IBM Software Group

IBM Java™ Native Memory

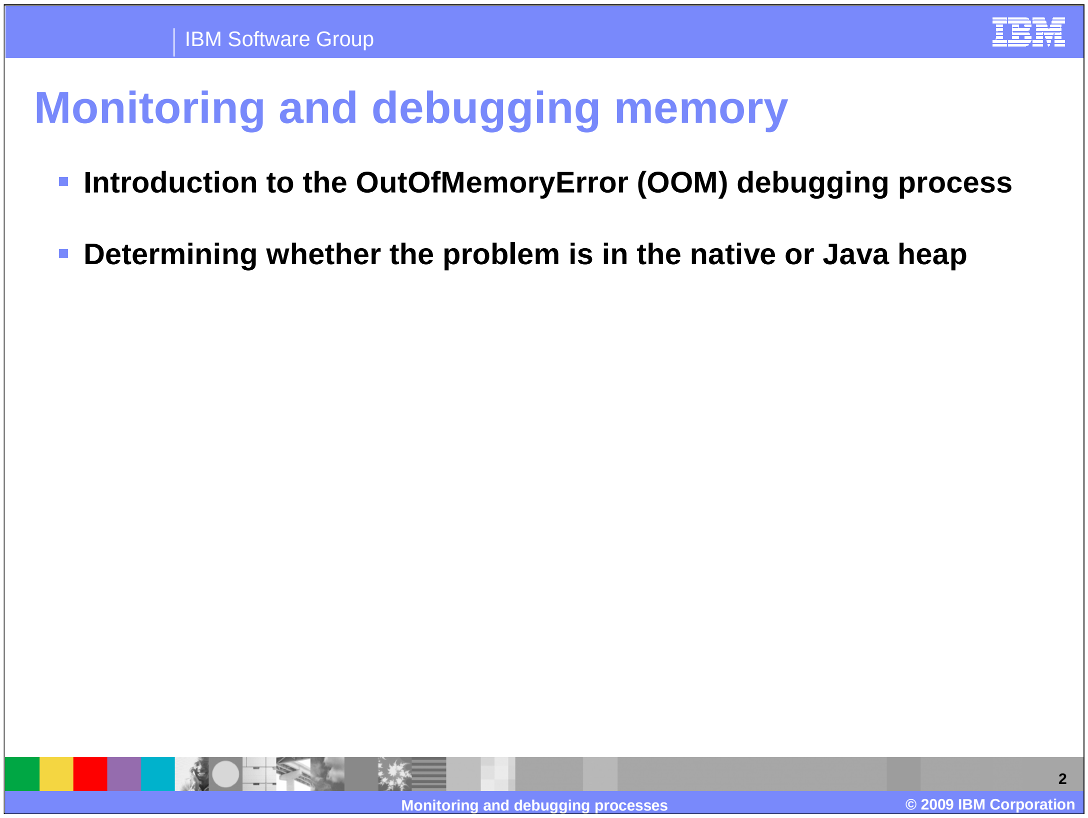
1: Monitoring and debugging processes

business on demand.

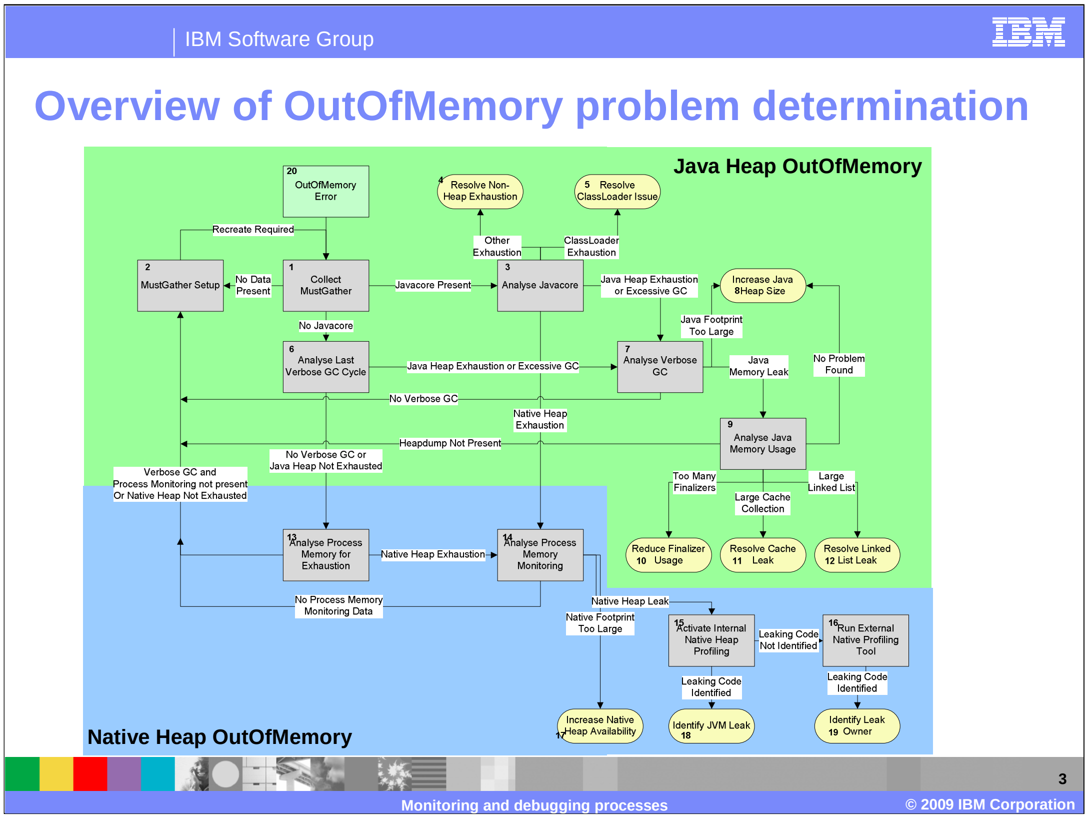
Updated July 20, 2009

This presentation is the first of three covering the native heap and will cover monitoring the process address space to see if the heap is filling and how to use the tools to find out what is using the space and allocating the memory.

You might find it useful to read an overview of process address space for operating systems before listening to this presentation. You can find such an overview in the IBM Java Information Center.



This presentation is a short introduction on how to debug process OutOfMemory (OOM) errors. OutOfMemory errors can look the same in the Java heap and the native heap, so you will see how to determine in which section of memory the OutOfMemory error occurred.



This flowchart is the overview of the OutOfMemory problem process for Java 5 and Java 6.

You can follow the flow from the OutOfMemory error to the yellow boxes, which are resolution points.

Starting in top-left with the OutOfMemory error box, which is number 20, you must collect MustGather. MustGather is an invented word for the collection of files that is useful to IBM to debug your problem.

If you have a javacore file, it can be used to determine if you ran out of memory on the Java Heap or the Native Heap.

In Java 5 SR2 and earlier, you could also have run out of class loaders.

If the problem was in the native heap, you drop down into the blue box at the bottom.

If you do not have the Javacore, use the last VerboseGC cycle or process address space monitoring.

Top half of this diagram is all Java Heap OOM, and the blue at the bottom is Native Heap OOM error.

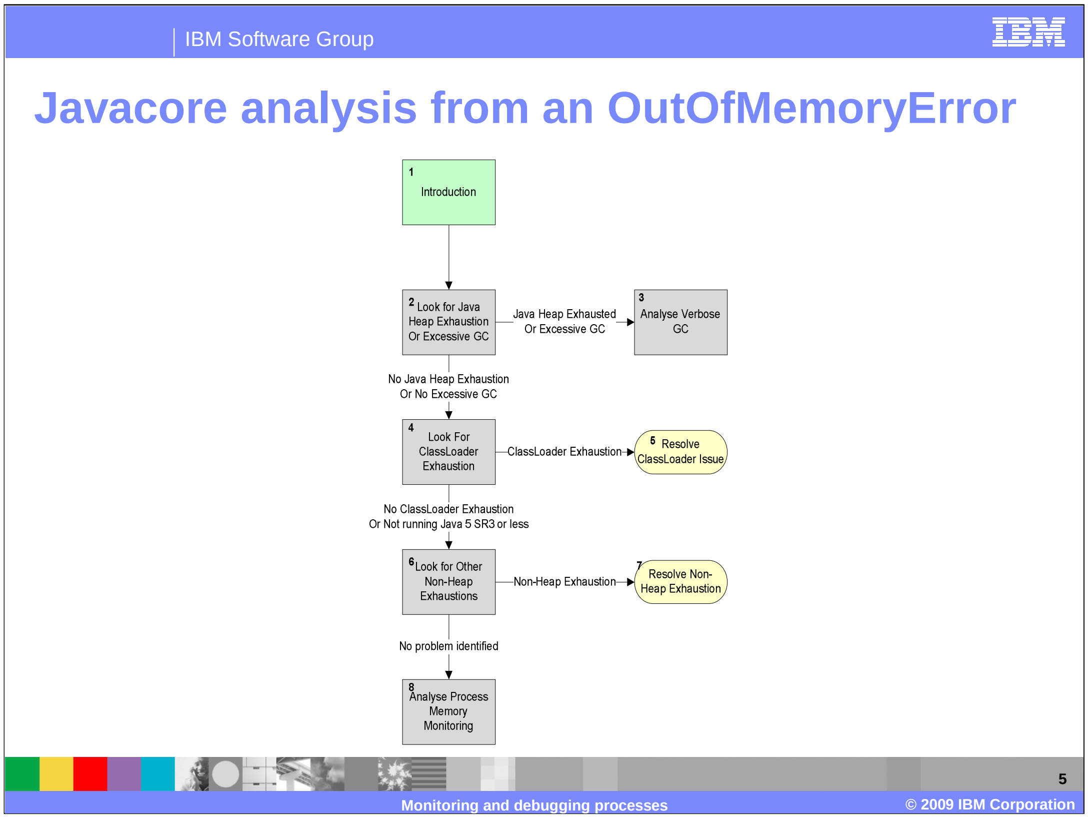
It is much the same diagram for 142, which is not covered here.

Determining the source of the OutOfMemory error

- **Typically, on the policy of elimination:**
  - If the Java heap is not exhausted, the native heap could be the problem.
- **This can be determined from any of three diagnostic files:**
  - Javadump file
  - verbose:gc
  - Process memory monitoring
- **verbose:gc and process memory monitoring can be used to identify problems before they occur**

You can quite accurately tell if the problem is in the Java heap, so it is a process of elimination to determine if it is a Native heap problem.

The three files mentioned here can help IBM Support find the root of the problem. The default file is the javadump file, because the other two must be manually activated. You should always run verbosegc and process memory monitoring, an overview of which is provided later in this presentation.



This slide shows how to analyze the javacore.

First, look for heap exhaustion or excessive Garbage Collection (GC).

Java 5 introduced a function that monitors time spent in garbage collection and if that takes too long it throws an OutOfMemory error rather than reduce performance further.

You must look for both heap exhaustion and excessive garbage collection, but they look the same.

If not heap exhaustion or excessive garbage collection, Java 5 has a limit on class loaders of 8196, which was an artificial limit. This limit was removed in SR3 because there was demand for more, but if you are using Java 5 SR2 and earlier you need to be aware of this limit.

Next, check for recognizable stack trace because any section of code can generate an OutOfMemory error. It is worth searching to find if the error you see is already known using a search engine or looking in Tech Notes. If not, assume the problem is in the native memory.

Javacore analysis

1- Check "Current Thread Details":
- Native Method implies allocation to native heap
- Java Method implies allocation to Java heap

2- Check "MEMINFO" subcomponent:
- Heap Space Free and Allocated shows free bytes in hex
- Other sections show VM native memory usages

3- Check "GC History":
- Garbage Collection (GC) Flight Recorder shows state of garbage collection

To see what an OutOfMemory error looks like in Javacore, look at current thread details. This is a good indicator of resources being exhausted

Using Meminfo, you can learn whether the Java heap is exhausted, or whether certain memory pools are large.

The final way is to look in the GC history. Some of this history is tracked, although not all of it is on by default, which is sent to a javadump. Only 100 records at most will be listed in the javadump, which is approximately 4 or 5 garbage collection cycles.

Javacore analysis: current thread details

- **Java heap:**

```
Current Thread Details
        "main" (TID:0x000C4600, sys_thread_t:0x0003696C, state:R, native ID:0x00000E48)
           at java/lang/StringBuffer.ensureCapacityImpl(StringBuffer.java:397)
            at java/lang/StringBuffer.append(StringBuffer.java:246)
            at memmonster.run(memmonster.java:9)
            at memmonster.main(memmonster.java:18)
```

- **Native heap**

```
Current Thread Details
        "Worker-4" (TID:0x74D48600, sys_thread_t:0x002A989C, state:R, native ID:0x00000164)
            at java/lang/ClassLoader.defineClassImpl(Native Method)
            at java/lang/ClassLoader.defineClass(ClassLoader.java:223(Compiled Code))
            at internal/baseadapt/DefaultClassLoader.defineClass(DefaultClassLoader.java:161(Compiled
           at baseadaptor/loader/ClasspathManager.defineClass(ClasspathManager.java:501(Compiled Code)
   ..
```

- **Class loaders:**

```
Current Thread Details
        "RMI RenewClean-[9.26.49.199:46506]" (TID:0x0000000167D6CA00, sys_thread_t:0x0000000164697208,
           at com/ibm/oti/vm/VM.initializeClassLoader(Native Method)
            at java/lang/ClassLoader.<init>(ClassLoader.java:114)
            at sun/reflect/ClassDefiner$1.run(ClassDefiner.java:60)
           at java/security/AccessController.doPrivileged(AccessController.java:191)
           at sun/reflect/ClassDefiner.defineClass(ClassDefiner.java:57)
        ..
```


Here are some examples of current thread details from the javacore.

There are three different problem types:

The first is Java heap exhaustion. From the red highlighted text, you can see that the method running is ensureCapacity. Because it is a Java method, then you know that this allocation is using the Java Heap.

The second is running defineClass and loading a new class. Because that is a native method, it is allocating on the native heap.

It is a fairly safe assumption that if it is a Java method, it is using the Java heap, and if it is a native method, it is using the native heap.

The third is the special case for class loaders in the native heap when it has hit the artificial limit, so that is why you need to search for that stack trace. If nothing turns up, it will be Java or native heap,

Javacore analysis: MEMINFO section

```
0SECTION       MEMINFO subcomponent dump routine
NULL           =================================
1STHEAPFREE    Bytes of Heap Space Free: 16f0c3d0
1STHEAPALLOC   Bytes of Heap Space Allocated: 25957c00
NULL
1STSEGTYPE     Internal Memory
NULL           segment          start            alloc            end               type     bytes
1STSEGMENT     000000015FE8EED8 000000017DA71C50 000000017DA81C3C 000000017DA81C50  01000040 10000
..             ..
1STSEGTYPE     Object Memory
NULL           segment          start            alloc            end               type     bytes
1STSEGMENT     0000000111DF47C8 0700000000000000 0700000025957C00 0700000025957C00  00000009 25957c00
NULL
1STSEGTYPE     Class Memory
NULL           segment          start            alloc            end               type     bytes
1STSEGMENT     000000013ACF02A8 00000001609D4090 00000001609D4478 00000001609D5750  00020040 16c0
..             ..
1STSEGTYPE     JIT Code Cache
NULL           segment          start            alloc            end               type     bytes
1STSEGMENT     0000000168CFD020 000000018AFC8850 000000018AFC8850 000000018B7C8850  00000068 800000
..             ..
1STSEGTYPE     JIT Data Cache
NULL           segment          start            alloc            end               type     bytes
1STSEGMENT     0000000111E77BD8 0000000179300F50 00000001795817F4 0000000179B00F50  00000068 800000
```


The second way is to look at the MEMINFO section in the Javacore.

The top of the example shows the heap space; how much was freed and allocated.

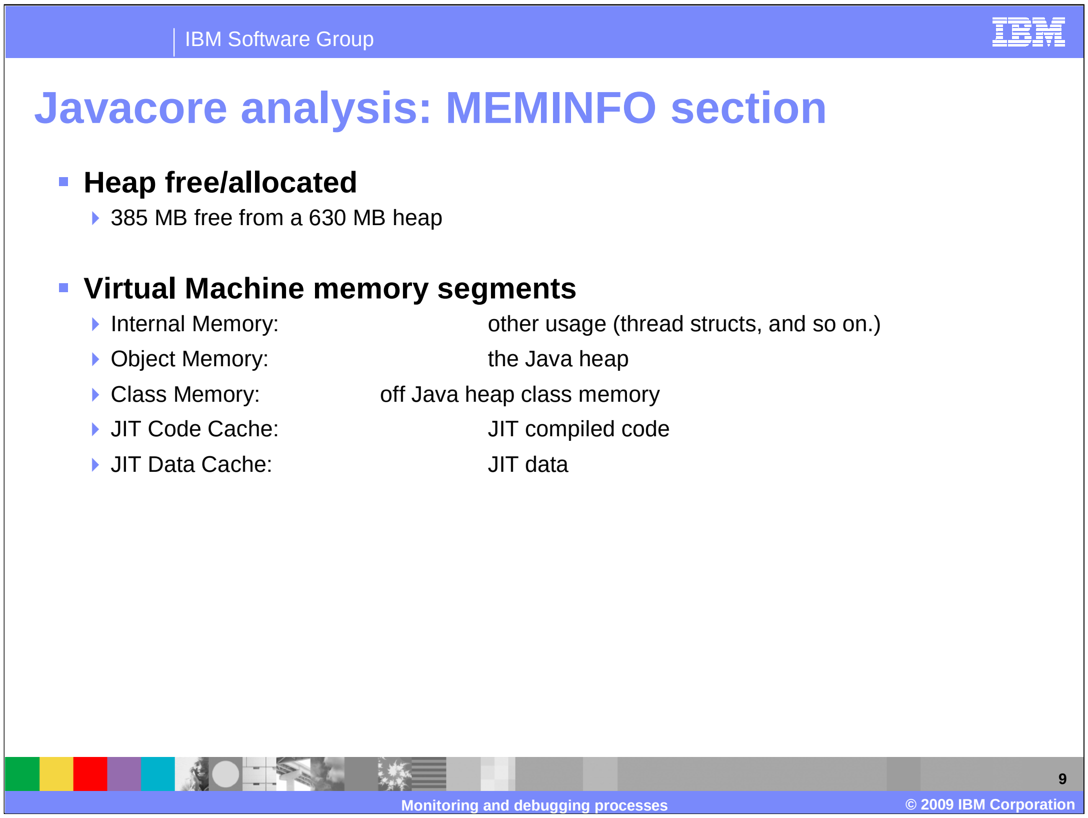
The numbers are displayed in hex and so you need to translate to decimal and perform a shift so it is in megabytes rather than bytes. This tells you how much memory was free at that point in time. If this value is small, you have probably run out of Java heap.

There are also pools of memory that the VM itself uses.

Memory used for JIT Code Cache is byte code that is converted to assembler and stored for running at higher speed .

JIT Data Cache is also memory stored for that purpose.

Object Memory is the Java heap again and is listed twice, the other place being in the summary.



This slide gives you the details of the discussion on the last slide.

If 385 MB is free, you can assume that you have not run out of Java heap.

Next is a list of the sections on the previous slide and what they mean.

Javacore analysis: garbage collection history

```
GC History
   17:32:08:745434927 GMT j9mm.81 -   J9AllocateIndexableObject() returning NULL! 100016 bytes requested for
   object of class 10E94108
   17:32:08:745387931 GMT j9mm.53 -   GlobalGC end: workstackoverflow=0 overflowcount=0 weakrefs=88 soft=0
   phantom=0 finalizers=22 newspace=0/0 oldspace=26576/268435456 loa=0/0
   17:32:08:745368743 GMT j9mm.61 -   Class unloading end
   17:32:08:745352892 GMT j9mm.60 -   Class unloading start
   17:32:08:745342046 GMT j9mm.59 -   Compact end
   17:32:08:499160734 GMT j9mm.58 -   Compact start
   17:32:08:499155728 GMT j9mm.57 -   Sweep end
   17:32:08:495968298 GMT j9mm.56 -   Sweep start
   17:32:08:495963014 GMT j9mm.55 -   Mark end
   17:32:08:484417686 GMT j9mm.54 -   Mark start
   17:32:08:484404060 GMT j9mm.52 -   GlobalGC start: weakrefs=88 soft=5 phantom=0 finalizers=22
   globalcount=15 scavengecount=0
   17:32:08:484373470 GMT j9mm.53 -   GlobalGC end: workstackoverflow=0 overflowcount=0 weakrefs=88 soft=5
   phantom=0 finalizers=22 newspace=0/0 oldspace=15144/268435456 loa=0/0
   17:32:08:484350945 GMT j9mm.59 -   Compact end
   17:32:08:475764182 GMT j9mm.58 -   Compact start
   17:32:08:475759176 GMT j9mm.57 -   Sweep end
   17:32:08:471208843 GMT j9mm.56 -   Sweep start
   17:32:08:471203559 GMT j9mm.55 -   Mark end
   17:32:08:456559232 GMT j9mm.54 -   Mark start
   17:32:08:456543937 GMT j9mm.52 -   GlobalGC start: weakrefs=88 soft=5 phantom=0 finalizers=22
   globalcount=14 scavengecount=0
```


The third way to be considered is the garbage collection history, which contains a lot of the same data as in verbosegc, but in a different format.

On the left are microsecond-accurate time stamps, so you know exactly when the events occurred. You can calculate the duration of the event using the start and end time stamps.

The important section to look at to determine if you ran out of Java heap is the text highlighted in red at the top.

In this case, you see J9allocateindexableobject returned null. This means that the J9allocateindexableobject function was used to allocate an object on the Java heap and, because it failed, null was returned. The object it was trying to allocate was 100,000 bytes, so 100 KB, and it was requested for an object of class 10E94108.

To find out which class this is referring to, you must look in the javadump because all loaded classes are at the end of the javadump file along with the class name. The class number displayed is the address where it was loaded to in memory.

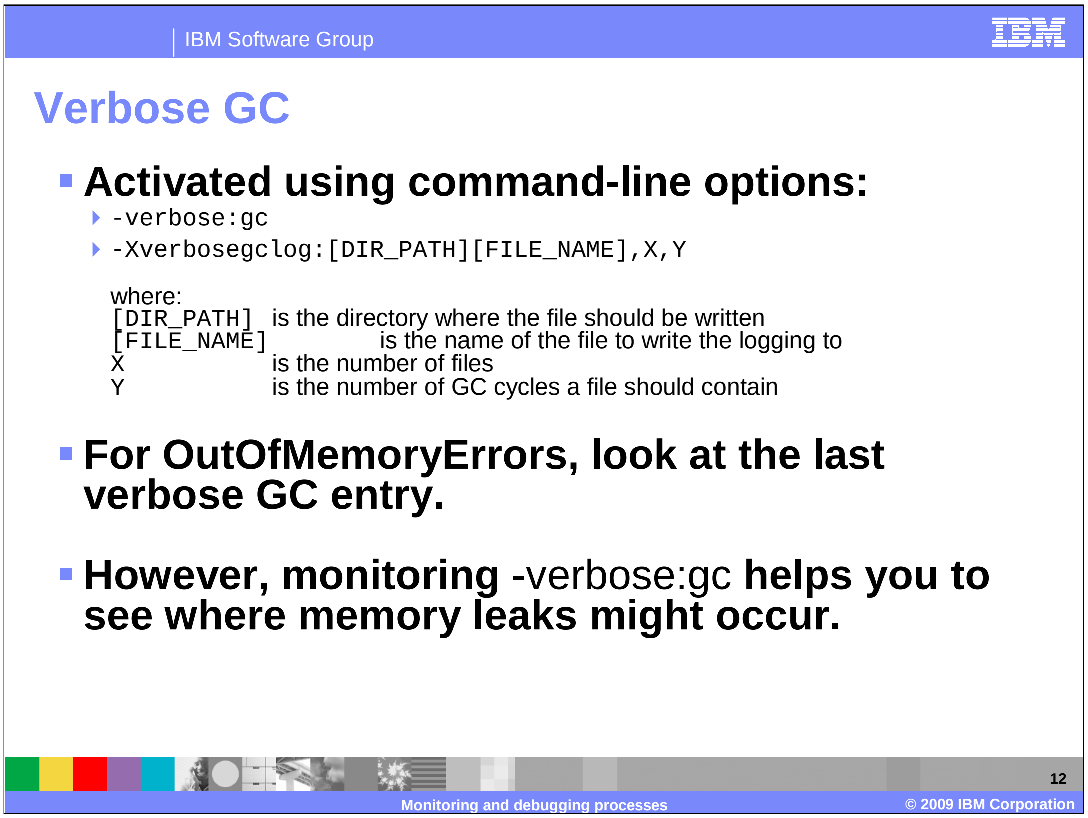
Javacore analysis: garbage collection history

```
GC History
18:41:16:388207799 GMT j9mm.101 - J9AllocateIndexableObject() returning NULL! 1048592 bytes requested for
   object of class 00164790 from memory space '' id=00000000
18:41:16:388047632 GMT j9mm.84 - Forcing J9AllocateIndexableObject() to fail due to excessive GC
18:41:16:387446239 GMT j9mm.82 - Excessive GC raised!
18:41:16:387414206 GMT j9mm.53 - GlobalGC end: workstackoverflow=0 overflowcount=0 weakrefs=2 soft=0
   phantom=0 finalizers=20 newspace=0/0 oldspace=1737904/268435456 loa=0/0
18:41:16:387120612 GMT j9mm.59 - Compact end
18:41:16:360574373 GMT j9mm.58 - Compact start
18:41:16:360562674 GMT j9mm.57 - Sweep end
18:41:16:356781894 GMT j9mm.56 - Sweep start
18:41:16:356677437 GMT j9mm.55 - Mark end
18:41:16:352318662 GMT j9mm.54 - Mark start
18:41:16:352301392 GMT j9mm.52 - GlobalGC start: weakrefs=2 soft=0 phantom=0 finalizers=20

   globalcount=99 scavengecount=0
```


The output here is the same as on the previous slide, but the second highlighted red line shows the text “forcing J9allocateindexableobject to fail”, so the JVM decided that too much time had been spent in the garbage collection cycle and that it was in a state of “excessive GC”, which forced the next allocation to fail.

These are the three ways that you can use to decide whether the failure was Java heap or Native heap, and you should confirm this by looking at other areas to check your decision.



Sometimes a javadump is not generated, and, in this case, verbosegc can tell you if you ran out of Java heap and process memory monitoring can tell you if ran out of native heap. It is sensible to make sure you check your decision with more data to ensure that you are looking at right area.

Verbosegc can be activated with –verbose:gc on the command line. The output is sent to the native standard error log by default, but you can specify a file to send it to using the –Xverbosegclog option.

If you turn on –Xverbosegclog without turning on –verbose:gc, -Xverbosegclog forces –verbosegc to be enabled.

The parameters X and Y are the number of files and number of cycles, so you can have a circular buffer of files to ensure that you keep the data about the most recent cycles. If you are logging to file, some messages will still go to the standard error file, mainly data about verbose:gc itself.

Be aware of these performance overheads:

A test was done at IBM to check overheads of verbose:GC. GC cycles were set up to last 100ms, which is a reasonable time but not a long time. The overhead was found to add about 3% to the duration of a GC cycle. If 100ms happens every 3 - 4 seconds, then 3% of 3% will represent a very small number. If GC cycles are minutes apart, the extra few milliseconds it is costing you to run –verbose:gc is negligible. The advantage of being able to resolve performance problems and OOM problems outweighs the overhead. It is good practice to log to a file so that you can restrict the file size as needed.

- Verbose:gc analysis

- **Can determine native or Java heap exhaustion from the last GC cycle:**

```
<af type="tenured" id="11" timestamp="Sun Jul 31 19:03:16 2005" intervalms="47.826">
  <minimum requested_bytes="5242896" />
  <time exclusiveaccessms="44.526" />
  <nursery freebytes="655360" totalbytes="786432" percent="83" />
  <tenured freebytes="4304552" totalbytes="267386880" percent="1" />
  <gc type="global" id="12" totalid="73" intervalms="47.904">
    <compaction movecount="11571" movebytes="16320456" reason="compact to meet allocation"
   />
    <refs_cleared soft="0" weak="0" phantom="0" />
    <finalization objectsqueued="0" />
    <timesms mark="17.205" sweep="10.251" compact="52.208" total="79.717" />
    <nursery freebytes="785168" totalbytes="786432" percent="99" />
    <tenured freebytes="4435600" totalbytes="267386880" percent="1" />
  </gc>
  <gc type="global" id="13" totalid="74" intervalms="0.030">
    <compaction movecount="12133" movebytes="262762572" reason="compact to meet
   allocation" />
    <refs_cleared soft="0" weak="0" phantom="0" />
    <finalization objectsqueued="0" />
    <timesms mark="21.701" sweep="9.644" compact="590.093" total="621.504" />
    <nursery freebytes="785168" totalbytes="786432" percent="99" />
    <tenured freebytes="4435616" totalbytes="267386880" percent="1" />
  </gc>
  <tenured freebytes="4435616" totalbytes="267386880" percent="1" />
  <time totalms="745.990" />
</af>
```

13

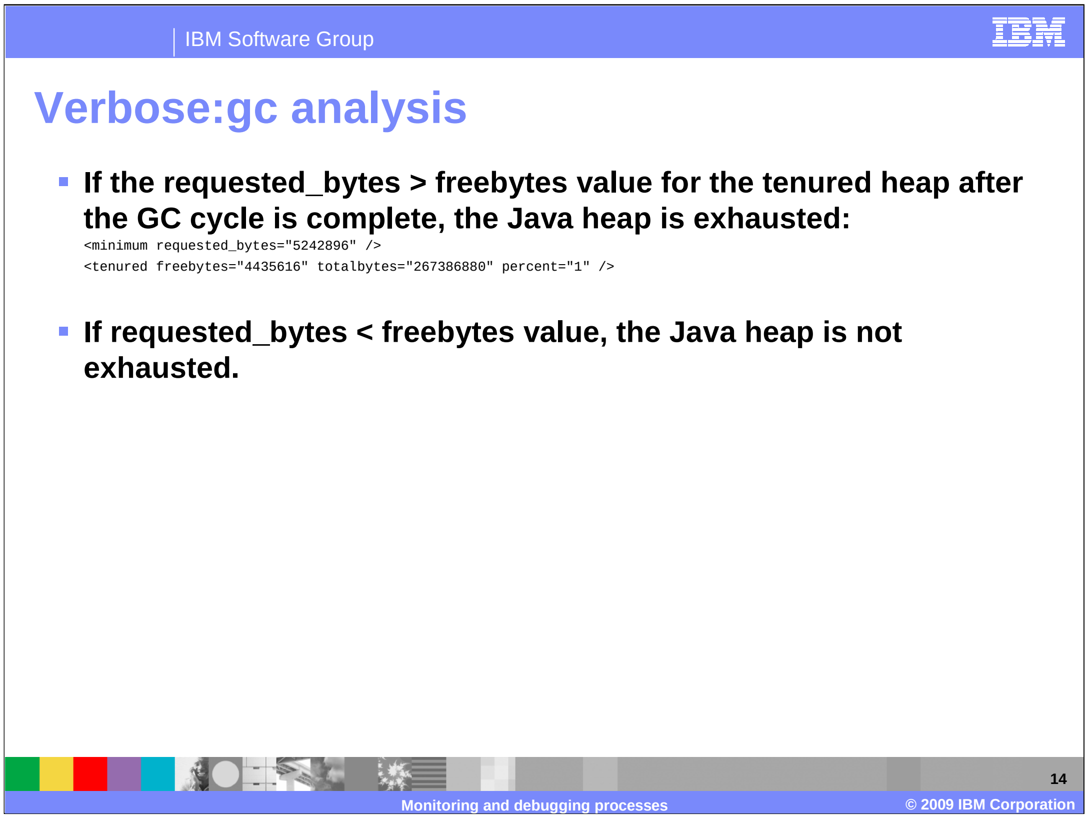
Java 5 and Java 6 use verbose:gc analysis. To see if the heap was exhausted since the last gc cycle, see the text highlighted in red.

The first red section of text, requested_bytes, is the size of object that it is trying to allocate.

The two values in red at the end, beginning with freebytes, are the amount of free space after the gc, and then the amount of free space after allocation.

If the requested size in bytes is bigger than the first freebytes value, not enough resources were freed up to do request.

If the two values at the bottom are different, the object was allocated after the gc cycle. If the values are the same, the allocation failed.



This slide gives you the details of the discussion on the last slide.

That is the end of the introduction about debugging OutOfMemory errors. The next presentation in this series is about monitoring the memory usage to look for OutOfMemory exceptions in the native heap.

Feedback

Your feedback is valuable

You can help improve the quality of IBM Education Assistant content to better meet your needs by providing feedback.

- Did you find this module useful?
- Did it help you solve a problem or answer a question?
- Do you have suggestions for improvements?

Click to send e-mail feedback:

mailto:iea@us.ibm.com?subject=Feedback_about_1-IntroAndDetermining.ppt

This module is also available in PDF format at: ../1-IntroAndDetermining.pdf

You can help improve the quality of IBM Education Assistant content by providing feedback.

Trademarks, copyrights, and disclaimers