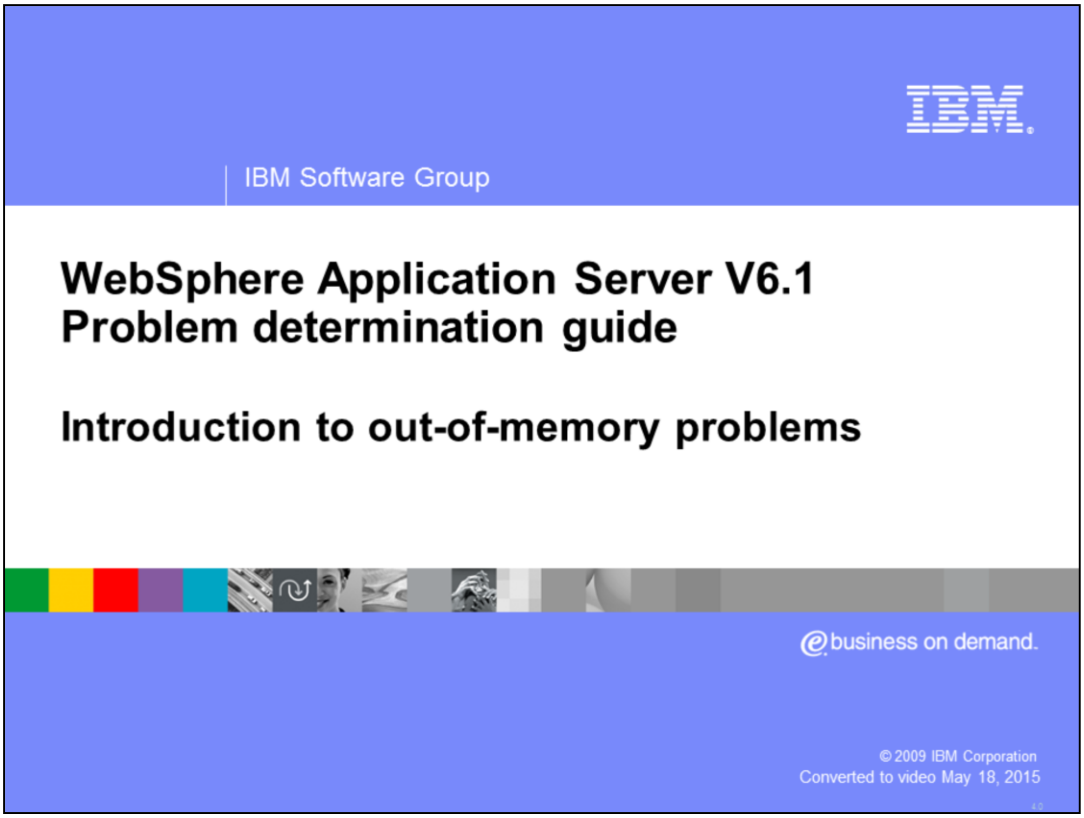
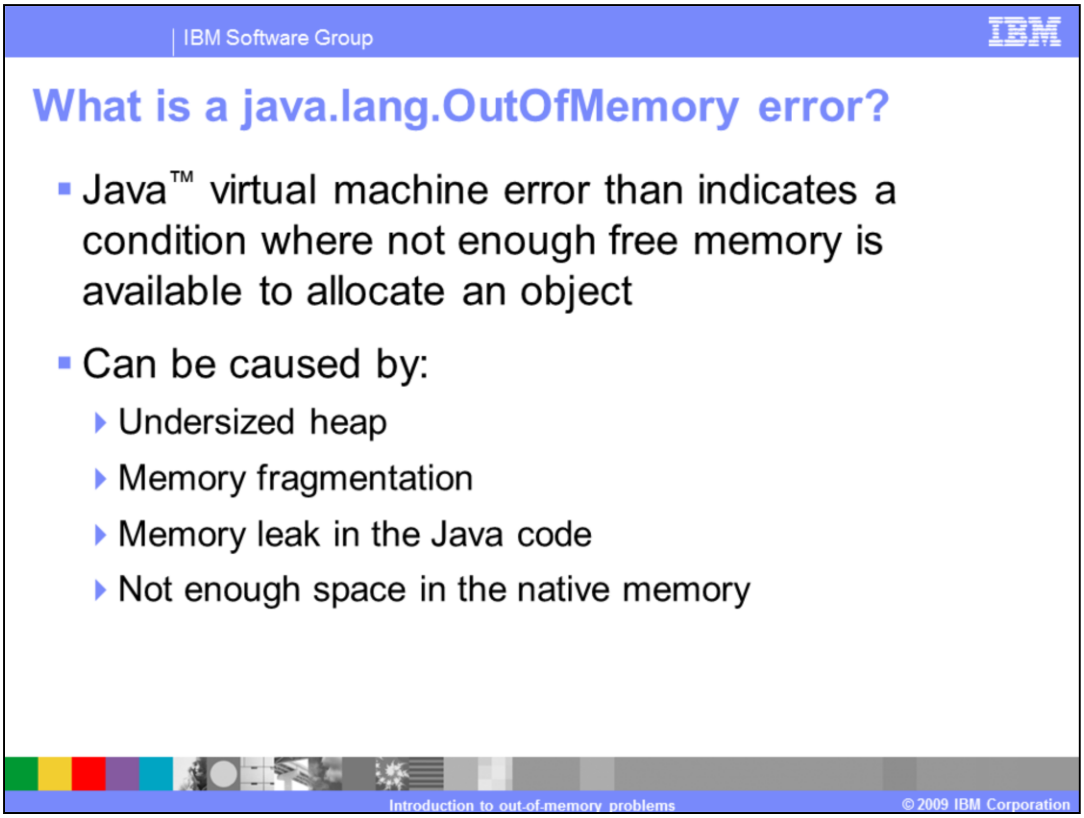
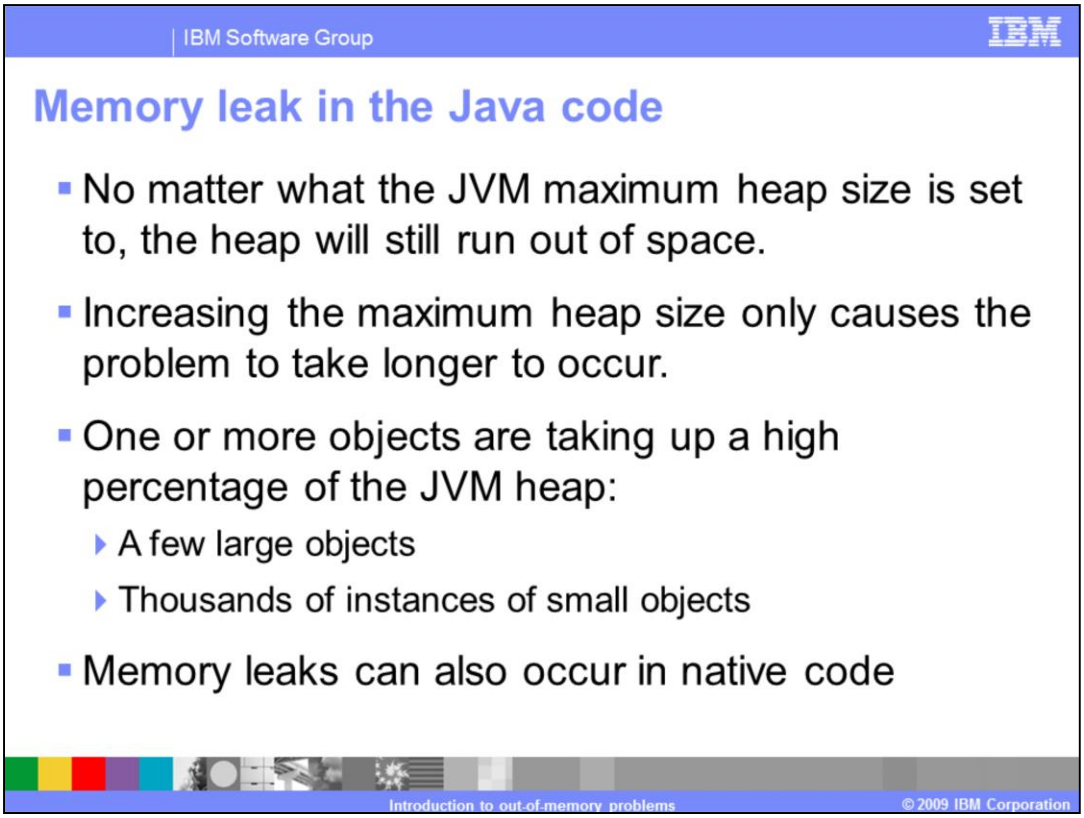


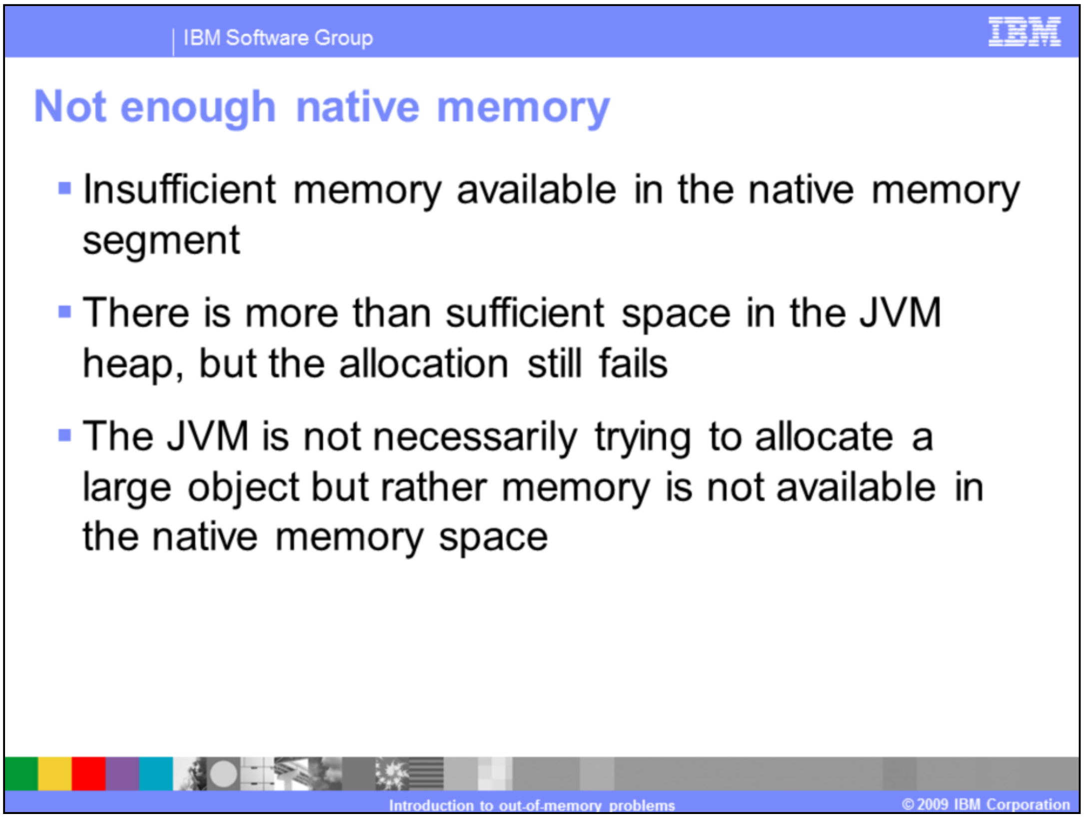
This unit covers the problem determination techniques associated with out of memory problems in WebSphere® Application Server.



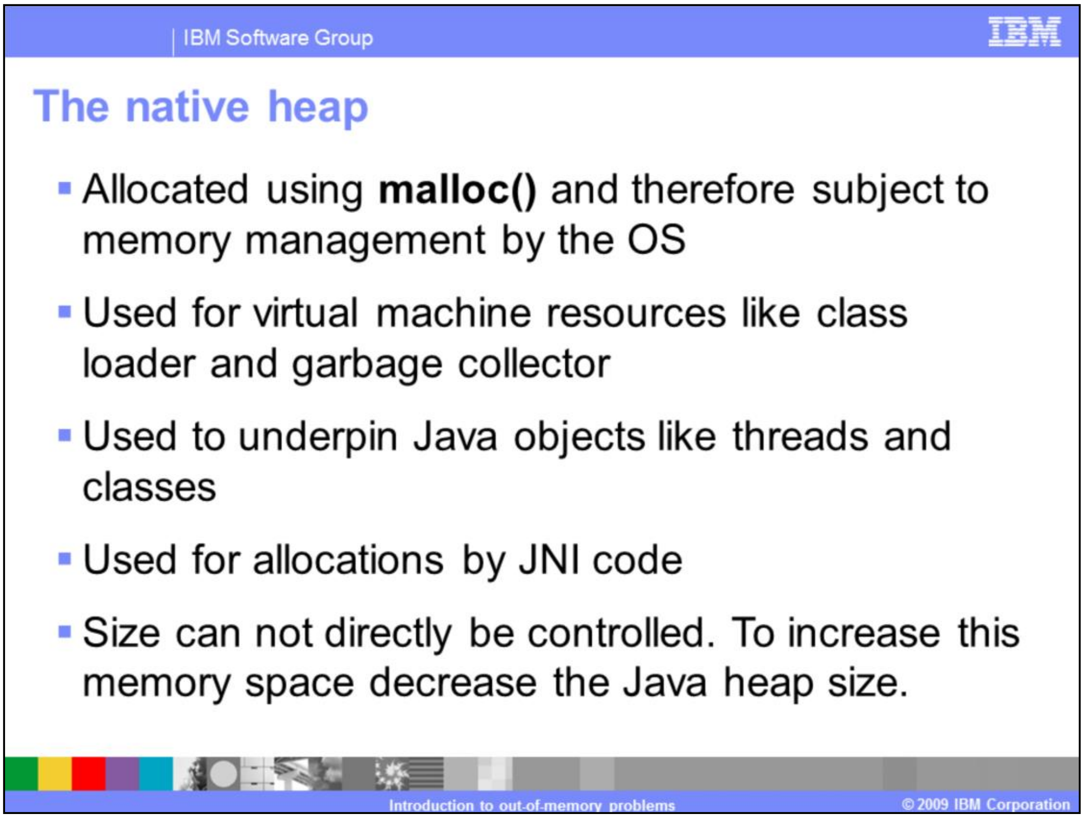
When the Java virtual machine, or JVM, tries to allocate an object and it fails, it runs garbage collection to free up heap space from objects that are no longer being used. If the object cannot be allocated after garbage collection, the JVM creates an OutOfMemoryError.



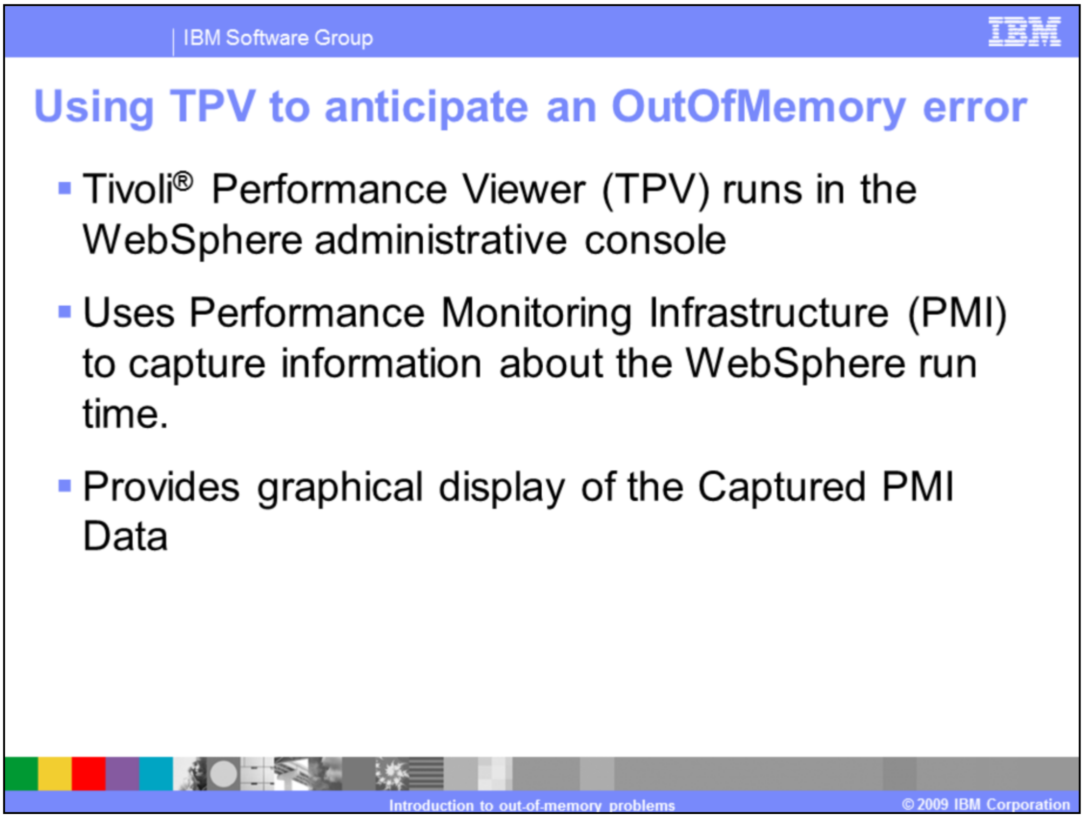
Memory is not explicitly allocated and deallocated in Java, like in C code,but it is still possible to create a memory leak. One example is to save an object into some type of collection. If the collection is a class object, and the class always stays loaded, the object will never be removed from the collection. If objects are continuously added to the collection, the collection can 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. Another commonly seen misuse of cache in WebSphere is the HTTP maximum Sessions parameter in WebSphere. HTTP Sessions can be written to store very large objects. If the HTTP maximum sessions parameter is set too high, then it will again consume a large portion of the JVM heap.



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. If the operating system is unable to provide memory to the JVM for heap expansion, the JVM will throw an out of memory. This can also occur within Java native interface code.



The native heap is the portion of the memory for the JVM process that does not include the Java heap. This space is used for holding data for the JVM’s internal components such as the garbage collector and class loader. Additionally, native memory is used to underpin or support Java data structures like threads and classes. The size of the native memory cannot be directly controlled. It is basically total memory minus the maximum Java heap size, and is further bounded by operating system constraints and libraries..



The Tivoli Performance Viewer is embedded within the WebSphere Administrative Client. It uses the Performance Monitoring infrastructure to capture information about the WebSphere runtime, such as the JVM heap size.

OutOfMemory indicators of the javacore file

- Look to see if the dump was due to an OutOfMemory condition

```
1TISIGINFO     Dump Event "throw" (00000010) Detail
  "java/lang/OutOfMemoryError" received
```

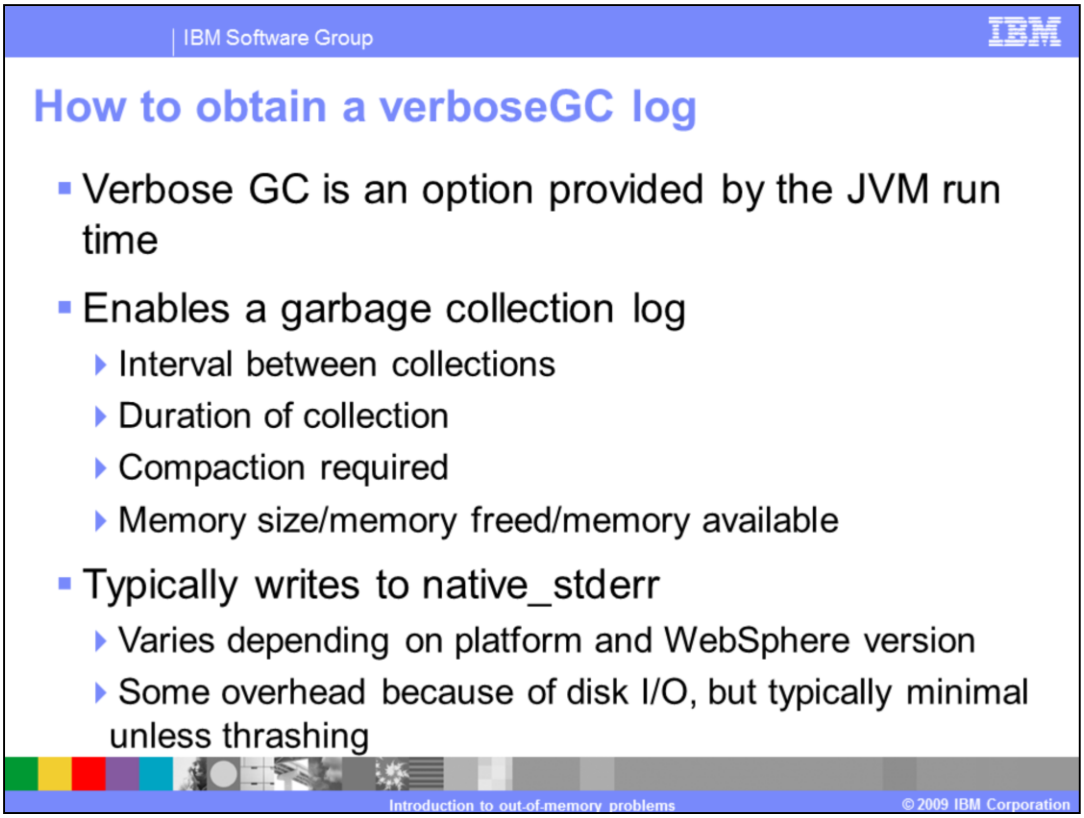
- Check MEMINFO subcomponent output (values in hexadecimal)

```
0SECTION       MEMINFO subcomponent dump routine
NULL           =================================
1STHEAPFREE    Bytes of Heap Space Free: 4bca320
1STHEAPALLOC   Bytes of Heap Space Allocated: c000000
```

- Check GC History

```
1STGCHTYPE     GC History
3STHSTTYPE     05:15:28:807904721 GMT j9mm.81 -   J9AllocateIndexableObject()
  returning NULL! 167772176 bytes requested for object of class 101189C8
```


When looking at a javacore file there are some key eye-catchers that indicate an OutOfMemory condition. One such eye-catcher is in the dump event section. The javacore should indicate very clearly that the dump was created due to an OutOfMemoryError.



The verbose garbage collection or verbosegc output is the diagnostic data used to identify what type of OutOfMemoryError condition is occurring on the system.

IBM

OutOfMemory: interpret verboseGC output

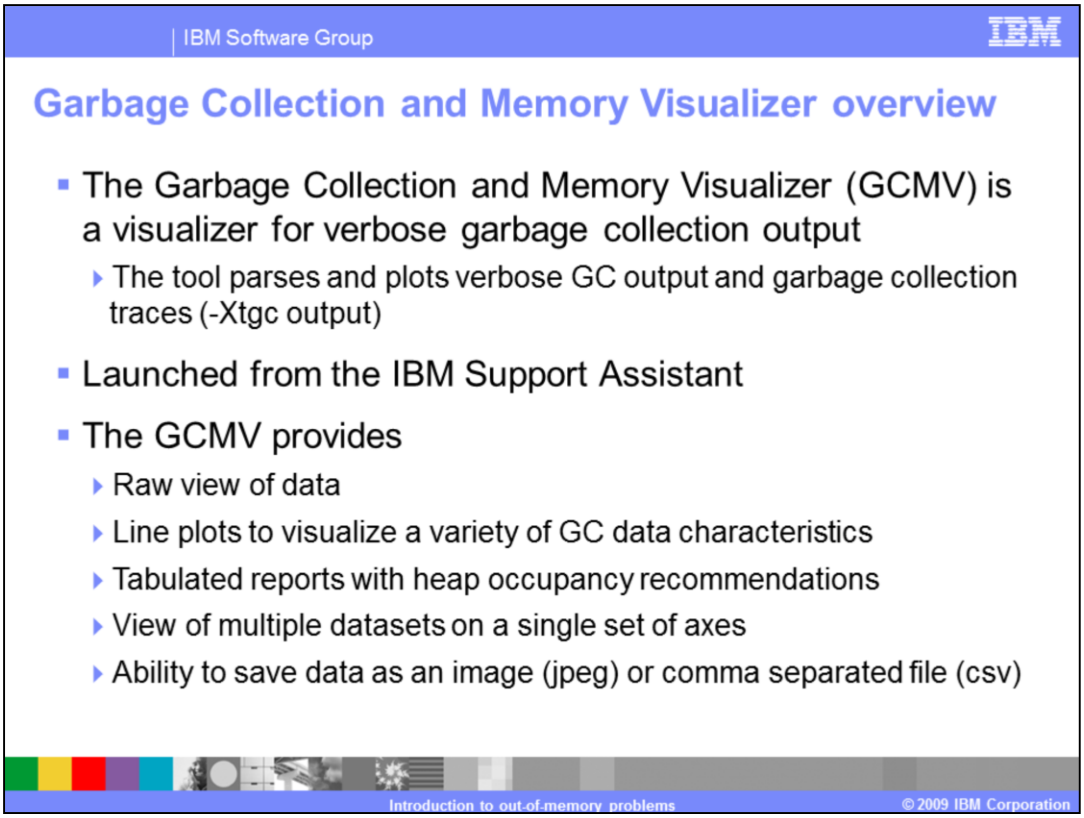
- Inspect the allocation failure
  - Find the allocation failure that caused the OutOfMemoryError
  - Check the size of the object to be allocated
  - Determine the size of the Java heap
  - Check to see what percentage of the heap is free
- Look for the pattern in previous allocation failures
  - Do you 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 you locate the OutOfMemoryError in the log, you want to first examine the allocation failure that caused the event to occur. You need to check the size of the object to be allocated, and the current size of the Java heap. You can also observe what percentage of the heap is currently free.

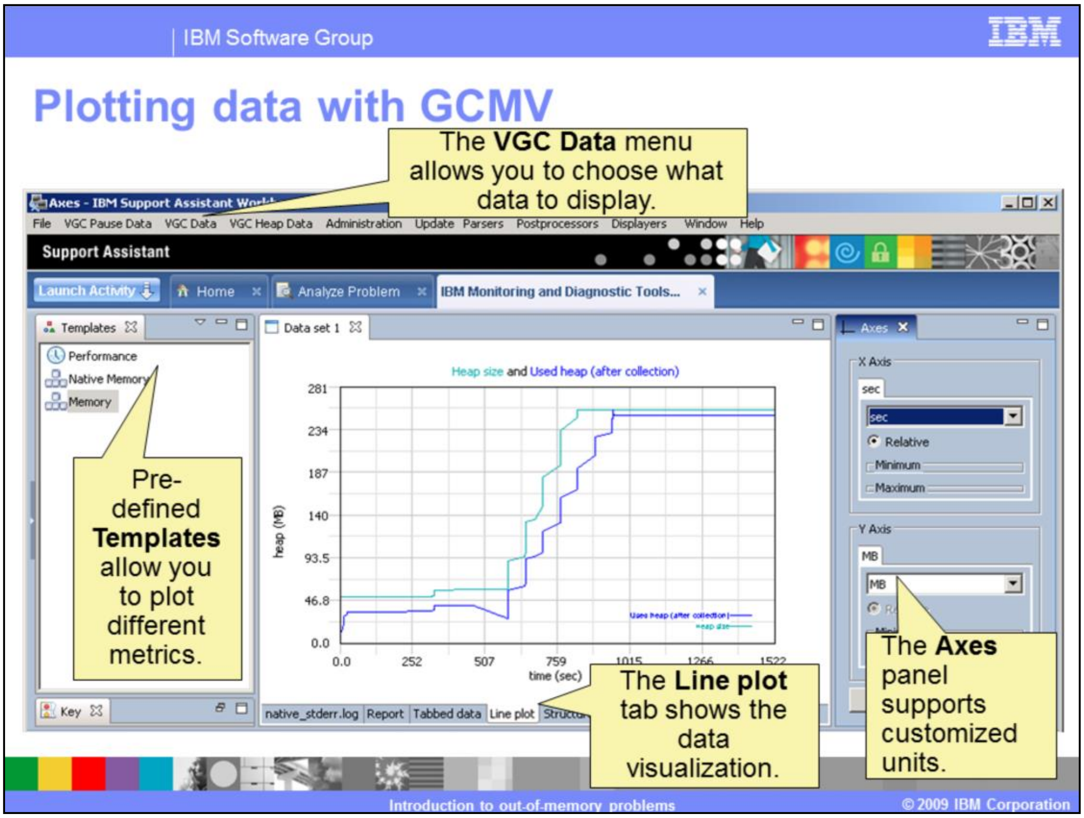
Error messages output showing dump events

```
JVMDUMP006I Processing Dump Event "throw", detail
 "java/lang/OutOfMemoryError" - Please Wait.

JVMDUMP007I JVM Requesting Heap Dump using
 'C:\Temp\Dumps\heapdump.20070715.173835.1012.phd'

JVMDUMP010I Heap Dump written to
 C:\Temp\Dumps\heapdump.20070715.173835.1012.phd

JVMDUMP007I JVM Requesting Java Dump using
 'C:\WASv61\profiles\AppSrv01\javacore.20070715.173
 835.1012.txt'

JVMDUMP010I Java Dump written to
 C:\WASv61\profiles\AppSrv01\javacore.20070715.1738
 35.1012.txt

JVMDUMP013I Processed Dump Event "throw", detail
 "java/lang/OutOfMemoryError"
```

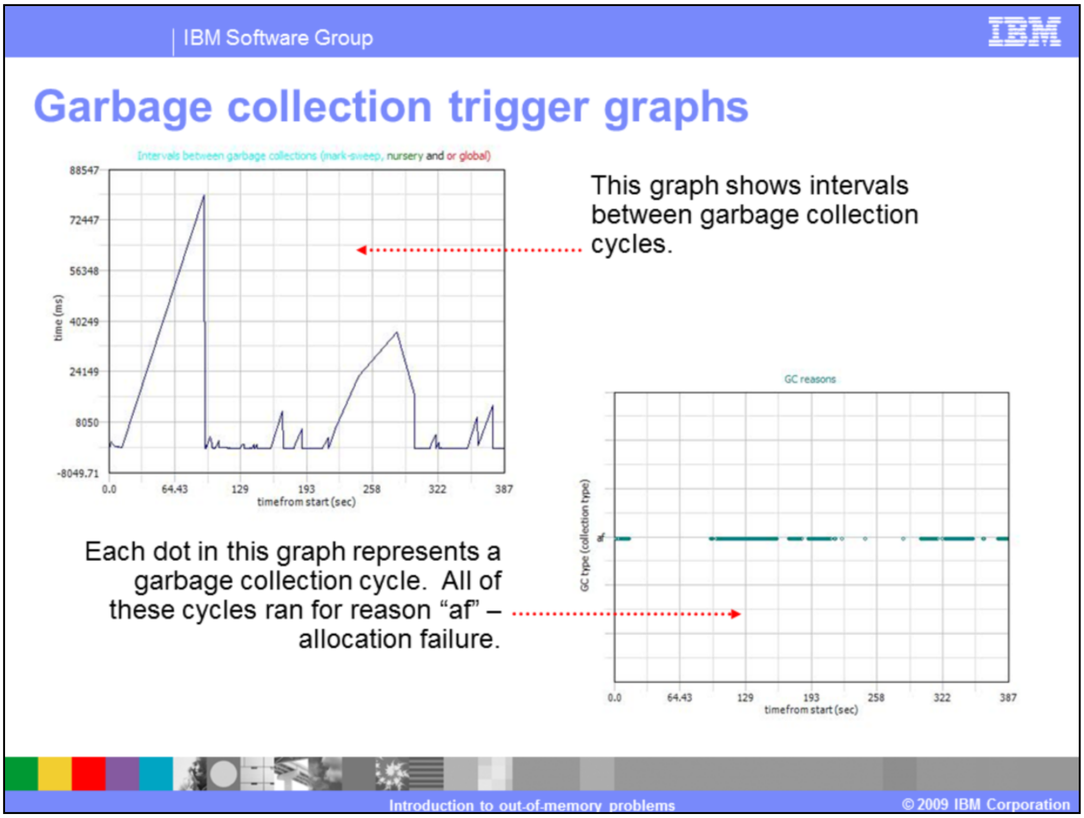

The example on this slide shows output from the verbose garbage collection log and shows different JVM-related diagnostic messages. In this case both a heapdump and a javadump were configured to be created if an OutOfMemory exception was thrown. Search the native_stderr.log output for “Heap Dump or “Java Dump” and look for the failure entry directly preceding the output.



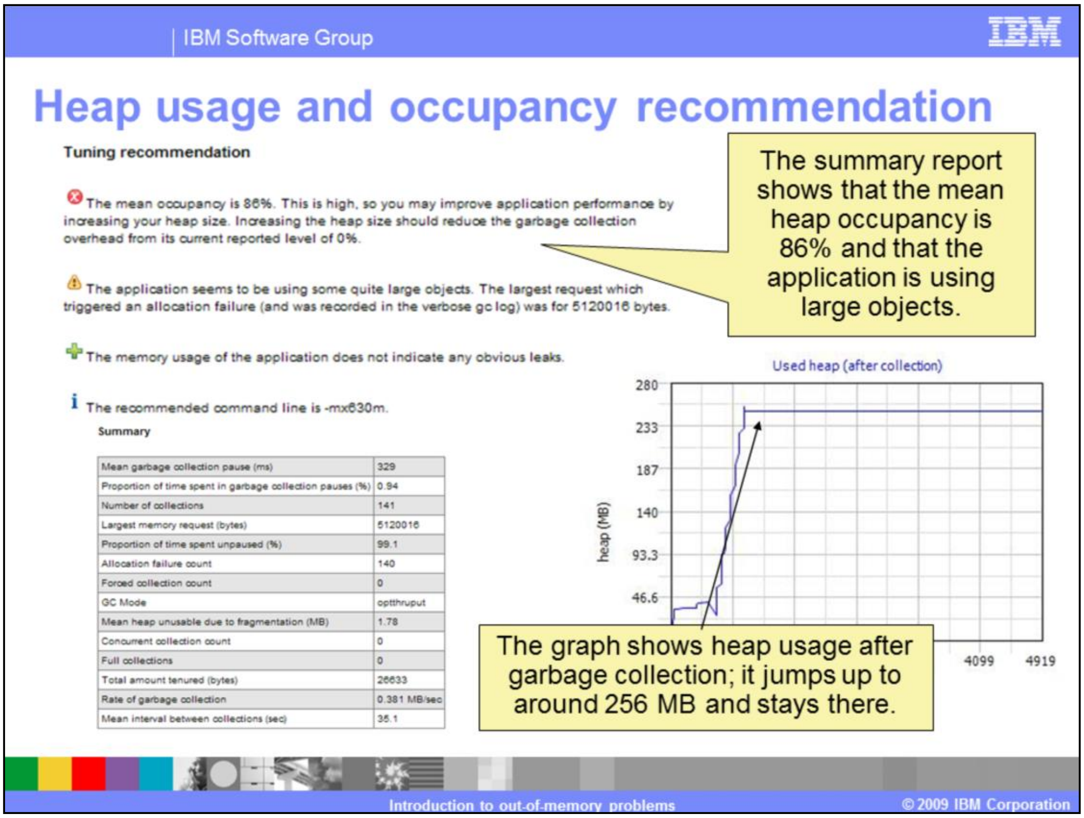
The Garbage Collection and Memory Visualizer tool is included with the IBM Support Assistant and provides reporting and visualizing for a verbose garbage collection log.



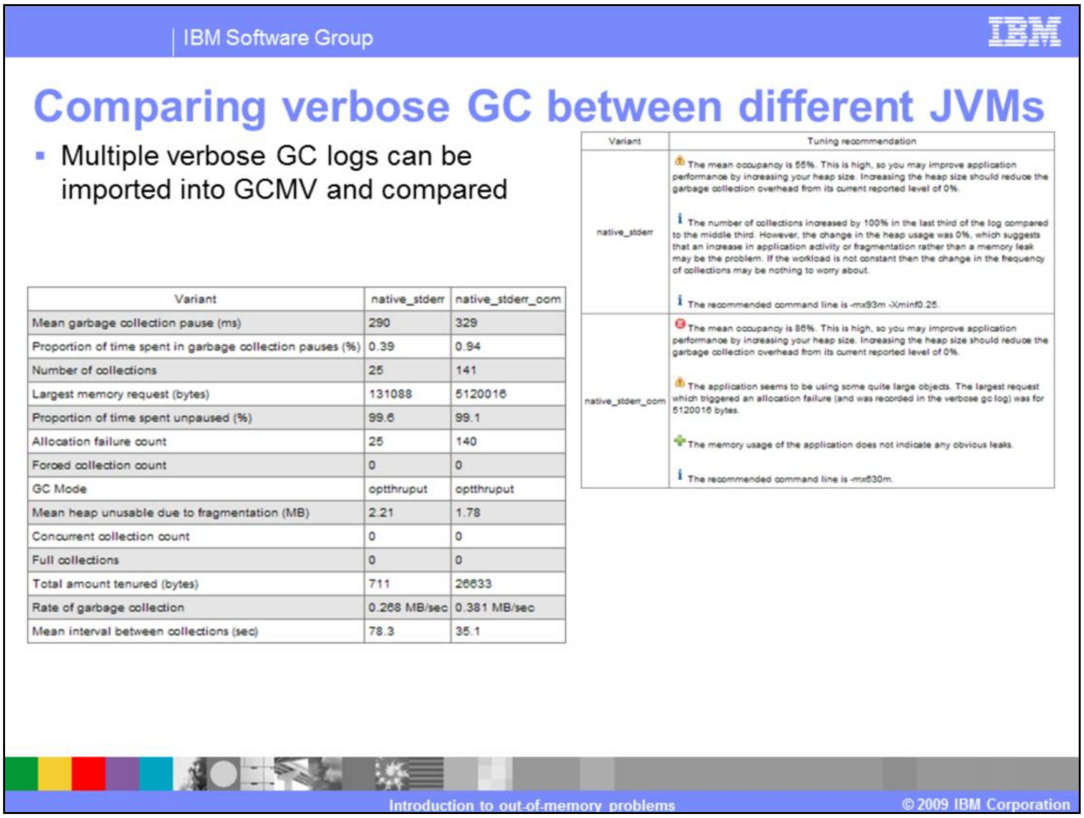
The shows a screen capture sample displaying all the major components of the graphical interface. Pictured are the line plot graphic view, the templates view, and the graph axes configuration view.



The graphs on this slide depict a sample graph showing the intervals between garbage collections on the top. The graph on the bottom of the slide shows a graph displaying the reason for each garbage collection.



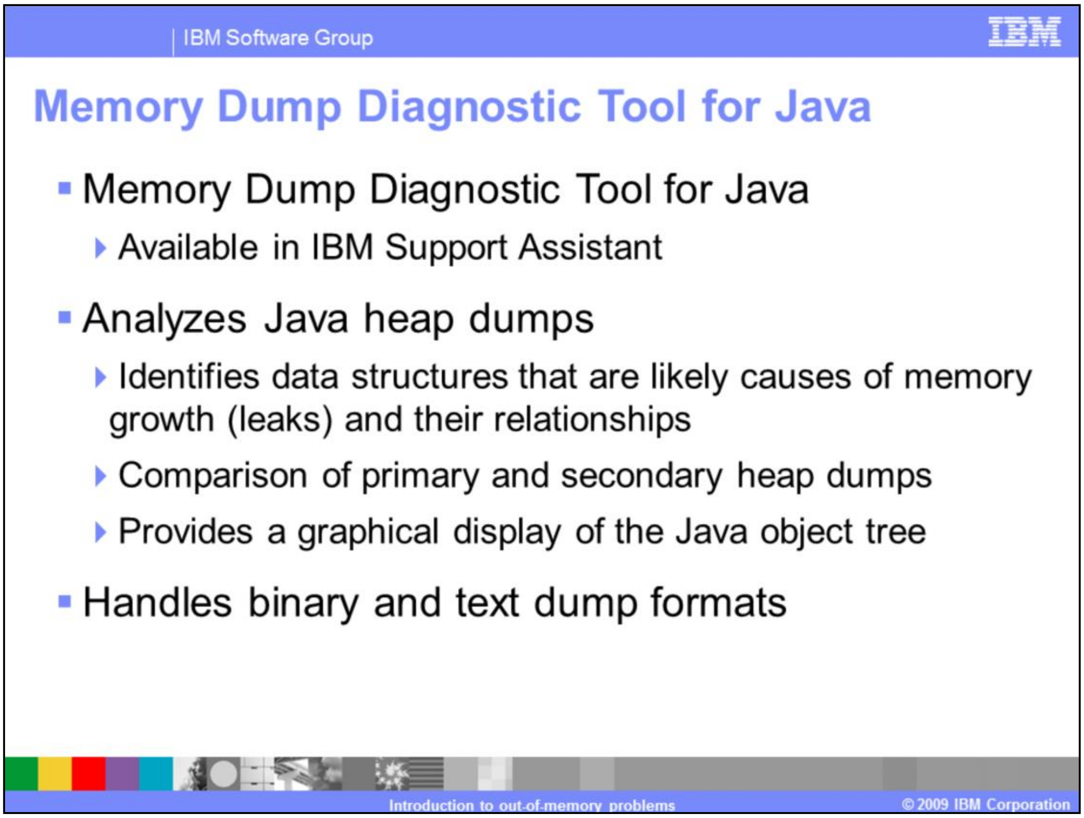
GCMV provides a summary report after parsing and analyzing the data contained in a verbose garbage collection log. The summary report makes recommendations for JVM command-line arguments that can better optimize the execution of the JVM.



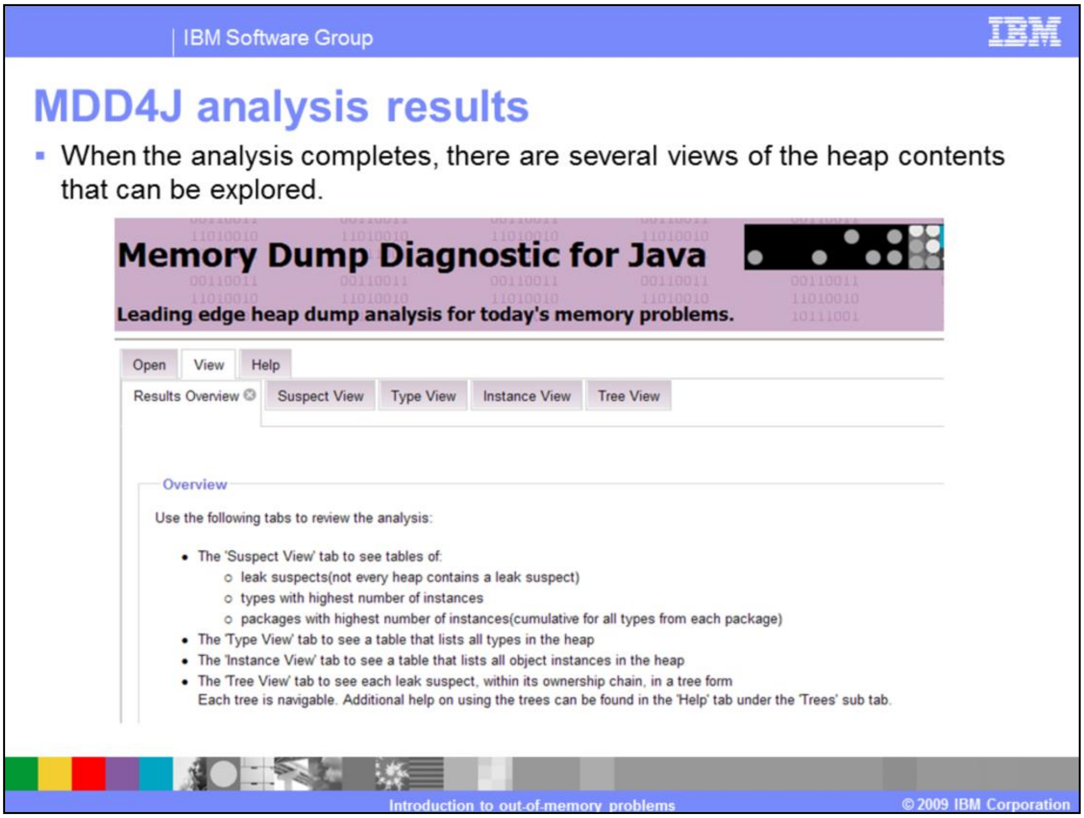
Variant	native_stderr	native_stderr_oom
Mean garbage collection pause (ms)	290	329
Proportion of time spent in garbage collection pauses (%)	0.39	0.94
Number of collections	25	141
Largest memory request (bytes)	131088	5120016
Proportion of time spent unpaused (%)	99.6	99.1
Allocation failure count	25	140
Forced collection count	0	0
GC Mode	optthruput	optthruput
Mean heap unusable due to fragmentation (MB)	2.21	1.78
Concurrent collection count	0	0
Full collections	0	0
Total amount tenured (bytes)	711	26633
Rate of garbage collection	0.268 MB/sec	0.381 MB/sec
Mean interval between collections (sec)	78.3	35.1

Variant	Tuning recommendation
native_stderr	The mean occupancy is 55%. This is high, so you may improve application performance by increasing your heap size. Increasing the heap size should reduce the garbage collection overhead from its current reported level of 0%. The number of collections increased by 100% in the last third of the log compared to the middle third. However, the change in the heap usage was 0%, which suggests that an increase in application activity or fragmentation rather than a memory leak may be the problem. If the workload is not constant then the change in the frequency of collections may be nothing to worry about. The recommended command line is -mx93m -Xminf0.25.
native_stderr_oom	The mean occupancy is 86%. This is high, so you may improve application performance by increasing your heap size. Increasing the heap size should reduce the garbage collection overhead from its current reported level of 0%. The application seems to be using some quite large objects. The largest request which triggered an allocation failure (and was recorded in the verbose gc log) was for 5120016 bytes. The memory usage of the application does not indicate any obvious leaks. The recommended command line is -mx630m.

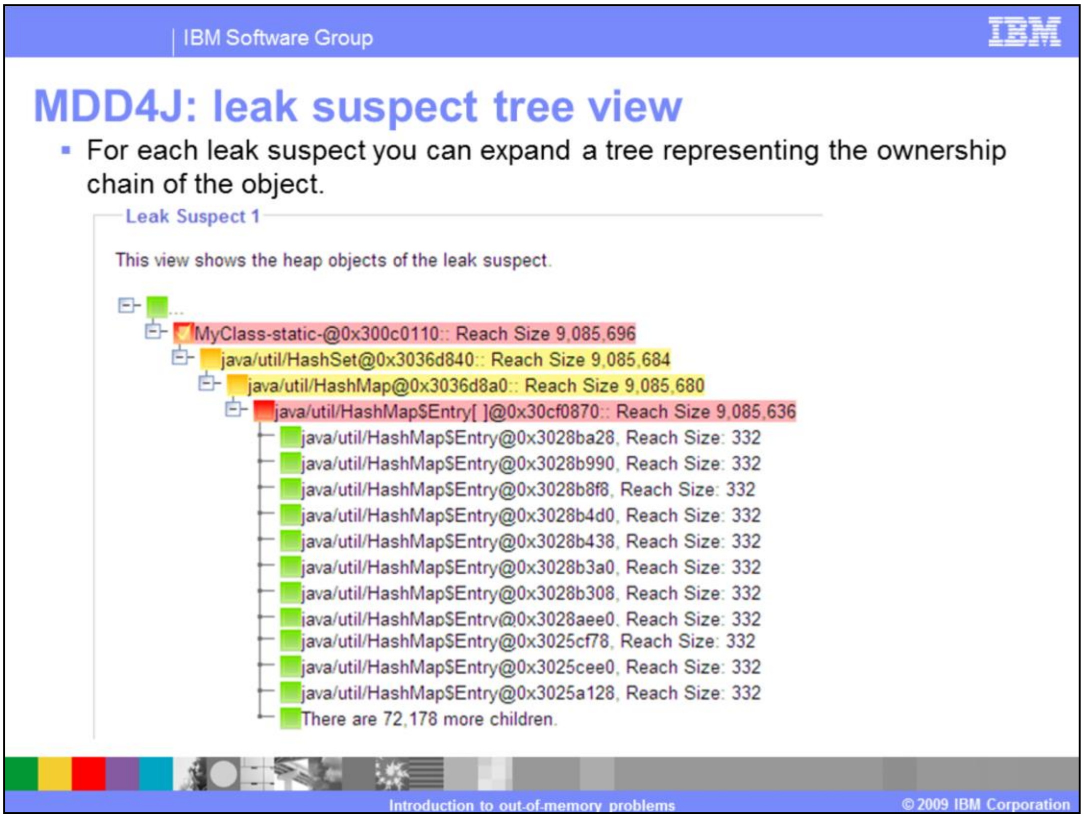
As previously mentioned, GCMV will take two garbage collection logs and compare them, providing feedback on tuning operations. The comparison feature is invaluable when it comes to JVM tuning operations.



The Memory Dump Diagnostic Tool for Java, or MDD4J, provides good analysis and interpretation of heapdump files. There are two forms of analysis that can be performed. Single heap dump analysis from an OutOfMemoryError will help in diagnosing the root cause of the crash. Multi-heapdump analysis from multiple heapdumps taken some time apart can help in determining the cause of a memory leak.



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 you are looking at a truncated heap dump which will not contain useful information.



The Suspects tab is where MDD4J shows you who is potentially leaking the memory.

MDD4J: type view

- The data type and number of all objects in the heapdump is displayed in the Object Types table.

Object Types

The table below lists all of the objects by type in the heapdump. Click on the column heading to sort the column for the entire table.

Type Name	Instance count	Array Instance count
unknown	0	106,848
boolean	0	0
byte	0	0
char	0	0
short	0	0
int	0	0
long	0	0
float	0	0
double	0	0
unknown static	0	0
MyClass static	0	0
sun/misc/Launcher$AppClassLoader	1	0
sun/misc/Launcher$ExtClassLoader	1	0
java/lang/Thread	5	2
java/lang/ref/Finalizer$FinalizerThread	1	0

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

MDD4J: instance view

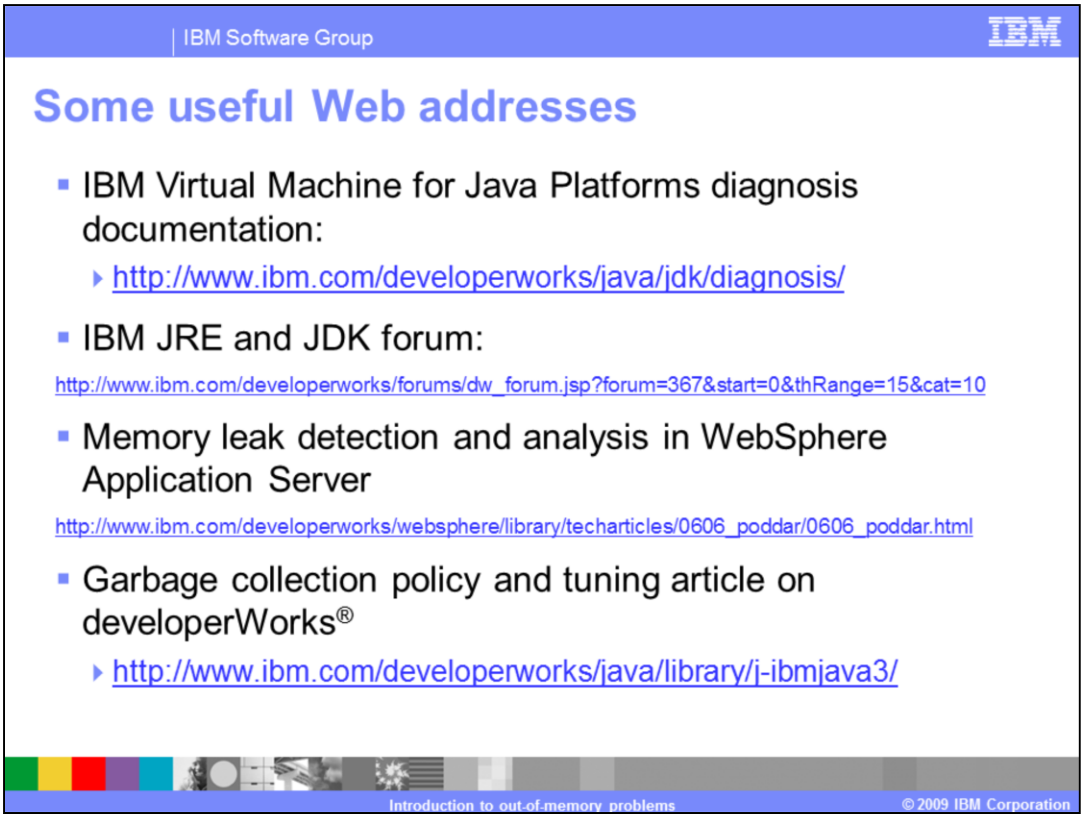
- A list of all object instances along with there *size*, *reach*, and *number of children* is displayed.

Object Instances

The table below lists information about all of the objects in the heapdump. Click on the column heading to sort the column for the entire table.

Class name	Object kind	Number of children	Size (bytes)	Reach size (bytes)	Address
unknown[ ]	primitive array	0	327,440	327,424	0x30070200
MyClass static	class	1	24	9,085,696	0x300c0110
java/util/HashSet	object	1	16	9,085,684	0x3036d840
unknown[ ]	primitive array	0	15,688	15,674	0x300c0210
sun/misc/Launcher$AppClassLoader	object	13	104	565,455	0x300c3f58
java/net/URLClassLoader$ClassFinder	object	2	24	51	0x300f2f38
sun/misc/Launcher$ExtClassLoader	object	14	104	563,763	0x300c3fc0
java/lang/ClassLoader$Finalizer	object	1	16	4	0x300f3050
java/util/Hashtable	object	1	48	107	0x300e7188
java/util/Hashtable	object	1	48	87	0x300e7118
java/security/cert/Certificate[ ]	array	0	16	3	0x3036d8d8
java/util/Vector	object	1	32	63	0x300e70b8
java/util/HashMap	object	1	56	95	0x300e7048
java/util/Vector	object	1	32	63	0x300f3098
java/util/Hashtable	object	1	48	408	0x300f3000

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



This slide contains several useful Web links that you can use to learn more about garbage collection tuning and memory leak detection.

Trademarks, copyrights, and disclaimers