# Checklist for OutOfMemoryError

## Types of OutOfMemoryErrors

* Java.lang.OutOfMemoryError: Java heap space
  + The heap has been consumed by objects, and free space cannot be reclaimed through garbage collection. Either the heap size needs to be increased or there is a memory leak. Heap dumps will help to identify what’s consuming the most space in the heap. Note that in some cases, the heap consumption is simply higher than the customer might have expected, and additional JVMs might be required to support the load.
* Java.lang.OutOfMemoryError: PermGen space
  + The PermGen space is memory allocated outside of the heap for class files. Adding the –XX:MaxPermSize=512m flag to the java startup options will allocate 512m to the PermGen space. Be aware that the combination of the Java heap and the PermGen space must not exceed OS limits (2GB on 32bit Windows).
* Java.lang.OutOfMemoryError: Unable to create new native thread
  + This can be caused by a number of issues, but in most cases it is caused by hitting some OS limit, such as maximum number of open files. Check the output of “ulimit –n” on Unix, and increase the value if needed.
  + It can also be fixed in some cases by reducing the stack size using the –Xss128k setting, or by reducing the maximum heap size, leaving more room for native threads.

## What to Include with the JIRA Request

In the case of crash due to heap exhaustion, please collect the following data (in addition to the standard performance background information):

### Basic Information

1. What operating system are they running?
2. What application server are they running?
3. What application and version are they running?
4. What vendor and version of Java are they running?
5. Does this happen in production or in test?
6. Can they reproduce the problem in their test environment?

### Please Collect

1. Snapshots of Memory heap dumps ( 3-4 heap dumps, taken in 5 minute intervals)
2. Thread dumps (3-4 thread dumps, taken in 5 minute intervals)
3. All of the configured JVM options ( such as –Xmx and - XX:MaxPermSize)
4. GC (Garbage Collection) logs
5. Application server logs
6. Application logs (log4j)
7. WDK/application logs
8. If a session leak is suspected, enable DFC resource diagnostic from dfc.properties and get the log file
9. The output of “ulimit –a” for the user running the application server (UNIX only)
10. If the issue occurs in multi-user testing with LoadRunner, attach the scripts for all tests to the task, as there is often an issue with the opendocviewcontainer request not being parameterized properly which causes forms to build up and eventually consume all available heap space.

If the issue is due to the PermGen space, try increasing the amount of memory allocated by increasing the MaxPermSize JVM startup parameter.

If the issue is due to native thread space, check the application server logs to see if there are more details about why it is happening. If you see “too many open files”, then use ulimit on UNIX platforms to check the settings for the maximum number of open files. Ask the customer to try increasing the value to a larger number. If it is not due to open files, try setting a lower stack size (-Xss) or reduce the heap size.

## How to Collect Heap Dumps

There are many ways to collect heap dumps, depending on the Java and application server versions.

1. Add the “-XX:+HeapDumpOnOutOfMemoryError” flag to the JVM startup options.

This flag will instruct the JVM to automatically create heap dumps when an OOM error occurs. The dumps will will generally be at least as large as the configured heap size.

1. Add the “-XX:+HeapDumpOnCtrlBreak” flag to the JVM startup options.

This flag will instruct the JVM to dump the heap to a file when Ctrl-Break or “kill -3 <PID>” is sent to the Java process. This is helpful when you want to generate heap dumps on command. Please note that in Windows, the JVM must be running in a console window, not as a service.

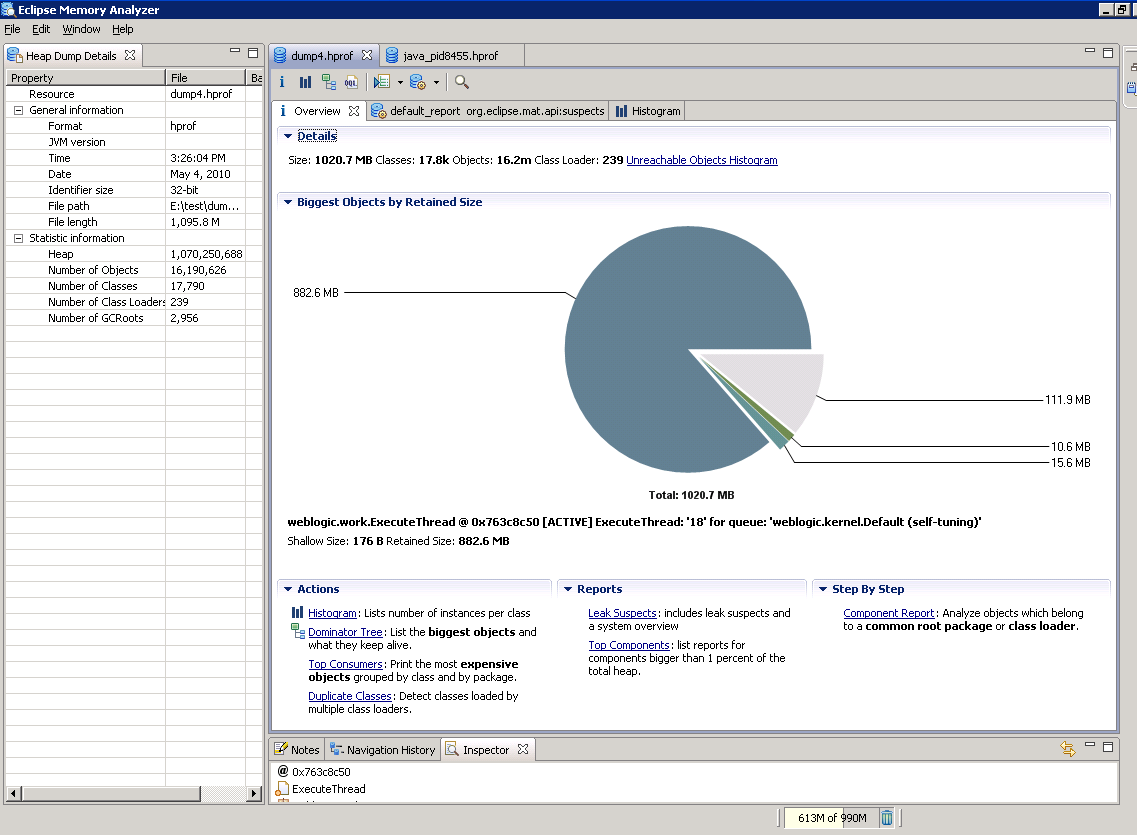
1. Use Jvisualvm or JMX to connect to the java process and request a heap dump.
2. In java6, you can use **jmap** if the app server is not started as a service:

$ jmap -dump:format=b,file=heap.bin <PID>

## How to Analyze a Heap Dump

As heap dumps can be very large, especially in 64-bit environments, it is sometimes necessary to perform the analysis in a 64-bit environment.

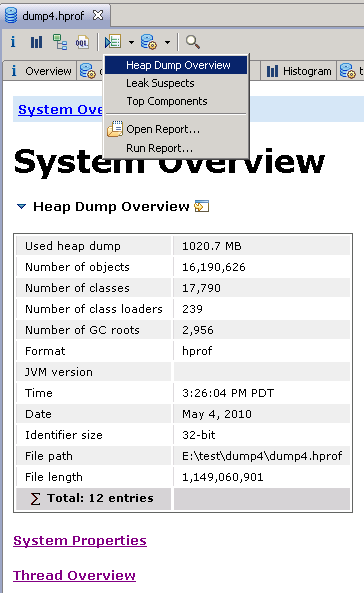
For smaller dumps (under 2GB) the Eclipse Memory Analyzer is very useful.



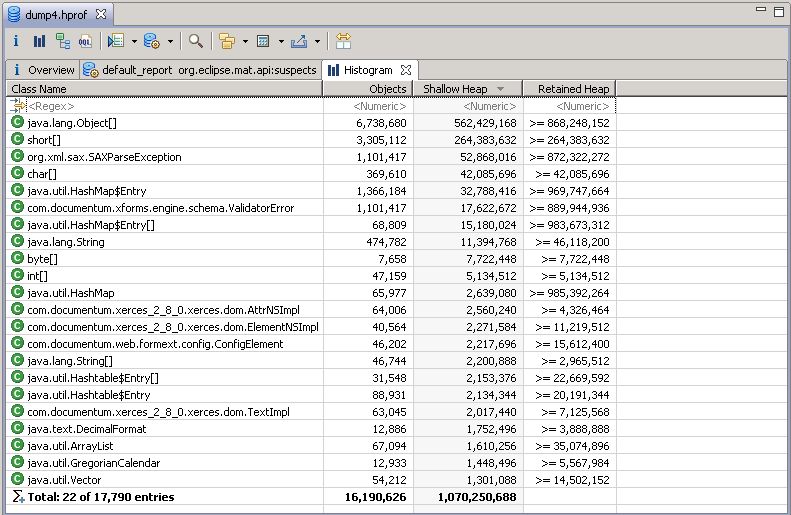
It is capable of analyzing heaps from all JVM vendors (a separate plugin is required when analyzing “phd” dumps from IBM Java, available here: <ftp://ftp.software.ibm.com/software/java/support/tools/dtfj/dtfj-updatesite.zip> ).

Up to a 2GB heap dump can be analyzed on a 32-bit OS.

The Heap Dump Overview report shows information about the dump and provides links to the java settings (System Properties link) and thread information (Thread Overview link).

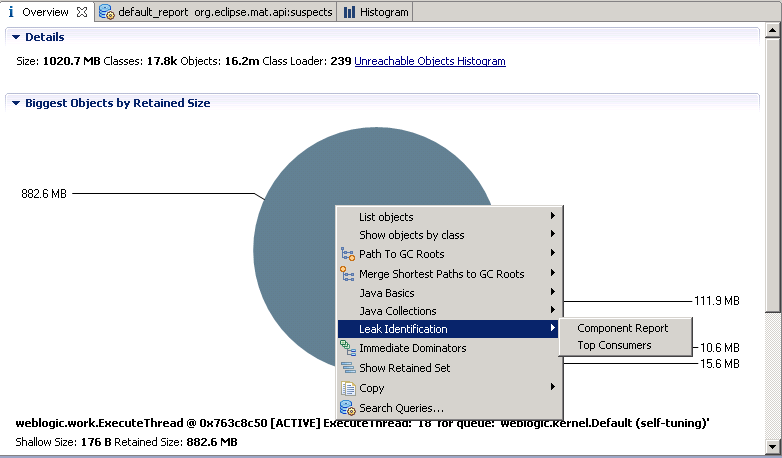


If you click on the Histogram button, you will see a list of classes and their memory consumption.

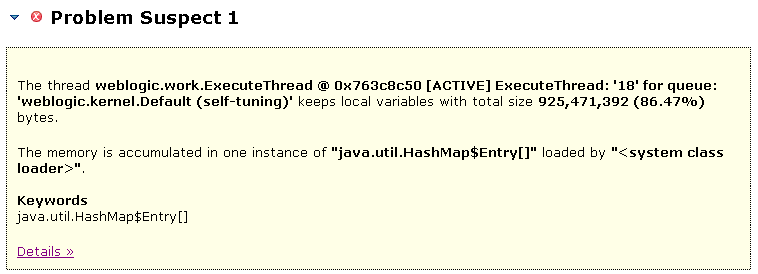


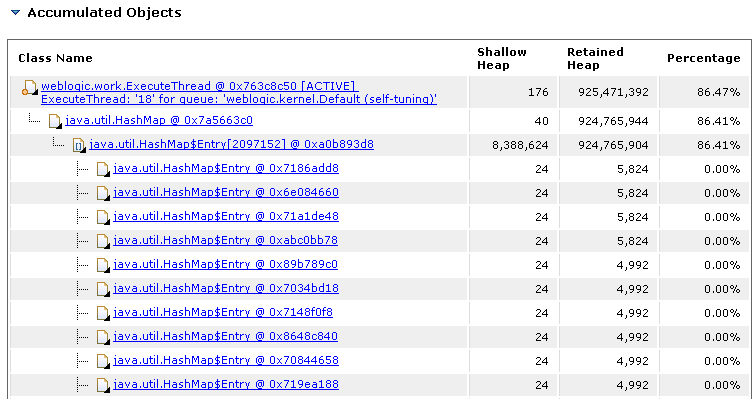
Shallow heap is the memory consumed by the object. Retained heap is the sum of all of the shallow heaps kept alive by that object. In the example above, we can see that there are over are over a million com.documentum.xforms.engine.schema.ValidatorError class objects. Those objects consume 17,622,672 bytes directly, and are holding on to 969,747,664 bytes in total.

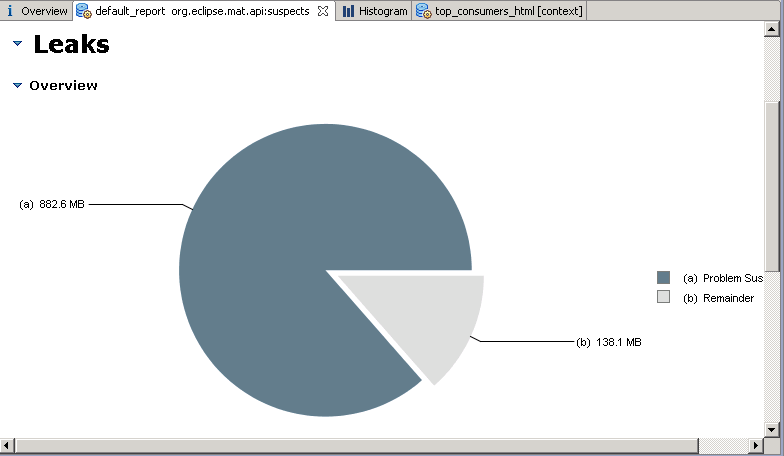
If you left-click on the Overview report, you can request a Leak Identification report which will help to pinpoint the classes consuming the most memory in the heap.



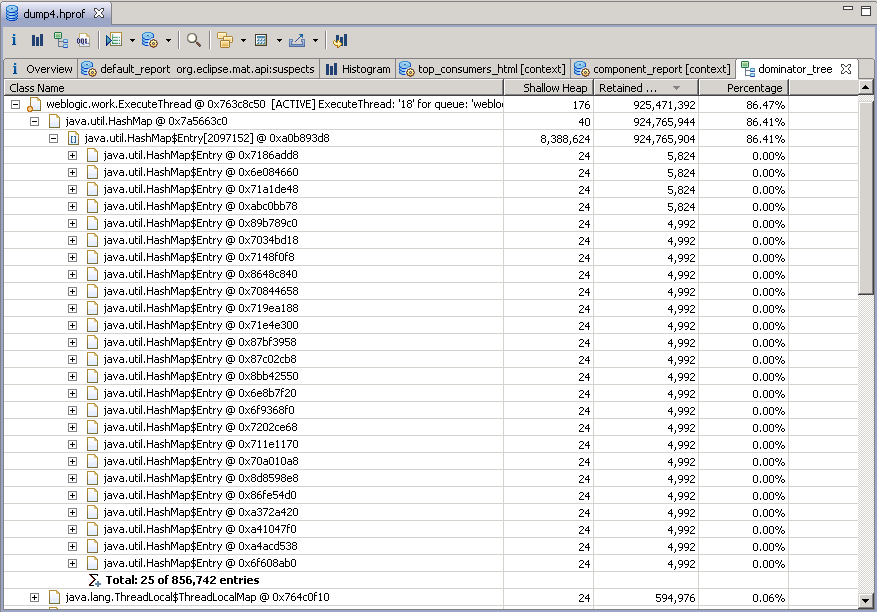
The leak suspect report allows you to drill down to the suspected root cause by looking for very large objects or collections of objects that might be due to a memory leak.







If you look for the largest drop in retained heap, you will get an idea of where the problem is.



In this case, we can see that there are 856,741 hash map entries that are consuming 86.41% of the heap, which is definitely worth drilling in to.

In summary, when investigating heap consumption issues (heap exhaustion, memory leaks etc) use the following steps to identify the root cause:

1. Instrument the JVM to dump the heap to a file automatically on OOM
2. Look at Heap Histogram to identify classes with high shallow and retained heap sizes
3. Look at the Leak Suspects report to jump to objects holding on to a lot of heap
4. Look at the Dominator Tree to look for large drops in Retained Size and/or large object counts
5. Try to identify the source of the leak by tracing back to the GC Roots, or drilling down to identify the Documentum class involved