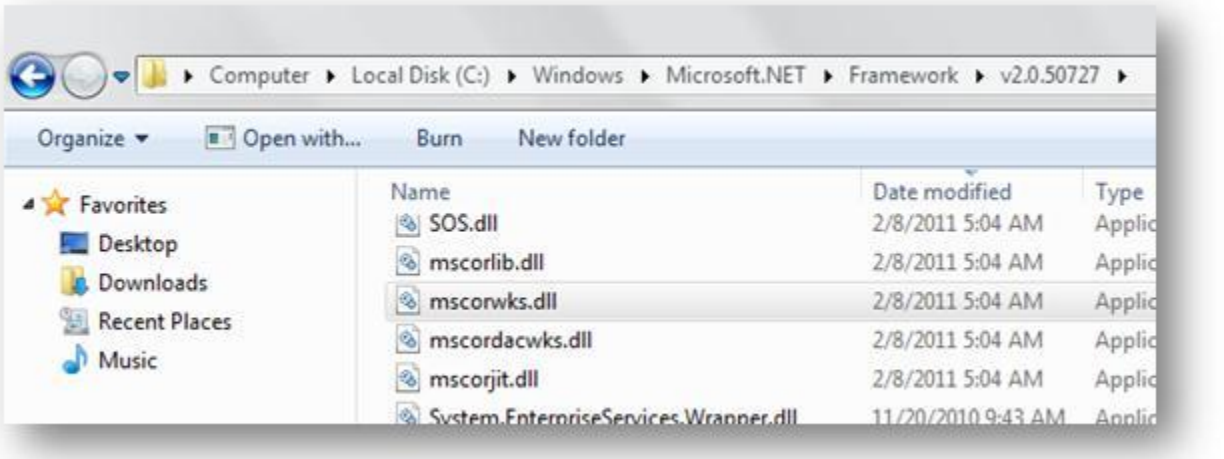
But in general, you simply don't need to work with these - so unless you're writing your own IDE/debugger, you can probably leave them alone.

The Common Language Runtime (CLR) is the core engine of the Microsoft .NET Framework that executes managed code. In simple terms it does this by taking the intermediate language and metadata in a managed assembly, JIT compiling the code on demand, building in memory representations of the types the assembly defines and uses and ensures the resulting code is safe, secure and verifiable and gets executed when it is meant to. This engine is itself implemented in native code. When we want to debug a .NET application using a native debugger like CDB or WinDBG (which we currently do a lot of if we want to debug it using post-mortem memory dump files) we have to use a "bridge" between the native debugger and the managed world because the native debugger does not inherently understand managed code. It is a native debugger.

To provide this bridge, the CLR helpfully ships with a debugger extension – SOS.DLL. This understands the internals of the CLR and so allows us to do things like outputting managed calls stacks, dumping the managed heap etc.

But from time to time, these internal data structures and details of the CLR change and so it is useful to abstract the interface to the CLR that this debugger extension needs from the actual internal implementation of the CLR that makes .NET applications work. Enter mscordacwks.dll. This provides the Data Access Component (DAC) that allows the SOS.DLL debugger extension to interpret the in memory data structures that maintain the state of a .NET application.

If you look in your framework folder you should always see a matching set of these 3 DLLs (mscorwks, mscordacwks, sos):



If you work with 64-bit you should also see a matching DLL set in the Framework64 folder.

I recently installed Windows updates (this updated versions of mscorwks.dll, etc.) and now am getting below errors:

-- note the version x86_x86_2.0.50727.4959 in below error
-- above denotes the process dump was on x86 system (with x86 debugger) and the process was x86
-- 4959 is the build number of the framework version
0:000> **.loadby sos mscorwks**
0:000> **!threads**
Failed to load data access DLL, 0x80004005
Verify that 1) you have a recent build of the debugger (6.2.14 or newer)
2) the file mscordacwks.dll that matches your version of mscorwks.dll is in the version directory
3) or, if you are debugging a dump file, verify that the file mscordacwks_<arch>_<arch>_<version>.dll is on your symbol path.
4) you are debugging on the same architecture as the dump file. For example, an IA64 dump file must be debugged on an IA64 machine.

You can also run the debugger command .cordll to control the debugger's load of mscordacwks.dll. .cordll -ve -u -l will do a verbose reload. If that succeeds, the SOS command should work on retry.

If you are debugging a minidump, you need to make sure that your executable path is pointing to mscorwks.dll as well.
0:000> **.cordll**
CLR DLL status: ERROR: Unable to load DLL mscordacwks_x86_x86_2.0.50727.4959.dll, Win32 error 0n2
0:000> **.cordll -ve -u -l**
CLR DLL status: No load attempts

Please refer below website for the options for this "Failed to load data access DLL" error. Further details below:
http://blogs.msdn.com/b/dougste/archive/2009/02/18/failed-to-load-data-access-dll-0x80004005-or-what-is-mscordacwks-dll.aspx

The simplest thing is to ask the person that gave you the dump file to look at to give you a **copy of the mscordacwks.dll**. Once you have it, check its file properties for the version number. It should be something like 2.0.50727.xxxx. Then rename it to

mscordacwks_AAA_AAA_2.0.50727.xxxx.dll

where xxxx is the appropriate bit of the version number and AAA is either x86 or AMD64 depending on whether you are dealing with a 32-bit or a 64-bit application dump. (The AMD64 is a legacy thing before we referred to x64). Then put this renamed copy into your debuggers directory (the one where WinDBG is installed). Then, as per the error message, tell the debugger to try again:

.cordll -ve -u -l

Although we try to ensure that every build of the CLR that is released (as a service pack, a hotfix or whatever) has its mscordacwks.dll indexed on the public **symbol server**, unfortunately it sometimes does not happen. But since it always ships as part of the CLR you always have the option of getting it from the machine the dump came from.

Intro to WinDBG for .NET Developers - part VIII (more CLR internals - Application Domains)

A process is the smallest unit of isolation available on the Windows operating system. This could pose a problem for an application server which may be required to host hundreds of ASP.NET applications on a single server. The server admin will want to isolate each ASP.NET application to prevent one application from interfering with another client's application on the same server, but the relative cost of launching and executing a process for hundreds of applications may be prohibitive.

Let's consider a SQL Server example. A little-known behavior of SQLCLR (.Net CLR component of SQL Server) is that SQLCLR creates one appdomain per assembly owner. This is NOT one appdomain per user, but one AppDomain per assembly owner. This is an example of multiple AppDomins in a single process.

PS. A single ASP.NET worker process will host ASP.NET applications. On Windows 2003, the ASP.NET worker process has the name w3wp.exe and runs under the NETWORK SERVICE account by default.

.NET ACADEMY

Q & A:
Let's continue on the managed/.NET memory dump obtained in previous email.

What is a "process" in an operating system?

Operating systems and runtime environments typically provide some form of isolation between applications. For example, Windows uses processes to isolate applications. This isolation is necessary to ensure that code running in one application cannot adversely affect other, unrelated applications.

A process is an instance of a computer program that is being executed. It contains the program code and its current activity. Depending on the operating system (OS), a process may be made up of multiple threads of execution that execute instructions concurrently.

A computer program is a passive collection of instructions, a process is the actual execution of those instructions. Several processes may be associated with the same program; for example (notepad, Internet Explorer, SQL Server, etc.), opening up several instances of the same program often means more than one process is being executed.

What is an Application Domain (also called AppDomain)?

Application domains provide an/another (besides processes etc.) isolation boundary for security, reliability, and versioning, and for unloading assemblies.

In the Common Language Runtime (CLR), an application domain is a mechanism (similar to a process in an operating system) used to isolate executed software applications from one another so that they do not affect each other. Each application domain has its own virtual address space which scopes the resources for the application domain using that address space.

Before .Net Framework 2.0 technology, the only way used to isolate applications running on the same machine is by the means of process boundaries. Each application run within a process, and each process has its own boundaries and memory addresses relative to it and this is how isolation from other processes was performed.

.Net Framework 2.0 introduces a new boundary called the Application Domains. Each application running within its main process boundaries and its application domain boundaries. So, one can think of an application domain as an extra shell to isolate the application and making it more secure and robust.

What is the difference between a process and an Application Domain?

A Common Language Runtime (CLR) application domain is contained within an operating system process. A process may contain many application domains.

What are some similarities between a process and an Application Domain?

Application Domains have isolation properties similar to that of operating system processes:

- Multiple threads can exist within a single application domain.
- The application in a domain can be stopped without affecting the state of another domain in the same process.
- A fault or exception in one domain does not affect an application in another domain or crash the entire process that hosts the domains.
- Configuration information is part of a domain's scope, not the scope of the process.
- Each domain can be assigned different security access levels.
- Code in one domain cannot directly access code in another.

Historically, process boundaries have been used to isolate applications running on the same computer. Each application is loaded into a separate process, which isolates the application from other applications running on the same computer.

The applications are isolated because memory addresses are process-relative; a memory pointer passed from one process to another cannot be used in any meaningful way in the target process. In addition, you cannot make direct calls between two processes. Instead, you must use proxies, which provide a level of indirection.

Application domains provide a more secure and versatile unit of processing that the common language runtime can use to provide isolation between applications. You can run several application domains in a single process with the same level of isolation that would exist in separate processes, but without incurring the additional overhead of making cross-process calls or switching between processes. The ability to run multiple applications within a single process dramatically increases server scalability.

How does an AppDomain get created?
What is the relation between AppDomain and assemblies (exe, dll)?

A common language runtime (CLR) host creates application domains automatically when they are needed. However, you can create your own application domains (using

System.AppDomain.CreateDomain) and load into them those assemblies that you want to manage personally.

AppDomains are usually created by hosts. Examples of hosts are the Windows Shell, ASP.NET, IE, SQLCLR. When we run a .NET application from the command-line, the host is the Shell. The Shell creates a new AppDomain for every application.

What are the different AppDomains created by CLR?

Before the CLR executes the first line of the managed code, it creates three application domains. Two of these are opaque from within the managed code and are not even visible to CLR hosts. They can only be created through the CLR bootstrapping process facilitated by the shim — mscoree.dll and mscorwks.dll (or mscorsvr.dll for multiprocessor systems). These two are the System Domain and the Shared Domain, which are singletons.

The third domain is the Default AppDomain, an instance of the AppDomain class that is the only named domain. For simple CLR hosts such as a console program, the default domain name is composed of the executable image name. Additional domains can be created from within managed code using the AppDomain.CreateDomain method or from unmanaged hosting code using the ICORRuntimeHost interface.

Complicated hosts like ASP.NET create multiple domains based on the number of applications in a given Web site.

System Domain
The SystemDomain is responsible for creating and initializing the SharedDomain and the default AppDomain. It loads the system library mscorlib.dll into SharedDomain. It also keeps process-wide string literals interned implicitly or explicitly.

SharedDomain
All of the domain-neutral code is loaded into SharedDomain. Mscorlib, the system library, is needed by the user code in all the AppDomains. It is automatically loaded into SharedDomain. Fundamental types from the System namespace like Object, ValueType, Array, Enum, String, and Delegate get preloaded into this domain during the CLR bootstrapping process.

DefaultDomain
DefaultDomain is an instance of AppDomain within which application code is typically executed. While some applications require additional AppDomains to be created at runtime (such as apps that have plug-in architectures or apps doing a significant amount of run-time code generation), most applications create one domain during their lifetime. All code that executes in this domain is context-bound at the domain level.

```
-- !DumpDomain gives information on at least 3 domains

-- Our console program, Test.exe, is loaded into an AppDomain which has a
name "Test.exe." Mscorlib.dll is loaded into the SharedDomain as it is the
core system library. A HighFrequencyHeap, LowFrequencyHeap, and StubHeap are
allocated in each domain. The Default AppDomain uses its own ClassLoader.
```

0:000> **!DumpDomain**

PDB symbol for mscorwks.dll not loaded

System Domain: 684f71a8
LowFrequencyHeap: 684f71cc
HighFrequencyHeap: 684f7218
StubHeap: 684f7264
Stage: OPEN
Name: None

Shared Domain: 684f6af8
LowFrequencyHeap: 684f6b1c
HighFrequencyHeap: 684f6b68
StubHeap: 684f6bb4
Stage: OPEN
Name: None
Assembly: 003cdde8

Domain 1: 003b9238
LowFrequencyHeap: 003b925c
HighFrequencyHeap: 003b92a8
StubHeap: 003b92f4
Stage: OPEN
SecurityDescriptor: 003b9f20
Name: test.exe
Assembly: 003cdde8
[C:\Windows\assembly\GAC_32\mscorlib\2.0.0.0__b77a5c561934e089\mscorlib.dll]
ClassLoader: 003cde58
SecurityDescriptor: 003cb080
  Module Name
66ad1000 C:\Windows\assembly\GAC_32\mscorlib\2.0.0.0__b77a5c561934e089\mscorlib.dll

Assembly: 003f5d90 [C:\cs\test.exe]
ClassLoader: 003f5e00
SecurityDescriptor: 003f8a88
  Module Name
72451000 C:\cs\test.exe

```
-- !DumpAssembly can return the assembly name
```

0:000> **!DumpAssembly 003cdde8**
Parent Domain: 684f6af8
Name: C:\Windows\assembly\GAC_32\mscorlib\2.0.0.0__b77a5c561934e089\mscorlib.dll
ClassLoader: 003cde58
SecurityDescriptor: 01238a28
  Module Name
66ad1000 C:\Windows\assembly\GAC_32\mscorlib\2.0.0.0__b77a5c561934e089\mscorlib.dll

```
-- !threads also gives us the default AppDomain, etc.
```

0:000> **!threads**

ThreadCount: 2
UnstartedThread: 0
BackgroundThread: 1
PendingThread: 0
DeadThread: 0
Hosted Runtime: no

	ID	OSID	ThreadOBJ	State	GC	Context	**Domain**	Count	APT	Exception
0	1	9a8	003bd5d8	a020	Enabled	01a54698:01a55fe8	**003b9238**	1	MTA	
2	2	854	003ca1b0	b220	Enabled	00000000:00000000	003b9238	0	MTA	(Finalizer)

```
-- Display the types defined in a module (using exe/module address for
example from !DumpDomain)
```

0:000> **!DumpModule -mt 72451000**
Name: C:\cs**test.exe**
Attributes: PEFile
Assembly: 003f5d90
LoaderHeap: 00000000
TypeDefToMethodTableMap: 72451528
TypeRefToMethodTableMap: 724514fc
MethodDefToDescMap: 72451534
FieldDefToDescMap: 72451540
MemberRefToDescMap: 72451510
FileReferencesMap: 72451380
AssemblyReferencesMap: 72451384
MetaData start address: 72454954 (644 bytes)

Types defined in this module

MT	TypeDef	Name
72454c4c	0x02000002	**program**

```
-- There will be one EEClass for each type loaded into an AppDomain.

-- There will be one MethodTable for each declared type and all the object
instances of the same type will point to the same MethodTable

-- EEClass and MethodTable are logically one data structure (together they
represent a single type).

-- "program" is the name of the class in our test.exe program

-- No need the specify the module name test.exe, can instead specify "*" for
all loaded managed modules
```

0:000> **!Name2EE test.exe!program**
Module: 72451000 (test.exe)
Token: 0x02000002
MethodTable: **72454c4c**
EEClass: **7245146c**

Name: program

0:000> **!Name2EE *!program**
Module: 66ad1000 (mscorlib.dll)

Module: 72451000 (test.exe)
Token: 0x02000002
MethodTable: 72454c4c
EEClass: 7245146c
Name: program

```
-- EEClass can be used with !DumpClass to show attributes as well as list the
fields of the type, besides knowing information on fields

-- Parent class is System.Object
-- System.Object is the ultimate base class of all classes in the .NET
Framework; it is the root of the type hierarchy.
```

0:000> **!DumpClass 7245146c**
Class Name: program
mdToken: 02000002 (C:\cs\test.exe)
Parent Class: 66ad3ef0
Module: 72451000
Method Table: 72454c4c
Vtable Slots: 4
Total Method Slots: 5
Class Attributes: 100000
NumInstanceFields: 0
NumStaticFields: 0

0:000> **!DumpClass 66ad3ef0**
Class Name: System.Object
mdToken: 02000002
(C:\Windows\assembly\GAC_32\mscorlib\2.0.0.0__b77a5c561934e089\mscorlib.dll)
Parent Class: 00000000
Module: 66ad1000
Method Table: 66d4078c
Vtable Slots: 4
Total Method Slots: a
Class Attributes: 102001
NumInstanceFields: 0
NumStaticFields: 0

```
-- MethodTable can be used with !DumpMT to list of all the methods
defined on the object.
```

0:000> **!DumpMT -MD 72454c4c**
EEClass: 7245146c
Module: 72451000
Name: program
mdToken: 02000002 (C:\cs\test.exe)

BaseSize: 0xc
ComponentSize: 0x0
Number of IFaces in IFaceMap: 0
Slots in VTable: 6

MethodDesc Table
  Entry MethodDesc   JIT Name
66c96a90 66b1494c PreJIT System.Object.ToString()
66c96ab0 66b14954 PreJIT System.Object.Equals(System.Object)
66c96b20 66b14984 PreJIT System.Object.GetHashCode()
66d07540 66b149a8 PreJIT System.Object.Finalize()
72454c10 724514f4 PreJIT program..ctor()
72453040 **724514e0** PreJIT **program.Main**()

```
-- MethodDesc with !DumpIL prints the IL (Intermediate Language) code
associated with a managed method (we can almost see the .Net code that we
used in this program) (below should be read from top to down).
```

0:000> **!DumpIL 724514e0**
ilAddr = 72454be0
IL_0000: nop
IL_0001: ldstr "Hello World"
IL_0006: call System.Console::WriteLine
IL_000b: nop
IL_000c: call System.Console::ReadLine
IL_0011: pop
IL_0012: ret

```
-- MethodDesc with !U presents an annotated disassembly of a managed method
(below should be read from top to down)
```

0:000> **!U -gcinfo -ehinfo 724514e0**
preJIT generated code
program.Main()
Begin 72454c20, size 28
72454c20 55 push ebp
72454c21 8bec mov ebp,esp
72454c23 e8d8e3ffff call test_ni+0x3000 (72453000) (System.Console.get_Out(), mdToken: 06000773)
72454c28 8bc8 mov ecx,eax
72454c2a 8b0568144572 mov eax,dword ptr [test_ni+0x1468 (72451468)]
72454c30 8b10 mov edx,dword ptr [eax]
72454c32 8b01 mov eax,dword ptr [ecx]
72454c34 ff90d8000000 call dword ptr [eax+0D8h]
72454c3a e8c9e3ffff call test_ni+0x3008 (72453008) (System.Console.get_In(), mdToken: 06000772)
72454c3f 8bc8 mov ecx,eax
72454c41 8b01 mov eax,dword ptr [ecx]
72454c43 ff5064 call dword ptr [eax+64h]
72454c46 5d pop ebp

72454c47 c3 ret

```
-- Conversely, we can also get the MethodDesc and the method with !DumpStack
(to view the top of the stack) (displays current native frame, displays
MethodDesc, displays method), !CLRStack (displays method)

-- If you have a memory dump or break into a live process with windbg, the
frame ntdll!KiFastSystemCallRet will be on top of the stack.
```

0:000> **!DumpStack**
OS Thread Id: 0x9a8 (0)
Current frame: ntdll!KiFastSystemCallRet
ChildEBP RetAddr Caller,Callee
002aec98 77816464 ntdll!ZwRequestWaitReplyPort+0xc
002aec9c 777412e1 kernel32!ConsoleClientCallServer+0x88, calling ntdll!NtRequestWaitReplyPort
002aecbc 77750fe8 kernel32!ReadConsoleInternal+0x1ac, calling kernel32!ConsoleClientCallServer
...
002aef6c 6725de5d (MethodDesc 0x66be2924 +0x15 System.IO.TextReader+SyncTextReader.ReadLine())
002aef78 72454c46 (MethodDesc **0x724514e0** +0x26 **program.Main**())
...

0:000> **!clrstack**
OS Thread Id: 0x9a8 (0)
ESP EIP
002aeed8 778170b4 [NDirectMethodFrameStandaloneCleanup: 002aeed8] System.IO.__ConsoleStream.ReadFile(Microsoft.Win32.SafeHandles.SafeFileHandle, Byte*, Int32, Int32 ByRef, IntPtr)
002aeef4 6725aeab System.IO.__ConsoleStream.ReadFileNative(Microsoft.Win32.SafeHandles.SafeFileHandle, Byte[], Int32, Int32, Int32, Int32 ByRef)
002aef20 6725adde System.IO.__ConsoleStream.Read(Byte[], Int32, Int32)
002aef40 66cca05b System.IO.StreamReader.ReadBuffer()
002aef54 66cc9eac System.IO.StreamReader.ReadLine()
002aef74 6725de5d System.IO.TextReader+SyncTextReader.ReadLine()
002aef80 72454c46 **program.Main**()

```
-- Assembly translation of the specified program code in memory (current
frame ntdll!KiFastSystemCallRet) displays the instruction pointer and
displays the stack pointer
```

0:000> **u ntdll!KiFastSystemCallRet**
ntdll!KiFastSystemCallRet:
778170b4 c3 ret

0:000> **r**
eip=**778170b4** esp=**002aec9c**

Intro to WinDBG for .NET Developers - part IX (still more CLR internals - Memory Consumption)

C and C++ programs have traditionally been prone to memory leaks because developers had to manually allocate and free memory. In Microsoft® .NET (and thus in SQLCLR too), it is not necessary to do this because .NET uses garbage collection to automatically reclaim unused memory. This makes memory usage safer and more efficient.

.NET ACADEMY

Q & A:
Let's continue on the managed/.NET memory dump obtained in previous email.

What is a Memory leak?

A memory leak occurs when a computer program consumes memory but is unable to release it back to the operating system.

What is RAM, Virtual Memory, Pagefile and all that stuff?

Please refer http://support.microsoft.com/kb/2267427 (RAM, Virtual Memory, Pagefile and all that stuff) for more details.

What are common issues for high .Net memory usage?

Common issues for high .NET Memory usage includes: Blocked finalizers, lots of memory in cache/sessions, lots of large objects and memory rooted in statics.

How to debug a .Net application for memory consumption?

```
-->>>>> Find the total size of the dump file in Windows Explorer:

-- Full memory dump of our sample program is around 49,737 KB = 48.5 MB

-->>>>> Compare to amount of committed memory (MEM_COMMIT). This should be
almost equal to the total size of the dump file in Windows Explorer. This
includes both native and .Net allocations.
DebugDiag can be used for a more detailed native memory usage breakup:
```

0:000> **!address -summary**
ProcessParametrs 00391de8 in range 00390000 003fb000
Environment 003907f0 in range 00390000 003fb000
---------- Usage SUMMARY ----------
TotSize (KB) Pct(Tots) Pct(Busy) Usage
3355000 (52564) : 02.51% 53.85% : RegionUsageIsVAD

```
-- RegionUsageIsVAD includes other regions not included in below. .Net makes
allocations in this region.
```

7a09f000 (1999484) : 95.35% 00.00% : RegionUsageFree

```
-- RegionUsageFree mentions free region. Actual free space out of 2 GB
usermode.
```
271a000 (**40040**) : 01.91% 41.02% : RegionUsageImage
```
-- RegionUsageImage Region where Dll's/modules get loaded.
```
2fe000 (**3064**) : 00.15% 03.14% : RegionUsageStack
3000 (**12**) : 00.00% 00.01% : RegionUsageTeb
1e0000 (**1920**) : 00.09% 01.97% : RegionUsageHeap
```
-- RegionUsageHeap Native/Umanaged allocations here.
```
0 (**0**) : 00.00% 00.00% : RegionUsagePageHeap
1000 (**4**) : 00.00% 00.00% : RegionUsagePeb
0 (**0**) : 00.00% 00.00% : RegionUsageProcessParametrs
0 (**0**) : 00.00% 00.00% : RegionUsageEnvironmentBlock
Tot: 7fff0000 (2097088 KB) Busy: 05f51000 (97604 KB)
-------------------- Type SUMMARY -------------------------
TotSize (KB) Pct(Tots) Usage
7a09f000 (1999484) : 95.35% : <free>
271b000 (40044) : 01.91% : MEM_IMAGE
125f000 (18812) : 00.90% : MEM_MAPPED
25d7000 (38748) : 01.85% : MEM_PRIVATE
-------------------- State SUMMARY -------------------------
TotSize (KB) Pct(Tots) Usage
3093000 (**49740**) : 02.37% : **MEM_COMMIT**
7a09f000 (1999484) : 95.35% : MEM_FREE
2ebe000 (47864) : 02.28% : MEM_RESERVE
Largest free region: Base 03a50000 - Size 63080000 (**1622528** KB)
-- Within RegionUsageFree what matter is the Largest contiguous memory
block(1622528 KB) (indicated in 'Largest free region'). .Net needs minimum of
64 MB on contiguous memory block (32-bit OS). Less than this we are prone to
OutofMemoryException.

-- 'Usage SUMMARY' has a 'Tot' section which mentions 2097088 KB = ~2 GB
usermode VAS space. This indicates 32bit Windows without /3 GB switch.
-- MEM_COMMIT is the committed memory, this should be fairly close to Private
Bytes in Perfmon.
-- MEM_RESERVE + MEM_COMMIT is the total reserved bytes, so this is Virtual
Bytes in Perfmon.

-- 49740 KB = 48 MB
-- Breakup of this 48 MB usage = 40040 + 3064 + 12 + 1920 + 0 + 4 + 0 + 0 =
45040 = 43 MB. Remaining (48 MB - 43 MB) are the .Net and other allocations
in RegionUsageIsVAD.

-->>>>> Let's try to know the .Net allocations. Examine the GC Heap Size with
!eeheap -gc . Compare this size to the total size of the dump file.

-- "Number of GC Heaps: 1" indicates we have one thread running the GC. If
this was a single processor machine, running the GC in workstation mode we
would have seen the actual collection on a thread that triggered GC. But if
this is a multi processor machine with n heaps (we could see this from the
!eeheap -gc output), so we have n threads dedicated to doing garbage
collection.
```

0:000> **!EEHeap**
```

Number of GC Heaps: 1
generation 0 starts at 0x01a51018
generation 1 starts at 0x01a5100c
generation 2 starts at 0x01a51000
ephemeral segment allocation context: none
segment begin allocated size
01a50000 01a51000 01a55ff4 0x00004ff4(20468)
-- Objects larger than 85 KB (85,000 bytes) are automatically placed in the large object heap. And such objects would also be visible in !DumpHeap -stat
Large object heap starts at 0x02a51000
segment begin allocated size
02a50000 02a51000 02a53250 0x00002250(8784)
Total Size 0x7244(29252)

GC Heap Size 0x7244(**29252**)

-- GC Heap Size = 29252 = 28 KB.

-- The value of the total (.Net/CLR) committed bytes is slightly larger than actual Gen 0 heap size + Gen 1 heap size + Gen 2 heap size + Large Object Heap size

-->>>>> Use the !DumpHeap -stat command to discover which .NET managed objects are taking up space on the managed/GC Heap. Compare this value to the GC Heap Size.

-- Alternatively this can be manually calculated with just !DumpHeap

-- You need to know how long these objects will exist and how large they are. The !gcroot command shows that the System.Object [] objects have a pinned reference (strong reference would mean they are rooted for example due to class references).

0:000> **!DumpHeap -stat**
Statistics:
MT Count TotalSize Class Name
66d44044 1 12 System.Text.DecoderExceptionFallback
003c0c08 7 100 Free
...
66d417a0 25 2124 System.Char[]
66d40b70 127 4876 System.String
66d14340 24 **9412 System.Object**[]
Total 327 objects
-- "TotalSize" column represents the object size times the number of objects.
-- The objects with the largest TotalSize entries are at the bottom of the list.
-- The "Free" entry represents objects that are not referenced and have been garbage collected, but whose memory hasn't been compacted and released to the operating system yet.

0:000> **!DumpHeap -type System.Object**
Address MT Size
...

01a53934 66d4078c 12
02a51010 66d14340 4096
02a52020 66d14340 528
02a52240 66d14340 4096
total 34 objects
Statistics:
MT Count TotalSize Class Name
66d4078c 10 120 System.Object
66d14340 24 9412 System.Object[]
Total 34 objects

0:000> **!DumpHeap -type System.Object -min 1000**
Address MT Size
02a51010 66d14340 4096
02a52240 66d14340 4096
total 2 objects
Statistics:
MT Count TotalSize Class Name
66d14340 2 8192 System.Object[]
Total 2 objects

0:000> **!GCRoot 02a52240**
Note: Roots found on stacks may be false positives. Run "!help gcroot" for
more info.
Scan Thread 0 OSTHread 9a8
Scan Thread 2 OSTHread 854
DOMAIN(003B9238):HANDLE(**Pinned**):1413f0:Root:02a52240(**System.Object**[])

0:000> **!Name2EE *!System.Object**
Module: 66ad1000 (mscorlib.dll)
Token: 0x02000002
MethodTable: **66d4078c**
EEClass: 66ad3ef0
Name: System.Object

Module: 72451000 (test.exe)

0:000> **!DumpHeap -mt 66d4078c**
Address MT Size
01a5118c 66d4078c 12
01a519b4 66d4078c 12
01a51d9c 66d4078c 12
01a51e6c 66d4078c 12
01a52508 66d4078c 12
01a52674 66d4078c 12
01a526e0 66d4078c 12
01a52dd8 66d4078c 12
01a5391c 66d4078c 12
01a53934 66d4078c 12
total 10 objects

```
Statistics:
    MT    Count    TotalSize Class Name
66d4078c      10         120 System.Object
Total 10 objects

0:000> !objsize 01a5118c
sizeof(01a5118c) =         12 (       0xc) bytes (System.Object)

-- If objects are not rooted (not "Strong"), then they may require
finalization. For example the stack of the Finalize thread may indicate its
blocked (e.g. if the finalize method for my class calls Sleep for some reason
and thus blocks the finalizer).

-- Below displays number of objects pending finalization, in each generation

0:000> !FinalizeQueue
SyncBlocks to be cleaned up: 0
MTA Interfaces to be released: 0
STA Interfaces to be released: 0
---------------------------------
generation 0 has 5 finalizable objects (003c98e8->003c98fc)
generation 1 has 0 finalizable objects (003c98e8->003c98e8)
generation 2 has 0 finalizable objects (003c98e8->003c98e8)
Ready for finalization 0 objects (003c98fc->003c98fc)
Statistics:
    MT    Count    TotalSize Class Name
66d44a8c       1         20 Microsoft.Win32.SafeHandles.SafeFileMappingHandle
66d44a34       1         20 Microsoft.Win32.SafeHandles.SafeViewOfFileHandle
66d3eb3c       2         40 Microsoft.Win32.SafeHandles.SafeFileHandle
66d41144       1         56 System.Threading.Thread
Total 5 objects

0:000> !threads -special
ThreadCount: 2
UnstartedThread: 0
BackgroundThread: 1
PendingThread: 0
DeadThread: 0
Hosted Runtime: no
                            PreEmptive   GC Alloc         Lock
    ID OSID ThreadOBJ    State    GC      Context      Domain   Count APT Exception
  0    1  9a8 003bd5d8     a020 Enabled  01a54698:01a55fe8 003b9238     1 MTA
  2    2  854 003ca1b0     b220 Enabled  00000000:00000000 003b9238     0 MTA (Finalizer)

    OSID     Special thread type
  1    bdc    DbgHelper
  2    854    Finalizer

0:000> ~2s;k
eax=00000000 ebx=015bf98c ecx=003b0a10 edx=00000001 esi=00000002 edi=00000000
```

eip=778170b4 esp=015bf93c ebp=015bf9d8 iopl=0 nv up ei pl zr na pe nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00000246
ntdll!KiFastSystemCallRet:
778170b4 c3 ret
ChildEBP RetAddr
015bf938 77816a04 ntdll!KiFastSystemCallRet
015bf93c 75c069dc ntdll!ZwWaitForMultipleObjects+0xc
015bf9d8 7773bc8e KERNELBASE!WaitForMultipleObjectsEx+0x100
015bfa20 7773bcfc kernel32!WaitForMultipleObjectsExImplementation+0xe0
015bfa3c 6801a5c6 kernel32!WaitForMultipleObjects+0x18
015bfa5c 6801e596 **mscorwks!WKS::WaitForFinalizerEvent**+0x7a
...
015bfbc4 680f7e9a mscorwks!WKS::GCHeap::FinalizerThreadStart+0xbb
...
015bfccc 00000000 ntdll!_RtlUserThreadStart+0x1b

Additional information (whitepapers available online)
==========
Production Debugging for .NET Framework Applications
.NET Framework Internals: How the CLR Creates Runtime Objects
CLR Inside Out: Investigating Memory Issues

Intro to WinDBG for .NET Developers - part X (still more CLR internals - Memory/VAS Fragmentation)

Applications that are free from memory leaks but perform dynamic memory allocation and de-allocation frequently tend to show gradual performance degradation if they are kept running for long periods. Finally, they crash. Why is this? Recurrent allocation and de-allocation of dynamic memory can cause the heap to become fragmented.

.NET ACADEMY

Q & A:
Let's continue on the managed/.NET memory dump obtained in previous email.

What is a heap?

The heap is the section of computer memory where all the variables created or initialized at runtime are stored.

How many heaps are present in a process?

Every process in Windows has one heap called the default heap. Processes can also have as many other dynamic heaps as they wish, simply by creating and destroying them on the fly (for example using HeapCreate function).
The system uses the default heap for all global and local memory management functions, and the C run-time library uses the default heap for supporting malloc functions.

What is Virtual Address Space (VAS)?

The virtual address space for a process is the set of virtual memory addresses that it can use. The address space for each process is private and cannot be accessed by other processes unless it is shared.
A virtual address does not represent the actual physical location of an object in memory; instead, the system maintains a page table for each process, which is an internal data structure used to translate virtual addresses into their corresponding physical addresses.

How to check for fragmentation in a memory dump?

The amount of fragmentation for a managed heap is indicated by how much space free objects take on the heap.
If the largest free block of virtual memory (indicated in !address) for the process is less than 64MB on a 32-bit OS (1GB on 64-bit), the OOM could be caused by running out of virtual memory. (On a 64-bit OS, it is unlikely that the application will run out of virtual memory space.)
The process can run out of virtual space if virtual memory is overly fragmented. It's not common for the managed heap to cause fragmentation of virtual memory, but it can happen.
The !eeheap –gc SOS command will show you where each garbage collection segment starts. You can correlate this with the output of !address to determine if the virtual memory is fragmented by the managed heap.

0:002> **!address -summary**

Largest free region: Base 03a50000 - Size **63080000** (1622528 KB)
-- 1622528 KB

0:002> **? 63080000**
Evaluate expression: 1661468672 = 63080000
-- 1661468672 bytes = 1622528 KB

-- Traverse the managed heap. The amount of fragmentation for a managed heap is indicated by how much space free objects take on the heap.

0:000> **!DumpHeap -type Free -stat**
total 7 objects
Statistics:
MT Count TotalSize Class Name
003c0c08 **7** **100** Free
Total 7 objects

--In this sample, the output indicates that there are 7 free objects for a total of about 100bytes. Thus, the fragmentation for this heap is 100bytes.

/*
"Free" objects are simply regions of space the garbage collector can use later.
If 30% or more of the heap contains "Free" objects, the process may suffer from
heap fragmentation. This is usually caused by pinning objects for a long time combined with a high rate of allocation.

When you try to determine if fragmentation is a problem, you need to understand what fragmentation means for different generations. For Gen 0, it is not a problem because the GC can allocate in the fragmented space. For Gen 1 and Gen 2, fragmentation can be a problem. In order to use the fragmented space in Gen 1 and Gen 2, the GC would have to collect and promote objects in order to fill those gaps. But since Gen 1 cannot be bigger than one segment, Gen 2 is what you typically need to worry about. Excessive pinning is usually the cause of too much fragmentation.

*/

-- !gchandles checks the Number of Pinned Handles. A lot of work was done to reduce fragmentation caused by pinning. But you still may see high levels of fragmentation if the app simply pins too much.

0:000> **!GCHandles**
GC Handle Statistics:
Strong Handles: 14
Pinned Handles: 4
Async Pinned Handles: 0
Ref Count Handles: 0
Weak Long Handles: 0
Weak Short Handles: 1
Other Handles: 0
Statistics:

```
    MT    Count    TotalSize Class Name
66d4078c       1           12 System.Object
66d4123c       1           28 System.SharedStatics
66d420bc       2           48 System.Reflection.Assembly
66d40eb4       1           72 System.ExecutionEngineException
66d40e24       1           72 System.StackOverflowException
66d40d94       1           72 System.OutOfMemoryException
66d41350       1          100 System.AppDomain
66d41144       2          112 System.Threading.Thread
66d42834       4          144 System.Security.PermissionSet
66d40f44       2          144 System.Threading.ThreadAbortException
66d14340       3         8720 System.Object[]
Total 19 objects
```

Intro to WinDBG for .NET Developers - part XI (still more CLR internals - Value types)

Every variable and constant has a type, as does every expression that evaluates to a value. The .NET Framework class library defines a set of built-in types as well as more complex types that represent a wide variety of logical constructs, such as the file system, network connections, collections and arrays of objects, and dates.

.NET ACADEMY

Q & A:
Let's continue on the managed/.NET memory dump obtained in previous email.

What is a variable?
A variable is a symbolic name associated with a value.
A variable name in computer source code is usually associated with a data storage location and thus also its contents, and these may change during the course of program execution.

What's the relation between "variables" in computer science and "types" in CLR/.Net?
Every variable has a type.

What are the difference types of "types" in CLR/.Net?
Each type in the CTS is defined as either a value type or a reference type. This includes all custom types in the .NET Framework class library and also your own user-defined types. Types that you define by using the struct keyword are value types; all the built-in numeric types are structs. Types that you define by using the class keyword are reference types. Reference types and value types have different compile-time rules, and different run-time behavior.

Sample program?

```
using System;

public class Beach11{

      // Signed Integers
      static sbyte s_myInt8 = 0;
      static short s_myInt16 = 1;
      static int s_myInt32 = 2;
      static long s_myInt64 = 4;

      // Unsigned integers
      static byte s_uInt8 = 0;
      static ushort s_uInt16 = 1;
      static uint s_uInt32 = 2;
      static ulong s_uInt64 = 3;

      static int s_myHex = 0xF;

      // Decimal

      static float s_myFloat = 5.05F;
      static double s_myDouble = 5.05;
```

```
static decimal s_myDecimal = 5.05M;

static char s_c = 't';
static bool s_b = true;

sbyte myInt8;
short myInt16;
int myInt32;
long myInt64;

byte uInt8;
ushort uInt16;
uint uInt32;
ulong uInt64;

int myHex;

float myFloat;
double myDouble;
decimal myDecimal;

char c;
bool b;

static void Main(){

      s_myInt8 = 0;
      s_myInt16 = 1;
      s_myInt32 = 2;
      s_myInt64 = 3;
      s_uInt8 = 4;
      s_uInt16 = 5;
      s_uInt32 = 6;
      s_uInt64 = 7;
      s_myHex = 0xF;
      s_myFloat = 5.05F;
      s_myDouble = 5.05;
      s_myDecimal = 5.05M;
      s_c = 'v';
      s_b = true;

      Beach11 b11 = new Beach11();

      b11.myInt8 = 10;
      b11.myInt16 = 11;
      b11.myInt32 = 12;
      b11.myInt64 = 13;
      b11.uInt8 = 14;
      b11.uInt16 = 15;
      b11.uInt32 = 16;
      b11.uInt64 = 17;
      b11.myHex = 0xF;
      b11.myFloat = 15.05F;
      b11.myDouble = 15.05;
      b11.myDecimal = 15.05M;

      b11.c = 'n';
```

```
            b11.b = false;

            Console.WriteLine("Waiting (can take a memory dump now).");
            Console.ReadLine();
        }

}
```

Debugging?

```
-- Get class details
```

0:000> **!DumpDomain**
Assembly: 00459c20 [C:\cs\test2.exe]
ClassLoader: 00459ca0
SecurityDescriptor: 0045aad0
 Module Name
00142c5c C:\cs\test2.exe

0:000> **!DumpModule -mt 00142c5c**
Types defined in this module
    MT   TypeDef Name

001431b0 0x02000002 **Beach11**

0:000> **!DumpMT 001431b0**
EEClass: **001412d0**

0:000> **!DumpClass 001412d0**
Class Name: Beach11
mdToken: 02000002 (C:\cs\test2.exe)
Parent Class: 66933ef0
Module: 00142c5c
Method Table: 001431b0
Vtable Slots: 4
Total Method Slots: 6
Class Attributes: 100001
NumInstanceFields: e
NumStaticFields: e

MT	Field	**Offset**	Type	**VT**	Attr	**Value**	Name
66b9f070	400000f	**32**	System.SByte	1	**instance**		myInt8
66b9eb94	4000010	2c	System.Int16	1	instance		myInt16
66ba2dbc	4000011	1c	System.Int32	1	instance		myInt32
66ba23fc	4000012	4	System.Int64	1	instance		myInt64
66ba3690	4000013	33	System.Byte	1	instance		uInt8
66b9e978	4000014	2e	System.UInt16	1	instance		uInt16
66b787a0	4000015	20	System.UInt32	1	instance		uInt32
66b79474	4000016	c	System.UInt64	1	instance		uInt64
66ba2dbc	4000017	24	System.Int32	1	instance		myHex
66b9a3b4	4000018	28	System.Single	1	instance		myFloat
66b74700	4000019	14	System.Double	1	instance		myDouble

66b77f08 400001a   38   System.Decimal 1 instance   myDecimal
66ba1850 400001b   30   System.Char 1 instance   c
66b7463c 400001c   34   System.Boolean 1 instance   b
66b9f070 4000001   34   System.SByte 1   **static**   0 s_myInt8
66b9eb94 4000002   38   System.Int16 1   static   1 s_myInt16
66ba2dbc 4000003   3c   System.Int32 1   static   2 s_myInt32
66ba23fc 4000004   1c   System.Int64 1   static 3 s_myInt64
66ba3690 4000005   40   System.Byte 1   static   4 s_uInt8
66b9e978 4000006   44   System.UInt16 1   static   5 s_uInt16
66b787a0 4000007   48   System.UInt32 1   static   6 s_uInt32
66b79474 4000008   24   System.UInt64 1   static 7 s_uInt64
66ba2dbc 4000009   4c   System.Int32 1   static   15 s_myHex
66b9a3b4 400000a   50   System.Single 1   static 5.050000 s_myFloat
66b74700 400000b   2c   System.Double 1   static 5.050000 s_myDouble
66b77f08 400000c   4   System.Decimal 1   static 01b0263c s_myDecimal
66ba1850 400000d   54   System.Char 1   static   76 s_c
66b7463c 400000e   58   System.Boolean 1   static   1 s_b

0:000> **!DumpHeap -type Beach11 -short**
01b02778
0:000> !DumpHeap -type Beach11
Address   MT   Size
01b02778 001431b0   76
total 1 objects
Statistics:
      MT   Count   TotalSize Class Name
001431b0   1   76 Beach11
Total 1 objects

```
-- Examine the static and instance values of an object
```

0:000> **!do 01b02778**
Name: Beach11
MethodTable: 001431b0
EEClass: 001412d0
Size: 76(0x4c) bytes
(C:\cs\test2.exe)
Fields:
      MT   Field   Offset   Type VT   Attr   Value Name
66b9f070 400000f   32   System.SByte 1 **instance**   **10** myInt8
66b9eb94 4000010   2c   System.Int16 1 instance   11 myInt16
66ba2dbc 4000011   1c   System.Int32 1 instance   12 myInt32
66ba23fc 4000012   4   System.Int64 1 instance 13 myInt64
66ba3690 4000013   33   System.Byte 1 instance   14 uInt8
66b9e978 4000014   2e   System.UInt16 1 instance   15 uInt16
66b787a0 4000015   20   System.UInt32 1 instance   16 uInt32
66b79474 4000016   c   System.UInt64 1 instance 17 uInt64
66ba2dbc 4000017   24   System.Int32 1 instance   15 myHex
66b9a3b4 4000018   28   System.Single 1 instance 15.050000 myFloat
66b74700 4000019   14   System.Double 1 instance 15.050000 myDouble

66b77f08 400001a 38 System.Decimal 1 instance 01b027b0 myDecimal
66ba1850 400001b 30 System.Char 1 instance 6e c
66b7463c 400001c 34 System.Boolean 1 instance 0 b
66b9f070 4000001 34 System.SByte 1 static 0 s_myInt8
66b9eb94 4000002 38 System.Int16 1 static 1 s_myInt16
66ba2dbc 4000003 3c System.Int32 1 static 2 s_myInt32
66ba23fc 4000004 1c System.Int64 1 static 3 s_myInt64
66ba3690 4000005 40 System.Byte 1 static 4 s_uInt8
66b9e978 4000006 44 System.UInt16 1 static 5 s_uInt16
66b787a0 4000007 48 System.UInt32 1 static 6 s_uInt32
66b79474 4000008 24 System.UInt64 1 static 7 s_uInt64
66ba2dbc 4000009 4c System.Int32 1 static 15 s_myHex
66b9a3b4 400000a 50 System.Single 1 static 5.050000 s_myFloat
66b74700 400000b 2c System.Double 1 static 5.050000 s_myDouble
66b77f08 400000c 4 System.Decimal 1 static 01b0263c s_myDecimal
66ba1850 400000d 54 System.Char 1 static 76 s_c
66b7463c 400000e 58 System.Boolean 1 static 1 s_b

Intro to WinDBG for .NET Developers - part XII (still more CLR internals - String/Reference types)

One area likely to cause confusion for those coming from a VB6 or Java background is the distinction between value types and reference types in CLR. This article explores their essential differences when programming in CLR/.Net.

.NET ACADEMY

Q & A:
Let's continue on the managed/.NET memory dump obtained in previous email.

What are some examples of reference types?

Built-in reference types include dynamic, object and string. Keywords class, interface, delegate are used to declare reference types.

What are reference types?
What are objects in programming?

Variables of reference types, referred to as objects, store references to the actual data (store the reference so do not store the actual data).

When a value-type instance is created, a single space in memory is allocated to store the value. Primitive types such as int, float, bool and char are value types, and work in the same way. When the runtime deals with a value type, it's dealing directly with its underlying data and this can be very efficient, particularly with primitive types.
With reference types, however, an object is created in memory, and then handled through a separate reference—rather like a pointer.

Programming example?

```
using System;

public class Beach12{

	static string s_str;
	string i_str;

	static void Main(){

		s_str = "This is a test.\nThis too is a test.";

	Beach12 b12 = new Beach12();

		b12.i_str = "This is a string of an instance.";

		Console.WriteLine("Waiting (can take a memory dump now).");
		Console.ReadLine();
	}
```

```
}
```

Debugging?

```
-- Get the class
```

0:003> **!DumpDomain**
Module Name
00192c5c C:\cs\test2.exe

0:003> **!DumpModule -mt 00192c5c**
Types defined in this module
MT TypeDef Name

00193034 0x02000002 **Beach12**

```
-- Get the string locations for the class object(s)
```

0:003> **!DumpHeap -type Beach12**
Address MT Size
01ec2760 00193034 12
total 1 objects
Statistics:
MT Count TotalSize Class Name
00193034 1 12 Beach12
Total 1 objects

0:003> **!do 01ec2760**
Name: Beach12
MethodTable: 00193034
EEClass: 0019125c
Size: 12(0xc) bytes
(C:\cs\test2.exe)
Fields:
MT Field **Offset** Type VT Attr **Value** Name
66200b70 4000002 **4** System.String 0 instance **01ec26b0** i_str
66200b70 4000001 4 System.String 0 static **01ec2658** s_str

```
-- Multiple ways to get value of string variables

-- I am using poi command. From the above result we can make out the test
variable is in the 4 offset and that's the reason for using poi(${$obj}+0x4)
-- poi denotes pointer-sized data from the specified address of the target
computer/application.
-- m_arrayLength member, the allocated size of the string (33).
-- Next 4 bytes is the m_stringLength member, the actual number of characters
in the string.
-- Next 4 bytes store the string, starting at m_firstChar.
```

0:003> **!do poi(01ec2760+0x4)**
Name: System.String

MethodTable: 66200b70
EEClass: 65fbd66c
Size: 82(0x52) bytes
(C:\Windows\assembly\GAC_32\mscorlib\2.0.0.0__b77a5c561934e089\mscorlib.dll)
String: This is a string of an instance.
Fields:
    MT   Field   Offset   Type VT   Attr   Value Name
66202dbc 4000096   4   System.Int32 1 instance   33 m_arrayLength
66202dbc 4000097   8   System.Int32 1 instance   32 m_stringLength
66201850 4000098   c   System.Char 1 instance   54 m_firstChar
66200b70 4000099   10   System.String 0   shared   static Empty
  >> Domain:Value 003b19b0:01ec1198 <<
662017a0 400009a   14   System.Char[] 0   shared   static WhitespaceChars
  >> Domain:Value 003b19b0:01ec16d8 <<

0:003> **!do 01ec26b0**
Name: System.String
MethodTable: 66200b70
EEClass: 65fbd66c
Size: 82(0x52) bytes
(C:\Windows\assembly\GAC_32\mscorlib\2.0.0.0__b77a5c561934e089\mscorlib.dll)
String: This is a string of an instance.
Fields:
    MT   Field   Offset   Type VT   Attr   Value Name
66202dbc 4000096   4   System.Int32 1 instance   33 m_arrayLength
66202dbc 4000097   8   System.Int32 1 instance   32 m_stringLength
66201850 4000098   c   System.Char 1 instance   54 m_firstChar
66200b70 4000099   10   System.String 0   shared   static Empty
  >> Domain:Value 003b19b0:01ec1198 <<
662017a0 400009a   14   System.Char[] 0   shared   static WhitespaceChars
  >> Domain:Value 003b19b0:01ec16d8 <<

0:003> **!do 01ec2658**
Name: System.String
MethodTable: 66200b70
EEClass: 65fbd66c
Size: 88(0x58) bytes
(C:\Windows\assembly\GAC_32\mscorlib\2.0.0.0__b77a5c561934e089\mscorlib.dll)
String: This is a test.
This too is a test.
Fields:
    MT   Field   Offset   Type VT   Attr   Value Name
66202dbc 4000096   4   System.Int32 1 instance   36 m_arrayLength
66202dbc 4000097   8   System.Int32 1 instance   35 m_stringLength
66201850 4000098   c   System.Char 1 instance   54 m_firstChar
66200b70 4000099   10   System.String 0   shared   static Empty
  >> Domain:Value 003b19b0:01ec1198 <<
662017a0 400009a   14   System.Char[] 0   shared   static WhitespaceChars
  >> Domain:Value 003b19b0:01ec16d8 <<

```
-- Dump all strings in the memory dump
-- .printf "\n%mu" prints a Unicode array of chars terminated in NULL
-- ${obj}+c = address of the first char in the string
```

.foreach (obj {!DumpHeap -type System.String -short}) {.printf "\n%mu",${obj}+c}

Intro to WinDBG for .NET Developers - part XIII (still more CLR internals - Operators)

We all like to *express* ourselves in different ways. Some smile, some laugh, some screech and some even shout. C# provides a large set of operators, which are symbols that specify which operations to perform in a *software expression*. In addition, many operators can be overloaded by the user, thus changing their meaning when applied to a user-defined type.

.NET ACADEMY

Q & A:
Let's continue on the managed/.NET memory dump obtained in previous email.

What are operators in programming?

In its simplest meaning in mathematics and logic, an operation is an action or procedure which produces a new value from one or more input values. There are two common types of operations: unary and binary. Unary operations involve only one value. Binary operations, on the other hand, take two values.

Programming languages generally support a set of **operators** that are similar to operations in mathematics.

What are some examples of operators in CLR/.Net?

C# provides a large set of operators, which are symbols that specify which operations to perform in a software expression. These include ==, !=, <, >, <=, >=, binary +, binary -, ^, &, |, ~, ++, --, sizeof(), etc.

Sample program?

```
using System;

public class Beach13{

      bool bl = ("Goa" == "aaa");   // false

      float x = 3 / 2;
      float y = 3 / (float) 2;

      int iand = 5 & 3;        // 101 & 011 = 001 (1)
      int ior = 5 | 3;         // 101 | 011 = 111 (7)
      int ixor = 5 ^ 3;        // 101 ^ 011 = 110 (6)
      int ileftshift = 5 << 1;       // 101 << 1 = 1010 (10)
      int irightshift = 5 >> 1;      // 101 >> 1 = 10 (2)
      int icompliment = ~5;          // ~00000101 = 11111010 (-6)

      static void Main(){

            Beach13 b13 = new Beach13();
```

```
            Console.WriteLine("Waiting (can take a memory dump now).");
            Console.ReadLine();
        }

}

/*

Note: ~ operator / Two's complement:

Binary numbers do not have signs. So 2's complement is used to represent
negative numbers.
The two's complement of a binary number is defined as the value obtained by
subtracting the number from a large power of two (specifically, from 2N for
an N-bit two's complement).
The two's-complement system has the advantage of not requiring that the
addition and subtraction circuitry examine the signs of the operands to
determine whether to add or subtract.
With only one exception, when we start with any number in two's-complement
representation, if we flip all the bits and add 1, we get the two's-
complement representation of the negative of that number.

For example, two's complement of 00000101 (5) is calculated as below:
00000101 flipped = 11111010
11111010 + 00000001 = 11111011
11111011 = -128 + 64 + 32 + 16 + 8 + 0 + 2 + 1 = -128+123 = -5

In our example, 11111010 is already a two's complement (with the ~ operator):
11111010:
-128 + 64 + 32 + 16 + 8 + 0 + 2 + 0
= -128 + 122
= -6

*/
```

Debugging?

```
-- Get the class and its values
```

0:003> **!DumpDomain**
Module Name
00162c5c C:\cs\test2.exe

0:003> **!DumpModule -mt 00162c5c**
Types defined in this module
MT TypeDef Name

0016307c 0x02000002 **Beach13**

0:003> **!DumpHeap -type Beach13**
Address MT Size
01fb26b4 0016307c 44
total 1 objects

Statistics:
    MT   Count   TotalSize Class Name
0016307c   1   44 Beach13
Total 1 objects

0:003> **!do 01fb26b4**
Fields:
    MT   Field   Offset   Type VT   Attr   **Value Name**
66d5463c 4000001   24   System.Boolean 1 instance   **0 bl**
66d7a3b4 4000002   4   System.Single 1 instance **1.000000 x**
66d7a3b4 4000003   8   System.Single 1 instance **1.500000 y**
66d82dbc 4000004   c   System.Int32 1 instance   **1 iand**
66d82dbc 4000005   10   System.Int32 1 instance   **7 ior**
66d82dbc 4000006   14   System.Int32 1 instance   **6 ixor**
66d82dbc 4000007   18   System.Int32 1 instance   **10 ileftshift**
66d82dbc 4000008   1c   System.Int32 1 instance   **2 irightshift**
66d82dbc 4000009   20   System.Int32 1 instance   **-6 icompliment**

Intro to WinDBG for .NET Developers - part XIV (still more CLR internals - Arrays)

Arrays are among the oldest and most important data structures, and are used by almost every program and are used to implement many other data structures, such as lists and strings. They effectively exploit the addressing logic of computers. In most modern computers and many external storage devices, the memory is a one-dimensional array of words, whose indices are their addresses. Processors, especially vector processors, are often optimized for array operations.

.NET ACADEMY

Q & A:
Let's continue on the managed/.NET memory dump obtained in previous email.

What is an array?

An array is a data structure that contains several variables of the same type. An array is declared with a type: type[] arrayName.

How many types of arrays?

Three types of arrays - single-dimensional, multidimensional, and jagged arrays.

What's the advantage of arrays?

Arrays guarantee constant time read and write access, O(1) {O(1) denotes constant}. Lookup operations of an instance of an element are linear time, O(n) {O(n) denotes linear}. Arrays are very efficient in most languages, as operations compute the address of an element via a simple formula based on the base address element of the array.

Sample program?

```
using System;

public class Beach14{

      public Beach14(){
            array_jagged[0] = new int[1]{0}  ;
            array_jagged[1] = new int[]{ 1, 2};
      }

      // Declare and set a single-dimensional array
      int[] array_single = new int[] {1, 5, 10};      // int[3]

      // Declare and set a multi-dimensional (rectangular) array
      int[,] array_multi = { {1, 2, 3}
                           , {3, 4, 5}};      // int [2, 3]

      // Declare and set a jagged array
      int[][] array_jagged = new int[2][];

      // Another array
```

```
        string[,] array_string = { { "aa", "ab" }
                                 , { "ba", "bb" }};    // string [2, 2]

        static void Main(){

              Beach14 b14 = new Beach14();

              Console.WriteLine("Waiting (can take a memory dump now).");
              Console.ReadLine();
        }

}
```

Debugging?

```
-------------------- Get the variable location(s)/addresse(s) on the heap for
the instance(s)
```

0:003> **!DumpDomain**
Module Name
001e2c5c C:\cs\test2.exe

0:003> **!DumpModule -mt 001e2c5c**
Types defined in this module
MT TypeDef Name

001e306c 0x02000002 **Beach14**
001e3148 0x02000003 <PrivateImplementationDetails>{AF7D8548-56D2-4B81-8245-A2335A7D351E}
001e30c8 0x02000004 <PrivateImplementationDetails>{AF7D8548-56D2-4B81-8245-A2335A7D351E}+__StaticArrayInitTypeSize=12
001e3108 0x02000005 <PrivateImplementationDetails>{AF7D8548-56D2-4B81-8245-A2335A7D351E}+__StaticArrayInitTypeSize=24

0:003> **!DumpHeap -type Beach14**
Address MT **Size**
01d426b4 001e306c **24**
total 1 objects
Statistics:
MT Count TotalSize Class Name
001e306c 1 24 Beach14
Total 1 objects

```
-- Below indicates MethodTable = 001e306c; Beach14 instance variable(s)
located at 01d4272c, 01d42744, 01d42778, 01d42790.
```

0:003> **dd 01d426b4**
01d426b4 **001e306c 01d4272c 01d42744 01d42778**
01d426c4 **01d42790 80000000** 63590b70 00000003
01d426d4 00000002 00610061 00000000 80000000
01d426e4 63590b70 00000003 00000002 00620061
01d426f4 00000000 80000000 63590b70 00000003

01d42704 00000002 00610062 00000000 80000000
01d42714 63590b70 00000003 00000002 00620062
01d42724 00000000 00000000 63592d0c 00000003

-- OR

0:003> **!do 01d426b4**
Name: Beach14
MethodTable: 001e306c
EEClass: 001e12d4
Size: 24(0x18) bytes
(C:\cs\test2.exe)
Fields:

MT	Field	Offset		Type	VT	Attr	Value	Name
63592d0c	**4000001**	**4**		**System.Int32[]**	0	**instance**	**01d4272c**	**array_single**
00000000	**4000002**	**8**		**ARRAY**	0	**instance**	**01d42744**	**array_multi**
6357ea08	**4000003**	**c**		**System.Int32[][]**	0	**instance**	**01d42778**	**array_jagged**
00000000	**4000004**	**10**		**ARRAY**	0	**instance**	**01d42790**	**array_string**

-------------------- FIRST VARIABLE array_single (01d4272c)

-- Let's dump the memory where the instance is located
-- I have inserted brackets for the 24 bytes
-- The first four bytes are the reference to the method table at 63592d0c
-- Then next four bytes for the Length of the array (3 elements)
-- The next three dwords (each dword is 4 bytes which is the size of our int on this system) are the actual values of the array {1, 5, 10} as hex values. For example, hex 0xa = decimal 10
-- The last four bytes are null

-- Below indicates MethodTableAddress = 63592d0c, NumberOfArrayElements = 00000003, FirstValueOfThisSingleDimensionalArray = 00000001

0:003> **dd 01d4272c**
01d4272c [**63592d0c 00000003 00000001 00000005**
01d4273c **0000000a 00000000**] 63ba4e24 00000006
01d4274c 00000002 00000003 00000000 00000000
01d4275c 00000001 00000002 00000003 00000003
01d4276c 00000004 00000005 00000000 6357ea08
01d4277c 00000002 63375f16 01d427c0 01d427d0
01d4278c 00000000 6357e910 00000004 63590b70
01d4279c 00000002 00000002 00000000 00000000

-- Below indicates an Int32 array

0:003> **!DumpMT 63592d0c**
EEClass: 6334e6a8
Module: 63321000
Name: **System.Int32[]**

-- Below indicates storage of 24 bytes required for this array and its descriptors

0:003> **!ObjSize 01d4272c**
sizeof(01d4272c) = **24** (0x18) **bytes** (**System.Int32[]**)

```
-- Can directly dump only the values
```

0:003> **dd 01d42734**
01d42734 **00000001 00000005 0000000a 00000000**

0:003> **dd 01d42738**
01d42738 **00000005 0000000a 00000000** 63ba4e24

```
-- Hex a = decimal 10
```

0:003> **.formats 0000000a**
Evaluate expression:
Hex: 0000000a
Decimal: 10

```
---------- OR

-- !DumpArray command allows you to examine elements of an array object.
-- -details asks the command to print out details of the element using
!DumpObj and !DumpVC format.

-- Below tells us that it is our Int32 array with 3 elements and a total size
of 24 bytes
```

0:003> **!DumpArray 01d4272c**
Name: System.Int32[]
MethodTable: 63592d0c
Size: 24(0x18) **bytes**
Array: Rank 1, **Number of elements 3, Type Int32**
[0] 01d42734
[1] 01d42738
[2] 01d4273c

0:003> **!DumpArray -details 01d4272c**
Name: System.Int32[]
MethodTable: 63592d0c
Size: 24(0x18) **bytes**
Array: Rank 1, **Number of elements 3**, Type Int32
[0] 01d42734
  Name: System.Int32
  MethodTable 63592dbc
  EEClass: 6334e71c
  Size: 12(0xc) bytes
  Fields:
    MT Field Offset Type VT Attr **Value** Name
  63592dbc 40003f0 0 System.Int32 1 instance **1** m_value
[1] 01d42738

```
  Fields:
      MT    Field   Offset          Type VT     Attr    Value Name
  63592dbc  40003f0      0       System.Int32  1 instance       5 m_value
[2] 01d4273c
  Fields:
      MT    Field   Offset          Type VT     Attr    Value Name
  63592dbc  40003f0      0       System.Int32  1 instance      10 m_value

-------------------- SECOND VARIABLE array_multi (01d42744)

-- Below indicates MethodTableAddress = 63ba4e24, NumberOfArrayElements = 6,
FirstValueOfThisMultiDimensionalArray = 00000001

0:003> dd 01d42744
01d42744  63ba4e24 00000006 00000002 00000003
01d42754  00000000 00000000 00000001 00000002
01d42764  00000003 00000003 00000004 00000005
01d42774  00000000 6357ea08 00000002 63375f16
01d42784  01d427c0 01d427d0 00000000 6357e910
01d42794  00000004 63590b70 00000002 00000002
01d427a4  00000000 00000000 01d426cc 01d426e4
01d427b4  01d426fc 01d42714 00000000 63592d0c

0:003> !ObjSize 01d42744
sizeof(01d42744) =        52 (      0x34) bytes (System.Int32[,])

-- Dump memory starting from the first value.

0:003> dd 01d4275c
01d4275c  00000001 00000002 00000003 00000003
01d4276c  00000004 00000005 00000000 6357ea08
01d4277c  00000002 63375f16 01d427c0 01d427d0
01d4278c  00000000 6357e910 00000004 63590b70
01d4279c  00000002 00000002 00000000 00000000
01d427ac  01d426cc 01d426e4 01d426fc 01d42714
01d427bc  00000000 63592d0c 00000001 00000000
01d427cc  00000000 63592d0c 00000002 00000001

---------- OR

0:003> !DumpArray -details 01d42744
Name: System.Int32[,]
MethodTable: 63ba4e24
Size: 52(0x34) bytes
Array: Rank 2, Number of elements 6, Type Int32
[0][0] 01d4275c
  Fields:
      MT    Field   Offset          Type VT     Attr    Value Name
  63592dbc  40003f0      0       System.Int32  1 instance       1 m_value
[0][1] 01d42760
```

Fields:
MT Field Offset Type VT Attr **Value** Name
63592dbc 40003f0 0 System.Int32 1 instance **2** m_value
[0][2] 01d42764
Fields:
MT Field Offset Type VT Attr **Value** Name
63592dbc 40003f0 0 System.Int32 1 instance **3** m_value
[1][0] 01d42768
Fields:
MT Field Offset Type VT Attr **Value** Name
63592dbc 40003f0 0 System.Int32 1 instance **3** m_value
[1][1] 01d4276c
Fields:
MT Field Offset Type VT Attr **Value** Name
63592dbc 40003f0 0 System.Int32 1 instance **4** m_value
[1][2] 01d42770
Fields:
MT Field Offset Type VT Attr **Value** Name
63592dbc 40003f0 0 System.Int32 1 instance **5** m_value

```
-------------------- THIRD VARIABLE array_jagged (01d42778)

-- Below indicates MethodTableAddress = 6357ea08, NumberOfArrayElements = 2,
FirstAddressOfThisJaggedArray = 01d427c0
```

0:003> **dd 01d42778**
01d42778 [**6357ea08 00000002** 63375f16 **01d427c0**
01d42788 **01d427d0** 00000000] 6357e910 00000004
01d42798 63590b70 00000002 00000002 00000000
01d427a8 00000000 01d426cc 01d426e4 01d426fc
01d427b8 01d42714 00000000 63592d0c 00000001
01d427c8 00000000 00000000 63592d0c 00000002
01d427d8 00000001 00000002 00000000 6359078c
01d427e8 00000000 40010000 6358eb3c 00000007

0:003> **!ObjSize 01d42778**
sizeof(01d42778) = **60** (0x3c) **bytes** (**System.Int32[][]**)

```
-- MethodTableAddress = 63592d0c, NumberOfThisArrayElements = 1,
ElementValue(s) = 0
```

0:003> **dd 01d427c0**
01d427c0 **63592d0c 00000001 00000000** 00000000
01d427d0 63592d0c 00000002 00000001 00000002
01d427e0 00000000 6359078c 00000000 40010000
01d427f0 6358eb3c 00000007 00000004 00000100
01d42800 00000000 63594568 00000000 00000000
01d42810 00000000 00000000 00000001 01d427f0
01d42820 00000100 00000000 63593eb8 00000000
01d42830 00000000 00000000 00000000 00000001

```
-- MethodTableAddress = 63592d0c, NumberOfThisArrayElements = 2,
ElementValue(s) = {1, 2}
```

0:003> **dd 01d427d0**
01d427d0 **63592d0c 00000002 00000001 00000002**
01d427e0 00000000 6359078c 00000000 40010000
01d427f0 6358eb3c 00000007 00000004 00000100
01d42800 00000000 63594568 00000000 00000000
01d42810 00000000 00000000 00000001 01d427f0
01d42820 00000100 00000000 63593eb8 00000000
01d42830 00000000 00000000 00000000 00000001
01d42840 00000000 635935e0 00000000 00000000

```
---------- OR
```

0:003> **!DumpArray -details 01d42778**
Name: System.Int32[][][]
MethodTable: 6357ea08
Size: 24(0x18) bytes
Array: Rank 1, Number of elements 2, Type SZARRAY
Element Methodtable: 63375f16
[0] **01d427c0**
  Name: System.Int32[]
  MethodTable: 63592d0c
  EEClass: 6334e6a8
  Size: 16(0x10) bytes
  Array: Rank 1, Number of elements 1, Type Int32
  Element Type: System.Int32
  Fields:
  None
[1] **01d427d0**
  Name: System.Int32[]
  MethodTable: 63592d0c
  EEClass: 6334e6a8
  Size: 20(0x14) bytes
  Array: Rank 1, Number of elements 2, Type Int32
  Element Type: System.Int32
  Fields:
  None

```
-- Dump first (jagged) array element
```

0:003> **!DumpArray -details 01d427c0**
Name: System.Int32[]
MethodTable: 63592d0c
EEClass: 6334e6a8
Size: 16(0x10) bytes
Array: Rank 1, Number of elements 1, Type Int32
Element Methodtable: 63592dbc
[0] 01d427c8

Name: System.Int32
MethodTable 63592dbc
EEClass: 6334e71c
Size: 12(0xc) bytes
Fields:
MT Field Offset Type VT Attr **Value** Name
63592dbc 40003f0 0 System.Int32 1 instance **0** m_value

```
-- Dump second (jagged) array element
```

0:003> **!DumpArray -details 01d427d0**
Name: System.Int32[]
MethodTable: 63592d0c
EEClass: 6334e6a8
Size: 20(0x14) bytes
Array: Rank 1, Number of elements 2, Type Int32
Element Methodtable: 63592dbc
[0] 01d427d8
Name: System.Int32
MethodTable 63592dbc
EEClass: 6334e71c
Size: 12(0xc) bytes
Fields:
MT Field Offset Type VT Attr **Value** Name
63592dbc 40003f0 0 System.Int32 1 instance **1** m_value
[1] 01d427dc
Name: System.Int32
MethodTable 63592dbc
EEClass: 6334e71c
Size: 12(0xc) bytes
Fields:
MT Field Offset Type VT Attr **Value** Name
63592dbc 40003f0 0 System.Int32 1 instance **2** m_value

```
-------------------- FOURTH VARIABLE array_string (01d42790)

-- Below indicates MethodTableAddress = 6357e910, NumberOfArrayElements = 4,
FirstAddressOfThisStringArray = 01d426cc
```

0:003> **dd 01d42790**
01d42790 [**6357e910 00000004** 63590b70 00000002
01d427a0 00000002 00000000 00000000 **01d426cc**
01d427b0 **01d426e4 01d426fc 01d42714** 00000000]
01d427c0 63592d0c 00000001 00000000 00000000
01d427d0 63592d0c 00000002 00000001 00000002
01d427e0 00000000 6359078c 00000000 40010000
01d427f0 6358eb3c 00000007 00000004 00000100
01d42800 00000000 63594568 00000000 00000000

0:003> **!ObjSize 01d42790**
sizeof(01d42790) = **144** (0x90) **bytes** (**System.Object[,]**)

```
-- Below for binary values for 1st string value
```

0:003> **dd 01d426cc**
01d426cc 63590b70 00000003 00000002 00610061
01d426dc 00000000 80000000 63590b70 00000003
01d426ec 00000002 00620061 00000000 80000000
01d426fc 63590b70 00000003 00000002 00610062
01d4270c 00000000 80000000 63590b70 00000003
01d4271c 00000002 00620062 00000000 00000000
01d4272c 63592d0c 00000003 00000001 00000005
01d4273c 0000000a 00000000 63ba4e24 00000006

```
-- Below for ASCII characters for 1st string value
-- ASCII conversion:
-- 70 = p, 0b = <whitespace>, 59 = Y, 63 = c, 00 = <null>, 61 = a, 62 = b
```

0:003> **db 01d426cc**
01d426cc 70 0b 59 63 03 00 00 00-02 00 00 00 **61** 00 **61** 00 p.Yc.....**a**.**a**.
01d426dc 00 00 00 00 00 00 00 80-70 0b 59 63 03 00 00 00 ....p.Yc....
01d426ec 02 00 00 00 **61** 00 **62** 00-00 00 00 00 00 00 00 80 ....**a**.**b**.....
01d426fc 70 0b 59 63 03 00 00 00-02 00 00 00 **62** 00 **61** 00 p.Yc.....**b**.**a**.
01d4270c 00 00 00 00 00 00 00 80-70 0b 59 63 03 00 00 00 ....p.Yc....
01d4271c 02 00 00 00 **62** 00 **62** 00-00 00 00 00 00 00 00 00 ....**b**.**b**.....
01d4272c 0c 2d 59 63 03 00 00 00-01 00 00 00 05 00 00 00 .-Yc............
01d4273c 0a 00 00 00 00 00 00 00-24 4e ba 63 06 00 00 00 ....$N.c....

```
-- Another way to dump 1st string value
```

0:003> **!do -nofields 01d426cc**
Name: System.String
MethodTable: 63590b70
EEClass: 6334d66c
Size: 22(0x16) bytes
String: **aa**

```
---------- OR
```

0:003> **!DumpArray -details -nofields 01d42790**
Name: System.String[,]
MethodTable: 6357e910
EEClass: 63345d20
Size: 48(0x30) bytes
Array: Rank 2, Number of elements 4, Type CLASS
Element Methodtable: 63590b70
[0][0] 01d426cc
  Name: System.String
  MethodTable: 63590b70

EEClass: 6334d66c
Size: 22(0x16) bytes
String: **aa**
[0][1] 01d426e4
Name: System.String
MethodTable: 63590b70
EEClass: 6334d66c
Size: 22(0x16) bytes
String: **ab**
[1][0] 01d426fc
Name: System.String
MethodTable: 63590b70
EEClass: 6334d66c
Size: 22(0x16) bytes
String: **ba**
[1][1] 01d42714
Name: System.String
MethodTable: 63590b70
EEClass: 6334d66c
Size: 22(0x16) bytes
String: **bb**

Intro to WinDBG for .NET Developers - part XV (still more CLR internals - Conditionals)

In previous topics, every program you saw contained a limited amount of sequential steps and then stopped. There were no decisions you could make with the input and the only constraint was to follow straight through to the end. The information here will help you branch into separate logical sequences based on decisions you make.

.NET ACADEMY

Q & A:
Let's continue on the managed/.NET memory dump obtained in previous email.

What are Conditional / Branching / Selection / Control statements?

These are statement that cause program control to be transferred to a specific flow based upon whether a certain condition is true or not.

The following keywords are used in such statements:

- if
- else
- switch
- case
- default.

Sample program?

```
using System;

public class Beach15{

	static void fn_if(){
		int x = 0;
		if (x == 1)
			Console.WriteLine(1);
	}

	static void fn_switch(){
		int x = 0;
		switch(x){
			case 1:
				Console.WriteLine(1);
				break;
		}
	}

	static void fn_ternary_op(){
		int x = 0;
		Console.WriteLine("{0}", (x == 1) ? 1 : 0);
	}

	static void Main(){

		Beach15 b15 = new Beach15();
```

```
        // Force JIT for !UBeach15.fn_if();
            Beach15.fn_switch();
            Beach15.fn_ternary_op();

            Console.WriteLine("Waiting (can take a memory dump now).");
            Console.ReadLine();
        }

}
```

Debugging?

```
-- Obtain the custom methods
```

0:003> **!DumpDomain**
Module Name
00462c5c C:\cs\test2.exe

0:003> **!DumpModule -mt 00462c5c**
Types defined in this module
MT TypeDef Name

00463034 0x02000002 Beach15

0:003> **!DumpMT -MD 00463034**
EEClass: 0046126c
Module: 00462c5c
Name: Beach15
mdToken: 02000002 (C:\cs\test2.exe)
BaseSize: 0xc
ComponentSize: 0x0
Number of IFaces in IFaceMap: 0
Slots in VTable: 9

MethodDesc Table
Entry MethodDesc JIT Name
63e76a90 63cf494c PreJIT System.Object.ToString()
63e76ab0 63cf4954 PreJIT System.Object.Equals(System.Object)
63e76b20 63cf4984 PreJIT System.Object.GetHashCode()
63ee7540 63cf49a8 PreJIT System.Object.Finalize()
0046c021 0046302c NONE Beach15..ctor()
0046c011 **00462ffc** NONE Beach15.**fn_if**()
0046c015 **00463008** NONE Beach15.**fn_switch**()
0046c019 **00463014** NONE Beach15.**fn_ternary_op**()
004e0070 00463020 JIT Beach15.Main()

```
-- Assembly language code associated with fn_if, fn_switch
-- fn_if, fn_switch have similar assembly code like below
-- No program action required within these straight forward functions
```

0:003> **!U 00462ffc**
Normal JIT generated code
Beach15.fn_if()
Begin 0046c011, size 5
0046c011 55 push ebp
0046c012 8bec mov ebp,esp
0046c014 5d pop ebp
0046c015 c3 ret

```
-- Assembly language code associated with fn_ternary_op
```

0:003> **!U 00463014**
Normal JIT generated code
Beach15.fn_ternary_op()
Begin 0046c019, size 34
0046c019 55 push ebp
0046c01a 8bec mov ebp,esp
0046c01c 57 push edi
0046c01d 56 push esi
0046c01e 8b3d3420fa02 mov edi,dword ptr ds:[2FA2034h] ("{0}")
0046c024 b9bc2d9d63 mov ecx,offset mscorlib_ni+0x272dbc (639d2dbc) (MT: System.Int32)
0046c029 e8f71ef1ff call 0033201c (JitHelp: CORINFO_HELP_NEWSFAST)
0046c02e 8bd0 mov edx,eax
0046c030 33c9 xor ecx,ecx
0046c032 894a04 mov dword ptr [edx+4],ecx
0046c035 8bf2 mov esi,edx
0046c037 e8cdd25663 call mscorlib_ni+0x22d400 (6398d400) (System.Console.get_Out(), mdToken: 06000773)
0046c03c 56 push esi
0046c03d 8bc8 mov ecx,eax
0046c03f 8bd7 mov edx,edi
0046c041 8b01 mov eax,dword ptr [ecx]
0046c047 ff90e0000000 call dword ptr [eax+0E0h]
0046c048 5e pop esi
0046c049 5f pop edi
0046c04a 5d pop ebp
0046c04b c3 ret

```
// Intermediate language code associated with fn_if()
```

0:003> **!DumpIL 00462ffc**
ilAddr = 01082050
IL_0000: nop // Automatically generated breakpoint in debug build. Consumes a processing cycle
IL_0001: ldc.i4.0 // Load/push int32 value 0 onto the stack
IL_0002: stloc.0 // Pop value from the stack and place in 0th local variable (x = 0)
IL_0003: ldloc.0 // Load 0th local variable onto the stack
IL_0004: ldc.i4.1 // Load/push int32 value 1 onto the stack
IL_0005: ceq // Compare equality of top two stack variables. If equal, 1 is pushed onto stack. Otherwise 0 is pushed onto stack (Checks the x == 1 condition)

IL_0007: ldc.i4.0 // Load/push int32 value 0 onto the stack
IL_0008: ceq // Checks the if condition (unoptimized codegen is designed to be clear, unambiguous, easy to debug)
IL_000a: stloc.1
IL_000b: ldloc.1
IL_000c: brtrue.s IL_0015 // Branch to IL_0015 if non-zero (true)
IL_000e: ldc.i4.1
IL_000f: call System.Console::WriteLine
IL_0014: nop
IL_0015: ret

```
// Intermediate language code associated with fn_switch()
```

0:003> **!DumpIL 00463008**
ilAddr = 01082074
IL_0000: nop
IL_0001: ldc.i4.0
IL_0002: stloc.0 // x = 0
IL_0003: ldloc.0
IL_0004: stloc.1
IL_0005: ldloc.1
IL_0006: ldc.i4.1
IL_0007: beq.s IL_000b // Transfer control if top two stack values are equal
IL_0009: br.s IL_0014
IL_000b: ldc.i4.1
IL_000c: call System.Console::WriteLine
IL_0011: nop
IL_0012: br.s IL_0014
IL_0014: ret

```
// Intermediate language code associated with fn_ternary_op()
```

0:003> **!DumpIL 00463014**
ilAddr = 01082098
IL_0000: nop
IL_0001: ldc.i4.0
IL_0002: stloc.0
IL_0003: ldstr "{0}"
IL_0008: ldloc.0
IL_0009: ldc.i4.1
IL_000a: beq.s IL_000f
IL_000c: ldc.i4.0
IL_000d: br.s IL_0010
IL_000f: ldc.i4.1
IL_0010: box System.Int32 // Convert value type to an object reference. For example, conversion from int to string in the WriteLine
IL_0015: call System.Console::WriteLine
IL_001a: nop
IL_001b: ret

Intro to WinDBG for .NET Developers - part XVI (still more CLR internals - Loops)

Another essential technique when writing software is looping - the ability to repeat a block of code X times. In CLR, they come in 4 different variants, and we will have a look at each one of them.

Q & A:
Let's continue on the managed/.NET memory dump obtained in previous email.

What are iteration statements in programming?

You can create loops by using the iteration statements. Iteration statements cause embedded statements to be executed a number of times, subject to the loop-termination criteria. These statements are executed in order, except when a jump statement is encountered.

The following keywords are used in iteration statements:

- do
- for
- foreach
- in
- while.

What are jump statements in programming?

Branching is performed using jump statements, which cause an immediate transfer of the program control. The following keywords are used in jump statements:

- break
- continue
- goto
- return
- throw.

Sample program?

```
using System;

public class Beach16{

	static void fn_while(){
		int x = 0;
		while (x < 10)
			Console.WriteLine(x++); // 0 to 9
	}

	static void fn_do_while(){
		int x = 0;
		do{
			Console.WriteLine(x++); // 0 to 9
		}while(x < 10);
```

```
        }

        static void fn_for(){
              int[] x = {0, 1, 2};
              for (int i = 0; i < x.Length; i++)
                    Console.WriteLine(x[i]);      // 0 to 2
        }

        static void fn_foreach(){
              int[] x = {0, 1, 2};
              foreach (int i in x)
                    Console.WriteLine(i);   // 0 to 2
        }

        static void Main(){

              Beach16 b16 = new Beach16();

              fn_while();
              fn_do_while();
              fn_for();
              fn_foreach();

              Console.WriteLine("Waiting (can take a memory dump now).");
              Console.ReadLine();
        }

}
```

Debugging?

```
-- Obtain the methods and their intermediate language byte code
```

0:003> **!DumpDomain**
Module Name
00342c5c C:\cs\test2.exe

0:003> **!DumpModule -mt 00342c5c**
Types defined in this module
MT TypeDef Name

00343040 0x02000002 **Beach16**

0:003> **!DumpMT -MD 00343040**
MethodDesc Table
Entry MethodDesc JIT Name
64546a90 643c494c PreJIT System.Object.ToString()
64546ab0 643c4954 PreJIT System.Object.Equals(System.Object)
64546b20 643c4984 PreJIT System.Object.GetHashCode()
645b7540 643c49a8 PreJIT System.Object.Finalize()
0034c025 00343038 NONE Beach16..ctor()
004d00c8 **00342ffc** JIT Beach16.**fn_while()**

004d0108 **00343008** JIT Beach16.**fn_do_while()**
004d0148 **00343014** JIT Beach16.**fn_for()**
004d01c0 **00343020** JIT Beach16.**fn_foreach()**
004d0070 0034302c JIT Beach16.Main()

0:003> **!DumpIL 00342ffc; !DumpIL 00343008; !DumpIL 00343014; !DumpIL 00343020**
ilAddr = 01332050 // fn_while()
IL_0000: nop // Automatic breakpoint in debug mode
IL_0001: ldc.i4.0
IL_0002: stloc.0 // Copy 0 to 0th variable
IL_0003: br.s IL_0010
IL_0005: ldloc.0
IL_0006: dup // Duplicate topmost stack value
IL_0007: ldc.i4.1
IL_0008: add
IL_0009: stloc.0
IL_000a: call System.Console::WriteLine
IL_000f: nop
IL_0010: ldloc.0
IL_0011: ldc.i4.s 10
IL_0013: clt // Compare values if first less than second
IL_0015: stloc.1
IL_0016: ldloc.1
IL_0017: brtrue.s IL_0105
IL_0019: ret
ilAddr = 01332078 // fn_do_while()
IL_0000: nop
IL_0001: ldc.i4.0
IL_0002: stloc.0
IL_0003: nop
IL_0004: ldloc.0
IL_0005: dup
IL_0006: ldc.i4.1
IL_0007: add
IL_0008: stloc.0
IL_0009: call System.Console::WriteLine
IL_000e: nop
IL_000f: nop
IL_0010: ldloc.0
IL_0011: ldc.i4.s 10
IL_0013: clt
IL_0015: stloc.1
IL_0016: ldloc.1
IL_0017: brtrue.s IL_0103
IL_0019: ret
ilAddr = 013320a0 // fn_for()
IL_0000: nop
IL_0001: ldc.i4.3
IL_0002: newarr System.Int32 // Push object reference to array

```
IL_0007: stloc.2
IL_0008: ldloc.2
IL_0009: ldc.i4.1
IL_000a: ldc.i4.1
IL_000b: stelem.i4     // Replace the array element with stack element
IL_000c: ldloc.2
IL_000d: ldc.i4.2
IL_000e: ldc.i4.2
IL_000f: stelem.i4
IL_0010: ldloc.2
IL_0011: stloc.0
IL_0012: ldc.i4.0
IL_0013: stloc.1
IL_0014: br.s IL_0023
IL_0016: ldloc.0
IL_0017: ldloc.1
IL_0018: ldelem.i4     // Load element (to stack)
IL_0019: call System.Console::WriteLine
IL_001e: nop
IL_001f: ldloc.1
IL_0020: ldc.i4.1
IL_0021: add
IL_0022: stloc.1
IL_0023: ldloc.1
IL_0024: ldloc.0
IL_0025: ldlen
IL_0026: conv.i4
IL_0027: clt
IL_0029: stloc.3
IL_002a: ldloc.3
IL_002b: brtrue.s IL_0116
IL_002d: ret
ilAddr = 013320dc      // fn_foreach()
IL_0000: nop
IL_0001: ldc.i4.3
IL_0002: newarr System.Int32
IL_0007: stloc.2
IL_0008: ldloc.2
IL_0009: ldc.i4.1
IL_000a: ldc.i4.1
IL_000b: stelem.i4
IL_000c: ldloc.2
IL_000d: ldc.i4.2
IL_000e: ldc.i4.2
IL_000f: stelem.i4
IL_0010: ldloc.2
IL_0011: stloc.0
IL_0012: nop
IL_0013: ldloc.0
IL_0014: stloc.3
```

IL_0015: ldc.i4.0
IL_0016: stloc.s VAR OR ARG 4
IL_0018: br.s IL_002c
IL_001a: ldloc.3
IL_001b: ldloc.s VAR OR ARG 4
IL_001d: ldelem.i4
IL_001e: stloc.1
IL_001f: ldloc.1
IL_0020: call System.Console::WriteLine
IL_0025: nop
IL_0026: ldloc.s VAR OR ARG 4
IL_0028: ldc.i4.1
IL_0029: add
IL_002a: stloc.s VAR OR ARG 4
IL_002c: ldloc.s VAR OR ARG 4
IL_002e: ldloc.3
IL_002f: ldlen
IL_0030: conv.i4
IL_0031: clt
IL_0033: stloc.s VAR OR ARG 5
IL_0035: ldloc.s VAR OR ARG 5
IL_0037: brtrue.s IL_011a
IL_0039: ret

0:003> **!U 00342ffc; !U 00343008; !U 00343014; !U 00343020**
Normal JIT generated code
Beach16.**fn_while()**
Begin 004d00c8, size 2e
004d00c8 55 push ebp
004d00c9 8bec mov ebp,esp
004d00cb 57 push edi
004d00cc 56 push esi
004d00cd 33f6 xor esi,esi
004d00cf eb14 jmp 004d00e5
004d00d1 8bfe mov edi,esi
004d00d3 46 inc esi
004d00d4 e827d30d64 call mscorlib_ni+0x22d400 (645ad400) (System.Console.get_Out(), mdToken: 06000773)
004d00d9 8bc8 mov ecx,eax
004d00db 8bd7 mov edx,edi
004d00dd 8b01 mov eax,dword ptr [ecx]
004d00df ff90bc000000 call dword ptr [eax+0BCh]
004d00e5 83fe0a cmp esi,0Ah
004d00e8 0f9cc0 setl al
004d00eb 0fb6c0 movzx eax,al
004d00ee 85c0 test eax,eax
004d00f0 75df jne 004d00d1
004d00f2 5e pop esi
004d00f3 5f pop edi
004d00f4 5d pop ebp

```
004d00f5 c3            ret
```
Normal JIT generated code
Beach16.**fn_do_while()**
```
Begin 004d0108, size 2c
004d0108 55            push    ebp
004d0109 8bec          mov     ebp,esp
004d010b 57            push    edi
004d010c 56            push    esi
004d010d 33ff          xor     edi,edi
004d010f 8bf7          mov     esi,edi
004d0111 47            inc     edi
004d0112 e8e9d20d64    call    mscorlib_ni+0x22d400 (645ad400) (System.Console.get_Out(),
mdToken: 06000773)
004d0117 8bc8          mov     ecx,eax
004d0119 8bd6          mov     edx,esi
004d011b 8b01          mov     eax,dword ptr [ecx]
004d011d ff90bc000000  call    dword ptr [eax+0BCh]
004d0123 83ff0a        cmp     edi,0Ah
004d0126 0f9cc0        setl    al
004d0129 0fb6c0        movzx   eax,al
004d012c 85c0          test    eax,eax
004d012e 75df          jne     004d010f
004d0130 5e            pop     esi
004d0131 5f            pop     edi
004d0132 5d            pop     ebp
004d0133 c3            ret
```
Normal JIT generated code
Beach16.**fn_for()**
```
Begin 004d0148, size 68
004d0148 55            push    ebp
004d0149 8bec          mov     ebp,esp
004d014b 57            push    edi
004d014c 56            push    esi
004d014d 53            push    ebx
004d014e ba03000000    mov     edx,3
004d0153 b9165f3d64    mov     ecx,offset mscorlib_ni+0x55f16 (643d5f16)
004d0158 e83b20d6ff    call    00232198 (JitHelp: CORINFO_HELP_NEWARR_1_VC)
004d015d 8bf0          mov     esi,eax
004d015f 837e0401      cmp     dword ptr [esi+4],1
004d0163 7645          jbe     004d01aa
004d0165 c7460c01000000  mov   dword ptr [esi+0Ch],1
004d016c 837e0402      cmp     dword ptr [esi+4],2
004d0170 7638          jbe     004d01aa
004d0172 c7461002000000  mov   dword ptr [esi+10h],2
004d0179 33ff          xor     edi,edi
004d017b eb1b          jmp     004d0198
004d017d 3b7e04        cmp     edi,dword ptr [esi+4]
004d0180 7328          jae     004d01aa
004d0182 8b5cbe08      mov     ebx,dword ptr [esi+edi*4+8]
```

```
004d0186 e875d20d64     call    mscorlib_ni+0x22d400 (645ad400) (System.Console.get_Out(),
mdToken: 06000773)
004d018b 8bc8           mov     ecx,eax
004d018d 8bd3           mov     edx,ebx
004d018f 8b01           mov     eax,dword ptr [ecx]
004d0191 ff90bc000000   call    dword ptr [eax+0BCh]
004d0197 47             inc     edi
004d0198 397e04         cmp     dword ptr [esi+4],edi
004d019b 0f9fc0         setg    al
004d019e 0fb6c0         movzx   eax,al
004d01a1 85c0           test    eax,eax
004d01a3 75d8           jne     004d017d
004d01a5 5b             pop     ebx
004d01a6 5e             pop     esi
004d01a7 5f             pop     edi
004d01a8 5d             pop     ebp
004d01a9 c3             ret
004d01aa e8fd2fdf65     call    mscorwks!JIT_RngChkFail (662c31ac)
004d01af cc             int     3
Normal JIT generated code
Beach16.fn_foreach()
Begin 004d01c0, size 68
004d01c0 55             push    ebp
004d01c1 8bec           mov     ebp,esp
004d01c3 57             push    edi
004d01c4 56             push    esi
004d01c5 53             push    ebx
004d01c6 ba03000000     mov     edx,3
004d01cb b9165f3d64     mov     ecx,offset mscorlib_ni+0x55f16 (643d5f16)
004d01d0 e8c31fd6ff     call    00232198 (JitHelp: CORINFO_HELP_NEWARR_1_VC)
004d01d5 8bf0           mov     esi,eax
004d01d7 837e0401       cmp     dword ptr [esi+4],1
004d01db 7645           jbe     004d0222
004d01dd c7460c01000000 mov     dword ptr [esi+0Ch],1
004d01e4 837e0402       cmp     dword ptr [esi+4],2
004d01e8 7638           jbe     004d0222
004d01ea c7461002000000 mov     dword ptr [esi+10h],2
004d01f1 33ff           xor     edi,edi
004d01f3 eb1b           jmp     004d0210
004d01f5 3b7e04         cmp     edi,dword ptr [esi+4]
004d01f8 7328           jae     004d0222
004d01fa 8b5cbe08       mov     ebx,dword ptr [esi+edi*4+8]
004d01fe e8fdd10d64     call    mscorlib_ni+0x22d400 (645ad400) (System.Console.get_Out(),
mdToken: 06000773)
004d0203 8bc8           mov     ecx,eax
004d0205 8bd3           mov     edx,ebx
004d0207 8b01           mov     eax,dword ptr [ecx]
004d0209 ff90bc000000   call    dword ptr [eax+0BCh]
004d020f 47             inc     edi
004d0210 397e04         cmp     dword ptr [esi+4],edi
```

```
004d0213 0f9fc0        setg    al
004d0216 0fb6c0        movzx   eax,al
004d0219 85c0          test    eax,eax
004d021b 75d8          jne     004d01f5
004d021d 5b            pop     ebx
004d021e 5e            pop     esi
004d021f 5f            pop     edi
004d0220 5d            pop     ebp
004d0221 c3            ret
004d0222 e8852fdf65    call    mscorwks!JIT_RngChkFail (662c31ac)
004d0227 cc            int     3
```

Intro to WinDBG for .NET Developers - part XVII (still more CLR internals - Methods)

Computer programs exist to solve problems and there are methods for solving those problems. So a method is a kind of a building block that solves a problem. Every executed instruction is done so in the context of a method.

.NET ACADEMY

Q & A:
Let's continue on the managed/.NET memory dump obtained in previous email.

What is a method / function / subroutine?

It is a code block containing a series of statements.
It's a portion of code within a larger program, performs a specific task.

What is method overloading?

In Java, .Net, etc., it is possible to define two or more methods within the same class that share the same name, as long as their parameter declarations are different. When this is the case, the methods are said to be **overloaded**, and the process is referred to as **method overloading**. Method overloading is one of the ways that languages implement polymorphism.

If you have never used a language that allows the overloading of methods, then the concept may seem strange at first. But as you will see, method overloading is one of the most exciting and useful features. When an overloaded method is invoked, CLR uses the type and/or number of arguments as its guide to determine which version of the overloaded method to actually call.

Thus, overloaded methods must differ in the type and/or number of their parameters. When CLR encounters a call to an overloaded method, it simply executes the version of the method whose parameters match the arguments used in the call.

Sample code?

```
using System;

public class Beach17{

	int i;

	Beach17() { i = 0; }

	Beach17(string str) { i = 1; }

	static void Main(){

		Beach17 b17_zero = new Beach17();
		Console.WriteLine("Value of i: " + b17_zero.i); // 0

		Beach17 b17_one = new Beach17("One string parameter");
		Console.WriteLine("Value of i: " + b17_one.i);  // 1
```

```
            Console.WriteLine("Waiting (can take a memory dump now).");
            Console.ReadLine();
        }

}
```

Debugging?

```
-- Obtain and dump the methods
```

0:003> **!DumpDomain**
Module Name
00262c5c C:\cs\test2.exe

0:003> **!DumpModule -mt 00262c5c**
Types defined in this module
MT TypeDef Name

00263028 0x02000002 **Beach17**

0:003> **!DumpMT -MD 00263028**
EEClass: 00261268
Module: 00262c5c
Name: Beach17
mdToken: 02000002 (C:\cs\test2.exe)
BaseSize: 0xc
ComponentSize: 0x0
Number of IFaces in IFaceMap: 0
Slots in VTable: 7

MethodDesc Table
Entry MethodDesc JIT Name
64546a90 643c494c PreJIT System.Object.ToString()
64546ab0 643c4954 PreJIT System.Object.Equals(System.Object)
64546b20 643c4984 PreJIT System.Object.GetHashCode()
645b7540 643c49a8 PreJIT System.Object.Finalize()
00500140 **00263008** JIT Beach17..**ctor()**
00500160 **00263010** JIT Beach17..**ctor(System.String)**
00500070 0026301c JIT Beach17.Main()

0:003> **!U 00263008; !U 00263010**
Normal JIT generated code
Beach17..**ctor()**
Begin 00500140, size 6
00500140 33d2 xor edx,edx
00500142 895104 mov dword ptr [ecx+4],edx // Move 0
00500145 c3 ret
Normal JIT generated code
Beach17..**ctor(System.String)**
Begin 00500160, size 8

00500160 c7410401000000 mov   dword ptr [ecx+4],1   // Move 1
00500167 c3   ret

0:003> **!DumpIL 00263008; !DumpIL 00263010**
ilAddr = 003a2050   // Default constructor - Beach17..ctor()
IL_0000: ldarg.0
IL_0001: call System.Object::.ctor
IL_0006: nop
IL_0007: nop
IL_0008: ldarg.0
IL_0009: ldc.i4.0
IL_000a: stfld Beach17::i
IL_000f: nop
IL_0010: ret
ilAddr = 003a2062   // Constructor with one parameter - Beach17..ctor(System.String)
IL_0000: ldarg.0
IL_0001: call System.Object::.ctor
IL_0006: nop
IL_0007: nop
IL_0008: ldarg.0
IL_0009: ldc.i4.1
IL_000a: stfld Beach17::i
IL_000f: nop
IL_0010: ret

Intro to WinDBG for .NET Developers - part XVIII (still more CLR internals - Classes)

When you write programs in CLR/.Net, all program data is wrapped in a class, whether it is a class you write or a class you use from the .Net Framework libraries.

.NET ACADEMY

Q & A:
Let's continue on the managed/.NET memory dump obtained in previous email.

What is a class in computer programming?

A class is a part of a computer program that a programmer creates to represent a thing in a way that a computer can understand. A class is written in a programming language and a programming language that can be used to write classes is called an Object-oriented programming language. Classes have fields, which represent a quality the thing has, and classes have methods, which represent what a thing can do.

A class is like a blueprint. It defines the data and behavior of a type.

A Wikipedia example: For example, a class could be a car, which could have a color field, four tire fields, and a drive method. Another, related class could be a truck, which would have similar fields, but not be exactly the same as a car. Both a car and a truck could be a kind of a third class which could be called a vehicle class. In this way, the programmer could create the parts of the program that are the same for both the car and the truck in programming the vehicle class, yet the programmer would be free to program how a car is different from a truck without duplicating all of the programming.

In this example, there are three classes: 1) a class called "car", 2) a class called "pick-up truck", and 3) a class called "vehicle".

An **instance** is an executable copy of a class. Another name for instance is **object**. There can be any number of objects of a given class in memory at any one time.

Sample program?

```
using System;

public class Beach18{

      int i = 0, j = 0;

      Beach18() : this (1, 1) {}

      Beach18(int i, int j) { this.i = 1; j = j; }

      static void Main(){

            Beach18 b18 = new Beach18();
            Console.WriteLine("Value of i: " + b18.i);      // 1
            Console.WriteLine("Value of j: " + b18.j);      // 0
```

```
            Console.WriteLine("Waiting (can take a memory dump now).");
            Console.ReadLine();
        }

}
```

Debugging?

```
-- Obtain the class, methods and variable values
```

0:003> **!DumpDomain**
Module Name
00292c5c C:\cs\test2.exe

0:003> **!DumpModule -mt 00292c5c**
Types defined in this module
MT TypeDef Name

00293034 0x02000002 **Beach18**
Types referenced in this module
MT TypeRef Name

63dd078c 0x01000001 System.Object
63dd2dbc 0x01000004 System.Int32
63dd0b70 0x01000005 System.String
63dd44dc 0x01000006 System.Console

```
-- Below can be dumped however these are internal implementations and can
change between .Net versions. So its better to use SOS-based commands since
they provide version-independent information about the common language
runtime (CLR) environment
```

0:003> **dd 00293034**
00293034 00000000 00000010 00050011 00000004 // 0x10 = BaseSize; 4
00293044 63dd078c 00292c5c 00293070 0029126c // 63dd078c = (First) type referenced in module, 00292c5c = ModuleAddress, 0029126c = EEClassAddress
00293054 00000000 00000000 63d26a90 63d26ab0 // 0, 0. 63d26a90, 63d26ab0, 63d26b20, 63d97540 = 4 EntryAddresses for methods
00293064 63d26b20 63d97540 0029c011 00000080
00293074 00000000 00000000 00000000 00000000
00293084 00000000 00000000 00000000 00000000
00293094 00000000 00000000 00000000 00000000
002930a4 00000000 00000000 00000000 00000000

0:003> **!EEVersion**
2.0.50727.5446 retail // SOS version
Workstation mode
SOS Version: 2.0.50727.5446 retail build

0:003> **!DumpMT -MD 00293034**

EEClass: 0029126c
Module: 00292c5c
Name: Beach18
mdToken: 02000002 (C:\cs\test2.exe)
BaseSize: 0x10
ComponentSize: 0x0
Number of IFaces in IFaceMap: 0
Slots in VTable: 7 `// 7 methods`

MethodDesc Table
  Entry MethodDesc JIT Name
63d26a90 63ba494c PreJIT System.Object.ToString()
63d26ab0 63ba4954 PreJIT System.Object.Equals(System.Object)
63d26b20 63ba4984 PreJIT System.Object.GetHashCode()
63d97540 63ba49a8 PreJIT System.Object.Finalize()
0029c011 00293014 NONE **Beach18..ctor()**
00420130 0029301c JIT **Beach18..ctor(Int32, Int32)**
00420070 00293028 JIT Beach18.Main()

0:003> **!DumpHeap -type Beach18**
Address MT Size
0195270c 00293034 16
total 1 objects
Statistics:
  MT Count TotalSize Class Name
00293034 1 16 Beach18
Total 1 objects

0:003> **!do 0195270c**
Name: Beach18
MethodTable: 00293034
EEClass: 0029126c
Size: 16(0x10) bytes
(C:\cs\test2.exe)
Fields:
  MT Field Offset Type VT Attr **Value Name**
63dd2dbc 4000001 4 System.Int32 1 instance **1 i** `// i = 1`
63dd2dbc 4000002 8 System.Int32 1 instance **0 j** `// j = 0`

Intro to WinDBG for .NET Developers - part XIX (still more CLR internals - Inheritance)

Different kinds of objects often have a certain amount in common with each other. Apples, Oranges, Papayas, for example, all share the characteristics of Fruits. Yet each also defines additional features that make them different.

In object-oriented programming languages, one of the earliest motivations for using inheritance was the reuse of code which already existed in another class.

.NET ACADEMY

Q & A:
Let's continue on the managed/.NET memory dump obtained in previous email.

What is inheritance in computer science?

Object-oriented programming allows classes to inherit commonly used state and behavior from other classes. For example, Fruit becomes a base/super class of Apples, Oranges, Papayas.

Explain the Inheritance Principle.

Inheritance is the process by which one object acquires the properties of another object.

Is multiple Inheritance possible in C#?

Like Java, C# does not support multiple inheritance, meaning that classes cannot inherit from more than one class. You can, however, use interfaces for that purpose in the same way as in Java.

Do structs support inheritance?

No, structs do not support inheritance, but they can implement interfaces.

What is the main advantage of using inheritance?

Code reuse. Reusability saves time in program development. It encourages the reuse of proven and debugged high-quality software, thus reducing problem after a system becomes functional.

Sample program?

```
using System;

// Base Class (parent class)
class Fruit{                          // Base Class automatically inherits from
System.Object

      public int quantity = 10;

      public int GetQuantity() { return quantity; }
}

// Derived Class (child class)
```

```
class Apple : Fruit{

      static void Main(){

            Apple a = new Apple();
            Fruit f = (Fruit) a;    // reverse (Fruit to Apple) will not work
(not every Apple is a Fruit)

            Apple aa = (Apple) f;   // possible since f is actually pointing
to an object (of Apple) which is of same/compatible type as aa

            if (aa is Apple)
                  Console.WriteLine ("aa is of type Apple.");     // aa is of
type Apple.
            if (aa is Fruit)
                  Console.WriteLine ("aa is of type Fruit.");     // aa is of
type Fruit.

            Console.WriteLine("Quantity: " + aa.GetQuantity());   // 10

            Console.WriteLine("Waiting (can take a memory dump now).");
            Console.ReadLine();

      }
}
```

Debugging?

```
-- Obtain types
```

0:003> **!DumpDomain**
Module Name
00242c5c C:\cs\test3.exe

0:003> **!DumpModule -mt 00242c5c**
Types defined in this module
MT TypeDef Name

00243024 0x02000002 **Fruit**
0024308c 0x02000003 **Apple**

```
-- Obtain methods of Fruit
```

0:003> **!DumpMT -MD 00243024**
EEClass: **00241280**
Module: 00242c5c
Name: Fruit
mdToken: 02000002 (C:\cs\test3.exe)
BaseSize: 0xc
ComponentSize: 0x0
Number of IFaces in IFaceMap: 0
Slots in VTable: 6

MethodDesc Table
  Entry MethodDesc   JIT Name
640a6a90   63f2494c   PreJIT System.Object.ToString()
640a6ab0   63f24954   PreJIT System.Object.Equals(System.Object)
640a6b20   63f24984   PreJIT System.Object.GetHashCode()
64117540   63f249a8   PreJIT System.Object.Finalize()
0024c015   0024301c   NONE Fruit..ctor()
0024c011   00243010   NONE **Fruit.GetQuantity**()

```
-- Obtain methods of Apple
```

0:003> **!DumpMT -MD 0024308c**
EEClass: **002412e4**
Module: 00242c5c
Name: Apple
mdToken: 02000003 (C:\cs\test3.exe)
BaseSize: 0xc
ComponentSize: 0x0
Number of IFaces in IFaceMap: 0
Slots in VTable: 6

MethodDesc Table
  Entry MethodDesc   JIT Name
640a6a90   63f2494c   PreJIT System.Object.ToString()
640a6ab0   63f24954   PreJIT System.Object.Equals(System.Object)
640a6b20   63f24984   PreJIT System.Object.GetHashCode()
64117540   63f249a8   PreJIT System.Object.Finalize()
0024c031   00243084   NONE Apple..ctor()
006c0070   00243078   JIT Apple.Main()

```
-- Obtain class heirarchy (base/derived) if any
```

0:003> **!DumpClass 00241280**
Class Name: Fruit
mdToken: 02000002 (C:\cs\test3.exe)
Parent Class: **63ee3ef0** `// Address of parent class of Fruit class`

0:003> **!DumpClass 63ee3ef0**
Class Name: **System.Object**
mdToken: 02000002
(C:\Windows\assembly\GAC_32\mscorlib\2.0.0.0__b77a5c561934e089\mscorlib.dll)
Parent Class: 00000000

0:003> **!DumpClass 002412e4**
Class Name: Apple
mdToken: 02000003 (C:\cs\test3.exe)
Parent Class: **00241280** `// Address of parent class of Apple class (which we already know as the EEClass Address of Fruit class)`

```
-- Obtain address of Apple instance/object
```

0:003> **!DumpHeap -type Apple**
Address MT Size
01e32754 0024308c 12
total 1 objects
Statistics:
MT Count TotalSize Class Name
0024308c 1 12 Apple
Total 1 objects

```
-- Dump variable value
```

0:003> **!do 01e32754**
Name: Apple
MethodTable: 0024308c
EEClass: 002412e4
Size: 12(0xc) bytes
(C:\cs\test3.exe)
Fields:
MT Field Offset Type VT Attr **Value Name**
64152dbc 4000001 4 System.Int32 1 instance **10 quantity**

Intro to WinDBG for .NET Developers - part XX (still more CLR internals - Method overriding)

When inheriting from another class, you may wish to change the default behavior of a method or create a different method signature. You can do this by overloading the method with your own code.

.NET ACADEMY

Q & A:
Let's continue on the managed/.NET memory dump obtained in previous email.

What is the difference between method overriding and method overloading?

An **overridden** method provides a new implementation of a member that is inherited from a base class. The method that is overridden by an override declaration is known as the overridden base method. The overridden base method must have the same signature as the override method (same parameters).

Method **overloading** allows a programmer to define several methods with the same name, as long as they take a different set of parameters.

Sample program?

```
using System;

// Base Class (parent class)
class Fruit{

      public int quantity = 10;

      public int QuantityMultiplied() { return quantity * 2; }

      public virtual int QuantityAdded() { return quantity + 50; }
}

// Derived Class (child class)
sealed class Apple : Fruit{

      public new int quantity = 2000;

      public new int QuantityMultiplied() { return base.quantity * 5; }

      public sealed override int QuantityAdded() { return base.quantity +
100; }

      static void Main(){

            Apple a = new Apple();

            Console.WriteLine("Quantity multiplied: " +
a.QuantityMultiplied());    // 50
```

```
            Console.WriteLine("Quantity added: " +
a.QuantityAdded());         // 110

            Fruit f = (Fruit) a;
            Console.WriteLine("Quantity multiplied: " +
f.QuantityMultiplied());    // 20
            Console.WriteLine("Quantity added: " +
f.QuantityAdded());         // 110

            Console.WriteLine("Waiting (can take a memory dump now).");
            Console.ReadLine();

      }
}
```

Debugging?

```
-- Dump the programming methods
```

0:003> **!DumpDomain**
Module Name
00432c5c C:\cs\test3.exe

0:003> **!DumpModule -mt 00432c5c**
Types defined in this module
MT TypeDef Name

0043302c 0x02000002 **Fruit**
004330b8 0x02000003 **Apple**

0:003> **!DumpMT -MD 0043302c**

MethodDesc Table
Entry MethodDesc JIT Name
65866a90 656e494c PreJIT System.Object.ToString()
65866ab0 656e4954 PreJIT System.Object.Equals(System.Object)
65866b20 656e4984 PreJIT System.Object.GetHashCode()
658d7540 656e49a8 PreJIT System.Object.Finalize()
0043c015 0043301c NONE **Fruit.QuantityAdded()**
0043c019 00433024 NONE Fruit..ctor()
0043c011 00433010 NONE **Fruit.QuantityMultiplied()**

0:003> **!DumpMT -MD 004330b8**

MethodDesc Table
Entry MethodDesc JIT Name
65866a90 656e494c PreJIT System.Object.ToString()
65866ab0 656e4954 PreJIT System.Object.Equals(System.Object)
65866b20 656e4984 PreJIT System.Object.GetHashCode()
658d7540 656e49a8 PreJIT System.Object.Finalize()
006601a0 0043309c JIT **Apple.QuantityAdded()**
0043c03d 004330b0 NONE Apple..ctor()

0043c031 00433090 NONE **Apple.QuantityMultiplied()**
00660070 004330a4 JIT Apple.Main()

0:003> **!DumpHeap -type Fruit**
Address MT Size
total **0** objects

0:003> **!DumpHeap -type Appl**
Address MT Size
01c92724 004330b8 16
total 1 objects

0:003> **!do 01c92724**
Fields:
MT Field Offset Type VT Attr **Value Name**
65912dbc 4000001 4 System.Int32 1 instance **10 quantity**
65912dbc 4000002 8 System.Int32 1 instance **2000 quantity**

Intro to WinDBG for .NET Developers - part XXI (still more CLR internals - Access Modifiers)

When choosing an access level, it is generally best to use the most restrictive level possible. This is because the more places a member can be accessed the more places it can be accessed incorrectly, which makes the code harder to debug.

.NET ACADEMY

Q & A:
Let's continue on the managed/.NET memory dump obtained in previous email.

What are Access Modifiers?

Access modifiers are keywords used to specify the declared accessibility of a member or a type. Access modifiers allow you to define who does or doesn't have access to certain features. **Four** access modifiers:

- public
- protected
- internal
- private.

The following **five** accessibility levels can be specified using the access modifiers:

- public: Access is not restricted.
- protected: Access is limited to the containing class or types derived from the containing class.
- Internal: Access is limited to the current assembly.
- protected internal: Access is limited to the current assembly or types derived from the containing class.
- private: Access is limited to the containing type.

What the default Access Modifiers for class and for types?

Classes and structs are declared as internal by default.

Types declared inside a class without an Access Modifier default to private.

Sample program?

```
using System;

public class Beach21_parent{

      private int i_private;
      protected int i_protected;
      protected internal int i_protected_internal;
      internal int i_internal;
      public int i_public;

}
```

```
public class Beach21_child : Beach21_parent{

      static void Main(){

            Beach21_child b21_child = new Beach21_child();
            // b21_child.i_private = 0;   Would give compiler error CS0122
that variable is inaccessible due to its (programming) protection level (even
though memory dump displays this variable)

            Console.WriteLine("Waiting (can take a memory dump now).");
            Console.ReadLine();
      }

}
```

Debugging?

```
-- Display the class fields
```

0:003> **!DumpDomain**
Module Name
00252c5c C:\cs\test2.exe

0:003> **!DumpModule -mt 00252c5c**
Types defined in this module
MT TypeDef Name

00253048 0x02000002 **Beach21_parent**
002530b0 0x02000003 **Beach21_child**

0:003> **!DumpMT -MD 00253048**
EEClass: **00251280**

0:003> **!DumpMT -MD 002530b0**
EEClass: **002512e4**

0:003> **!DumpClass 00251280**
NumInstanceFields: **5**
NumStaticFields: 0

MT	Field	Offset	Type	VT	Attr	Value	**Name**
64d92dbc	4000001	4	System.Int32	1	instance		**i_private**
64d92dbc	4000002	8	System.Int32	1	instance		**i_protected**
64d92dbc	4000003	c	System.Int32	1	instance		**i_protected_internal**
64d92dbc	4000004	10	System.Int32	1	instance		**i_internal**
64d92dbc	4000005	14	System.Int32	1	instance		**i_public**

```
-- Below displays the five variables for class Beach21_child (only four of
these are programatically accessible in this child class)
```

0:003> **!DumpClass 002512e4**
NumInstanceFields: **5**
NumStaticFields: 0

```
    MT    Field   Offset          Type VT     Attr    Value Name
64d92dbc  4000001      4       System.Int32  1 instance        i_private
64d92dbc  4000002      8       System.Int32  1 instance        i_protected
64d92dbc  4000003      c       System.Int32  1 instance        i_protected_internal
64d92dbc  4000004     10       System.Int32  1 instance        i_internal
64d92dbc  4000005     14       System.Int32  1 instance        i_public

0:003> !DumpHeap -type Beach21_parent
Address       MT     Size
total 0 objects

0:003> !DumpHeap -type Beach21_child
Address       MT     Size
018426b4 002530b0      28
total 1 objects

-- Below also displays the five variables for class Beach21_child (only four
of these are programmatically accessible in this child class)

0:003> !do 018426b4
Name: Beach21_child
MethodTable: 002530b0
EEClass: 002512e4
Size: 28(0x1c) bytes
(C:\cs\test2.exe)
Fields:
    MT    Field   Offset          Type VT     Attr    Value Name
64d92dbc  4000001      4       System.Int32  1 instance        0 i_private
64d92dbc  4000002      8       System.Int32  1 instance        0 i_protected
64d92dbc  4000003      c       System.Int32  1 instance        0 i_protected_internal
64d92dbc  4000004     10       System.Int32  1 instance        0 i_internal
64d92dbc  4000005     14       System.Int32  1 instance        0 i_public
```

Intro to WinDBG for .NET Developers - part XXII (still more CLR internals - Static)

Sometimes you will want to create behavior that is not linked to any individual object, instead being available to all instances and to objects of other classes. This is where static members become useful.

A good example of a static method is the Main method, which is used in every executable program. When a program is launched, no instances of any class are present in memory. As the Main method is static, it can be called without creating an object and can then assume control of the program. It is the Main method's task to create the objects that the program requires to function correctly.

Q & A:
Let's continue on the managed/.NET memory dump obtained in previous email.

What is a static/class member and what is an instance member?

A C# class can contain both static and non-static members. When we declare a member with the help of the keyword static, it becomes a **static member**. A static member belongs to the class rather than to the objects of the class. Hence static members are also known as **class members** and **non-static members** are known as **instance members**.

How to make a non-static class behave similar to a static class?

A private constructor prevents the class from being instantiated (from a different class).

What is a static constructor?

Static constructor is used to initialize static data members as soon as the class is referenced first time, whereas an instance constructor is used to create an instance of that class with <new> keyword. A static constructor does not take access modifiers or have parameters and can't access any non-static data member of a class (similar to a static class).

Sample application?

```
using System;

public static class Beach22{

    public const int i_const = 0;         // A constant expression is an
expression that can be fully evaluated at compile time
                                    // Can be directly replaced by the
compilers with the actual constant value during compile time
      public static int i_static = i_const;     // 0

      public static int Add (int x, int y){
            return (x + y);
      }

      static void Main(){
```

```
            // Beach22 b22 = new Beach22();      // Will give error since
static class cannot be instantiated

            Console.WriteLine("Adding two numbers: " + Beach22.Add(2,
3));     // 5

            Console.WriteLine("Waiting (can take a memory dump now).");
            Console.ReadLine();
      }

}
```

Debugging?

```
-- Obtain the static variable
```

0:003> **!DumpDomain**
Module Name
00242c5c C:\cs\test2.exe

0:003> **!DumpModule -mt 00242c5c**
Types defined in this module
MT TypeDef Name

0024302c 0x02000002 **Beach22**

0:003> **!DumpMT -MD 0024302c**
EEClass: **00241268**

MethodDesc Table
Entry MethodDesc JIT Name
63a76a90 638f494c PreJIT System.Object.ToString()
63a76ab0 638f4954 PreJIT System.Object.Equals(System.Object)
63a76b20 638f4984 PreJIT System.Object.GetHashCode()
63ae7540 638f49a8 PreJIT System.Object.Finalize()
00400070 00243024 JIT Beach22..cctor()
0024c011 0024300c NONE **Beach22.Add**(Int32, Int32)
00400090 00243018 JIT Beach22.Main()

0:003> **!DumpHeap -type Beach22**
Address MT Size
total **0** objects
Statistics:
MT Count TotalSize Class Name
Total 0 objects

0:003> **!DumpClass 00241268**
NumInstanceFields: 0
NumStaticFields: 1
MT Field Offset Type VT Attr **Value Name**

63b22dbc 4000002 1c System.Int32 1 static **0 i_static**

Intro to WinDBG for .NET Developers - part XXIII (still more CLR internals - Properties)

With a public field, the underlying data type must always be the same because calling code depends on the field being the same. However, with a property, you can change the implementation.

.NET ACADEMY

Q & A:
Let's continue on the managed/.NET memory dump obtained in previous email.

What are properties in C#?
What are accessors?

Properties allow you to control the accessibility of a classes variables, and is the recommended way to access variables from the outside in an object oriented programming language like C#.

A **property** is a member that provides a flexible mechanism to read, write, or compute the value of a private field. Properties can be used as if they are public data members, but they are actually special methods called **accessors** (get, set accessors). This enables data to be accessed easily and still helps promote the safety and flexibility of methods.

Properties are nothing but natural extension of data fields. A property is much like a combination of a variable and a method.

How to add a ReadOnly property in C#?

If a property does not have a set accessor, it becomes a ReadOnly property.

Sample program?

```
using System;

public class Beach23{

	private int _quantity = 0;

	public int quantity{
		get { return _quantity;}
		set{
			if (value > 0)
				_quantity += value;
			else
				incorrect--;
		}
	}

	public int incorrect = 0;

	static void Main(){

		Beach23 b23 = new Beach23();
```

```
            b23.quantity += 5;      // 5

            Console.WriteLine("Waiting (can take a memory dump now).");
            Console.ReadLine();
        }

}
```

Debugging?

```
-- Obtain field values
```

0:003> **!DumpDomain**
Module Name
00252c5c C:\cs\test2.exe

0:003> **!DumpModule -mt 00252c5c**
Types defined in this module
MT TypeDef Name

00253040 0x02000002 **Beach23**

0:003> **!DumpMT -MD 00253040**

MethodDesc Table
Entry MethodDesc JIT Name
63e96a90 63d1494c PreJIT System.Object.ToString()
63e96ab0 63d14954 PreJIT System.Object.Equals(System.Object)
63e96b20 63d14984 PreJIT System.Object.GetHashCode()
63f07540 63d149a8 PreJIT System.Object.Finalize()
0025c01d 00253038 NONE Beach23..ctor()
0025c011 00253014 NONE **Beach23.get_quantity**()
003d00c8 00253020 JIT **Beach23.set_quantity**(Int32)
003d0070 0025302c JIT Beach23.Main()

0:003> **!DumpHeap -type Beach23**
Address MT Size
01ac26b4 00253040 16
total 1 objects

0:003> **!do 01ac26b4**
Fields:
MT Field Offset Type VT Attr **Value Name**
63f42dbc 4000001 4 System.Int32 1 instance **5 _quantity**
63f42dbc 4000002 8 System.Int32 1 instance **0 incorrect**

Intro to WinDBG for .NET Developers - part XXIV (still more CLR internals - Indexers)

C# introduces a new concept known as Indexers which are used for treating an object as an array. The indexers are usually known as smart arrays in C# community. Defining a C# indexer is much like defining properties.

.NET ACADEMY

Q & A:
Let's continue on the managed/.NET memory dump obtained in previous email.

What are indexers?

Indexers to provide array-like access to the classes.

The **'this'** is a special keyword to indicate the object of the current class.

Compare indexers and properties.

Indexers are similar to properties, except that the get and set accessors of indexers take parameters, while property accessors do not.

Sample program?

```
using System;

public class Beach24{

	private string[] _str = new string[10];

	public string this[int i]{
		set{
			_str[i] = value;
		}
		get{
			return _str[i];
		}
	}

	public int this[string str_param]{
		get{
			return System.Array.IndexOf(_str, str_param);
		}
	}

	static void Main(){

		Beach24 b24 = new Beach24();
		b24[0] = "Goan Beach";
		Console.WriteLine(b24[0]);		// Goan Beach
		Console.WriteLine(b24["Goan Beach"]);	// 0

		Console.WriteLine("Waiting (can take a memory dump now).");
```

```
            Console.ReadLine();
        }

}
```

Debugging?

```
-- Obtain the variable values
```

0:003> **!DumpDomain**
Module Name
00362c5c C:\cs\test2.exe

0:003> **!DumpModule -mt 00362c5c**
Types defined in this module
MT TypeDef Name

0036304c 0x02000002 **Beach24**

0:003> **!DumpMT -MD 0036304c**

MethodDesc Table
Entry MethodDesc JIT Name
65ce6a90 65b6494c PreJIT System.Object.ToString()
65ce6ab0 65b64954 PreJIT System.Object.Equals(System.Object)
65ce6b20 65b64984 PreJIT System.Object.GetHashCode()
65d57540 65b649a8 PreJIT System.Object.Finalize()
006b0118 00363038 JIT Beach24..ctor()
0036c011 00363008 NONE **Beach24.set_Item(Int32, System.String)**
0036c015 00363014 NONE **Beach24.get_Item(Int32)**
006b0148 00363020 JIT **Beach24.get_Item(System.String)**
006b0070 0036302c JIT Beach24.Main()

0:003> **!DumpHeap -type Beach24**
Address MT Size
01a826dc 0036304c 12
total 1 objects

0:003> **!do 01a826dc**
Fields:
MT Field Offset Type VT Attr **Value Name**
65d64340 4000001 4 System.Object[] 0 instance **01a826e8 _str**

0:003> **!do 01a826e8**
Name: System.Object**[]**
MethodTable: 65d64340
EEClass: 65b4da74
Size: 56(0x38) bytes
Array: Rank 1, Number of elements 10, Type CLASS
Element Type: System.String

Fields:
None

0:003> **!DumpArray 01a826e8**
Name: System.String[]
MethodTable: 65d64340
EEClass: 65b4da74
Size: 56(0x38) bytes
Array: Rank 1, Number of elements 10, Type CLASS
Element Methodtable: 65d90b70
[0] 01a82658
[1] null
[2] null
[3] null
[4] null
[5] null
[6] null
[7] null
[8] null
[9] null

0:003> **!do 01a82658**
Name: System.String
MethodTable: 65d90b70
EEClass: 65b4d66c
Size: 38(0x26) bytes
(C:\Windows\assembly\GAC_32\mscorlib\2.0.0.0__b77a5c561934e089\mscorlib.dll)
String: **Goan Beach**

Intro to WinDBG for .NET Developers - part XXV (still more CLR internals - Interface)

There are great ways for putting together plug-n-play like architectures where components can be interchanged at will. Since all interchangeable components implement the same interface, they can be used without any extra programming. The interface forces each component to expose specific public members that will be used in a certain way.

.NET ACADEMY

Q & A:

What's an interface?

An interface contains only the signatures of methods, properties, events or indexers. A class or struct that implements the interface must implement the members of the interface that are specified in the interface definition.

What is the difference between classes/structs and interfaces?

Classes and structs implement interfaces in a manner similar to how classes inherit a base class or struct, with two exceptions:

•A class or struct can implement more than one interface.
•When a class or struct implements an interface, it receives only the method names and signatures, because the interface itself contains no implementations.

Does C# support multiple inheritance?

Not yet, use interfaces instead.

Why can't you specify the accessibility modifier for methods inside the interface?

They all must be public. It's public by default.

Can you inherit multiple interfaces?

Yes.

Sample program?

```
using System;

public interface IWeight{     // "public" required so cReturn can use
interface ("internal" by default)

      int Weight(int _qty);
      string Hi();
      string Message { get; }
}
```

```
public class Beach25 : IWeight {

    int _quantity;

      public int Weight(int _qty){
            return _quantity * _qty;
      }

      public string Hi(){
            return "Hello";
      }

      public string Message{
            get{
                  return "Hello Message";
            }
      }

      static void Main(){

            // Class implements interface
            Beach25 b25 = new Beach25();
            b25._quantity = 10;
            Console.WriteLine(b25.Weight(5));   // 50

            // Interface can call class methods (defined by interface)
            IWeight iw = new Beach25();   // Cannot create an instance of
itself ('interface' or 'abstract class')
            Console.WriteLine(iw.Hi());         // Hello
            Console.WriteLine(iw.Message);            // Hello Message

            // Method takes interface parameter
            Console.WriteLine(cReturn.ReturnMessage(iw));   // Hello Message!

            Console.WriteLine("Waiting (can take a memory dump now).");
            Console.ReadLine();
      }

}

public class cReturn{

      // Method takes interface parameter
      static public string ReturnMessage(IWeight iw){
            return iw.Message + "!";
      }

}
```

Debugging?

```
-- Dump interface
```

0:003> **!DumpDomain**
Module Name
00182c5c C:\cs\test2.exe

0:003> **!DumpModule -mt 00182c5c**
    MT   TypeDef Name

0018306c 0x02000002 **IWeight**
001830f4 0x02000003 Beach25
00183174 0x02000004 cReturn

0:003> **!DumpMT -MD 0018306c**
EEClass: 00181310
Module: 00182c5c
Name: IWeight
mdToken: 02000002 (C:\cs\test2.exe)
BaseSize: 0x0
ComponentSize: 0x0
Number of IFaces in IFaceMap: 0
Slots in VTable: 3

MethodDesc Table
  Entry MethodDesc   JIT Name
0018c010   0018300c   NONE **IWeight.Weight(Int32)**
0018c01c   0018302c   NONE **IWeight.Hi()**
0018c028   0018304c   NONE **IWeight.get_Message()**

0:003> **!DumpHeap -type IWeight**
Address   MT   Size
total **0** objects

0:003> **!DumpHeap -type Beach25**
Address   MT   Size
01d826b4 001830f4   12
01d844c0 001830f4   12
total 2 objects

0:003> **!do 01d826b4**
Name: Beach25
MethodTable: 001830f4
EEClass: 001812ac
Fields:
    MT   Field   Offset   Type VT   Attr   **Value Name**
645d2dbc 4000001   4   System.Int32 1 instance   **10 _quantity**

0:003> **!do 01d844c0**
Name: Beach25
MethodTable: 001830f4
EEClass: 001812ac
Fields:
    MT   Field   Offset   Type VT   Attr   **Value Name**
645d2dbc 4000001   4   System.Int32 1 instance   **0 _quantity**