```
--------------------------------------------------------------------------------
NOTE: THIS FILE CONTAINS SOS DOCUMENTATION. THE FORMAT OF THE FILE IS:

<optional comments>
COMMAND: <cmd name, all lower case>
<descriptive text of the command>
\\ <these are two backslashes, immediately followed by a newline>

<repeat the sequence above>

The first command is "contents" which is the general help screen. The rest
correspond to SOS command names. This file is embedded as a resource in the SOS
binary. Be sure to list any new commands here.
--------------------------------------------------------------------------------



COMMAND: contents.
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)	ThreadState
DumpStackObjects (dso)	IP2MD
DumpHeap	U
DumpVC	DumpStack
GCRoot	EEStack
ObjSize	CLRStack
FinalizeQueue	GCInfo
PrintException (pe)	EHInfo
TraverseHeap	BPMD
	COMState

Examining CLR data structures	Diagnostic Utilities
DumpDomain	VerifyHeap
EEHeap	VerifyObj
Name2EE	FindRoots
SyncBlk	HeapStat
DumpMT	GCWhere
DumpClass	ListNearObj (lno)
DumpMD	GCHandles
Token2EE	GCHandleLeaks
EEVersion	FinalizeQueue (fq)
DumpModule	FindAppDomain
ThreadPool	SaveModule

```
DumpAssembly                        ProcInfo
DumpSigElem                         StopOnException (soe)
DumpRuntimeTypes                    DumpLog
DumpSig                             VMMap
RCWCleanupList                      VMStat
DumpIL                              MinidumpMode
DumpRCW                             AnalyzeOOM (ao)
DumpCCW

Examining the GC history            Other
----------------------------        ----------------------------
HistInit                            FAQ
HistRoot
HistObj
HistObjFind
HistClear
\\

COMMAND: 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 (CLR.DLL). Newer versions of the
Windows Debugger provide a command to make it easy to load the right copy of
SOS.DLL:
```

".loadby sos clr"

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

"lmvm clr"

in the debugger. Note that if you are running CoreCLR (e.g. Silverlight) then you should replace "clr" with "coreclr".

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 clr
0:000> !DumpStackObjects
Failed to find runtime DLL (clr.dll), 0x80004005
Extension commands need clr.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 clr!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.

>> What other tools can I use to find my bug?

Turn on Managed Debugging Assistants. These enable additional runtime diagnostics, particularly in the area of PInvoke/Interop. Adam Nathan has written some great information about that:

http://blogs.msdn.com/adam_nathan/

>> Does SOS support DML?

Yes. SOS respects the .prefer_dml option in the debugger. If this setting is turned on, then SOS will output DML by default. Alternatively, you may leave it off and add /D to the beginning of a command to get DML based output for it. Not all SOS commands support DML output.

\\

COMMAND: stoponexception.

```
!StopOnException [-derived]
                 [-create | -create2]
                 <Exception>
                 [<Pseudo-register number>]
```

!StopOnException helps when you want the Windows Debugger to stop on a particular managed exception, say a System.OutOfMemoryException, but continue

running if other exceptions are thrown. The command can be used in two ways:

1) When you just want to stop on one particular CLR exception

   At the debugger prompt, anytime after loading SOS, type:

   !StopOnException -create System.OutOfMemoryException 1

   The pseudo-register number (1) indicates that SOS can use register $t1 for maintaining the breakpoint. The -create parameter allows SOS to go ahead and set up the breakpoint as a first-chance exception. -create2 would set it up as a 2nd-chance exception.

2) When you need more complex logic for stopping on a CLR exception

   !StopOnException can be used purely as a predicate in a larger expression. If you type:

   !StopOnException System.OutOfMemoryException 3

   then register $t3 will be set to 1 if the last thrown exception on the current thread is a System.OutOfMemoryException. Otherwise, $t3 will be set to 0. Using the Windows Debugger scripting language, you could chain such calls together to stop on various exception types. You'll have to manually create such predicates, for example:

```
   sxe -c "!soe System.OutOfMemoryException 3;
           !soe -derived System.IOException 4;
           .if(@$t3==1 || @$t4==1) { .echo 'stop' } .else {g}"
```

The -derived option will cause StopOnException to set the pseudo-register to 1 even if the thrown exception type doesn't exactly match the exception type given, but merely derives from it. So, "-derived System.Exception" would catch every exception in the System.Exception heirarchy.

The pseudo-register number is optional. If you don't pass a number, SOS will use pseudo-register $t1.

Note that !PrintException with no parameters will print out the last thrown exception on the current thread (if any). You can use !soe as a shortcut for !StopOnException.
\\

COMMAND: minidumpmode.
!MinidumpMode <0 or 1>

Minidumps created with ".dump /m" or ".dump" have a very small set of CLR-specific data, just enough to run a subset of SOS commands correctly. You are able to run other SOS commands, but they may fail with unexpected errors because required areas of memory are not mapped in or only partially mapped

in. At this time, SOS cannot reliably detect if a dump file is of this type (for one thing, custom dump commands can map in additional memory, but there is no facility to read meta-information about this memory). You can turn this option on to protect against running unsafe commands against small minidumps.

By default, MinidumpMode is 0, so there is no restriction on commands that will run against a minidump.
\\

COMMAND: dumpobj.
!DumpObj [-nofields] <object address>

This command allows you to examine the fields of an object, as well as learn important properties of the object such as the EEClass, the MethodTable, and the size.

You might find an object pointer by running !DumpStackObjects and choosing from the resultant list. Here is a simple object:

```
0:000> !DumpObj a79d40
Name: Customer
MethodTable: 009038ec
EEClass: 03ee1b84
Size: 20(0x14) bytes
 (C:\pub\unittest.exe)
```

```
      Fields:
            MT    Field   Offset                 Type  VT     Attr    Value Name
      009038ec  4000008        4             Customer   0 instance 00a79ce4 name
      009038ec  4000009        8                 Bank   0 instance 00a79d2c bank
```

Note that fields of type Customer and Bank are themselves objects, and you can run !DumpObj on them too. You could look at the field directly in memory using the offset given. "dd a79d40+8 l1" would allow you to look at the bank field directly. Be careful about using this to set memory breakpoints, since objects can move around in the garbage collected heap.

What else can you do with an object? You might run !GCRoot, to determine what roots are keeping it alive. Or you can find all objects of that type with "!DumpHeap -type Customer".

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

The abbreviation !do can be used for brevity.

The arguments in detail:

-nofields: do not print fields of the object, useful for objects like String

\\

COMMAND: dumparray.

!DumpArray

[-start <startIndex>]

[-length <length>]

[-details]

[-nofields]

<array object address>

This command allows you to examine elements of an array object.

The arguments in detail:

-start <startIndex>: optional, only supported for single dimension array.
Specify from which index the command shows the elements.

-length <length>: optional, only supported for single dimension array.
Specify how many elements to show.

-details: optional. Ask the command to print out details
of the element using !DumpObj and !DumpVC format.

-nofields: optional, only takes effect when -details is used. Do
not print fields of the elements. Useful for arrays of
objects like String

Example output:

0:000> !dumparray -start 2 -length 3 -details 00ad28d0

```
Name: Value[]
MethodTable: 03e41044
EEClass: 03e40fc0
Size: 132(0x84) bytes
Array: Rank 1, Number of elements 10, Type VALUETYPE
Element Type: Value
[2] 00ad28f0
    Name: Value
    MethodTable 03e40f4c
    EEClass: 03ef1698
    Size: 20(0x14) bytes
     (C:\bugs\225271\arraytest.exe)
    Fields:
          MT    Field   Offset                 Type       Attr    Value Name
    5b9a628c  4000001        0         System.Int32   instance        2 x
    5b9a628c  4000002        4         System.Int32   instance        4 y
    5b9a628c  4000003        8         System.Int32   instance        6 z
[3] 00ad28fc
    Name: Value
    MethodTable 03e40f4c
    EEClass: 03ef1698
    Size: 20(0x14) bytes
     (C:\bugs\225271\arraytest.exe)
    Fields:
          MT    Field   Offset                 Type       Attr    Value Name
```

```
          5b9a628c  4000001        0         System.Int32   instance        3 x
          5b9a628c  4000002        4         System.Int32   instance        6 y
          5b9a628c  4000003        8         System.Int32   instance        9 z
      [4] 00ad2908
          Name: Value
          MethodTable 03e40f4c
          EEClass: 03ef1698
          Size: 20(0x14) bytes
           (C:\bugs\225271\arraytest.exe)
          Fields:
                MT    Field   Offset                 Type       Attr    Value Name
          5b9a628c  4000001        0         System.Int32   instance        4 x
          5b9a628c  4000002        4         System.Int32   instance        8 y
          5b9a628c  4000003        8         System.Int32   instance       12 z


\\

COMMAND: dumpstackobjects.
!DumpStackObjects [-verify] [top stack [bottom stack]]

This command will 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.
```

If you use the -verify option, each non-static CLASS field of an object candidate is validated. This helps to eliminate false positives. It is not on by default because very often in a debugging scenario, you are interested in objects with invalid fields.

The abbreviation !dso can be used for brevity.

\\

COMMAND: dumpheap.

```
!DumpHeap [-stat]
          [-strings]
          [-short]
          [-min <size>]
          [-max <size>]
          [-live]
          [-dead]
          [-thinlock]
          [-startAtLowerBound]
          [-mt <MethodTable address>]
          [-type <partial type name>]
          [start [end]]
```

!DumpHeap is a powerful command that traverses the garbage collected heap, collection statistics about objects. With it's 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.

When called without options, the output is first a list of objects in the heap, followed by a report listing all the types found, their size and number:

```
0:000> !dumpheap
 Address       MT     Size
00a71000 0015cde8       12 Free
00a7100c 0015cde8       12 Free
00a71018 0015cde8       12 Free
00a71024 5ba58328       68
00a71068 5ba58380       68
00a710ac 5ba58430       68
00a710f0 5ba5dba4       68
...
total 619 objects
Statistics:
      MT    Count TotalSize Class Name
5ba7607c        1        12
System.Security.Permissions.HostProtectionResource
5ba75d54        1        12
System.Security.Permissions.SecurityPermissionFlag
5ba61f18        1        12 System.Collections.CaseInsensitiveComparer
...
```

```
      0015cde8        6     10260      Free
      5ba57bf8      318     18136 System.String
      ...
```

"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. 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[]
```

The arguments in detail:

```
-stat     Restrict the output to the statistical type summary
-strings  Restrict the output to a statistical string value summary
-short    Limits output to just the address of each object. This allows you
          to easily pipe output from the command to another debugger
          command for automation.
-min      Ignore objects less than the size given in bytes
```

-max Ignore objects larger than the size given in bytes

-live Only print live objects

-dead Only print dead objects (objects which will be collected in the next full GC)

-thinlock Report on any ThinLocks (an efficient locking scheme, see !SyncBlk documentation for more info)

-startAtLowerBound

Force 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.)

-mt List only those objects with the MethodTable given

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

```
start     Begin listing from this address
end       Stop listing at this address
```

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>[]".

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:

```
0:000> !eeheap -gc
Number of GC Heaps: 1
generation 0 starts at 0x00c32754
generation 1 starts at 0x00c32748
generation 2 starts at 0x00a71000
 segment    begin allocated     size
00a70000 00a71000  010443a8 005d33a8(6108072)
Large object heap starts at 0x01a71000
 segment    begin allocated     size
01a70000 01a71000  01a75000 0x00004000(16384)
Total Size  0x5d73a8(6124456)
------------------------------
```

```
GC Heap Size  0x5d73a8(6124456)

0:000> !dumpheap 1a71000 1a75000
 Address       MT     Size
01a71000 5ba88bd8     2064
01a71810 0019fe48     2032 Free
01a72000 5ba88bd8     4096
01a73000 0019fe48     4096 Free
01a74000 5ba88bd8     4096
total 5 objects
Statistics:
      MT    Count TotalSize Class Name
0019fe48        2      6128      Free
5ba88bd8        3     10256 System.Object[]
Total 5 objects
```

Finally, if GC heap corruption is present, you may see an error like this:

```
0:000> !dumpheap -stat
object 00a73d24: does not have valid MT
curr_object : 00a73d24
Last good object: 00a73d14
----------------
```

That indicates a serious problem. See the help for !VerifyHeap for more

information on diagnosing the cause.

\\

COMMAND: dumpvc.

!DumpVC <MethodTable address> <Address>

!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. You need to know the MethodTable address to tell SOS how to interpret the fields, as a value class is not a first-class object with it's own MethodTable as the first field. For example:

```
0:000> !DumpObj a79d98
Name: Mainy
MethodTable: 009032d8
EEClass: 03ee1424
Size: 28(0x1c) bytes
 (C:\pub\unittest.exe)
Fields:
      MT    Field   Offset                 Type       Attr    Value Name
0090320c  4000010        4            VALUETYPE   instance 00a79d9c m_valuetype
009032d8  400000f        4                CLASS     static 00a79d54 m_sExcep
```

m_valuetype is a value type. The value in the MT column (0090320c) is the

MethodTable for it, and the Value column provides the start address:

```
0:000> !DumpVC 0090320c 00a79d9c
Name: Funny
MethodTable 0090320c
EEClass: 03ee14b8
Size: 28(0x1c) bytes
 (C:\pub\unittest.exe)
Fields:
      MT    Field   Offset                 Type       Attr    Value Name
0090320c  4000001        0                CLASS   instance 00a743d8
signature
0090320c  4000002        8         System.Int32   instance     2345 m1
0090320c  4000003       10       System.Boolean   instance        1 b1
0090320c  4000004        c         System.Int32   instance     1234 m2
0090320c  4000005        4                CLASS   instance 00a79d98
backpointer
```

!DumpVC is quite a specialized function. Some managed programs make heavy use of value classes, while others do not.

\\

COMMAND: gcroot.

!GCRoot [-nostacks] <Object address>

!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.

First, all stacks will be searched for roots, then handle tables, and finally the freachable queue of the finalizer. Some caution about the stack roots: !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.

Because people often want to restrict the search to gc handles and freachable objects, there is a -nostacks option.
\\

COMMAND: objsize.
!ObjSize [<Object address>]

With no parameters, !ObjSize lists the size of all objects found on managed threads. It also enumerates all GCHandles in the process, and totals the size of any objects pointed to by those handles. In calculating object size, !ObjSize includes the size of all child objects in addition to the parent.

For example, !DumpObj lists a size of 20 bytes for this Customer object:

```
0:000> !do a79d40
Name: Customer
MethodTable: 009038ec
EEClass: 03ee1b84
Size: 20(0x14) bytes
 (C:\pub\unittest.exe)
Fields:
      MT    Field   Offset                 Type       Attr    Value Name
009038ec  4000008        4                CLASS   instance 00a79ce4 name
009038ec  4000009        8                CLASS   instance 00a79d2c bank
009038ec  400000a        c       System.Boolean   instance        1 valid
```

but !ObjSize lists 152 bytes:

```
0:000> !ObjSize a79d40
sizeof(00a79d40) =      152 (    0x98) bytes (Customer)
```

This is because a Customer points to a Bank, has a name, and the Bank points to an Address string. You can use !ObjSize to identify any particularly large objects, such as a managed cache in a web server.

While running ObjSize with no arguments may point to specific roots that hold

onto large amounts of memory it does not provide information regarding the amount of managed memory that is still alive. This is due to the fact that a number of roots can share a common subgraph, and that part will be reported in the size of all the roots that reference the subgraph.

Please note the -aggregate parameter to !ObjSize has been removed. Please see '!DumpHeap -live' and '!DumpHeap -dead' for that functionality.

\\

COMMAND: finalizequeue.

!FinalizeQueue [-detail] | [-allReady] [-short]

This command lists the objects registered for finalization. Here is output from a simple program:

```
0:000> !finalizequeue
SyncBlocks to be cleaned up: 0
MTA Interfaces to be released: 0
STA Interfaces to be released: 1
generation 0 has 4 finalizable objects (0015bc90->0015bca0)
generation 1 has 0 finalizable objects (0015bc90->0015bc90)
generation 2 has 0 finalizable objects (0015bc90->0015bc90)
Ready for finalization 0 objects (0015bca0->0015bca0)
Statistics:
```

```
              MT    Count TotalSize Class Name
      5ba6cf78        1         24 Microsoft.Win32.SafeHandles.SafeFileHandle
      5ba5db04        1         68 System.Threading.Thread
      5ba73e28        2        112 System.IO.StreamWriter
      Total 4 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 "(0015bc90->0015bca0)" means that if you look at memory in that range, you'll see the object pointers that are registered:

```
0:000> dd 15bc90 15bca0-4
0015bc90  00a743f4 00a79f00 00a7b3d8 00a7b47c
```

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).

Specifying -short will inhibit any display related to SyncBlocks or RCWs.

The arguments in detail:

-allReady Specifying this argument will allow for the display of all objects

that are ready for finalization, whether they are already marked by the GC as such, or whether the next GC will. The objects that are not in the "Ready for finalization" list are finalizable objects that are no longer rooted. This option can be very expensive, as it verifies whether all the objects in the finalizable queues are still rooted or not.

-short Limits the output to just the address of each object. If used in conjunction with -allReady it enumerates all objects that have a finalizer that are no longer rooted. If used independently it lists all objects in the finalizable and "ready for finalization" queues.

-detail Will display 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.

\\

COMMAND: printexception.

!PrintException [-nested] [-lines] [-ccw] [<Exception object address>] [<CCW pointer>]

This will format fields of any object derived from System.Exception. One of the more 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.

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.

!PrintException 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.

!PrintException can display source information if available, by specifying the -lines command line argument.

!PrintException prints the exception object corresponding to a given CCW pointer, which can be specified using the -ccw option.

The abbreviation !pe can be used for brevity.
\\

COMMAND: traverseheap.
!TraverseHeap [-xml] [-verify] <filename>

!TraverseHeap writes out a file in a format understood by the CLR Profiler.

You can download the CLR Profiler from this link:

http://www.microsoft.com/downloads/details.aspx?FamilyId=86CE6052-D7F4-4AEB-9B7A-94635BEEBDDA&displaylang=en

It creates a graphical display of the GC heap to help you analyze the state of your application.

If you pass the -verify option it will do more sanity checking of the heap as it dumps it. Use this option if heap corruption is suspected.

If you pass the "-xml" flag, the file is instead written out in an easy to understand xml format:

```
<gcheap>
    <types>
        <type id="1" name="System.String">
        ...
    </types>
    <roots>
        <root kind="handle" address="0x00a73ff0"/>
        <root kind="stack" address="0x0069f0e0"/>
        ...
    </roots>
    <objects>
```

```
            <object address="0x00b73030" typeid="1" size="300"/>
            <object address="0x00b75054" typeid="5" size="20">
                <member address="0x00b75088" />
                ...
            </object>
            ...
        </objects>
    </gcheap>
```

You can break into your process, load SOS, take a snapshot of your heap with this function, then continue.

\\

COMMAND: threadstate.

!ThreadState value

The !Threads command outputs, among other things, the state of the thread. This is a bit field which corresponds to various states the thread is in. To check the state of the thread, simply pass that bit field from the output of !Threads into !ThreadState.

Example:

```
    0:003> !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  250 0019b068      a020 Disabled 02349668:02349fe8 0015def0     0
MTA

       2    2  944 001a6020      b220 Enabled  00000000:00000000 0015def0     0
MTA (Finalizer)

    0:003> !ThreadState b220

        Legal to Join

        Background

        CLR Owns

        CoInitialized

        In Multi Threaded Apartment


Possible thread states:

    Thread Abort Requested

    GC Suspend Pending

    User Suspend Pending

    Debug Suspend Pending

    GC On Transitions

    Legal to Join

    Yield Requested

    Hijacked by the GC
```

```
    Blocking GC for Stack Overflow
    Background
    Unstarted
    Dead
    CLR Owns
    CoInitialized
    In Single Threaded Apartment
    In Multi Threaded Apartment
    Reported Dead
    Fully initialized
    Task Reset
    Sync Suspended
    Debug Will Sync
    Stack Crawl Needed
    Suspend Unstarted
    Aborted
    Thread Pool Worker Thread
    Interruptible
    Interrupted
    Completion Port Thread
    Abort Initiated
    Finalized
    Failed to Start
    Detached
\\
```

COMMAND: threads.

!Threads [-live] [-special]

!Threads lists all the mananaged threads in the process.

-live: optional. Only print threads associated with a live thread.

-special: optional. With this switch, the command will display all the special threads created by CLR. Those threads might not be managed threads so they might not be shown in the first part of the command's output. Example of special threads include: GC threads (in concurrent GC and server GC), Debugger helper threads, Finalizer threads, AppDomain Unload threads, and Threadpool timer threads.

Each thread has many attributes, many of which can be ignored. The important ones are discussed below:

There are three ID columns:

1) The debugger shorthand ID (When the runtime is hosted this column might display the special string "<<<<" when this internal thread object is not associated with any physical thread - this may happen when the host reuses the runtime internal thread object)
2) The CLR Thread ID
3) The OS thread ID.

If PreEmptiveGC is enabled for a thread, then a garbage collection can occur while that thread is running. For example, if you break in while a managed thread is making a PInvoke call to a Win32 function, that thread will be in PreEmptive GC mode.

The Domain column indicates what AppDomain the thread is currently executing in. You can pass this value to !DumpDomain to find out more.

The APT column gives the COM apartment mode.

Exception will list the last thrown exception (if any) for the thread. More details can be obtained by passing the pointer value to !PrintException. If you get the notation "(nested exceptions)", you can get details on those exceptions by switching to the thread in question, and running "!PrintException -nested".
\\

COMMAND: clrstack.
!CLRStack [-a] [-l] [-p] [-n] [-f]
!CLRStack [-a] [-l] [-p] [-i] [variable name] [frame]

CLRStack attempts to provide a true stack trace for managed code only. It is handy for clean, simple traces when debugging straightforward managed programs. 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>. The -a (all) parameter is a short-cut for -l and -p combined.

The -f option (full mode) displays the native frames intermixing them with the managed frames and the assembly name and function offset for the managed frames.

If the debugger has the option SYMOPT_LOAD_LINES specified (either by the .lines or .symopt commands), SOS will look up the symbols for every managed frame and if successful will display the corresponding source file name and line number. The -n (No line numbers) parameter can be specified to disable this behavior.

When you see methods with the name "[Frame:...", that indicates a transition between managed and unmanaged code. You could run !IP2MD on the return addresses in the call stack to get more information on each managed method.

On x64 platforms, Transition Frames are not displayed at this time. To avoid heavy optimization of parameters and locals one can request the JIT compiler to not optimize functions in the managed app by creating a file myapp.ini (if your program is myapp.exe) in the same directory. Put the following lines in myapp.ini and re-run:

[.NET Framework Debugging Control]

```
GenerateTrackingInfo=1

AllowOptimize=0
```

The -i option is a new EXPERIMENTAL addition to CLRStack and will use the ICorDebug interfaces to display the managed stack and variables. With this option you can also view and expand arrays and fields for managed variables. If a stack frame number is specified in the command line, CLRStack will show you the parameters and/or locals only for that frame (provided you specify -l or -p or -a of course). If a variable name and a stack frame number are specified in the command line, CLRStack will show you the parameters and/or locals for that frame, and will also show you the fields for that variable name you specified. Here are some examples:

```
   !CLRStack -i -a           : This will show you all parameters and locals for
all frames

   !CLRStack -i -a 3         : This will show you all parameters and locals, for
frame 3

   !CLRStack -i var1 0       : This will show you the fields of 'var1' for frame 0

   !CLRStack -i var1.abc 2   : This will show you the fields of 'var1', and expand
                               'var1.abc' to show you the fields of the 'abc'
field,
                               for frame 2.

   !CLRStack -i var1.[basetype] 0   : This will show you the fields of 'var1', and
                                      expand the base type of 'var1' to show you
its
                                      fields.
```

!CLRStack -i var1.[6] 0 : If 'var1' is an array, this will show you the element

at index 6 in the array, along with its fields

The -i options uses DML output for a better debugging experience, so typically you should only need to execute "!CLRStack -i", and from there, click on the DML hyperlinks to inspect the different managed stack frames and managed variables.

\\

COMMAND: ip2md.

!IP2MD <Code address>

Given an address in managed JITTED code, IP2MD attempts to find the MethodDesc associated with it. For example, this output from K:

```
0:000> K
ChildEBP RetAddr
00a79c78 03ef02ab image00400000!Mainy.Top()+0xb
00a79c78 03ef01a6 image00400000!Mainy.Level(Int32)+0xb
00a79c78 5d3725a1 image00400000!Mainy.Main()+0xee
0012ea04 5d512f59 clr!CallDescrWorkerInternal+0x30
0012ee34 5d7946aa clr!CallDescrWorker+0x109

0:000> !IP2MD 03ef01a6
MethodDesc:   00902f40
Method Name:  Mainy.Main()
```

```
    Class:        03ee1424
    MethodTable:  009032d8
    mdToken:      0600000d
    Module:       001caa38
    IsJitted:     yes
    CodeAddr:     03ef00b8
    Transparency: Critical
    Source file:  c:\Code\prj.mini\exc.cs @ 39
```

We have taken a return address into Mainy.Main, and discovered information about that method. You could run !U, !DumpMT, !DumpClass, !DumpMD, or !DumpModule on the fields listed to learn more.

The "Source line" output will only be present if the debugger can find the symbols for the managed module containing the given <code address>, and if the debugger is configured to load line number information.
\\

COMMAND: u.
!U [-gcinfo] [-ehinfo] [-n] [-o] <MethodDesc address> | <Code address>

Presents an annotated disassembly of a managed method when given a MethodDesc pointer for the method, or a code address within the method body. Unlike the debugger "U" function, the entire method from start to finish is printed, with annotations that convert metadata tokens to names.

```
      <example output>

      ...

      03ef015d b901000000       mov     ecx,0x1

      03ef0162 ff156477a25b     call   dword ptr [mscorlib_dll+0x3c7764
(5ba27764)] (System.Console.InitializeStdOutError(Boolean), mdToken: 06000713)

      03ef0168 a17c20a701       mov     eax,[01a7207c] (Object: SyncTextWriter)

      03ef016d 89442414         mov     [esp+0x14],eax
```

If you pass the -gcinfo flag, you'll get inline display of the GCInfo for the method. You can also obtain this information with the !GCInfo command.

If you pass the -ehinfo flag, you'll get inline display of exception info for the method. (Beginning and end of try/finally/catch handlers, etc.). You can also obtain this information with the !EHInfo command.

If you pass the -o flag, the byte offset of each instruction from the beginning of the method will be printed in addition to the absolute address of the instruction.

If the debugger has the option SYMOPT_LOAD_LINES specified (either by the .lines or .symopt commands), and if symbols are available for the managed module containing the method being examined, the output of the command will include the source file name and line number corresponding to the disassembly. The -n (No line numbers) flag can be specified to disable this

behavior.

<example output>

...

c:\Code\prj.mini\exc.cs @ 38:

001b00b0 8b0d3020ab03   mov   ecx,dword ptr ds:[3AB2030h] ("Break in debugger. When done type <Enter> to continue: ")

001b00b6 e8d5355951   call   mscorlib_ni+0x8b3690 (51743690) (System.Console.Write(System.String), mdToken: 0600091b)

001b00bb 90   nop

c:\Code\prj.mini\exc.cs @ 39:

001b00bc e863cdc651   call   mscorlib_ni+0xf8ce24 (51e1ce24) (System.Console.ReadLine(), mdToken: 060008f6)

>>> 001b00c1 90   nop

...

\\

COMMAND: dumpstack.

!DumpStack [-EE] [-n] [top stack [bottom stack]]

[x86 and x64 documentation]

This command provides a verbose stack trace obtained by "scraping." Therefore the output is very noisy and potentially confusing. The command is good for viewing the complete call stack when "kb" gets confused. For best results,

make sure you have valid symbols.

-EE will only show managed functions.

If the debugger has the option SYMOPT_LOAD_LINES specified (either by the .lines or .symopt commands), SOS will look up the symbols for every managed frame and if successful will display the corresponding source file name and line number. The -n (No line numbers) parameter can be specified to disable this behavior.

You can also pass a stack range to limit the output. Use the debugger extension !teb to get the top and bottom stack values.

\\

COMMAND: eestack.
!EEStack [-short] [-EE]

This command runs !DumpStack on all threads in the process. The -EE option is passed directly to !DumpStack. The -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.

See the documentation for !DumpStack for more info.

\\

COMMAND: ehinfo.

!EHInfo (<MethodDesc address> | <Code address>)

!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.

Sample output:

```
0:000> !ehinfo 33bbd3a
MethodDesc: 03310f68
Method Name: MainClass.Main()
Class: 03571358
MethodTable: 0331121c
mdToken: 0600000b
Module: 001e2fd8
IsJitted: yes
CodeAddr: 033bbca0
Transparency: Critical
```

```
      EHHandler 0: TYPED catch(System.IO.FileNotFoundException)
      Clause: [033bbd2b, 033bbd3c] [8b, 9c]
      Handler: [033bbd3c, 033bbd50] [9c, b0]

      EHHandler 1: FINALLY
      Clause: [033bbd83, 033bbda3] [e3, 103]
      Handler: [033bbda3, 033bbdc5] [103, 125]

      EHHandler 2: TYPED catch(System.Exception)
      Clause: [033bbd7a, 033bbdc5] [da, 125]
      Handler: [033bbdc5, 033bbdd6] [125, 136]

\\

COMMAND: gcinfo.
!GCInfo (<MethodDesc address> | <Code address>)
```

!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.

Here is sample output where you can see the change in register state. Normally

you would print this output out and read it alongside a disassembly of the method. For example, the notation "reg EDI becoming live" at offset 0x11 of the method might correspond to a "mov edi,ecx" statement.

```
0:000> !gcinfo 5b68dbb8   (5b68dbb8 is the start of a JITTED method)
entry point 5b68dbb8
preJIT generated code
GC info 5b9f2f09
Method info block:
    method      size    = 0036
    prolog      size    =  19
    epilog      size    =   8
    epilog     count    =   1
    epilog      end     = yes
    saved reg.  mask    = 000B
    ebp frame           = yes
    fully interruptible=yes
    double align        = no
    security check      = no
    exception handlers = no
    local alloc         = no
    edit & continue     = no
    varargs             = no
    argument   count    =   4
    stack frame size    =   1
```

```
    untracked count    =   5
    var ptr tab count  =   0
    epilog        at   002E
36 D4 8C C7 AA |
93 F3 40 05    |

Pointer table:
14             |             [EBP+14H] an untracked  local
10             |             [EBP+10H] an untracked  local
0C             |             [EBP+0CH] an untracked  local
08             |             [EBP+08H] an untracked  local
44             |             [EBP-04H] an untracked  local
F1 79          | 0011        reg EDI becoming live
72             | 0013        reg ESI becoming live
83             | 0016        push ptr  0
8B             | 0019        push ptr  1
93             | 001C        push ptr  2
9B             | 001F        push ptr  3
56             | 0025        reg EDX becoming live
4A             | 0027        reg ECX becoming live
0E             | 002D        reg ECX becoming dead
10             | 002D        reg EDX becoming dead
E0             | 002D        pop  4 ptrs
F0 31          | 0036        reg ESI becoming dead
38             | 0036        reg EDI becoming dead
```

FF |

This 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.
\\

COMMAND: comstate.
!COMState

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

COMMAND: bpmd.
!BPMD [-nofuturemodule] <module name> <method name> [<il offset>]
!BPMD <source file name>:<line number>
!BPMD -md <MethodDesc>
!BPMD -list
!BPMD -clear <pending breakpoint number>
!BPMD -clearall

!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. -nofuturemodule can be used to suppress creating a breakpoint against a module that has not yet been loaded.

Management of the list of pending breakpoints can be done via !BPMD -list, !BPMD -clear, and !BPMD -clearall commands. !BPMD -list generates a list of all of the pending breakpoints. If the pending breakpoint has a non-zero module id, then that pending breakpoint is specific to function in that particular loaded module. If the pending breakpoint has a zero module id, then the breakpoint applies to modules that have not yet been loaded. Use !BPMD -clear or !BPMD -clearall to remove pending breakpoints from the list.

This brings up a good question: "I want to set a breakpoint on the main method of my application. How can I do this?"

1) If you know the full path to SOS, use this command and skip to step 6

   .load <the full path to sos.dll>

2) If you don't know the full path to sos, its usually next to clr.dll

   You can wait for clr to load and then find it.

   Start the debugger and type:

   sxe -c "" clrn

3) g

4) You'll get the following notification from the debugger:

"CLR notification: module 'mscorlib' loaded"

5) Now you can load SOS. Type

.loadby sos clr

6) Add the breakpoint with command such as:

!bpmd myapp.exe MyApp.Main

7) g

8) You will stop at the start of MyApp.Main. If you type "bl" you will see the breakpoint listed.

You can specify breakpoints by file and line number if:

a) You have some version of .Net Framework installed on your machine. Any OS from Vista onwards should have .Net Framework installed by default.

b) You have PDBs for the managed modules that need breakpoints, and your symbol path points to those PDBs.

This is often easier than module and method name syntax. For example:

!bpmd Demo.cs:15

To correctly specify explicitly implemented methods make sure to retrieve the method name from the metadata, or from the output of the "!dumpmt -md" command. For example:

```
public interface I1
{
    void M1();
}
public class ExplicitItfImpl : I1
{
    ...
    void I1.M1()        // this method's name is 'I1.M1'
    { ... }
}

!bpmd myapp.exe ExplicitItfImpl.I1.M1
```

!BPMD works equally well with generic types. Adding a breakpoint on a generic type sets breakpoints on all already JIT-ted generic methods and sets a pending breakpoint for any instantiation that will be JIT-ted in the future.

Example for generics:

Given the following two classes:

```
class G3<T1, T2, T3>
{
      ...
      public void F(T1 p1, T2 p2, T3 p3)
```

```
        { ... }
    }

    public class G1<T> {
        // static method
        static public void G<W>(W w)
        { ... }
    }
```

One would issue the following commands to set breapoints on G3.F() and G1.G():

```
!bpmd myapp.exe G3`3.F
!bpmd myapp.exe G1`1.G
```

And for explicitly implemented methods on generic interfaces:

```
    public interface IT1<T>
    {
        void M1(T t);
    }

    public class ExplicitItfImpl<U> : IT1<U>
    {
        ...
        void IT1<U>.M1(U u) // this method's name is 'IT1<U>.M1'
```

```
        { ... }
    }

    !bpmd bpmd.exe ExplicitItfImpl`1.IT1<U>.M1
```

Additional examples:

If IT1 and ExplicitItfImpl are types declared inside another class, Outer, the bpmd command would become:

```
!bpmd bpmd.exe Outer+ExplicitItfImpl`1.Outer.IT1<U>.M1
```

(note that the fully qualified type name for ExplicitItfImpl became Outer+ExplicitItfImpl, using the '+' separator, while the method name is Outer.IT1<U>.M1, using a '.' as the separator)

Furthermore, if the Outer class resides in a namespace, NS, the bpmd command to use becomes:

```
!bpmd bpmd.exe NS.Outer+ExplicitItfImpl`1.NS.Outer.IT1<U>.M1
```

!BPMD does not accept offsets nor parameters in the method name. You can add an IL offset as an optional parameter seperate from the name. If there are overloaded methods, !bpmd will set a breakpoint for all of them.

In the case of hosted environments such as SQL, the module name may be complex, like 'price, Version=0.0.0.0, Culture=neutral, PublicKeyToken=null'. For this case, just be sure to surround the module name with single quotes, like:

```
!bpmd 'price, Version=0.0.0.0, Culture=neutral, PublicKeyToken=null' Price.M2
```

\\

COMMAND: dumpdomain.

```
!DumpDomain [<Domain address>]
```

When called with no parameters, !DumpDomain will list all the AppDomains in the process. It enumerates each Assembly loaded into those AppDomains as well.
In addition to your application domain, and any domains it might create, there are two special domains: the Shared Domain and the System Domain.

Any Assembly pointer in the output can be passed to !DumpAssembly. Any Module pointer in the output can be passed to !DumpModule. Any AppDomain pointer can be passed to !DumpDomain to limit output only to that AppDomain. Other functions provide an AppDomain pointer as well, such as !Threads where it lists the current AppDomain for each thread.

\\

COMMAND: eeheap.

!EEHeap [-gc] [-loader]

!EEHeap enumerates process memory consumed by internal CLR data structures. You can limit the output by passing "-gc" or "-loader". All information will be displayed otherwise.

The information for the Garbage Collector lists the ranges of each Segment in the managed heap. This can be useful if you believe you have an object pointer. If the pointer falls within a segment range given by "!EEHeap -gc", then you do have an object pointer, and can attempt to run "!DumpObj" on it.

Here is output for a simple program:

```
0:000> !eeheap -gc
Number of GC Heaps: 1
generation 0 starts at 0x00a71018
generation 1 starts at 0x00a7100c
generation 2 starts at 0x00a71000
 segment    begin allocated     size
00a70000 00a71000  00a7e01c 0000d01c(53276)
Large object heap starts at 0x01a71000
 segment    begin allocated     size
01a70000 01a71000  01a76000 0x00005000(20480)
Total Size   0x1201c(73756)
------------------------------
```

```
GC Heap Size    0x1201c(73756)
```

So the total size of the GC Heap is only 72K. On a large web server, with multiple processors, you can expect to see a GC Heap of 400MB or more. The Garbage Collector attempts to collect and reclaim memory only when required to by memory pressure for better performance. You can also see the notion of "generations," wherein the youngest objects live in generation 0, and long-lived objects eventually get "promoted" to generation 2.

The loader output lists various private heaps associated with AppDomains. It also lists heaps associated with the JIT compiler, and heaps associated with Modules. For example:

```
0:000> !EEHeap -loader
Loader Heap:
---------------------------------------
System Domain: 5e0662a0
LowFrequencyHeap:008f0000(00002000:00001000) Size: 0x00001000 bytes.
HighFrequencyHeap:008f2000(00008000:00001000) Size: 0x00001000 bytes.
StubHeap:008fa000(00002000:00001000) Size: 0x00001000 bytes.
Total size: 0x3000(12288)bytes
---------------------------------------
Shared Domain: 5e066970
LowFrequencyHeap:00920000(00002000:00001000) 03e30000(00010000:00003000)
Size: 0x00004000 bytes.
```

```
    Wasted: 0x00001000 bytes.

    HighFrequencyHeap:00922000(00008000:00001000) Size: 0x00001000 bytes.

    StubHeap:0092a000(00002000:00001000) Size: 0x00001000 bytes.

    Total size: 0x6000(24576)bytes

    -------------------------------------

    Domain 1: 14f000

    LowFrequencyHeap:00900000(00002000:00001000) 03ee0000(00010000:00003000)
Size: 0x00004000 bytes.

    Wasted: 0x00001000 bytes.

    HighFrequencyHeap:00902000(00008000:00003000) Size: 0x00003000 bytes.

    StubHeap:0090a000(00002000:00001000) Size: 0x00001000 bytes.

    Total size: 0x8000(32768)bytes

    -------------------------------------

    Jit code heap:

    Normal JIT:03ef0000(00010000:00002000) Size: 0x00002000 bytes.

    Total size: 0x2000(8192)bytes

    -------------------------------------

    Module Thunk heaps:

    Module 5ba22410: Size: 0x00000000 bytes.

    Module 001c1320: Size: 0x00000000 bytes.

    Module 001c03f0: Size: 0x00000000 bytes.

    Module 001caa38: Size: 0x00000000 bytes.

    Total size: 0x0(0)bytes

    -------------------------------------

    Module Lookup Table heaps:
```

```
Module 5ba22410:Size: 0x00000000 bytes.
Module 001c1320:Size: 0x00000000 bytes.
Module 001c03f0:Size: 0x00000000 bytes.
Module 001caa38:03ec0000(00010000:00002000) Size: 0x00002000 bytes.
Total size: 0x2000(8192)bytes
-------------------------------------
Total LoaderHeap size: 0x15000(86016)bytes
=====================================
```

By using !EEHeap to keep track of the growth of these private heaps, we are able to rule out or include them as a source of a memory leak.

\\

COMMAND: name2ee.

!Name2EE <module name> <type or method name>

!Name2EE <module name>!<type or method name>

This function allows you to turn a class name into a MethodTable and EEClass. It turns a method name into a MethodDesc. Here is an example for a method:

```
0:000> !name2ee unittest.exe MainClass.Main
Module: 001caa38
Token: 0x0600000d
MethodDesc: 00902f40
Name: MainClass.Main()
```

```
      JITTED Code Address: 03ef00b8
```

and for a class:

```
      0:000> !name2ee unittest!MainClass
      Module: 001caa38
      Token: 0x02000005
      MethodTable: 009032d8
      EEClass: 03ee1424
      Name: MainClass
```

The module you are "browsing" with Name2EE needs to be loaded in the process. To get a type name exactly right, first browse the module with ILDASM. You can also pass * as the <module name> to search all loaded managed modules. <module name> can also be the debugger's name for a module, such as mscorlib or image00400000.

The Windows Debugger syntax of <module>!<type> is also supported. You can use an asterisk on the left of the !, but the type on the right side needs to be fully qualified.

If you are looking for a way to display a static field of a class (and you don't have an instance of the class, so !dumpobj won't help you), note that once you have the EEClass, you can run !DumpClass, which will display the value of all static fields.

There is yet one more way to specify a module name. In the case of modules loaded from an assembly store (such as a SQL db) rather than disk, the module name will look like this:

price, Version=0.0.0.0, Culture=neutral, PublicKeyToken=null

For this kind of module, simply use price as the module name:

```
0:044> !name2ee price Price
Module: 10f028b0 (price, Version=0.0.0.0, Culture=neutral,
PublicKeyToken=null)
Token: 0x02000002
MethodTable: 11a47ae0
EEClass: 11a538c8
Name: Price
```

Where are we getting these module names from? Run !DumpDomain to see a list of all loaded modules in all domains. And remember that you can browse all the types in a module with !DumpModule -mt <module pointer>.

\\

COMMAND: syncblk.

!SyncBlk [-all | <syncblk number>]

A SyncBlock is a holder for extra information that doesn't need to be created for every object. It can hold COM Interop data, HashCodes, and locking information for thread-safe operations.

When called without arguments, !SyncBlk will print the list of SyncBlocks corresponding to objects that are owned by a thread. For example, a

```
lock(MyObject)
{
    ....
}
```

statement will set MyObject to be owned by the current thread. A SyncBlock will be created for MyObject, and the thread ownership information stored there (this is an oversimplification, see NOTE below). If another thread tries to execute the same code, they won't be able to enter the block until the first thread exits.

This makes !SyncBlk useful for detecting managed deadlocks. Consider that the following code is executed by Threads A & B:

```
Resource r1 = new Resource();
Resource r2 = new Resource();

...
```

```
lock(r1)                          lock(r2)
{                                 {
    lock(r2)                          lock(r1)
    {                                 {
        ...                               ...
    }                                 }
}                                 }
```

This is a deadlock situation, as Thread A could take r1, and Thread B r2, leaving both threads with no option but to wait forever in the second lock statement. !SyncBlk will detect this with the following output:

```
0:003> !syncblk
Index SyncBlock MonitorHeld Recursion Owning Thread Info   SyncBlock Owner
  238 001e40ec            3         1 001e4e60   e04   3   00a7a194 Resource
  239 001e4124            3         1 001e5980   ab8   4   00a7a1a4 Resource
```

It means that Thread e04 owns object 00a7a194, and Thread ab8 owns object 00a7a1a4. Combine that information with the call stacks of the deadlock:

(threads 3 and 4 have similar output)

```
0:003> k
ChildEBP RetAddr
0404ea04 77f5c524 SharedUserData!SystemCallStub+0x4
```

```
0404ea08 77e75ee0 ntdll!NtWaitForMultipleObjects+0xc
0404eaa4 5d9de9d6 KERNEL32!WaitForMultipleObjectsEx+0x12c
0404eb38 5d9def80 clr!Thread::DoAppropriateAptStateWait+0x156
0404ecc4 5d9dd8bb clr!Thread::DoAppropriateWaitWorker+0x360
0404ed20 5da628dd clr!Thread::DoAppropriateWait+0xbb
0404ede4 5da4e2e2 clr!CLREvent::Wait+0x29d
0404ee70 5da4dd41 clr!AwareLock::EnterEpilog+0x132
0404ef34 5da4efa3 clr!AwareLock::Enter+0x2c1
0404f09c 5d767880 clr!AwareLock::Contention+0x483
0404f1c4 03f00229 clr!JITutil_MonContention+0x2c0
0404f1f4 5b6ef077 image00400000!Worker.Work()+0x79
...
```

By looking at the code corresponding to Worker.Work()+0x79 (run "!u 03f00229"), you can see that thread 3 is attempting to acquire the Resource 00a7a1a4, which is owned by thread 4.

NOTE:
It is not always the case that a SyncBlock will be created for every object that is locked by a thread. In version 2.0 of the CLR and above, a mechanism called a ThinLock will be used if there is not already a SyncBlock for the object in question. ThinLocks will not be reported by the !SyncBlk command. You can use "!DumpHeap -thinlock" to list objects locked in this way.
\\

COMMAND: dumpmt.

!DumpMT [-MD] <MethodTable address>

Examine 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.

\\

COMMAND: dumpclass.

!DumpClass <EEClass address>

The EEClass is a data structure associated with an object type. !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.

You can get an EEClass to look at from !DumpMT, !DumpObj, !Name2EE, and !Token2EE among others.

\\

COMMAND: dumpmd.

!DumpMD <MethodDesc address>

This 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 902f40
Method Name: Mainy.Main()
Class: 03ee1424
MethodTable: 009032d8
mdToken: 0600000d
Module: 001caa78
IsJitted: yes
CodeAddr: 03ef00b8
```

If IsJitted is "yes," you can run !U on the CodeAddr pointer to see a disassembly of the JITTED code. You can also call !DumpClass, !DumpMT, !DumpModule on the Class, MethodTable and Module fields above.

\\

COMMAND: token2ee.

!Token2EE <module name> <token>

This function allows you to turn a metadata token into a MethodTable or MethodDesc. Here is an example showing class tokens being resolved:

```
0:000> !token2ee unittest.exe 02000003
Module: 001caa38
Token: 0x02000003
MethodTable: 0090375c
```

```
EEClass: 03ee1ae0
Name: Bank
0:000> !token2ee image00400000 02000004
Module: 001caa38
Token: 0x02000004
MethodTable: 009038ec
EEClass: 03ee1b84
Name: Customer
```

The module you are "browsing" with Token2EE needs to be loaded in the process. This function doesn't see much use, especially since a tool like ILDASM can show the mapping between metadata tokens and types/methods in a friendlier way. But it could be handy sometimes.

You can pass "*" for <module name> to find what that token maps to in every loaded managed module. <module name> can also be the debugger's name for a module, such as mscorlib or image00400000.
\\

COMMAND: eeversion.
!EEVersion

This prints the Common Language Runtime version. It also tells you if the code is running in "Workstation" or "Server" mode, a distinction which affects the garbage collector. The most apparent difference in the debugger is that in

"Server" mode there is one dedicated garbage collector thread per CPU.

A handy supplement to this function is to also run "lm v m clr". That will provide more details about the CLR, including where clr.dll is loaded from.
\\

COMMAND: dumpmodule.
!DumpModule [-mt] <Module address>

You can get a Module address from !DumpDomain, !DumpAssembly and other functions. Here is sample output:

```
0:000> !DumpModule 1caa50
Name: C:\pub\unittest.exe
Attributes: PEFile
Assembly: 001ca248
LoaderHeap: 001cab3c
TypeDefToMethodTableMap: 03ec0010
TypeRefToMethodTableMap: 03ec0024
MethodDefToDescMap: 03ec0064
FieldDefToDescMap: 03ec00a4
MemberRefToDescMap: 03ec00e8
FileReferencesMap: 03ec0128
AssemblyReferencesMap: 03ec012c
```

```
MetaData start address: 00402230 (1888 bytes)
```

The Maps listed map metadata tokens to CLR data structures. Without going into too much detail, you can examine memory at those addresses to find the appropriate structures. For example, the TypeDefToMethodTableMap above can be examined:

```
0:000> dd 3ec0010
03ec0010  00000000 00000000 0090320c 0090375c
03ec0020  009038ec ...
```

This means TypeDef token 2 maps to a MethodTable with the value 0090320c. You can run !DumpMT to verify that. The MethodDefToDescMap takes a MethodDef token and maps it to a MethodDesc, which can be passed to !DumpMD.

There is a new option "-mt", which will display the types defined in a module, and the types referenced by the module. For example:

```
0:000> !dumpmodule -mt 1aa580
Name: C:\pub\unittest.exe
...<etc>...
MetaData start address: 0040220c (1696 bytes)

Types defined in this module
```

```
      MT    TypeDef Name
      ------------------------------------------------------------------------
      030d115c 0x02000002 Funny
      030d1228 0x02000003 Mainy

      Types referenced in this module

      MT    TypeRef Name
      ------------------------------------------------------------------------
      030b6420 0x01000001 System.ValueType
      030b5cb0 0x01000002 System.Object
      030fceb4 0x01000003 System.Exception
      0334e374 0x0100000c System.Console
      03167a50 0x0100000e System.Runtime.InteropServices.GCHandle
      0336a048 0x0100000f System.GC

\\

COMMAND: threadpool.
!ThreadPool

This command lists basic information about the ThreadPool, including the number
of work requests in the queue, number of completion port threads, and number of
timers.
\\
```

COMMAND: dumpassembly.

!DumpAssembly <Assembly address>

Example output:

```
0:000> !dumpassembly 1ca248
Parent Domain: 0014f000
Name: C:\pub\unittest.exe
ClassLoader: 001ca060
  Module Name
001caa50 C:\pub\unittest.exe
```

An assembly can consist of multiple modules, and those will be listed. You can get an Assembly address from the output of !DumpDomain.

\\

COMMAND: dumpruntimetypes.

!DumpRuntimeTypes

!DumpRuntimeTypes finds all System.RuntimeType objects in the gc heap and prints the type name and MethodTable they refer too. Sample output:

```
 Address   Domain       MT Type Name
-------------------------------------------------------------------------------
```

```
--
        a515f4   14a740 5baf8d28 System.TypedReference
        a51608   14a740 5bb05764 System.Globalization.BaseInfoTable
        a51958   14a740 5bb05b24 System.Globalization.CultureInfo
        a51a44   14a740 5bb06298 System.Globalization.GlobalizationAssembly
        a51de0   14a740 5bb069c8 System.Globalization.TextInfo
        a56b98   14a740 5bb12d28
System.Security.Permissions.HostProtectionResource
        a56bbc   14a740 5baf7248 System.Int32
        a56bd0   14a740 5baf3fdc System.String
        a56cfc   14a740 5baf36a4 System.ValueType
      ...
```

This command will print a "?" in the domain column if the type is loaded into multiple

AppDomains. For example:

```
    0:000> !DumpRuntimeTypes
     Address   Domain       MT Type Name
    ------------------------------------------------------------------------------
     28435a0        ?   3f6a8c System.TypedReference
     28435b4        ?   214d6c System.ValueType
     28435c8        ?   216314 System.Enum
     28435dc        ?   2147cc System.Object
     284365c        ?   3cd57c System.IntPtr
     2843670        ?   3feaac System.Byte
```

```
     2843684        ?  23a544c System.IEquatable`1[[System.IntPtr, mscorlib]]
     2843784        ?   3c999c System.Int32
     2843798        ?   3caa04 System.IEquatable`1[[System.Int32, mscorlib]]
```

\\

COMMAND: dumpsig.

!DumpSig <sigaddr> <moduleaddr>

This command dumps the signature of a method or field given by <sigaddr>. This is useful when you are debugging parts of the runtime which returns a raw PCCOR_SIGNATURE structure and need to know what its contents are.

Sample output for a method:

```
    0:000> !dumpsig 0x000007fe`ec20879d 0x000007fe`eabd1000
    [DEFAULT] [hasThis] Void (Boolean,String,String)
```

The first section of the output is the calling convention. This includes, but is not limited to, "[DEFAULT]", "[C]", "[STDCALL]", "[THISCALL]", and so on. The second portion of the output is either "[hasThis]" or "[explicit]" for whether the method is an instance method or a static method respectively. The third portion of the output is the return value (in this case a "void"). Finally, the method's arguments are printed as the final portion of the output.

Sample output for a field:

```
    0:000> !dumpsig 0x000007fe`eb7fd8cd 0x000007fe`eabd1000
    [FIELD] ValueClass System.RuntimeTypeHandle
```

!DumpSig will also work with generics. Here is the output for the following function:

```
    public A Test(IEnumerable<B> n)

    0:000> !dumpsig 00000000`00bc2437 000007ff00043178
    [DEFAULT] [hasThis] __Canon (Class System.Collections.Generic.IEnumerable`1<__Canon>)
```

\\

COMMAND: dumpsigelem.

!DumpSigElem <sigaddr> <moduleaddr>

This command dumps a single element of a signature object. For most circumstances, you should use !DumpSig to look at individual signature objects, but if you find a signature that has been corrupted in some manner you can use !DumpSigElem to read out the valid portions of it.

If we look at a valid signature object for a method we see the following:

```
0:000> !dumpsig 0x000007fe`ec20879d 0x000007fe`eabd1000
[DEFAULT] [hasThis] Void (Boolean,String,String)
```

We can look at the individual elements of this object by adding the offsets into the

object which correspond to the return value and parameters:

```
0:000> !dumpsigelem 0x000007fe`ec20879d+2 0x000007fe`eabd1000
Void
0:000> !dumpsigelem 0x000007fe`ec20879d+3 0x000007fe`eabd1000
Boolean
0:000> !dumpsigelem 0x000007fe`ec20879d+4 0x000007fe`eabd1000
String
0:000> !dumpsigelem 0x000007fe`ec20879d+5 0x000007fe`eabd1000
String
```

We can do something similar for fields. Here is the full signature of a field:

```
0:000> !dumpsig 0x000007fe`eb7fd8cd 0x000007fe`eabd1000
[FIELD] ValueClass System.RuntimeTypeHandle
```

Using !DumpSigElem we can find the type of the field by adding the offset of it (1) to

the address of the signature:

```
0:000> !dumpsigelem 0x000007fe`eb7fd8cd+1 0x000007fe`eabd1000
ValueClass System.RuntimeTypeHandle
```

!DumpSigElem will also work with generics. Let a function be defined as follows:

```
    public A Test(IEnumerable<B> n)
```

The elements of this signature can be obtained by adding offsets into the signature

when calling !DumpSigElem:

```
    0:000> !dumpsigelem 00000000`00bc2437+2 000007ff00043178
    __Canon
    0:000> !dumpsigelem 00000000`00bc2437+4 000007ff00043178
    Class System.Collections.Generic.IEnumerable`1<__Canon>
```

The actual offsets that you should add are determined by the contents of the signature itself. By trial and error you should be able to find various elements of the signature.

\\

COMMAND: rcwcleanuplist.

!RCWCleanupList [address]

A RuntimeCallableWrapper is an internal CLR structure used to host COM objects which are exposed to managed code. This is exposed to managed code through the System.__ComObject class, and when objects of this type are collected, and a reference to the underlying COM object is no longer needed, the corresponding RCW is cleaned up. If you are trying to debug an issue related to one of these

RCWs, then you can use the !RCWCleanupList function to display which COM objects will be released the next time a cleanup occurs.

If given an address, this function will display the RCWCleanupList at that address.
If no address is specified, it displays the default cleanup list, printing the wrapper, the context, and the thread of the object.

Example:

```
    0:002> !rcwcleanuplist 001c04d0
    RuntimeCallableWrappers (RCW) to be cleaned:
         RCW  CONTEXT   THREAD Apartment
      1d54e0   192008   181180       STA
      1d4140   192178        0       MTA
      1dff50   192178        0       MTA
    MTA Interfaces to be released: 2
    STA Interfaces to be released: 1
```

Note that CLR keeps track of which RCWs are bound to which managed objects through the SyncBlock of the object. As such, you can see more information about RCW objects through the !SyncBlk command. You can find more information about RCW cleanup through the !FinalizeQueue command.

\\

```
COMMAND: dumpil.
!DumpIL <Managed DynamicMethod object> |
        <DynamicMethodDesc pointer> |
        <MethodDesc pointer> |
        /i <IL pointer>
```

!DumpIL prints the IL code associated with a managed method. We added this function specifically to debug DynamicMethod code which was constructed on the fly. Happily it works for non-dynamic code as well.

You can use it in four ways:

1) If you have a System.Reflection.Emit.DynamicMethod object, just pass the pointer as the first argument.
2) If you have a DynamicMethodDesc pointer you can use that to print the IL associated with the dynamic method.
3) If you have an ordinary MethodDesc, you can see the IL for that as well, just pass it as the first argument.
4) If you have a pointer directly to the IL, specify /i followed by the the IL address. This is useful for writers of profilers that instrument IL.

Note that dynamic IL is constructed a bit differently. Rather than referring to metadata tokens, the IL points to objects in a managed object array. Here

is a simple example of the output for a dynamic method:

```
  0:000> !dumpil b741dc
  This is dynamic IL. Exception info is not reported at this time.
  If a token is unresolved, run "!do <addr>" on the addr given
  in parenthesis. You can also look at the token table yourself, by
  running "!DumpArray 00b77388".

  IL_0000: ldstr 70000002 "Inside invoked method "
  IL_0005: call 6000003 System.Console.WriteLine(System.String)
  IL_000a: ldc.i4.1
  IL_000b: newarr 2000004 "System.Int32"
  IL_0010: stloc.0
  IL_0011: ldloc.0
  IL_0012: ret
```

\\

COMMAND: verifyheap.

!VerifyHeap

!VerifyHeap is a diagnostic tool that checks the garbage collected heap for signs of corruption. It walks objects one by one in a pattern like this:

```
    o = firstobject;
```

```
    while(o != endobject)
    {
        o.ValidateAllFields();
        o = (Object *) o + o.Size();
    }
```

If an error is found, !VerifyHeap will report it. I'll take a perfectly good object and corrupt it:

```
0:000> !DumpObj a79d40
Name: Customer
MethodTable: 009038ec
EEClass: 03ee1b84
Size: 20(0x14) bytes
 (C:\pub\unittest.exe)
Fields:
      MT    Field   Offset                 Type       Attr    Value Name
009038ec  4000008        4                CLASS   instance 00a79ce4 name
009038ec  4000009        8                CLASS   instance 00a79d2c bank
009038ec  400000a        c       System.Boolean   instance        1 valid

0:000> ed a79d40+4 01  (change the name field to the bogus pointer value 1)
0:000> !VerifyHeap
object 01ee60dc: bad member 00000003 at 01EE6168
Last good object: 01EE60C4.
```

If this gc heap corruption exists, there is a serious bug in your own code or in the CLR. In user code, an error in constructing PInvoke calls can cause this problem, and running with Managed Debugging Assistants is advised. If that possibility is eliminated, consider contacting Microsoft Product Support for help.

\\

COMMAND: verifyobj.

!VerifyObj <object address>

!VerifyObj is a diagnostic tool that checks the object that is passed as an argument for signs of corruption.

```
0:002> !verifyobj 028000ec
object 0x28000ec does not have valid method table

0:002> !verifyobj 0680017c
object 0x680017c: bad member 00000001 at 06800184
```

\\

COMMAND: findroots.

!FindRoots -gen <N> | -gen any | <object address>

The "-gen" form causes the debugger to break in the debuggee on the next collection of the specified generation. The effect is reset as soon as the break occurs, in other words, if you need to break on the next collection you would need to reissue the command.

The last form of this command is meant to be used after the break caused by the other forms has occurred. Now the debuggee is in the right state for !FindRoots to be able to identify roots for objects from the current condemned generations.

!FindRoots is a diagnostic command that is meant to answer the following question:

"I see that GCs are happening, however my objects have still not been collected. Why? Who is holding onto them?"

The process of answering the question would go something like this:

1. Find out the generation of the object of interest using the !GCWhere command, say it is gen 1:

      !GCWhere <object address>

2. Instruct the runtime to stop the next time it collects that generation using the !FindRoots command:

```
!FindRoots -gen 1
g
```

3. When the next GC starts, and has proceeded past the mark phase a CLR notification will cause a break in the debugger:

```
(fd0.ec4): CLR notification exception - code e0444143 (first chance)
CLR notification: GC - end of mark phase.
      Condemned generation: 1.
```

4. Now we can use the !FindRoots <object address> to find out the cross generational references to the object of interest. In other words, even if the object is not referenced by any "proper" root it may still be referenced by an older object (from an older generation), from a generation that has not yet been scheduled for collection. At this point !FindRoots will search those older generations too, and report those roots.

```
0:002> !findroots 06808094
older generations::Root:  068012f8(AAA.Test+a)->
  06808094(AAA.Test+b)
```

\\

COMMAND: heapstat.

!HeapStat [-inclUnrooted | -iu]

This command shows the generation sizes for each heap and the total, how much free space there is in each generation on each heap. If the -inclUnrooted option is specified the report will include information about the managed objects from the GC heap that are not rooted anymore.

Sample output:

```
      0:002> !heapstat
      Heap     Gen0         Gen1         Gen2         LOH
      Heap0    177904       12           306956       8784
      Heap1    159652       12           12           16
      Total    337556       24           306968       8800

      Free space:                                            Percentage
      Heap0    28           12           12           64           SOH:  0% LOH:
0%
      Heap1    104          12           12           16           SOH:  0%
LOH:100%
      Total    132          24           24           80

      0:002> !heapstat -inclUnrooted
      Heap     Gen0         Gen1         Gen2         LOH
      Heap0    177904       12           306956       8784
      Heap1    159652       12           12           16
      Total    337556       24           306968       8800
```

```
      Free space:                                                    Percentage
      Heap0    28          12          12          64          SOH:  0% LOH:
0%
      Heap1    104         12          12          16          SOH:  0%
LOH:100%
      Total    132         24          24          80

      Unrooted objects:                                              Percentage
      Heap0    152212      0           306196      0           SOH: 94% LOH:
0%
      Heap1    155704      0           0           0           SOH: 97% LOH:
0%
      Total    307916      0           306196      0

The percentage column contains a breakout of free or unrooted bytes to total
bytes.

\\

COMMAND: analyzeoom.
!AnalyzeOOM

!AnalyzeOOM displays the info of the last OOM occurred on an allocation request to
the GC heap (in Server GC it displays OOM, if any, on each GC heap).

To see the managed exception(s) use the !Threads command which will show you
managed exception(s), if any, on each managed thread. If you do see an
```

OutOfMemoryException exception you can use the !PrintException command on it.

To get the full callstack use the "kb" command in the debugger for that thread.

For example, to display thread 3's stack use ~3kb.

OOM exceptions could be because of the following reasons:

1) allocation request to GC heap

   in which case you will see JIT_New* on the call stack because managed code called new.

2) other runtime allocation failure

   for example, failure to expand the finalize queue when GC.ReRegisterForFinalize is

   called.

3) some other code you use throws a managed OOM exception

   for example, some .NET framework code converts a native OOM exception to managed

   and throws it.

The !AnalyzeOOM command aims to help you with investigating 1) which is the most difficult because it requires some internal info from GC. The only exception is we don't support allocating objects larger than 2GB on CLR v2.0 or prior. And this command will not display any managed OOM because we will throw OOM right away instead of even trying to allocate it on the GC heap.

There are 2 legitimate scenarios where GC would return OOM to allocation requests -

one is if the process is running out of VM space to reserve a segment; the other is if the system is running out physical memory (+ page file if you have one) so GC can not commit memory it needs. You can look at these scenarios by using performance counters or debugger commands. For example for the former scenario the "!address -summary" debugger command will show you the largest free region in the VM. For the latter scenario you can look at the "Memory\% Committed Bytes In Use" see if you are running low on commit space. One important thing to keep in mind is when you do this kind of memory analysis it could an aftereffect and doesn't completely agree with what this command tells you, in which case the command should be respected because it truly reflects what happened during GC.

The other cases should be fairly obvious from the callstack.

Sample output:

```
0:011> !ao
---------Heap 2 ---------
Managed OOM occurred after GC #28 (Requested to allocate 1234 bytes)
Reason: Didn't have enough memory to commit
Detail: SOH: Didn't have enough memory to grow the internal GC datastructures (800000 bytes) -
        on GC entry available commit space was 500 MB
---------Heap 4 ---------
Managed OOM occurred after GC #12 (Requested to allocate 100000 bytes)
```

Reason: Didn't have enough memory to allocate an LOH segment

Detail: LOH: Failed to reserve memory (16777216 bytes)

\\

COMMAND: gcwhere.

!GCWhere <object address>

!GCWhere displays the location in the GC heap of the argument passed in.

```
0:002> !GCWhere 02800038
Address  Gen Heap segment  begin    allocated size
02800038 2    0   02800000 02800038 0282b740  12
```

When the argument lies in the managed heap, but is not a valid *object* address the "size" is displayed as 0:

```
0:002> !GCWhere 0280003c
Address  Gen Heap segment  begin    allocated size
0280003c 2    0   02800000 02800038 0282b740  0
```

\\

COMMAND: listnearobj.

!ListNearObj <object address>

!ListNearObj is a diagnostic tool that displays the object preceeding and succeeding the address passed in:

The command looks for the address in the GC heap that looks like a valid beginning of a managed object (based on a valid method table) and the object following the argument address.

```
0:002> !ListNearObj 028000ec
Before: 0x28000a4           72 (0x48      ) System.StackOverflowException
After:  0x2800134           72 (0x48      )
System.Threading.ThreadAbortException
Heap local consistency confirmed.

0:002> !ListNearObj 028000f0
Before: 0x28000ec           72 (0x48      ) System.ExecutionEngineException
After:  0x2800134           72 (0x48      )
System.Threading.ThreadAbortException
Heap local consistency confirmed.
```

The command considers the heap as "locally consistent" if:

```
prev_obj_addr + prev_obj_size = arg_addr && arg_obj + arg_size =
next_obj_addr
```

OR

```
prev_obj_addr + prev_obj_size = next_obj_addr
```

When the condition is not satisfied:

```
0:002> !lno 028000ec
Before: 0x28000a4           72 (0x48      ) System.StackOverflowException
After:  0x2800134           72 (0x48      ) System.Threading.ThreadAbortException
Heap local consistency not confirmed.
```

\\

COMMAND: dumplog.

!DumpLog [-addr <addressOfStressLog>] [<Filename>]

To aid in diagnosing hard-to-reproduce stress failures, the CLR team added an in-memory log capability. The idea was to avoid using locks or I/O which could disturb a fragile repro environment. The !DumpLog function allows you to write that log out to a file. If no Filename is specified, the file "Stresslog.txt" in the current directory is created.

The optional argument addr allows one to specify a stress log other then the default one.

```
0:000> !DumpLog
Attempting to dump Stress log to file 'StressLog.txt'
.................
```

```
      SUCCESS: Stress log dumped
```

To turn on the stress log, set the following registry keys under HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\.NETFramework:

```
(DWORD) StressLog = 1
(DWORD) LogFacility = 0xffffffbf (this is a bit mask, almost all logging is on.
                                  This is also the default value if the key
                                  isn't specified)
(DWORD) StressLogSize = 65536    (this is the default value if the key isn't
                                  specified)
(DWORD) LogLevel = 6             (this is the default value if the key isn't
                                  specified.  The higher the number the more
                                  detailed logs are generated.  The maximum
                                  value is decimal 10)
```

StressLogSize is the size in bytes of the in-memory log allocated for each thread in the process. In the case above, each thread gets a 64K log. You could increase this to get more logging, but more memory will be required for this log in the process. For example, 20 threads with 524288 bytes per thread has a memory demand of 10 Megabytes. The stress log is circular so new entries will replace older ones on threads which have reached their buffer limit.

The log facilities are defined as follows:

```
GC           0x00000001
GCINFO       0x00000002
STUBS        0x00000004
JIT          0x00000008
LOADER       0x00000010
METADATA     0x00000020
SYNC         0x00000040
EEMEM        0x00000080
GCALLOC      0x00000100
CORDB        0x00000200
CLASSLOADER  0x00000400
CORPROF      0x00000800
REMOTING     0x00001000
DBGALLOC     0x00002000
EH           0x00004000
ENC          0x00008000
ASSERT       0x00010000
VERIFIER     0x00020000
THREADPOOL   0x00040000
GCROOTS      0x00080000
INTEROP      0x00100000
MARSHALER    0x00200000
IJW          0x00400000
ZAP          0x00800000
STARTUP      0x01000000
```

```
    APPDOMAIN    0x02000000
    CODESHARING  0x04000000
    STORE        0x08000000
    SECURITY     0x10000000
    LOCKS        0x20000000
    BCL          0x40000000
```

Here is some sample output:

```
      3560   9.981137099 : `SYNC`               RareEnablePremptiveGC: entering.
      Thread state = a030

      3560   9.981135033 : `GC`GCALLOC`GCROOTS` ========== ENDGC 4194 (gen = 2,
      collect_classes = 0) ============{

      3560   9.981125826 : `GC`                         Segment mem 00C61000 alloc
      = 00D071F0 used 00D09254 committed 00D17000

      3560   9.981125726 : `GC`                     Generation 0 [00CED07C,
00000000
      ] cur = 00000000

      3560   9.981125529 : `GC`                     Generation 1 [00CED070,
00000000
      ] cur = 00000000
```

```
      3560   9.981125103 : `GC`                    Generation 2 [00C61000,
00000000
      ] cur = 00000000

      3560   9.981124963 : `GC`                GC Heap 00000000

      3560   9.980618994 : `GC`GCROOTS`        GcScanHandles (Promotion Phase =
0)
```

The first column is the OS thread ID for the thread appending to the log, the second column is the timestamp, the third is the facility category for the log entry, and the fourth contains the log message. The facility field is expressed as `facility1`facility2`facility3`. This facilitates the creation of filters for displaying only specific message categories. To make sense of this log, you would probably want the Shared Source CLI to find out exactly where the log comes from.

\\

COMMAND: findappdomain.

!FindAppDomain <Object address>

!FindAppDomain will attempt to resolve the AppDomain of an object. For example, using an Object Pointer from the output of !DumpStackObjects:

```
      0:000> !findappdomain 00a79d98
      AppDomain: 0014f000
```

```
    Name: unittest.exe
    ID: 1
```

You can find out more about the AppDomain with the !DumpDomain command. Not every object has enough clues about it's origin to determine the AppDomain. Objects with Finalizers are the easiest case, as the CLR needs to be able to call those when an AppDomain shuts down.

\\

COMMAND: savemodule.

!SaveModule <Base address> <Filename>

This command allows you to take a image loaded in memory and write it to a file. This is especially useful if you are debugging a full memory dump, and don't have the original DLLs or EXEs. This is most often used to save a managed binary to a file, so you can disassemble the code and browse types with ILDASM.

The base address of an image can be found with the "LM" debugger command:

```
    0:000> lm
    start    end        module name
    00400000 00408000   image00400000     (deferred)
    10200000 102ac000   MSVCR80D     (deferred)
    5a000000 5a0b1000   mscoree      (deferred)
    5a140000 5a29e000   clrjit     (deferred)
```

```
5b660000 5c440000   mscorlib_dll     (deferred)
5d1d0000 5e13c000   clr     (deferred)
...
```

If I wanted to save a copy of clr.dll, I could run:

```
0:000> !SaveModule 5d1d0000 c:\pub\out.tmp
4 sections in file
section 0 - VA=1000, VASize=e82da9, FileAddr=400, FileSize=e82e00
section 1 - VA=e84000, VASize=24d24, FileAddr=e83200, FileSize=ec00
section 2 - VA=ea9000, VASize=5a8, FileAddr=e91e00, FileSize=600
section 3 - VA=eaa000, VASize=c183c, FileAddr=e92400, FileSize=c1a00
```

The diagnostic output indicates that the operation was successful. If c:\pub\out.tmp already exists, it will be overwritten.

\\

COMMAND: gchandles.

!GCHandles [-type handletype] [-stat] [-perdomain]

!GCHandles provides statistics about GCHandles in the process.

Paremeters:

stat - Only display the statistics and not the list of handles and what they point to.

```
    perdomain - Break down the statistics by the app domain in which
                the handles reside.
    type - A type of handle to filter it by.  The handle types are:
           Pinned
           RefCounted
           WeakShort
           WeakLong
           Strong
           Variable
           AsyncPinned
           SizedRef
```

Sometimes the source of a memory leak is a GCHandle leak. For example, code might keep a 50 Megabyte array alive because a strong GCHandle points to it, and the handle was discarded without freeing it.

The most common handles are "Strong Handles," which keep the object they point to alive until the handle is explicitly freed. "Pinned Handles" are used to prevent the garbage collector from moving an object during collection. These should be used sparingly, and for short periods of time. If you don't follow that precept, the gc heap can become very fragmented.

Here is sample output from a very simple program. Note that the "RefCount" field only applies to RefCount Handles, and this field will contain the reference count:

```
0:000> !GCHandles

      Handle Type          Object     Size RefCount Type

    001611c0 Strong      01d00b58       84
System.IndexOutOfRangeException

    001611c4 Strong      01d00b58       84
System.IndexOutOfRangeException

    001611c8 Strong      01d1b48c       40          System.Diagnostics.LogSwitch

    001611d0 Strong      01cfd2c0       36          System.Security.PermissionSet

    001611d4 Strong      01cf7484       56          System.Object[]

    001611d8 Strong      01cf1238       32          System.SharedStatics

    001611dc Strong      01cf11c8       84
System.Threading.ThreadAbortException

    001611e0 Strong      01cf1174       84
System.Threading.ThreadAbortException

    001611e4 Strong      01cf1120       84
System.ExecutionEngineException

    001611e8 Strong      01cf10cc       84          System.StackOverflowException

    001611ec Strong      01cf1078       84          System.OutOfMemoryException

    001611f0 Strong      01cf1024       84          System.Exception

    001611f8 Strong      01cf2068       48          System.Threading.Thread

    001611fc Strong      01cf1328      112          System.AppDomain

    001613ec Pinned      02cf3268     8176          System.Object[]

    001613f0 Pinned      02cf2258     4096          System.Object[]

    001613f4 Pinned      02cf2038      528          System.Object[]

    001613f8 Pinned      01cf121c       12          System.Object

    001613fc Pinned      02cf1010     4116          System.Object[]
```

```
    Statistics:
          MT    Count    TotalSize Class Name
    563266dc        1           12 System.Object
    56329708        1           32 System.SharedStatics
    5632bc38        1           36 System.Security.PermissionSet
    5635f934        1           40 System.Diagnostics.LogSwitch
    5632759c        1           48 System.Threading.Thread
    5632735c        1           84 System.ExecutionEngineException
    56327304        1           84 System.StackOverflowException
    563272ac        1           84 System.OutOfMemoryException
    563270c4        1           84 System.Exception
    56328914        1          112 System.AppDomain
    56335f78        2          168 System.IndexOutOfRangeException
    563273b4        2          168 System.Threading.ThreadAbortException
    563208d0        5        16972 System.Object[]
    Total 19 objects

    Handles:
        Strong Handles:       14
        Pinned Handles:       5
\\

COMMAND: gchandleleaks.
!GCHandleLeaks
```

This command is an aid in tracking down GCHandle leaks. It searches all of memory for any references to the Strong and Pinned GCHandles in the process, and reports what it found. If a handle is found, you'll see the address of the reference. This might be a stack address or a field within an object, for example. If a handle is not found in memory, you'll get notification of that too.

The command has diagnostic output which doesn't need to be repeated here. One thing to keep in mind is that anytime you search all of memory for a value, you can get false positives because even though the value was found, it might be garbage in that no code knows about the address. You can also get false negatives because a user is free to pass that GCHandle to unmanaged code that might store the handle in a strange way (shifting bits, for example).

For example, a GCHandle valuetype is stored on the stack with the low bit set if it points to a Pinned handle. So !GCHandleLeaks ignores the low bit in it's searches.

That said, if a serious leak is going on, you'll get a ever-growing set of handle addresses that couldn't be found.
\\

COMMAND: vmmap.
!VMMap

!VMMap traverses the virtual address space and lists the type of protection applied to each region. Sample output:

```
0:000> !VMMap
Start    Stop     Length    AllocProtect  Protect       State    Type
00000000-0000ffff 00010000               NA            Free
00010000-00011fff 00002000  RdWr         RdWr          Commit   Private
00012000-0001ffff 0000e000               NA            Free
00020000-00020fff 00001000  RdWr         RdWr          Commit   Private
00021000-0002ffff 0000f000               NA            Free
00030000-00030fff 00001000  RdWr                       Reserve  Private
...
```

\\

COMMAND: vmstat.

!VMStat

Provides a summary view of the virtual address space, ordered by each type of protection applied to that memory (free, reserved, committed, private, mapped, image). The TOTAL column is (AVERAGE * BLK COUNT). Sample output below:

```
0:000> !VMStat
~~~~           ~~~~~~~        ~~~~~~~        ~~~~~~~  ~~~~~~~~~~
~~~~~
TYPE           MINIMUM        MAXIMUM        AVERAGE  BLK COUNT
```

TOTAL

Free:

Small	4,096	65,536	48,393	27	1,306,611
Medium	139,264	528,384	337,920	4	1,351,680
Large	6,303,744	974,778,368	169,089,706	12	2,029,076,472
Summary	4,096	974,778,368	47,249,646	43	2,031,734,778

Reserve:

Small	4,096	65,536	43,957	41	1,802,237
Medium	249,856	1,019,904	521,557	6	3,129,342
Large	2,461,696	16,703,488	11,956,224	3	35,868,672
Summary	4,096	16,703,488	816,005	50	40,800,250

\\

COMMAND: procinfo.

!ProcInfo [-env] [-time] [-mem]

!ProcInfo lists the environment variables for the process, kernel CPU time, as well as memory usage statistics.

\\

COMMAND: histinit.

!HistInit

Before running any of the Hist - family commands you need to initialize the SOS structures from the stress log saved in the debuggee. This is achieved by the HistInit command.

Sample output:

```
0:001> !HistInit
Attempting to read Stress log
STRESS LOG:
    facilitiesToLog  = 0xffffffff
    levelToLog       = 6
    MaxLogSizePerThread = 0x10000 (65536)
    MaxTotalLogSize = 0x1000000 (16777216)
    CurrentTotalLogChunk = 9
    ThreadsWithLogs  = 3
    Clock frequency  = 3.392 GHz
    Start time         15:26:31
    Last message time  15:26:56
    Total elapsed time 25.077 sec
.....................................
```

```
      ---------------------------- 2407 total
entries ------------------------------



      SUCCESS: GCHist structures initialized


\\


COMMAND: histobjfind.
!HistObjFind <obj_address>


To examine log entries related to an object whose present address is known one
would use this command. The output of this command contains all entries that
reference the object:


      0:003> !HistObjFind 028970d4
       GCCount   Object                                    Message
      ----------------------------------------------------------
          2296 028970d4 Promotion for root 01e411b8 (MT = 5b6c5cd8)
          2296 028970d4 Relocation NEWVALUE for root 00223fc4
          2296 028970d4 Relocation NEWVALUE for root 01e411b8
      ...
          2295 028970d4 Promotion for root 01e411b8 (MT = 5b6c5cd8)
          2295 028970d4 Relocation NEWVALUE for root 00223fc4
          2295 028970d4 Relocation NEWVALUE for root 01e411b8
```

```
      ...

\\

COMMAND: histroot.
!HistRoot <root>
```

The root value obtained from !HistObjFind can be used to track the movement of an object through the GCs.

HistRoot provides information related to both promotions and relocations of the root specified as the argument.

```
      0:003> !HistRoot 01e411b8
       GCCount    Value       MT Promoted?                Notes
      ------------------------------------------------------------
          2296 028970d4 5b6c5cd8       yes
          2295 028970d4 5b6c5cd8       yes
          2294 028970d4 5b6c5cd8       yes
          2293 028970d4 5b6c5cd8       yes
          2292 028970d4 5b6c5cd8       yes
          2291 028970d4 5b6c5cd8       yes
          2290 028970d4 5b6c5cd8       yes
          2289 028970d4 5b6c5cd8       yes
          2288 028970d4 5b6c5cd8       yes
```

```
          2287 028970d4 5b6c5cd8       yes
          2286 028970d4 5b6c5cd8       yes
          2285 028970d4 5b6c5cd8       yes
           322 028970e8 5b6c5cd8       yes Duplicate promote/relocs
      ...
```

\\

COMMAND: histobj.

!HistObj <obj_address>

This command examines all stress log relocation records and displays the chain of GC relocations that may have led to the address passed in as an argument. Conceptually the output is:

```
            GenN    obj_address   root1, root2, root3,
            GenN-1  prev_obj_addr root1, root2,
            GenN-2  prev_prev_oa  root1, root4,
            ...
```

Sample output:

```
      0:003> !HistObj 028970d4
       GCCount   Object                                        Roots
      -----------------------------------------------------------
          2296 028970d4 00223fc4, 01e411b8,
```

```
          2295 028970d4 00223fc4, 01e411b8,
          2294 028970d4 00223fc4, 01e411b8,
          2293 028970d4 00223fc4, 01e411b8,
          2292 028970d4 00223fc4, 01e411b8,
          2291 028970d4 00223fc4, 01e411b8,
          2290 028970d4 00223fc4, 01e411b8,
          2289 028970d4 00223fc4, 01e411b8,
          2288 028970d4 00223fc4, 01e411b8,
          2287 028970d4 00223fc4, 01e411b8,
          2286 028970d4 00223fc4, 01e411b8,
          2285 028970d4 00223fc4, 01e411b8,
           322 028970d4 01e411b8,
             0 028970d4

\\

COMMAND: histclear.
!HistClear

This command releases any resources used by the Hist-family of commands.
Generally there's no need to call this explicitly, as each HistInit will first
cleanup the previous resources.

      0:003> !HistClear
      Completed successfully.
```

\\

COMMAND: dumprcw.

!DumpRCW <RCW address>

This command lists information about a Runtime Callable Wrapper. You can use !DumpObj to obtain the RCW address corresponding to a managed object.

The output contains all COM interface pointers that the RCW holds on to, which is useful for investigating lifetime issues of interop-heavy applications.

\\

COMMAND: dumpccw.

!DumpCCW <CCW address or COM IP>

This command lists information about a COM Callable Wrapper. You can use !DumpObj to obtain the CCW address corresponding to a managed object or pass a COM interface pointer to which the object has been marshaled.

The output contains the COM reference count of the CCW, which is useful for investigating lifetime issues of interop-heavy applications.

\\