IBM Software Group

SW5706
Introduction to WebSphere® out of memory problems

This unit covers the problem determination techniques associated with out of memory problems in WebSphere Application Server.

Unit objectives

After completing this unit, you should be able to:

- Describe what causes OutOfMemory conditions
- Use the Tivoli Performance Viewer to detect OutOfMemoryConditions
- Obtain a VerboseGC log
- Obtain a Java™ heap dump
- Interpret a verboseGC log
- Analyze a java heap dump

After completing this unit, you will be able to:

Define an out-of-memory condition

Use Tivoli Performance Viewer to detect out-of-memory conditions

Obtain and interpret a verboseGC log

Obtain and interpret a heap dump

Discuss tools for analyzing out-of-memory problems

What is a java.lang.OutOfMemory error?

- Java Virtual Machine error
- Not enough memory to allocate an object; can be caused by the following:
  - The Java heap is too small
  - Memory is available in the heap, but it is fragmented (not contiguous)
  - Memory leak in the Java code
  - Memory leak in native code

When the Java virtual machine (JVM) tries to allocate an object and it fails, it runs garbage collection to free up heap space from objects that are no longer being used. If the object cannot be allocated after garbage collection, the JVM throws a java.lang.OutOfMemoryError. There are four conditions that cause the OutOfMemoryError to be thrown. The first is that the JVM heap's maximum size is too small. Second, there may be enough unallocated space in the JVM heap, but it is not contiguous, which is called fragmentation. Third, the Java code could be continuously creating objects and never relinquishes them, thus causing a memory leak. The final condition is a leak in native code. In this case the JVM heap has plenty of contiguous space available, but the Operating System fails to provide memory for the object allocation.

JVM heap is too small

- How do you know the heap is too small?
  - The JVM heap usually displays constant growth until it reaches the Maximum heap size
  - The JVM heap never achieves a steady state
- Need to increase the Java Maximum heap size in the administrative console
  - **Servers -> Application Servers -> server -> Java and Process Management -> Process Definition -> Java Virtual Machine -> Maximum heap size**
  - Set as part of a Performance Tuning Process outlined in the InfoCenter
    - http://publib.boulder.ibm.com/infocenter/wasinfo/v6r0/index.jsp?topic=/com.ibm.websphere.base.doc/info/aes/ae/tprf_tunejvm.html

Applications require a minimum amount of memory to reach a stable state. The stable state occurs when the heap is no longer consistently growing. To reach a stable state, the application has to run through its commonly used code paths and instantiate all of the frequently referenced objects. The load used to test for the stable state is very important. Usually more memory is required to reach a stable state under higher loads. If the JVM is configured with a maximum heap that is too small and never allows the JVM to reach a stable state, then allocation failures will occur and the JVM will throw OutOfMemoryError. Applications should go through rigorous performance tuning before being deployed into a production environment. One goal of performance tuning is to determine the application's stable state and allocate the appropriate amount of memory to the application server.

Memory fragmentation

- The difference between the Maximum heap size and the current heap size is only slightly greater than the size of the object to be allocated.
  - Some fragmentation will always occur.
  - Should be treated as if the heap was too small.
- The object being allocated is excessively large.
  - The JVM is attempting to allocate an object that takes up a significant portion of the heap by itself.
  - The application developer should attempt to reduce the size of the object being allocated, and if that's not possible, increase the JVM heap.

When memory is allocated for an object, the object gets a range of contigous memory from the heap. The next object will get another range of contiguous memory addresses from the heap. At the same time memory is being allocated, it is being deallocated and returned to the heap. Because memory is constantly in the process of being allocated and deallocated, it is impossible for all available memory in the heap to be in one contiguous block. Instead, chunks of allocated memory are fragmented throughout the heap. To be able to allocate an object, memory must be in a contiguous block. It is possible that there is enough available memory in the heap to fullfill an allocation request, but there is not a single block of contiguous memory available so the request fails. There are two specific cases of fragmentation that are commonly seen. The first occurs when the difference between the current heap size and the maximum heap size is only slightly greater than the size of the object to be allocated. Since some fragmentation will always occur within any memory management system, this case should be treated as if the heap size was too small. The second case is when the JVM continuously attempts to allocate excessively large objects. To illustrate the problem of large object allocations, imagine the java heap as a stack of seven blocks. Each block represents one unit of memory. Now imagine that you have an allocation request for three units of memory. If the memory could be allocated from the top or bottom of the heap, there would still be four contiguous memory units available. We would still be able to satisfy an allocation request for four memory units. Now imagine that the only contiguous three blocks available are the center three, so you allocate them. Soon after, all other blocks become deallocated, thus there are four available units of memory. Unfortunately, since the three units in the middle of the heap are allocated, that leaves only two contiguous units on each end. This means that any request to allocate an object that requires more than two units of memory will fail. Fragmentation becomes more of a problem as the allocation requests get larger. In the second case, the best practice is to reduce the size of the object being allocated. If this isn't possible, then the only other alternative is to increase the size of the JVM heap.

Memory leak in the Java code

- No matter what the JVM Maximum heap size is set to, the heap will still run out of space.
- Increasing the Maximum heap size only causes the problem to take longer to occur.
- One or more objects are taking up a high percentage of the JVM heap.

Memory is not explicitly allocated and deallocated in Java, like in C and C++,but it is still possible to create a memory leak. One example would be to save an object into some type of collection. If the collection is a class object, and the class always stays loaded, the object will never be removed from the collection. If objects are continuously added to the collection, the collection could grow until it consumes a significant portion of the java heap. The misuse of object caches is a common cause of memory leaks seen in applications running in WebSphere. One example is the configuration of a large PreparedStatement cache. The PreparedStatement cache size is the total number of PreparedStatements that WebSphere will maintain in the cache. PreparedStatement objects are the parent objects of ResultSet objects, where data from a PreparedStatement is stored. If the ResultSet objects associated with the PreparedStatements in cache are fairly large, and the PreparedStatement cache size is large, then the PreparedStatement cache can grow very quickly and consume signigicant amounts of available memory. Another commonly seen misuse of cache in WebSphere is the HTTP max Sessions parameter in WebSphere. HTTP Sessions can be written to store very large objects. If the HTTP max sessions parameter is set too high, then it will again consume a large portion of the JVM heap.

Not enough native memory

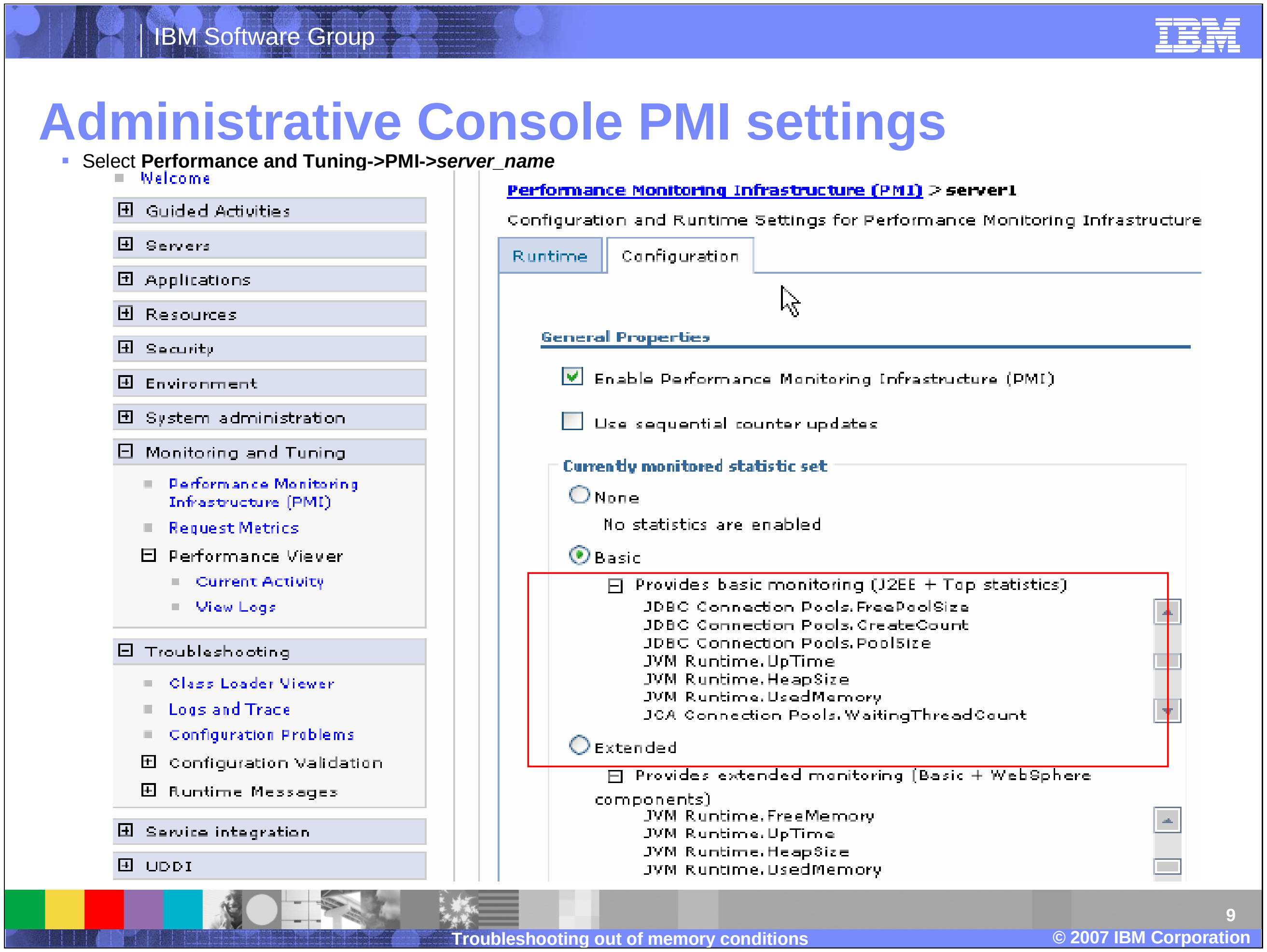
- There is more than sufficient space in the JVM heap, but the allocation still fails
- The JVM is not trying to allocate a large object

When the Java Virtual Machine is unable to find enough contiguous memory in the heap for object allocation, and it has already called the garbage collection routine, it will then attempt to grow the JVM heap. In order to grow the JVM heap the JVM must request, and be granted, system memory from the operating system Memory Management Unit (MMU). If the MMU is unable to provide memory to the JVM for heap expansion, the JVM will throw OutOfMemoryError. Every Java thread running within a JVM also has an associated thread stack, which requires system memory. If there is not enough memory in the system for the allocation of a Java thread, the system will again throw OutOfMemoryError. Finally, through the Java Native Interface (JNI) the application can access system libraries, which require native system memory to be loaded. If the application fails to remove unused references to JNI objects it will cause a native memory leak. If the system is low on native memory to begin with and is unable to provide memory for the object allocation, the JVM will also throw OutOfMemoryError.

Using TPV to anticipate an OutOfMemory error

- Tivoli Performance Viewer (TPV) runs in the WebSphere Administrative Console
- Uses Performance Monitoring Infrastructure (PMI) to capture information about the WebSphere runtime.
- Provides graphical display of the Captured PMI Data.

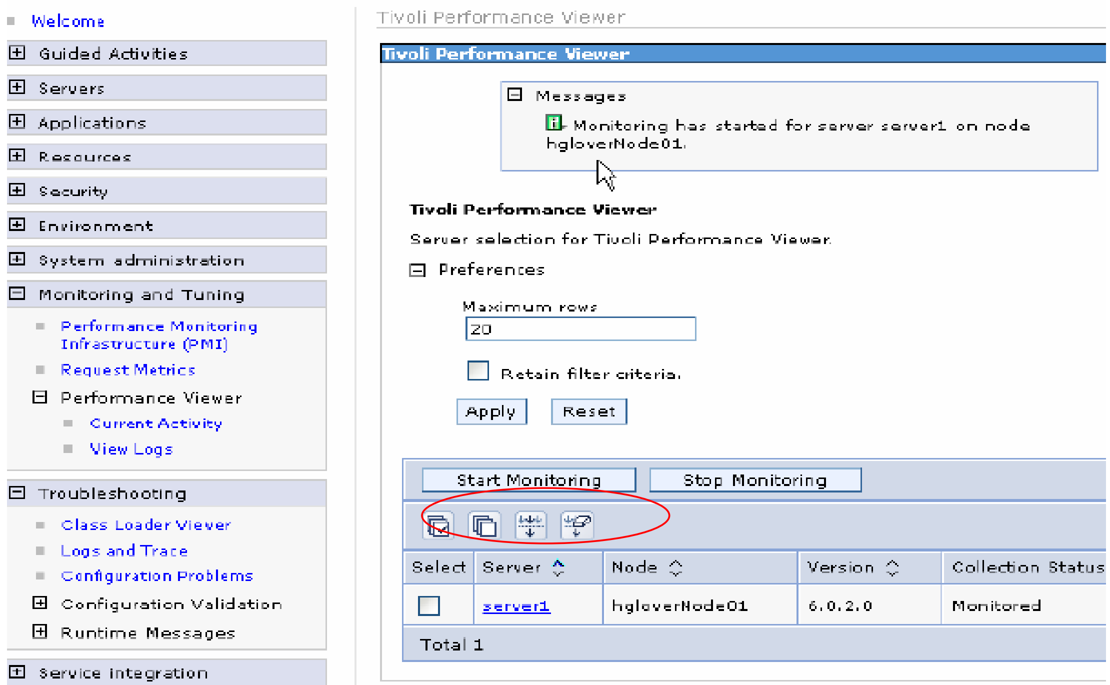
The Tivoli Performance Viewer is embedded within the WebSphere Administrative Client. It uses the PMI infrastructure to capture information about the WebSphere runtime, such as the JVM heap size. We can change the PMI settings in the WebSphere console to capture the information we want to see, set the TPV to begin logging the PMI request information, and view the graphical representation of the data collected. In this case, we want to monitor the size of the JVM heap.



Under Monitoring and tuning, select the `Performance Monitoring Infrastructure‘ (PMI) link. Under the `Configuration‘ tab, select the checkbox to enable PMI. Then select ‘Basic‘ from the `Currently monitored statistic set.‘ From the picture we see this includes `JVM Runtime.HeapSize‘ statistics.

Administrative Console TPV Monitoring

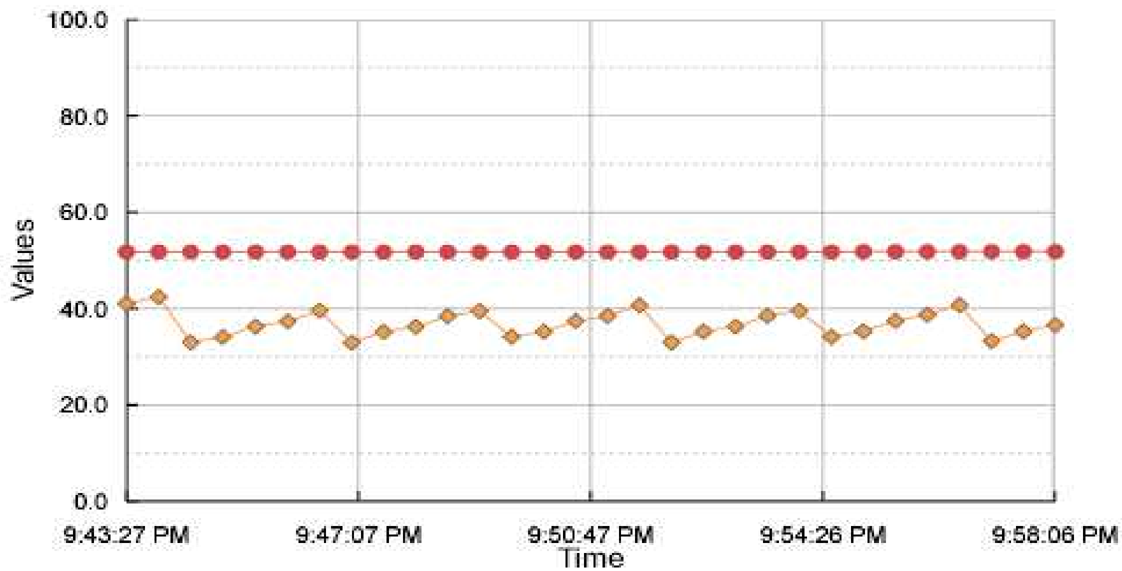
- Select **Monitoring and Tuning->Performance Viewer->Current Activity->*server_name***
- Click **Start Monitoring** button



To start monitoring go to `Performance Viewer' and select the `Current Activity' link. Select the Application Server you wish to monitor, and hit the `Start Monitoring' button. Then select the link to your Application Server, in this case `server1', and hit the `Start Logging' button.

Administrative Console TPV Graph

Red line = Heap Size   Orange line = Free Memory



You may need to leave that page and come back in, but the TPV will provide a Graph that inlcudes the JVM Heap Size of the WebSphere runtime over time. In this case the Heap Size is the red line, and the Free Memory is represented by the orange line. We can monitor these values to determine when an OutOfMemory condition is about to occur. In this case, the heap usage is behaving normally. The heap size grows until grabage collection and then returns to the previous relative minimum. If the heap size does not return to the same size as before, the sawtooth patten is continuously increasing, then there is probably a memory leak in the application server.

How to obtain a verboseGC log

- From the administrative console Select **Servers -> Application Servers -> *server_name* -> Java and Process Management -> Process Definition -> Additional Properties -> Java Virtual Machine**
- Select the **Verbose Garbage Collection** checkbox
  - For HP-UX, add the following parameter to the Generic JVM Arguments on the Java Virtual Machine Settings Page :
  - -Xverbosegc:file=<name>
  - For Solaris, add the following parameter to the Generic JVM Arguments on the Java Virtual Machine Settings Page :
  - -XX:+PrintGCDetails -XX:+PrintGCTimeStamps -XX:+PrintHeapAtGC
- Apply and save changes to the master configuration.
- Restart the Application Server

The verboseGC (verbose garbage collection) output is the diagnostic data used to identify what type of OutOfMemoryError condition is occurring on the system. The verboseGC output shows the size of the object that is being allocated, how much memory is available on the Java Heap, and the JVM's response to the allocation request. In many cases the VerboseGC output will give us a good idea how to resolve the problem, and even if it doesn't it will help us to narrow the scope of the problem. There are some small differences in the VerboseGC output between the Sun,HP-UX, and IBM JVMs. The IBM JVM (Windows®, AIX®, and Linux®) produces highly detailed output by default, and places the information in the native_stderr.log file. The HP-UX JVM also provides detailed ouput, but you have to tell it where to write the output. The Sun JVM on Solaris will write the verboseGC output to the native_stdout.log file, but it does not provide detailed output by default. You have to provide the paramters to tell it to print out the detailed information about the GC event, the timestamp of the GC event, and the size of the JVM heap at the time of the GC event.

Obtaining Java heap dumps

- Used for analyzing the actual contents of the Java heap
  - Shows the objects using the heap memory
  - Needed for memory leaks
- IBM Java heap dump for IBM JVMs
  - Windows
  - AIX
  - Linux
- Java memory dumps on Solaris
  - IBM Heapdump Agent for 1.4.2_08 and earlier
  - JMAP for 1.4.2_09 – 1.4.2_11
  - Sun heapdump functionality for 1.4.2_12 and above
- Java memory dumps on HP-UX
  - HPROF

A memory dump of the JVM heap is the primary diagnostic data for identifying memory leak culprits. With the use of special tools, you can analyze the heap dumps to tell us what objects are taking up significant chunks of heap memory. We can also tell who is responsible for instantiating those objects. For IBM JVM's, which run on Windows, AIX, and Linux platforms, you obtain IBM Java Heap Dumps. The JVM is easily configured within the Administrative console to generate an IBM Heap Dump. The advantage of the IBM Heapdump is that there are tools available to parse and process the heapdump file. On Solaris, the tool depends on the version of the Sun JVM. Depending on the version of the JVM you will use the IBM Heapdump Agent, JMAP, or Sun's own heapdump functionality. For HP-UX, the suggested tool for obtaining a dump of the java heap is the Heapdump agent.

How to obtain a Java heap dump with the IBM JVM

- From the Administrative Console select **Servers -> Application Servers -> *server_name* -> Java and Process Management -> Process Definition -> Environment Entries -> New**
- Add the following name-value pairs :
  - IBM_HEAPDUMP true
  - IBM_HEAP_DUMP true
  - IBM_HEAPDUMPDIR *your_dump_directory*
  - IBM_HEAPDUMP_OUTOFMEMORY true
  - IBM_JAVADUMP_OUTOFMEMORY true
  - IBM_JAVA_HEAPDUMP_TEXT true
- Apply and Save changes to the master configuration.
- Restart the Application Server

For the IBM JVM, command line parameters must be passed to the Java executable on startup in order to obtain an IBM Java Heapdump. The command line parameters can be set within the WebSphere Administrative console. After setting the paramters, the Application Server must be restarted in order for the JVM to be initialized with the heapdump settings.

Once setup, the IBM JVM will automatically dump a Java heapdump when an OutOfMemoryError occurs or can be forced by sending a kill -3 to the application server process. There are several options for configuring the Heap Dump. You can specify the location, format, and if one should automatically be generated when an out of memory occurs.

Java memory dumps on Solaris

- For now, contact IBM Customer Support for instructions
  - Sun JVM version 1.4.2_08 and earlier use IBM Heapagent
    - Not supported for production environments without assistance from IBM Support.
  - Sun JVM version 1.4.2_09, 1.4.2_10, and 1.4.2_11 use JMapp
    - Command line tool that ships with the Sun JVM on Solaris
    - jmap –histo <PID> or jmap –histo <core file>
      - Lists each class in the heap, the number of instances of that class, and total memory used by all instances.
      - Use the output operator to send output to a file.
    - Refer to Sun's J2SE 5.0 Trouble-Shooting and Diagnostic Guide for more information.
  - Sun JVM version 1.4.2_12 and higher provides ability to dump Java Heap
      - JVM command line option –XX:+HeapDumpOnCtrlBreak
      - SIGQUIT signal (kill -3) triggers the JVM to generate a Heapdump
      - Heapdump can be parsed using MDD4J

The Sun 1.4.2 JVM has recently undergone a series of changes to it's serviceability code. Because of that, depending on the the version of the service release of the 1.4.2 JVM, there are three different options for obtaining a dump of the JVM heap. For Sun JVM version 1.4.2_08 and earlier, IBM provides the IBM Heapdump agent which can be used to obtain an IBM Java heap dump. The advantage of the IBM Java heapdump is there are tools available to parse and analyze the file. The Heapdump Agent is not supported for production environments without assistance from IBM, so users should contact IBM Support in order to obtain a heapdump for Sun JVM 1.4.2_08 and earlier. Changes in the JVM code for Sun JVM version 1.4.2_09 will cause the JVM to crash if you attempt to attach the IBM Heapdump Agent. Because of this, the IBM Java Development teams suggests that you use JMapp to obtain a dump of the Java heap instead. JMapp is a command line tool that is packaged along with the Sun JVM on Solaris. JMapp can be run on a running process or a core file to obtain a dump of the Java heap. Beginning with Sun JVM 1.4.2_12, Sun provides it's own heapdump mechanism. To configure the Sun Java heapdump, set the JVM command line option, restart the WebSphere Application Server, and then issue a kill -3 against the WebSphere Application Server JVM process to generate a heapdump. The advantage of the Sun Java heapdump is that you will be able to parse and analyze it using MDD4J.

How to obtain a heap dump on HP-UX

- Contact IBM Support
  - Normally use HPROF
    - Not useful with larger heap sizes which are typically used with WebSphere
    - Costly performance impact
    - Work with IBM Support find the best solution for your environment

Obtaining a dump of the Java heap on HP-UX is more difficult than with the IBM or Sun JVMs. HPROF is the best tool available, but it often hangs with Java heaps greater than 500 MB. Since most WebSphere Application Server JVM heap sizes are greater than 500 MB, HPROF is rarely useful. It also incurs a heavy performance cost, which negatively affects the behavior of the application. One advantage for HPROF is that there is a tool, HAT, available to parse and analyze HPROF dumps. Users that need to obtain a dump of the Java heap for WebSphere running on HP-UX should contact IBM Support.

Tools available for analysis

- VerboseGC
  - PMAT (Pattern Mapping Analysis Tool)
    - IGAA
    - Developer works
  - HPJTune
    - HP-UX
    - http://www.hp.com/products1/unix/java/java2/hpjtune/index.html
  - GC Portal
    - Solaris
- Java heap dumps
  - MDD4J
    - IBM Heap dumps
  - Hat
    - HProf dumps on Solaris and HP-UX
    - https://hat.dev.java.net/

There are several tools available for analyzing diagnostic data for OutOfMemoryError conditions. These tools provide expert analysis, and in the case of MDD4J and HAT, they make analyzing the objects in a Java heap possible.

PMAT is a common tool used to analyze Verbose:gc output for IBM JVMs. It is packaged within ISA and can be launched from within IGAA. PMAT will let you know if you are dealing with insufficient heap space or large object allocations.

HP-UX and Solaris have similar tools. HPJTune will analyze GC analysis formated by the JVM running on HP-UX. Likewise, the Solaris GC Portal will analyze GC output from a Sun JVM. The GC Portal is a Web Application that must be installed and run on a Web Application Server. A Java heap dump is impossible to interpret without an analysis tool. MDD4J is the preferred tool for analyzing IBM heapdumps from IBM JVMs or the IBM Heapdump Agent, as well as Solaris 1.4.2_12 and above heapdumps. MDD4J is installed under ISA and can automatically be launched from within IGAA. The Heap Analysis Tool, or HAT, is a tool designed to parse Solaris and HP-UX Hprof dumps into a human readable object tree. See the HAT documentation for more information.

Sample VerboseGC output: allocation failure

```
<AF[15]: Allocation Failure. need 10485776 bytes, 39 ms
since last AF>
<AF[15]: managing allocation failure, action=2
(43700440/85129728)>
  <GC(15): freeing class
sun.reflect.GeneratedMethodAccessor8(105962D8)>
  <GC(15): unloaded and freed 1 class>
  <GC(15): GC cycle started Wed May 17 12:08:11 2006
  <GC(15): freed 3449536 bytes, 55% free
(47149976/85129728), in 184 ms>
  <GC(15): mark: 171 ms, sweep: 13 ms, compact: 0 ms>
  <GC(15): refs: soft 0 (age >= 6), weak 0, final 0, phantom
0>
<AF[15]: completed in 185 ms>
```


Here is a sample verboseGC output generated by the PlantsByWebSphere application. This sample shows a single allocation failure, and the subsequent garbage collection.

How to interpret VerboseGC output (1 of 2)

Line 1: <AF[15]: Allocation Failure. need 10485776 bytes, 39 ms since last AF>

- How many allocation failures have occurred
- How many bytes are needed
- Time since last allocation failure

Line 2: <AF[15]: managing allocation failure, action=2 (43700440/85129728)>

- How much free space is available in the heap

Line 5: <GC(15): GC cycle started Wed May 17 12:08:11 2006

- Shows the start and timestamp of garbage collection event

Line 6: <GC(15): freed 3449536 bytes, 55% free (47149976/85129728), in 184 ms>

- How many bytes were freed, and what percentage of the heap is free
- How long it took for garbage collection to run

Line 9: <AF[15]: completed in 185 ms>

- Total amount of time it took to process the allocation failure

We should start analyzing the verboseGC output by first identifying the key fields in an allocation failure. The first line indicates that an allocation failure occured. The number in brackets is the current count of allocation failures. This line also shows us how many bytes are needed in order to satisfy the memory requirement for the object allocation request, and the time in milliseconds since the last allocation failure. On line 2, it provides more detailed information about the current state of the java heap. The information in parenthesis indicates that total heap size is over 85,000,000 bytes, and that there are over 43,000,000 bytes available. In this case, there is not enough contiguous space available, so garbage collection is started. Lines 5 and 6 show details of the garbage collection. On line 5, garbage collection starts at 12:08:11 on May 17, 2006. On line 6, GC has completed after freeing over 3,000,000 bytes in 184 ms. At this time, 55 % of the heap is now free. Finally, on line 9, it shows that it took a total of 185 ms to process allocation failure 15.

How to interpret VerboseGC Output (2 of 2)

- Inspect the allocation failure
  - Find the allocation failure that caused the OutOfMemoryError
  - Check the size of the object to be allocated
  - Determine the size of the Java heap
  - Check to see what percent of the heap is free
- Look for the pattern in previous allocation failures
  - Do we continuously get allocation failures that cause the JVM to increase the heap?
  - Is this an isolated allocation failure caused by a request for a large object?

Once we locate the OutOfMemoryError in the log, we want to first examine the Allocation Failure (AF) that caused the event to occur. We need to check the size of the object to be allocated, and the current size of the Java heap. We can also observe what percentage of the heap is currently free. This allows us to determine whether Garbage Collection should have been able to free enough memory for the object allocation, and if not, why. For instance, it would be easy to determine that the maximum heap size was not large enough for all of the requests, or if the object to be allocated is too big relative to the JVM heap. Once we come to a conclusion on the final AF, we should begin looking at AF events leading up to the OutOfMemoryError. Does the JVM heap keep increasing until we finally run into the Max Heap Size? Or does the AF that leads to the OutOfMemoryError seem to be an isolated event? By asking these questions we can determine which of the 4 causes of OutOfMemoryErrors we are experiencing.

Sample VerboseGC output: OutOfMemory (1 of 3)

```
<AF[26]: Allocation Failure. need 167772176 bytes, 55031 ms
since last AF>
<AF[26]: managing allocation failure, action=2
(49959168/256637440)>
  <GC(26): GC cycle started Thu Jun 15 13:50:39 2006
  <GC(26): freed 60297656 bytes, 42% free
(110256824/256637440), in 1442 ms>
  <GC(26): mark: 704 ms, sweep: 16 ms, compact: 722 ms>
  <GC(26): refs: soft 200 (age >= 4), weak 0, final 146,
phantom 10>
  <GC(26): moved 314304 objects, 23247400 bytes, reason=1,
used 48 more bytes>
<AF[26]: managing allocation failure, action=3
(110256824/256637440)>
  <GC(26): need to expand mark bits for 268433920-byte heap>
  <GC(26): expanded mark bits by 184320 to 4194304 bytes>
```


Here is another example that shows the processing of an OutOfMemory condition. The verbose garbage collection output comes from an IBM 1.4.2 JVM. The output will vary for other JVM's, but they will generally produce the same type of information. The first remark in bold in the verbose garbage collection output indicates that there was an allocation failure trying to allocate an object that's requesting 167.7 M of memory. The second line in bold shows the state of the Java heap at that time. The heap size is 256 M, of which only about 50 M of memory is free. The third line in bold shows that the GC cycle freed about 60 M giving a total of 110 M, or 42 % of the 256 M Java heap, available for allocation. This is still insufficient for an allocation request of 167.7, so in the 4 th line in bold the output shows that the garbage collector is starting to manage the allocation failure. The first thing that the garbage collector does is request that the JVM memory management unit expand the heap. Here we can see in the fifth line in bold that the JVM is attempting to expand the heap to 268 M, which in this case happens to be the maximum value.

Sample VerboseGC output: OutOfMemory (2 of 3)

<GC(26): need to expand alloc bits for 268433920-byte heap>

<GC(26): expanded alloc bits by 184320 to 4194304 bytes>

<GC(26): need to expand FR bits for 268433920-byte heap>

<GC(26): expanded FR bits by 368640 to 8388608 bytes>

<GC(26): expanded heap fully by 11796480 to 268433920 bytes, 45% free>

<AF[26]: managing allocation failure, action=4 (122053304/268433920)>

<AF[26]: managing allocation failure, action=6 (122053304/268433920)>

JVMDG217: Dump Handler is Processing OutOfMemory - Please Wait.

JVMDG315: JVM Requesting Heap dump file

In this first bold line we see that the JVM managed to expand the heap by about 12 meg to 268 M, and 45 % of it is free. On the next line, though, we see that the total free space is still only about 122 M, which is not enough to satisfy the 167 M request. The allocation failure routine then begins OutOfMemory processing. On the last line we see the JVM getting ready to produce a heap dump file.

Sample VerboseGC output: OutOfMemory (3 of

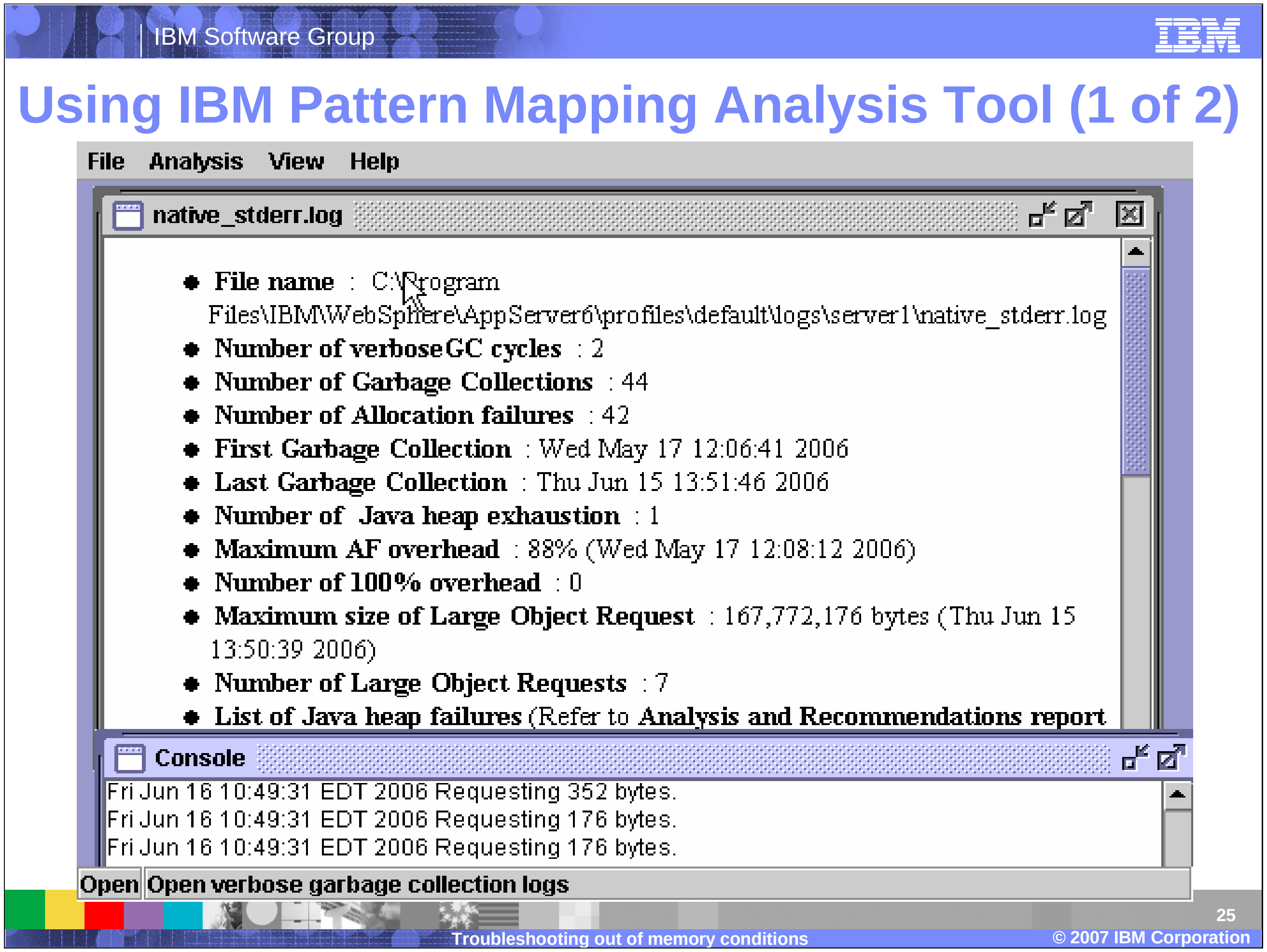
```
JVMDG318: Heap dump file written to C:\Program
Files\IBM\WebSphere\AppServer6\profiles\default\
heapdump.20060615.135039.1448.phd

JVMDG303: JVM Requesting Java core file

JVMDG304: Java core file written to C:\Program
Files\IBM\WebSphere\AppServer6\profiles\default\
javacore.20060615.135054.1448.txt

JVMDG274: Dump Handler has Processed
OutOfMemory.

<AF[26]: Insufficient space in Javaheap to
satisfy allocation request>

<AF[26]: completed in 18805 ms>
```


The verboseGC output shows us where the heap dump file is written to disk. The output then shows the JVM dumping a Java core file, and also identifies the location on disk.

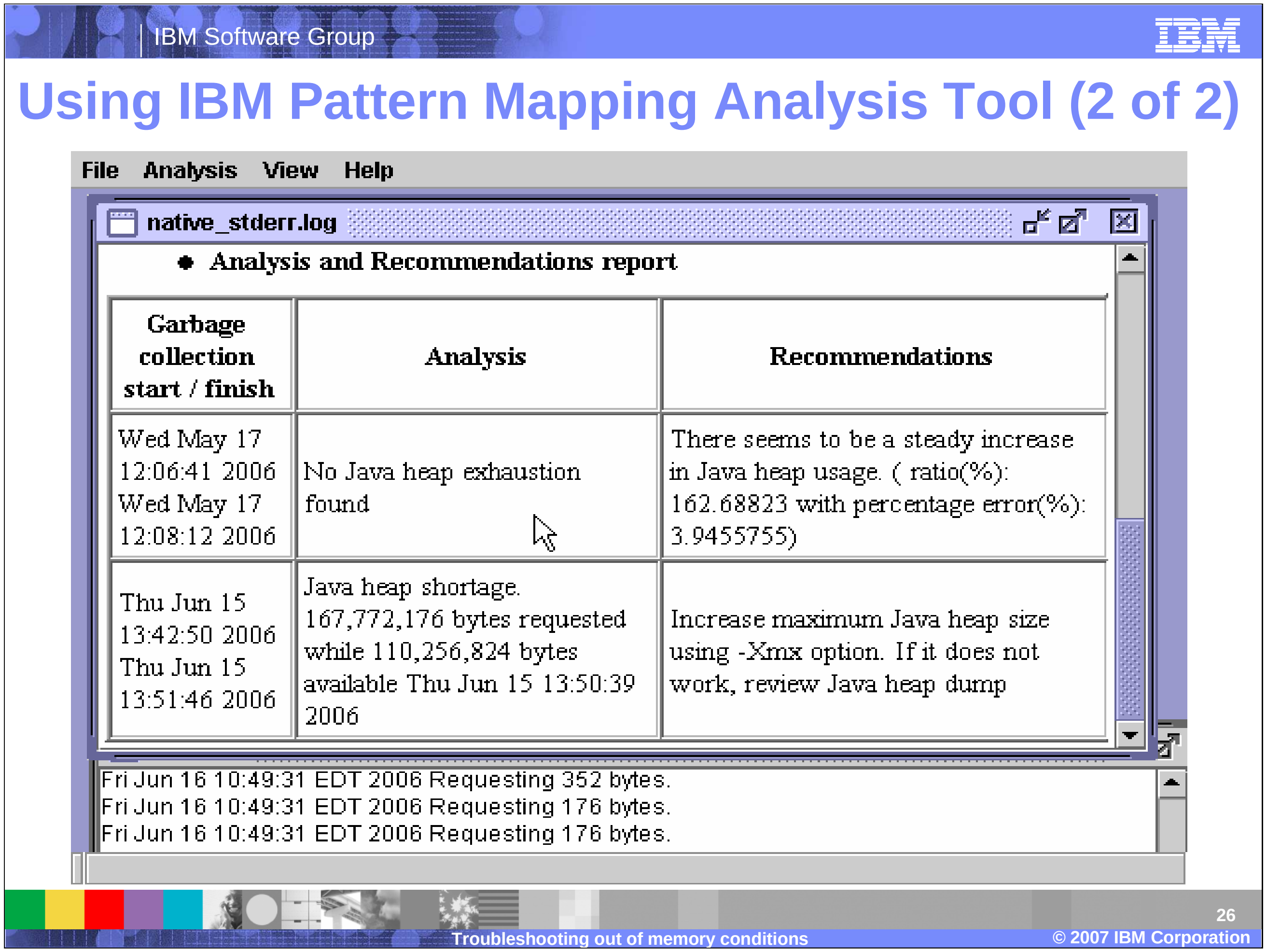
IBM Pattern Mapping and Analysis Tool

- IBM Pattern Mapping and Analysis Tool (PMAT)
  - Launched from the IBM Support Assistant
  - Tabulated and graphical views
  - Provides summary analysis
  - Analysis includes recommendations
  - Works with IBM JVM 1.3.x or 1.4.x

PMAT is a tool written by IBM Support that parses VerboseGC output from IBM 1.3.X or 1.4.X JVM's. PMAT will display all GC events in text, tabulated, and graphical views. One nice feature of PMAT is that it will also interpret the verboseGC information and provide recommendations. PMAT is available as a tool pluggin within IBM Support Assistant.



The PMAT tools contains two windows, one contains the the summary report and the other displays console events.



If you scroll to the bottom of the summary page, the tool provides recommendations. In this example the tool first suggests increasing the java heap, and if that fails it instructs the user to review a Java heap dump.

How to analyze a Java heap dump

- Heapdump
  - Use tools to parse the object trees
    - Almost impossible to do manually
  - Look for root objects holding significant memory resources
    - Includes memory allocated for the root object themselves as well as all the memory required for all the objects in the reference tree
  - Check reference tree for sudden drop in memory
    - Expand the reference tree, and examine each object as you traverse downwards
    - Compare the memory with the object right above it in the tree
    - A significant drop in memory indicates a potential memory leak
- JMap
  - Contact support to parse output
  - Manually inspect the output list
    - Difficult and time consuming

The most important diagnostic information for debugging memory leaks is a dump of the JVM heap. With proper tools, we can inspect the objects allocated in the JVM heap, traverse the object tree, and look for potential memory leak candidates. We do this by searching for root objects holding a significant portion of the memory in the heap, and traverse the object tree until we find a a location where the memory drops significantly from parent to child. When we find such a drip in memory, the parent becomes a memory leak suspect. There are two types of output that are usefull to IBM support. The first is the IBM formatted heap dump file (php). We can use the MDD4J tool to parse and display the information within the heap dumps. The other type of output is Sun‘s JMAP output. Parsing JMAP output is difficult and time consuming, and requires IBM development assistance.

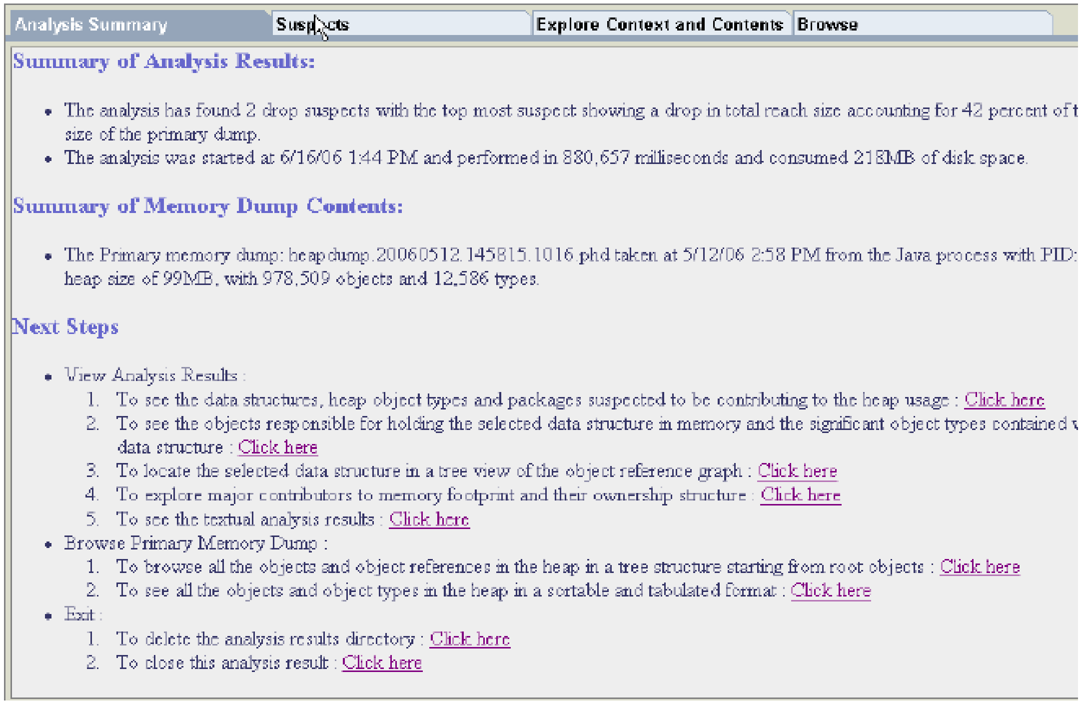
Memory Dump Diagnostic for Java tool

- Memory Dump Diagnostic for Java tool (MDD4J)
  - Analyzes IBM Portable Heap Dump (phd) files
  - Launched from within ISA
  - Analysis of primary heap dump
  - Comparison of primary and secondary heap dumps
  - Provides a graphical display of the Java object tree
  - Displays potential memory leak candidates

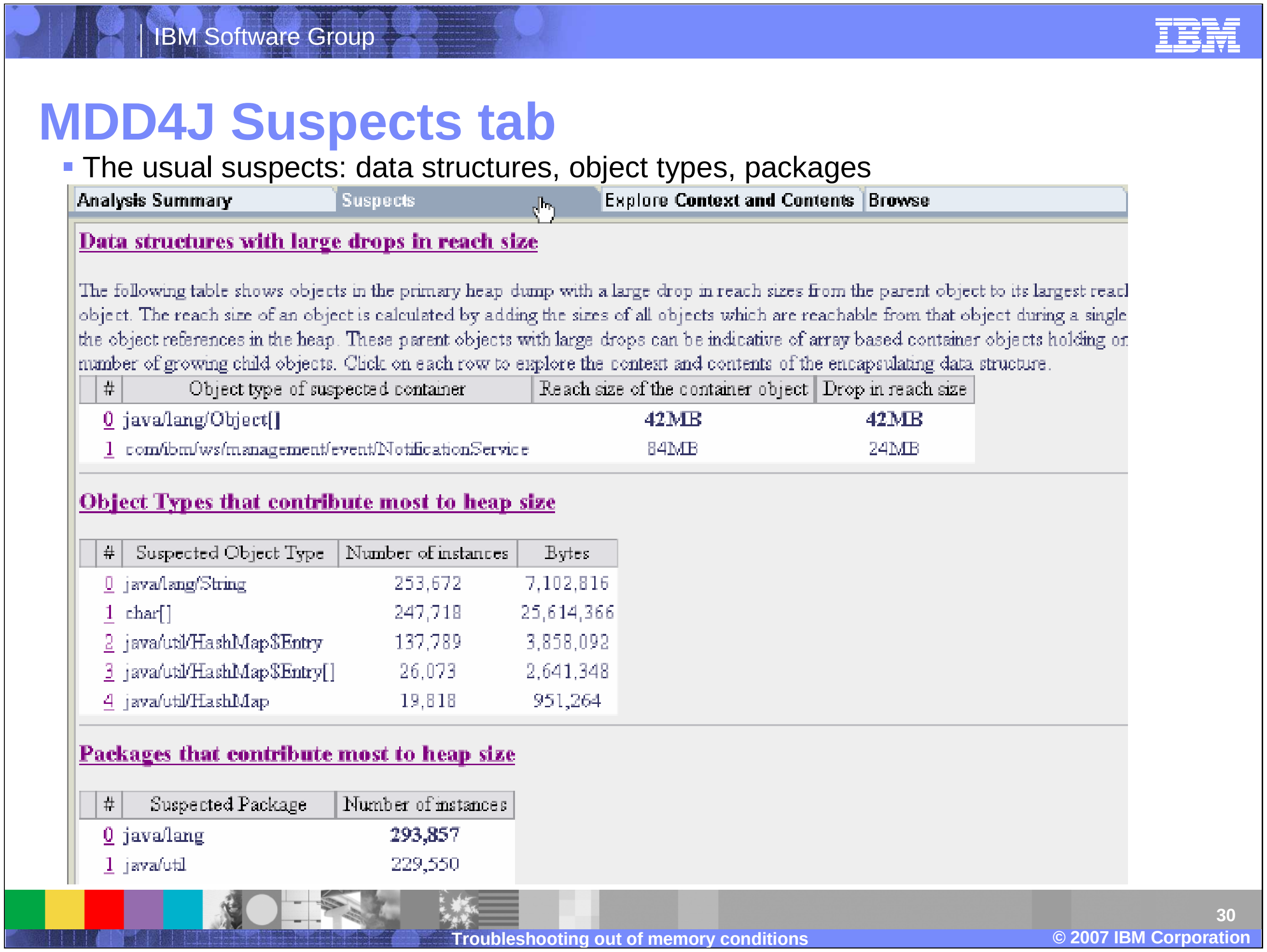
The MDD4J tool provides good analysis and interpretation of phd files. There are two forms of analysis that can be performed. The first is the analysis of a single heap dump from a single OutOfMemoryError failure. The secod is a comparison of a primary (or failure) and secondary (or baseline) heap dumps. This is useful when we are unable to determine the leaking object from the analysis of a single heapdump from the OutOfMemory failure. The baseline heap dump is provided from a JVM running in a healthy state, before the leak has consumed signifcant memory resources. We then compare a failure heap dump from the baseline to determine what the change in allocated objects is. We are focused on the analysis of primary heap dump files. The MDD4J tool will parse the heapdump and provide the results to the user. The first information of interest is the list of memory leak suspects. The tech preview version of MDD4J lists a single suspect, and the subsequent release will list 5 potential suspects. The tool also displays the data graphicaly and within a traversable object tree.

MDD4J Summary tab

Summary and Next Steps



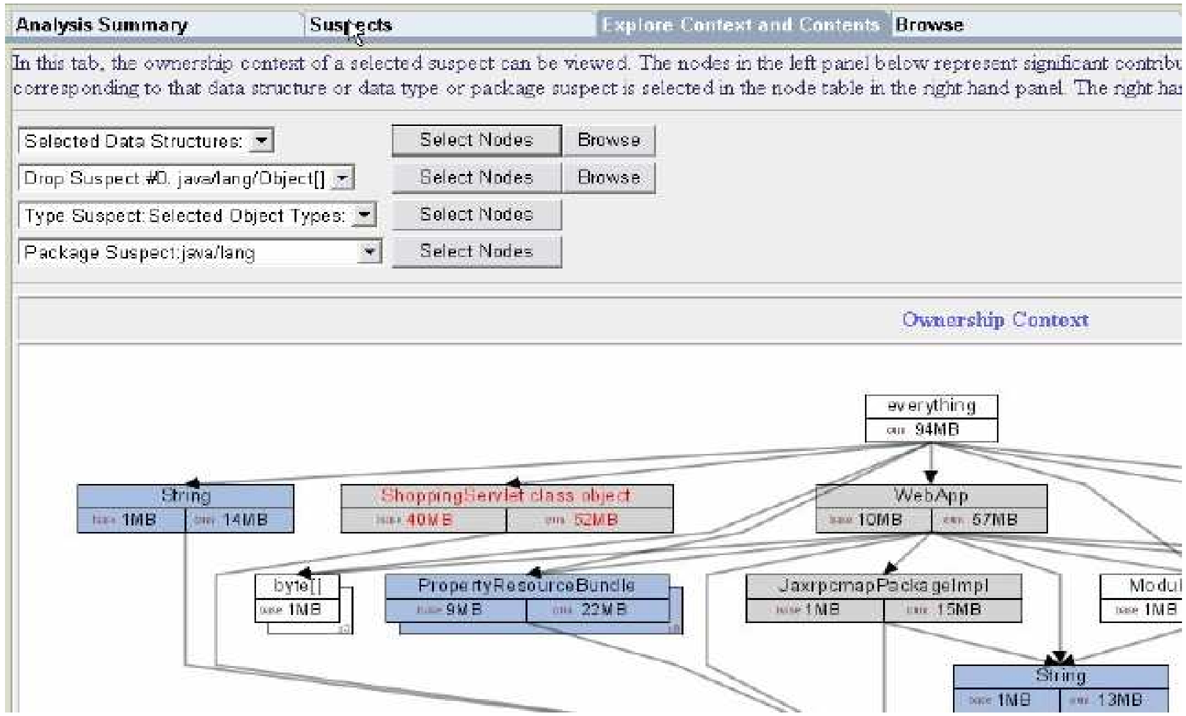
The Analysis Summary page gives a quick summary of information extracted from the heap dump. This information includes the size of the memory dumped to file, and the number of objects contained in memory. It's important to verify that the heap size dumped to file is correct, otherwise we are looking at a truncated heap dump which will not contain useful information.



The Suspects tab is where MDD4J shows us who it thinks is leaking the memory. The result provided by the current technical preview version of MDD4J only shows one potential suspect. In most cases the first suspect identified will not be the leaking object. In the next version, the top 5 suspects will be displayed, giving us much more reliable information.

MDD4J Explore Context and Contents tab

- Shows ownership context of select suspects



The Explore Context and Contents Tab shows the ownership context of selected suspects. Users can selet nodes listed on the drop down list, and graphically explore the ownership context to identify objects consuming significant memory.

The Browse tab allows us to traverse the object tree looking for significant drops in memory usage. On the left, we can see the details of the highlighed object in the tree on the right. The key information is the Total Reach Size, which tells us how much memory is being used by the highlighted object, as well as all of the referenced objects below it in the tree. We can identify the leaking object by traversing down the tree until we go from a parent to one of it's children, and the Total Reach Size drops significantly.

Unit summary

Having completed this unit, you should be able to:

- Describe what causes OutOfMemory conditions
- Use the Tivoli Performance Viewer to detect OutOfMemoryConditions
- Obtain a VerboseGC log
- Obtain a java heap dump
- Interpret a verboseGC log
- Analyze a java heap dump

Congratulations, you have completed the unit on problem determination for out of memory errors. You should now be able to:

Identify what caused an Out of memory condition, use Tivoli Performance viewer to detect OOM conditions, and obtain and analyze the pertinent diagnostic information.

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 SW5706G17_OutOfMemory.ppt

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

Trademarks, copyrights, and disclaimers

The following terms are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both:

AIX Current IBM WebSphere

Windows, and the Windows logo are registered trademarks of Microsoft Corporation in the United States, other countries, or both.

J2SE, Java, JVM, and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.