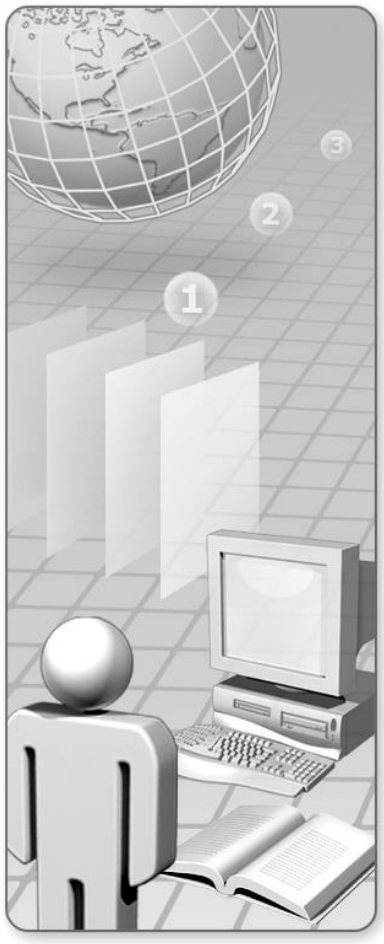
Microsoft | Services

Troubleshooting .NET Memory Issues



Lab: Troubleshooting .NET Memory Issues

Objectives

After completing this lab, you will be able to:

- Have a greater understanding, from a debugging perspective, of how .NET manages memory.
- Use Perfmon to identify a managed memory leak.
- Use WinDBG and SOS to determine where managed memory is increasing and perform further troubleshooting to find the root cause of the problem.
- Stress an ASP.NET application, and troubleshoot a high memory situation in that application.

Estimated time to complete this lab: 90 minutes

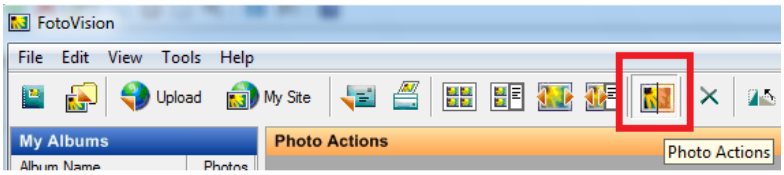
Note: This lab focuses on the concepts in this module and as a result might not comply with Microsoft® security recommendations.

Exercise 1: Diagnosing and Debugging Memory Increase

In this exercise, you will use Perfmon to diagnose an increase in memory and determine whether it is a managed or native issue. From there, you will use WinDBG and SOS to debug the issue and find the root cause.

Task Description

1. 0.Open FotoVision.
   A. In Windows Explorer, go to <FotoVisionDir>\C#\Desktop\Bin and double-click **FotoVision.exe**.
2. Start Perfmon.
   A. On the Start menu, click Run, and then type **<FotoVisionDir>\C#\Desktop\Lab3_Ex1.msc**
3. Play with the application so the Perfmon counters are getting updated.
   A. Double click on few of the photos
   B. Click on the Photo Actions button :



   C. Click the Grayscale button
   D. Make sure the Perfmon counters like # Gen 0 Collections have some values. (Check the Last value in Perfmon)
   E. Make sure you do not save the changes when exiting the application you should press the Reset button to avoid the save.
4. Attach WinDBG and load SOS.
   A. On the Start menu, click Run and then type: **WinDBG -pn FotoVision.exe**
   B. Click OK.
   C. Load SOS.

5. Look at the managed heap(s) and document the size of each generation in each heap.

   A. Run the "**!eeheap -gc**" command.

```
0:009> !eeheap -gc
Number of GC Heaps: 1
generation 0 starts at 0x01f5819c
generation 1 starts at 0x01f1100c
generation 2 starts at 0x01f11000
ephemeral segment allocation context: none
 segment     begin allocated  size
01f10000  01f11000  01f941a8  0x831a8(537000)
Large object heap starts at 0x02f11000
 segment     begin allocated  size
02f10000  02f11000  02f19650  0x8650(34384)
Total Size:              Size: 0x8b7f8 (571384) bytes.
------------------------------
GC Heap Size:            Size: 0x8b7f8 (571384) bytes.
```

   B. Based on your output, what is the size of each generation?

      Gen 0 Size: _____

      Gen 1 Size: _____

      Gen 2 Size: _____

   C. Whats the size of the Large Object Heap:

      LOH Size: _____

> **Help:** For the above output generation sizes are calculated the following way:
>
> Gen 0 size: 0x01f941a8 - 0x01f5819c = 0x3C00C which is **245772** in decimal
>
> Gen 1 size: 0x01f5819c - 0x01f1100c = 0x47190 which is **291216** in decimal
>
> Gen 2 size: 0x01f1100c - 0x01f11000 = 0xC which is **12** in decimal
>
> **Note:** You have to do the same calculation for the output in your WinDbg session. Please note that the values might be different for each and every run.

   D. Now that you know how to calculate the generation sizes manually, note that **Psscor2**/**Psscor4** and **SosEx** have a command to do this.
   Run **!SosEx.dumpgen X** command, where X is 0, 1, 2. See if it matches what you calculated in the previous step. Gen0 may be off just a bit.

E. There's another heap managed by .NET—the Loader heap. Run the "**!eeheap -loader**" command to find its size and then document it here:

   Loader Heap Size: _____

6. Compare Perfmon to the output of the "!eeheap" command.

   A. *What does Perfmon report as the amount of memory in the Large Object Heap (LOH)? Does this match with what you wrote down in the previous step?*

   B. *What does Perfmon report as the amount of memory in Gen2? Is this the same number that you wrote down in the previous step?*

   C. *What does Perfmon report as the amount of memory in Gen1? Is this the same number that you wrote down in the previous step?*

   D. *What does Perfmon report as the amount of memory in Gen0? Is this the same number that you wrote down in the previous step? If not, why not?*
   Hint: to see what "Gen 0 Heap Size" really represents, see the Perfmon Help on the topic:

      i. In Perfmon, click the **+** button to display the Counter dialog box.

      ii. In the drop-down list, click the .**NET CLR Memory** object.

      iii. In the list of counters, click **Gen 0 Heap Size**.

      iv. Click the **Show description** checkbox to get more information about the counters. Compare the **Gen 0 Heap Size** description to the descriptions for the Gen1 and Gen2 Heap Size.

7. Record the number of each generational GC collection.

   A. Go to Perfmon and record the values of each counter:

      # Gen 0 Collections: _____

      # Gen 1 Collections: _____

      # Gen 2 Collections: _____

   B. Go back to WinDBG. In the command line, type "**g**" to release control to begin using the application.

8. Create and populate a new photo album.

   A. In FotoVision, click **New Album** on the File menu to create a new album. Make sure to select the album using the mouse to make the right pane appear.

   B. On the right-hand side of FotoVision, name the new album "**Misc**" (without quotes). Be sure to use this exact name, because this lab depends on the album name to be "Misc."

C. On the File menu, click **Import Photos**. Browse to **<FotoVisionDir>\C#\Desktop**, click **Mountains_spitzer_f.jpg**, and then click **Open.**

Click OK on the informative message:

D. On the FotoVision toolbar, click the **Upload** button to upload the albums and their photos to the Web site.

9. Stop Perfmon, break in with WinDBG, and see what has changed in the managed heap(s).

   A. In Perfmon, click the **Pause** button on the toolbar to stop tracking and in WinDBG, click Break on the Debug menu to pause the application.

   B. With Perfmon stopped, you can compare new numbers to old numbers without overwriting the old numbers. Write down the "Last" values recorded by Perfmon for the counters:

      # Gen 0 Collections: _____

      # Gen 1 Collections: _____

      # Gen 2 Collections: _____

      Gen 0 Size: _____

      Gen 1 Size: _____

      Gen 2 Size: _____

      LOH Size: _____

      Comparing the values before and after you uploaded the new album, which generation increased the most, and by how much?

      Generation: _____

      Amount of increase: _____

> **Note:** Keep in mind that each iteration of this exercise will provide varying results over time.

   C. Run the "**!eeheap -gc**" command and compare the output to the previous output. *Does the growth that you saw in Perfmon match what you see in this output?* In other words, if Perfmon shows that the number of Gen1 GCs changed, then you should note a change in the beginning address of Gen 0 and Gen 1. If Perfmon shows a large drop in the LOH heap size, then you should also see an increase in the number of Gen2Collections as well as a change in the beginning address of Gen2, Gen1, and Gen0.

10. Allocate more memory and view the changes in WinDBG.

    A. In WinDBG, type "**g**" to release the debugger and allow process execution to continue.

B. On the Perfmon toolbar, click the **Play** button to allow Perfmon to log statistics again.

C. In FotoVision, import the additional photos from the **<FotoVisionDir>\C#\Desktop** directory and put them in the **Misc** album.

D. On the FotoVision toolbar, click the **Upload** button again and note the changes in the Perfmon counters.

Based on your observations, what can you theorize about what the application is doing in the background to cause the pattern you see in Perfmon and WinDBG? In other words, is the application making a large number of tiny native allocations, a small number of large managed allocations, and so on?

Look at **Private Bytes**, **# Bytes In All Heaps**, **!DumpHeap** data, and so forth to formulate a theory.

E. Close perfmon, windbg & FotoVision.

Exercise 2: Sudden Memory Increase

In this exercise, you will use the steps outlined in exercise 1 to do the initial troubleshooting on an application that experiences a sharp and sudden increase in the memory footprint. Then you'll debug further to find the root cause of the problem.

Task Description

1. 0.From this point on in the lab manual the installation directory will be referred to as <FotoVisionWeb>.

   <FotoVisionWeb>=http://localhost/FotoVision40

2. Load the Web site.

   A. On the Start menu, click **Run**, and then type **<FotoVisionWeb>**.

   B. If you followed the steps correctly in Exercise 1, you should see the Misc album with your photos in it.

3. Configure Perfmon to monitor current activity.

   A. On the Start menu, click **Run**, and then type "**Perfmon**".

   B. Capture the following counters indicated in the following table. To make things simple, be sure to do this on the right instance of your ASP.NET worker process (w3wp.exe):

   If there are more than one instances of w3wp you need to identify the right one. Use the following command to identify your worker process:

   (a) IIS 7.0/7.5 – C:\Windows\System32\inetsrv**appcmd list wp**

   in combination with C:\Windows\system32\inetsrv**appcmd list app**

   Note: You will need to use an elevated command prompt

   (b) IIS 6.0 – C:\Windows\System32\cscript iisapp.vbs

   And Also make sure to add ".Net CLR Memory\Process ID" counter
   or "Process\ID Process "counter
   which will show you which w3wp instance is using which PID.

PERFORMANCE OBJECT	COUNTER
Process	Private Bytes

PERFORMANCE OBJECT	COUNTER
.NET CLR Memory	# Bytes in all Heaps
.NET CLR Memory	# Gen 0 Collections
.NET CLR Memory	# Gen 1 Collections
.NET CLR Memory	# Gen 2 Collections
.NET CLR Memory	# Total Committed Bytes
.NET CLR Memory	Gen 0 Heap Size
.NET CLR Memory	Gen 1 Heap Size
.NET CLR Memory	Gen 2 Heap Size
.NET CLR Memory	Large Object Heap Size

C. Note that you may have to adjust the scale for a few of the counters—in addition to the graph's maximum limit (60 is good place to start)—in order to be able to view all the counters at once in an easy-to-read format. Once you have added all the counters.
Also make sure Private bytes & Bytes in All heaps use the same scale.

4. View the Perfmon counters and document their values.

   A. After 10 seconds or so of gathering data with Perfmon, pause logging so that you can take your time viewing which counters changed and by how much.

   B. Document the value of the counters. Use the value that Perfmon lists as "Last" for each counter.

      Private Bytes: _____

      #Bytes in All Heaps: _____

5. Browse the Misc album.

   A. Restart the Perfmon logging by hitting the Start button.

   On the home page, click the word "**Misc**," which represents the album you created earlier. Doing this will allow you to view all the photos in the album.

6. View Perfmon.

   A. Switch over to Perfmon and view the graph. After 10 seconds or so, pause logging so that you can take your time viewing which counters changed and by how much.

   B. Document the value of the counters. Use the value that Perfmon lists as "Last" for each counter.

Private Bytes: _____

#Bytes in All Heaps: _____

Did the value of either counter change? If so, based on the change, would you say that this is a managed memory issue or a native memory issue?

Amount of increase in Private Bytes: _____

Amount of increase in #Bytes in All Heaps: _____

Amount of increase in native only memory: _____

If necessary, change the graph maximum to get a better view.

C. Look through the rest of the counters to see if you can make a more specific determination of where the memory growth occurs.
Is there a specific generation, perhaps?

7. Reproduce the issue.

   A. Click the **Play** button in Perfmon to allow it to continue logging data.

   B. Back in the FotoVision Web application; click the **Land** album on the left.

   C. Click the **Misc** album once more.

   D. Back in Perfmon, have the counters changed from where you last viewed them? Write down the values of the corresponding amounts as compared to the values you recorded in the previous step:

   Amount of increase in Private Bytes: _____

   Amount of increase in #Bytes in All Heaps: _____

   Amount of increase in native only memory: _____

   Does this change your opinion on where the memory growth might be concentrated?
   This reproduction of the problem illustrates that you often need more than just one sampling of data for a memory issue. In production applications, you typically need a long Perfmon log (that is, more than 5 or 6 hours) to get a better grasp on the application's memory pattern. But in our simplified lab application, one minute should suffice to demonstrate the tactics used to diagnose and troubleshoot a problem of sudden memory growth in managed memory.

8. Dump the process.

   Perfmon has told us about as much as it can regarding the sudden memory growth. While it can tell us when, how much, and—to a certain degree—where the memory growth occurred, it cannot tell us the source of the problem.

A. Open an admin command prompt by typing **cmd** after clicking the Start Menu and right-clicking "Run as administrator" on cmd.exe.

B. Refer back to Step 3B if there is more than one instance of w3wp running to identify the right one.

C. In the command window, type (please do **not** copy out of the description)

procdump –accepteula –ma <PID_of_target_w3wp.exe> c:\temp

This will create a mini dump with heap of the target process and store it in c:\temp.

> **Note**:
>
> procdump can be found on https://technet.microsoft.com/de-de/sysinternals/dd996900 and is part of the Microsoft Sysinternals Suite https://technet.microsoft.com/de-de/sysinternals
>
> Make sure that the target directory c:\temp exist.

Based on the current memory footprint of the process it might take long time to get the dump created.

Note: One dump file may be all that is needed to find the source of the issue. Generally speaking, it takes much less time to resolve a managed memory issue than a native memory leak.

9. Open the dump file.

A. On the Start menu, click **Run**, and then type "**WinDBG**".

B. On the **File** menu, click **Open Crash Dump**. (If at any time you are prompted to save your workspace, click "**No**." Browse to the dump file you just created and open it.

C. Load SOS.

10. Begin dump analysis.

> **Note:** The numbers you find in your analysis and the numbers presented here as a result of some SOS commands will likely vary from trial to trial. So don't worry if your SOS output doesn't match exactly what is printed here.

A. Look at the managed heaps, because that's where Perfmon says the memory is. Do this by using the "**!eeheap -gc**" command. (See output below)

```
0:000> !eeheap -gc
Number of GC Heaps: 2
------------------------------
Heap 0 (01334ef0)
generation 0 starts at 0x01a9e424
generation 1 starts at 0x01a975e0
generation 2 starts at 0x01a40038
ephemeral segment allocation context: none
 segment     begin allocated  size
01a40000  01a40038  01ad6430  0x963f8(615416)
Large object heap starts at 0x09a40038
 segment     begin allocated  size
09a40000  09a40038  0b15d268  0x171d230(24236592)
0e9f0000  0e9f0038  107f0068  0x1e00030(31457328)
12af0000  12af0038  148f0068  0x1e00030(31457328)
109f0000  109f0038  127f0078  0x1e00040(31457344)
18b30000  18b30038  19a30058  0xf00020(15728672)
Heap Size:               Size: 0x80b36e8 (134952680) bytes.
------------------------------
Heap 1 (01336258)
generation 0 starts at 0x05adf110
generation 1 starts at 0x05ad9900
generation 2 starts at 0x05a40038
ephemeral segment allocation context: none
 segment     begin allocated  size
05a40000  05a40038  05ae911c  0xa90e4(692452)
Large object heap starts at 0x0ba40038
 segment     begin allocated  size
0ba40000  0ba40038  0c940058  0xf00020(15728672)
14af0000  14af0038  168f0068  0x1e00030(31457328)
16b30000  16b30038  18930068  0x1e00030(31457328)
Heap Size:               Size: 0x4ba9164 (79335780) bytes.
------------------------------
GC Heap Size:            Size: 0xcc5c84c (214288460) bytes
```

Which part of the heap (.NET Heap, Large Object Heap?) does contains most of the memory?
Does this agree with the Perfmon output?

B. To find the types that make up these large allocations, dump the heap and specify a minimum size based on the output of the "!eeheap -gc" command. Use the "**!DumpHeap**" command to do this. Remember, you can always run the "!help dumpheap" command to find out more about the command and its options.
eg: !dumpheap –min 85000 –live will give you all objects >= 85000 in size which are **alive** (that means rooted).

```
0:000> 0:000> !dumpheap -min 85000 -live
 Address       MT     Size
1f2068c0 73c95670 15728652
2a1f1010 73c95670 15728652
50241010 73c95670 15728652
69651010 73c95670 15728652
201f1020 73c95670 15728652
06791010 73c95670 15728652
36871010 73c95670 15728652
4bab1010 73c95670 15728652
```

```
52351010 73c95670 15728652
212624a8 73c95670 15728652
281f1010 73c95670 15728652
67da1010 73c95670 15728652
6b651010 73c95670 15728652
221f1020 73c95670 15728652
0b791010 73c95670 15728652
30d61010 73c95670 15728652
231f1020 73c95670 15728652
07791010 73c95670 15728652
38ab1010 73c95670 15728652
5b041010 73c95670 15728652
6a651010 73c95670 15728652
….
Statistics:
      MT    Count    TotalSize Class Name
019c68e8        3      2652926      Free
73c95670       17    177419259 System.Byte[]
Total 88 objects
Fragmented blocks larger than 0.5 MB:
    Addr     Size      Followed by
132400c0    0.5MB         132ca4f0 System.Threading.WaitCallback
1d2b2348    1.1MB         1d3c1664 System.Threading.WaitCallback
```

Note:
You can figure out what generation an object is in by using he **"!eeheap -gc**" output and locating which address range the object falls into.

Alternatively use **!SosEx.gcgen** <address>

System.Byte[] is the culprit. Unfortunately, this is a common type when it comes to large allocations. But at least we can see a pattern—the large objects seem to be two different sizes instead of a range of different sizes. This may make it easier to find where the problem is occurring in your code.

11. To obtain more information about one of these objects, take a look at its root tree.

    A. Pick one of the objects in the output of your previous command and use the "**!gcroot**" command on it. Note that not all the objects may show a root tree if you are **not** using –live for dumpheap, so you may have to try this command on more than one object address before you get a tree. What you're looking for is a root tree with a strong handle.

```
0:000> 0:000> !gcroot 6a651010
HandleTable:
    028212f4 (pinned handle)
    -> 1f1fc4e0 System.Object[]
    -> 19295c48 FotoVisionWeb.FileManager+FileManagerCache
    -> 19295c54 System.Collections.Hashtable
    -> 1139b4d8 System.Collections.Hashtable+bucket[]
    -> 1d2abe98 System.IO.MemoryStream
    -> 6a651010 System.Byte[]
Found 1 unique roots (run '!GCRoot -all' to see all roots).
```

Look at this output and determine the name of the custom component, located near the top of the tree that is rooting the hash table.

If you see that the byte array is only rooted by a RefCnt handle try another byte array to see if you can find a more interesting root.

B. To learn more about this custom object before you delve into the source to find where the problem originates, dump the FileManagerCache object using !dumpobject or !do for short.

```
0:000> !do 19295c48
Name:        FotoVisionWeb.FileManager+FileManagerCache
MethodTable: 03fe483c
EEClass:     03dbe748
Size:        12(0xc) bytes
File:        C:\Windows\Microsoft.NET\Framework\v4.0.30319\Temporary ASP.NET
Files\fotovision40\2966e65a\37e9cde9\App_Code.uzxhehcf.dll
Fields:
      MT    Field   Offset                 Type VT     Attr    Value Name
73c945e8  400002b        4 ...ections.Hashtable  0 instance 19295c54 imageShop
```

Of what type is imageShop?

C. Now dump imageShop to see what else you can learn about it:

```
0:000> !do 19295c54
Name:        System.Collections.Hashtable
MethodTable: 73c945e8
EEClass:     738eedfc
Size:        52(0x34) bytes
File:
C:\WINDOWS\Microsoft.Net\assembly\GAC_32\mscorlib\v4.0_4.0.0.0__b77a5c561934e089\m
scorlib.dll
Fields:
      MT    Field   Offset                 Type VT     Attr    Value Name
73c91238  4000c66        4 ...ashtable+bucket[]  0 instance 1139b4d8 buckets
73c93b04  4000c67       18         System.Int32  1 instance       34 count
73c93b04  4000c68       1c         System.Int32  1 instance        6 occupancy
73c93b04  4000c69       20         System.Int32  1 instance       64 loadsize
73c8b278  4000c6a       24        System.Single  1 instance 0.720000 loadFactor
73c93b04  4000c6b       28         System.Int32  1 instance       38 version
73c8b370  4000c6c       2c       System.Boolean  1 instance        0
isWriterInProgress
7388c7d8  4000c6d        8 ...tions.ICollection  0 instance 00000000 keys
7388c7d8  4000c6e        c ...tions.ICollection  0 instance 00000000 values
73c867e0  4000c6f       10 ...IEqualityComparer  0 instance 00000000 _keycomparer
73c92554  4000c70       14        System.Object  0 instance 00000000 _syncRoot
```

Based on the output of this command, how many entries are in the hash table?

To get the above information use the !DumpArray (!da for short) command on the **buckets** address in the Hashtable output from the previous step:

```
0:000> !da -details 1139b4d8
Name:        System.Collections.Hashtable+bucket[]
MethodTable: 73c91238
EEClass:     7399f7a0
```

```
Size:        1080(0x438) bytes
Array:       Rank 1, Number of elements 89, Type VALUETYPE
Element Methodtable: 73c9127c
[0] 1139b4e0
    Name:        System.Collections.Hashtable+bucket
    MethodTable: 73c9127c
    EEClass:     7399f82c
    Size:        20(0x14) bytes
    File:
C:\WINDOWS\Microsoft.Net\assembly\GAC_32\mscorlib\v4.0_4.0.0.0__b77a5c561934e089\mscorlib.dll
    Fields:
              MT    Field   Offset                 Type VT     Attr    Value Name
        73c92554  4000c7a        0            System.Object      0     instance     00000000     key
        73c92554  4000c7b        4            System.Object      0     instance     00000000     val
        73c93b04  4000c7c        8             System.Int32      1     instance            0     hash_coll
[1] 1139b4ec
    Name:        System.Collections.Hashtable+bucket
    MethodTable: 73c9127c
    EEClass:     7399f82c
    Size:        20(0x14) bytes
    File:
C:\WINDOWS\Microsoft.Net\assembly\GAC_32\mscorlib\v4.0_4.0.0.0__b77a5c561934e089\mscorlib.dll
    Fields:
              MT    Field   Offset                 Type VT     Attr    Value Name
        73c92554  4000c7a        0            System.Object      0     instance     00000000     key
        73c92554  4000c7b        4            System.Object      0     instance     00000000     val
        73c93b04  4000c7c        8             System.Int32      1     instance            0     hash_coll
[2] 1139b4f8
    Name:        System.Collections.Hashtable+bucket
    MethodTable: 73c9127c
    EEClass:     7399f82c
    Size:        20(0x14) bytes
    File:
C:\WINDOWS\Microsoft.Net\assembly\GAC_32\mscorlib\v4.0_4.0.0.0__b77a5c561934e089\mscorlib.dll
    Fields:
              MT    Field   Offset                 Type VT     Attr    Value Name
        73c92554  4000c7a        0            System.Object      0     instance     112975cc     key
        73c92554  4000c7b        4            System.Object      0     instance     1128457c     val
        73c93b04  4000c7c        8             System.Int32      1     instance     -494864451     hash_coll
[3] 1139b504
    Name:        System.Collections.Hashtable+bucket
    MethodTable: 73c9127c
    EEClass:     7399f82c
    Size:        20(0x14) bytes
    File:
C:\WINDOWS\Microsoft.Net\assembly\GAC_32\mscorlib\v4.0_4.0.0.0__b77a5c561934e089\mscorlib.dll
    Fields:
              MT    Field   Offset                 Type VT     Attr    Value Name
        73c92554  4000c7a        0            System.Object      0     instance     00000000     key
        73c92554  4000c7b        4            System.Object      0     instance     00000000     val
        73c93b04  4000c7c        8             System.Int32      1     instance            0     hash_coll
[4] 1139b510
    Name:        System.Collections.Hashtable+bucket
    MethodTable: 73c9127c
    EEClass:     7399f82c
    Size:        20(0x14) bytes
    File:
C:\WINDOWS\Microsoft.Net\assembly\GAC_32\mscorlib\v4.0_4.0.0.0__b77a5c561934e089\mscorlib.dll
```

```
    Fields:
              MT    Field   Offset                 Type VT     Attr    Value Name
        73c92554  4000c7a        0            System.Object      0     instance     19307638     key
        73c92554  4000c7b        4            System.Object      0     instance     192fd574     val
        73c93b04  4000c7c        8             System.Int32      1     instance     694613943     hash_coll
[5] 1139b51c
    Name:        System.Collections.Hashtable+bucket
    MethodTable: 73c9127c
    EEClass:     7399f82c
    Size:        20(0x14) bytes
    File:
C:\WINDOWS\Microsoft.Net\assembly\GAC_32\mscorlib\v4.0_4.0.0.0__b77a5c561934e089\mscorlib.dll
    Fields:
              MT    Field   Offset                 Type VT     Attr    Value Name
        73c92554  4000c7a        0            System.Object      0     instance     00000000     key
        73c92554  4000c7b        4            System.Object      0     instance     00000000     val
        73c93b04  4000c7c        8             System.Int32      1     instance            0     hash_coll
[6] 1139b528
    Name:        System.Collections.Hashtable+bucket
    MethodTable: 73c9127c
    EEClass:     7399f82c
    Size:        20(0x14) bytes
    File:
C:\WINDOWS\Microsoft.Net\assembly\GAC_32\mscorlib\v4.0_4.0.0.0__b77a5c561934e089\mscorlib.dll
    Fields:
              MT    Field   Offset                 Type VT     Attr    Value Name
        73c92554  4000c7a        0            System.Object      0     instance     00000000     key
        73c92554  4000c7b        4            System.Object      0     instance     00000000     val
        73c93b04  4000c7c        8             System.Int32      1     instance            0     hash_coll
[7] 1139b534
    Name:        System.Collections.Hashtable+bucket
    MethodTable: 73c9127c
    EEClass:     7399f82c
    Size:        20(0x14) bytes
    File:
C:\WINDOWS\Microsoft.Net\assembly\GAC_32\mscorlib\v4.0_4.0.0.0__b77a5c561934e089\mscorlib.dll
    Fields:
              MT    Field   Offset                 Type VT     Attr    Value Name
        73c92554  4000c7a        0            System.Object      0     instance     1128dae0     key
        73c92554  4000c7b        4            System.Object      0     instance     1124e1e0     val
        73c93b04  4000c7c        8             System.Int32      1     instance     694613946     hash_coll
[8] 1139b540
    Name:        System.Collections.Hashtable+bucket
    MethodTable: 73c9127c
    EEClass:     7399f82c
    Size:        20(0x14) bytes
    File:
C:\WINDOWS\Microsoft.Net\assembly\GAC_32\mscorlib\v4.0_4.0.0.0__b77a5c561934e089\mscorlib.dll
    Fields:
              MT    Field   Offset                 Type VT     Attr    Value Name
        73c92554  4000c7a        0            System.Object      0     instance     00000000     key
        73c92554  4000c7b        4            System.Object      0     instance     00000000     val
        73c93b04  4000c7c        8             System.Int32      1     instance            0     hash_coll
[9] 1139b54c
    Name:        System.Collections.Hashtable+bucket
    MethodTable: 73c9127c
    EEClass:     7399f82c
    Size:        20(0x14) bytes
```

```
    File:
C:\WINDOWS\Microsoft.Net\assembly\GAC_32\mscorlib\v4.0_4.0.0.0__b77a5c561934e089\mscorlib.dll
    Fields:
              MT    Field   Offset                 Type VT     Attr    Value Name
        73c92554  4000c7a        0            System.Object      0     instance     1b2aa0d0     key
        73c92554  4000c7b        4            System.Object      0     instance     19309214     val
        73c93b04  4000c7c        8             System.Int32      1     instance     578883777     hash_coll
[10] 1139b558
    Name:        System.Collections.Hashtable+bucket
    MethodTable: 73c9127c
    EEClass:     7399f82c
    Size:        20(0x14) bytes
    File:
C:\WINDOWS\Microsoft.Net\assembly\GAC_32\mscorlib\v4.0_4.0.0.0__b77a5c561934e089\mscorlib.dll
    Fields:
              MT    Field   Offset                 Type VT     Attr    Value Name
        73c92554  4000c7a        0            System.Object      0     instance     13234aa8     key
        73c92554  4000c7b        4            System.Object      0     instance     1322c51c     val
        73c93b04  4000c7c        8             System.Int32      1     instance     139588433     hash_coll
…
```

D. Because the key in the key/value pairs is a string, you may be able to learn something helpful about the strings before you investigate the source code. Now dump some of the strings to see what they contain.

(output below is **shortened** – no MethodTable, EEClass, for better readability):

```
0:000> !do 112975cc
Name:        System.String
String:      IMG-C:\LabFiles\WebSites\FotoVision40\photos\Misc\9514774.jpg14:32:55
0:000> !do 05adef9c
Name:        System.String
String:      IMG-C:\LabFiles\WebSites\FotoVision40\photos\Misc\9514774.jpg14:31:23

0:000> !do 1128dae0
Name:        System.String
String:      IMG-C:\LabFiles\WebSites\FotoVision40\photos\Misc\9514774.jpg14:31:26

0:000> !do 1b2aa0d0
Name:        System.String
String IMG-
C:\LabFiles\WebSites\FotoVision40\photos\Misc\bubble_croman_big.jpg14:31:23

0:000> !do 01a97450
Name:        System.String
String:      IMG-C:\LabFiles\WebSites
\FotoVision40\photos\Misc\9514774.jpg14:25:18

0:000> !do 13234aa8
Name:        System.String
String:      IMG-C:\LabFiles\WebSites
\FotoVision40\photos\Misc\9514774.jpg14:45:00

0:000> !do 192db100
Name:        System.String
String:      IMG-C:\LabFiles\WebSites
\FotoVision40\photos\Misc\Mountains_spitzer_f.jpg14:45:00
```

```
0:000> !do 05ad97cc
Name:        System.String
String:      IMG-C:\LabFiles\WebSites
\FotoVision40\photos\Misc\bubble_croman_big.jpg14:25:18
```

There's some repetition here. This might be a contributing factor of the memory growth problem.

12. Find the problem in the source code.

    A. Based on your debug output, look into the source code of the Web application of FotoVision to find where exactly this memory growth occurs. Once you've found it, document the source file and line number:

    Source filename: _____

    Line number: _____

Optional Exercise 3:
Use of advanced commands for data analysis with Windbg

1. The following topic describes the **advanced** usage of Windows Debugger Commands on **.foreach.** with the example above.

To avoid dumping the strings manually like in the former chapter (with **!do** on the key element) we can use the .foreach command for doing that. This command can be used to execute **any** command (called OutCommands) on the output of the first command (called InCommands), below the syntax of the command:

.foreach [Options] (Variable { InCommands }) { OutCommands }

Options

/pS InitialSkipNumber

Causes some initial tokens to be skipped. InitialSkipNumber specifies the number of output tokens that will not be passed to the specified OutCommands.

/ps SkipNumber

Causes tokens to be skipped repeatedly each time a command is processed. After each time a token is passed to the specified OutCommands, a number of tokens equal to the value of SkipNumber will be ignored.

Variable

Specifies a variable name storing the output of inCommands.

Use the ${ } (Alias Interpreter) token to make sure aliases are replaced even if they are next to other text.

InCommands

Specifies one or more commands whose output will be parsed; the resulting tokens will be passed to OutCommands. The output from InCommands is **not** displayed.

OutCommands.

Specifies one or more commands which will be executed for each token. Whenever the Variable string occurs it will be replaced by the current token.

2. Below the output of !da **without** -details. What we need is the address of each element within the array.
The beginning of the output (the first **22 elements**) must be **skipped** using **/pS 0n22** because we need only the **address** of the elements.
The index of the element must be skipped as well. We need the **second** element **only** - we do that using **/ps 0n1 skipping the first entry.**

```
0:000> !da  1139b4d8
Name:        System.Collections.Hashtable+bucket[]
MethodTable: 73c91238
EEClass:     7399f7a0
Size:        1080(0x438) bytes
Array:       Rank 1, Number of elements 89, Type VALUETYPE
Element Methodtable: 73c9127c
[0] 1139b4e0
[1] 1139b4ec
[2] 1139b4f8
[3] 1139b504
[4] 1139b510
[5] 1139b51c
```

3. Lets try if it would work using **.echo.** This command prints the output of command in order to see what we would get as an input for the OutCommand:

/pS 0n22 skips the first 0n22 elements
/ps 01 skips the index [0],[1],..
myVar stores the address we need: 1139b4e0, 1139b4ec, 1139b4f8
.echo ${myVar} prints the output

```
0:000> .foreach /pS 0n22 /ps 0n1 (myVar { !da 1139b4d8 } ) { .echo ${myVar}}
1139b4e0
1139b4ec
1139b4f8
1139b504
1139b510
1139b51c
```

1. The output looks ok – let's check how to dump the "key" element

   Below the structure of the first array element – Offset tells us where the key element is stored.

```
0:000> !da -details 1139b4d8
Name:        System.Collections.Hashtable+bucket[]
MethodTable: 73c91238
EEClass:     7399f7a0
Size:        1080(0x438) bytes
Array:       Rank 1, Number of elements 89, Type VALUETYPE
Element Methodtable: 73c9127c
[0] 1139b4e0
    Name:        System.Collections.Hashtable+bucket
    MethodTable: 73c9127c
    EEClass:     7399f82c
    Size:        20(0x14) bytes
    File:
C:\WINDOWS\Microsoft.Net\assembly\GAC_32\mscorlib\v4.0_4.0.0.0__b77a5c561934e089\m
scorlib.dll
```

```
    Fields:
MT        Field    Offset Type             VT     Attr      Value     Name
73c92554  4000c7a  0      System.Object    0     instance   00000000  key
73c92554  4000c7b  4      System.Object    0     instance   00000000  val
73c93b04  4000c7c  8      System.Int32     1     instance          0  hash_coll
```

It´s on **OffSet 0 within the array element.**
That means that the address stored on location 1139b4e0 + 0 points to the string. To summarize:

key = pointer to key of type string -> can be found on address 1139b4e0
to shorten the output we use !do –nofields on the pointer poi(..) to key.
Let's dump fhe first element of the array:

```
0:000> !do -nofields poi(1139b8d0)
Name:        System.String
MethodTable: 73c921b4
EEClass:     73883eb4
Size:        182(0xb6) bytes
File:
C:\WINDOWS\Microsoft.Net\assembly\GAC_32\mscorlib\v4.0_4.0.0.0__b77a5c561934e089\m
scorlib.dll
String:      IMG-
C:\LabFiles\WebSites\FotoVision40\photos\Misc\chasma_marsExpress_f50.jpg14:31:23
```

2. The complete command looks like that:
   .foreach /pS 0n22 /ps 0n1 (myVar { !da 1139b4d8 }) { !do -nofields poi(${myVar})}

   Dumps the array on address **1139b4d8**
   skips first 22 tags
   skips each 1st parameter
   Execute the following commands on the output**:!do -nofields poi(….).**
   Let's try it by yourself:

```
0:000> .foreach /pS 0n22 /ps 0n1 (myVar { !da 1139b4d8 } )  { !do -nofields
poi(${myVar})}
Invalid parameter -nofields poi(1139b4e0) <- we found a null pointer
Name:        System.String
MethodTable: 73c921b4
EEClass:     73883eb4
Size:        152(0x98) bytes
File:
C:\WINDOWS\Microsoft.Net\assembly\GAC_32\mscorlib\v4.0_4.0.0.0__b77a5c561934e089\m
scorlib.dll
String:      IMG-C:\LabFiles\WebSites\FotoVision40\photos\Misc\9514774.jpg14:32:55
Name:        System.String
MethodTable: 73c921b4
EEClass:     73883eb4
```

```
Size:        152(0x98) bytes
File:
C:\WINDOWS\Microsoft.Net\assembly\GAC_32\mscorlib\v4.0_4.0.0.0__b77a5c561934e089\m
scorlib.dll
String:      IMG-C:\LabFiles\WebSites\FotoVision40\photos\Misc\9514774.jpg14:31:23
Name:        System.String
MethodTable: 73c921b4
EEClass:     73883eb4
Size:        152(0x98) bytes
File:
C:\WINDOWS\Microsoft.Net\assembly\GAC_32\mscorlib\v4.0_4.0.0.0__b77a5c561934e089\m
scorlib.dll
String:      IMG-C:\LabFiles\WebSites\FotoVision40\photos\Misc\9514774.jpg14:31:26
Name:        System.String
MethodTable: 73c921b4
EEClass:     73883eb4
Size:        172(0xac) bytes
File:
C:\WINDOWS\Microsoft.Net\assembly\GAC_32\mscorlib\v4.0_4.0.0.0__b77a5c561934e089\m
scorlib.dll
String:      IMG-
C:\LabFiles\WebSites\FotoVision40\photos\Misc\bubble_croman_big.jpg14:31:23
Name:        System.String
MethodTable: 73c921b4
EEClass:     73883eb4
Size:        146(0x92) bytes
..
```

Commands like above can be very helpful dealing with larger amount of data.

Exercise 4:

Use PerfView to narrow down the memory issue from Exercise 2

From http://blogs.msdn.com/b/vancem/archive/2011/12/28/publication-of-the-perfview-performance-analysis-tool.aspx:

Notable features of PerfView:

Good help. We went to some trouble to put a lot of 'just in time' information into the tool. You can see even from the screenshot above there are a bunch of blue hyperlinks scattered in the UI. Each of these take you the relevant part of the user's guide, and the user's guide is even more packed with hyperlinks to related information (there is also full text search). There are even videos of PerfView in use, however we are still in the process of getting these published externally, so for now you can use the videos (I will post soon when that works). Please take a look at the links at the top of the views, as they give you important 'perf theory' that you REALLY do need to understand to insure that don't misinterpret the data you collect.

Super easy deployment. Simply copy the single EXE and run. It will work on any Vista+ Windows OS (Win2k8, Win7). This makes it easy to put PerfView on a USB drive or a network share, and take it wherever you need to to collect the data. It really can't get easier.

Support for both CPU and **Memory Investigations**. PerfView is based on the same technology as the XPERF/WPR tool called ETW. In fact data generated by one two can be viewed in the other, so if you like XPERF you are more than welcome to use whichever tool you prefer. However for managed code expecially PerfView has some advantages. In particular PerfView allows you to take snapshots of the GC heap (even 50GB heaps), without interrupting the process (great for servers), to understand the memory performance of your application.

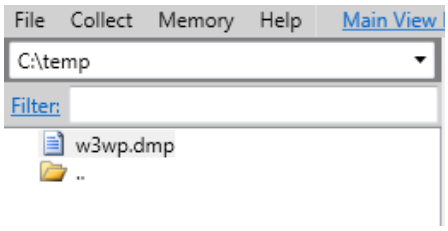
Excellent grouping facilities. After you have investigated your 'trivial' performance problems (were 50%+ of your time is spent in a small amount of code), your profiles quickly become 'flat' and it is harder to find additional optimization opportunities. One reason it is hard is because your performance cost is 'scattered' among hundreds (or thousands), of 'helper' functions operating system routines, so that it is hard to find 'big' chunks of lost perf. The solution to this is to group functions together into logical groups (that perform some recognizable semantic action), so that you can highlight how much each semantic action is costing you. PerfView has the best facilities for doing this that I have ever seen in any performance tool, and it is the main reason why I use this tool over any other (once you have grouping, you can never go back to a tool without it).

Drilling into feature. Performance investigations typically have two phases. In the first phase you are interested in grouping costs until you have a small number (< 10) of semantically relevant nodes that account for a majority of your cost. Once you have done this, you want to investigate each of these (now known to be very relevant), nodes in turn. PerfView has a feature called 'Drill Into' that lets you do just this. It allows you to identify a set of samples (a cost), and open a new viewer for JUST THOSE SAMPLES), which can then be grouped and filtered in new ways (typically removing some of the grouping to reveal one deeper level of abstraction), so you can understand that cost (and just that cost), in more detail. Super useful.

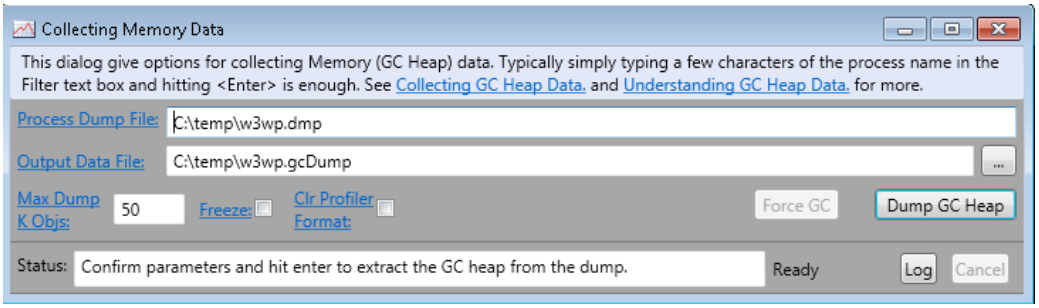
Diffing. Tracking down regressions is a common operation for performance professionals, and so the ability to do a diff of two traces is an important feature. However, PerfView’s grouping features are even more valuable here, because when you compare two runs there are typically 'expected' differences (after all the builds did change), and you need a way of 'ignoring' differences that you understand, and don't care about. PerfView's grouping and drilling into features give you powerful ways of ignoring differences you don't care about and highlighting those you do care about. Again, no other tool I have ever used comes close to doing the job that PerfView does.

Open the Dump with PerfView

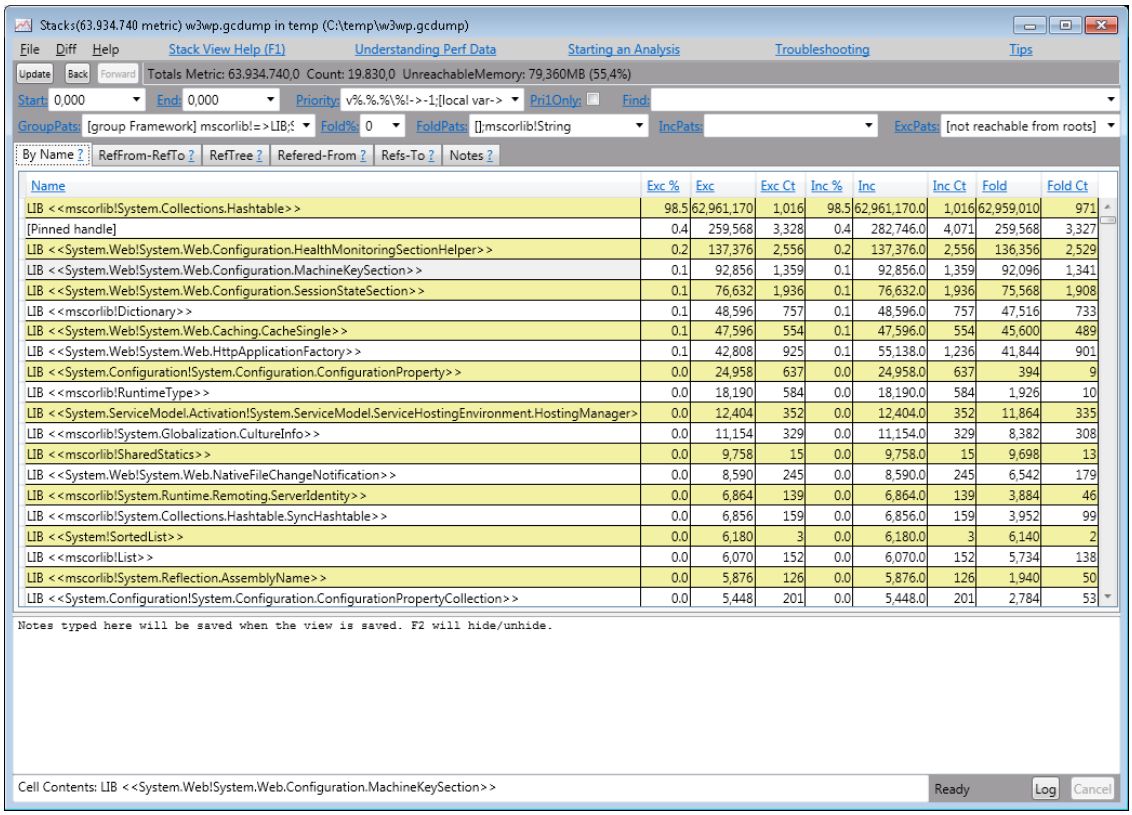
1. Start PerfView – PerfView is located in c:\LabFiles\Tools\PerView
2. Navigate to the Dump created in Exercise 1- in the PerfView file explorer pane:



3. Double Click on the Dump File – the Collecting Memory Data Dialog should appear:



4. Use the Dump GC Heap to Dump the Heap – a new Dialog should open showing an analysis of the memory:



Name	Exc %	Exc	Exc Ct	Inc %	Inc	Inc Ct	Fold	Fold Ct
LIB <<mscorlib!System.Collections.Hashtable>>	98.5	62,961,170	1,016	98.5	62,961,170.0	1,016	62,959,010	971
[Pinned handle]	0.4	259,568	3,328	0.4	282,746.0	4,071	259,568	3,327
LIB <<System.Web!System.Web.Configuration.HealthMonitoringSectionHelper>>	0.2	137,376	2,556	0.2	137,376.0	2,556	136,356	2,529
LIB <<System.Web!System.Web.Configuration.MachineKeySection>>	0.1	92,856	1,359	0.1	92,856.0	1,359	92,096	1,341
LIB <<System.Web!System.Web.Configuration.SessionStateSection>>	0.1	76,632	1,936	0.1	76,632.0	1,936	75,568	1,908
LIB <<mscorlib!Dictionary>>	0.1	48,596	757	0.1	48,596.0	757	47,516	733
LIB <<System.Web!System.Web.Caching.CacheSingle>>	0.1	47,596	554	0.1	47,596.0	554	45,600	489
LIB <<System.Web!System.Web.HttpApplicationFactory>>	0.1	42,808	925	0.1	55,138.0	1,236	41,844	901
LIB <<System.Configuration!System.Configuration.ConfigurationProperty>>	0.0	24,958	637	0.0	24,958.0	637	394	9
LIB <<mscorlib!RuntimeType>>	0.0	18,190	584	0.0	18,190.0	584	1,926	10
LIB <<System.ServiceModel.Activation!System.ServiceModel.ServiceHostingEnvironment.HostingManager>	0.0	12,404	352	0.0	12,404.0	352	11,864	335
LIB <<mscorlib!System.Globalization.CultureInfo>>	0.0	11,154	329	0.0	11,154.0	329	8,382	308
LIB <<mscorlib!SharedStatics>>	0.0	9,758	15	0.0	9,758.0	15	9,698	13
LIB <<System.Web!System.Web.NativeFileChangeNotification>>	0.0	8,590	245	0.0	8,590.0	245	6,542	179
LIB <<mscorlib!System.Runtime.Remoting.ServerIdentity>>	0.0	6,864	139	0.0	6,864.0	139	3,884	46
LIB <<mscorlib!System.Collections.Hashtable.SyncHashtable>>	0.0	6,856	159	0.0	6,856.0	159	3,952	99
LIB <<System!SortedList>>	0.0	6,180	3	0.0	6,180.0	3	6,140	2
LIB <<mscorlib!List>>	0.0	6,070	152	0.0	6,070.0	152	5,734	138
LIB <<mscorlib!System.Reflection.AssemblyName>>	0.0	5,876	126	0.0	5,876.0	126	1,940	50
LIB <<System.Configuration!System.Configuration.ConfigurationPropertyCollection>>	0.0	5,448	201	0.0	5,448.0	201	2,784	53

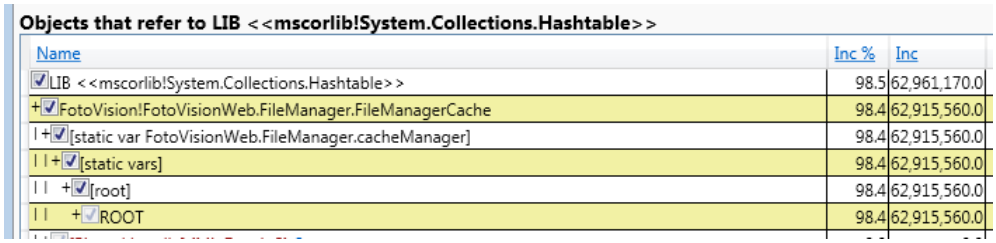
"By Name" shows all interesting types and the amount of memory their using:
Double Click on Type to see all roots for this type of object
"Ref Tree" shows the complete memory reference Tree.

Can you spot how much Memory is used by the File Manager Cache Inclusive (including all referenced objects) in percent?

File Manager Cache Usage % _____

Memory used by the File Manager Cache

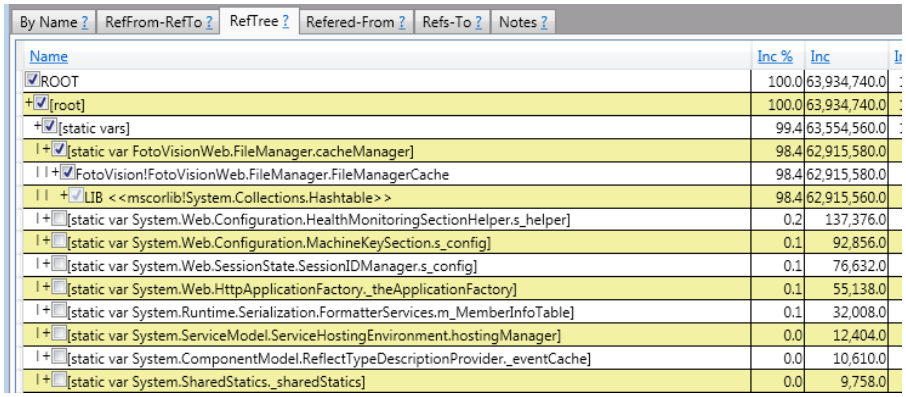
- The File Manager Cache uses 98.4 percent of all Memory (see the “Inc%” Column below)



Objects that refer to LIB <<mscorlib!System.Collections.Hashtable>>

Name	Inc %	Inc
LIB <<mscorlib!System.Collections.Hashtable>>	98.5	62,961,170.0
+ FotoVision!FotoVisionWeb.FileManager.FileManagerCache	98.4	62,915,560.0
+ [static var FotoVisionWeb.FileManager.cacheManager]	98.4	62,915,560.0
+ [static vars]	98.4	62,915,560.0
+ [root]	98.4	62,915,560.0
+ ROOT	98.4	62,915,560.0

- Below the view in the “Ref Tree” showing the same Dump in a different view (roots from the top showing the complete reference tree within the application)



By Name ? | RefFrom-RefTo ? | RefTree ? | Refered-From ? | Refs-To ? | Notes ?

Name	Inc %	Inc
ROOT	100.0	63,934,740.0
+ [root]	100.0	63,934,740.0
+ [static vars]	99.4	63,554,560.0
+ [static var FotoVisionWeb.FileManager.cacheManager]	98.4	62,915,580.0
+ FotoVision!FotoVisionWeb.FileManager.FileManagerCache	98.4	62,915,580.0
+ LIB <<mscorlib!System.Collections.Hashtable>>	98.4	62,915,560.0
+ [static var System.Web.Configuration.HealthMonitoringSectionHelper.s_helper]	0.2	137,376.0
+ [static var System.Web.Configuration.MachineKeySection.s_config]	0.1	92,856.0
+ [static var System.Web.SessionState.SessionIDManager.s_config]	0.1	76,632.0
+ [static var System.Web.HttpApplicationFactory._theApplicationFactory]	0.1	55,138.0
+ [static var System.Runtime.Serialization.FormatterServices.m_MemberInfoTable]	0.1	32,008.0
+ [static var System.ServiceModel.ServiceHostingEnvironment.hostingManager]	0.0	12,404.0
+ [static var System.ComponentModel.ReflectTypeDescriptionProvider._eventCache]	0.0	10,610.0
+ [static var System.SharedStatics._sharedStatics]	0.0	9,758.0