Debugging Out of Memory Exceptions

Even though the CLR heap manager and the garbage collector work hard to ensure that memory is automatically managed and used in the most efficient way possible, bad programming can still cause serious issues in .NET applications. In this part of the chapter, we will take a look at how a .NET application can exhaust enough memory to fail with an `OutOfMemoryException` and how we can use the debuggers to figure out the source of the problem. It is important to note that the example we will use illustrates how memory can be exhausted in the managed world and does not cover the various ways in which resources can be leaked in native code when invoked via the interoperability services layer. In Chapter 7, "Interoperability," we will look at an example of a native resource leak caused by improper invocations from managed code.

The application we will use to illustrate the problem is shown in Listing 5-9.

Listing 5-9. Example of an application that causes an eventual OutOfMemoryException

```
using System;
using System.IO;
using System.Xml.Serialization;


namespace Advanced.NET.Debugging.Chapter5
{


    public class Person
    {
        private string name;
        private string social;
        private int age;


        public string Name
        {
            get { return name; }
            set { this.name=value;}
        }


        public string SocialSecurity
        {
            get { return social; }
            set { this.social= value; }
        }


        public int Age
        {
            get { return age; }
            set { this.age = value; }
        }


        public Person() {}
        public Person(string name, string ss, int age)
        {
            this.name = name; this.social = ss; this.age = age;
        }
    }


    class OOM
    {
        static void Main(string[] args)
        {
            OOM o = new OOM();
            o.Run();
        }


        public void Run()
        {
            XmlRootAttribute root = new XmlRootAttribute();
            root.ElementName = "MyPersonRoot";
            root.Namespace = "http://www.contoso.com";
            root.IsNullable = true;


            while (true)
```

```
            {
                Person p = new Person();
                p.Name = "Mario Hewardt";
                p.SocialSecurity = "xxx-xx-xxxx";
                p.Age = 99;


                XmlSerializer ser = new
                    XmlSerializer(typeof(Person), root);
                Stream s = new
                    FileStream("c:\\ser.txt", FileMode.Create);


                ser.Serialize(s, p);
                s.Close();
            }
        }
    }
}
```

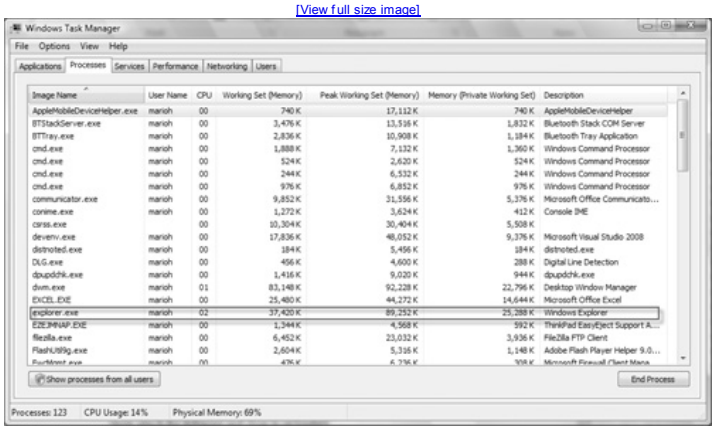
The source code and binary for Listing 5-9 can be found in the following folders:

- Source code: `C:\ADND\Chapter5\OOM`
- Binary: `C:\ADNDBin\05OOM.exe`

The application is pretty straightforward and consists of a `Person` class and an `OOM` class. The `OOM` class contains a `Run` method that sits in a tight loop creating instances of the `Person` class and serializes the instance into XML stored in a file on the local drive. When we run this application, we would like to monitor the memory consumption to see if it steadily increases over time, which could eventually lead to an `OutOfMemoryException` being thrown. What tools do we have at our disposal to monitor the memory consumption of a process? We have several options. The most basic option is to simply use task manager (shortcut `SHIFT-CTRL-ESC`). Task manager can display per-process memory information such as the working set, commit size, and paged/nonpaged pool. By default, only the Memory (Private Working Set) is enabled. To enable other process information, the Select Columns menu choice on the View menu can be used. The Windows Task Manager has several different tabs, and the tab of most interest when looking at per-process details is the Processes tab. The Processes tab shows a number of rows where each row represents a running process. Each of the columns in turn shows a specific piece of information about the process. Figure 5-13 shows an example of Windows Task Manager with a number of different memory details enabled in the Processes tab.

Figure 5-13. Example of Windows Task Manager Processes tab

[View full size image]



In Figure 5-13, we can see, for example, that `explorer.exe`'s working set size is 37,420K. Before we can move forward and effectively utilize Windows Task Manager for memory-related investigations, we have to have a clear understanding of what each of the possible memory-related columns means. Table 5-2 details the most commonly used columns and their descriptions.

Table 5-2. Windows Task Manager Memory-Related Columns

Column	Description
Memory – Working Set	Amount of memory in the private working set as well as the shared memory
Memory – Peak Working Set	Maximum amount of working set used by the process
Memory – Working Set Delta	Amount of change in the working set
Memory – Private Working Set	Amount of memory the process is using minus shared memory
Memory – Commit Size	Amount of virtual memory committed by the process

Pre-Windows Vista Task Manager

Some much-needed changes were made in Windows Vista and later versions to better capture the memory-related process information. Prior to Windows Vista, Windows Task Manager had a column named VM size, which, contrary to popular belief, indicated the amount of private bytes a

process was consuming. Similarly, the Mem Usage column corresponds to the working set (including shared memory) of the process. Finally, a feature we will utilize in Chapter 8, "Postmortem Debugging," is the capability to create dump files simply by right-clicking on the process and choosing the Create Dump File item.

Let's run `005OOM.exe` and watch the Memory – Working Set, Memory – Private Working Set, and Memory – Commit Size columns. Table 5-3 shows the results taken at periodic (approximately 60-second) intervals.

Table 5-3. Memory Usage of 005OOM.exe

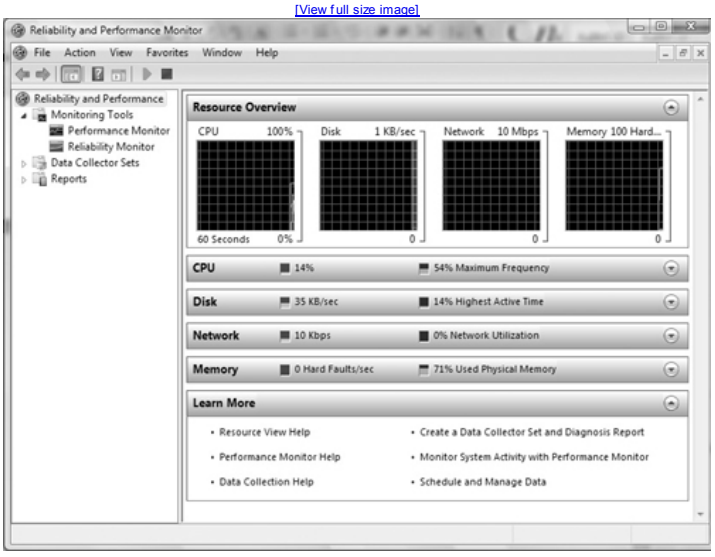
Interval	Working Set (K)	Private Working Set (K)	Commit Size (K)
1	18,000	7,000	16,000
2	24,000	11,000	19,000
3	28,000	13,000	22,000
4	33,000	16,000	25,000
5	38,000	19,000	28,000
6	42,000	22,000	30,000

From Table 5-3, we can see that we have a steady increase across the board. Both the working set sizes as well as the commit size are continuously growing. If this application is allowed to run indefinitely, chances are high that it could eventually run out of memory and an `OutOfMemoryException` would be thrown. Although using the Windows Task Manager is useful to get an overview of the memory consumed, what information does it present to us as far as figuring out the source of the excessive memory consumption? Is the memory located on the native heap or the managed heap? Is it located on the heap period or elsewhere?

To find the answers to those questions, we need a more granular tool to aid us: the Windows Reliability and Performance Monitor. The Windows Reliability and Performance Monitor tool is a powerful and extensible tool that can be used to investigate the state of the system as a whole or on a per-process basis. The tool uses several different data sources such as performance counters, trace logs, and configuration information. During .NET debug sessions, performance counters is the most commonly used data source. A performance counter is an entity that is responsible for publishing a specific performance characteristic of an application or service at regular time intervals or under specific conditions. For example, a Web service servicing credit card transactions can publish a performance counter that shows how many failed transactions have occurred over time. The Windows Reliability and Performance tool knows where to gather the performance counter data and displays the results in a nice graphical and historical view. To run the tool, click the Windows Start button and type `perfmon.exe` in the search tool (prior to Windows Vista, select run and then type `perfmon.exe`). Figure 5-14 shows an example of the start screen of the tool.

Figure 5-14. Windows Reliability and Performance Monitor

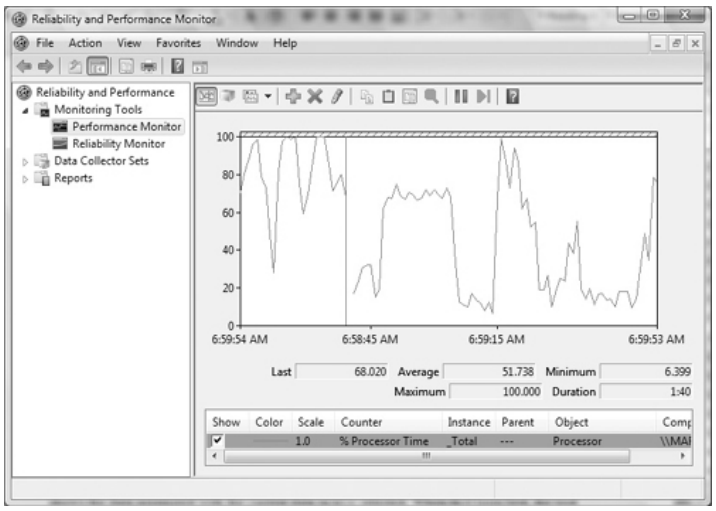
[View full size image]



The left-hand pane shows the different data sources available to the tool. As mentioned earlier, performance counters are used heavily when diagnosing .NET applications and are located under the Monitoring Tools node under Performance Monitor. The right-hand pane shows the data associated with the current data source selected. When first launched, the tool shows an overview of the system state including CPU, Disk, Memory, and Network utilization. Figure 5-15 shows the tool after the Performance Monitor item is selected.

Figure 5-15. Performance Monitor

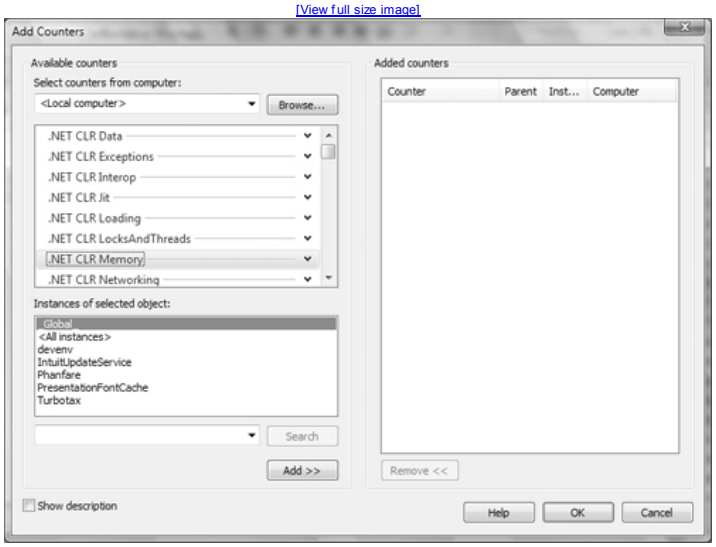
[View full size image]



The right-hand pane now displays a visual representation of the selected performance counters over time. By default, the Processor Time counter is always selected when the tool is first launched. To add counters, right-click in the right pane and select Add Counters, which brings up the Add Counters dialog shown in Figure 5-16.

Figure 5-16. Add Counters dialog

[View full size image]



The Add Counters dialog has two parts. The first part is the left side's Available Counters options, which shows a drop-down list of all available counter categories as well as the instances of the available objects that the performance counters can collect and display data on. For example, if the .NET CLR Memory performance counter category is selected, the list of available instances shows the processes that are available. The right pane simply shows all the performance counters that have been added.

Now that we know how to add and display performance counters in the tool, let's try it out on our sample application. The first question we have to answer before blindly adding random performance counters is, which CLR counters are we specifically interested in based on the symptoms we are seeing? Table 5-4 shows the available CLR performance counter categories as well as their associated descriptions.

Table 5-4. CLR-Specific Performance Counters Categories

Category	Description
.NET CLR Data	Runtime statistics on data (such as SQL) performance
.NET CLR Exceptions	Runtime statistics on CLR exception handling such as number of exceptions thrown
.NET CLR Interop	Runtime statistics on the interoperability services such as number of marshalling operations
.NET CLR Jit	Runtime statistics on the Just In Time compiler such as number of methods JITTED
.NET CLR Loading	Runtime statistics on the CLR class/assembly loader such as total number of bytes in the loader heap
.NET CLR LocksAndThreads	Runtime statistics on locks and threads such as the contention rate of a lock
.NET CLR Memory	Runtime statistics on the managed heap and garbage collector such as the number of collections in each generation
.NET CLR Networking	Runtime statistics on networking such as datagrams sent and received
.NET CLR Remoting	Runtime statistics on remoting such as remote calls per second
.NET CLR Security	Runtime statistics on security the total number of runtime checks

Based on the plethora of available categories, in our specific example, we are interested in finding out more details on the memory consumption (.NET CLR Memory) of our sample application. Table 5-5 shows the specific counters available in this category as well as their descriptions.

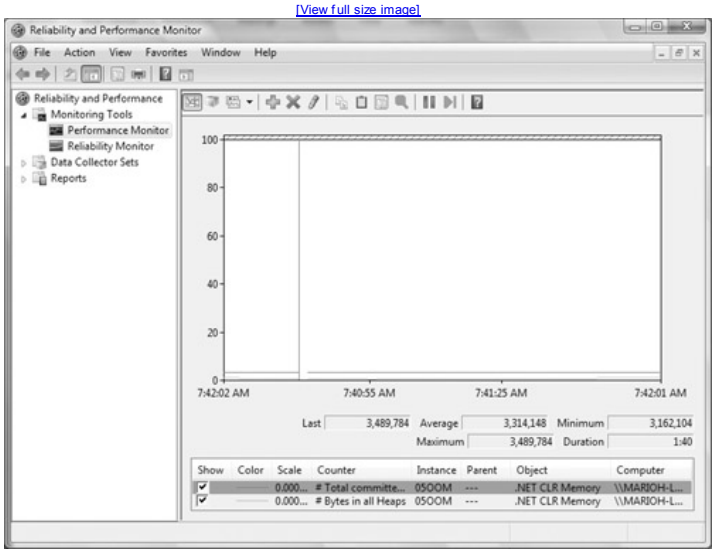
Table 5-5. .NET CLR Memory Performance Counters

Performance Counter	Description
# Bytes in all heaps	The total number of bytes in all heaps (gen 0, gen 1, gen 2, and large object heap).
# GC Handles	Total number of GC handles.
# Gen 0 collections	Total number of generation 0 garbage collections.
# Gen 1 collections	Total number of generation 1 garbage collections.
# Gen 2 collections	Total number of generation 2 garbage collections.
# Induced GC	Total number of times a call to GC.Collect has been made.
# Pinned objects	Total number of pinned objects in the managed heap during the last garbage collection. Please note that it only displays the number of pinned objects from the generations that were collected. As such, if a garbage collection resulted in only generation 0 being collected, this number only states how many pinned objects were in that generation.
# of Sink blocks in use	Current number of sync blocks in use. Useful when diagnosing performance problems related to heavy synchronization usage.
# Total committed bytes	Total number of virtual bytes committed by the CLR heap manager.
# Total reserved bytes	Total number of virtual bytes reserved by the CLR heap manager.
% Time in GC	Percentage of total elapsed time spent in the garbage collector since the last garbage collection.
Allocated bytes/sec	Number of allocated bytes per second. Updated at the beginning of every garbage collection.
Finalization Survivors	The number of garbage-collected objects that survives a collection due to waiting for finalization.
Gen 0 heap size	Maximum number of bytes that can be allocated in generation 0.
Gen 0 Promoted bytes/sec	Number of promoted bytes per second in generation 0.
Gen 1 heap size	Current number of bytes in generation 1.
Gen 1 Promoted bytes/sec	Number of promoted bytes per second in generation 1.
Gen 2 heap size	Current number of bytes in generation 1.
Large object heap size	Current size of the large object heap.
Process ID	Process identifier of process being monitored.
Promoted finalization – Memory from gen 0	The number of bytes that are promoted to generation 1 due to waiting to be finalized.
Promoted memory from Gen 0	The number of bytes promoted from generation 0 to generation 1 (minus objects that are waiting to be finalized).
Promoted memory from Gen 1	The number of bytes promoted from generation 1 to generation 2 (minus objects that are waiting to be finalized).

To monitor our sample application's memory usage, let's pick the # total bytes counter as well as the # total committed bytes counter. This can give us valuable clues as to whether the memory is on the managed heap or elsewhere in the process. Start the `05OOM.exe` application followed by launching the Windows Reliability and Performance Monitoring tool. Add the two counters and specify the `05OOM.exe` instance in the list of available instances. Figure 5-17 shows the output of the tool after about two minutes of `05OOM.exe` runtime.

Figure 5-17. Monitoring 05OOM.exe total and committed bytes

[View full size image]

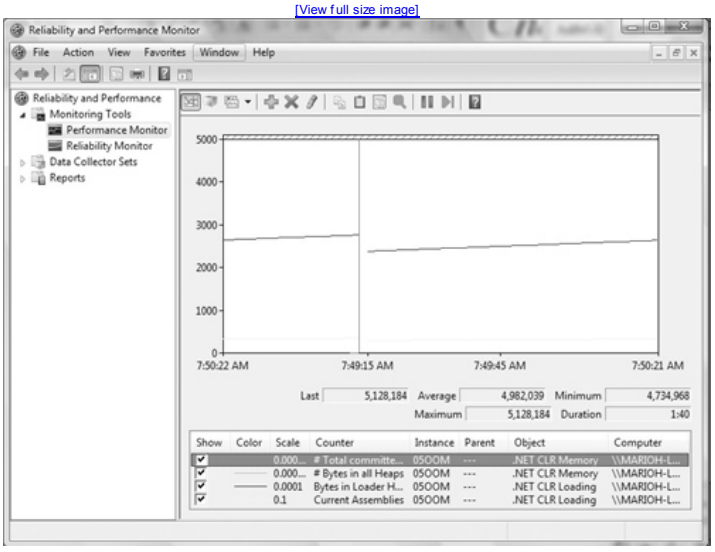


The counters look pretty stable with no major uptick. Yet, if we look at the 05OOM process in Windows Task Manager, we can see that memory consumption is increasing quite a bit. Where is the memory coming from? At this point, we have eliminated the managed heap as being the cause for

memory usage growth, and our strategy is now to use the other various counters available to see if we can spot an uptick. For example, let's choose the bytes in loader heap and current assemblies (both under the .NET CLR Loading category) and see what the output shows. (See Figure 5-18.)

Figure 5-18. Monitoring 05OOM.exe current assemblies and bytes in loader heap performance counters

[View full size image]



Note that you may have to change the vertical scale maximum (under properties) to a larger number depending on how long the application has been executing. In Figure 5-18, the vertical scale maximum has been set to 5000. This time, we can see some more interesting data. Both the bytes in loader heap and current assemblies performance counters are slowly increasing over time. One of our theories is that we are looking at a potential assembly leak. To verify this, we can attach the debugger to the `05OOM.exe` process (`ntsd -pn 05oom.exe`) and use the `eeheap -loader` command:

```
0:003> !eeheap -loader
Loader Heap:
---------------------------------------
System Domain: 7a3bc8b8
LowFrequencyHeap: Size: 0x0(0)bytes.
HighFrequencyHeap: 002a2000(8000:1000) Size: 0x1000(4096)bytes.
StubHeap: 002aa000(2000:2000) Size: 0x2000(8192)bytes.
Virtual Call Stub Heap:
  IndcellHeap: Size: 0x0(0)bytes.
  LookupHeap: Size: 0x0(0)bytes.
  ResolveHeap: Size: 0x0(0)bytes.
  DispatchHeap: Size: 0x0(0)bytes.
  CacheEntryHeap: Size: 0x0(0)bytes.
Total size: 0x3000(12288)bytes
---------------------------------------
Shared Domain: 7a3bc560
LowFrequencyHeap: 002d0000(2000:1000) Size: 0x1000(4096)bytes.
HighFrequencyHeap: 002d2000(8000:1000) Size: 0x1000(4096)bytes.
StubHeap: 002da000(2000:1000) Size: 0x1000(4096)bytes.
Virtual Call Stub Heap:
  IndcellHeap: 00870000(2000:1000) Size: 0x1000(4096)bytes.
  LookupHeap: 00875000(2000:1000) Size: 0x1000(4096)bytes.
  ResolveHeap: 0087b000(5000:1000) Size: 0x1000(4096)bytes.
  DispatchHeap: 00877000(4000:1000) Size: 0x1000(4096)bytes.
  CacheEntryHeap: 00872000(3000:1000) Size: 0x1000(4096)bytes.
Total size: 0x7000(28672)bytes
---------------------------------------
Domain 1: 304558
LowFrequencyHeap: 002b0000(2000:2000) 00ca0000(10000:10000) 01cf0000(10000:10000)
04070000(10000:10000) 04170000(10000:10000)
...
...
...
 165e0000(10000:10000) 166b0000(10000:10000) 16770000(10000:10000)
16830000(10000:10000) 16900000(10000:10000) 169c0000(10000:10000)
16a80000(10000:a000) Size: 0x16fc000(24100864)bytes.
HighFrequencyHeap: 002b2000(8000:8000) 03e50000(10000:10000) 04370000(10000:10000)
046c0000(10000:10000) 04a10000(10000:10000)
...
...
...
15bf0000(10000:10000) 15f30000(10000:10000) 16270000(10000:10000)
165a0000(10000:10000) 168f0000(10000:a000) Size: 0x572000(5709824)bytes.
StubHeap: 002ba000(2000:1000) Size: 0x1000(4096)bytes.
Virtual Call Stub Heap:
  IndcellHeap: Size: 0x0(0)bytes.
  LookupHeap: Size: 0x0(0)bytes.
  ResolveHeap: 002ca000(6000:1000) Size: 0x1000(4096)bytes.
```

```
  DispatchHeap: 002c7000(3000:1000) Size: 0x1000(4096)bytes.
  CacheEntryHeap: 002c2000(4000:1000) Size: 0x1000(4096)bytes.
Total size: 0x1c71000(29822976)bytes
-----------------------------------------
Jit code heap:
LoaderCodeHeap: 165f0000(10000:b000) Size: 0xb000(45056)bytes.
LoaderCodeHeap: 15de0000(10000:10000) Size: 0x10000(65536)bytes.
LoaderCodeHeap: 15600000(10000:10000) Size: 0x10000(65536)bytes.
...
...
...
LoaderCodeHeap: 04710000(10000:10000) Size: 0x10000(65536)bytes.
LoaderCodeHeap: 009e0000(10000:10000) Size: 0x10000(65536)bytes.
Total size: 0x23b000(2338816)bytes
-----------------------------------------
Module Thunk heaps:
Module 790c2000: Size: 0x0(0)bytes.
Module 002d2564: Size: 0x0(0)bytes.
...
...
...
Module 168f8e40: Size: 0x0(0)bytes.
Module 168f93b8: Size: 0x0(0)bytes.
Module 168f9930: Size: 0x0(0)bytes.
Total size: 0x0(0)bytes
-----------------------------------------
Module Lookup Table heaps:
Module 790c2000: Size: 0x0(0)bytes.
Module 002d2564: Size: 0x0(0)bytes.
Module 002d21d8: Size: 0x0(0)bytes.
...
...
...
Module 168f93b8: Size: 0x0(0)bytes.
Module 168f9930: Size: 0x0(0)bytes.
Total size: 0x0(0)bytes
-----------------------------------------
Total LoaderHeap size: 0x1eb6000(32202752)bytes
=========================================
```

The first two domains (system and shared) seem to look reasonable, but the default application domain has a ton of data in it. More specifically, it contains the bulk of the overall loader heap (size `32202752`). Why does the application domain contain so much data? We can get further information about the default application domain by using the `DumpDomain` command and specifying the address of the default application domain (found in output from the previous `eeheap` command):

```
0:003> !DumpDomain 304558
-----------------------------------------
Domain 1: 00304558
LowFrequencyHeap: 0030457c
HighFrequencyHeap: 003045d4
StubHeap: 0030462c
Stage: OPEN
SecurityDescriptor: 00305ab8
Name: 05OOM.exe
Assembly: 0030d1b0
[C:\Windows\assembly\GAC_32\mscorlib\2.0.0.0__b77a5c561934e089\mscorlib.dll]
ClassLoader: 002fc988
SecurityDescriptor: 0030dfd8
  Module Name
790c2000 C:\Windows\assembly\GAC_32\mscorlib\2.0.0.0__b77a5c561934e089\mscorlib.dll
002d2564 C:\Windows\assembly\GAC_32\mscorlib\2.0.0.0__b77a5c561934e089\sortkey.nlp
002d21d8 C:\Windows\assembly\GAC_32\mscorlib\2.0.0.0__b77a5c561934e089\sorttbls.nlp


Assembly: 0032f1b8 [C:\ADNDBin\05OOM.exe]
ClassLoader: 002fd168
SecurityDescriptor: 00330f30
  Module Name
002b2c3c C:\ADNDBin\05OOM.exe


Assembly: 0033bb98
[C:\Windows\assembly\GAC_MSIL\System.Xml\2.0.0.0__b77a5c561934e089\System.Xml.dll]
ClassLoader: 002fd408
SecurityDescriptor: 00326b18
  Module Name
639f8000
C:\Windows\assembly\GAC_MSIL\System.Xml\2.0.0.0__b77a5c561934e089\System.Xml.dll
...
...
...
```

```
Assembly: 00346408 [4ql4a3hq, Version=0.0.0.0, Culture=neutral,
PublicKeyToken=null]
ClassLoader: 003423a8
SecurityDescriptor: 00346380
  Module Name
002b46f8 4ql4a3hq, Version=0.0.0.0, Culture=neutral, PublicKeyToken=null
Assembly: 003465a0 [lx4qjutk, Version=0.0.0.0, Culture=neutral, PublicKeyToken=null]
ClassLoader: 00342488
SecurityDescriptor: 00346518
  Module Name
002b4ce4 lx4qjutk, Version=0.0.0.0, Culture=neutral, PublicKeyToken=null


Assembly: 003466b0 [uds1hfbo, Version=0.0.0.0, Culture=neutral, PublicKeyToken=null]
ClassLoader: 003424f8
SecurityDescriptor: 00346628
  Module Name
002b5258 uds1hfbo, Version=0.0.0.0, Culture=neutral, PublicKeyToken=null


...
...
...
```

As we can see, there are numerous assemblies loaded into the default application domain. Furthermore, the names of the assemblies seem rather random. Why are all these assemblies being loaded? Our code in Listing 5-9 certainly doesn't directly load any assemblies, which means that these assemblies have to be dynamically generated. To further investigate what these assemblies contain, we can pick one of them and dump out the associated module information using the `DumpModule` command:

```
0:003> !DumpModule 002b5258
Name: uds1hfbo, Version=0.0.0.0, Culture=neutral, PublicKeyToken=null
Attributes: PEFile
Assembly: 003466b0
LoaderHeap: 00000000
TypeDefToMethodTableMap: 00ca2df8
TypeRefToMethodTableMap: 00ca2e10
MethodDefToDescMap: 00ca2e6c
FieldDefToDescMap: 00ca2ed0
MemberRefToDescMap: 00ca2ef8
FileReferencesMap: 00ca2fec
AssemblyReferencesMap: 00ca2ff0
MetaData start address: 00cc07c8 (4344 bytes)
```

Next, we dump the metadata of the module using the `dc` command specifying the starting address and the ending address (starting address + size of metadata):

```
0:003> dc 00cc07c8 00cc07c8+0n4344
00cc07c8  424a5342 00010001 00000000 0000000c  BSJB............
00cc07d8  302e3276 3730352e 00003732 00050000  v2.0.50727......
00cc07e8  0000006c 00000528 00007e23 00000594  l...(...#~......
00cc07f8  0000077c 72745323 73676e69 00000000  |...#Strings....
...
...
...
00cc0d58  00000000 6f4d3c00 656c7564 6475003e  .....<Module>.ud
00cc0d68  66683173 642e6f62 58006c6c 65536c6d  s1hfbo.dll.XmlSe
00cc0d78  6c616972 74617a69 576e6f69 65746972  rializationWrite
00cc0d88  72655072 006e6f73 7263694d 666f736f  rPerson.Microsof
00cc0d98  6d582e74 65532e6c 6c616972 74617a69  t.Xml.Serializat
00cc0da8  2e6e6f69 656e6547 65746172 7734164   ion.GeneratedAss
00cc0db8  6c626d65 6d580079 7265536c 696c6169  embly.XmlSeriali
00cc0dc8  6974617a 65526e6f 72656461 73726550  zationReaderPers
00cc0dd8  58006e6f 65536c6d 6c616972 72657a69   on.XmlSerializer
00cc0de8  65500031 6e6f7372 69726553 7a696c61  1.PersonSerializ
00cc0df8  58007265 65536c6d 6c616972 72657a69  er.XmlSerializer
00cc0e08  746e6f43 74636172 73795300 2e6d6574  Contract.System.
00cc0e18  006c6d58 74737953 582e6d65 532e6c6d  Xml.System.Xml.S
00cc0e28  61697265 617a696c 6e6f6974 6c6d5800  erialization.Xml
00cc0e38  69726553 7a696c61 6f697461 6972576e  SerializationWri
00cc0e48  00726574 536c6d58 61697265 617a696c  ter.XmlSerializa
00cc0e58  6e6f6974 64616552 58007265 65536c6d  tionReader.XmlSe
00cc0e68  6c616972 72657a69 6c6d5800 69726553  rializer.XmlSeri
...
...
...
```

Now we are getting somewhere. From the output of the metadata, we can see that the module associated with the assembly contains references to some form of XML serialization. Furthermore, it seems that the module contains XML serialization types that are specific to the serialization of the `Person` class in our code. Based on this evidence, we can now hypothesize that the XML serialization code in our application is causing all of these dynamic

assemblies to be generated. The next step is the documentation for the `XmlSerializer` class. MSDN clearly states that using the `XmlSerializer` class for performance reasons may in fact create a specialized dynamic assembly to handle the serialization. More specifically, seven of the `XmlSerializer` constructors result in dynamic assemblies being generated, whereas the remaining two have reuse logic that reduces the number of dynamic assemblies.

The preceding scenario illustrates how we can use the Windows Task Manager to monitor the overall memory usage of a .NET application and the Windows Reliability and Performance Monitor tool to drill down into the CLR specifics. The scenario assumes that we had the luxury of running and monitoring the application live. In many cases, the application simply runs until it runs out of memory and throws an `OutOfMemoryException`. If we let our sample application run indefinitely, the `OutOfMemoryException` would have been reported as follows:

```
(1830.1f20): CLR exception - code e0434f4d (first/second chance not available)
eax=0027ed2c ebx=e0434f4d ecx=00000001 edx=00000000 esi=0027edb4 edi=00338510
eip=775842eb esp=0027ed2c ebp=0027ed7c iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
*** ERROR: Symbol file could not be found.  Defaulted to export symbols for
kernel32.dll -
kernel32!RaiseException+0x58:
```

As discussed earlier, to get further information on the managed exception, we can use the `PrintException` command:

```
0:000> kb
ChildEBP RetAddr  Args to Child
0027ed7c 79f071ac e0434f4d 00000001 00000001 kernel32!RaiseException+0x58
0027eddc 79f0a780 51e10dac 00000001 00000000
 mscorwks!RaiseTheExceptionInternalOnly+0x2a8
*** WARNING: Unable to verify checksum for System.ni.dll
0027ee80 7a53e025 0027f14c 79f0a3d9 0027f338 mscorwks!JIT_Rethrow+0xbf
0027ef4c 7a53d665 51df597c 00000000 51de0050 System_ni+0xfe025
0027ef80 7a4d078a 51df597c 51de0050 638fcb39 System_ni+0xfd665
*** WARNING: Unable to verify checksum for System.Xml.ni.dll
0027efec 638fb6e5 00000000 51de02cc 00000000 System_ni+0x9078a
0027f078 638fa683 51ddff88 00000000 51de02cc System_Xml_ni+0x15b6e5
0027f09c 63960d09 00000000 00000000 00000000 System_Xml_ni+0x15a683
0027f0c4 6396090c 00000000 00000000 00000000 System_Xml_ni+0x1c0d09
0027f120 79e7c74b 00000000 0027f158 0027f1b0 System_Xml_ni+0x1c090c
00000000 00000000 00000000 00000000 00000000 mscorwks!CallDescrWorker+0x33
0:000> !PrintException 51e10dac
Exception object: 51e10dac
Exception type: System.OutOfMemoryException
Message: <none>
InnerException: <none>
StackTrace (generated):
    SP       IP       Function
    0027EE94 7942385A mscorlib_ni!System.Reflection.Assembly.Load
(Byte[], Byte[], System.Security.Policy.Evidence)+0x3a
    0027EEB0 7A4BF513 System_ni!Microsoft.CSharp.CSharpCodeGenerator.FromFileBatch
(System.CodeDom.Compiler.CompilerParameters, System.String[])+0x3ab
    0027EF00 7A53E025 System_ni!Microsoft.CSharp.CSharpCodeGenerator.FromSourceBatch
(System.CodeDom.Compiler.CompilerParameters, System.String[])+0x1f1
    0027EF58 7A53D665 System_ni!Microsoft.CSharp.CSharpCodeGenerator.System.CodeDom.
Compiler.ICodeCompiler.CompileAssemblyFromSourceBatch
(System.CodeDom.Compiler.CompilerParameters, System.String[])+0x29
    0027EF8C 7A4D078A System_ni!System.CodeDom.Compiler.CodeDomProvider.
CompileAssemblyFromSource(System.CodeDom.Compiler.CompilerParameters,
System.String[])+0x16
    0027EF98 638FCB39 System_Xml_ni!System.Xml.Serialization.Compiler.Compile
(System.Reflection.Assembly, System.String,
System.Xml.Serialization.XmlSerializerCompilerParameters,
System.Security.Policy.Evidence)+0x269
    0027F000 638FB6E5
System_Xml_ni!System.Xml.Serialization.TempAssembly.GenerateAssembly
(System.Xml.Serialization.XmlMapping[], System.Type[], System.String,
System.Security.Policy.Evidence,
System.Xml.Serialization.XmlSerializerCompilerParameters,
System.Reflection.Assembly, System.Collections.Hashtable)+0x7e9
    0027F094 638FA683 System_Xml_ni!System.Xml.Serialization.TempAssembly..ctor
(System.Xml.Serialization.XmlMapping[], System.Type[], System.String, System.String,
System.Security.Policy.Evidence)+0x4b
    0027F0B4 63960D09 System_Xml_ni!System.Xml.Serialization.XmlSerializer..ctor
(System.Type, System.Xml.Serialization.XmlAttributeOverrides, System.Type[],
System.Xml.Serialization.XmlRootAttribute, System.String, System.String,
System.Security.Policy.Evidence)+0xed
    0027F0E4 6396090C System_Xml_ni!System.Xml.Serialization.XmlSerializer..ctor
                                          (System.Type, System.Xml.Serialization.XmlRootAttribute)+0x28
                                          0027F0F4 009201D6 05OOM!Advanced.NET.Debugging.Chapter5.OOM.Run()+0xe6
                                          0027F118 009200A7
                                          05OOM!Advanced.NET.Debugging.Chapter5.OOM.Main(System.String[])+0x37StackTraceString: <non
HResult: 8007000e
There are nested exceptions on this thread. Run with -nested for details
```

At this point, the application has already failed and we can't rely on runtime monitoring tools to gauge the application's memory usage. In situations like this, we have to rely solely on the debugger commands to analyze where the memory is being consumed. Unfortunately, there is no single cookbook recipe on the exact commands and steps to take, but as a general rule of thumb, utilizing the various diagnostics commands (such as `eeheap`, `dumpheap`, `dumpdomain`, etc.) can give invaluable clues as to where in the CLR the memory is being consumed. The excessive memory consumption can, of course, also be as a result of a native code leak, which we will see an example of in Chapter 7, "Interoperability."

Immediately Break on OutOfMemoryException

When a process gets into a situation where it is running out of memory, things can get very tricky and the application may not be able to properly handle the condition. Because an `OutOfMemoryException` gets propagated up the chain and does not fault the process until the exception is deemed unhandled, a lot of code may still get executed as part of the unwinding making troubleshooting more difficult in certain situations. Furthermore, if the code is hosted in a process that it does not own, the process may catch all kinds of exceptions and continue running. To ensure that an `OutOfMemoryException` always breaks under the debugger, the CLR introduced a registry value called `GCBreakOnOOM` (DWORD) under the following registry path: `HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\.NET Framework`. The value can be set to 1, in which case an event log message is logged; it can be set to 2, in which case the out of memory condition causes a break in the debugger; or it can be set to 4, in which case a more extensive event log is written that includes memory statistics at the point where the out of memory condition was encountered.