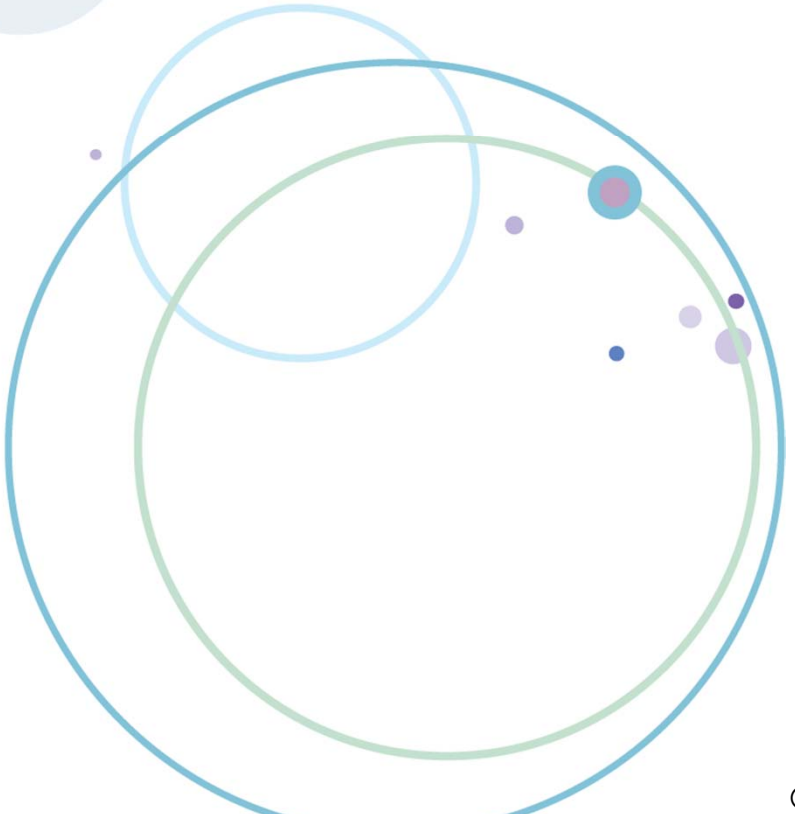
Introduction to WebSphere out-of-memory problems



Unit objectives

After completing this unit, you should be able to:

- Define out-of-memory conditions
- Use monitoring tools to detect out-of-memory conditions
- Obtain and interpret a verbose GC log by using GCMV
- Obtain and analyze heap dumps and system core dumps
- Describe tools for analyzing out-of-memory problems

Topics

- OutOfMemory error overview
- Interpret verbose GC output
- Analyze Java heap dumps and system core dumps

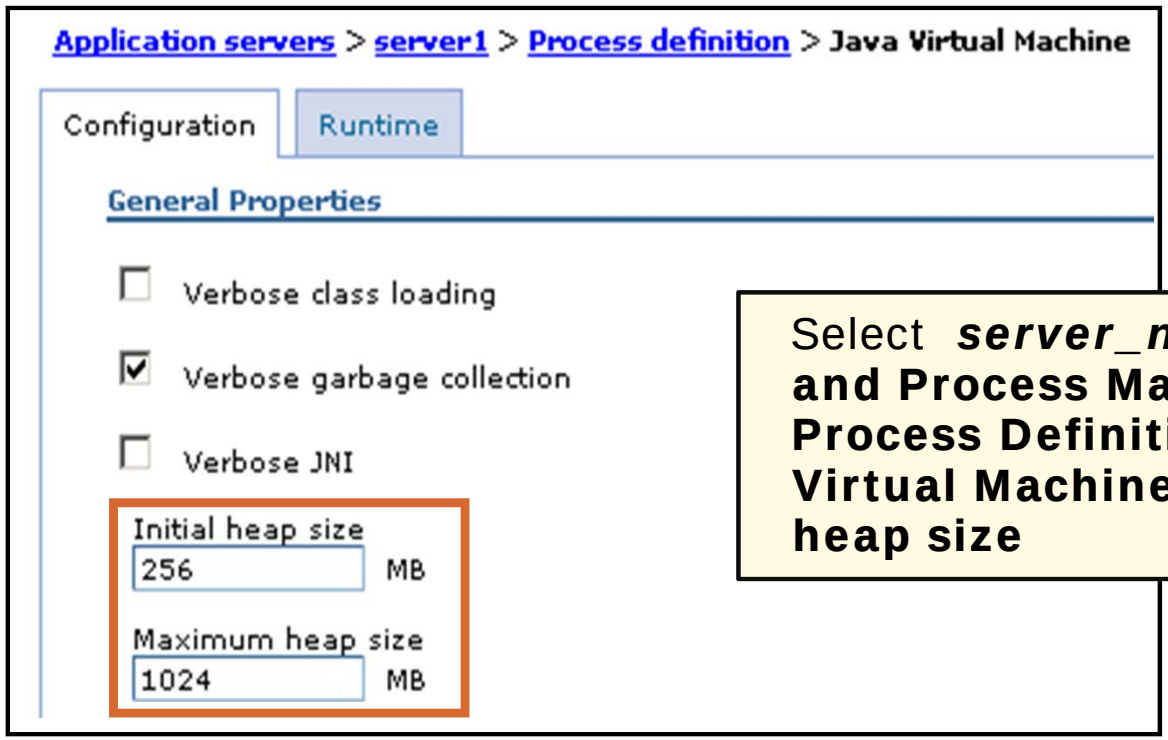
OutOfMemory error overview

What is a java.lang.OutOfMemoryError?

- Java virtual machine error
- Not enough memory to allocate an object; some of the causes are:
  - The Java heap is too small
  - Memory is available in the heap, but it is fragmented (rare for Java 6)
  - Memory leak in the Java code
  - Not enough space in the native memory
- If available, first analyze the javacore file for memory issues:
  - Verify OutOfMemory as the cause for the javacore
  - Check heap size information
- Generate and analyze garbage collection and memory information:
  - Verbose GC
  - Javacore
  - Heap dump
  - System dumps

JVM heap is too small

- How do you know that the heap is too small?
  - If the JVM heap shows constant growth until it reaches the maximum heap size, it is too small
  - The JVM heap never achieves a steady state if it is too small
- Must increase the Java maximum heap size in the administrative console



Select ***server_name*** **> Java and Process Management > Process Definition > Java Virtual Machine > Maximum heap size**

Memory fragmentation

- Space is available on the heap, but not in contiguous blocks large enough to allocate most objects
- Some fragmentation always occurs
  - Should be treated as if the heap were too small
  - Tenured size might be too small for generational garbage collection
- The object that is being allocated is excessively large
  - The JVM is required 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 that is being allocated
  - If that is not possible, increase the JVM heap
- Heap fragmentation is not common in recent versions of Java (5 and later) because of improved garbage collection modes (gencon and balanced)

Symptoms of memory leak in the Java code

- No matter what the JVM maximum heap size is set to, the heap is still going to run out of memory
- Increasing the maximum heap size merely causes the problem to take longer to occur
- One or more objects are taking up a high percentage of the JVM heap:
  - A few large objects
  - Thousands of instances of small objects
- Memory leaks can also occur in native code
  - "**Iceberg objects**": Threads, classes, and direct bytebuffers that have a small Java heap footprint, but a large native heap footprint

Class loader memory leaks

- Many memory leaks manifest themselves as class loader leaks
  - Classes with the same name can be loaded multiple times in a single JVM, each in a different class loader
  - The class loader retains a reference to every class it loaded
  - These references often remain after a web application reload
  - With each reload, more classes are pinned which leads to an out of memory error
  - The application code or JRE triggered code can cause class loader memory leaks
  - Beginning in V8.5, you can configure WebSphere Application Server to detect, prevent, and take action, if possible, on class loader memory leaks by using the memory leak detection policy

- You can also use the IBM Classloader Analyzer tool, available in IBM Support Assistant, to detect and analyze class loader problems

Not enough native memory

- Insufficient memory available in the native memory segment
- There is more than sufficient space in the JVM heap, but the allocation still fails
- The JVM is not necessarily trying to allocate a large object, but rather memory is not available in the native memory space
- The JVM attempts to create an OutOfMemoryError, but might not have the necessary resources

Java process restrictions

- Not all Java process space is available to the Java application heap
- The Java runtime needs memory for:
  - The Java virtual machine resources
  - Backing resources for some Java objects
- This memory area is part of the native heap
- Most memory that is not allocated to the Java heap is available to the native heap
  - Available memory space minus Java heap ≈ native heap
- Effectively, the Java process maintains two memory pools: Java and native

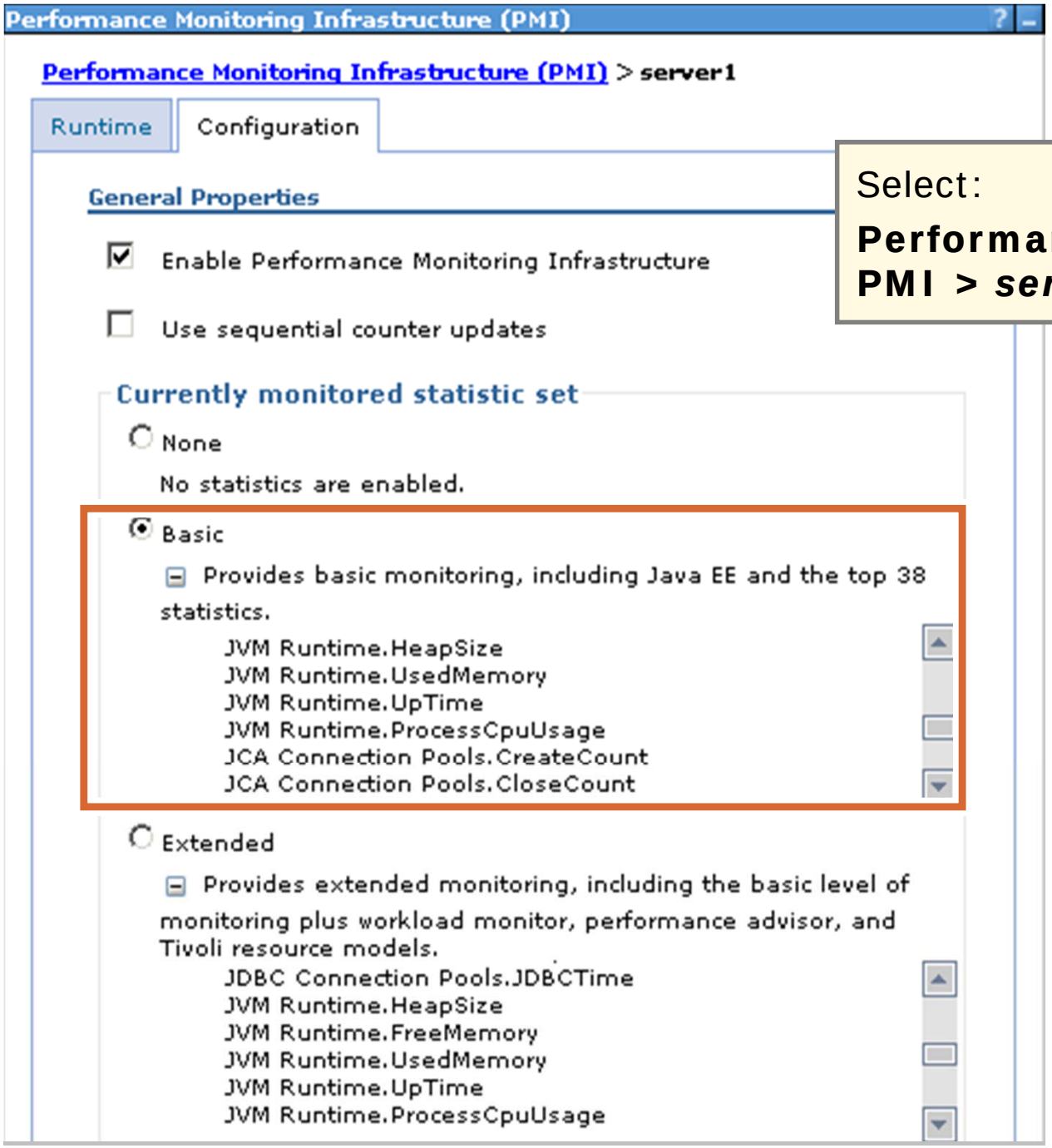
The native heap

- Allocated using `malloc()` and `mmap()` therefore subject to memory management by the OS
- Used for virtual machine resources:
  - Execution engine
  - Class loader
  - Garbage collector infrastructure
- Used to underpin Java objects:
  - Threads, classes, AWT objects
- Used for allocations by JNI code
- Size cannot directly be controlled
  - To increase this memory space, decrease the Java heap size

Using Tivoli Performance Viewer to anticipate an OutOfMemoryError

- Tivoli Performance Viewer runs in the administrative console
- Uses Performance Monitoring Infrastructure (PMI) to capture information about the WebSphere run time
- Provides graphs and charts of the captured PMI data

Administrative console PMI settings

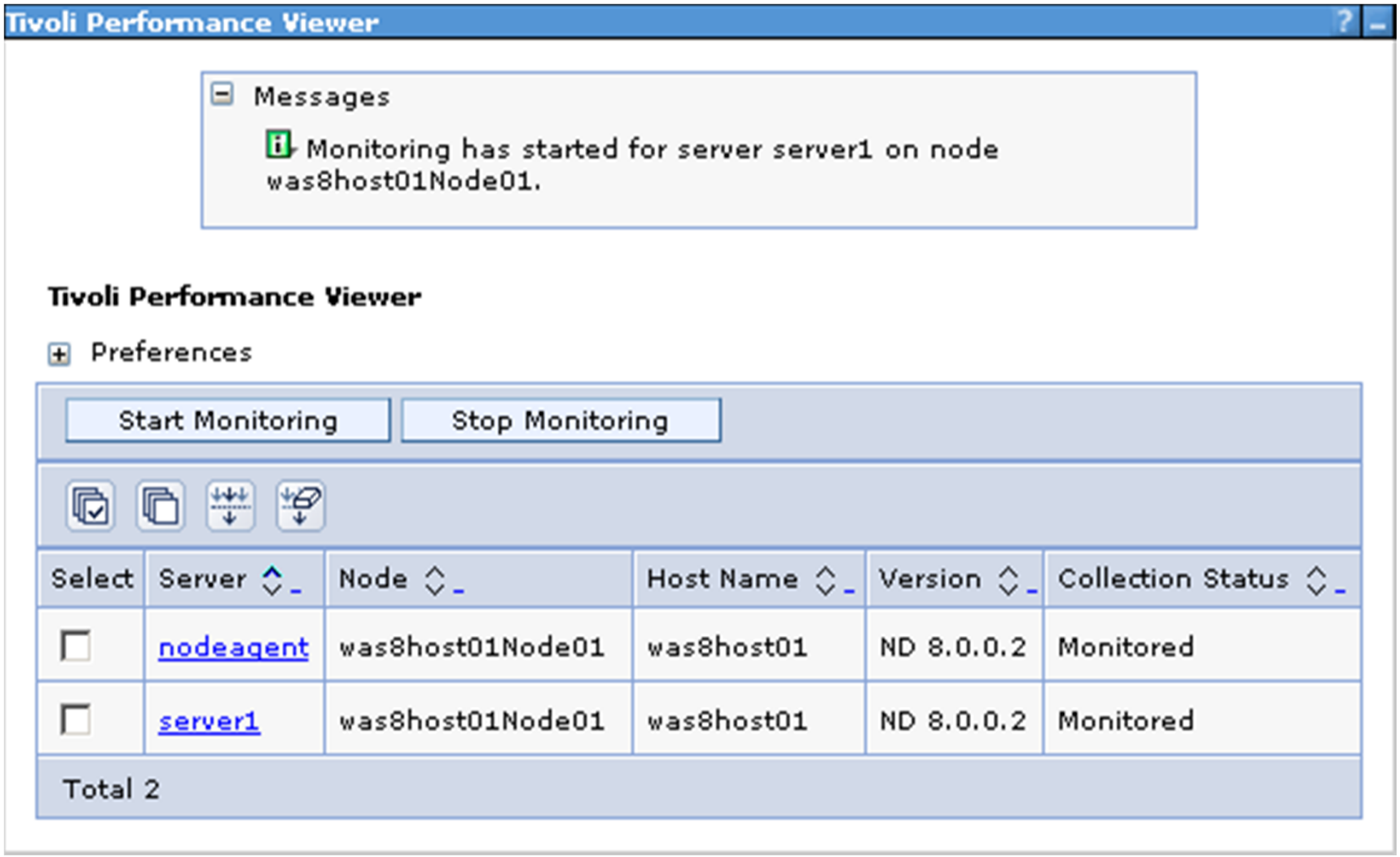


Select:

Performance and Tuning > PMI > *server_name*

Administrative console Tivoli Performance Viewer monitoring

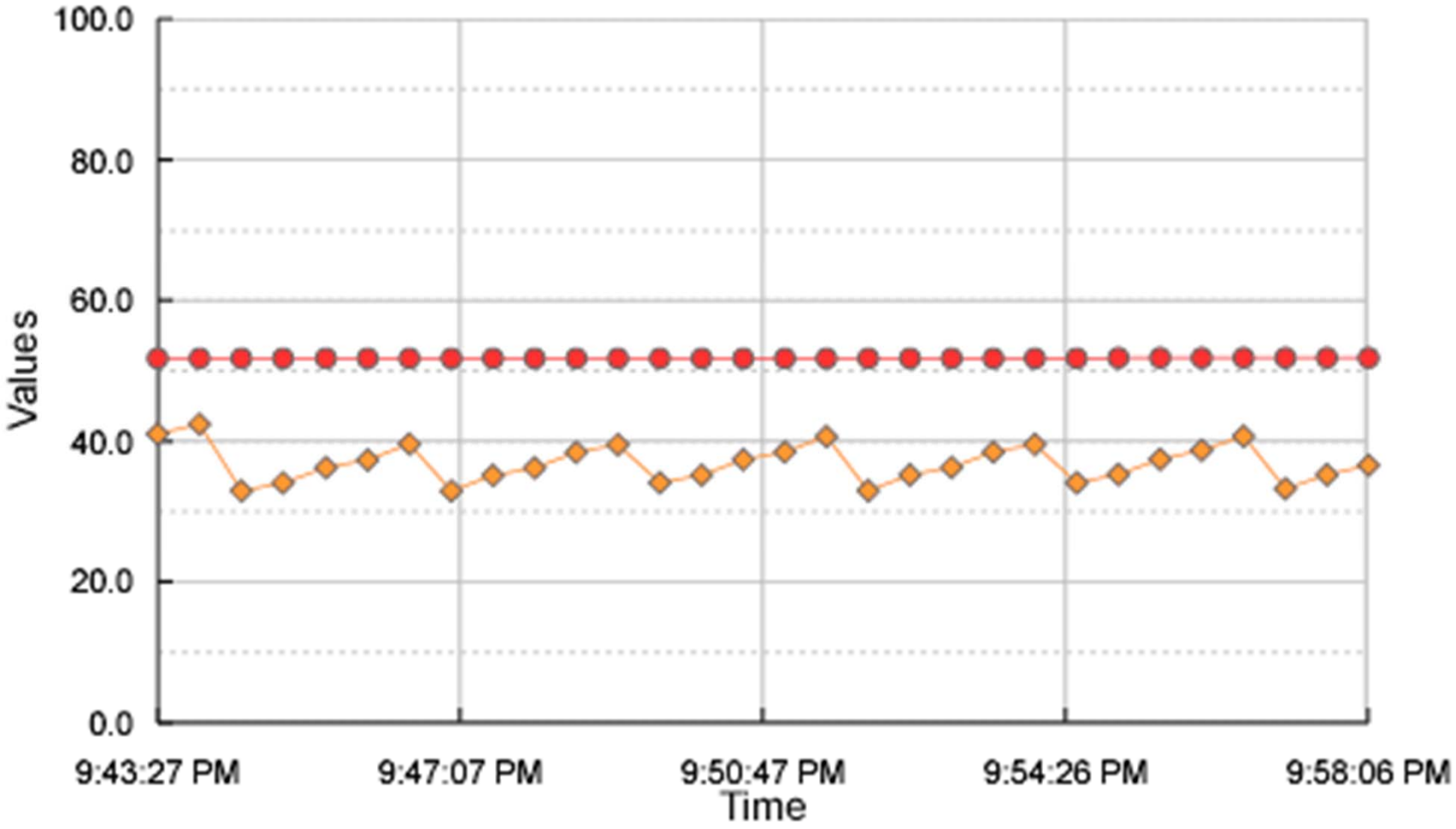
- Select **Monitoring and Tuning > Performance Viewer > Current Activity > *server_name***, and click **Start Monitoring**



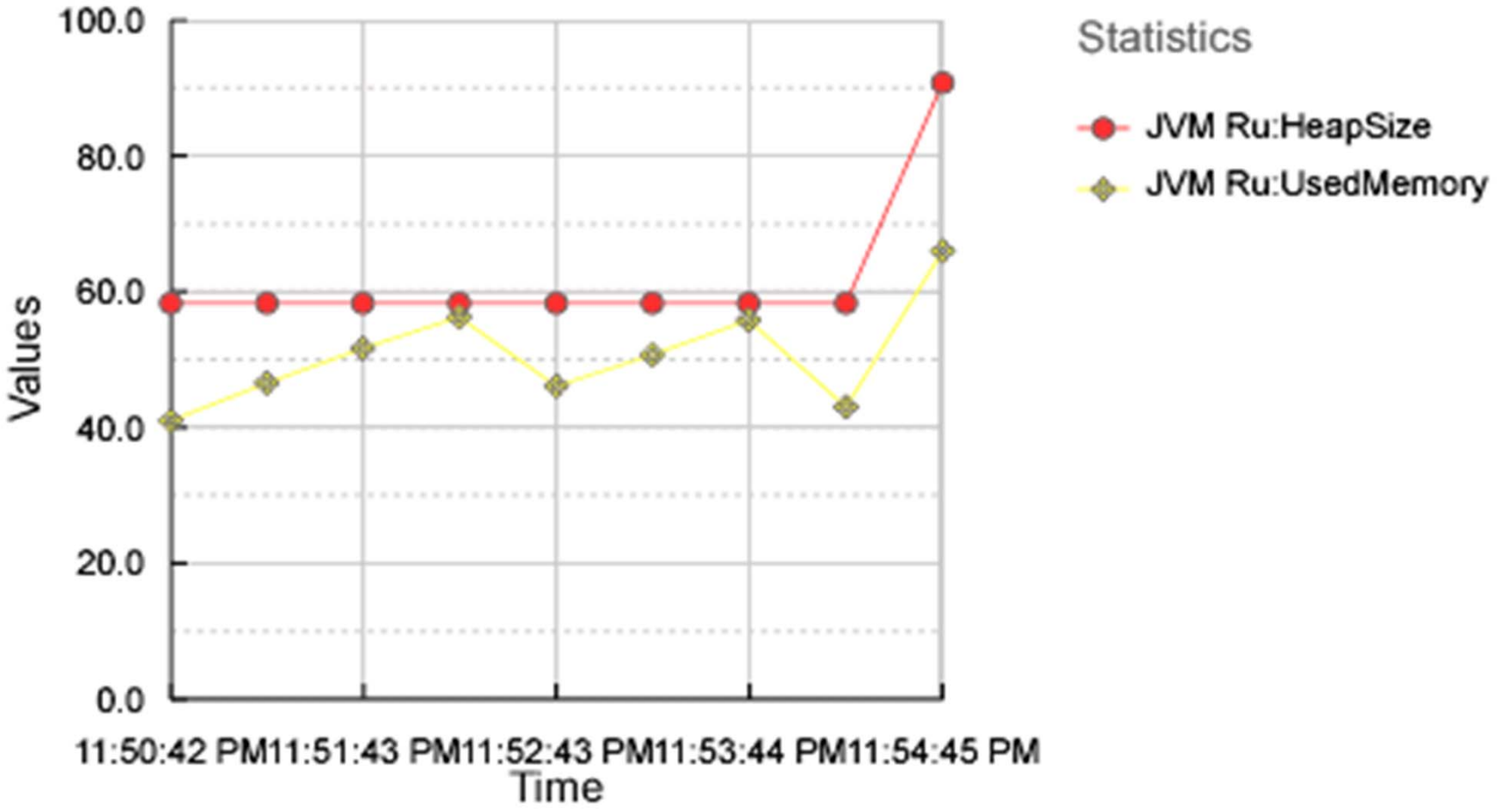
Administrative console Tivoli Performance Viewer graph

Red line (top) = Heap size

Orange line = Used memory



Tivoli Performance Viewer graph: Heap expansion



Tivoli Performance Viewer usage

- Tivoli Performance Viewer
  - Monitor internal thread pools and heap statistics
- Know the expected behavior of your application: I/O intensive, JMS, number of EJBs, expected load requirements
- Use Tivoli Performance Viewer to monitor the heap usage
  - If the used heap continues to grow overtime with no corresponding increase in user load, there might be a memory leak
  - Use a profiler to dig into the specific heap to determine where the problem exists
  - The Java Virtual Machine Tool Interface (JVMTI) is a native programming interface that provides tools the ability to inspect the state of the Java virtual machine (JVM)

OutOfMemory indicators in the javacore file

- Look to see whether the dump was due to an OutOfMemoryError

```
1TISIGINFO     Dump Event "systhrow" (00040000) Detail
"java/lang/OutOfMemoryError" received
```

- Check MEMINFO subcomponent output

```
0SECTION       MEMINFO subcomponent dump routine
NULL           ================================
1STHEAPTOTAL   Total memory:           268435456 (0x10000000)
1STHEAPINUSE   Total memory in use:    268013560 (0x0FF98FF8)
1STHEAPFREE    Total memory free:         421896 (0x00067008)
```

- Check GC history

```
j9mm.133 -   Allocation failure start: newspace=18080/56623104
oldspace=435936/201326592 loa=0/0 requestedbytes=1024008
```

- Free memory indicator
  - If the free space it tight, increasing the size of the heap might solve the problem
  - If there is much free space available, the problem might be due to a large object allocation or problem with native memory allocation

How to obtain a verboseGC log

- The JVM runtime provides the Verbose GC option
- Enables a garbage collection log
  - Interval between collections
  - Duration of collection
  - Whether compaction is required
  - Memory size, memory that is freed, memory available
- Enable the verbose GC for the server by using the administration console
  - **Servers > Server Types > WebSphere application servers > *server_name***
  - Under Server Infrastructure, click **Java and Process Management > Process Definition > Java Virtual Machine**
  - Select **Verbose Garbage Collection** check box
  - Restart the server or configure on runtime tab
- Usually writes to native_stderr
  - Varies depending on the operating system and WebSphere version
  - Some performance impact because of disk I/O, but usually minimal unless thrashing

Default behavior for OutOfMemory exceptions

- Beginning with Version 8.0.0.2, WebSphere Application Server comes with IBM Java 6 R26 on supported operating systems
  - In this version, the default behavior for OutOfMemory (OOM) exceptions is changed
- By default, the first OOM for the lifetime of a Java process produces the following dumps:
  - PHD-formatted heap dump
  - Java dump file
  - Snap dump file
  - Operating system dump: **core** file on Linux, AIX, and IBM i, **user-mode minidump** with full memory on Windows operating systems, and **SYSTDUMP** on z/OS
- The second, third, and fourth OOM exceptions produce only a PHD-formatted heap dump and a Java dump file
- Therefore, the change in default behavior is an extra system dump on the first OOM exception

System dump files

- A system dump is a superset of a PHD heap dump
- A system dump also includes memory contents (strings, primitives, variable names, and others), thread and frame local information, some native memory information, and more
- This added information can solve a larger class of problems, provide more general insight into a running JVM, and ultimately reduce the time that it takes to solve a problem
- In earlier versions of IBM Java, a system dump had to be post-processed using the `jextract` tool
- With recent versions of IBM Java, a DTFJ-capable tool, such as the Memory Analyzer Tool, can directly load a system dump without any post-processing for OOMs
- For crashes, however, `jextract` should still be used

Obtaining system core dumps

- In addition to being automatically triggered the first time an OutOfMemory error occurs, system core dumps can be triggered manually in several ways:
  - Using wsadmin
  - Using the system dump button in the administrative console
  - Using OS commands
- Run the following `wsadmin` commands:

```
jvm=AdminControl.completeObjectName(
   'type=JVM,<server_name>,*')
AdminControl.invoke(jvm,'generateSystemDump')
```

- This function is not supported when using a non-IBM Java virtual machine (JVM)
  - HP-UX and Solaris operating systems

Java heap dumps

- A heap dump contains all the reachable objects that are on the Java heap
  - Those objects that the Java application uses
- Heap dumps are created by default for OutOfMemory exceptions
  - By default, heap dumps are in binary (PHD) format
  - Can also be generated in test format
- Used for analyzing the actual contents of the Java heap
  - Shows the objects that are using the heap memory
  - Shows the objects that are using large amounts of memory on the Java heap, and what is preventing the Garbage Collector from collecting them
  - Useful for debugging memory leaks in Java applications

Heap dumps versus system core dumps

- The Memory Analyzer tool accepts a full system core file or a heap dump file
  - Which is better?
- A system core dump allows you to do everything a heap dump can, however
  - It is considerably larger
  - It usually takes a little longer to create
- The size and time to create the system core must be balanced against its advantages, such as:
  - Accurate GC root information
  - Native heap memory
  - Ability to see the values of primitive fields and strings (might be a privacy issue)
  - More views such as threads and their related stacks and frame local references
- Unless the write time or size make system cores unmanageable, consider the use of them instead of heap dumps

Obtaining Java heap dumps (1 of 3)

- Generated from a running JVM in these ways:
  - Explicit, manual generation, such as using `wsadmin` commands
  - JVM-triggered generation, by using dump agents to handle exceptions
  - Automated heap dump generation facility

- IBM Java heap dump for IBM JVMs
  - Windows, AIX, and Linux
  - JDK V5.0 and later: Use command-line argument `-Xdump:heap`
  - JDK V1.4.2 and earlier: Use environment variables

Obtaining Java heap dumps (2 of 3)

- Oracle JVM version 1.4.2_08 and earlier use IBM Heapagent
  - Not supported for production environments without assistance from IBM Support
- Oracle JVM version 1.4.2_09, 1.4.2_10, and 1.4.2_11 use JMapp
  - Command line tool that comes with the Oracle 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 that is used by all instances
  - Use the output operator to send output to a file
- Oracle JVM version 1.4.2_12 and higher allows Java heap dump
  - JVM command line option `-XX:+HeapDumpOnCtrlBreak`
  - `SIGQUIT signal (kill -3)` triggers the JVM to generate a heapdump
  - `Heapdump` can be parsed by using IBM Memory Analyzer

Obtaining Java heap dumps (3 of 3)

- Java memory dumps on HP-UX
  - HPROF profiler agent: Not useful with larger heap sizes, which are typically used with WebSphere
  - Jmap
- Work with IBM Support to find the best solution for your environment

Use wsadmin to trigger a heap dump

- From a command prompt, start the wsadmin shell by typing:

```
wsadmin –lang jython –user <user_name> -password <password>
```

- Run the following `wsadmin` commands:

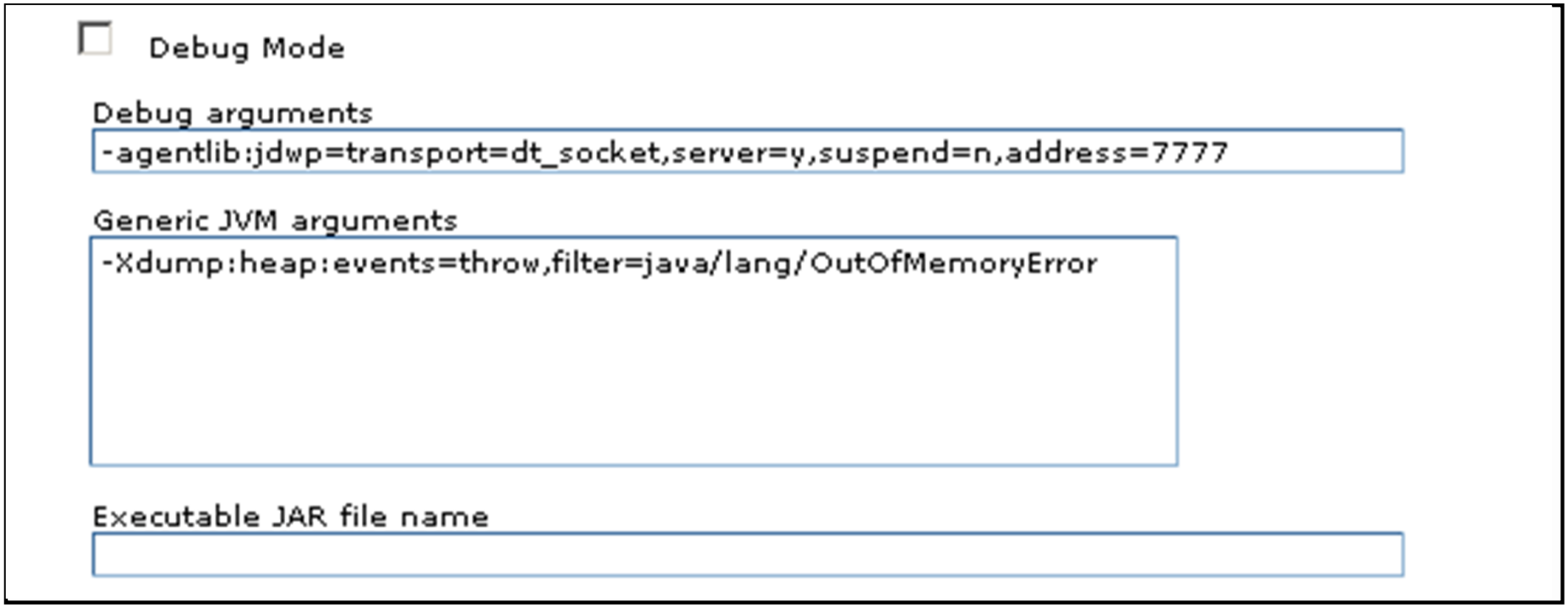
```
jvm=AdminControl.completeObjectName(
  'type=JVM,<server_name>,*')
AdminControl.invoke(jvm,'generateHeapDump')
```

- This function is not supported when using a non-IBM Java virtual machine (JVM)
  - HP-UX and Solaris operating systems

Java heap dumps by using dump agents

- To obtain a Java heap dump on a user signal, or if the application catches the resulting signal, add `-Xdump` command-line options to the Generic JVM arguments setting for a server
- For example, to create heap dumps, even when the application catches an `OutOfMemoryError`, use the argument

```
-Xdump:heap:events=throw,filter=java/lang/OutOfMemoryError
```



Generic arguments that use Xdump:heap

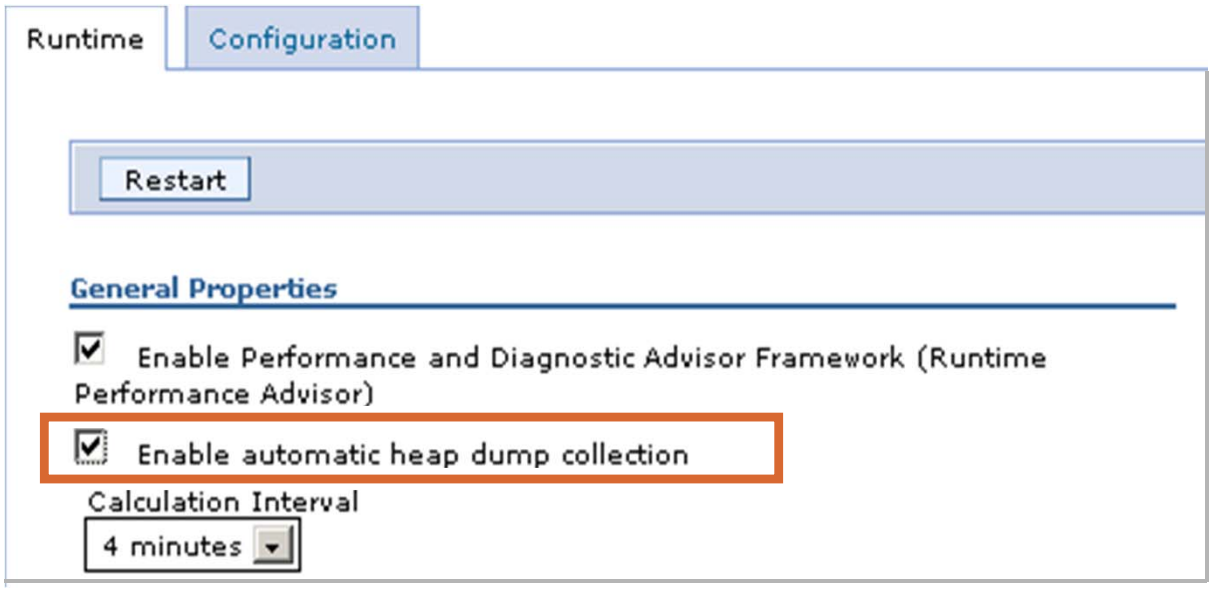
- `-Xdump:heap:events=...` enables heap dump creation for specified events
- Events can be:
  - `user` (the JVM receives the SIGQUIT or SIGBREAK signal from the OS)
  - `vmstart, vmstop` (the virtual machine is started or stopped)
  - `throw` (an exception is thrown; use filter on exception class)
  - `systhrow` (the JVM is about to throw a Java exception)
  - Several other events
- `-Xdump:heap:none` disable heap dump creation
- `-Xdump:heap, opts=PHD+CLASSIC` enables heap dump creation and creates the file in both binary and text format
- `-Xdump:heap:file=C:\dumps\heapdump.%Y%m%d.%H%M%S.%pid.%seq.phd` specifies the location and format of the heap dump file name

Generic arguments that use Xdump:system

- `-Xdump:system:events=...` enables system dump creation for specified events
- Events can be:
  - `gpf`
  - `abort`
  - `systhrow` (the JVM is about to throw a Java exception)
  - Several other events
- `-Xdump:system:none` disable system core dump creation
- `-Xdump:system` enables system core dump creation
- `-Xdump:system:file=/usr/dumps/core.%Y%m%d.%H%M%S.%pid.%seq.dmp` specifies the location and format of the system core dump file name

Automatic heap dump generation with IBM JDK V6

- Works only with IBM JVMs
- Works with the lightweight memory leak detection mechanism
- Heap dumps are generated when a memory leak is detected or when OutOfMemoryError is detected
  - False positives might occur with gencon and balanced GC policies
- Perform the following steps in the administrative console:
  1. Click ***server_name*** **> Performance and Diagnostic Advisor Configuration**
  2. Select the **Enable automatic heap dump collection** check box on the Runtime tab



Interpret verbose GC output

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 failure an isolated allocation failure caused by a request for a large object?

JVM messages that show dump events

- The IBM JDK will log messages to `native_stderr.log`

```
JVMDUMP039I Processing dump event "systhrow", detail
"java/lang/OutOfMemoryError" at 2012/03/15 15:51:49 - please wait.

JVMDUMP032I JVM requested System dump using
'C:\Sysdumps\core.20120315.155149.412.0002.dmp' in response to an event
JVMDUMP010I System dump written to
C:\Sysdumps\core.20120315.155149.412.0002.dmp
JVMDUMP032I JVM requested Heap dump using
'C:\Dumps\heapdump.20120315.155149.412.0003.phd' in response to an event
JVMDUMP010I Heap dump written to
C:\Dumps\heapdump.20120315.155149.412.0003.phd
JVMDUMP032I JVM requested Java dump using
'C:\Dumps\javacore.20120315.155149.412.0004.txt' in response to an event
JVMDUMP010I Java dump written to
C:\Dumps\javacore.20120315.155149.412.0004.txt
JVMDUMP032I JVM requested Snap dump using 'C:\Program
Files\IBM\WebSphere\AppServer\profiles\profile1\Snap.20120315.155149.412
.0005.trc' in response to an event

JVMDUMP013I Processed dump event "systhrow", detail
"java/lang/OutOfMemoryError".
```

Verbose GC output for optthruput policy

```
<af type="tenured" id="1" timestamp="Sun Mar 12 19:12:55 2006" intervalms="0.000">
   <minimum requested_bytes="16" />
   <time exclusiveaccessms="0.025" />
   <tenured freebytes="23592960" totalbytes="471859200" percent="5" >
      <soa freebytes="0" totalbytes="448266240" percent="0" />
      <loa freebytes="23592960" totalbytes="23592960" percent="100" />
   </tenured>
   <gc type="global" id="3" totalid="3" intervalms="11620.259">
      <refs_cleared soft="0" weak="72" phantom="0" />
      <finalization objectsqueued="9" />
      <timesms mark="74.734" sweep="7.611" compact="0.000" total="82.420" />
         <tenured freebytes="409273392" totalbytes="471859200" percent="86" >
         <soa freebytes="385680432" totalbytes="448266240" percent="86" />
      <loa freebytes="23592960" totalbytes="23592960" percent="100" />
      </tenured>
   </gc>
   <tenured freebytes="409272720" totalbytes="471859200" percent="86" >
      <soa freebytes="385679760" totalbytes="448266240" percent="86" />
      <loa freebytes="23592960" totalbytes="23592960" percent="100" />
   </tenured>
   <time totalms="83.227" />
</af>
```

Allocation request details: id, when and time between AF

Heap occupancy details before GC

Heap occupancy details after GC

Heap occupancy details after allocation

Java 6.26 GC output: optthruput policy (1 of 3)

```
<exclusive-start id="44" timestamp="2012-02-24T10:09:28.704" intervalms="2859.607">
 <response-info timems="0.017" idlems="0.017" threads="0" lastid="00149A00"
  lastname="main" />
</exclusive-start>

<af-start id="45" totalBytesRequested="10016" tsT:09.704" intervalms="2859.783" />
<cycle-start id="46" type="global" contextid="0" tsT:09.704" intervalms="2859.852" />
<gc-start id="47" type="global" contextid="46" timestamp="2012-02-24T10:09.720">
  <mem-info id="48" free="2669000" total="52428800" percent="5">
    <mem type="tenure" free="2669000" total="52428800" percent="5">
      <mem type="soa" free="47560" total="49807360" percent="0" />
      <mem type="loa" free="2621440" total="2621440" percent="100" /> </mem>
  </mem-info>
</gc-start>
<allocation-stats totalBytes="27037240" >
  <allocated-bytes non-tlh="9534880" tlh="17502360" />
  <largest-consumer threadName="main" threadId="00149A00" bytes="26956888" />
</allocation-stats>
```

Request that caused allocation failure

Small object area (soa) has 0% free

Java 6.26 GC output: optthruput policy (2 of 3)

```
<gc-op id="49" type="mark" timems="63.254" contextid="46" timestamp="2012-02-
  24T10:09:28.767">

  <trace-info objectcount="325795" scancount="260990" scanbytes="8497304" />

  <finalization candidates="659" enqueued="78" />

  <references type="soft" candidates="3271" cleared="0" enqueued="0"
  dynamicThreshold="18" maxThreshold="32" />

  <references type="weak" candidates="12682" cleared="0" enqueued="0" />

  <references type="phantom" candidates="1" cleared="0" enqueued="0" />

  <stringconstants candidates="21495" cleared="100"  />

</gc-op>

<gc-op id="50" type="classunload" timems="0.011" contextid="46" timestamp="2012-02-
  24T10:09:28.767">

  <classunload-info classloadercandidates="85" classloadersunloaded="0"
  classesunloaded="0" quiescems="0.000" setupms="0.007" scanms="0.002"
  postms="0.002" />

</gc-op>
```

GC operation = Mark phase

GC operation = Classunload phase

Java 6.26 GC output: optthruput policy (3 of 3)

```
<gc-op id="51" type="sweep" timems="2.933" contextid="46" timestamp="2012-02-
  24T10:09:28.767" />
<gc-end id="52" type="global" contextid="46" durationms="68.258" timestamp="2012-02-
  24T10:09:28.767">
  <mem-info id="53" free="26769528" total="52428800" percent="51">
    <mem type="tenure" free="26769528" total="52428800" percent="51">
      <mem type="soa" free="24148088" total="49807360" percent="48" />
      <mem type="loa" free="2621440" total="2621440" percent="100" />
    </mem>
    <pending-finalizers system="77" default="1" reference="0" classloader="0" />
  </mem-info>
</gc-end>
<cycle-end id="54" type="global" contextid="46" timestamp="2012-02-24T10:09:28.767" />
<allocation-satisfied id="55" threadId="00149A00" bytesRequested="10016" />
<af-end id="56" timestamp="2012-02-24T10:09:28.767" />
<exclusive-end id="57" timestamp="2012-02-24T10:09:767" durationms="74.224" />
```

GC operation = sweep phase

48% of soa recovered

Memory request was satisfied

Verbose GC output for gencon policy

```
<af type="nursery" id="35" timestamp="Thu Aug 11 21:47:11 2005"
   intervalms="10730.361">
     <minimum requested_bytes="144" />
     <time exclusiveaccessms="1.193" />
     <nursery freebytes="0" totalbytes="1226833920" percent="0" />
     <tenured freebytes="68687704" totalbytes="209715200" percent="32" >
          <soa freebytes="58201944" totalbytes="199229440" percent="29" />
          <loa freebytes="10485760" totalbytes="10485760" percent="100" />
      </tenured>
      <gc type="scavenger" id="35" totalid="35" intervalms="10731.054">
           <flipped objectcount="1059594" bytes="56898904" />
           <tenured objectcount="12580" bytes="677620" />
           <refs_cleared soft="0" weak="691" phantom="39" />
           <finalization objectsqueued="1216" />
           <scavenger tiltratio="90" />
           <nursery freebytes="1167543760" totalbytes="1226833920" percent="95"
   tenureage="14" />
           <tenured freebytes="67508056" totalbytes="209715200" percent="32" >
                <soa freebytes="57022296" totalbytes="199229440" percent="28" />
                <loa freebytes="10485760" totalbytes="10485760" percent="100" />
           </tenured>
           <time totalms="368.309" />
       </gc>
       <nursery freebytes="1167541712" totalbytes="1226833920" percent="95" />
       <tenured freebytes="67508056" totalbytes="209715200" percent="32" >
              <soa freebytes="57022296" totalbytes="199229440" percent="28" />
             <loa freebytes="10485760" totalbytes="10485760" percent="100" />
        </tenured>
        <time totalms="377.634" />
</af>
```

Allocation request details: id, when and time between AF

Heap occupancy details before GC

Details about the scavenge

Heap occupancy details after GC

Java 6.26 GC output: gencon policy (1 of 3)

```
<exclusive-start id="26" timestamp="2012-02-23T20:53:38.440"
  intervalms="251.038">
  <response-info timems="0.010" idlems="0.010" threads="0"
  lastid="0014A400" lastname="main" />
</exclusive-start>

<af-start id="27" totalBytesRequested="25424" timestamp="2012-02-
  23T20:53:38.440" intervalms="250.537" />
<cycle-start id="28" type="scavenge" contextid="0" timestamp="2012-02-
  23T20:53:38.440" intervalms="250.497" />
<gc-start id="29" type="scavenge" contextid="28" timestamp="2012-02-
  23T20:53:38.440">

  <mem-info id="30" free="38928232" total="45875200" percent="84">
    <mem type="nursery" free="0" total="6553600" percent="0" />
    <mem type="tenure" free="38928232" total="39321600" percent="98">
      <mem type="soa" free="36962152" total="37355520" percent="98" />
      <mem type="loa" free="1966080" total="1966080" percent="100" />
    </mem>
    <remembered-set count="5703" />
  </mem-info>
</gc-start>
```

Request that caused allocation failure

GC operation = scavenge

Nursery has 0 memory free before the scavenge

Java 6.26 GC output: gencon policy (2 of 3)

```
<allocation-stats totalBytes="4002592" >
  <allocated-bytes non-tlh="21376" tlh="3981216" />
  <largest-consumer threadName="main" threadId="0014A400" bytes="4002592" />
</allocation-stats>
<gc-op id="31" type="scavenge" timems="8.391" contextid="28"
  timestamp="2012-02-23T20:53:38.440">
  <scavenger-info tenureage="9" tiltratio="50" />
  <memory-copied type="nursery" objects="41652" bytes="2148492"
  bytesdiscarded="392" />
  <finalization candidates="118" enqueued="0" />
  <references type="soft" candidates="36" cleared="0" enqueued="0"
  dynamicThreshold="32" maxThreshold="32" />
  <references type="weak" candidates="21" cleared="0" enqueued="0" />
  <references type="phantom" candidates="1" cleared="0" enqueued="0" />
</gc-op>
<gc-end id="32" type="scavenge" contextid="28" durationms="9.100"
  timestamp="2012-02-23T20:53:38.440">
```

GC op scavenge begins

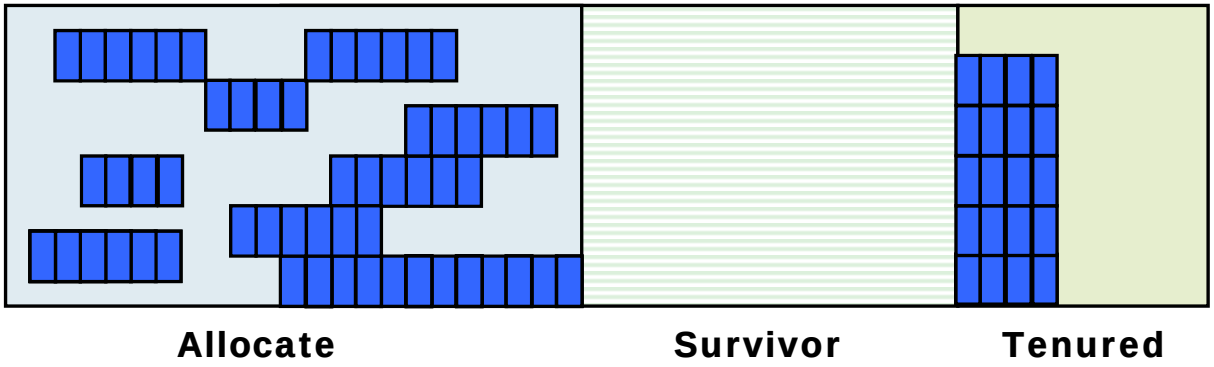
Java 6.26 GC output: gencon policy (3 of 3)

```
<mem-info id="33" free="43171168" total="45875200" percent="94">
    <mem type="nursery" free="4242936" total="6553600" percent="64" />
    <mem type="tenure" free="38928232" total="39321600" percent="98">
      <mem type="soa" free="36962152" total="37355520" percent="98" />
      <mem type="loa" free="1966080" total="1966080" percent="100" />
    </mem>
    <remembered-set count="2743" />
  </mem-info>
</gc-end>
<cycle-end id="34" type="scavenge" contextid="28" timestamp="2012-02-
  23T20:53:38.456" />
<allocation-satisfied id="35" threadId="0014A400" bytesRequested="25424" />
<af-end id="36" timestamp="2012-02-23T20:53:38.456" />
<exclusive-end id="37" timstamp="2012-2-23T20:53:38.456"
  durationms="13.062" />
```

Scavenge recovers 64% of the nursery

Memory request was satisfied

Verbose GC output for gencon failures



- Scavenging can fail due to a complete lack of space.
  - The scavenge is stopped and a global collect is attempted
  - Message in verbose GC log is: warning details="aborted collection"
- Nursery collections sometimes determine that they are insufficient.
  - Failure to complete a nursery collect
  - Insufficient resources
- Collect is promoted to a global GC.
  - Message in verbose GC log is: `percolating_collect reason="insufficient remaining tenure space"`

IBM Monitoring and Diagnostic Tools for Java – Garbage Collection and Memory Visualizer (GCMV)

GCMV: Verbose GC visualizer and analyzer

- Available through the IBM Support Assistant

- Reduces barrier to understanding verbose output
  - Visualize GC data to track trends and relationships
  - Analyze results and provide general feedback
  - Extend to use output that is related to application
  - Build plug-able analyzers for specific application needs

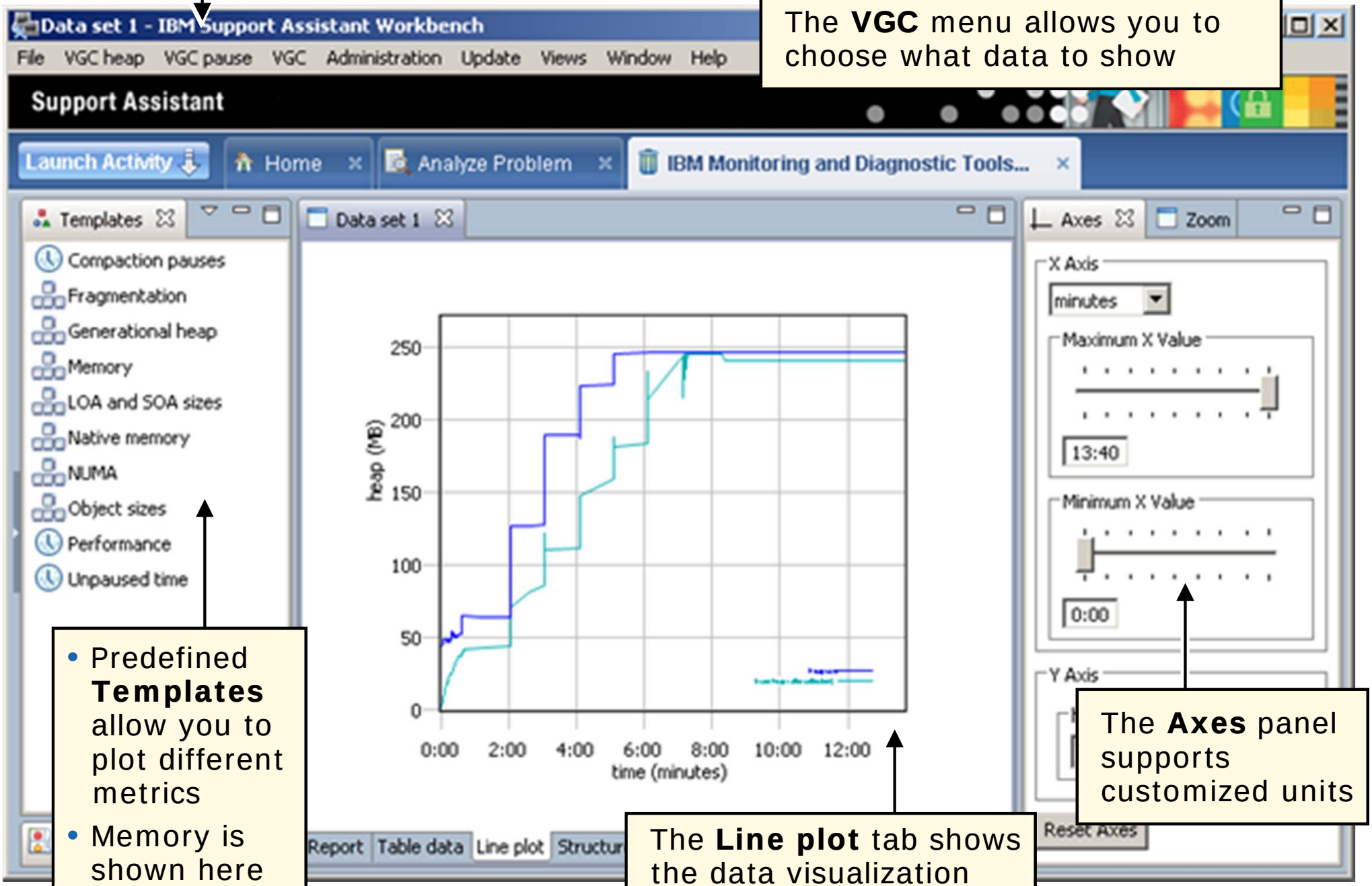
Garbage Collection and Memory Visualizer overview

- The Garbage Collection and Memory Visualizer (GCMV) is a visualizer for verbose garbage collection output
  - The tool parses and plots verbose GC output and garbage collection traces (`-Xtgc` output)
- Started from the IBM Support Assistant
- The GCMV provides:
  - Raw view of data
  - Line plots to visualize various GC data characteristics
  - Tabulated reports with heap occupancy recommendations
  - View of multiple data sets on a single set of axes
  - Ability to save data as an image (jpeg) or comma-separated file (csv)

GCMV usage scenarios

- Investigate performance problems
  - Long periods of pausing or not responding
- Evaluate your heap size
  - Check heap occupancy and adjust heap size if needed
- Garbage collection policy tuning
  - Examine GC characteristics, compare different policies
- Look for memory growth
  - Heap consumption slowly increasing over time
  - Evaluate the general health of an application

Plotting verbose GC data with the GCMV



The **VGC** menu allows you to choose what data to show

- Predefined **Templates** allow you to plot different metrics
- Memory is shown here

The **Line plot** tab shows the data visualization

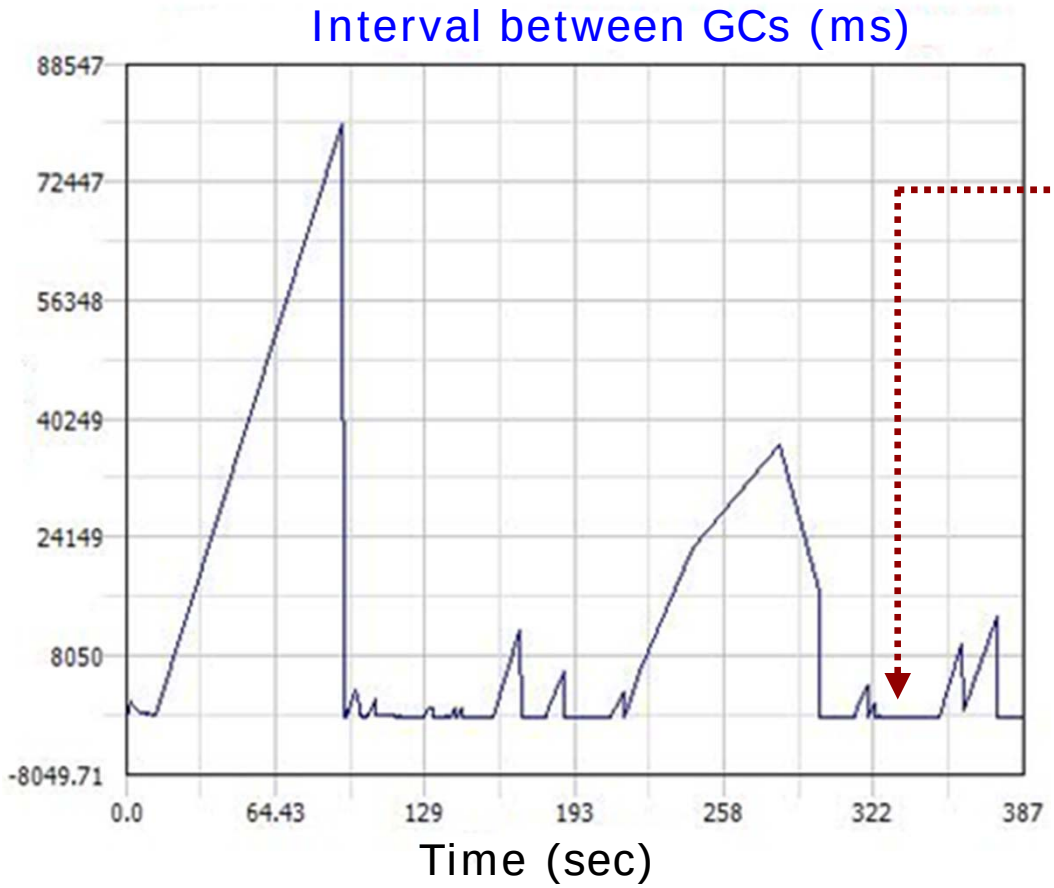
The **Axes** panel supports customized units

Types of graphs

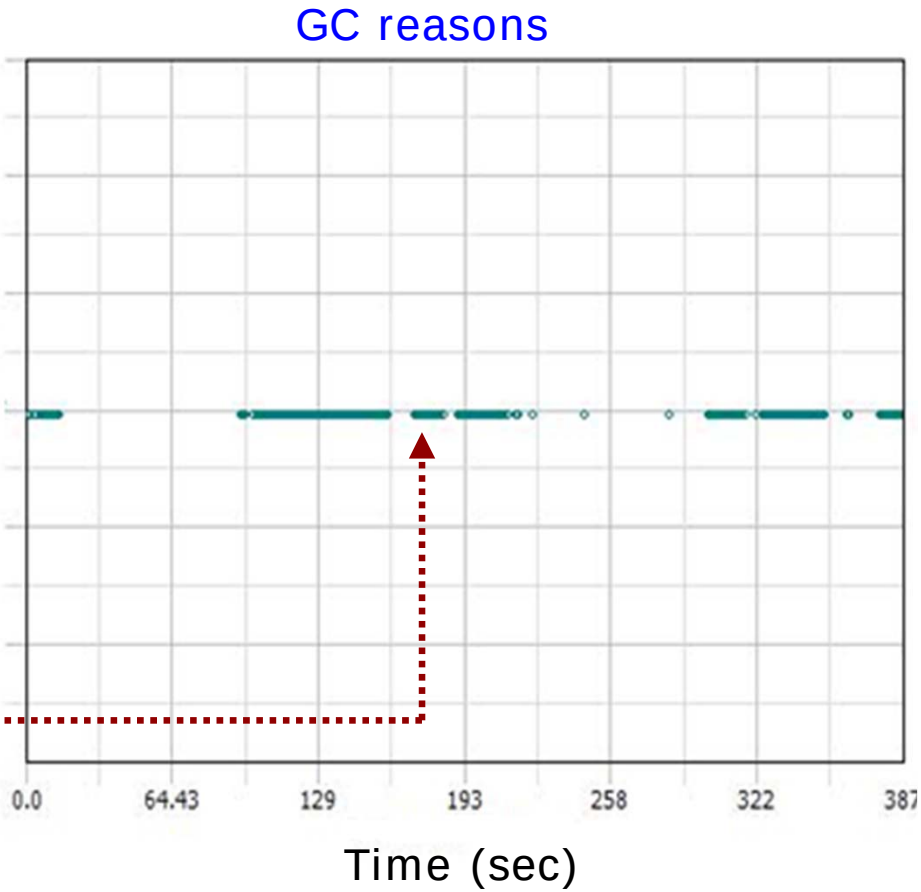
The GCMV has built-in support for over 40 different types of graphs

- These graphs are configured in the VGC Data, VGC Pause Data, and VGC Heap Data menus
- Options vary depending on the current data set and the parsers and post-processors that are enabled
- Some of the VGC graph types are:
  - Used total heap
  - Pause times (mark-sweep-compact collections)
  - Pause times (totals, including exclusive access)
  - Compact times
  - Weak references that are cleared
  - Soft references that are cleared
  - Free tenured heap (after collection)
  - Tenured heap size
  - Tenure age
  - Free LOA (after collection)
  - Free SOA (after collection)
  - Total LOA
  - Total SOA

Garbage collection trigger graphs



- This graph shows intervals between garbage collection cycles
- Notice that it shrinks to near 0 ms for extended periods (horizontal portions of the graph)



- Each dot in this graph represents a garbage collection cycle
- All of these cycles ran for reason "af" – allocation failure
- Notice that the dot concentration lines up with the trigger graph

Heap usage and occupancy recommendation (1 of 2)

The summary report shows that the mean heap occupancy is 50% and that the application is using large objects

- Your application appears to be leaking memory. This is indicated by the used heap increasing at a greater rate than the application workload (measured by the amount of data freed).
- The Java Heap has been exhausted, leading to an out of memory error. You should consider increasing the Java Heap size using -Xmx if space allows. You can analyse the usage of the Java Heap for a memory leak by using the ISA Tool Add-on, IBM Monitoring and Diagnostic Tools for Java - Memory Analyzer.
- 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 5120008 bytes.
- 8% of allocation failures were caused by large allocations. Consider using balanced mode.
- 430 global garbage collects took on average 2,570% longer than the average nursery collect. If you consider this abnormally high and unacceptable, consider using balanced mode.
- i The mean occupancy in the nursery is 50% which is close to optimal.

Heap usage and occupancy recommendation (2 of 2)

- Based on the usage summary that is shown, the tool suggests:
  - Increasing the nursery size
  - Increasing the tenure age

Summary

Concurrent collection count	7
Forced collection count	4
GC Mode	gencon
Global collections - Mean garbage collection pause (ms)	293
Global collections - Mean interval between collections (minutes)	1.63
Global collections - Number of collections	437
Global collections - Total amount tenured (MB)	101416
Largest memory request (bytes)	5120008
Number of collections triggered by allocation failure	1086
Nursery collections - Mean garbage collection pause (ms)	11.5
Nursery collections - Mean interval between collections (ms)	82267
Nursery collections - Number of collections	660
Nursery collections - Total amount flipped (MB)	1761
Nursery collections - Total amount tenured (MB)	662
Proportion of time spent in garbage collection pauses (%)	0.27
Proportion of time spent unpaused (%)	99.73
Rate of garbage collection (MB/minutes)	10.1

Comparing verbose GC between different JVMs (1 of 2)

- Multiple verbose GC logs can be imported into GCMV and compared

Variant	OOM.log	PBW.log
Concurrent collection count	6	0
Forced collection count	0	0
GC Mode	gencon	gencon
Global collections - Mean garbage collection pause (ms)	313	166
Global collections - Mean interval between collections (minutes)	0.16	0.91
Global collections - Number of collections	47	1
Global collections - Total amount tenured (MB)	9515	46.4
Largest memory request (bytes)	5120008	392416
Number of collections triggered by allocation failure	75	48
Nursery collections - Mean garbage collection pause (ms)	16.1	11.9
Nursery collections - Mean interval between collections (ms)	13670717	9878838
Nursery collections - Number of collections	34	47
Nursery collections - Total amount flipped (MB)	338	96.6
Nursery collections - Total amount tenured (MB)	159	7.09
Proportion of time spent in garbage collection pauses (%)	3.31	0.28
Proportion of time spent unpaused (%)	96.69	99.72
Rate of garbage collection (MB/minutes)	145	224

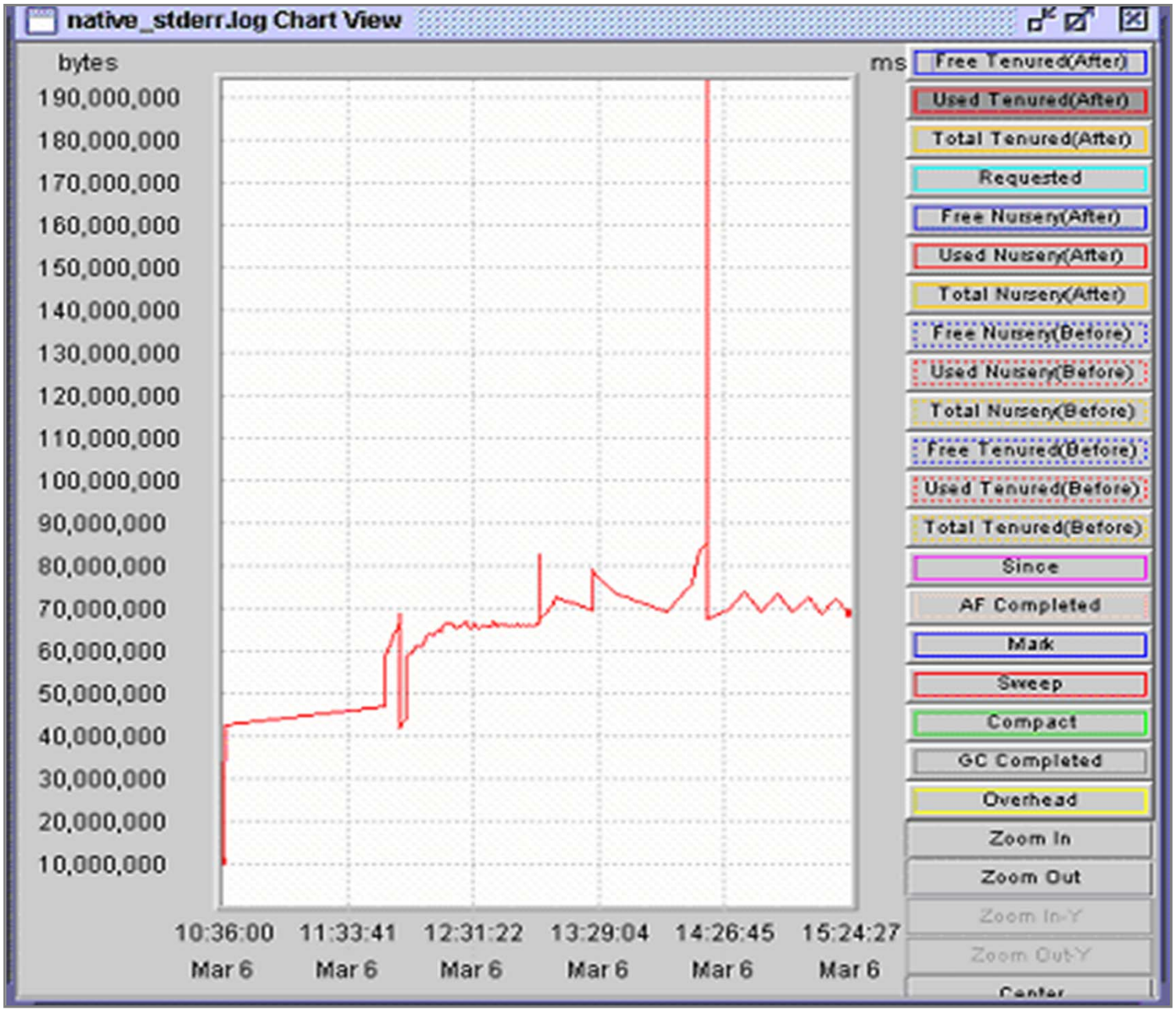
Comparing verbose GC between different JVMs (2 of 2)

Variant	Tuning recommendation
OOM.log	The Java Heap has been exhausted, leading to an out of memory error. You should consider increasing the Java Heap size using -Xmx if space allows. You can analyse the usage of the Java Heap for a memory leak by using the ISA Tool Add-on, IBM Monitoring and Diagnostic Tools for Java - Memory Analyzer. 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 5120008 bytes. 46 global garbage collects took on average 1,969% longer than the average nursery collect. If you consider this abnormally high and unacceptable, consider using balanced mode. i The mean occupancy in the nursery is 52% which is close to optimal.
PBW.log	1 global garbage collects took on average 1,387% longer than the average nursery collect. If you consider this abnormally high and unacceptable, consider using balanced mode. The mean occupancy in the nursery is 10%. This is low, so the gencon policy is probably an optimal policy for this workload. i The mean occupancy in the tenured area is 69% which is close to optimal.

IBM Pattern Modeling and Analysis Tool (PMAT)

- Predates GCMV but can also be useful when analyzing verbose GC
  - Available from the IBM Support Assistant
  - Works with IBM JVM 1.3 and up
- Provides the following features:
  - Tabulated and graphical views
  - Summary usage analysis
  - Analysis includes tuning recommendations
- Suggestion: GCMV is your first choice; use PMAT only if you have some particular reason to prefer it

PMAT: Chart view



Analyze Java heap dumps and system core dumps

OutOfMemory: Interpret heap dumps

Heapdump:

- Use tools to parse the object trees
  - Almost impossible to do manually
- Look for root objects that hold significant memory resources
  - Includes memory that is allocated for the root object themselves and all the memory that is required for all the objects in the reference tree
- Check reference tree for sudden drop in memory
  - Expand the tree; 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
  - A long chain of similarly sized objects also indicates a potential leak

Jmap:

- Contact support to parse output
- Manually inspect the output list

How to analyze a Java heap dump

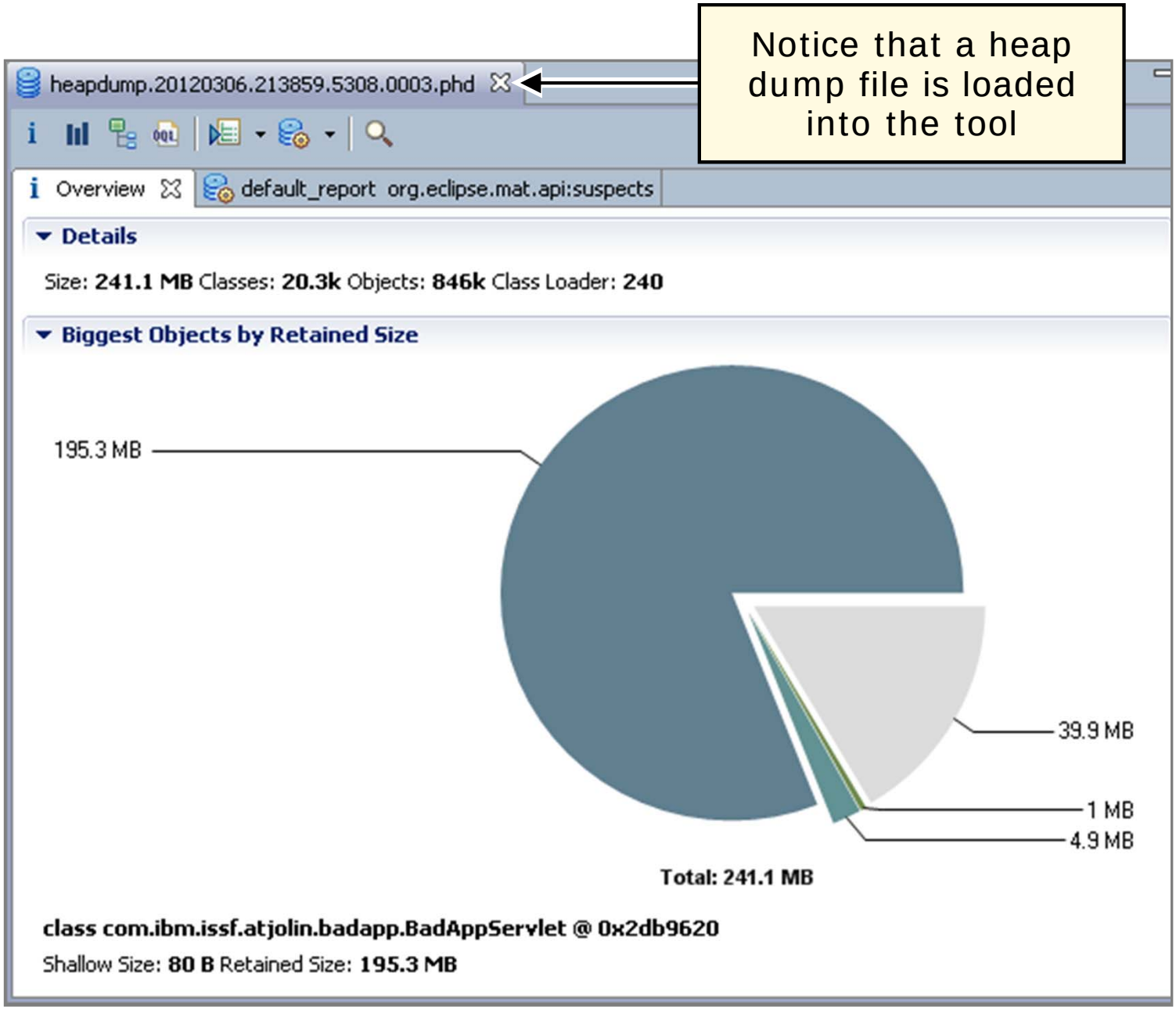
- Single dump analysis
  - Inspects a single heap dump file for container objects that have large reach size as compared to their largest child object
  - Most commonly used to analyze automatically generated heap dumps after an OutOfMemoryException
- Comparative analysis (rarely necessary)
  - Analyzes heap growth between two dumps that are taken over time
  - Identifies the objects with the largest size difference and their ownership relationships
- Analysis results
  - Summary of analysis results that show heap contents, size, and growth
  - Lists suspected data structures, data types, and packages that contribute to the growth in heap usage
  - Tabular views of all the objects and data types in the memory dump with filters and sorted columns

IBM Monitoring and Diagnostic Tools for Java – Memory Analyzer Version 1.1

- Java heap analysis tool that is based on the Eclipse Memory Analyzer (MAT)
  - Available from IBM Support Assistant
- Memory Analyzer extends Eclipse MAT version 1.1 using the DTFJ
  - Enables Java heap analysis by using system core dumps and IBM Portable Heap Dumps (PHD)
- By using Memory Analyzer, you can:
  - Diagnose and resolve memory leaks that involve the Java heap
  - Derive architectural understanding of your Java application through footprint analysis
  - Improve application performance by tuning memory footprint and optimizing Java collections and Java cache usage
  - Produce analysis plug-ins with capabilities specific to your application
- Analyzes heapdumps for leaks and usage patterns
  - Identifies data structures that are potential leaks
  - Shows memory usage by Java package
  - Identifies areas for improvement to memory usage

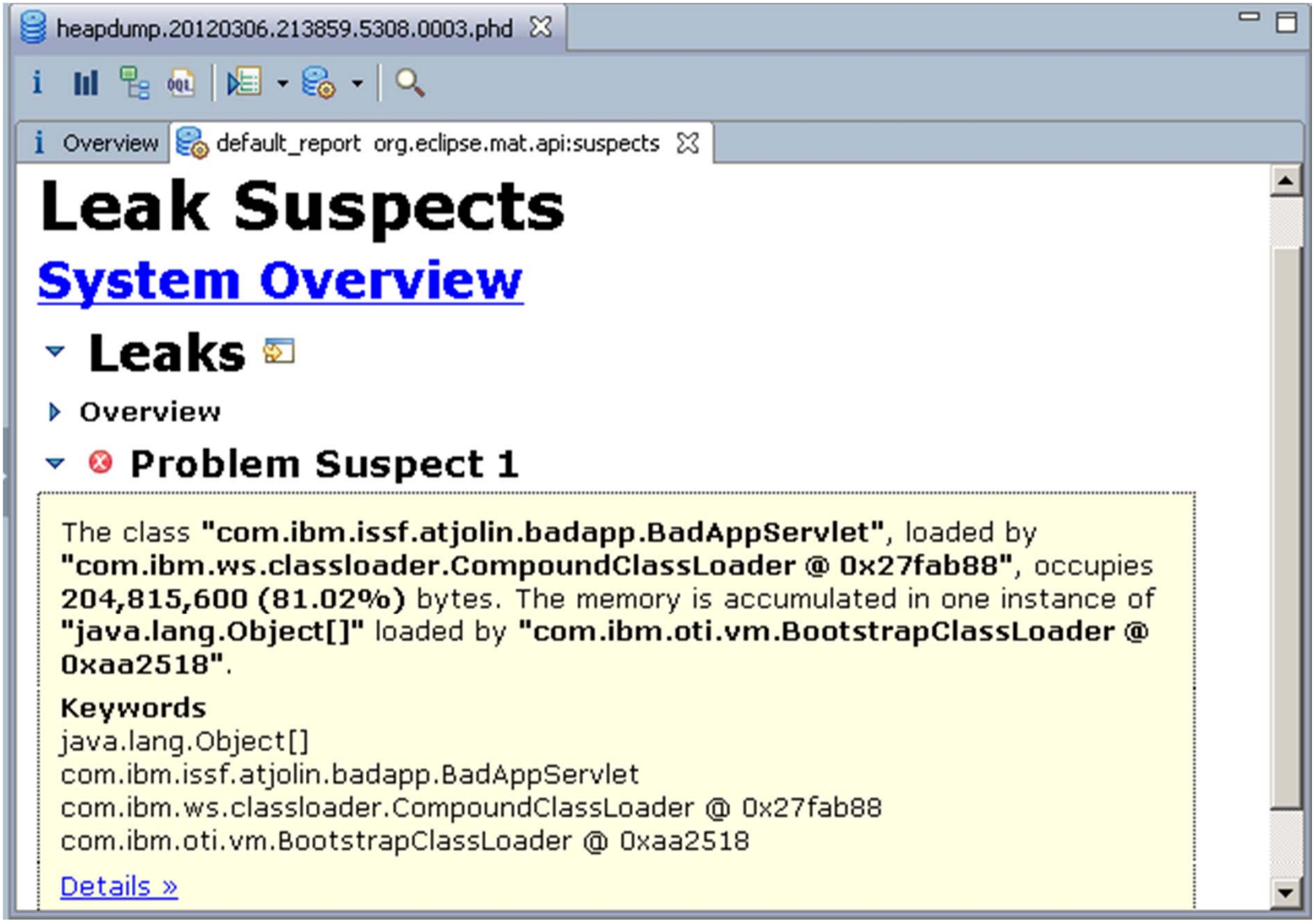
Memory Analyzer overview

- Pie chart of biggest objects and links to common analysis options
  - Histogram
  - Denominator tree
  - Top consumers
  - Duplicate classes



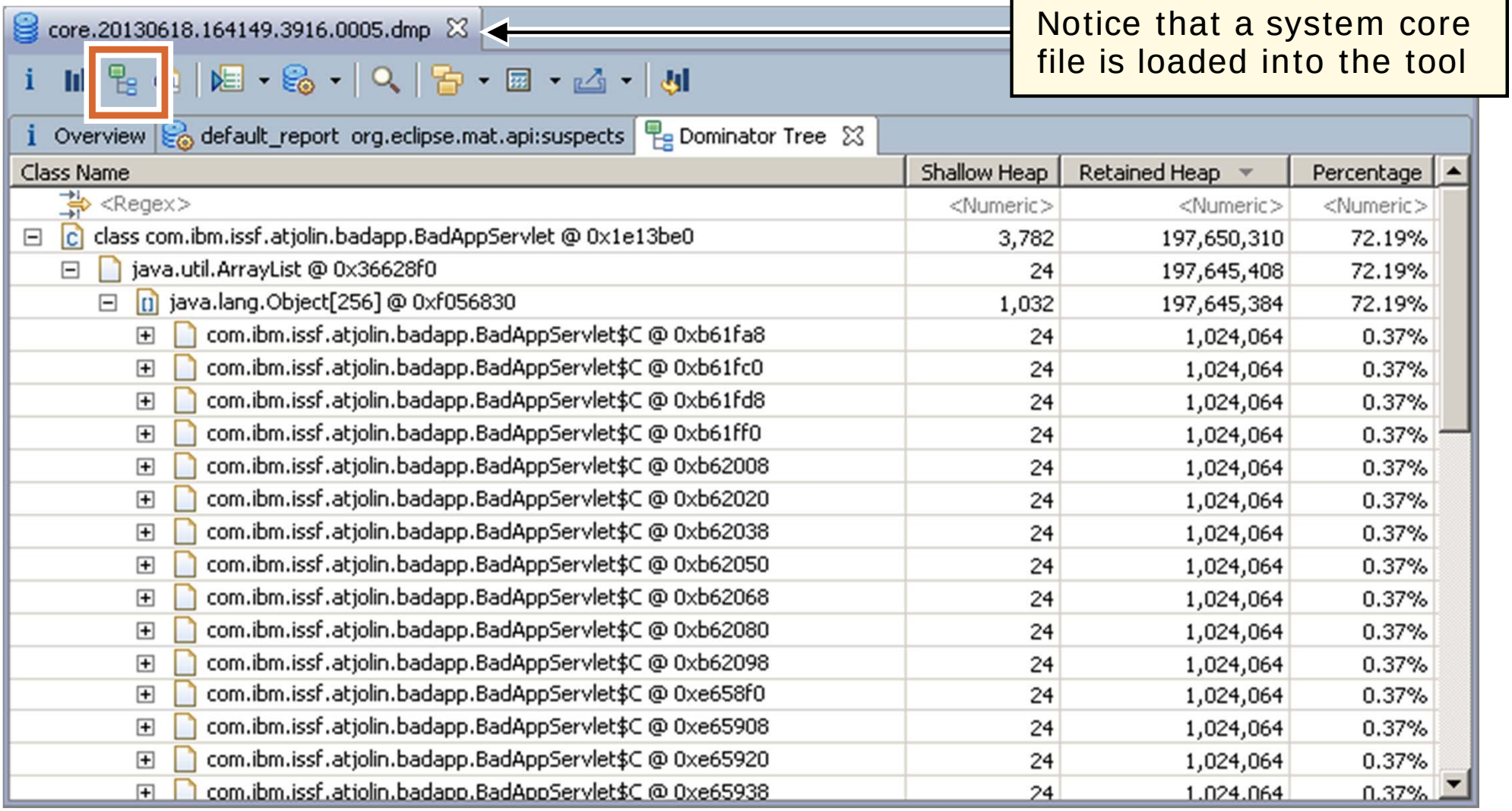
Memory Analyzer leak suspects: Default report

- One or more leak suspects are listed



Denominator tree view

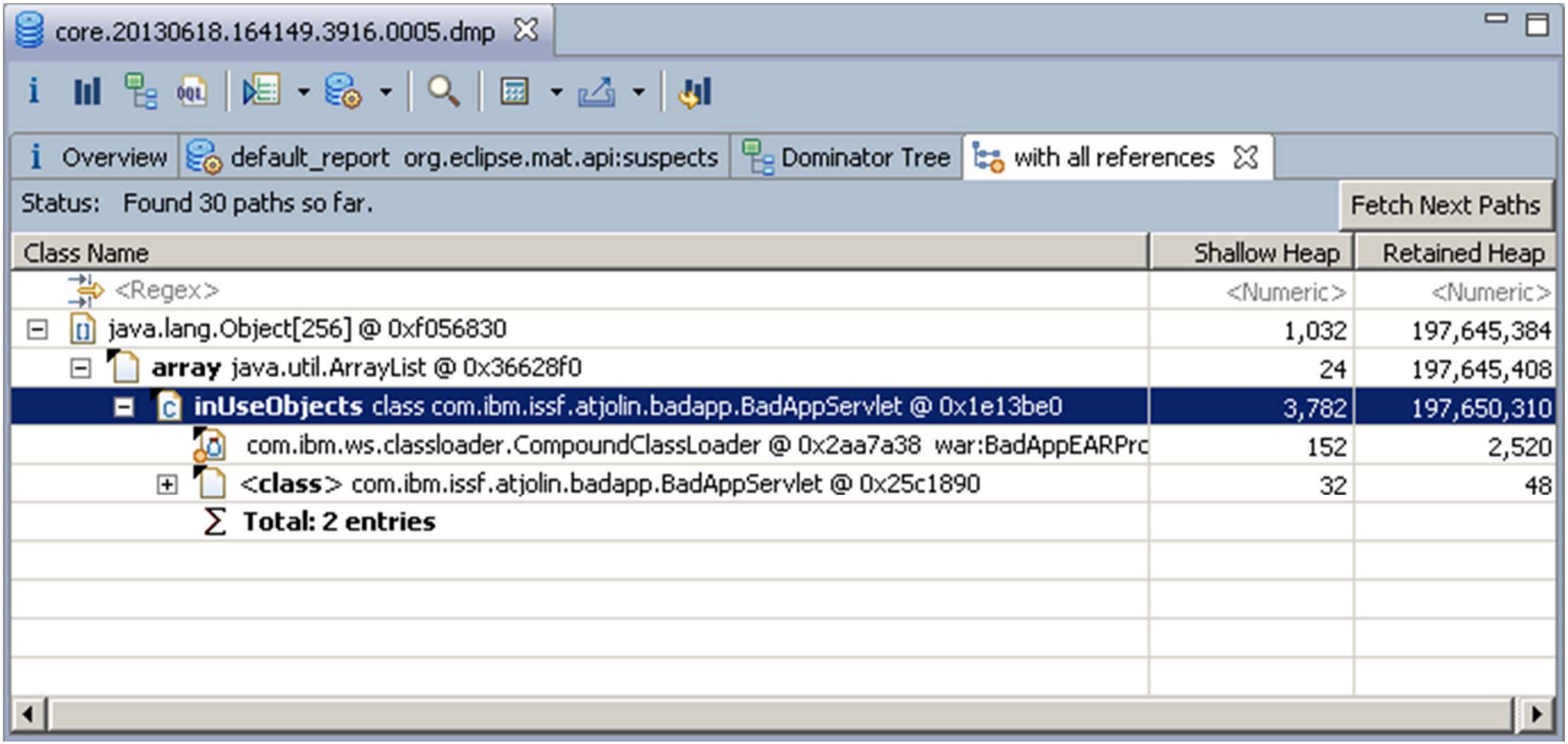
- MAT provides other views
  - The most useful for heap analysis is the Denominator Tree, which identifies the objects that hold the most heap



Class Name	Shallow Heap	Retained Heap	Percentage
<Regex>	<Numeric>	<Numeric>	<Numeric>
class com.ibm.issf.atjolin.badapp.BadAppServlet @ 0x1e13be0	3,782	197,650,310	72.19%
java.util.ArrayList @ 0x36628f0	24	197,645,408	72.19%
java.lang.Object[256] @ 0xf056830	1,032	197,645,384	72.19%
com.ibm.issf.atjolin.badapp.BadAppServlet$C @ 0xb61fa8	24	1,024,064	0.37%
com.ibm.issf.atjolin.badapp.BadAppServlet$C @ 0xb61fc0	24	1,024,064	0.37%
com.ibm.issf.atjolin.badapp.BadAppServlet$C @ 0xb61fd8	24	1,024,064	0.37%
com.ibm.issf.atjolin.badapp.BadAppServlet$C @ 0xb61ff0	24	1,024,064	0.37%
com.ibm.issf.atjolin.badapp.BadAppServlet$C @ 0xb62008	24	1,024,064	0.37%
com.ibm.issf.atjolin.badapp.BadAppServlet$C @ 0xb62020	24	1,024,064	0.37%
com.ibm.issf.atjolin.badapp.BadAppServlet$C @ 0xb62038	24	1,024,064	0.37%
com.ibm.issf.atjolin.badapp.BadAppServlet$C @ 0xb62050	24	1,024,064	0.37%
com.ibm.issf.atjolin.badapp.BadAppServlet$C @ 0xb62068	24	1,024,064	0.37%
com.ibm.issf.atjolin.badapp.BadAppServlet$C @ 0xb62080	24	1,024,064	0.37%
com.ibm.issf.atjolin.badapp.BadAppServlet$C @ 0xb62098	24	1,024,064	0.37%
com.ibm.issf.atjolin.badapp.BadAppServlet$C @ 0xe658f0	24	1,024,064	0.37%
com.ibm.issf.atjolin.badapp.BadAppServlet$C @ 0xe65908	24	1,024,064	0.37%
com.ibm.issf.atjolin.badapp.BadAppServlet$C @ 0xe65920	24	1,024,064	0.37%
com.ibm.issf.atjolin.badapp.BadAppServlet$C @ 0xe65938	24	1,024,064	0.37%

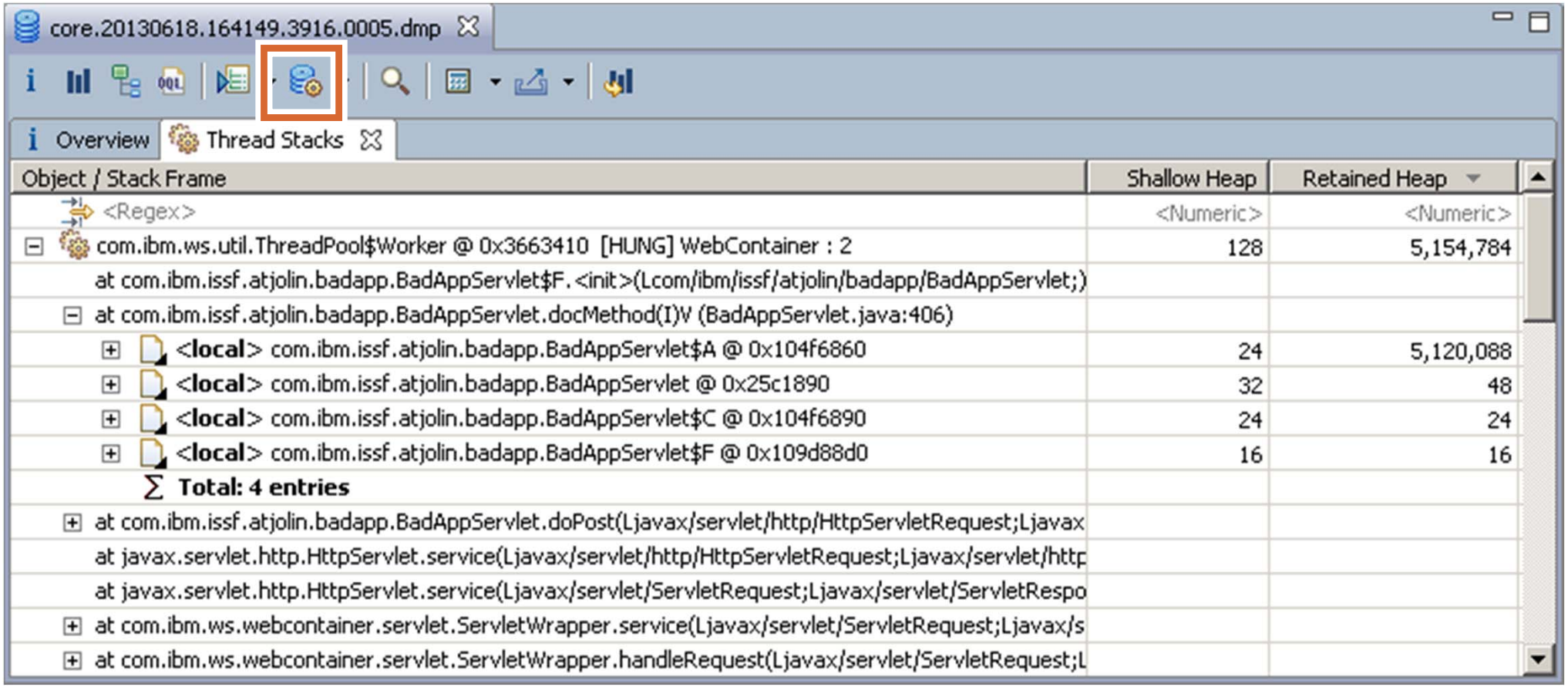
Path to GC roots

- After identifying the object that holds the most heap space alive, you can use the **path to GC roots** option to see what is keeping it alive:
  - Right-click the object to see the menu
  - You can ask to see all, or just the shortest path to the GC root



Thread stack view

- MAT also provides a Thread Stacks view that can be useful in understanding how a thread is influencing your heap usage
  - Accessed through the **Query browser > Java basics menu**

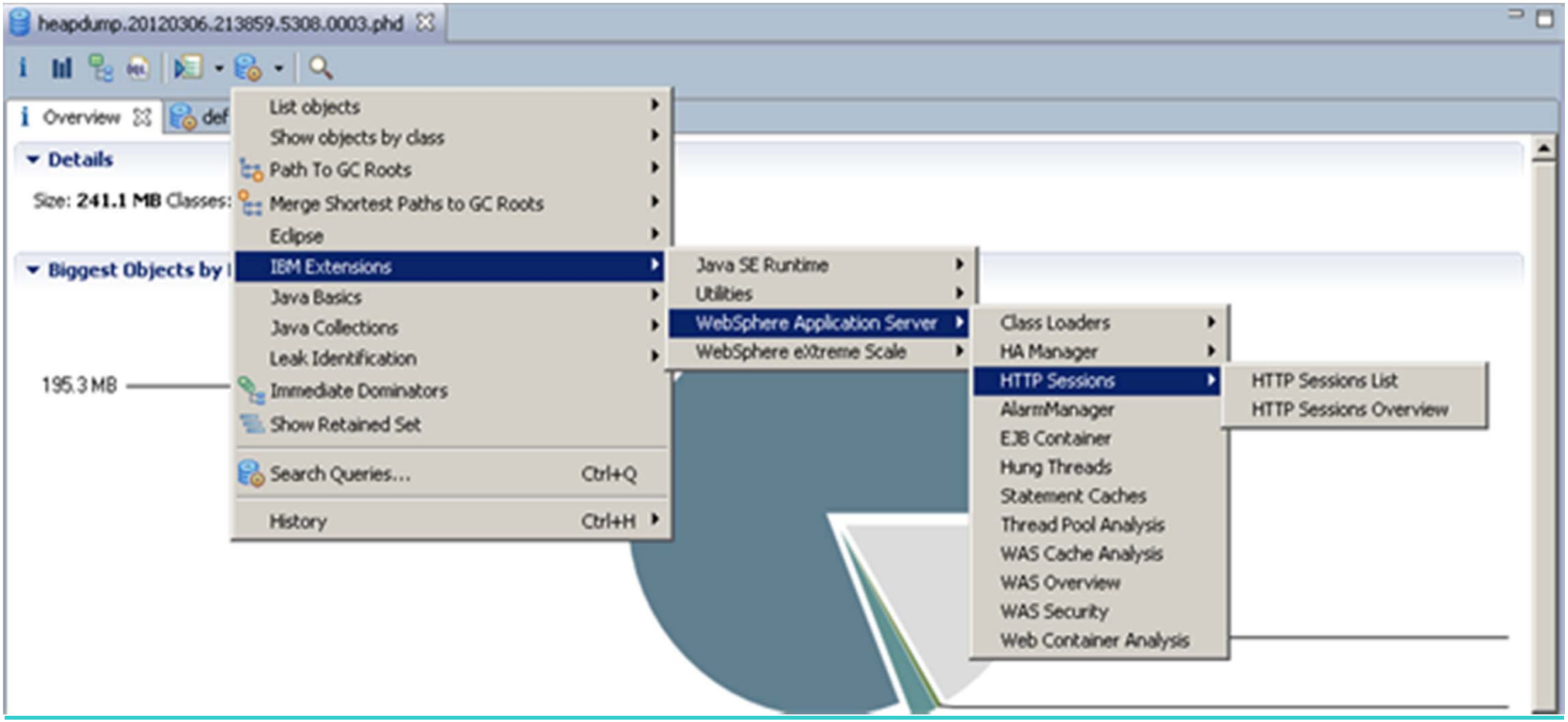


IBM Extensions for Memory Analyzer

- The IBM Extensions for Memory Analyzer offer both additional capabilities for debugging generic Java applications, and capabilities for debugging specific IBM software products
- Extensions are currently available for:
  - Java SE runtime
  - WebSphere Application Server
  - CICS Transaction Gateway
- By using the IBM Extensions for Memory Analyzer you can:
  - Visualize the state of both your code and that of the IBM software products that are used in the application
  - Confirm the configuration of the IBM products by looking at the loaded configuration, and the state of the configured components: for example, cache and thread pool configurations
  - Inspect the size and contents of IBM product components, for example cache or HTTP session contents

IBM Extensions for Memory Analyzer

- The IBM Extensions can be accessed by clicking the “Disks” icon



HeapAnalyzer

- Analyzes heapdump for leaks and builds object hierarchy
  - Detection method is suited for large collections of objects
  - Simplifies hierarchy to one parent that might cause artifacts
- Available from IBM Support Assistant
- Available on alphaWorks:
  - `http://www.alphaworks.ibm.com/tech/heapanalyzer`
- Suited for quick and simple hierarchy inspection
- Some features include:
  - List of Java heap leak suspects
  - List of gaps among allocated objects/classes/arrays
  - Java objects, classes, and arrays search engine
  - List of objects, classes, and arrays by type, object, size, size of child, and other parameters
  - List of available heap spaces by size
  - Tree of Java heap dump
- Suggestion: Memory Analyzer is your first choice, use HeapAnalyzer only if you have some particular reason to prefer it

HeapAnalyzer summary

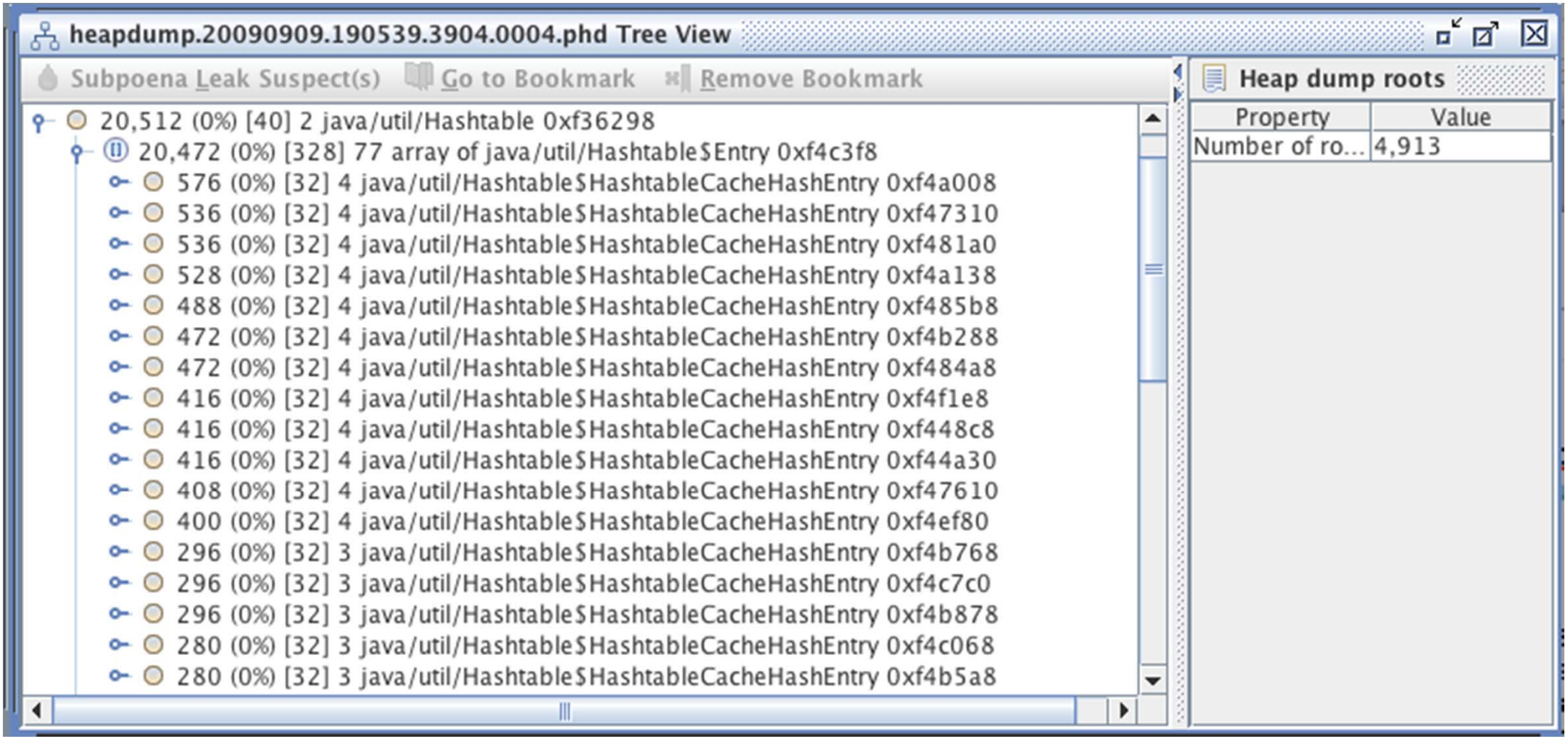
- Analysis of heap usage and general information

Analysis of heapdump.20090909.190539.3904.0004.phd

Property	Value
Heap dump file name	/Users/wareiche/Documents/WebSphere 7 PD Course/heapdump...
Java Version	J2RE 5.0 IBM J9 2.3 Windows XP x86-32 build 20080809_2189...
Number of Classes	12,799
Number of Objects	280,501
Number of ObjectArrays	43,025
Number of PrimitiveArrays	74,222
Total Number of Instances	410,547
Total Number of References	1,239,033
Number of roots	4,913
Number of types	12,799
Heap range	0x93e60 to 0x187938d0
Java heap usage	20,525,272 bytes
Dark Matter	2,526,016 bytes (12.306858 %)
WARNING!!	Java garbage collection trace may report 23,051,288 bytes are u...
	Actual Java heap usage is 20,525,272 bytes due to excessive dar...

HeapAnalyzer tree

- Tree-based exploration of simplified object hierarchy can help identify leak suspects



Unit summary

Having completed this unit, you should be able to:

- Define out-of-memory conditions
- Use monitoring tools to detect out-of-memory conditions
- Obtain and interpret a verbose GC log by using GCMV
- Obtain and analyze heap dumps and system core dumps
- Describe tools for analyzing out-of-memory problems

Checkpoint questions

1. List three causes of Java heap OutOfMemory error.

2. What JVM-related statistics can be monitored with the PMI Basic setting?

3. What diagnostic data does verboseGC output provide for identifying an OutOfMemory condition?

Checkpoint answers

1. The following are three causes of Java heap OutOfMemory error:
   - Insufficient maximum heap size for user load
   - Memory leak in the Java code
   - Memory leak in native code

2. The following statistics can be monitored with the PMI Basic setting:
   - JVM UpTime
   - JVM HeapSize
   - JVM UsedMemory

3. The following diagnostic data is provided:
   - Size of object that is allocated
   - Available memory on the Java heap
   - JVM response to an allocation request

Exercise 6

Troubleshooting an out-of-memory condition

Exercise objectives

After completing this exercise, you should be able to:

- Use Tivoli Performance Viewer to detect out-of-memory conditions
- Configure the lightweight memory leak detection mechanism
- Obtain verbose GC data and interpret it by using the Garbage Collection and Memory Visualizer tool (GCMV)
- Trigger a heap dump and analyze it by using the Memory Analyzer tool
- Examine the diagnostic data artifacts of an application that has a memory leak