Troubleshoot the JVM like never before

JVM Troubleshooting Guide

Java Code Geeks
JAVA 2 JAVA DEVELOPERS RESOURCE CENTER

Pierre-Hugues Charbonneau
Ilias Tsagklis

Table of Contents

Oracle HotSpot JVM Memory

Java HotSpot VM Heap space

This section will provide you with a high level overview of the different Java Heap memory spaces of the Oracle Java HotSpot VM. This understanding is quite important for any individual involved in production support given how frequent memory problems are observed. Proper knowledge of the Java VM Heap Space is critical.

Your Java VM is basically the foundation of your Java program which provides you with dynamic memory management services, garbage collection, Threads, IO and native operations and more.

The Java Heap Space is the memory "container" of you runtime Java program which provides to your Java program the proper memory spaces it needs (Java Heap, Native Heap) and managed by the JVM itself.

The JVM HotSpot memory is split between 3 memory spaces:

- The Java Heap
- The PermGen (permanent generation) space
- The Native Heap (C-Heap)

Here is the breakdown for each one of them:

Memory Space	Start-up arguments and tuning	Monitoring strategies	Description
Java Heap	-Xmx (maximum Heap space) -Xms (minimum Heap size) EX: -Xmx1024m -Xms1024m	- verbose GC - JMX API - JConsole - Other monitoring tools	The Java Heap is storing your primary Java program Class instances.
PermGen	-XX:MaxPermSize (maximum size) -XX:PermSize (minimum size) EX: -XX:MaxPermSize=512	- verbose GC - JMX API - JConsole - Other monitoring tools	The Java HotSpot VM permanent generation space is the JVM storage used mainly to store your Java Class objects such as names and method of the Classes, internal JVM objects and other JIT optimization related

	m -XX:PermSize=256m		data.
Native Heap (C-Heap)	Not configurable directly. For a 32-bit VM, the C-Heap capacity = 4 Gig – Java Heap - PermGen For a 64-bit VM, the C-Heap capacity = Physical server total RAM & virtual memory – Java Heap - PermGen	- Total process size check in Windows and Linux - pmap command on Solaris & Linux - svmon command on AIX	The C-Heap is storing objects such as MMAP file, other JVM and third party native code objects.

Java Heap Space - Overview & life cycle

Your Java program life cycle typically looks like this:

- Java program coding (via Eclipse IDE etc.) e.g. HelloWorld.java
- Java program compilation (Java compiler or third party build tools such as Apache Ant, Apache Maven..) e.g. HelloWord.class
- Java program start-up and runtime execution e.g. via your HelloWorld.main() method

Now let's dissect your HelloWorld.class program so you can better understand.

- At start-up, your JVM will load and cache some of your static program and JDK libraries to the Native Heap, including native libraries, Mapped Files such as your program Jar file(s), Threads such as the main start-up Thread of your program etc.
- Your JVM will then store the "static" data of your HelloWorld.class Java program to the PermGen space (Class metadata, descriptors, etc.).
- Once your program is started, the JVM will then manage and dynamically allocate the memory of your Java program to the Java Heap (YoungGen & OldGen). This is why it is so important that you understand how much memory your Java program needs to you can properly fine-tuned the capacity of your Java Heap controlled via -Xms & -Xmx JVM parameters. Profiling, Heap Dump analysis allow you to determine your Java program memory footprint.
- Finally, the JVM has to also dynamically release the memory from the Java Heap Space that your program no longer need; this is called the garbage collection process. This process can be easily monitored via the JVM verbose GC or a monitoring tool of your choice such as Jconsole.

Java HotSpot VM PermGen space

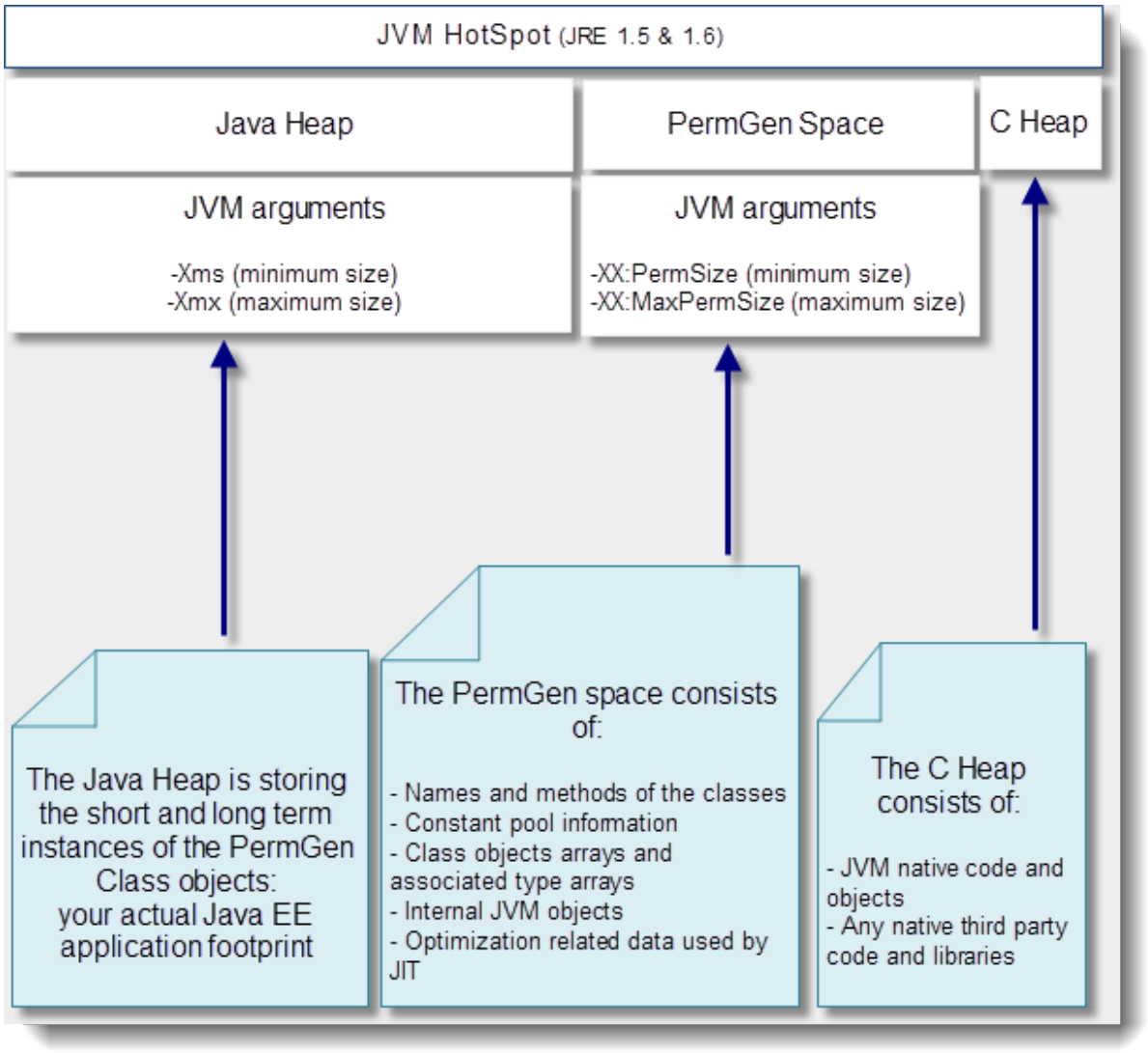
The Java HotSpot VM permanent generation space is the JVM storage used mainly to store your Java Class objects. The Java Heap is the primary storage that is storing the actual short and long term

instances of your PermGen Class objects.

The PermGen space is fairly static by nature unless using third party tool and/or Java Reflection API which relies heavily on dynamic class loading.

It is important to note that this memory storage is applicable only for a Java HotSpot VM; other JVM vendors such as IBM and Oracle JRockit do not have such fixed and configurable PermGen storage and are using other techniques to manage the non Java Heap memory (native memory).

Find below a graphical view of a JVM HotSpot Java Heap vs. PermGen space breakdown along with its associated attributes and capacity tuning arguments.



Apart from the Oracle HotSpot JVM, there are other virtual machines provided by differented vendors. The following sections examine the memory configurations used by other JVMs. Understanding those is quite important given the implementation and naming convention differences between HotSpot and

the other JVMs.

IBM JVM Memory

The IBM VM memory is split between 2 memory spaces:

- The Java Heap (nursery and tenured spaces)
- The Native Heap (C-Heap)

Here is the breakdown for each one of them:

Memory Space	Start-up arguments and tuning	Monitoring strategies	Description
Java Heap	-Xmx (maximum Heap space) -Xms (minimum Heap size)- verbose GC - JMX API - IBM monitoring tools EX: -Xmx1024m -Xms1024m GC policy Ex: -Xgcpolicy:gencon (enable gencon GC policy)	- verbose GC - JMX API - IBM monitoring tools	The IBM Java Heap is typically split between the nursery and tenured space (YoungGen, OldGen). The gencon GC policy (combo of concurrent and generational GC) is typically used for Java EE platforms in order to minimize the GC pause time.
Native Heap (C-Heap)	Not configurable directly. For a 32-bit VM, the C-Heap capacity = 4 Gig – Java Heap For a 64-bit VM, the C-Heap capacity = Physical server total RAM & virtual memory – Java Heap	- svmon command	The C-Heap is storing class metadata objects including library files, other JVM and third party native code objects.

As you might noticed, there is no PermGen space for the IBM VM. The PermGen space is only applicable to the HotSpot VM. The IBM VM is using the Native Heap for Class metadata related data.

Also note that Oracle is also starting to remove the PermGen space for the HotSpot VM, as we will discuss in a next section.

Oracle JRockit JVM Memory

The JRockit VM memory is split between 2 memory spaces:

- The Java Heap (YoungGen and OldGen)
- The Native memory space (Classes pool, C-Heap, Threads...)

Memory Space	Start-up arguments and tuning	Monitoring strategies	Description
Java Heap	-Xmx (maximum Heap space) -Xms (minimum Heap size) EX: -Xmx1024m -Xms1024m	- verbose GC - JMX API - JRockit Mission Control tools suite	The JRockit Java Heap is typically split between the YoungGen (short-lived objects), OldGen (long-lived objects).
Native memory space	Not configurable directly. For a 32-bit VM, the native memory space capacity = 2-4 Gig – Java Heap *** Process size limit of 2 GB, 3 GB or 4 GB depending of your OS *** For a 64-bit VM, the native memory space capacity = Physical server total RAM & virtual memory – Java Heap	- Total process size check in Windows and Linux - pmap command on Solaris & Linux - JRockit JRCMD tool	The JRockit Native memory space is storing the Java Classes metadata, Threads and objects such as library files, other JVM and third party native code objects.

Similar to the IBM VM, there is no PermGen space for the JRockit VM. The PermGen space is only applicable to the HotSpot VM. The JRockit VM is using the Native Heap for Class metadata related data.

The JRockit VM tend to uses more native memory in exchange for better performance. JRockit does not have an interpretation mode, compilation only, so due to its additional native memory needs the process size tends to use a couple of hundred MB larger than the equivalent Sun JVM size. This should not be a big problem unless you are using a 32-bit JRockit with a large Java Heap requirement; in this scenario, the risk of OutOfMemoryError due to Native Heap depletion is higher for a JRockit VM (e.g. for a 32-bit VM, bigger is the Java Heap, smaller is memory left for the Native Heap).

Oracle's strategy, being the vendor for both HotSpot and JRockit product lines, is to merge the two Vms to a single JVM project that will include the best features of each one. This will also simplify JVM tuning since right now failure to understand the differences between these 2 VM's can lead to bad tuning recommendations and performance problems.

Tips for proper Java Heap size

Determination of proper Java Heap size for a production system is not a straightforward exercise. Multiple performance problem can occur due to inadequate Java Heap capacity and tuning. This section will provide some tips that can help you determine the optimal Java heap size, as a starting point, for your current or new production environment. Some of these tips are also very useful regarding the prevention and resolution of OutOfMemoryError problems, including memory leaks.

Please note that these tips are intended to “help you” determine proper Java Heap size. Since each IT environment is unique, you are actually in the best position to determine precisely the required Java Heap specifications of your client’s environment.

#1 - JVM: you always fear what you don't understand

How can you expect to configure, tune and troubleshoot something that you don't understand? You may never have the chance to write and improve Java VM specifications but you are still free to learn its foundation in order to improve your knowledge and troubleshooting skills. Some may disagree, but from my perspective, the thinking that Java programmers are not required to know the internal JVM memory management is an illusion.

Java Heap tuning and troubleshooting can especially be a challenge for Java & Java EE beginners. Find below a typical scenario:

- Your client production environment is facing OutOfMemoryError on a regular basis and causing lot of business impact. Your support team is under pressure to resolve this problem.
- A quick Google search allows you to find examples of similar problems and you now believe (and assume) that you are facing the same problem.
- You then grab JVM -Xms and -Xmx values from another person OutOfMemoryError problem case, hoping to quickly resolve your client's problem.
- You then proceed and implement the same tuning to your environment. 2 days later you realize problem is still happening (even worse or little better)...the struggle continues...

What went wrong?

- You failed to first acquire proper understanding of the root cause of your problem.
- You may also have failed to properly understand your production environment at a deeper level (specifications, load situation etc.). Web searches is a great way to learn and share knowledge but you have to perform your own due diligence and root cause analysis.
- You may also be lacking some basic knowledge of the JVM and its internal memory management, preventing you to connect all the dots together.

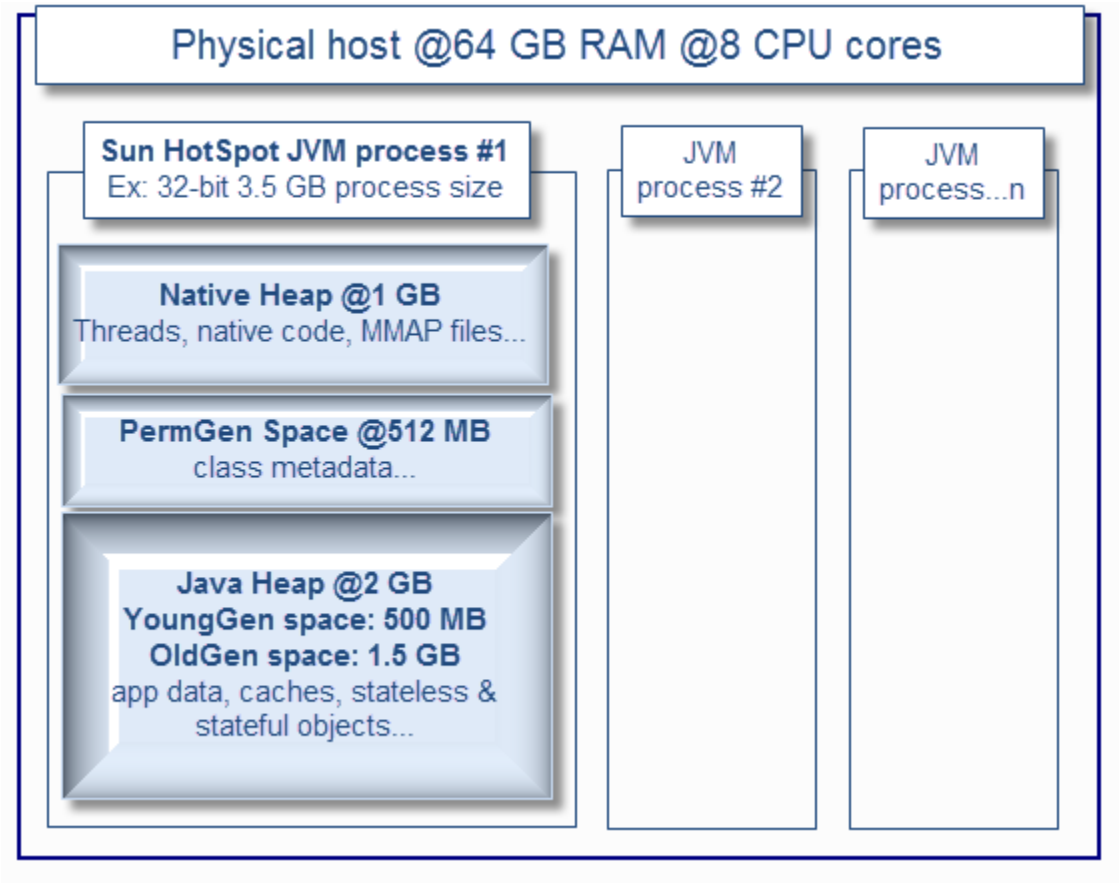
My #1 tip and recommendation to you is to learn and understand the basic JVM principles along with its different memory spaces. Such knowledge is critical as it will allow you to make valid recommendations to your clients and properly understand the possible impact and risk associated with future tuning considerations.

As a reminder, the Java VM memory is split up to 3 memory spaces:

- The Java Heap: Applicable for all JVM vendors, usually split between YoungGen (nursery) & OldGen (tenured) spaces.
- The PermGen (permanent generation): Applicable to the Sun HotSpot VM only (PermGen space will be removed in future Java updates)
- The Native Heap (C-Heap): Applicable for all JVM vendors.

As you can see, the Java VM memory management is more complex than just setting up the biggest value possible via –Xmx. You have to look at all angles, including your native and PermGen space requirement along with physical memory availability (and # of CPU cores) from your physical host(s).

It can get especially tricky for 32-bit JVM since the Java Heap and native Heap are in a race. The bigger your Java Heap, the smaller the native Heap. Attempting to setup a large Heap for a 32-bit VM e.g .2.5 GB+ increases risk of native OutOfMemoryError depending of your application(s) footprint, number of Threads etc. 64-bit JVM resolves this problem but you are still limited to physical resources availability and garbage collection overhead (cost of major GC collections go up with size). The bottom line is that the bigger is not always the better so please do not assume that you can run all your 20 Java EE applications on a single 16 GB 64-bit JVM process.



#2 - Data and application is king: review your static footprint requirement

Your application(s) along with its associated data will dictate the Java Heap footprint requirement. By static memory, I mean "predictable" memory requirements as per below.

- Determine how many different applications you are planning to deploy to a single JVM process e.g. number of EAR files, WAR files, jar files etc. The more applications you deploy to a single JVM, higher demand on native Heap.
- Determine how many Java classes will be potentially loaded at runtime; including third part API's. The more class loaders and classes that you load at runtime, higher demand on the HotSpot VM PermGen space and internal JIT related optimization objects.
- Determine data cache footprint e.g. internal cache data structures loaded by your application (and third party API's) such as cached data from a database, data read from a file etc. The more data caching that you use, higher demand on the Java Heap OldGen space.
- Determine the number of Threads that your middleware is allowed to create. This is very important since Java threads require enough native memory or OutOfMemoryError will be thrown.

For example, you will need much more native memory and PermGen space if you are planning to deploy 10 separate EAR applications on a single JVM process vs. only 2 or 3. Data caching not

serialized to a disk or database will require extra memory from the OldGen space.

Try to come up with reasonable estimates of the static memory footprint requirement. This will be very useful to setup some starting point JVM capacity figures before your true measurement exercise (e.g. tip #4). For 32-bit JVM, I usually do not recommend a Java Heap size high than 2 GB (-Xms2048m, -Xmx2048m) since you need enough memory for PermGen and native Heap for your Java EE applications and threads.

This assessment is especially important since too many applications deployed in a single 32-bit JVM process can easily lead to native Heap depletion; especially in a multi threads environment.

For a 64-bit JVM, a Java Heap size of 3 GB or 4 GB per JVM process is usually my recommended starting point.

#3 - Business traffic set the rules: review your dynamic footprint requirement

Your business traffic will typically dictate your dynamic memory footprint. Concurrent users & requests generate the JVM GC "heartbeat" that you can observe from various monitoring tools due to very frequent creation and garbage collections of short & long lived objects. As you saw from the above JVM diagram, a typical ratio of YoungGen vs. OldGen is 1:3 or 33%.

For a typical 32-bit JVM, a Java Heap size setup at 2 GB (using generational & concurrent collector) will typically allocate 500 MB for YoungGen space and 1.5 GB for the OldGen space.

Minimizing the frequency of major GC collections is a key aspect for optimal performance so it is very important that you understand and estimate how much memory you need during your peak volume.

Again, your type of application and data will dictate how much memory you need. Shopping cart type of applications (long lived objects) involving large and non-serialized session data typically need large Java Heap and lot of OldGen space. Stateless and XML processing heavy applications (lot of short lived objects) require proper YoungGen space in order to minimize frequency of major collections.

Example:

- You have 5 EAR applications (~2 thousands of Java classes) to deploy (which include middleware code as well...).
- Your native heap requirement is estimated at 1 GB (has to be large enough to handle Threads creation etc.).
- Your PermGen space is estimated at 512 MB.
- Your internal static data caching is estimated at 500 MB.
- Your total forecast traffic is 5000 concurrent users at peak hours.
- Each user session data footprint is estimated at 500 K.
- Total footprint requirement for session data alone is 2.5 GB under peak volume.

As you can see, with such requirement, there is no way you can have all this traffic sent to a single JVM 32-bit process. A typical solution involves splitting (tip #5) traffic across a few JVM processes and / or physical host (assuming you have enough hardware and CPU cores available).

However, for this example, given the high demand on static memory and to ensure a scalable environment in the long run, I would also recommend 64-bit VM but with a smaller Java Heap as a starting point such as 3 GB to minimize the GC cost. You definitely want to have extra buffer for the OldGen space so I typically recommend up to 50% memory footprint post major collection in order to keep the frequency of Full GC low and enough buffer for fail-over scenarios.

Most of the time, your business traffic will drive most of your memory footprint, unless you need significant amount of data caching to achieve proper performance which is typical for portal (media) heavy applications. Too much data caching should raise a yellow flag that you may need to revisit some design elements sooner than later.

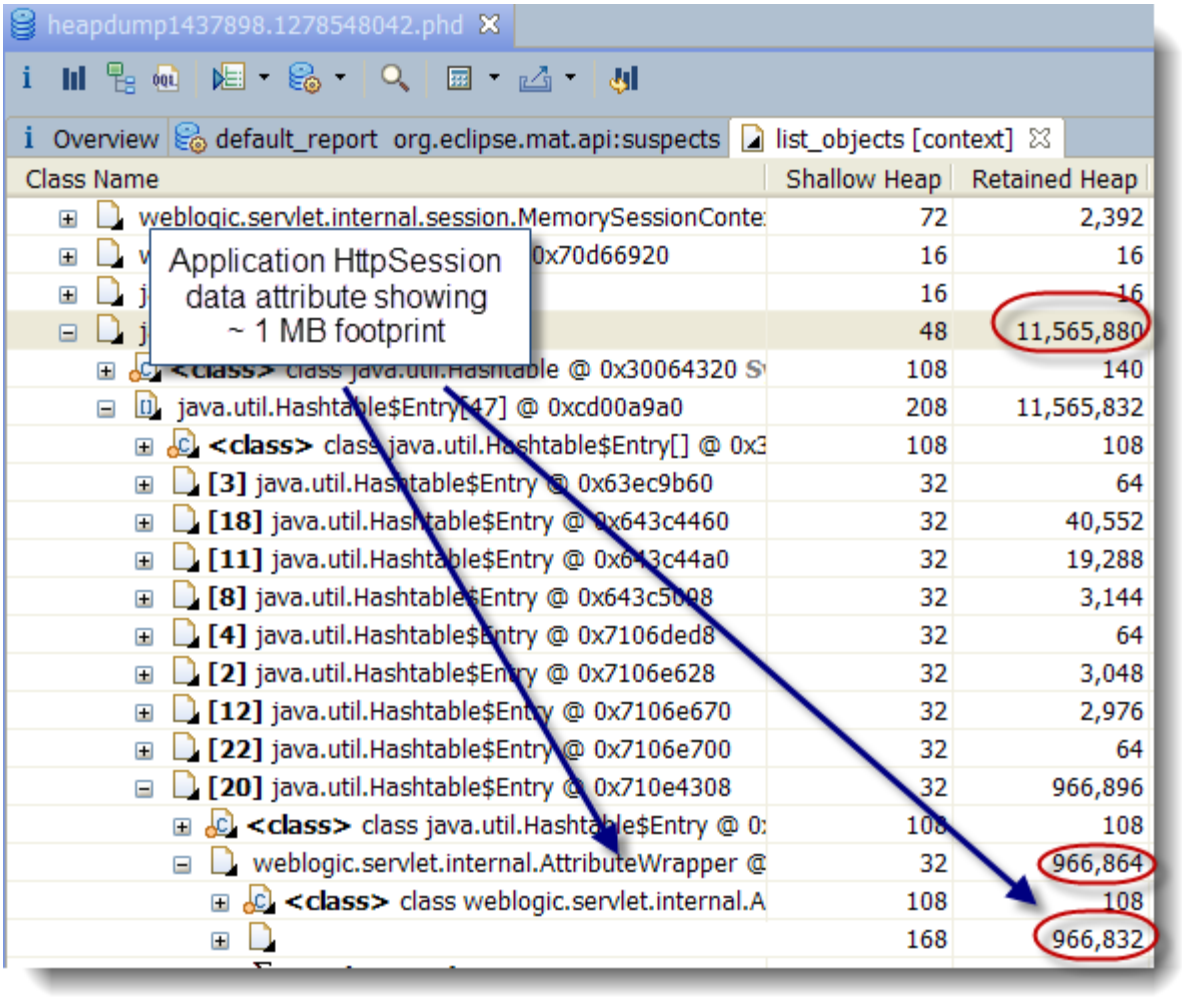
#4 - Don't guess it, measure it!

At this point you should:

- Understand the basic JVM principles and memory spaces
- Have a deep view and understanding of all applications along with their characteristics (size, type, dynamic traffic, stateless vs. stateful objects, internal memory caches etc.)
- Have a very good view or forecast on the business traffic (# of concurrent users etc.) and for each application
- Some ideas if you need a 64-bit VM or not and which JVM settings to start with
- Some ideas if you need more than one JVM (middleware) processes

But wait, your work is not done yet. While this above information is crucial and great for you to come up with "best guess" Java Heap settings, it is always best and recommended to simulate your application(s) behaviour and validate the Java Heap memory requirement via proper profiling, load & performance testing.

You can learn and take advantage of tools such as JProfiler. From my perspective, learning how to use a profiler is the best way to properly understand your application memory footprint. Another approach I use for existing production environments is heap dump analysis using the Eclipse MAT tool. Heap Dump analysis is very powerful and allow you to view and understand the entire memory footprint of the Java Heap, including class loader related data and is a must do exercise in any memory footprint analysis; especially memory leaks.



Java profilers and heap dump analysis tools allow you to understand and validate your application memory footprint, including detection and resolution of memory leaks. Load and performance testing is also a must since this will allow you to validate your earlier estimates by simulating your forecast concurrent users. It will also expose your application bottlenecks and allow you to further fine tune your JVM settings. You can use tools such as Apache JMeter which is very easy to learn and use or explore other commercial products.

Finally, I have seen quite often Java EE environments running perfectly fine until the day where one piece of the infrastructure start to fail e.g. hardware failure. Suddenly the environment is running at reduced capacity (reduced # of JVM processes) and the whole environment goes down. What happened?

There are many scenarios that can lead to domino effects but lack of JVM tuning and capacity to handle fail-over (short term extra load) is very common. If your JVM processes are running at 80%+ OldGen space capacity with frequent garbage collections, how can you expect to handle any fail-over scenario?

Your load and performance testing exercise performed earlier should simulate such scenario and you

should adjust your tuning settings properly so your Java Heap has enough buffer to handle extra load (extra objects) at short term. This is mainly applicable for the dynamic memory footprint since fail-over means redirecting a certain % of your concurrent users to the available JVM processes (middleware instances).

#5 - Divide and conquer

At this point you have performed dozens of load testing iterations. You know that your JVM is not leaking memory. Your application memory footprint cannot be reduced any further. You tried several tuning strategies such as using a large 64-bit Java Heap space of 10 GB+, multiple GC policies but still not finding your performance level acceptable?

In my experience I found that, with current JVM specifications, proper vertical and horizontal scaling which involved creating a few JVM processes per physical host and across several hosts will give you the throughput and capacity that you are looking for. Your IT environment will also more fault tolerant if you break your application list in a few logical silos, with their own JVM process, Threads and tuning values.

This "divide and conquer" strategy involves splitting your application(s) traffic to multiple JVM processes and will provide you with:

- Reduced Java Heap size per JVM process (both static & dynamic footprint)
- Reduced complexity of JVM tuning
- Reduced GC elapsed and pause time per JVM process
- Increased redundancy and fail-over capabilities
- Aligned with latest Cloud and IT virtualization strategies

The bottom line is that when you find yourself spending too much time in tuning that single elephant 64-bit JVM process, it is time to revisit your middleware and JVM deployment strategy and take advantage of vertical & horizontal scaling. This implementation strategy is more taxing for the hardware but will really pay off in the long run.

Java Threading: JVM Retained memory analysis

Having discussed the various heap spaces of the JVM, this section will provide you with a tutorial allowing you to determine how much and where Java heap space is retained from your active application Java threads. A true case study from an Oracle Weblogic 10.0 production environment will be presented in order for you to better understand the analysis process.

We will also attempt to demonstrate that excessive garbage collection or Java heap space memory footprint problems are often not caused by true memory leaks but instead due to thread execution patterns and high amount of short lived objects.

Background

Java threads are part of the JVM fundamentals. Your Java heap space memory footprint is driven not

only by static and long lived objects but also by short lived objects.

OutOfMemoryError problems are often wrongly assumed to be due to memory leaks. We often overlook faulty thread execution patterns and short lived objects they "retain" on the Java heap until their executions are completed. In this problematic scenario:

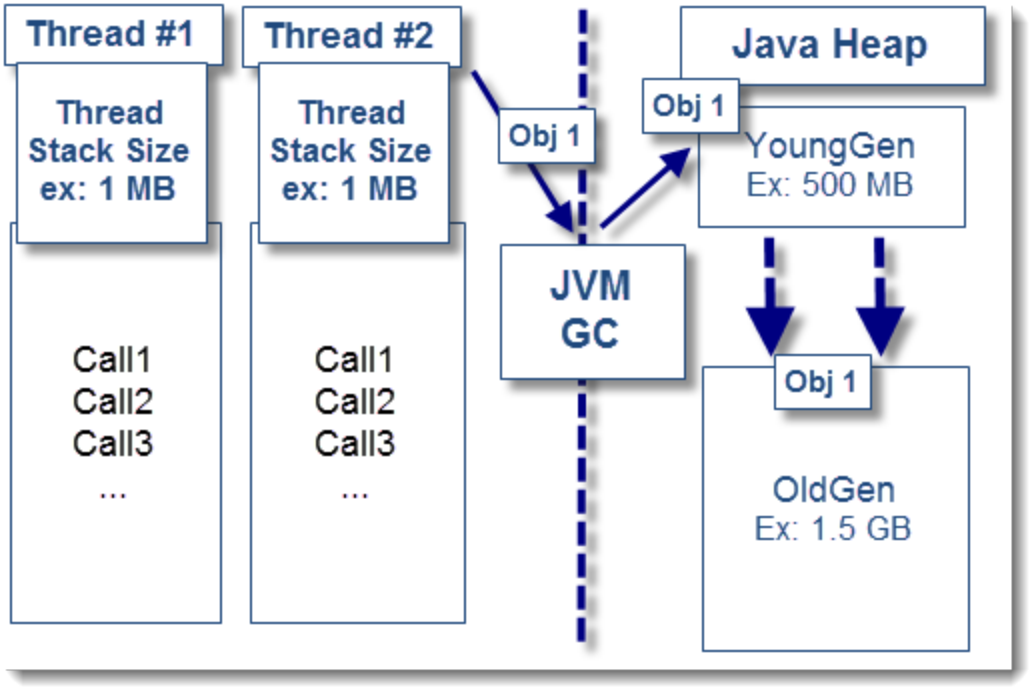
- Your "expected" application short lived / stateless objects (XML, JSON data payload etc.) become retained by the threads for too long (thread lock contention, huge data payload, slow response time from remote system etc.).
- Eventually such short lived objects get promoted to the long lived object space e.g. OldGen/tenured space by the garbage collector.
- As a side effect, this is causing the OldGen space to fill up rapidly, increasing the Full GC (major collections) frequency.
- Depending of the severity of the situation this can lead to excessive GC garbage collection, increased JVM paused time and ultimately “OutOfMemoryError: Java heap space”.
- Your application is now down, you are now puzzled on what is going on.
- Finally, you are thinking to either increase the Java heap or look for memory leaks...are you really on the right track?

In the above scenario, you need to look at the thread execution patterns and determine how much memory each of them retain at a given time.

OK I get the picture but what about the thread stack size?

It is very important to avoid any confusion between thread stack size and Java memory retention. The thread stack size is a special memory space used by the JVM to store each method call. When a thread calls method A, it "pushes" the call onto the stack. If method A calls method B, it gets also pushed onto the stack. Once the method execution completes, the call is "popped" off the stack.

The Java objects created as a result of such thread method calls are allocated on the Java heap space. Increasing the thread stack size will definitely not have any effect. Tuning of the thread stack size is normally required when dealing with java.lang.stackoverflowerror or “OutOfMemoryError: unable to create new native thread” problems.



Case study and problem context

The following analysis is based on a true production problem we investigated recently.

1. Severe performance degradation was observed from a Weblogic 10.0 production environment following some changes to the user web interface (using Google Web Toolkit and JSON as data payload).
2. Initial analysis did reveal several occurrences of “OutOfMemoryError: Java heap space” errors along with excessive garbage collection. Java heap dump files were generated automatically (-XX:+HeapDumpOnOutOfMemoryError) following OOM events.
3. Analysis of the verbose:gc logs did confirm full depletion of the 32-bit HotSpot JVM OldGen space (1 GB capacity).
4. Thread dump snapshots were also generated before and during the problem.
5. The only problem mitigation available at that time was to restart the affected Weblogic server when problem was observed.
6. A rollback of the changes was eventually performed which did resolve the situation.

The team first suspected a memory leak problem from the new code introduced.

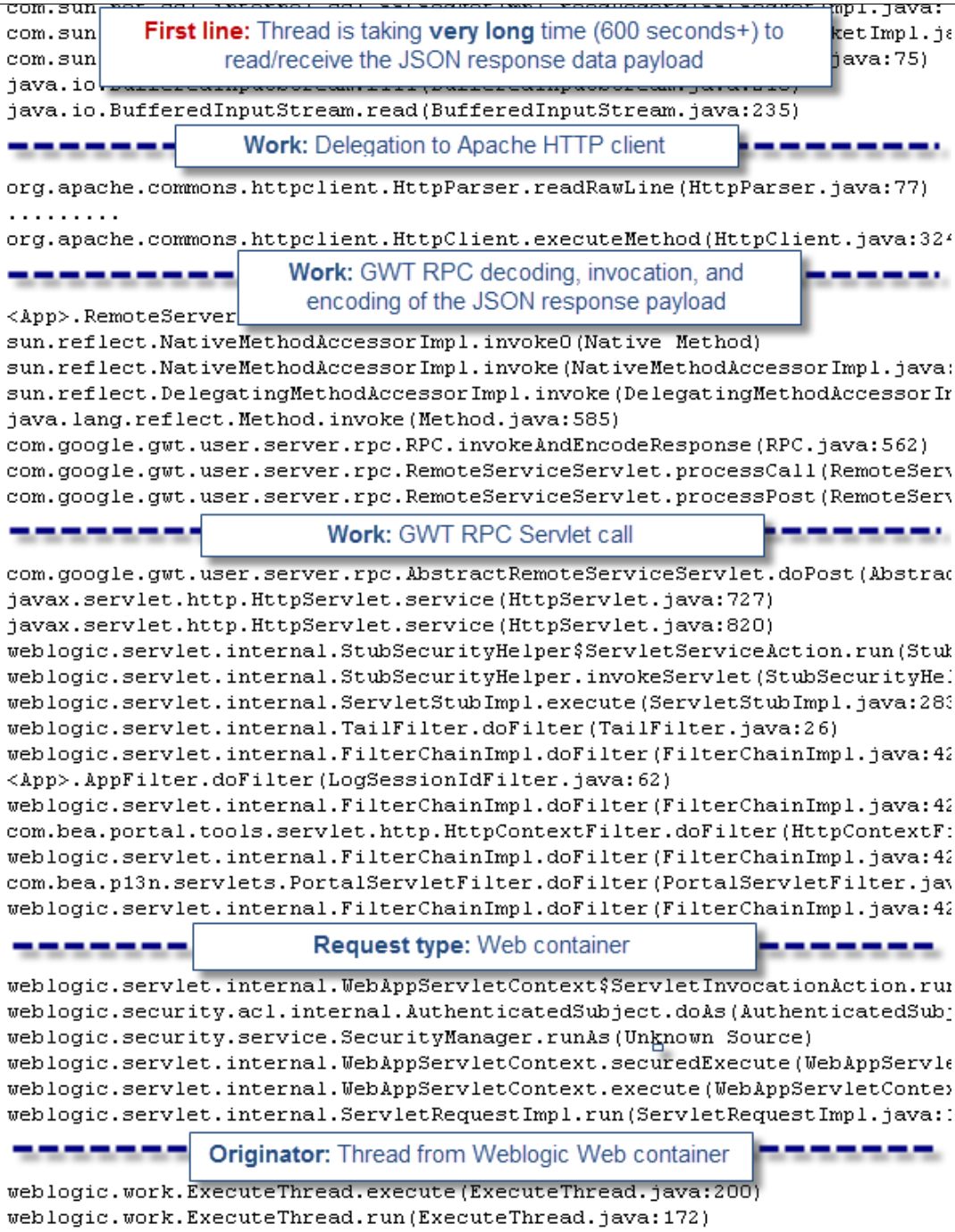
Thread dump analysis: looking for suspects

The first step we took was to perform an analysis of the generated thread dump data. Thread dump will often show you the culprit threads allocating memory on the Java heap. It will also reveal any hogging or stuck thread attempting to send and receive data payload from a remote system.

The first pattern we noticed was a good correlation between OOM events and STUCK threads observed from the Weblogic managed servers (JVM processes). Find below the primary thread

pattern found:

```
<10-Dec-2012 1:27:59 o'clock PM EST> <Error> <BEA-000337>
<[STUCK] ExecuteThread: '22' for queue:
'weblogic.kernel.Default (self-tuning)'
has been busy for "672" seconds working on the request
which is more than the configured time of "600" seconds.
```

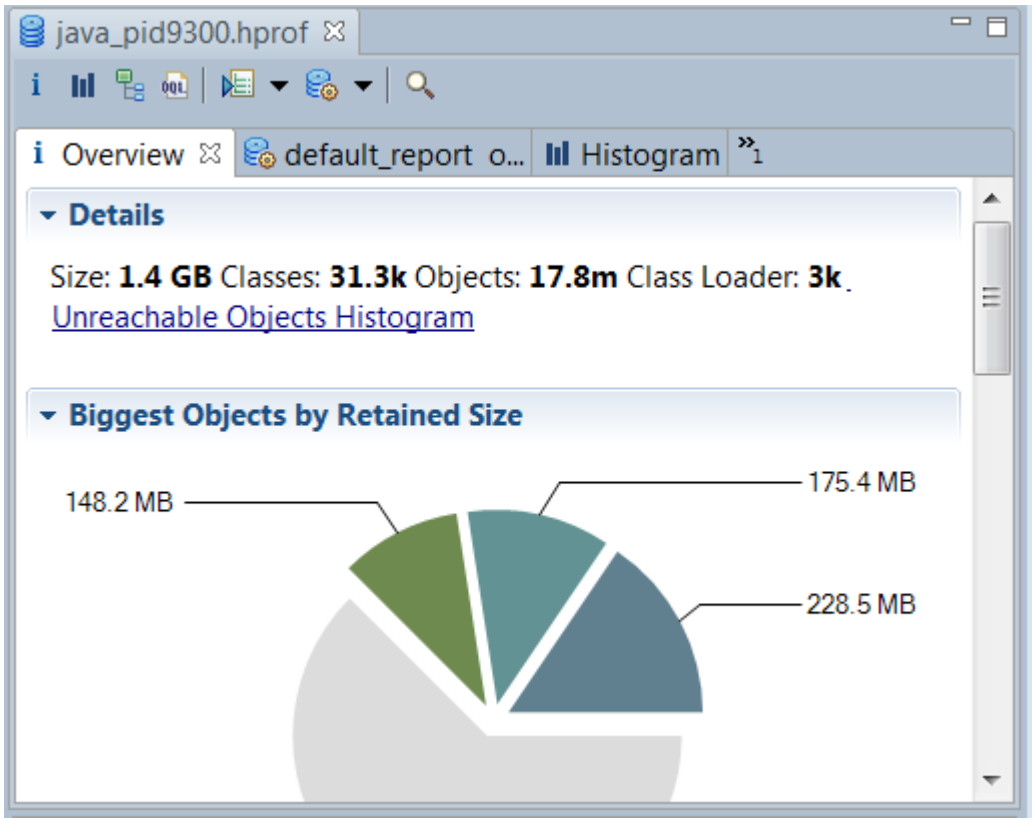


As you can see, the above thread appears to be STUCK or taking very long time to read and receive the JSON response from the remote server. Once we found that pattern, the next step was to correlate this finding with the JVM heap dump analysis and determine how much memory these stuck threads were taking from the Java heap.

Heap dump analysis: retained objects exposed!

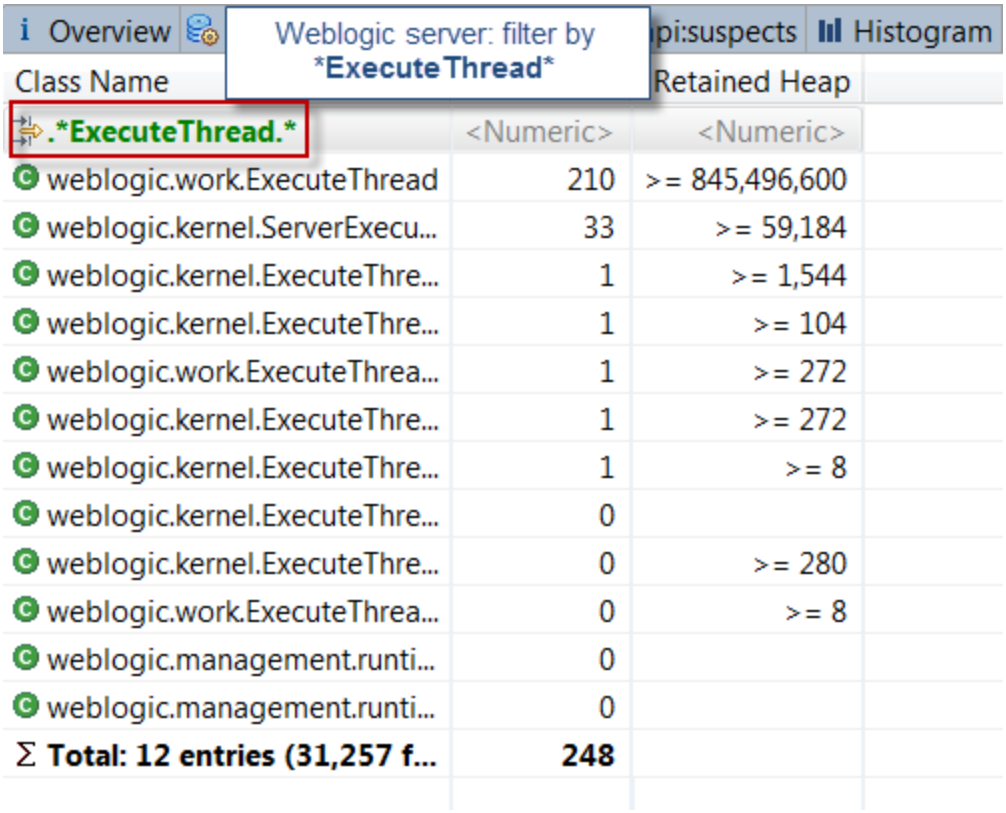
The Java heap dump analysis was performed using MAT. We will now list the different analysis steps which did allow us to pinpoint the retained memory size and source.

1. Load the HotSpot JVM heap dump



2. Select the HISTOGRAM view and filter by "ExecuteThread"

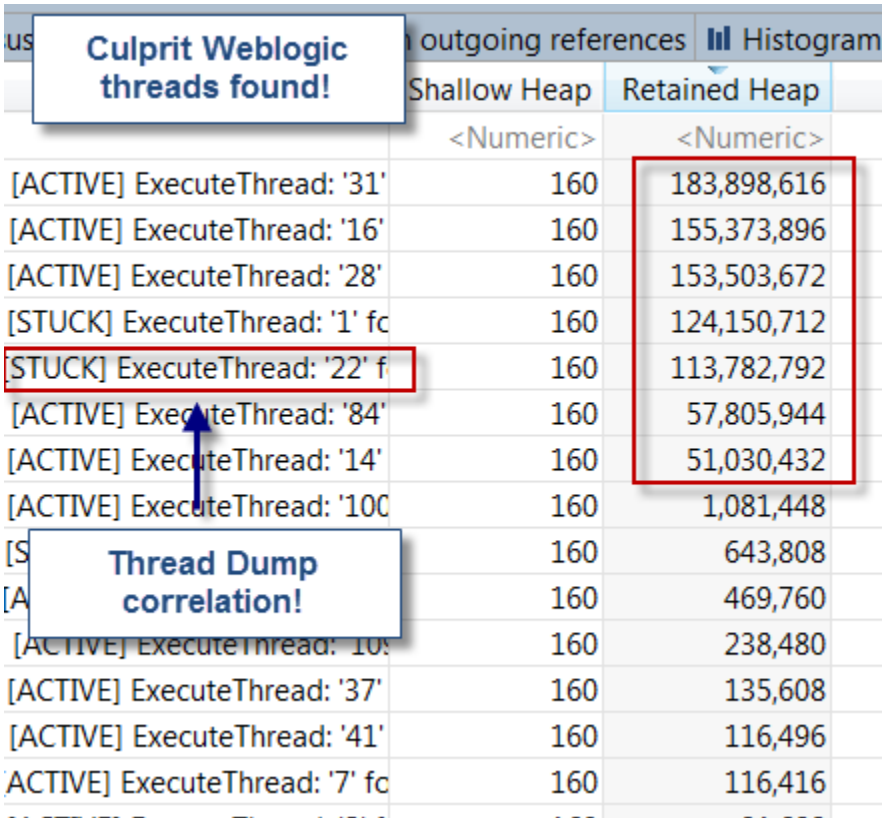
** ExecuteThread is the Java class used by the Weblogic kernel for thread creation & execution **



As you can see, this view was quite revealing. We can see a total of 210 Weblogic threads created. The total retained memory footprint from these threads is 806 MB. This is pretty significant for a 32-bit JVM process with 1 GB OldGen space. This view alone is telling us that the core of the problem and memory retention originates from the threads themselves.

3. Deep dive into the thread memory footprint analysis

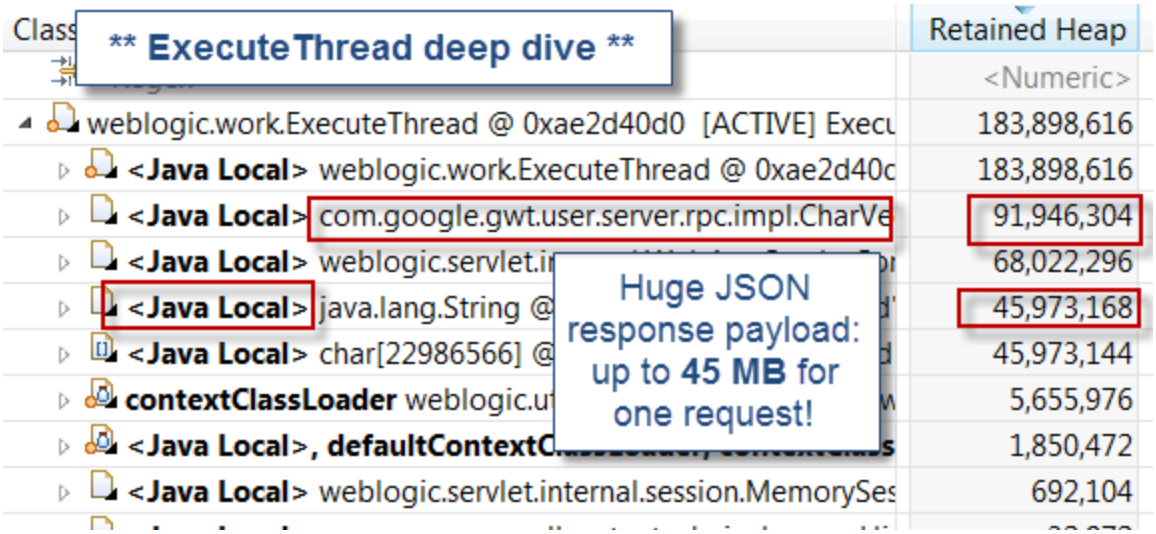
The next step was to deep dive into the thread memory retention. To do this, simply right click over the ExecuteThread class and select: List objects > with outgoing references.



As you can see, we were able to correlate STUCK threads from the thread dump analysis with high memory retention from the heap dump analysis. The finding was quite surprising.

4. Thread Java Local variables identification

The final analysis step did require us to expand a few thread samples and understand the primary source of memory retention.



As you can see, this last analysis step did reveal huge JSON response data payload at the root cause. That pattern was also exposed earlier via the thread dump analysis where we found a few threads taking very long time to read & receive the JSON response; a clear symptom of huge data payload footprint.

It is crucial to note that short lived objects created via local method variables will show up in the heap dump analysis. However, some of those will only be visible from their parent threads since they are not referenced by other objects, like in this case. You will also need to analyze the thread stack trace in order to identify the true caller, followed by a code review to confirm the root cause.

Following this finding, our delivery team was able to determine that the recent JSON faulty code changes were generating, under some scenarios, huge JSON data payload up to 45 MB+. Given the fact that this environment is using a 32-bit JVM with only 1 GB of OldGen space, you can understand that only a few threads were enough to trigger severe performance degradation.

This case study is clearly showing the importance of proper capacity planning and Java heap analysis, including the memory retained from your active application & Java EE container threads.

Java 8: From PermGen to Metaspace

Java 8 will introduce some new language and runtime features. One of these features is the complete removal of the Permanent Generation (PermGen) space which has been announced by Oracle since the release of JDK 7. Interned strings, for example, have already been removed from the PermGen space since JDK 7. The JDK 8 release finalizes its decommissioning. In this section, we shall discuss the PermGen successor: Metaspace.

We will also compare the runtime behavior of the HotSpot 1.7 vs. HotSpot 1.8 (b75) when executing a Java program “leaking” class metadata objects.

The final specifications, tuning flags and documentation around Metaspace should be available once

Java 8 is officially released.

Metaspace: A new memory space is born

The JDK 8 HotSpot JVM is now using native memory for the representation of class metadata and is called Metaspace; similar to the Oracle JRockit and IBM JVM's.

The good news is that it means no more java.lang.OutOfMemoryError: PermGen space problems and no need for you to tune and monitor this memory space anymore…not so fast. While this change is invisible by default, we will show you next that you will still need to worry about the class metadata memory footprint. Please also keep in mind that this new feature does not magically eliminate class and classloader memory leaks. You will need to track down these problems using a different approach and by learning the new naming convention.

In summary:

PermGen space situation

- This memory space is completely removed.
- The PermSize and MaxPermSize JVM arguments are ignored and a warning is issued if present at start-up.

Metaspace memory allocation model

- Most allocations for the class metadata are now allocated out of native memory.
- The classes that were used to describe class metadata have been removed.

Metaspace capacity

- By default class metadata allocation is limited by the amount of available native memory (capacity will of course depend if you use a 32-bit JVM vs. 64-bit along with OS virtual memory availability).
- A new flag is available (MaxMetaspaceSize), allowing you to limit the amount of native memory used for class metadata. If you don’t specify this flag, the Metaspace will dynamically re-size depending of the application demand at runtime.

Metaspace garbage collection

- Garbage collection of the dead classes and classloaders is triggered once the class metadata usage reaches the “MaxMetaspaceSize”.
- Proper monitoring & tuning of the Metaspace will obviously be required in order to limit the frequency or delay of such garbage collections. Excessive Metaspace garbage collections may be a symptom of classes, classloaders memory leak or inadequate sizing for your application.

Java heap space impact

Some miscellaneous data has been moved to the Java heap space. This means you may observe

an increase of the Java heap space following a future JDK 8 upgrade.

Metaspace monitoring

- Metaspace usage is available from the HotSpot 1.8 verbose GC log output.
- Jstat & JVisualVM have not been updated at this point based on our testing with b75 and the old PermGen space references are still present.

Enough theory now, let's see this new memory space in action via our leaking Java program…

PermGen vs. Metaspace runtime comparison

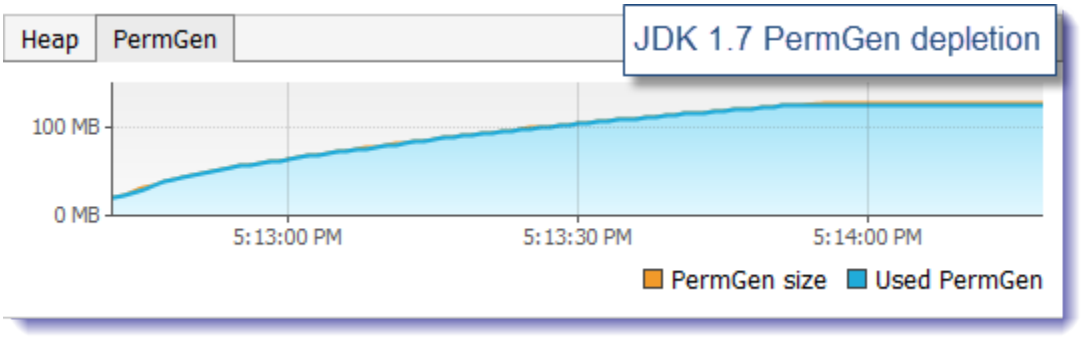
In order to better understand the runtime behavior of the new Metaspace memory space, we created a class metadata leaking Java program. You can download the source here.

The following scenarios will be tested:

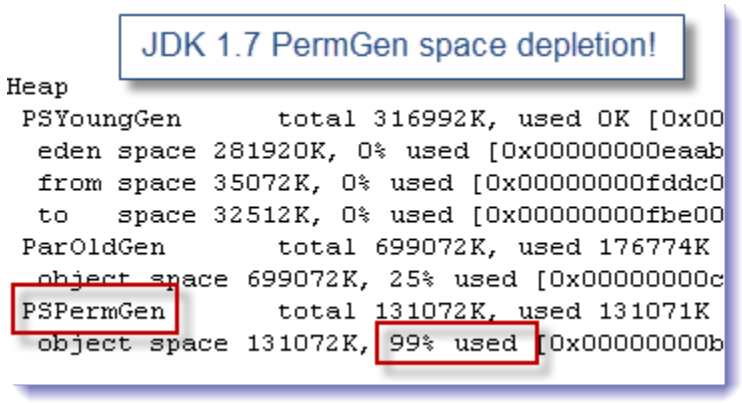
- Run the Java program using JDK 1.7 in order to monitor & deplete the PermGen memory space set at 128 MB.
- Run the Java program using JDK 1.8 (b75) in order to monitor the dynamic increase and garbage collection of the new Metaspace memory space.
- Run the Java program using JDK 1.8 (b75) in order to simulate the depletion of the Metaspace by setting the MaxMetaspaceSize value at 128 MB.

JDK 1.7 @64-bit – PermGen depletion

- Java program with 50K configured iterations
- Java heap space of 1024 MB
- Java PermGen space of 128 MB (-XX:MaxPermSize=128m)



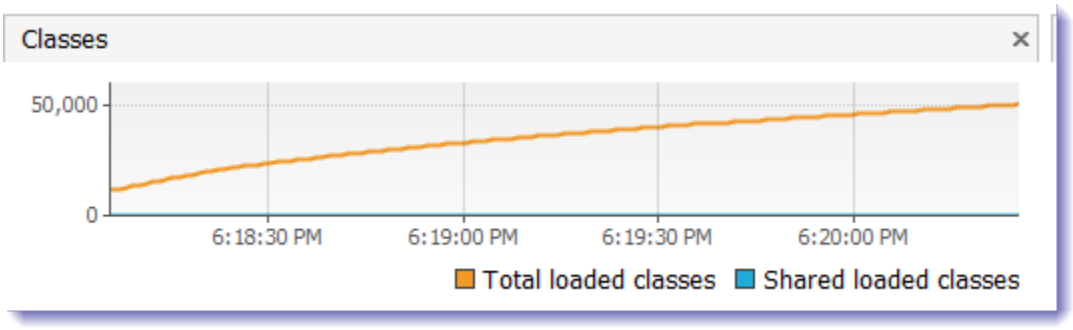
As you can see form JVisualVM, the PermGen depletion was reached after loading about 30K+ classes. We can also see this depletion from the program and GC output.

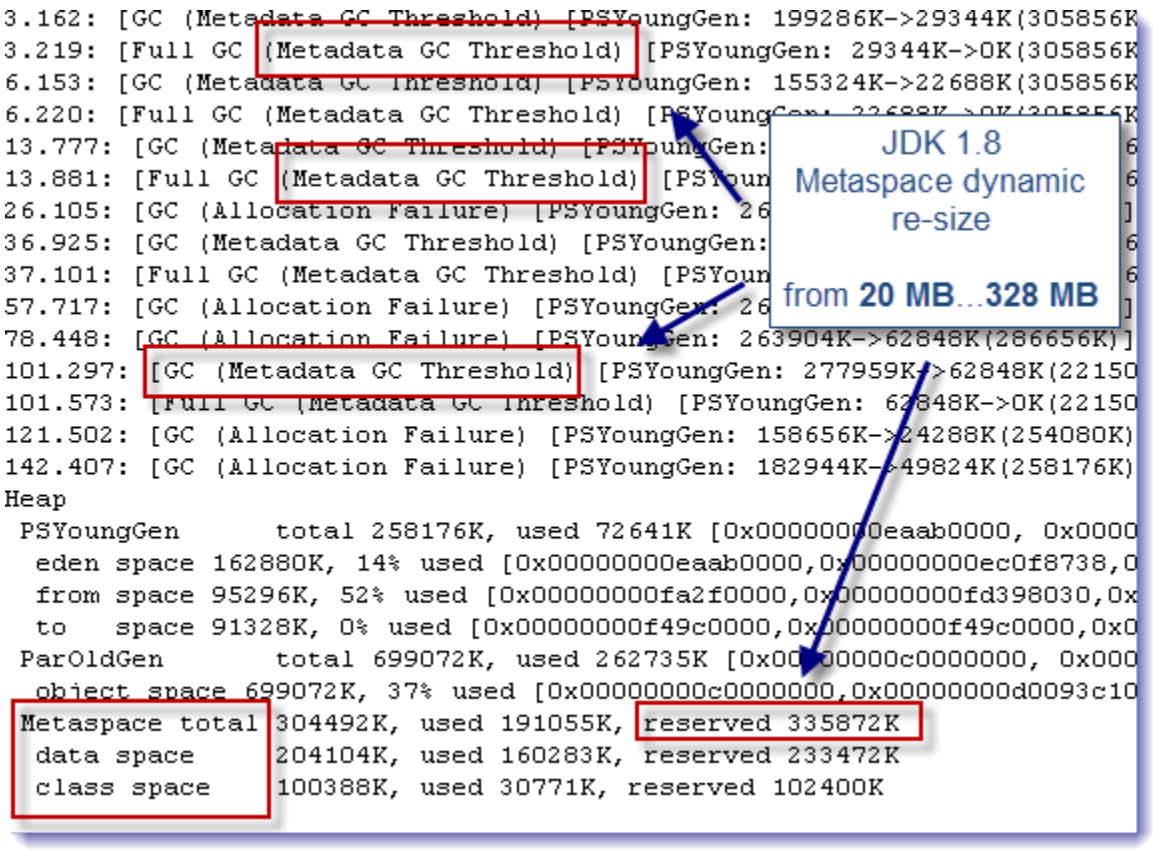


Now let's execute the program using the HotSpot JDK 1.8 JRE.

JDK 1.8 @64-bit – Metaspace dynamic re-size

- Java program with 50K configured iterations
- Java heap space of 1024 MB
- Java Metaspace space: unbounded (default)






```
3.162: [GC (Metadata GC Threshold) [PSYoungGen: 199286K->29344K(305856K
3.219: [Full GC (Metadata GC Threshold) [PSYoungGen: 29344K->0K(305856K
6.153: [GC (Metadata GC Threshold) [PSYoungGen: 155324K->22688K(305856K
6.220: [Full GC (Metadata GC Threshold) [PSYoungGen: 22688K->0K(305856K
13.777: [GC (Metadata GC Threshold) [PSYoungGen:                       6
13.881: [Full GC (Metadata GC Threshold) [PSYoun                       6
26.105: [GC (Allocation Failure) [PSYoungGen: 26                       ]
36.925: [GC (Metadata GC Threshold) [PSYoungGen:                       6
37.101: [Full GC (Metadata GC Threshold) [PSYoun                       6
57.717: [GC (Allocation Failure) [PSYoungGen: 26                       ]
78.448: [GC (Allocation Failure) [PSYoungGen: 263904K->62848K(286656K)]
101.297: [GC (Metadata GC Threshold) [PSYoungGen: 277959K->62848K(22150
101.573: [Full GC (Metadata GC Threshold) [PSYoungGen: 62848K->0K(22150
121.502: [GC (Allocation Failure) [PSYoungGen: 158656K->24288K(254080K)
142.407: [GC (Allocation Failure) [PSYoungGen: 182944K->49824K(258176K)
Heap
 PSYoungGen      total 258176K, used 72641K [0x00000000eaab0000, 0x0000
  eden space 162880K, 14% used [0x00000000eaab0000,0x00000000ec0f8738,0
  from space 95296K, 52% used [0x00000000fa2f0000,0x00000000fd398030,0x
  to   space 91328K, 0% used [0x00000000f49c0000,0x00000000f49c0000,0x0
 ParOldGen       total 699072K, used 262735K [0x00000000c0000000, 0x000
  object space 699072K, 37% used [0x00000000c0000000,0x00000000d0093c10
 Metaspace total 304492K, used 191055K, reserved 335872K
  data space     204104K, used 160283K, reserved 233472K
  class space    100388K, used 30771K, reserved 102400K
```

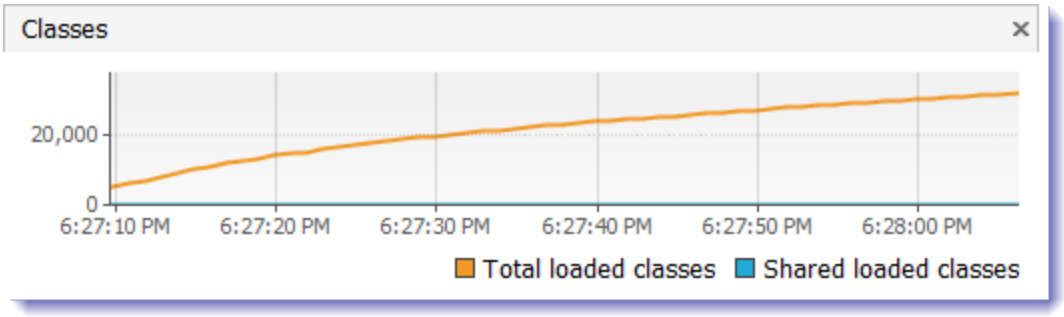
As you can see from the verbose GC output, the JVM Metaspace did expand dynamically from 20 MB up to 328 MB of reserved native memory in order to honor the increased class metadata memory footprint from our Java program. We could also observe garbage collection events in the attempt by the JVM to destroy any dead class or classloader object. Since our Java program is leaking, the JVM had no choice but to dynamically expand the Metaspace memory space.

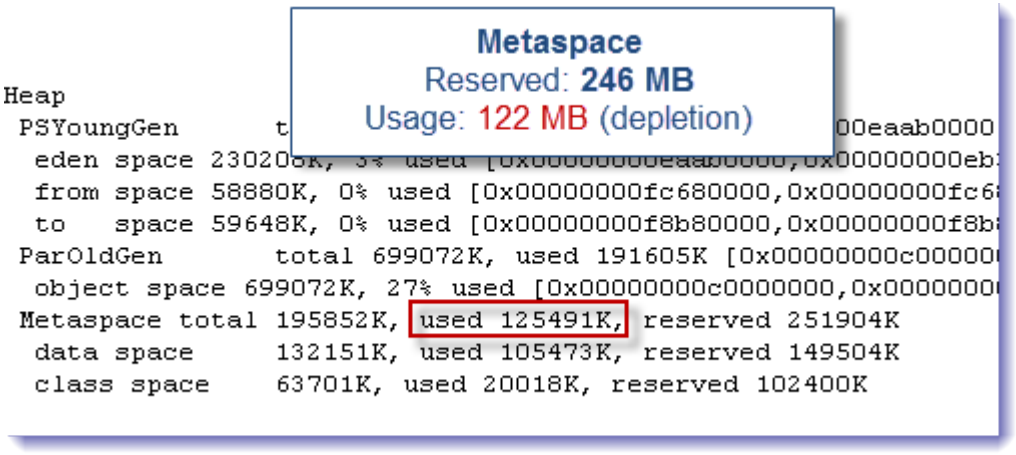
The program was able to run its 50K of iterations with no OOM event and loaded 50K+ Classes.

Let's move to our last testing scenario.

JDK 1.8 @64-bit – Metaspace depletion

- Java program with 50K configured iterations
- Java heap space of 1024 MB
- Java Metaspace space: 128 MB (-XX:MaxMetaspaceSize=128m)





As you can see form JVisualVM, the Metaspace depletion was reached after loading about 30K+ classes; very similar to the run with the JDK 1.7. We can also see this from the program and GC output. Another interesting observation is that the native memory footprint reserved was twice as much as the maximum size specified. This may indicate some opportunities to fine tune the Metaspace resize policy, if possible, in order to avoid native memory waste.

Capping the Metaspace at 128 MB like we did for the baseline run with JDK 1.7 did not allow us to complete the 50K iterations of our program. A new OOM error was thrown by the JVM. The above OOM event was thrown by the JVM from the Metaspace following a memory allocation failure.

Final Words on Metaspace

The current observations definitely indicate that proper monitoring & tuning will be required in order to stay away from problems such as excessive Metaspace GC or OOM conditions triggered from our last testing scenario.

HPROF - Memory leak analysis with Eclipse Memory Analyzer Tool (MAT)

In this section, we will show you how you can analyze a JVM memory leak problem by generating and analyzing a HotSpot JVM HPROF Heap Dump file.

A real life case study will be used for that purpose: Weblogic 9.2 memory leak affecting the Weblogic Admin server.

Environment specifications

- Java EE server: Oracle Weblogic Server 9.2 MP1
- Middleware OS: Solaris 10
- Java VM: Sun HotSpot 1.5.0_22
- Platform type: Middle tier

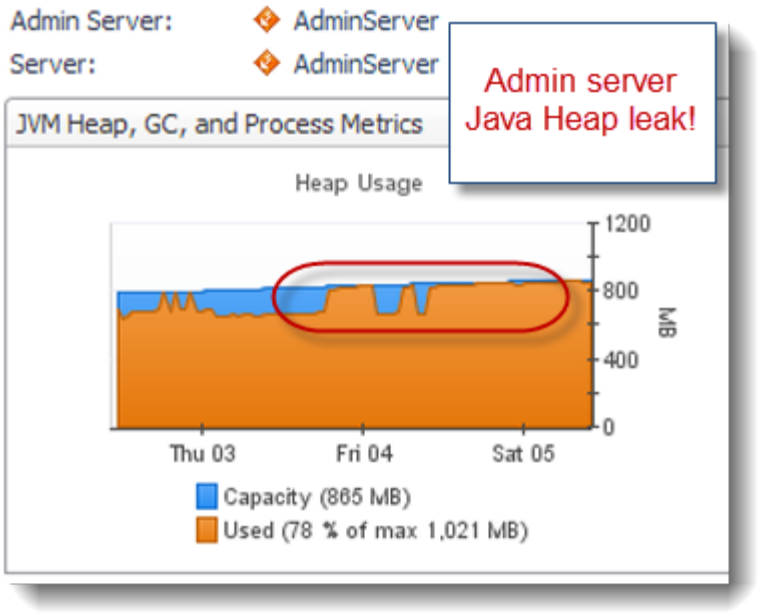
Monitoring and troubleshooting tools

- Quest Foglight (JVM and garbage collection monitoring)
- jmap (hprof / Heap Dump generation tool)
- Memory Analyzer 1.1 via IBM support assistant (hprof Heap Dump analysis)
- Platform type: Middle tier

Step #1 - WLS 9.2 Admin server JVM monitoring and leak confirmation

The Quest Foglight Java EE monitoring tool was quite useful to identify a Java Heap leak from our Weblogic Admin server. As you can see below, the Java Heap memory is growing over time.

If you are not using any monitoring tool for your Weblogic environment, my recommendation to you is to at least enable verbose:gc of your HotSpot VM. Please visit the Java 7 verbose:gc tutorial below on this subject for more detailed instructions.



Step #2 - Generate a Heap Dump from your leaking JVM

Following the discovery of a JVM memory leak, the goal is to generate a Heap Dump file (binary format) by using the Sun JDK jmap utility.

*** Please note that jmap Heap Dump generation will cause your JVM to become unresponsive so please ensure that no more traffic is sent to your affected / leaking JVM before running the jmap utility ***

```
<JDK HOME>/bin/jmap -heap:format=b <Java VM PID>
```

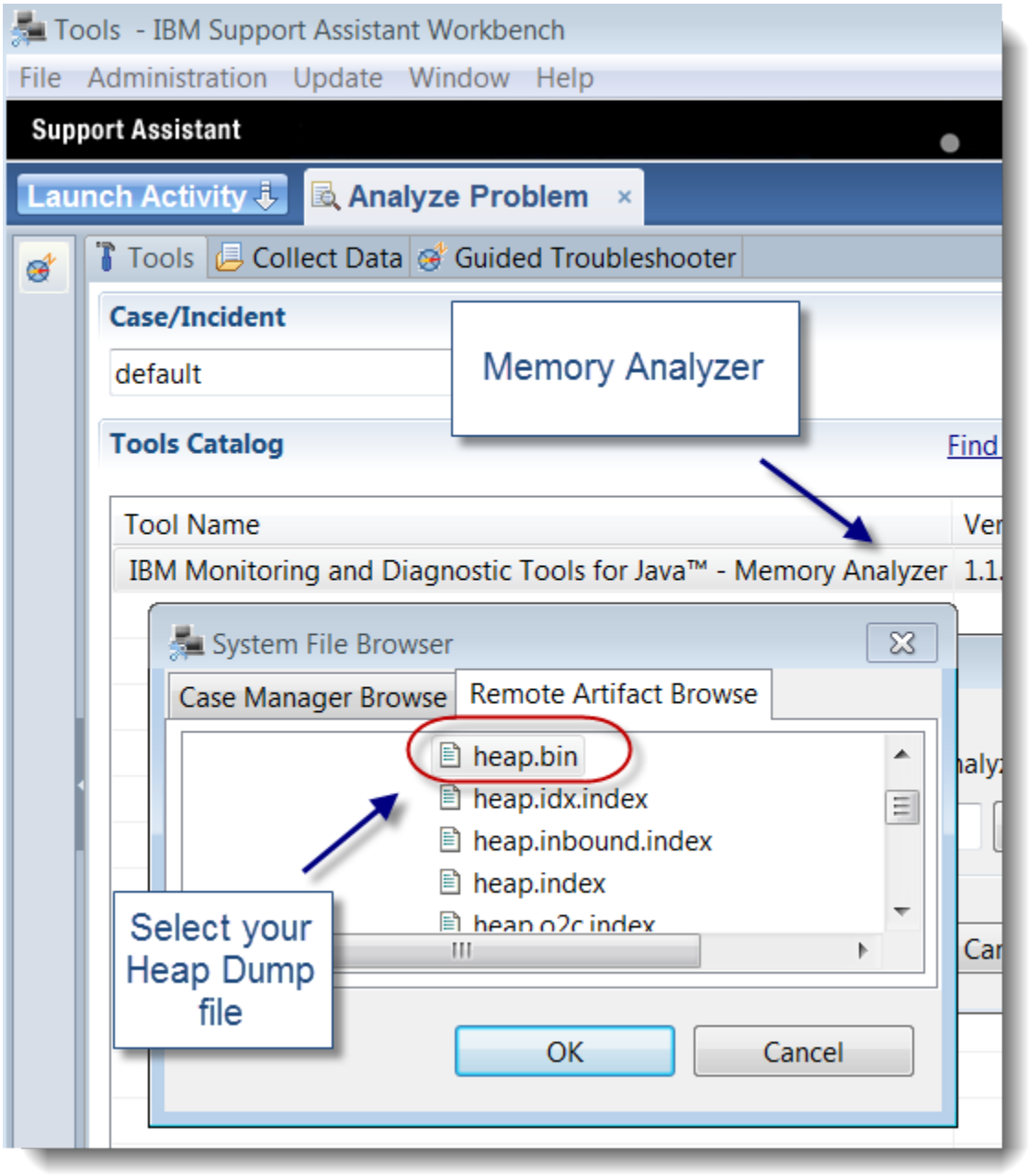
This command will generate a Heap Dump binary file (heap.bin) of your leaking JVM. The size of the file and elapsed time of the generation process will depend of your JVM size and machine specifications / speed.

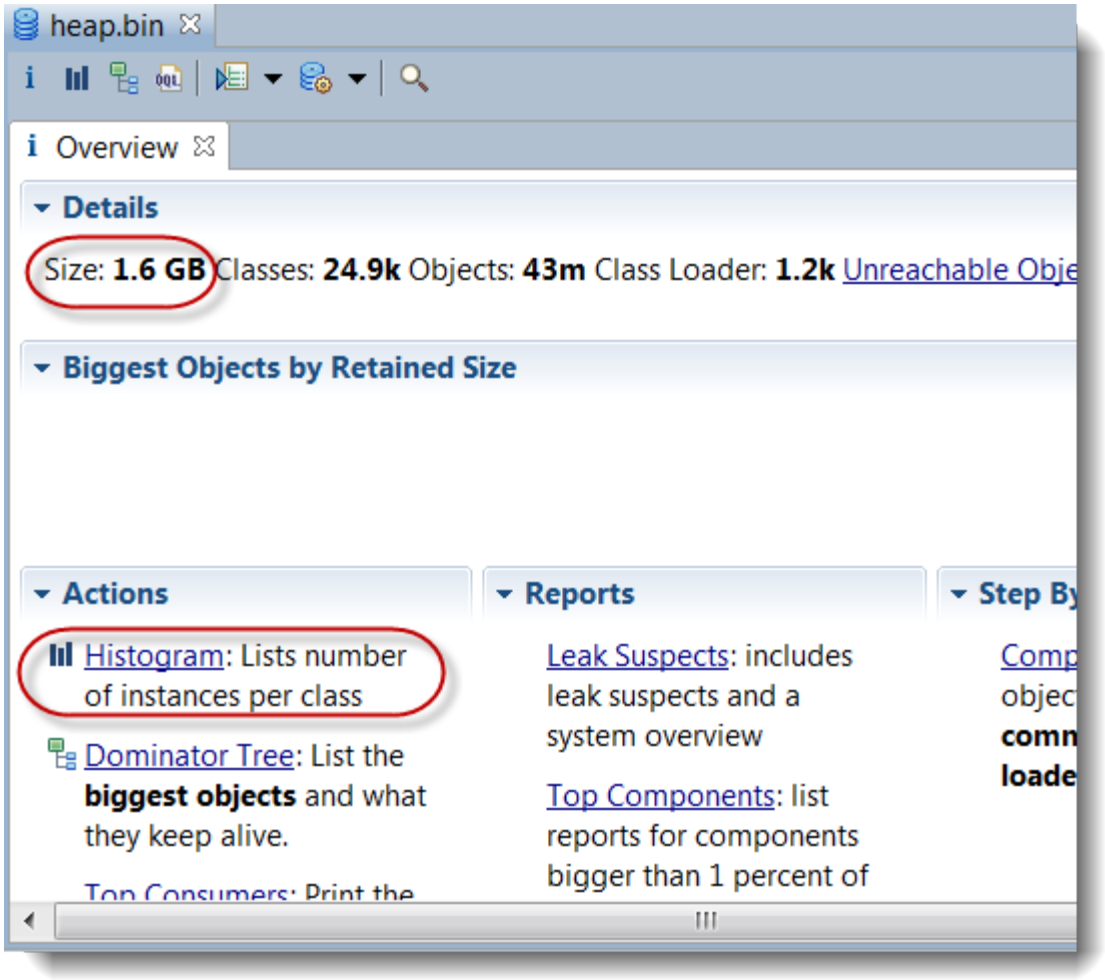
For our case study, a binary Heap Dump file of ~2GB was generated in about 1 hour elapsed time.

Sun HotSpot 1.5/1.6/1.7 Heap Dump file will also be generated automatically as a result of a OutOfMemoryError and by adding -XX:+HeapDumpOnOutOfMemoryError in your JVM start-up arguments.

Step #3 - Load your Heap Dump file in Memory Analyzer tool

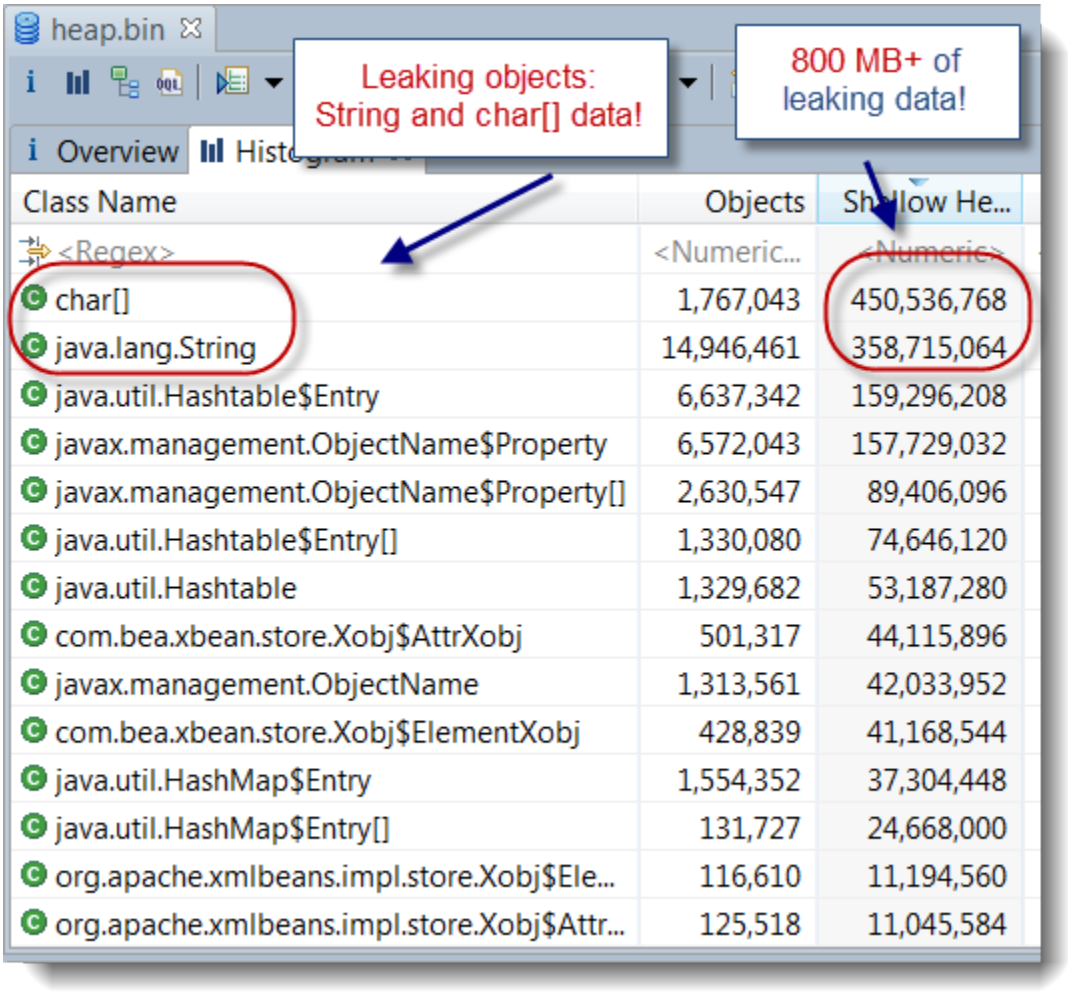
It is now time to load your Heap Dump file in the Memory Analyzer tool. The loading process will take several minutes depending of the size of your Heap Dump and speed of your machine.


Tools - IBM Support Assistant Workbench
File Administration Update Window Help
Support Assistant
Launch Activity
Analyze Problem
Tools
Collect Data
Guided Troubleshooter
Case/Incident
default
Memory Analyzer
Tools Catalog
Find
Tool Name
IBM Monitoring and Diagnostic Tools for Java™ - Memory Analyzer
System File Browser
Case Manager Browse
Remote Artifact Browse
heap.bin
heap.idx.index
heap.inbound.index
heap.index
Select your Heap Dump file
OK
Cancel

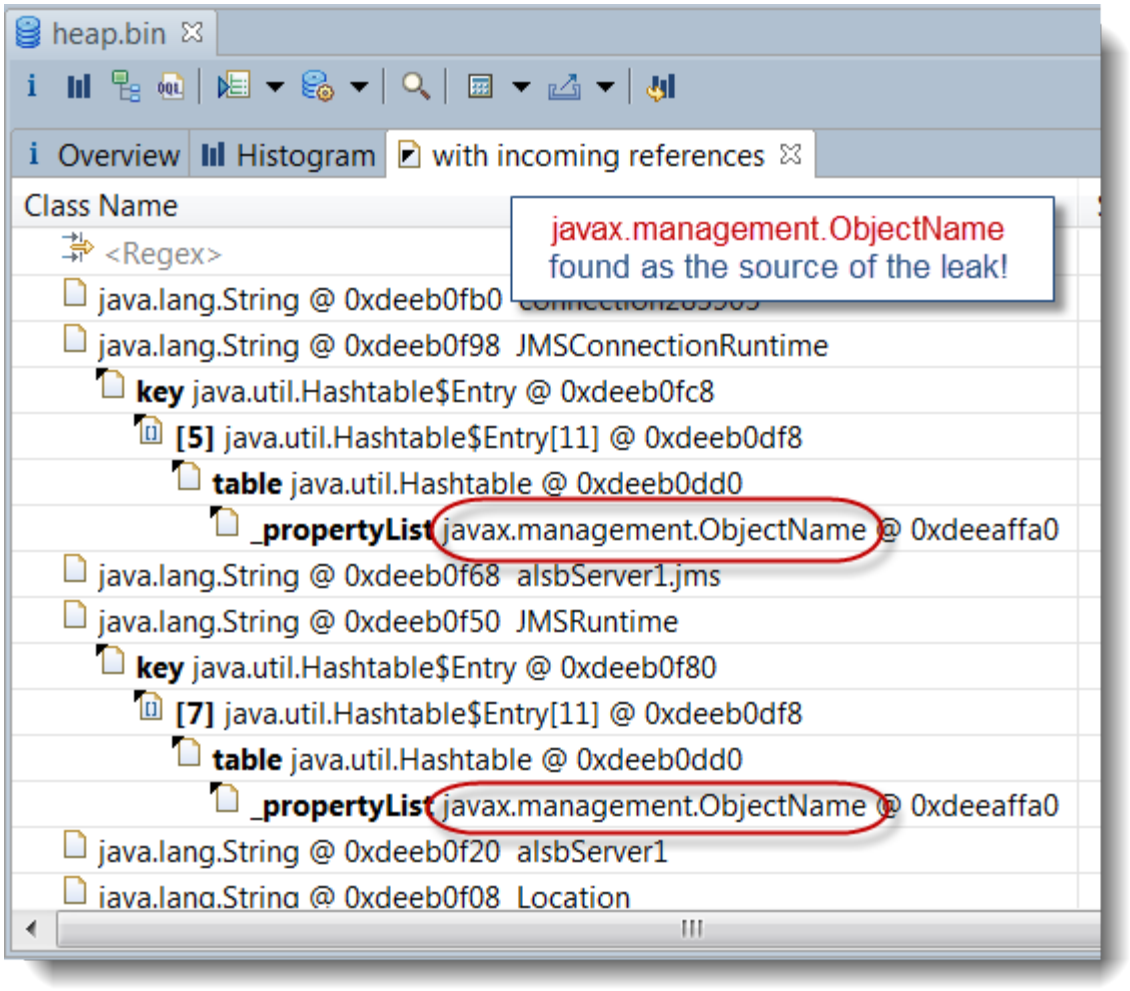


Step #4 - Analyze your Heap Dump

The Memory Analyzer provides you with many features, including a Leak Suspect report. For this case study, the Java Heap histogram was used as a starting point to analyze the leaking objects and the source.



For our case study, java.lang.String and char[] data were found as the leaking Objects. Now question is what is the source of the leak e.g. references of those leaking Objects. Simply right click over your leaking objects and select >> List Objects > with incoming references.



As you can see, javax.management.ObjectName objects were found as the source of the leaking String & char[] data. The Weblogic Admin server is communicating and pulling stats from its managed servers via MBeans / JMX which create javax.management.ObjectName for any MBean object type. Now question is why Weblogic 9.2 is not releasing properly such Objects…

Root cause: Weblogic javax.management.ObjectName leak!

Following our Heap Dump analysis, a review of the Weblogic known issues was performed which did reveal the following Weblogic 9.2 bug below:

- Weblogic Bug ID: CR327368
- Description: Memory leak of javax.management.ObjectName objects on the Administration Server used to cause OutOfMemory error on the Administration Server.
- Affected Weblogic version(s): WLS 9.2
- Fixed in: WLS 10 MP1

This finding was quite conclusive given the perfect match of our Heap Dump analysis, WLS version

and this known problem description.

Hopefully, this tutorial along with the case study has helped you understand how you can pinpoint the source of a Java Heap leak using jmap and the Memory Analyzer tool.

JVM verbose GC output tutorial

This section will provide you with a detailed tutorial on how to enable and read Java 7 HotSpot VM verbose gc output data.

I recommend that you compile and run the sample program on your end as well.

We also recently created a Java verbose GC tutorial video explaining this analysis process.

Tutorial specifications and tools

- OS: Windows 7 - 64-bit
- Java VM: Sun Java 7 HotSpot (build 21.0-b17)
- IDE: Eclipse Java EE IDE for Web Developer v4.1

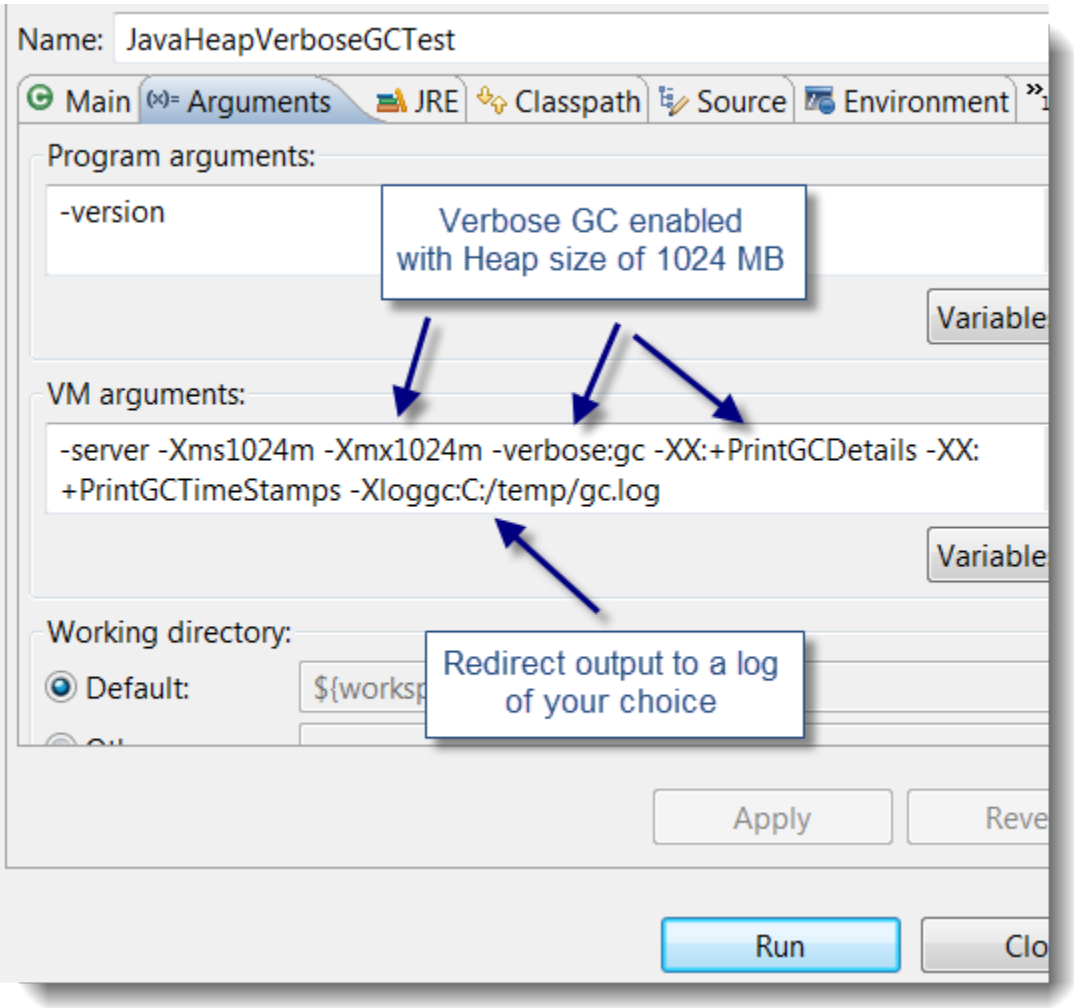
Step #1 - Compile our sample Java program

We created a sample Java program in order to load the Java Heap and trigger an explicit GC in order to generate some interesting verbose GC output. This program is simply loading about 3 million instances of java.lang.String in a static Map data structure and triggers an explicit GC (via System.gc()) followed by the removal of 2 million instances along with a second explicit GC before exiting.

```
package org.ph.javaee.tools.jdk7;

import java.util.Map;
import java.util.HashMap;

/**
* JavaHeapVerboseGCTest
* @author Pierre-Hugues Charbonneau
*
*/
public class JavaHeapVerboseGCTest {

   private static Map<String, String> mapContainer = new HashMap<String, String>();

   /**
   * @param args
   */
   public static void main(String[] args) {

     System.out.println("Java 7 HotSpot Verbose GC Test Program v1.0");
     System.out.println("Author: Pierre-Hugues Charbonneau");
     System.out.println("http://javaeesupportpatterns.blogspot.com/");

     String stringDataPrefix = "stringDataPrefix";

     // Load Java Heap with 3 M java.lang.String instances
     for (int i=0; i<3000000; i++) {
       String newStringData = stringDataPrefix + i;
       mapContainer.put(newStringData, newStringData);
     }

     System.out.println("MAP size: "+mapContainer.size());
     System.gc(); // Explicit GC!

     // Remove 2 M out of 3 M
     for (int i=0; i<2000000; i++) {
       String newStringData = stringDataPrefix + i;
       mapContainer.remove(newStringData);
     }

     System.out.println("MAP size: "+mapContainer.size());
     System.gc();
     System.out.println("End of program!");

   }
}
```

Step #2 - Enable verbose GC via the JVM start-up arguments

The next step is to enable the verbose GC via the JVM start-up arguments and specify a name and location for our GC log file.



Step #3 - Execute our sample Java program

At this point, it is now time to execute our sample program and generate the JVM verbose GC output.

Verbose GC output high level analysis

It is now time to review the generated GC output.

First, let's start with the raw data. As you can see below, the GC output is divided into 3 main sections:

- 5 Minor collections (YoungGen space collections) identified as PSYoungGen
- 2 Major collections (triggered by System.gc()) identified as Full GC (System)
- A detailed Java Heap breakdown of each memory space

```
0.437: [GC [PSYoungGen: 262208K->43632K(305856K)]
 262208K->137900K(1004928K), 0.1396040 secs]
 [Times: user=0.45 sys=0.01, real=0.14 secs]

0.785: [GC [PSYoungGen: 305840K->43640K(305856K)]
 400108K->291080K(1004928K), 0.2197630 secs]
 [Times: user=0.56 sys=0.03, real=0.22 secs]

1.100: [GC [PSYoungGen: 164752K->43632K(305856K)]
 412192K->340488K(1004928K), 0.0878209 secs]
 [Times: user=0.37 sys=0.00, real=0.09 secs]

1.188: [Full GC (System) [PSYoungGen: 43632K->0K(305856K)]
 [PSOldGen: 296856K->340433K(699072K)]
 340488K->340433K(1004928K)
 [PSPermGen: 1554K->1554K(16384K)], 0.4053311 secs]
 [Times: user=0.41 sys=0.00, real=0.40 secs]

1.883: [GC [PSYoungGen: 262208K->16K(305856K)]
 602641K->340449K(1004928K), 0.0326756 secs]
 [Times: user=0.09 sys=0.00, real=0.03 secs]

2.004: [GC [PSYoungGen: 92122K->0K(305856K)]
 432556K->340433K(1004928K), 0.0161477 secs]
 [Times: user=0.06 sys=0.00, real=0.02 secs]

2.020: [Full GC (System) [PSYoungGen: 0K->0K(305856K)]
 [PSOldGen: 340433K->125968K(699072K)]
 340433K->125968K(1004928K)
 [PSPermGen: 1555K->1555K(16384K)], 0.2302415 secs]
 [Times: user=0.23 sys=0.00, real=0.23 secs]

Heap
PSYoungGen total 305856K, used 5244K [0x3dac0000, 0x53010000, 0x53010000)
 eden space 262208K, 2% used [0x3dac0000,0x3dfdf168,0x4dad0000)
 from space 43648K, 0% used [0x4dad0000,0x4dad0000,0x50570000)
 to space 43648K, 0% used [0x50570000,0x50570000,0x53010000)
PSOldGen total 699072K, used 125968K [0x13010000, 0x3dac0000, 0x3dac0000)
 object space 699072K, 18% used [0x13010000,0x1ab140a8,0x3dac0000)
PSPermGen total 16384K, used 1560K [0x0f010000, 0x10010000, 0x13010000)
 object space 16384K, 9% used [0x0f010000,0x0f1960b0,0x10010000)
```

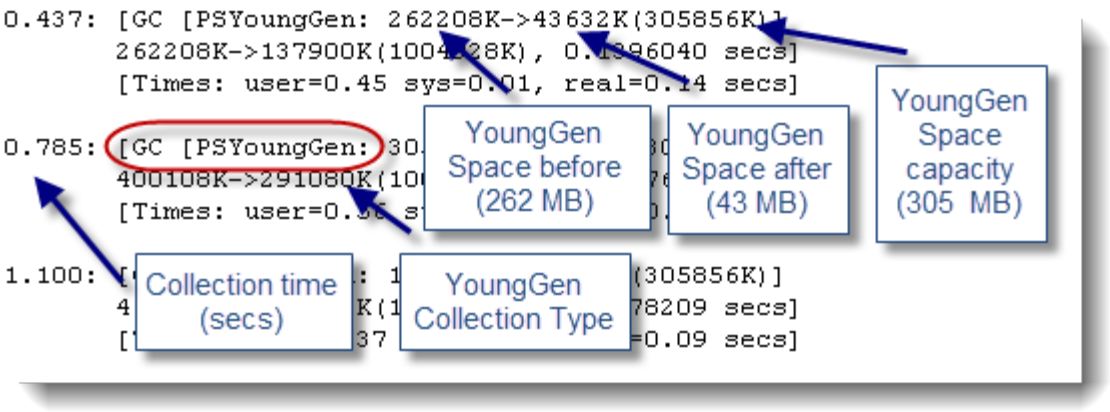
Verbose GC data interpretation and sequence

As you can see from the verbose GC output, the OldGen space was at 340 MB after the initial loading of 3M String instances in our HashMap. It did go down to 126 MB following the removal of 2M String

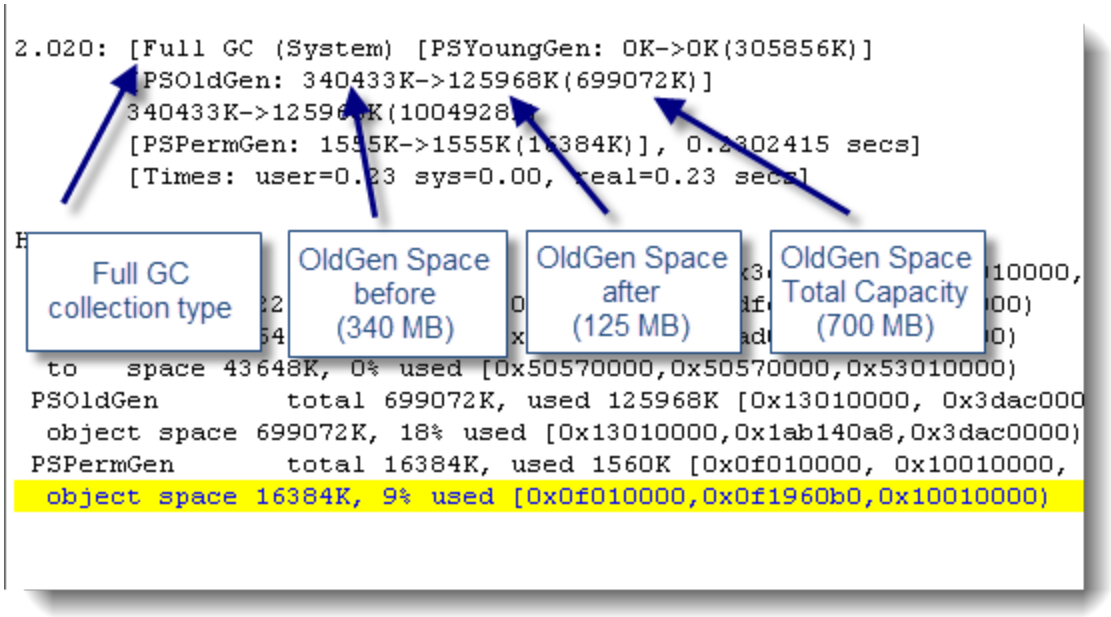
instances.

Now find below explanation and snapshots on how you can read the GC output data in more detail for each Java Heap space.

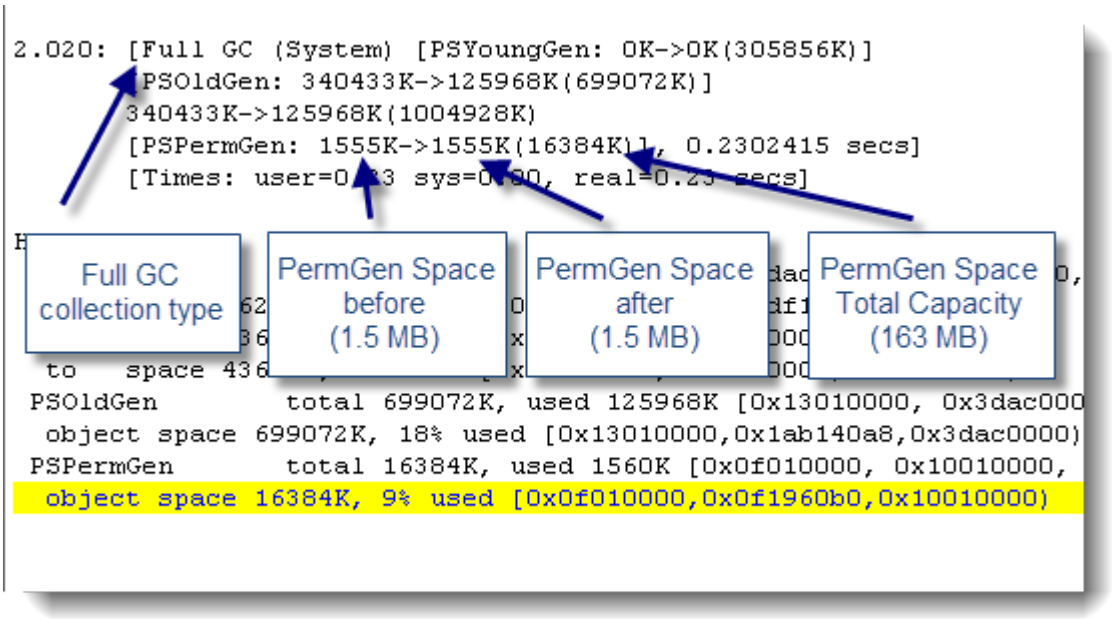
YoungGen space analysis



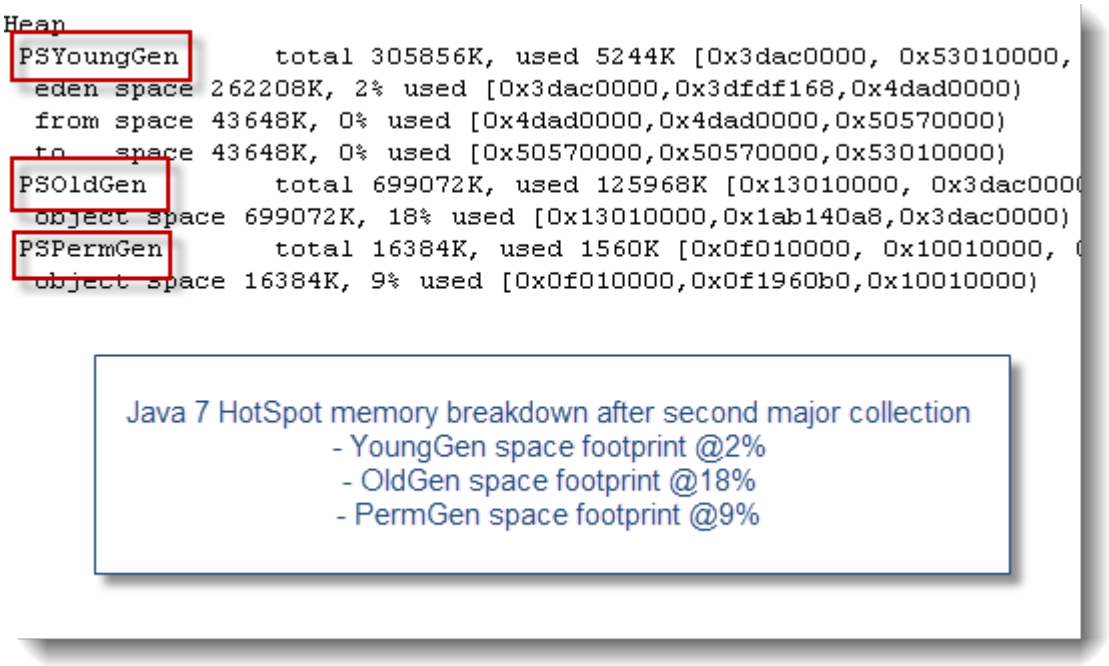
OldGen space analysis



PermGen space analysis



Java Heap breakdown analysis



Hopefully this sample Java program and verbose GC output analysis has helped you understand how to read and interpret this critical data.

Analyzing thread dumps

Introduction to thread dump analysis

This section will teach you how to analyze a JVM Thread Dump and pinpoint the root cause of your problem(s). From my perspective, Thread Dump analysis is the most important skillset to master for any individual involved in Java EE production support. The amount of information that you can derive from Thread Dump snapshots is often much beyond than what you can think of.

You may find complementary training videos here and here.

Before going deeper into Thread Dump analysis and problem patterns, it is very important that you understand the fundamentals.

Java VM overview

The Java virtual machine is really the foundation of any Java EE platform. This is where your middleware and applications are deployed and active.

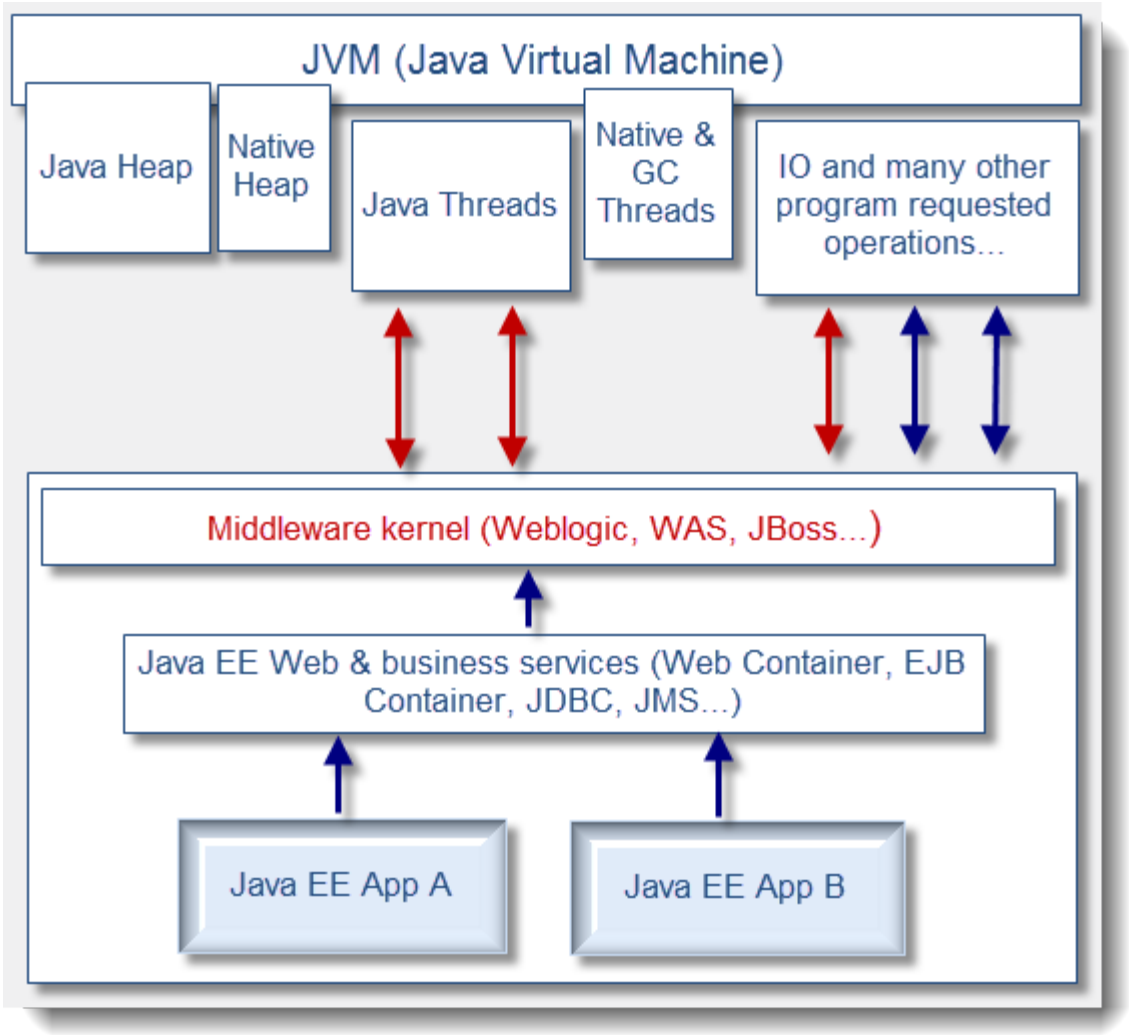
The JVM provides the middleware software and your Java / Java EE program with:

- A runtime environment for your Java / Java EE program (bytecode format).
- Several program features and utilities (IO facilities, data structure, Threads management, security, monitoring etc.).
- Dynamic memory allocation and management via the garbage collector.

Your JVM can reside on many OS (Solaris, AIX, Windows etc.) and depending of your physical server specifications, you can install 1...n JVM processes per physical / virtual server.

JVM and Middleware software interactions

Find below a diagram showing you a high level interaction view between the JVM, middleware and application(s).



This is showing you a typical and simple interaction diagram between the JVM, middleware and application. As you can see, the Threads allocation for a standard Java EE application are done mainly between the middleware kernel itself and JVM (there are some exceptions when application itself or some APIs create Threads directly but this is not common and must be done very carefully).

Also, please note that certain Threads are managed internally within the JVM itself such as GC (garbage collection) Threads in order to handle concurrent garbage collections.

Since most of the Thread allocations are done by the Java EE container, it is important that you understand and recognize the Thread Stack Trace and identify it properly from the Thread Dump data. This will allow you to understand quickly the type of request that the Java EE container is attempting to execute.

From a Thread Dump analysis perspective, you will learn how to differentiate between the different Thread Pools found from the JVM and identify the request type.

JVM Thread Dump - what is it?

A JVM Thread Dump is a snapshot taken at a given time which provides you with a complete listing of all created Java Threads.

Each individual Java Thread found gives you information such as:

- Thread name; often used by middleware vendors to identify the Thread Id along with its associated Thread Pool name and state (running, stuck etc.)

- Thread type & priority ex: `daemon prio=3` *** middleware softwares typically create their Threads as daemon meaning their Threads are running in background; providing services to its user e.g. your Java EE application ***

- Java Thread ID ex: `tid=0x000000011e52a800` *** This is the Java Thread Id obtained via java.lang.Thread.getId() and usually implemented as an auto-incrementing long 1..n***

- Native Thread ID ex: `nid=0x251c`*** Crucial information as this native Thread Id allows you to correlate for example which Threads from an OS perspective are using the most CPU within your JVM etc. ***

- Java Thread State and detail ex: `waiting for monitor entry [0xffffffffea5afb000]`
`java.lang.Thread.State: BLOCKED (on object monitor)`
*** Allows to quickly learn about Thread state and its potential current blocking condition ***

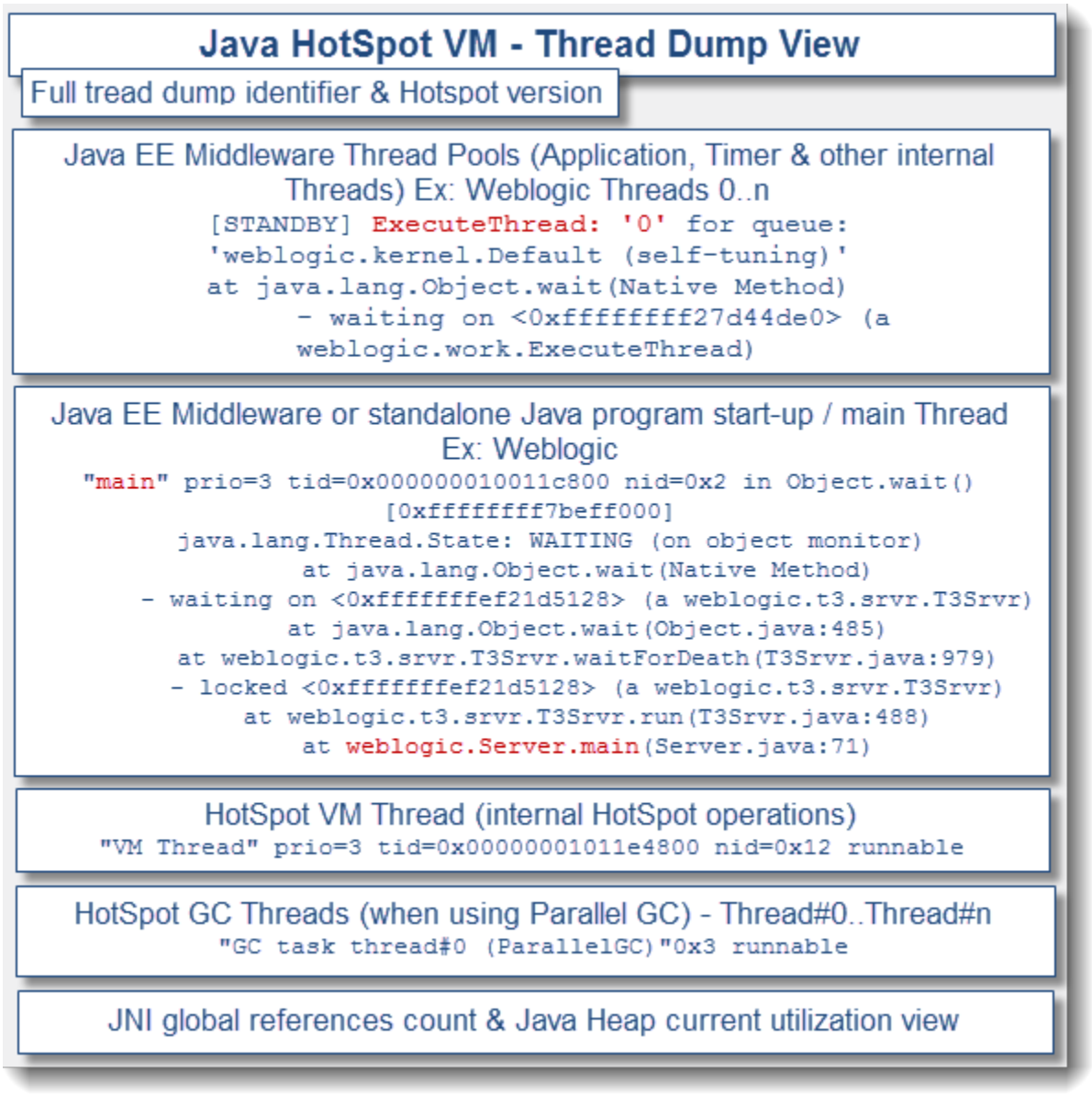
- Java Thread Stack Trace; this is by far the most important data that you will find from the Thread Dump. This is also where you will spent most of your analysis time since the Java Stack Trace provides you with 90% of the information that you need in order to pinpoint root cause of many problem pattern types as you will learn later in the training sessions

- Java Heap breakdown; starting with HotSpot VM 1.6, you will also find at the bottom of the Thread Dump snapshot a breakdown of the HotSpot memory spaces utilization such as your Java Heap (YoungGen, OldGen) & PermGen space. This is quite useful when excessive GC is suspected as a possible root cause so you can do out-of-the-box correlation with Thread data / patterns found

```
Heap
PSYoungGen total 466944K, used 178734K [0xffffffff45c00000, 0xffffffff70800000,
0xffffffff70800000)
 eden space 233472K, 76% used
[0xffffffff45c00000,0xffffffff50ab7c50,0xffffffff54000000)
 from space 233472K, 0% used
[0xffffffff62400000,0xffffffff62400000,0xffffffff70800000)
 to space 233472K, 0% used
[0xffffffff54000000,0xffffffff54000000,0xffffffff62400000)
PSOldGen    total 1400832K, used 1400831K [0xfffffffef0400000,
0xffffffff45c00000, 0xffffffff45c00000)
 object space 1400832K, 99% used
[0xfffffffef0400000,0xffffffff45bfffb8,0xffffffff45c00000)
PSPermGen    total 262144K, used 248475K [0xfffffffed0400000,
0xfffffffee0400000, 0xfffffffef0400000)
 object space 262144K, 94% used
[0xfffffffed0400000,0xfffffffedf6a6f08,0xfffffffee0400000)
```

Thread Dump breakdown overview

In order for you to better understand, find below a diagram showing you a visual breakdown of a HotSpot VM Thread Dump and its common Thread Pools found:



As you can there are several pieces of information that you can find from a HotSpot VM Thread Dump. Some of these pieces will be more important than others depending of your problem pattern.

For now, find below a detailed explanation for each Thread Dump section as per our sample HotSpot Thread Dump:

Full thread dump identifier
This is basically the unique keyword that you will find in your middleware / standalong Java standard output log once you generate a Thread Dump (ex: via kill -3 <PID> for UNIX). This is the beginning of the Thread Dump snapshot data.

```
Full thread dump Java HotSpot(TM) 64-Bit Server VM (20.0-b11 mixed mode):
```

Java EE middleware, third party & custom application Threads

This portion is the core of the Thread Dump and where you will typically spend most of your analysis time. The number of Threads found will depend on your middleware software that you use, third party libraries (that might have its own Threads) and your application (if creating any custom Thread, which is generally not a best practice).

In our sample Thread Dump, Weblogic is the middleware used. Starting with Weblogic 9.2, a self-tuning Thread Pool is used with unique identifier "'weblogic.kernel.Default (self-tuning)"

```
"[STANDBY] ExecuteThread: '414' for queue: 'weblogic.kernel.Default (self-
tuning)'" daemon prio=3 tid=0x000000010916a800 nid=0x2613 in Object.wait()
[0xffffffffe9edff000]
 java.lang.Thread.State: WAITING (on object monitor)
   at java.lang.Object.wait(Native Method)
   - waiting on <0xffffffff27d44de0> (a weblogic.work.ExecuteThread)
   at java.lang.Object.wait(Object.java:485)
   at weblogic.work.ExecuteThread.waitForRequest(ExecuteThread.java:160)
   - locked <0xffffffff27d44de0> (a weblogic.work.ExecuteThread)
   at weblogic.work.ExecuteThread.run(ExecuteThread.java:181)
```

HotSpot VM Thread

This is an internal Thread managed by the HotSpot VM in order to perform internal native operations. Typically you should not worry about this one unless you see high CPU (via Thread Dump & prstat / native Thread id correlation).

```
"VM Periodic Task Thread" prio=3 tid=0x0000000101238800 nid=0x19 waiting on
condition
```

HotSpot GC Thread

When using HotSpot parallel GC (quite common these days when using multi physical cores hardware), the HotSpot VM create by default or as per your JVM tuning a certain # of GC Threads. These GC Threads allow the VM to perform its periodic GC cleanups in a parallel manner, leading to an overall reduction of the GC time; at the expense of increased CPU utilization.

```
"GC task thread#0 (ParallelGC)" prio=3 tid=0x0000000100120000 nid=0x3 runnable
"GC task thread#1 (ParallelGC)" prio=3 tid=0x0000000100131000 nid=0x4 runnable
```

This is crucial data as well since when facing GC related problems such as excessive GC, memory leaks etc, you will be able to correlate any high CPU observed from the OS / Java process(es) with these Threads using their native id value (nid=0x3).

JNI global references count

JNI (Java Native Interface) global references are basically Object references from the native code to a Java object managed by the Java garbage collector. Its role is to prevent collection of an object that is still in use by native code but technically with no "live" references in the Java code.

It is also important to keep an eye on JNI references in order to detect JNI related leaks. This can happen if you program use JNI directly or using third party tools like monitoring tools which are prone to native memory leaks.

```
JNI global references: 1925
```

Java Heap utilization view

This data was added back to JDK 1 .6 and provides you with a short and fast view of your HotSpot Heap. I find it quite useful when troubleshooting GC related problems along with HIGH CPU since you get both Thread Dump & Java Heap in a single snapshot allowing you to determine (or to rule out) any pressure point in a particular Java Heap memory space along with current Thread computing currently being done at that time. As you can see in our sample Thread Dump, the Java Heap OldGen is maxed out!

```
Heap
 PSYoungGen      total 466944K, used 178734K [0xffffffff45c00000,
0xffffffff70800000, 0xffffffff70800000)
 eden space 233472K, 76% used
[0xffffffff45c00000,0xffffffff50ab7c50,0xffffffff54000000)
 from space 233472K, 0% used
[0xffffffff62400000,0xffffffff62400000,0xffffffff70800000)
 to   space 233472K, 0% used
[0xffffffff54000000,0xffffffff54000000,0xffffffff62400000)
 PSOldGen        total 1400832K, used 1400831K [0xfffffffef0400000,
0xffffffff45c00000, 0xffffffff45c00000)
 object space 1400832K, 99% used
[0xfffffffef0400000,0xffffffff45bfffb8,0xffffffff45c00000)
 PSPermGen       total 262144K, used 248475K [0xfffffffed0400000,
0xfffffffee0400000, 0xfffffffef0400000)
 object space 262144K, 94% used
[0xfffffffed0400000,0xfffffffedf6a6f08,0xfffffffee0400000)
```

In order for you to quickly identify a problem pattern from a Thread Dump, you first need to understand how to read a Thread Stack Trace and how to get the “story” right. This means that if I ask you to tell me what the Thread #38 is doing; you should be able to precisely answer; including if Thread Stack Trace is showing a healthy (normal) vs. hang condition.

Java Stack Trace revisited

Most of you are familiar with Java stack traces. This is typical data that we find from server and application log files when a Java Exception is thrown. In this context, a Java stack trace is giving us the code execution path of the Thread that triggered the Java Exception such as a

java.lang.NoClassDefFoundError, java.lang.NullPpointerException etc. Such code execution path allows us to see the different layers of code that ultimately lead to the Java Exception.

Java stack traces must always be read from bottom-up:

- The line at the bottom will expose the originator of the request such as a Java / Java EE container Thread.
- The first line at the top of the stack trace will show you the Java class where that last Exception got triggered.

Let's go through this process via a simple example. We created a sample Java program simply executing some Class methods calls and throwing an Exception. The program output generated is as per below:

```
JavaStrackTraceSimulator
Author: Pierre-Hugues Charbonneau
http://javaeesupportpatterns.blogspot.com

Exception in thread "main" java.lang.IllegalArgumentException:
  at org.ph.javaee.training.td.Class2.call(Class2.java:12)
  at org.ph.javaee.training.td.Class1.call(Class1.java:14)
  at org.ph.javaee.training.td.JavaSTSimulator.main(JavaSTSimulator.java:20)
```

- Java program JavaSTSimulator is invoked (via the "main" Thread)
- The simulator then invokes method call() from Class1
- Class1 method call() then invokes Class2 method call()
- Class2 method call()throws a Java Exception: java.lang.IllegalArgumentException
- The Java Exception is then displayed in the log / standard output

As you can see, the code execution path that lead to this Exception is always displayed from bottom-up.

The above analysis process should be well known for any Java programmer. What you will see next is that the Thread Dump Thread stack trace analysis process is very similar to above Java stack trace analysis.

Thread Dump: Thread Stack Trace analysis

Thread Dump generated from the JVM provides you with a code level execution snapshot of all the "created" Threads of the entire JVM process. Created Threads does not mean that all these Threads are actually doing something. In a typical Thread Dump snapshot generated from a Java EE container JVM:

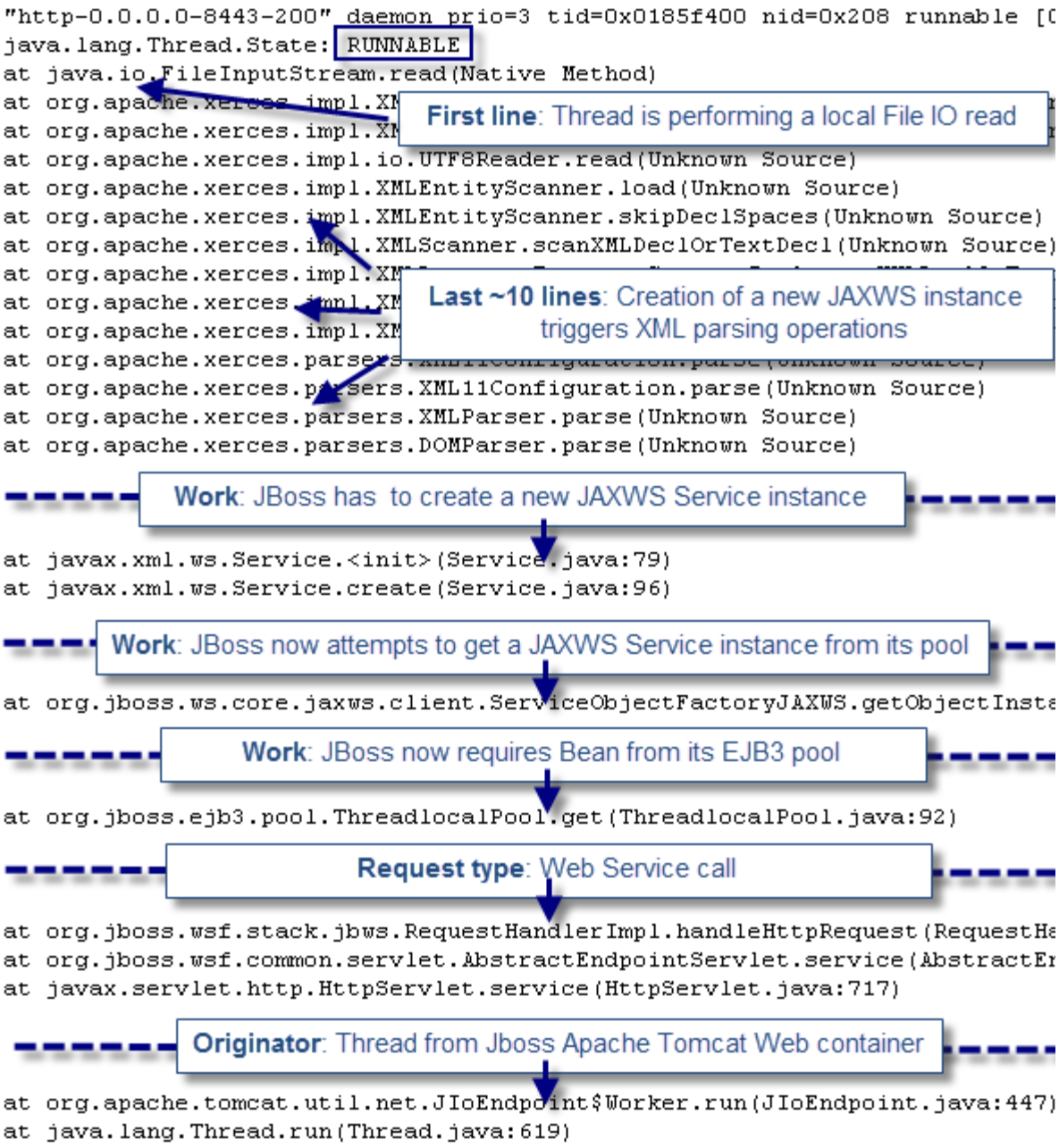
- Some Threads could be performing raw computing tasks such as XML parsing, IO / disk access etc.
- Some Threads could be waiting for some blocking IO calls such as a remote Web Service call,

a DB / JDBC query etc.
- Some Threads could be involved in garbage collection at that time e.g. GC Threads
- Some Threads will be waiting for some work to do (Threads not doing any work typically go in wait() state)
- Some Threads could be waiting for some other Threads work to complete e.g. Threads waiting to acquire a monitor lock (synchronized block{}) on some objects

A Thread stack trace provides you with a snapshot of its current execution. The first line typically includes native information of the Thread such as its name, state, address etc. The current execution stack trace has to be read from bottom-up. Please follow the analysis process below. The more experience you get with Thread Dump analysis, the faster you will able to read and identify very quickly the work performed by each Thread:

- Start to read the Thread stack trace from the bottom.
- First, identify the originator (Java EE container Thread, custom Thread, GC Thread, JVM internal Thread, standalone Java program "main" Thread etc.).
- The next step is to identify the type of request the Thread is executing (WebApp, Web Service, JMS, Remote EJB (RMI), internal Java EE container etc.).
- The next step is to identify from the execution stack trace your application module(s) involved the actual core work the Thread is trying to perform. The complexity of analysis will depend of the layers of abstraction of your middleware environment and application.
- The next step is to look at the last ~10-20 lines prior to the first line. Identify the protocol or work the Thread is involved with e.g. HTTP call, Socket communication, JDBC or raw computing tasks such as disk access, class loading etc.
- The next step is to look at the first line. The first line usually tells a LOT on the Thread state since it is the current piece of code executed at the time you took the snapshot.
- The combination of the last 2 steps is what will give you the core of information to conclude of what work and / or hanging condition the Thread is involved with.

Now find below a visual breakdown of the above steps using a real example of a Thread Dump Thread stack trace captured from a JBoss 5 production environment. In this example, many Threads were showing a similar problem pattern of excessive IO when creating new instances of JAX-WS Service instances.



As you can see, the last 10 lines along with the first line will tell us what hanging or slow condition the Thread is involved with, if any. The lines from the bottom will give us detail of the originator and type of request.

Java Thread CPU analysis on Windows

This section will provide you with a tutorial on how you can quickly pinpoint the Java thread

contributors to a high CPU problem on the Windows OS. Windows, like other OS such as Linux, Solaris & AIX allow you to monitor the CPU utilization at the process level but also for individual Thread executing a task within a process.

For this tutorial, we created a simple Java program that will allow you to learn this technique in a step by step manner.

Troubleshooting tools

The following tools will be used below for this tutorial:

- Windows Process Explorer (to pinpoint high CPU Thread contributors)
- JVM Thread Dump (for Thread correlation and root cause analysis at code level)

High CPU simulator Java program

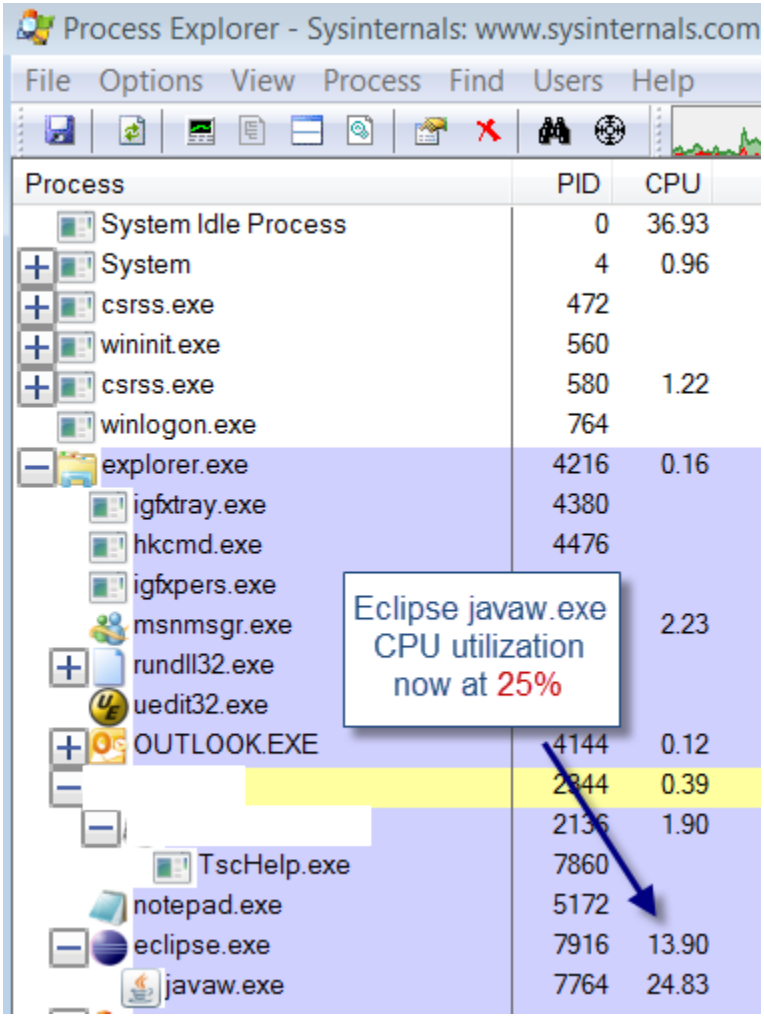
The simple program below is simply looping and creating new String objects. It will allow us to perform this CPU per Thread analysis. I recommend that you import it in an IDE of your choice e.g. Eclipse and run it from there. You should observe an increase of CPU on your Windows machine as soon as you execute it.

```
package org.ph.javaee.tool.cpu;

/**
 * HighCPUSimulator
 * @author Pierre-Hugues Charbonneau
 * http://javaeesupportpatterns.blogspot.com
 *
 */
public class HighCPUSimulator {

	private final static int NB_ITERATIONS = 500000000;

	// ~1 KB data footprint
	private final static String DATA_PREFIX =
"datadatadatadatadatadatadatadatadatadatadatadatadatadatadatadatadatadatadatadata
datadatadatadatadatadatadatadatadatadatadatadatadatadatadatadatadatadatadatadatad
atadatadatadatadatadatadatadatadatadatadatadatadatadatadatadatadatadatadatadatada
tadatadatadatadatadatadatadatadatadatadatadatadatadatadatadatadatadatadatadatadat
adatadatadatadatadatadatadatadatadatadatadatadatadatadatadatadatadatadatadatadata
datadatadatadatadatadatadatadatadatadatadatadatadatadata";

	public static void main(String[] args) {

		System.out.println("HIGH CPU Simulator 1.0");
		System.out.println("Author: Pierre-Hugues Charbonneau");

System.out.println("http://javaeesupportpatterns.blogspot.com/");

		try {

			for (int i = 0; i < NB_ITERATIONS; i++) {

				// Perform some String manipulations
to slowdown and expose looping process...
				String data = DATA_PREFIX + i;

			}

		} catch (Throwable any) {
			System.out.println("Unexpected Exception! "
any.getMessage()
				+ " [" + any +
"]");

		}

		System.out.println("HighCPUSimulator done!");
	}

}
```

Step #1 - Launch Process Explorer

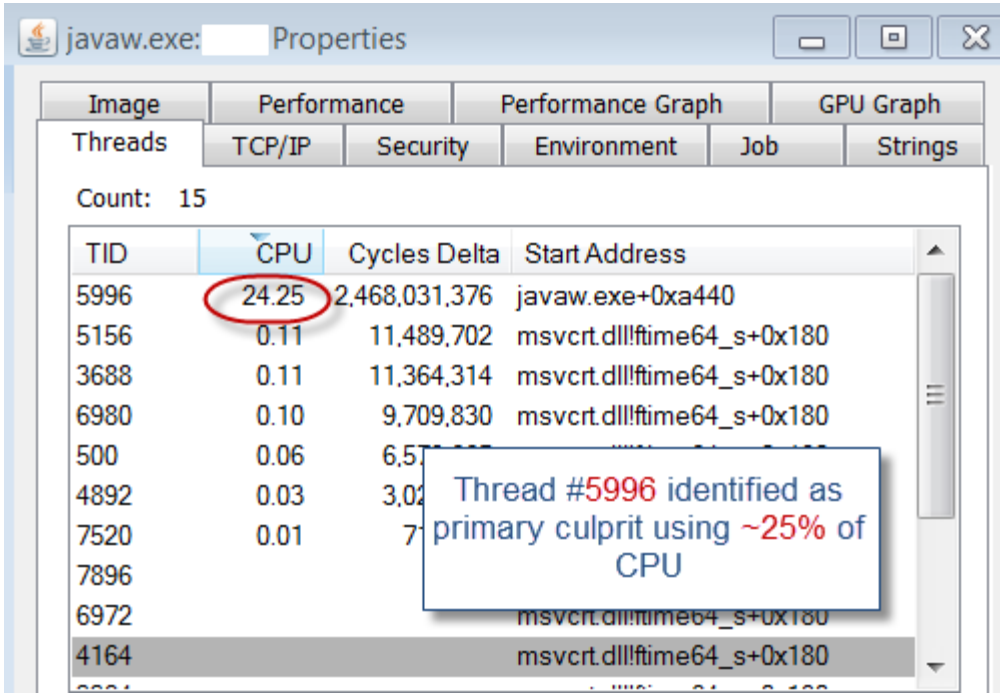
The Process Explorer tool visually shows the CPU usage dynamically. It is good for live analysis. If you need historical data on CPU per Thread then you can also use Windows perfmon with % Processor Time & Thread Id data counters. You can download Process Explorer from the link here.

In our example, you can see that the Eclipse javaw.exe process is now using ~25% of total CPU utilization following the execution of our sample program.



Step #2 - Launch Process Explorer Threads view

The next step is to display the Threads view of the javaw.exe process. Simply right click on the javaw.exe process and select Properties. The Threads view will be opened as per below snapshot:



- The first column is the Thread Id (decimal format)
- The second column is the CPU utilization % used by each Thread
- The third column is also another counter indicating if Thread is running on the CPU

In our example, we can see our primary culprit is Thread Id #5996 using ~ 25% of CPU.

Step #3 - Generate a JVM Thread Dump

At this point, Process Explorer will no longer be useful. The goal was to pinpoint one or multiple Java Threads consuming most of the Java process CPU utilization which is what we achieved. In order to go the next level in your analysis you will need to capture a JVM Thread Dump. This will allow you to correlate the Thread Id with the Thread Stack Trace so you can pinpoint that type of processing is consuming such high CPU.

JVM Thread Dump generation can be done in a few manners. If you are using JRockit VM you can simply use the jrcmd tool as per below example:

```
..\jrockit_160_24_D1.1.2-4\bin>jrcmd 4540 print_threads
4540:

===== FULL THREAD DUMP ===============
Wed Apr 25 10:28:44 2012
Oracle JRockit(R) R28.1.3-11-141760-1.6.0_24-20110301-1429-windows-

"Main Thread" id=1 idx=0x4 tid=5996 prio=5 alive, native_blocked
 at org/ph/javaee/tool/cpu/HighCPUSimulator.main
    (HighCPUSimulator.java:31)
 at jrockit/vm/RNI.c2java(IIIII)V(Native Method)
 -- end of trace

"(Signal Handler)" id=2 idx=                                    e_block

"(OC Main Thread)" id=3 idx=                                    e_wait:

"(GC Worker Thread 1)" id=? idx=0x10 tid=7484 prio=5 alive, daemon
```

Sample program main Java Thread (tid=5996) identified as primary culprit

Once you have the Thread Dump data, simply search for the Thread Id and locate the Thread Stack Trace that you are interested in.

For our example, the Thread "Main Thread" which was fired from Eclipse got exposed as the primary culprit which is exactly what we wanted to demonstrate.

```
"Main Thread" id=1 idx=0x4 tid=5996 prio=5 alive, native_blocked
  at org/ph/javaee/tool/cpu/HighCPUSimulator.main
     (HighCPUSimulator.java:31)
  at jrockit/vm/RNI.c2java(IIIII)V(Native Method)
  -- end of trace
```

Step #4 - Analyze the culprit Thread(s) Stack Trace and determine root cause

At this point you should have everything that you need to move forward with the root cause analysis. You will need to review each Thread Stack Trace and determine what type of problem you are dealing with. That final step is typically where you will spend most of your time and problem can be simple such as infinite looping or complex such as garbage collection related problems.

In our example, the Thread Dump did reveal the high CPU originates from our sample Java program around line 31. As expected, it did reveal the looping condition that we engineered on purpose for this tutorial.

Case Study - Too many open files

This case study describes the complete root cause analysis and resolution of a File Descriptor (Too

many open files) related problem that we faced following a migration from Oracle ALSB 2.6 running on Solaris OS to Oracle OSB 11g running on AIX.

This section will also provide you with proper AIX OS commands you can use to troubleshoot and validate the File Descriptor configuration of your Java VM process.

Environment specifications

- Java EE server: Oracle Service Bus 11g
- Middleware OS: IBM AIX 6.1
- Java VM: IBM JRE 1.6.0 SR9 - 64 bit
- Platform type: Service Bus - Middle Tier

Problem overview

Problem type: java.net.SocketException: Too many open files error was observed under heavy load causing our Oracle OSB managed servers to suddenly hang.

Such problem was observed only during high load and did require our support team to take corrective action e.g. shutdown and restart the affected Weblogic OSB managed servers.

Gathering and validation of facts

As usual, a Java EE problem investigation requires gathering of technical and non technical facts so we can either derive other facts and/or conclude on the root cause. Before applying a corrective measure, the facts below were verified in order to conclude on the root cause:

- What is the client impact? *HIGH; Full JVM hang*
- Recent change of the affected platform? *Yes, recent migration from ALSB 2.6 (Solaris OS) to Oracle OSB 11g (AIX OS)*
- Any recent traffic increase to the affected platform? *No*
- What is the health of the Weblogic server? *Affected managed servers were no longer responsive along with closure of the Weblogic HTTP (Server Socket) port*
- Did a restart of the Weblogic Integration server resolve the problem? *Yes but temporarily only*

Conclusion #1: The problem appears to be load related

Weblogic server log files review

A quick review of the affected managed servers log did reveal the error below:

```
java.net.SocketException: Too many open files
```

This error indicates that our Java VM process was running out of File Descriptor. This is a severe condition that will affect the whole Java VM process and cause Weblogic to close its internal Server Socket port (HTTP/HTTPS port) preventing any further inbound & outbound communication to the affected managed server(s).

File Descriptor - Why so important for an Oracle OSB environment?

The File Descriptor capacity is quite important for your Java VM process. The key concept you must understand is that File Descriptors are not only required for pure File Handles but also for inbound and outbound Socket communication. Each new Java Socket created to (inbound) or from (outound) your Java VM by Weblogic kernel Socket Muxer requires a File Descriptor allocation at the OS level.

An Oracle OSB environment can require a significant number of Sockets depending how much inbound load it receives and how much outbound connections (Java Sockets) it has to create in order to send and receive data from external / downstream systems (System End Points).

For that reason, you must ensure that you allocate enough File Descriptors / Sockets to your Java VM process in order to support your daily load; including problematic scenarios such as sudden slowdown of external systems which typically increase the demand on the File Descriptor allocation.

Runtime File Descriptor capacity check for Java VM and AIX OS

Following the discovery of this error, our technical team did perform a quick review of the current observed runtime File Descriptor capacity & utilization of our OSB Java VM processes. This can be done easily via the following AIX command:

```
procfiles <Java PID> | grep rlimit & lsof -p <Java PID> | wc -l
```

Java VM process File Descriptor total capacity

```
>> procfiles 5425732 | grep rlimit
  Current rlimit: 2000 file descriptors
```

Java VM process File Descriptor current utilization

```
>> lsof -p <Java PID> | wc -l
 1920
```

As you can see, the current capacity was found at 2000; which is quite low for a medium size Oracle OSB environment. The average utilization under heavy load was also found to be quite close to the upper limit of 2000.

The next step was to verify the default AIX OS File Descriptor limit via the command:

```
>> ulimit -S -n
  2000
```

Conclusion #2: The current File Descriptor limit for both OS and OSB Java VM appears to be quite low and setup at 2000. The File Descriptor utilization was also found to be quite close to the upper limit

which explains why so many JVM failures were observed at peak load.

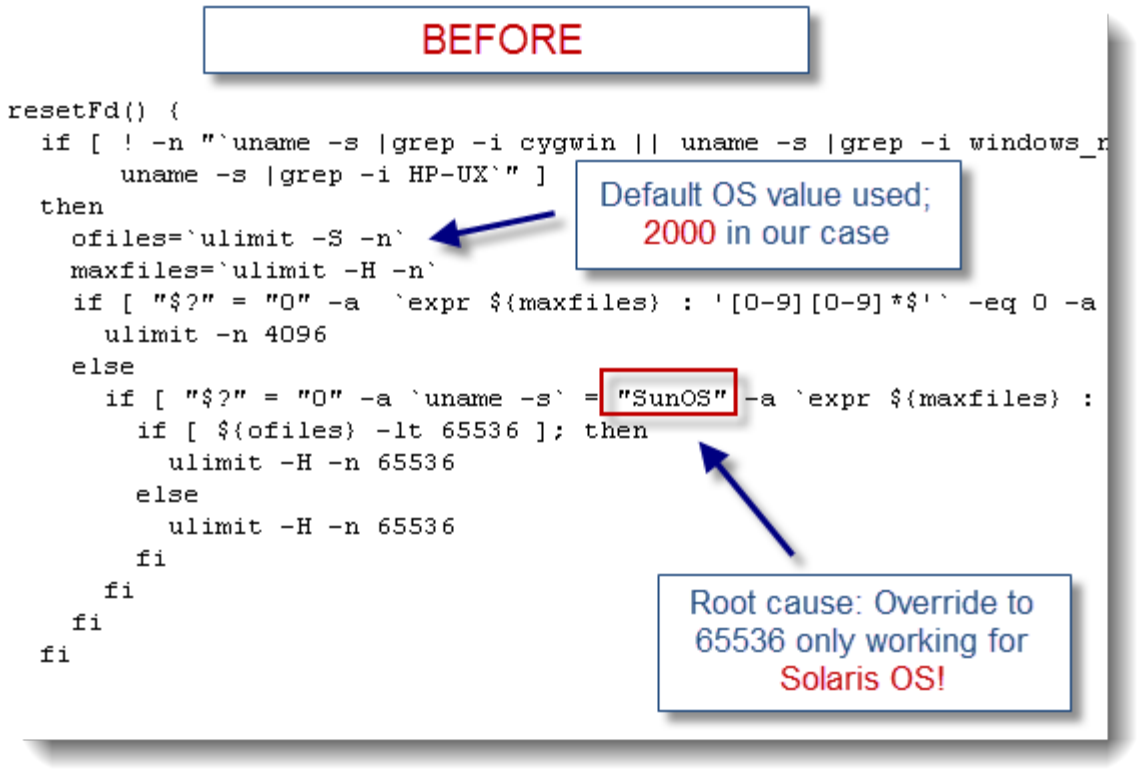
Weblogic File Descriptor configuration review

The File Descriptor limit can typically be overwritten when you start your Weblogic Java VM. Such configuration is managed by the WLS core layer and script can be found at the following location:

```
<WL_HOME>/wlserver_10.3/common/bin/commEnv.sh
```

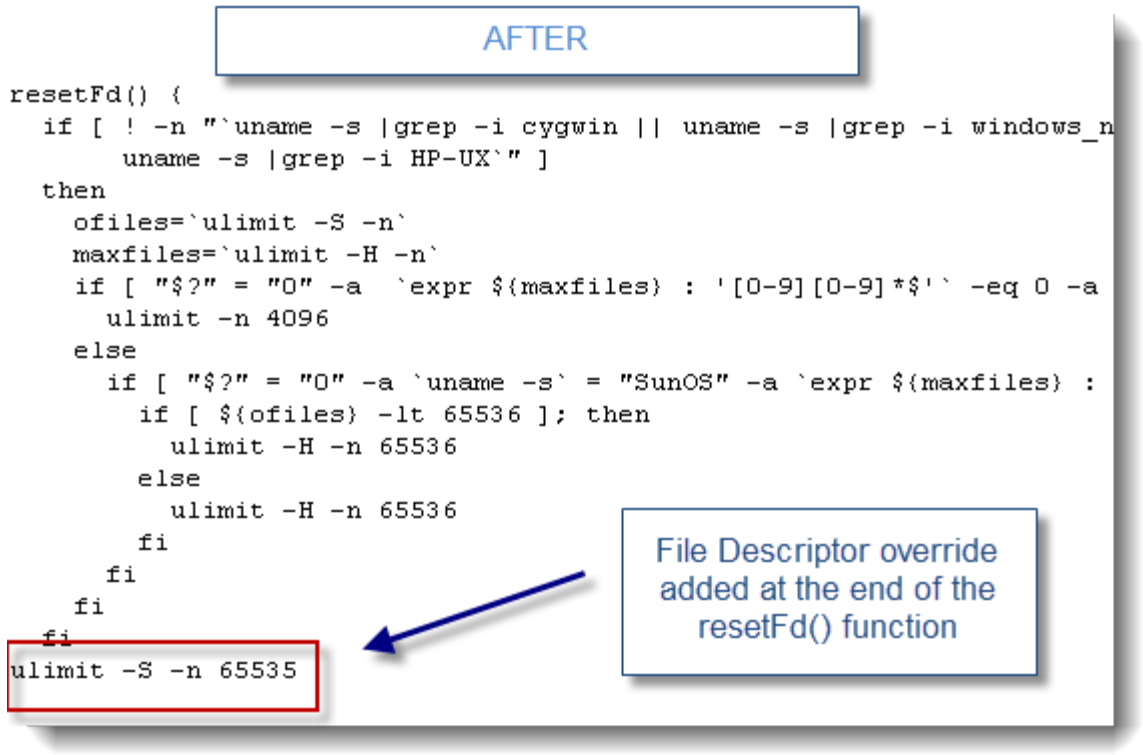
Root cause: File Descriptor override only working for Solaris OS!

As you can see with the script screenshot below, the override of the File Descriptor limit via ulimit is only applicable for Solaris OS (SunOS) which explains why our current OSB Java VM running on AIX OS did end up with the default value of 2000 vs. our older ALSB 2.6 environment running on Solaris OS which had a File Descriptor limit of 65536.



Solution: script tweaking for AIX OS

The resolution of this problem was done by modifying the Weblogic commEnv script as per below. This change did ensure a configuration of 65536 File Descriptor (from 2000); including for the AIX OS:



*** Please note that the activation of any change to the Weblogic File Descriptor configuration requires a restart of both the Node Manager (if used) along with the managed servers. ***

A runtime validation was also performed following the activation of the new configuration which did confirm the new active File Descriptor limit:

```
>> procfiles 6416839 | grep rlimit
  Current rlimit: 65536 file descriptors
```

No failure has been observed since then.

Conclusion and recommendations

When upgrading your Weblogic Java EE container to a new version, please ensure that you verify your current File Descriptor limit as per the above case study. From a capacity planning perspective, please ensure that you monitor your File Descriptor utilizaiton on a regular basis in order to identify any potential capacity problem, Socket leak etc.

GC overhead limit exceeded – Analysis and Patterns

This section will provide you with a description of this new JVM 1.6 HotSpot OutOfMemoryError error message and how you should attack this problem until its resolution.

You can also refer to this post for a real case study on a Java Heap problem (*OutOfMemoryError: GC overhead limit exceeded*) affecting a JBoss production system.

java.lang.OutOfMemoryError: GC overhead limit exceeded - what is it?

Everyone involved in Java EE production support is familiar with OutOfMemoryError problems since they are one of the most common problem type you can face. However, if your environment recently upgraded to Java HotSpot 1.6 VM, you may have observed this error message in the logs: *java.lang.OutOfMemoryError: GC overhead limit exceeded*.

GC overhead limit exceeded is a new policy that was added by default for the Java HotSpot VM 1.6 only. It basically allows the VM to detect potential OutOfMemoryError conditions earlier and before it runs out of Java Heap space; allowing the JVM to abort the current Thread(s) processing with this OOM error.

The official Sun statement is provided below:

The parallel / concurrent collector will throw an OutOfMemoryError if too much time is being spent in garbage collection: if more than 98% of the total time is spent in garbage collection and less than 2% of the heap is recovered, an OutOfMemoryError will be thrown. This feature is designed to prevent applications from running for an extended period of time while making little or no progress because the heap is too small. If necessary, this feature can be disabled by adding the option -XX:-UseGCOverheadLimit to the command line.

Is it useful for Java EE production systems?

I have found on most of my problem cases that this new policy is useful at some level since it is preventing a full JVM hang and allowing you to take some actions such as data collection, JVM Heap Dump, JVM Thread Dump etc. before the whole JVM becomes unresponsive.

But don't expect this new feature to fix your Java Heap problem, it is meant to prevent a full JVM hang and to abort some big memory allocation etc. you must still perform your own analysis and due diligence.

Is there any scenario where it can cause more harm than good?

Yes, Java applications dealing with large memory allocations / chunks could see much more frequent OOM due to GC overhead limit exceeded. Some applications dealing with a long GC elapsed time but healthy overall memory usage could also be affected.

In the above scenarios, you may want to consider turning OFF this policy and see if it's helping your environment stability.

java.lang.OutOfMemoryError: GC overhead limit exceeded - can I disable it?

Yes, you can disable this default policy by simply adding this parameter at your JVM start-up:

```
-XX:-UseGCOverheadLimit
```

Please keep in mind that this error is very likely to be a symptom of a JVM Heap / tuning problem so my recommendation to you is always to focus on the root cause as opposed to the symptom.

java.lang.OutOfMemoryError: GC overhead limit exceeded - how can I fix it?

You should not worry too much about the GC overhead limit error itself since it's very likely just a symptom / hint. What you must focus on is on your potential Java Heap problem(s) such as Java Heap leak, improper Java Heap tuning etc. Find below a list of high level steps to troubleshoot further:

- If not done already, enabled verbose GC >> -verbose:gc
- Analyze the verbose GC output and determine the memory footprint of the Java Heap; including the ratio of Young Gen vs. Old Gen. Having an old gen footprint too high will lead to too many frequent Full GC and ultimately to the OOM: GC overhead limit exceeded.
- Analyze the verbose GC output or use a tool like JConsole to determine if your Java Heap is leaking over time. This can be observed via monitoring of the HotSpot old gen space.
- Look at your young Gen requirement as well, if you application generates a lot of short live objects then your Java Heap space must be big enough in order for the VM to allocate a bigger Young Gen space.
- If facing a Java Heap leak and/or if you have concern on your Old Gen footprint then add the following parameter to your start-up JVM arguments: -XX:+HeapDumpOnOutOfMemoryError. This will generate a Heap Dump (hprof format) on OOM event that you can analyze using a tool like Memory Analyzer or Jhat.

Heap Analysis

Now we will provide you with a sample program and a tutorial on how to analyze your Java HotSpot Heap footprint using Memory Analyzer following an OutOfMemoryError. I highly recommend that you execute and analyse the Heap Dump yourself using this tutorial in order to better understand these principles.

Troubleshooting tools

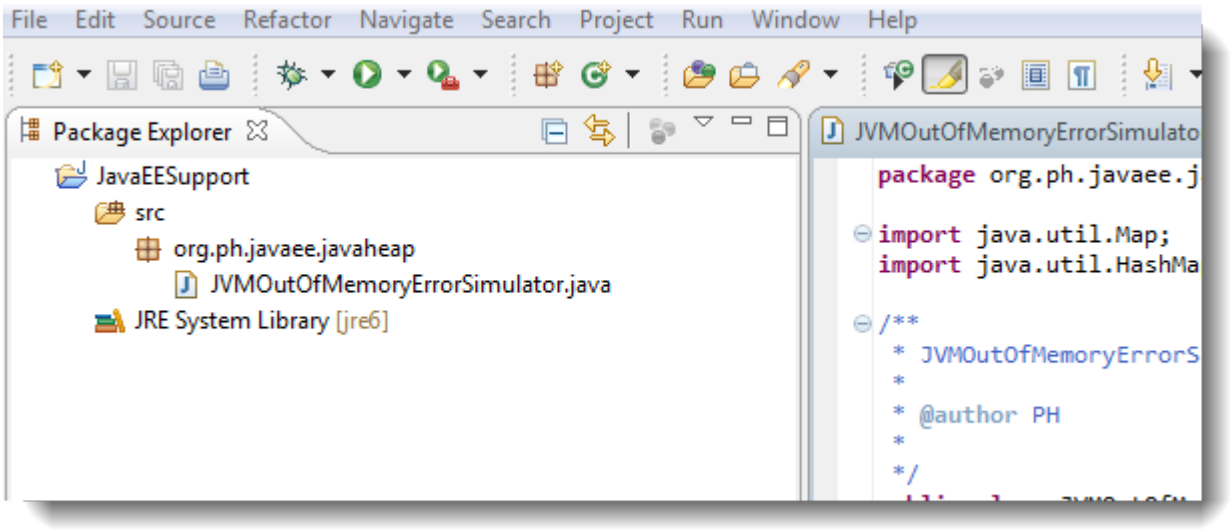
*** all these tools can be downloaded for free ***

- Eclipse Indigo Release
- Memory Analyzer via IBM Support Assistant 4.1 (HotSpot Heap Dump analysis)
- Java VM: Windows HotSpot JRE 1.6.0_24 64-bit

Sample Java program

The simple sample Java program below will be used to triggered an OutOfMemoryError; allowing us to analyze the generated HotSpot Heap Dump file. Simply create a new Java class : JVMOutOfMemoryErrorSimulator.java to the Eclipse project of your choice and either rename or keep the current package as is.

This program is basically creating multiple String instances within a Map data structure until the Java Heap depletion.

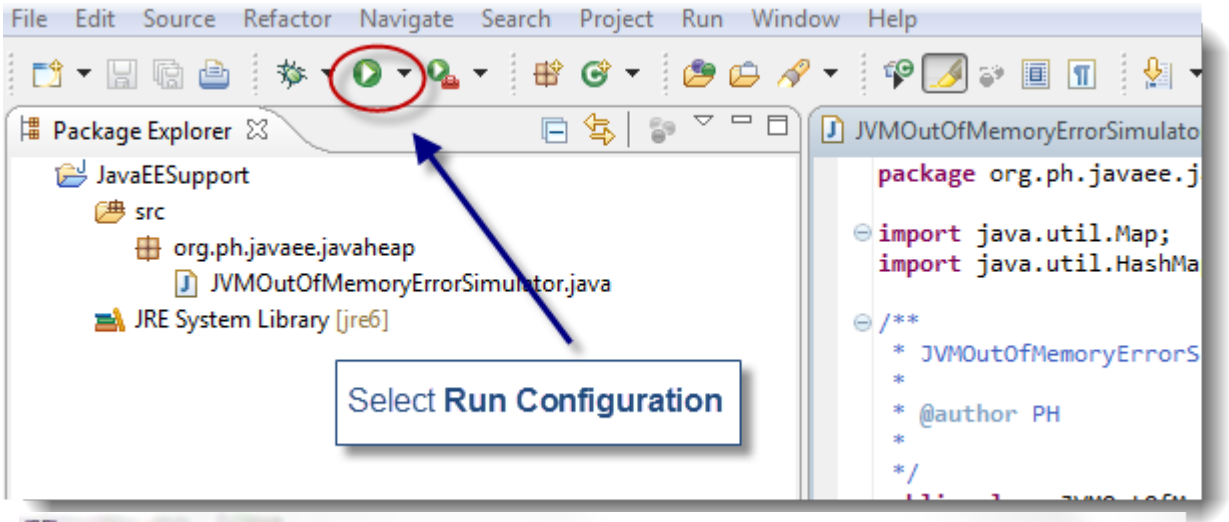


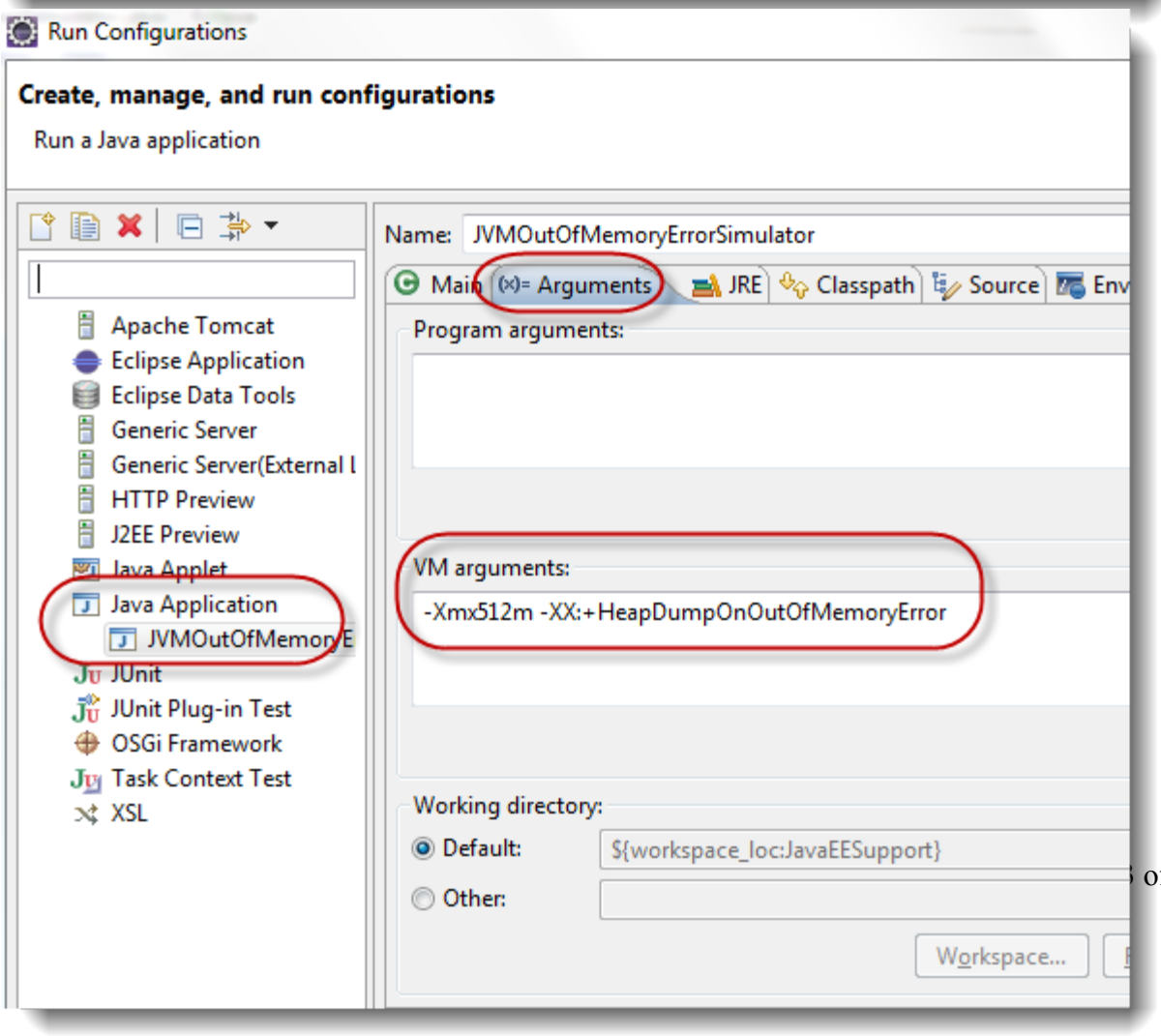
```
package org.ph.javaee.javaheap;

import java.util.Map;
import java.util.HashMap;

/**
 * JVMOutOfMemoryErrorSimulator
 *
 * @author PH
 *
 */
public class JVMOutOfMemoryErrorSimulator {

   private final static int NB_ITERATIONS = 500000;

   // ~1 KB data footprint
   private final static String LEAKING_DATA_PREFIX =
"datadatadatadatadatadatadatadatadatadatadatadatadatadatadatadatadatadatadatadatadatadatadatadatadat
adatadatadatadatadatadatadatadatadatadatadatadatadatadatadatadatadatadatadatadatadatadatadatadatadat
adatadatadatadatadatadatadatadatadatadatadatadatadatadatadatada";

   // Map used to stored our leaking String instances
   private static Map<String, String> leakingMap;
   static {
      leakingMap = new HashMap<String, String>();
   }

   public static void main(String[] args) {

      System.out.println("JVM OutOfMemoryError Simulator 1.0");
      System.out.println("Author: Pierre-Hugues Charbonneau");
      System.out.println("http://javaeesupportpatterns.blogspot.com/");

      try {

         for (int i = 0; i < NB_ITERATIONS; i++) {

            String data = LEAKING_DATA_PREFIX + i;
            // Add data to our leaking Map data structure...
            leakingMap.put(data, data);

         }

      } catch (Throwable any) {
         if (any instanceof java.lang.OutOfMemoryError) {
            System.out.println("OutOfMemoryError triggered! "
                  + any.getMessage() + " [" + any + "]");

         } else {
            System.out.println("Unexpected Exception! " + any.getMessage()
                  + " [" + any + "]");
         }
      }
      System.out.println("simulator done!");
   }

}
```

Step #1 - Setup your JVM start-up arguments

First, setup your Eclipse Java runtime arguments as per below. For our example, we used an external JRE 1.6 outside the Eclipse IDE with a Java Heap maximum capacity of 512 MB.

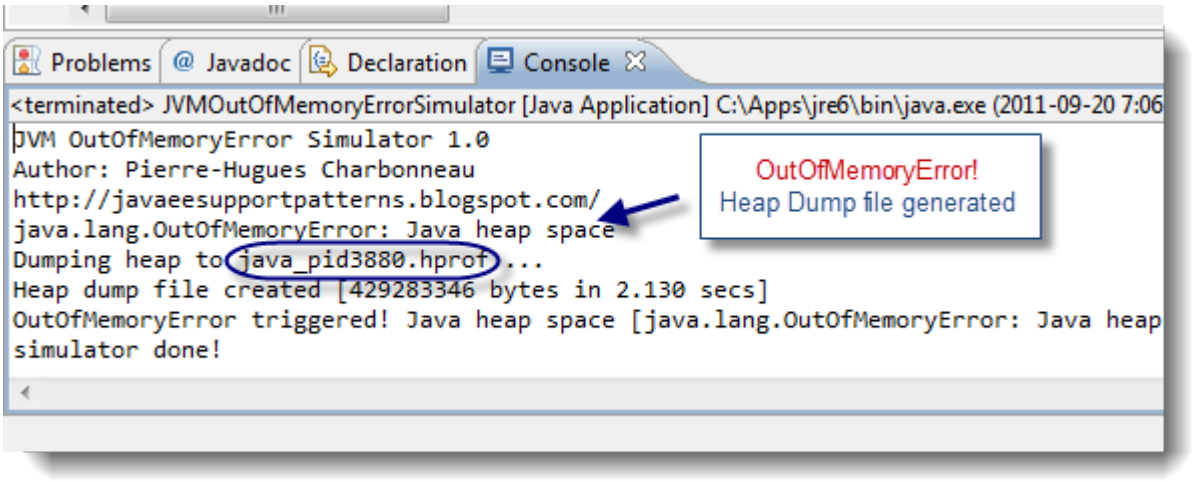
The key JVM argument allowing us to generate a Heap Dump is -XX: +HeapDumpOnOutOfMemoryError which tells the JVM to generate a Heap Dump following an OutOfMemoryError condition.





Step #2 - Run the sample Java program

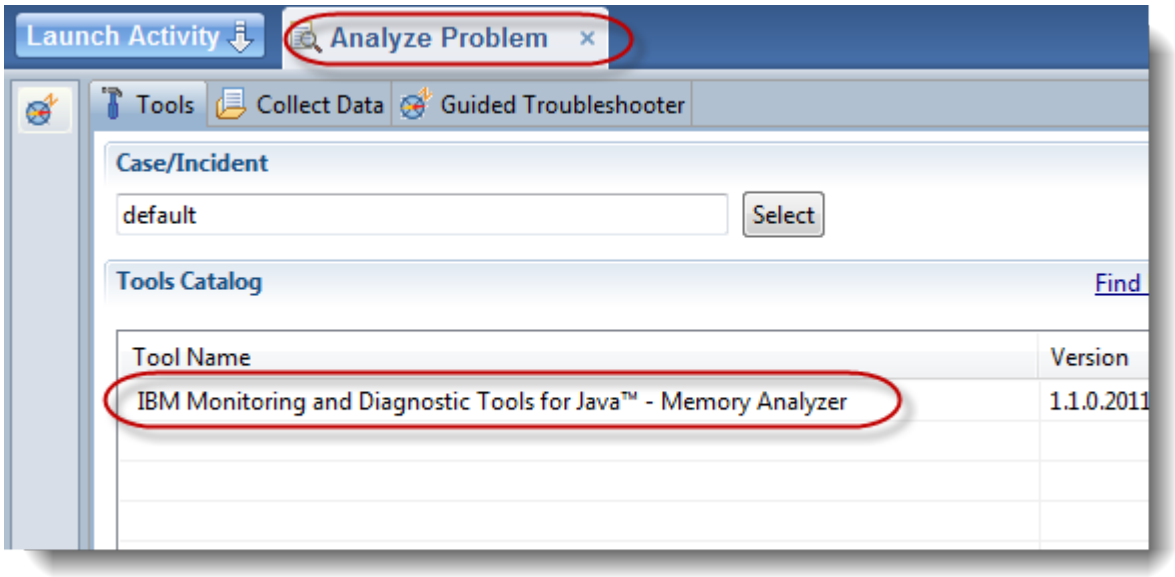
The next step is to run our Java program. Depending on your computer specs, this program will run between 5-30 seconds before exiting with an OutOfMemoryError.

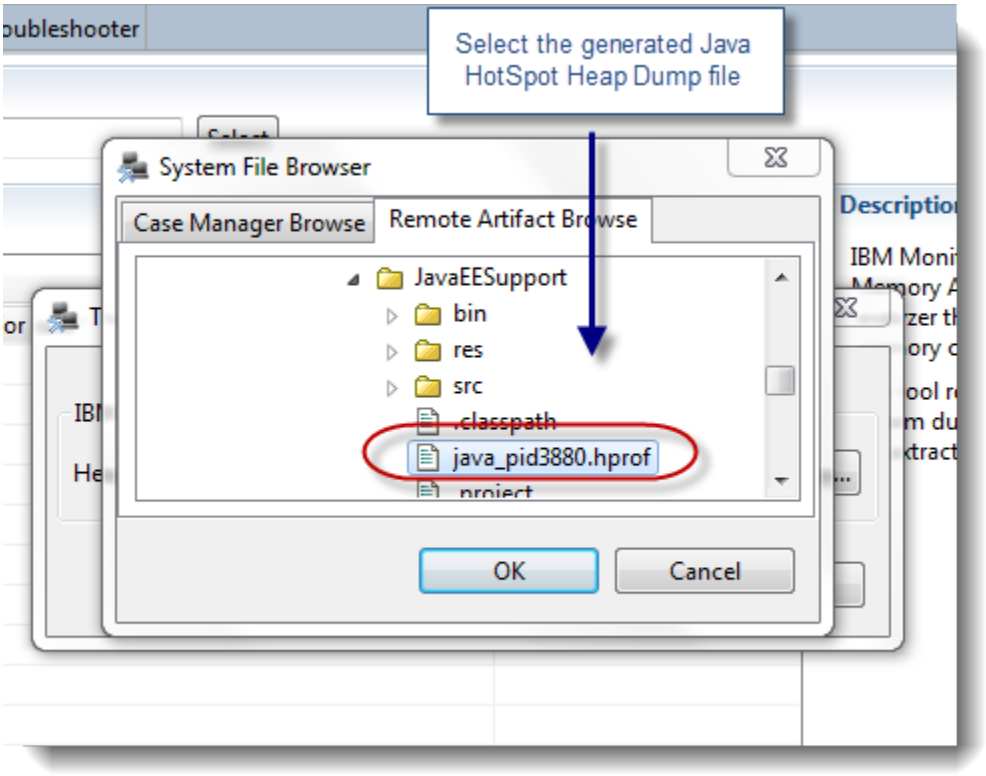


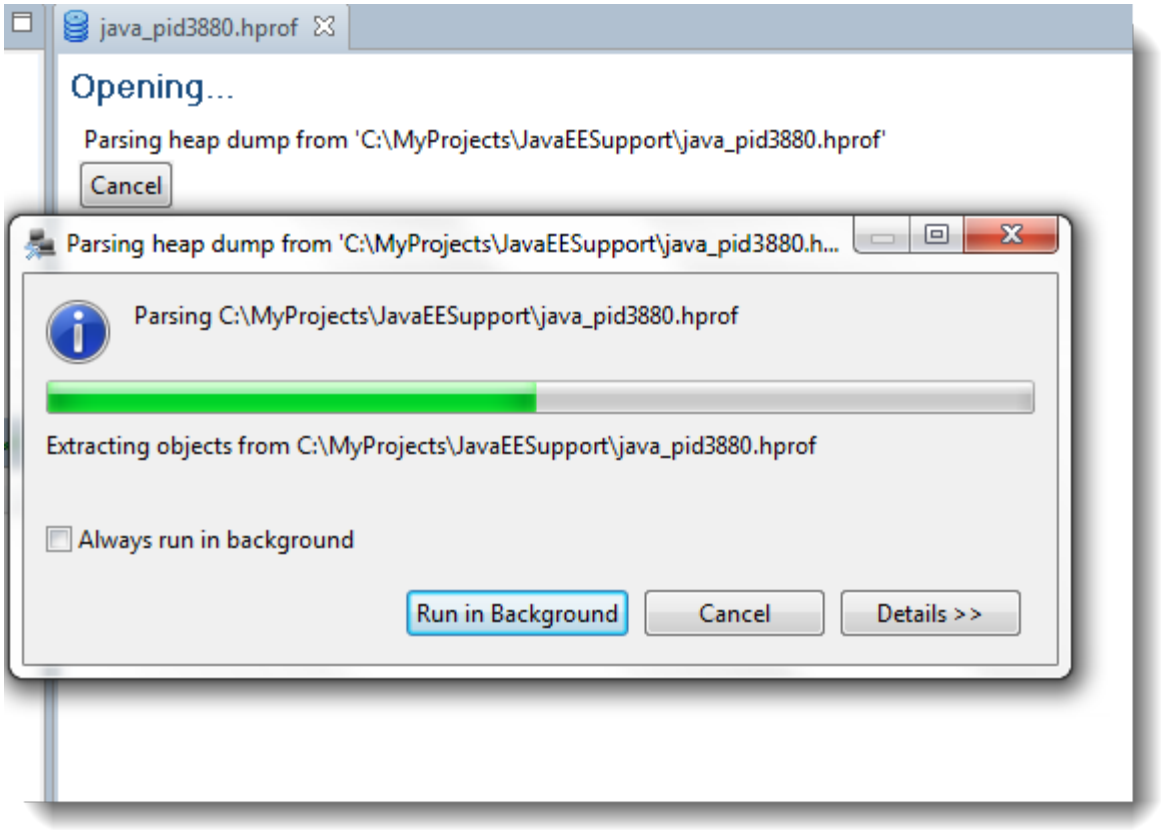
As you can see, the JVM generated a Heap Dump file java_pid3880.hprof. It is now time to fire the Memory Analyzer tool and analyze the JVM Heap Dump.

Step #3 - Load the Heap Dump

Analyzing a Heap Dump is an analysis activity that can be simple or very complex. The goal of this tutorial is to give you the basics of Heap Dump analysis. For more Heap Dump analysis, please refer to the other case studies presented in this handbook.

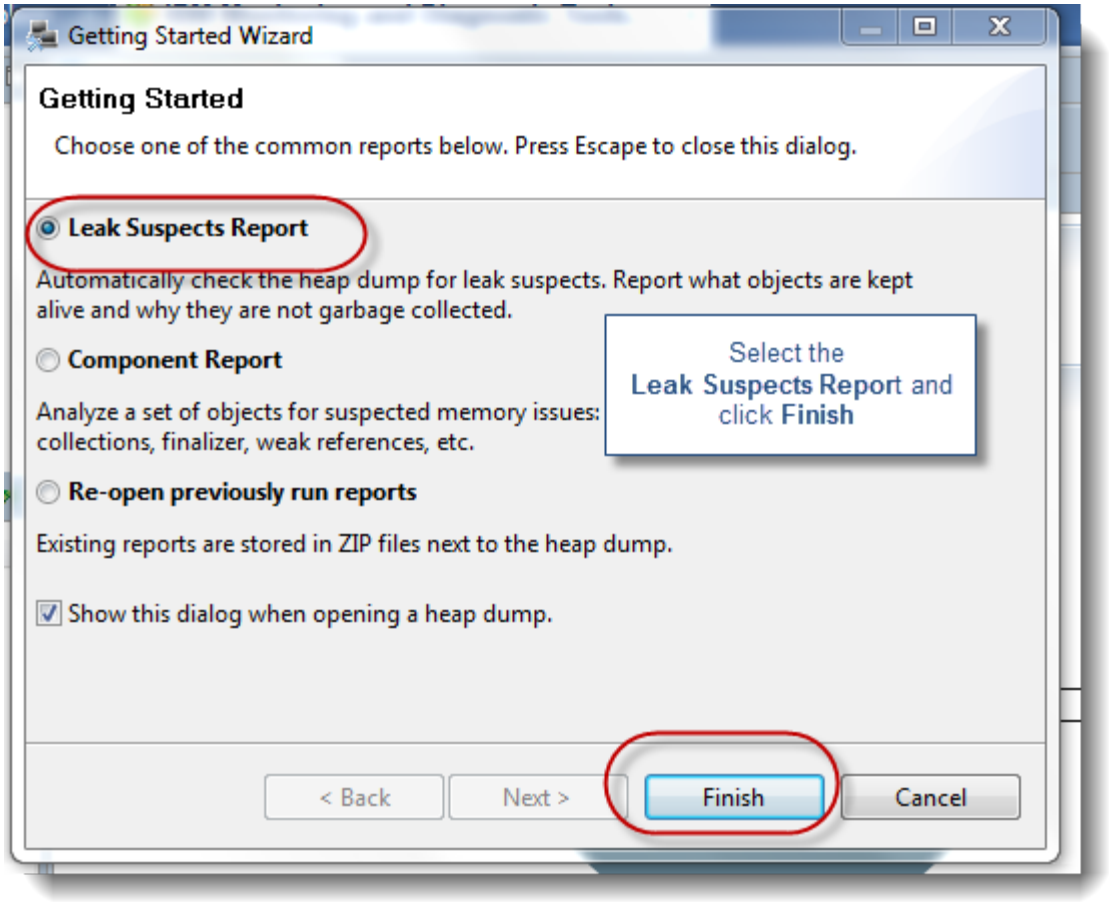


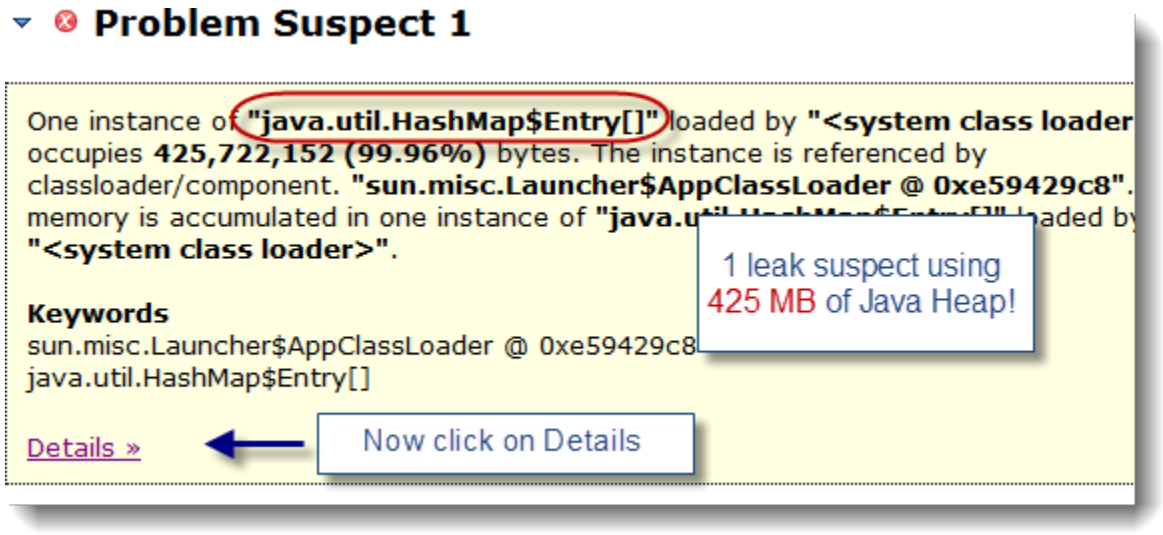

Select the generated Java HotSpot Heap Dump file
System File Browser
Case Manager Browse
Remote Artifact Browse
JavaEESupport
bin
res
src
.classpath
java_pid3880.hprof
OK
Cancel

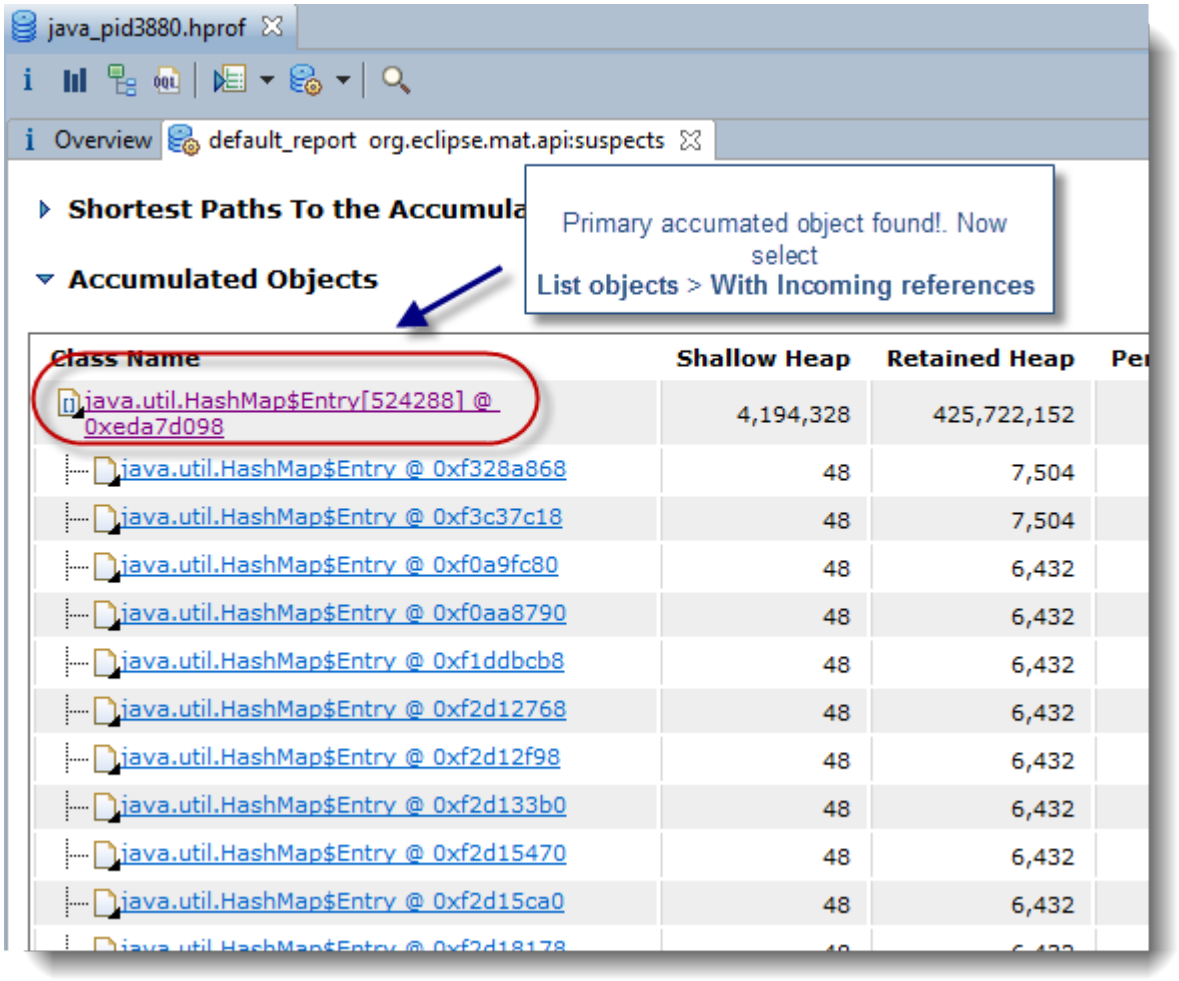


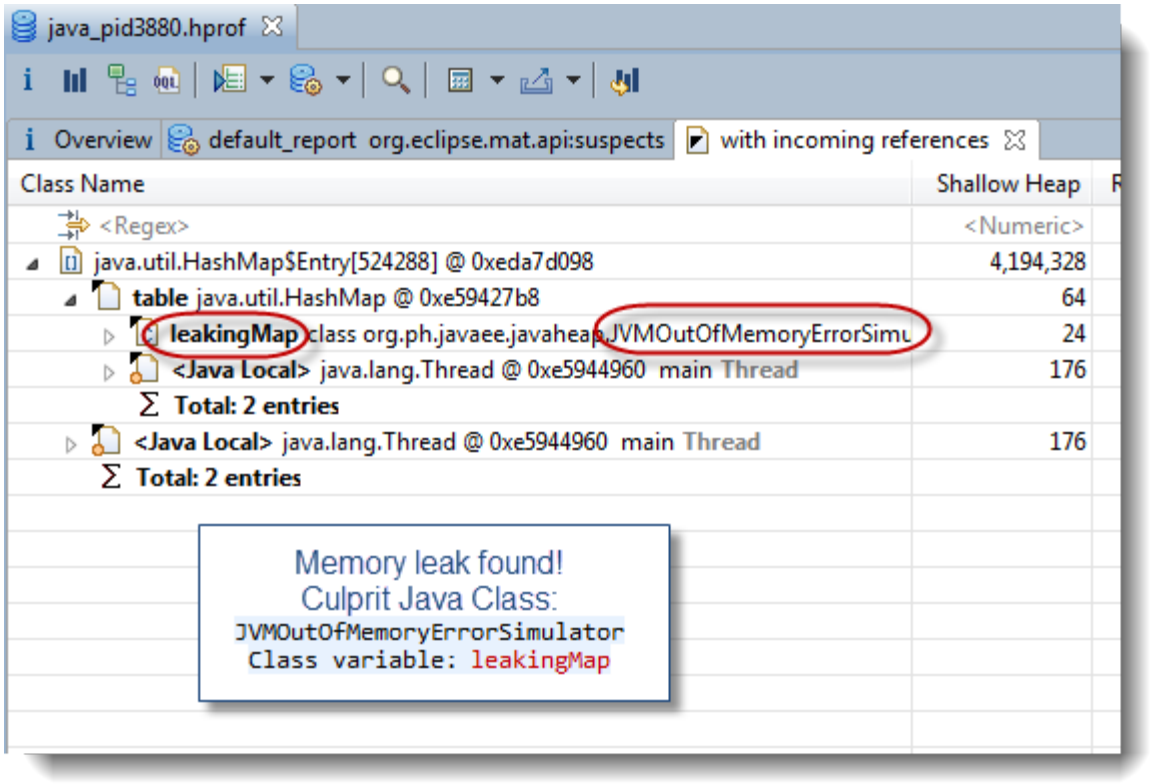
Step #4 - Analyze Heap Dump

Below are the snapshots and analysis steps that you can follow to understand the memory leak that we simulated in our sample Java program.


Getting Started Wizard
Getting Started
Choose one of the common reports below. Press Escape to close this dialog.
Leak Suspects Report
Automatically check the heap dump for leak suspects. Report what objects are kept alive and why they are not garbage collected.
Component Report
Analyze a set of objects for suspected memory issues: collections, finalizer, weak references, etc.
Re-open previously run reports
Existing reports are stored in ZIP files next to the heap dump.
Show this dialog when opening a heap dump.
< Back
Next >
Finish
Cancel
Select the Leak Suspects Report and click Finish


Problem Suspect 1
One instance of "java.util.HashMap$Entry[]" loaded by "<system class loader
occupies 425,722,152 (99.96%) bytes. The instance is referenced by
classloader/component. "sun.misc.Launcher$AppClassLoader @ 0xe59429c8".
memory is accumulated in one instance of "java.u
"<system class loader>".
Keywords
sun.misc.Launcher$AppClassLoader @ 0xe59429c8
java.util.HashMap$Entry[]
Details »
1 leak suspect using 425 MB of Java Heap!
Now click on Details


java_pid3880.hprof
Overview
default_report org.eclipse.mat.api:suspects
Shortest Paths To the Accumula
Accumulated Objects
Primary accumated object found!. Now select
List objects > With Incoming references
Class Name
Shallow Heap
Retained Heap
java.util.HashMap$Entry[524288] @ 0xeda7d098
4,194,328
425,722,152
java.util.HashMap$Entry @ 0xf328a868
48
7,504
java.util.HashMap$Entry @ 0xf3c37c18
48
7,504
java.util.HashMap$Entry @ 0xf0a9fc80
48
6,432
java.util.HashMap$Entry @ 0xf0aa8790
48
6,432
java.util.HashMap$Entry @ 0xf1ddbcb8
48
6,432
java.util.HashMap$Entry @ 0xf2d12768
48
6,432
java.util.HashMap$Entry @ 0xf2d12f98
48
6,432
java.util.HashMap$Entry @ 0xf2d133b0
48
6,432
java.util.HashMap$Entry @ 0xf2d15470
48
6,432
java.util.HashMap$Entry @ 0xf2d15ca0
48
6,432



As you can see, the Heap Dump analysis using the Memory Analyzer tool was able to easily identify our primary leaking Java class and data structure.

Conclusion

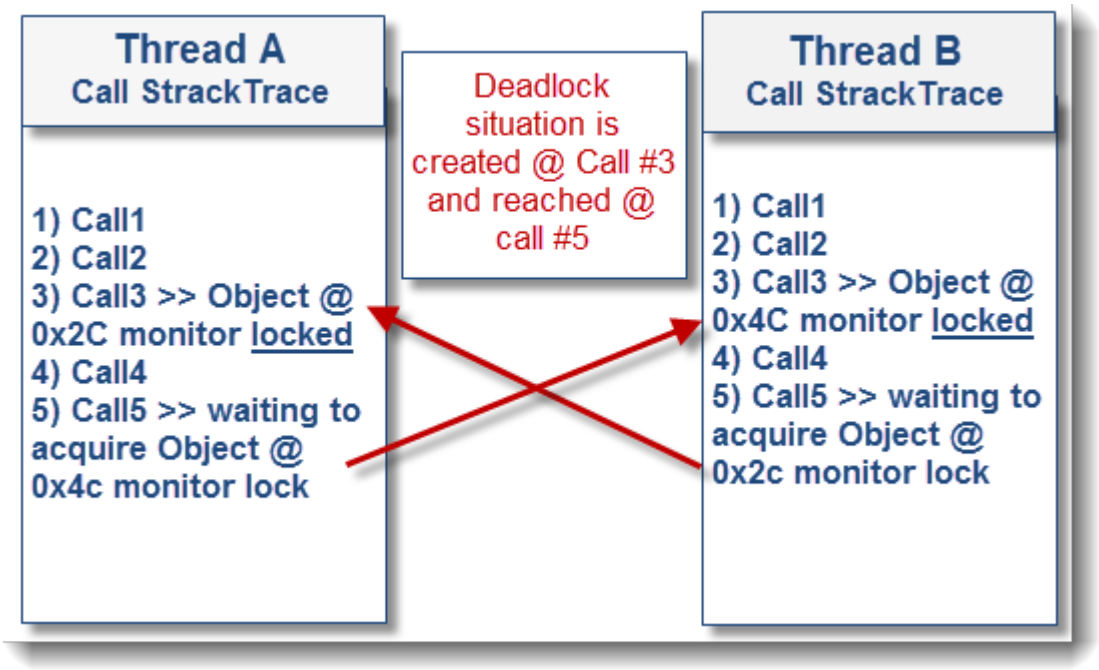
I hope this simple Java program and Heap Dump analysis tutorial has helped you understand the basic principles of Java Heap analysis using the raw Heap Dump data. This analysis is critical when dealing with OutOfMemoryError: GC overhead problems since those are symptoms of either Java Heap leak of Java Heap footprint / tuning problem.

Java deadlock troubleshooting and analysis

This section will revisit the classic thread problem of deadlocks and summarize some key troubleshooting and resolution techniques that can be used.

Java deadlock: what is it?

A true Java deadlock can essentially be described as a situation where two or more threads are blocked forever, waiting for each other. This situation is very different from other more commons "day-to-day" thread problem patterns such as lock contention & thread races, threads waiting on blocking IO calls etc. Such lock-ordering deadlock situation can be visualized as presented below:



In the above visual example, the attempt by Thread A & Thread B to acquire 2 locks in different orders is fatal. Once threads reached the deadlocked state, they can never recover, forcing you to restart the affected JVM process.

There is also another type of deadlock: resource deadlock. This is by far the most common thread problem pattern I have seen in my experience with Java EE enterprise system troubleshooting. A resource deadlock is essentially a scenario where one or multiple threads are waiting to acquire a resource which will never be available such as JDBC Pool depletions.

Lock-ordering deadlocks

You should know by now that I am a big fan of JVM thread dump analysis; crucial skill to acquire for individuals either involved in Java/Java EE development or production support. The good news is that Java-level deadlocks can be easily identified out-of-the-box by most JVM thread dump formats (HotSpot, IBM VM...) since they contain a native deadlock detection mechanism which will actually show you the threads involved in a true Java-level deadlock scenario along with the execution stack trace. JVM thread dump can be captured via the tool of your choice such as JVisualVM, jstack or natively such as kill -3 <PID> on Unix-based OS. Find below the JVM Java-level deadlock detection section after running lab 1:

Now this is the easy part…The core of the root cause analysis effort is to understand why such threads are involved in a deadlock situation at the first place. Lock-ordering deadlocks could be triggered from your application code but unless you are involved in high concurrency programming, chances are that the culprit code is a third part API or framework that you are using or the actual Java EE container itself, when applicable.

Now let's review below the lock-ordering deadlock resolution strategies presented by Heinz in his presentation "HOL6500 - Finding And Solving Java Deadlocks":

Deadlock resolution by global ordering (see lab1 solution)

Essentially involves the definition of a global ordering for the locks that would always prevent deadlock (please see lab1 solution)

Deadlock resolution by TryLock (see lab2 solution)

- Lock the first lock
- Then try to lock the second lock
- If you can lock it, you are good to go
- If you cannot, wait and try again

The above strategy can be implemented using Java Lock & ReantrantLock which also gives you also flexibility to setup a wait timeout in order to prevent thread starvation in the event the first lock is acquired for too long.

```
public interface Lock {
  void lock();
  void lockInterruptibly() throws InterruptedException;
  boolean tryLock();
  boolean tryLock(long timeout, TimeUnit unit)
    throws InterruptedException;
  void unlock();
  Condition newCondition();
}
```

If you look at the JBoss AS7 implementation, you will notice that Lock & ReantrantLock are widely used from core implementation layers such as:

- Deployment service
- EJB3 implementation (widely used)
- Clustering and session management
- Internal cache & data structures (LRU, ConcurrentReferenceHashMap...)

Now and as per Heinz's point, the deadlock resolution strategy #2 can be quite efficient but proper care is also required such as releasing all held lock via a finally{} block otherwise you can transform your deadlock scenario into a livelock.

Resource deadlocks

Now let's move to resource deadlock scenarios. I'm glad that Heinz's lab #3 covered this since from my experience this is by far the most common "deadlock" scenario that you will see, especially if you are developing and supporting large distributed Java EE production systems.

Now let's get the facts right.

- Resource deadlocks are not true Java-level deadlocks.
- The JVM Thread Dump will not magically should you these types of deadlocks. This means more work for you to analyze and understand this problem as a starting point.
- Thread dump analysis can be especially confusing when you are just starting to learn how to read Thread Dump since threads will often show up as RUNNING state vs. BLOCKED state for Java-level deadlocks. For now, it is important to keep in mind that thread state is not that important for this type of problem e.g. RUNNING state != healthy state.
- The analysis approach is very different than Java-level deadlocks. You must take multiple thread dump snapshots and identify thread problem/wait patterns between each snapshot. You will be able to see threads not moving e.g. threads waiting to acquire a resource from a pool and other threads that already acquired such resource and hanging.
- Thread Dump analysis is not the only data point/fact important here. You will need to collect other facts such statistics on the resource(s) the threads are waiting for, overall middleware or environment health etc. The combination of all these facts will allow you to conclude on the root cause along with a resolution strategy which may or may not involve code change.

Java Thread deadlock - Case Study

This section will describe the complete root cause analysis of a recent Java deadlock problem observed from a Weblogic 11g production system running on the IBM JVM 1.6. It will also demonstrate the importance of mastering Thread Dump analysis skills; including for the IBM JVM Thread Dump format.

Environment specifications

- Java EE server: Oracle Weblogic Server 11g & Spring 2.0.5
- OS: AIX 5.3
- Java VM: IBM JRE 1.6.0
- Platform type: Portal & ordering application

Monitoring and troubleshooting tools

- JVM Thread Dump (IBM JVM format)
- Compuware Server Vantage (Weblogic JMX monitoring & alerting)

Problem overview

A major stuck Threads problem was observed & reported from Compuware Server Vantage and affecting 2 of our Weblogic 11g production managed servers causing application impact and timeout conditions from our end users.

Gathering and validation of facts

As usual, a Java EE problem investigation requires gathering of technical and non-technical facts so we can either derived other facts and/or conclude on the root cause. Before applying a corrective measure, the facts below were verified in order to conclude on the root cause:

- What is the client impact? *MEDIUM (only 2 managed servers / JVM affected out of 16)*
- Recent change of the affected platform? *Yes (new JMS related asynchronous component)*
- Any recent traffic increase to the affected platform? *No*
- How does this problem manifest itself? *A sudden increase of Threads was observed leading to rapid Thread depletion*
- Did a Weblogic managed server restart resolve the problem? *Yes, but problem is returning after few hours (unpredictable & intermittent pattern)*

Conclusion #1: The problem is related to an intermittent stuck Threads behaviour affecting only a few Weblogic managed servers at the time.
Conclusion #2: Since problem is intermittent, a global root cause such as a non-responsive downstream system is not likely.

Thread Dump analysis - first pass

The first thing to do when dealing with stuck Thread problems is to generate a JVM Thread Dump.

This is a golden rule regardless of your environment specifications & problem context. A JVM Thread Dump snapshot provides you with crucial information about the active Threads and what type of processing / tasks they are performing at that time.

Now back to our case study, an IBM JVM Thread Dump (javacore.xyz format) was generated which did reveal the following Java Thread deadlock condition below:

```
1LKDEADLOCK    Deadlock detected !!!
NULL           ---------------------
NULL
2LKDEADLOCKTHR  Thread "[STUCK] ExecuteThread: '8' for queue:
'weblogic.kernel.Default (self-tuning)'" (0x000000012CC08B00)
3LKDEADLOCKWTR    is waiting for:
4LKDEADLOCKMON      sys_mon_t:0x0000000126171DF8 infl_mon_t:
0x0000000126171E38:
4LKDEADLOCKOBJ
weblogic/jms/frontend/FESession@0x07000000198048C0/0x07000000198048D8:
3LKDEADLOCKOWN    which is owned by:
2LKDEADLOCKTHR  Thread "[STUCK] ExecuteThread: '10' for queue:
'weblogic.kernel.Default (self-tuning)'" (0x000000012E560500)
3LKDEADLOCKWTR    which is waiting for:
4LKDEADLOCKMON      sys_mon_t:0x000000012884CD60 infl_mon_t:
0x000000012884CDA0:
4LKDEADLOCKOBJ
weblogic/jms/frontend/FEConnection@0x0700000019822F08/0x0700000019822F20:
3LKDEADLOCKOWN    which is owned by:
2LKDEADLOCKTHR  Thread "[STUCK] ExecuteThread: '8' for queue:
'weblogic.kernel.Default (self-tuning)'" (0x000000012CC08B00)
```

This deadlock situation can be translated as per below:

- Weblogic Thread #8 is waiting to acquire an Object monitor lock owned by Weblogic Thread #10
- Weblogic Thread #10 is waiting to acquire an Object monitor lock owned by Weblogic Thread #8

Conclusion: Both Weblogic Threads #8 & #10 are waiting on each other; forever!

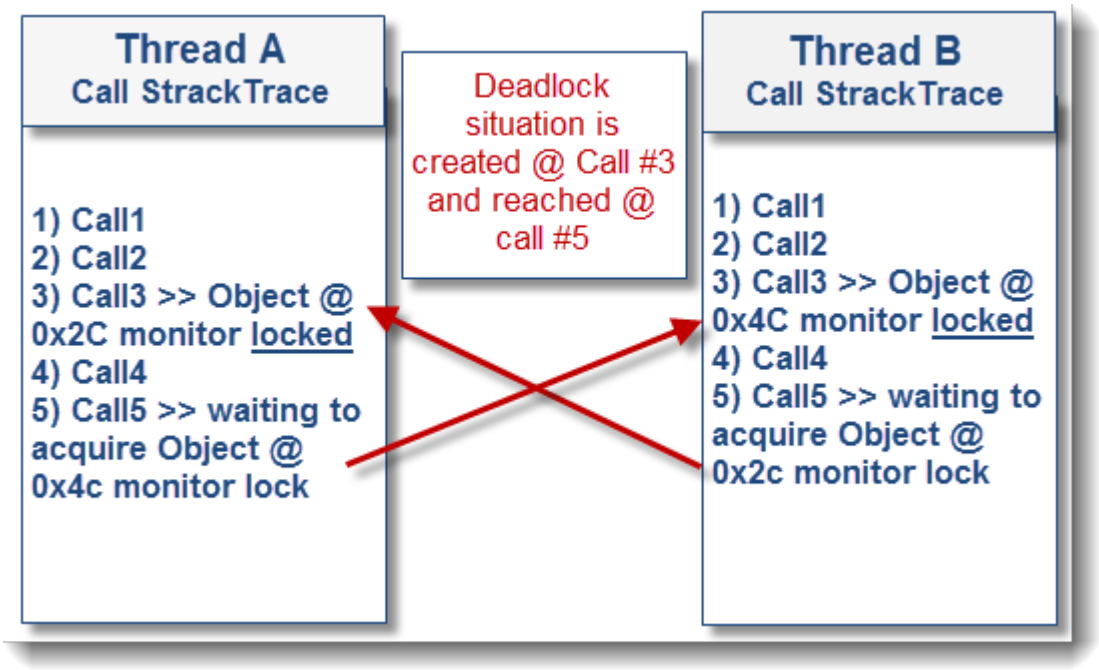
Now before going any deeper in this root cause analysis, let me provide you a high level overview on Java Thread deadlocks.

Java Thread deadlock overview

Most of you are probably familiar with Java Thread deadlock principles but did you really experience a true deadlock problem?

From my experience, true Java deadlocks are rare and I have only seen ~5 occurrences over the last 10 years. The reason is that most stuck Threads related problems are due to Thread hanging conditions (waiting on remote IO call etc.) but not involved in a true deadlock condition with other Thread(s).

A Java Thread deadlock is a situation for example where Thread A is waiting to acquire an Object monitor lock held by Thread B which is itself waiting to acquire an Object monitor lock held by Thread A. Both these Threads will wait for each other forever. This situation can be visualized as per below diagram:



Thread deadlock is confirmed...now what can you do?

Once the deadlock is confirmed (most JVM Thread Dump implementations will highlight it for you), the next step is to perform a deeper dive analysis by reviewing each Thread involved in the deadlock situation along with their current task & wait condition.

Find below the partial Thread Stack Trace from our problem case for each Thread involved in the deadlock condition:

*** Please note that the real application Java package name was renamed for confidentiality purposes* **

Weblogic Thread #8

```
"[STUCK] ExecuteThread: '8' for queue: 'weblogic.kernel.Default (self-tuning)'"
J9VMThread:0x000000012CC08B00, j9thread_t:0x00000001299E5100,
java/lang/Thread:0x070000001D72EE00, state:B, prio=1
(native thread ID:0x111200F, native priority:0x1, native policy:UNKNOWN)
Java callstack:
   at weblogic/jms/frontend/FEConnection.stop(FEConnection.java:671(Compiled
Code))
   at weblogic/jms/frontend/FEConnection.invoke(FEConnection.java:1685(Compiled
Code))
   at
weblogic/messaging/dispatcher/Request.wrappedFiniteStateMachine(Request.java:96
1(Compiled Code))
   at
weblogic/messaging/dispatcher/DispatcherImpl.syncRequest(DispatcherImpl.java:18
4(Compiled Code))
   at
weblogic/messaging/dispatcher/DispatcherImpl.dispatchSync(DispatcherImpl.java:2
12(Compiled Code))
   at
weblogic/jms/dispatcher/DispatcherAdapter.dispatchSync(DispatcherAdapter.java:4
3(Compiled Code))
   at weblogic/jms/client/JMSConnection.stop(JMSConnection.java:863(Compiled
Code))
   at weblogic/jms/client/WLConnectionImpl.stop(WLConnectionImpl.java:843)
   at
org/springframework/jms/connection/SingleConnectionFactory.closeConnection(Sing
leConnectionFactory.java:342)
   at
org/springframework/jms/connection/SingleConnectionFactory.resetConnection(Sing
leConnectionFactory.java:296)
   at org/app/JMSReceiver.receive()
.....................................................................
```

Weblogic Thread #10

```
"[STUCK] ExecuteThread: '10' for queue: 'weblogic.kernel.Default (self-tuning)'"
J9VMThread:0x000000012E560500, j9thread_t:0x000000012E35BCE0,
java/lang/Thread:0x070000001ECA9200, state:B, prio=1
(native thread ID:0x4FA027, native priority:0x1, native policy:UNKNOWN)
Java callstack:
 at weblogic/jms/frontend/FEConnection.getPeerVersion(FEConnection.java:1381(Compiled
Code))
 at weblogic/jms/frontend/FESession.setUpBackEndSession(FESession.java:755(Compiled
Code))
 at weblogic/jms/frontend/FESession.consumerCreate(FESession.java:1025(Compiled Code))
 at weblogic/jms/frontend/FESession.invoke(FESession.java:2995(Compiled Code))
 at
weblogic/messaging/dispatcher/Request.wrappedFiniteStateMachine(Request.java:961(Compile
d Code))
 at
weblogic/messaging/dispatcher/DispatcherImpl.syncRequest(DispatcherImpl.java:184(Compile
d Code))
 at
weblogic/messaging/dispatcher/DispatcherImpl.dispatchSync(DispatcherImpl.java:212(Compil
ed Code))
 at
weblogic/jms/dispatcher/DispatcherAdapter.dispatchSync(DispatcherAdapter.java:43(Compile
d Code))
 at weblogic/jms/client/JMSSession.consumerCreate(JMSSession.java:2982(Compiled Code))
 at weblogic/jms/client/JMSSession.setupConsumer(JMSSession.java:2749(Compiled Code))
 at weblogic/jms/client/JMSSession.createConsumer(JMSSession.java:2691(Compiled Code))
 at weblogic/jms/client/JMSSession.createReceiver(JMSSession.java:2596(Compiled Code))
 at weblogic/jms/client/WLSessionImpl.createReceiver(WLSessionImpl.java:991(Compiled
Code))
 at
org/springframework/jms/core/JmsTemplate102.createConsumer(JmsTemplate102.java:204(Compi
led Code))
 at org/springframework/jms/core/JmsTemplate.doReceive(JmsTemplate.java:676(Compiled
Code))
 at org/springframework/jms/core/JmsTemplate$10.doInJms(JmsTemplate.java:652(Compiled
Code))
 at org/springframework/jms/core/JmsTemplate.execute(JmsTemplate.java:412(Compiled
Code))
 at
org/springframework/jms/core/JmsTemplate.receiveSelected(JmsTemplate.java:650(Compiled
Code))
 at
org/springframework/jms/core/JmsTemplate.receiveSelected(JmsTemplate.java:641(Compiled
Code))
 at org/app/JMSReceiver.receive()
.....................................................................
```

As you can see in the above Thread Strack Traces, such deadlock did originate from our application code which is using the Spring framework API for the JMS consumer implementation (very useful when not using MDB's). The Stack Traces are quite interesting and revealing that both Threads are in a race condition against the same Weblogic JMS consumer session / connection and leading to a deadlock situation:

- Weblogic Thread #8 is attempting to reset and close the current JMS connection
- Weblogic Thread #10 is attempting to use the same JMS Connection / Session in order to create a new JMS consumer
- Thread deadlock is triggered!

Root cause: non Thread safe Spring JMS SingleConnectionFactory implementation

A code review and a quick research from Spring JIRA bug database did reveal the following Thread safe defect below with a perfect correlation with the above analysis:

SingleConnectionFactory's resetConnection is causing deadlocks with underlying OracleAQ's JMS connection – Bug Link

A patch for Spring SingleConnectionFactory was released back in 2009 which did involve adding proper synchronized{} block in order to prevent Thread deadlock in the event of a JMS Connection reset operation:

```
synchronized (connectionMonitor) {
 //if condition added to avoid possible deadlocks when trying to reset the
target connection
 if (!started) {
  this.target.start();
  started = true;
 }
}
```

Solution

Our team is currently planning to integrate this Spring patch in to our production environment shortly. The initial tests performed in our test environment are positive.

Conclusion

I hope this case study has helped understand a real-life Java Thread deadlock problem and how proper Thread Dump analysis skills can allow you to quickly pinpoint the root cause of stuck Thread related problems at the code level.

Java concurrency: the hidden thread deadlocks

Most Java programmers are familiar with the Java thread deadlock concept. It essentially involves 2 threads waiting forever for each other. This condition is often the result of flat (synchronized) or ReentrantLock (read or write) lock-ordering problems.

```
Found one Java-level deadlock:
=============================
"pool-1-thread-2":
 waiting to lock monitor 0x0237ada4 (object 0x272200e8, a java.lang.Object),
 which is held by "pool-1-thread-1"
"pool-1-thread-1":
 waiting to lock monitor 0x0237aa64 (object 0x272200f0, a java.lang.Object),
 which is held by "pool-1-thread-2"
```

The good news is that the HotSpot JVM is always able to detect this condition for you...or is it?

A recent thread deadlock problem affecting an Oracle Service Bus production environment has forced us to revisit this classic problem and identify the existence of "hidden" deadlock situations.

This section will demonstrate and replicate via a simple Java program a very special lock-ordering deadlock condition which is not detected by the latest HotSpot JVM 1.7. You will also find a video here explaining you the Java sample program and the troubleshooting approach used.

The crime scene

I usually like to compare major Java concurrency problems to a crime scene where you play the lead investigator role. In this context, the "crime" is an actual production outage of your client IT environment. Your job is to:

- Collect all the evidences, hints & facts (thread dump, logs, business impact, load figures...)
- Interrogate the witnesses & domain experts (support team, delivery team, vendor, client...)

The next step of your investigation is to analyze the collected information and establish a potential list of one or many "suspects" along with clear proofs. Eventually, you want to narrow it down to a primary suspect or root cause. Obviously the law "innocent until proven guilty" does not apply here, exactly the opposite.

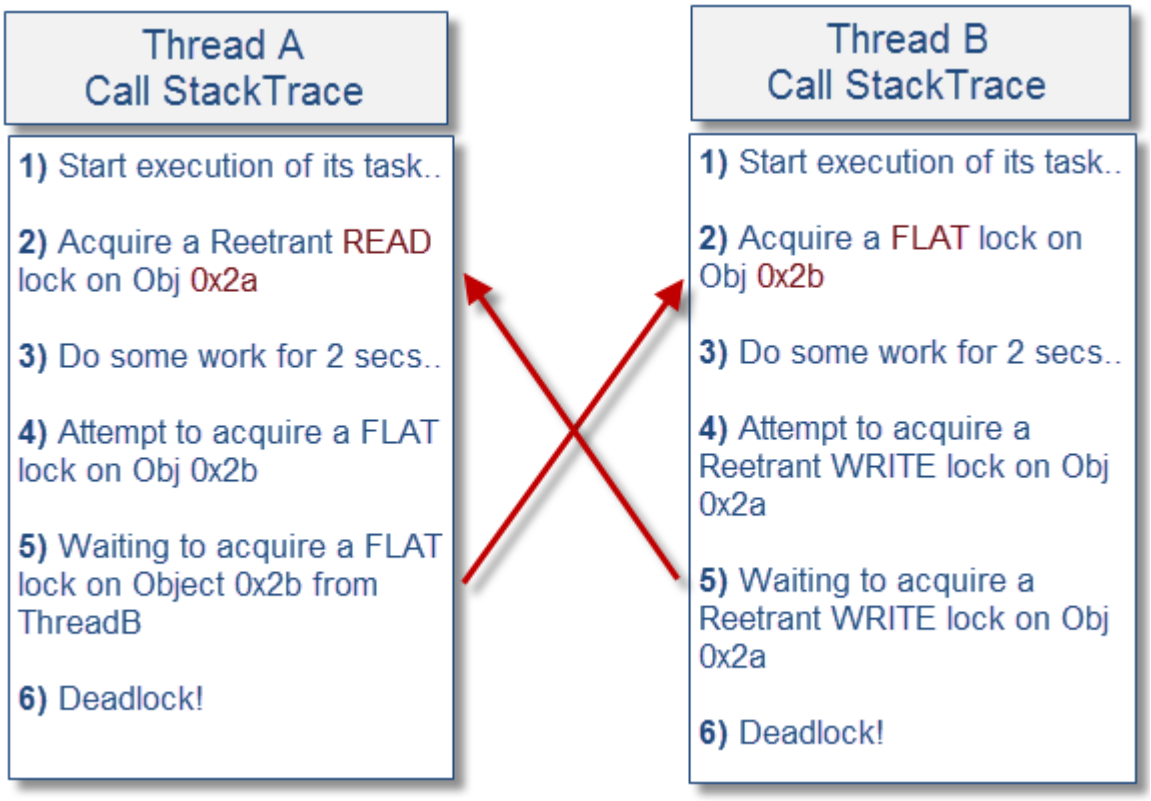
Lack of evidence can prevent you to achieve the above goal. What you will see next is that the lack of deadlock detection by the Hotspot JVM does not necessary prove that you are not dealing with this problem.

The suspect

In this troubleshooting context, the "suspect" is defined as the application or middleware code with the following problematic execution pattern.

- Usage of FLAT lock followed by the usage of ReentrantLock WRITE lock (execution path #1)
- Usage of ReentrantLock READ lock followed by the usage of FLAT lock (execution path #2)
- Concurrent execution performed by 2 Java threads but via a reversed execution order

The above lock-ordering deadlock criteria's can be visualized as per below:



Now let's replicate this problem via our sample Java program and look at the JVM thread dump output.

Sample Java program

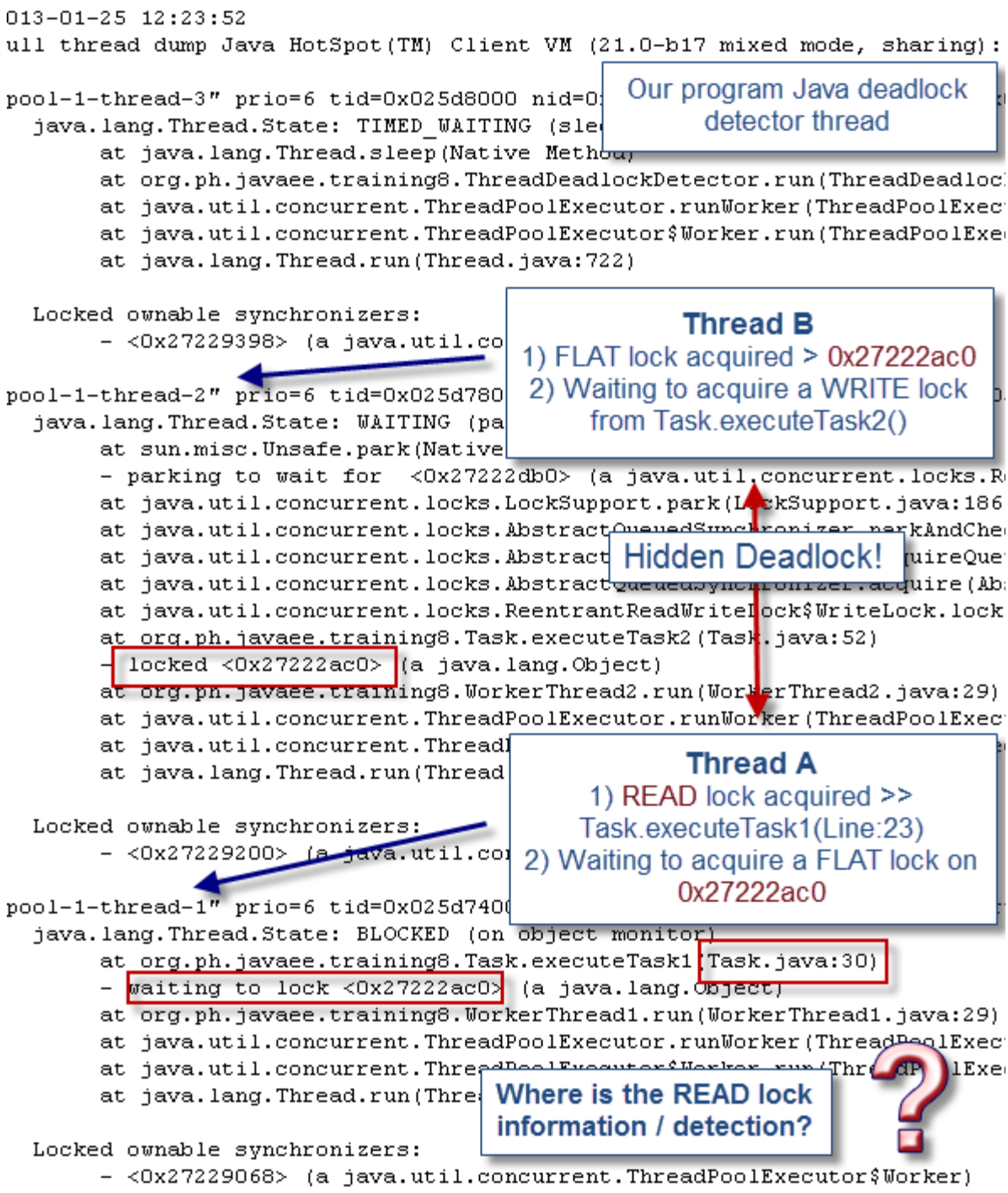
This above deadlock conditions was first identified from our Oracle OSB problem case. We then re-created it via a simple Java program. You can download the entire source code of our program here.

The program is simply creating and firing 2 worker threads. Each of them execute a different execution path and attempt to acquire locks on shared objects but in different orders. We also created a deadlock detector thread for monitoring and logging purposes.

For now, find below the Java class implementing the 2 different execution paths.

```
package org.ph.javaee.training8;

import java.util.concurrent.locks.ReentrantReadWriteLock;

public class Task {

   // Object used for FLAT lock
   private final Object sharedObject = new Object();
   // ReentrantReadWriteLock used for WRITE & READ locks
   private final ReentrantReadWriteLock lock = new ReentrantReadWriteLock();

   public void executeTask1() {

      // 1. Attempt to acquire a ReentrantReadWriteLock READ lock
      lock.readLock().lock();

      // Wait 2 seconds to simulate some work...
      try { Thread.sleep(2000);}catch (Throwable any) {}

      try {
         // 2. Attempt to acquire a Flat lock...
         synchronized (sharedObject) {}
      }
      // Remove the READ lock
      finally {
         lock.readLock().unlock();
      }

      System.out.println("executeTask1() :: Work Done!");
   }

   public void executeTask2() {

      // 1. Attempt to acquire a Flat lock
      synchronized (sharedObject) {

         // Wait 2 seconds to simulate some work...
         try { Thread.sleep(2000);}catch (Throwable any) {}

         // 2. Attempt to acquire a WRITE lock
         lock.writeLock().lock();

         try {
            // Do nothing
         }

         // Remove the WRITE lock
         finally {
            lock.writeLock().unlock();
         }
      }

      System.out.println("executeTask2() :: Work Done!");
   }

   public ReentrantReadWriteLock getReentrantReadWriteLock() {
      return lock;
   }
}
```

As soon ad the deadlock situation was triggered, a JVM thread dump was generated using JvisualVM.



As you can see from the Java thread dump sample. The JVM did not detect this deadlock condition (e.g. no presence of Found one Java-level deadlock) but it is clear these 2 threads are in deadlock

state.

Root cause: ReetrantLock READ lock behavior

The main explanation we found at this point is associated with the usage of the ReetrantLock READ lock. The read locks are normally not designed to have a notion of ownership. Since there is not a record of which thread holds a read lock, this appears to prevent the HotSpot JVM deadlock detector logic to detect deadlock involving read locks.

Some improvements were implemented since then but we can see that the JVM still cannot detect this special deadlock scenario.

Now if we replace the read lock (execution pattern #1) in our program by a write lock, the JVM will finally detect the deadlock condition but why?

```
Found one Java-level deadlock:
=============================
"pool-1-thread-2":
 waiting for ownable synchronizer 0x272239c0, (a
java.util.concurrent.locks.ReentrantReadWriteLock$NonfairSync),
 which is held by "pool-1-thread-1"
"pool-1-thread-1":
 waiting to lock monitor 0x025cad3c (object 0x272236d0, a java.lang.Object),
 which is held by "pool-1-thread-2"
```

```
Java stack information for the threads listed above:
===================================================
"pool-1-thread-2":
 at sun.misc.Unsafe.park(Native Method)
 - parking to wait for <0x272239c0> (a
java.util.concurrent.locks.ReentrantReadWriteLock$NonfairSync)
 at java.util.concurrent.locks.LockSupport.park(LockSupport.java:186)
 at
java.util.concurrent.locks.AbstractQueuedSynchronizer.parkAndCheckInterrupt(Abs
tractQueuedSynchronizer.java:834)
 at
java.util.concurrent.locks.AbstractQueuedSynchronizer.acquireQueued(AbstractQue
uedSynchronizer.java:867)
 at
java.util.concurrent.locks.AbstractQueuedSynchronizer.acquire(AbstractQueuedSyn
chronizer.java:1197)
 at
java.util.concurrent.locks.ReentrantReadWriteLock$WriteLock.lock(ReentrantReadW
riteLock.java:945)
 at org.ph.javaee.training8.Task.executeTask2(Task.java:54)
 - locked <0x272236d0> (a java.lang.Object)
 at org.ph.javaee.training8.WorkerThread2.run(WorkerThread2.java:29)
 at
java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1110)
 at
java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:603)
 at java.lang.Thread.run(Thread.java:722)
"pool-1-thread-1":
 at org.ph.javaee.training8.Task.executeTask1(Task.java:31)
 - waiting to lock <0x272236d0> (a java.lang.Object)
 at org.ph.javaee.training8.WorkerThread1.run(WorkerThread1.java:29)
 at
java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1110)
 at
java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:603)
 at java.lang.Thread.run(Thread.java:722)
```

This is because write locks are tracked by the JVM similar to flat locks. This means the HotSpot JVM deadlock detector appears to be currently designed to detect:

- Deadlock on Object monitors involving FLAT locks
- Deadlock involving Locked ownable synchronizers associated with WRITE locks

The lack of read lock per-thread tracking appears to prevent deadlock detection for this scenario and significantly increase the troubleshooting complexity.

I suggest that you read Doug Lea's comments on this whole issue since concerns were raised back in 2005 regarding the possibility to add per-thread read-hold tracking due to some potential lock overhead.

Find below my troubleshooting recommendations if you suspect a hidden deadlock condition involving read locks:

- Analyze closely the thread call stack trace, it may reveal some code potentially acquiring read locks and preventing other threads to acquire write locks.
- If you are the owner of the code, keep track of the read lock count via the usage of the lock.getReadLockCount() method

OutOfMemoryError patterns

An OutOfMemoryError problem is one of the most frequent and complex problems a Java EE application support person can face with a production system. This section will focus on a particular OOM flavour: PermGen space depletion of a Java HotSpot VM.

Find below some of the most common patterns of OutOfMemoryError due to the depletion of the PermGen space.

Pattern	Symptoms	Possible root cause scenarios	Resolution
OOM observed during or after a migration of a Java EE server to newer version	- OOM may be observed on server start-up at deployment time - OOM may be observed very shortly after server start-up and after 1 or 2+ hours of production traffic	- Higher PermGen capacity is often required due to increased Java EE server vendor code and libraries	- Increase your PermGen space capacity via -XX:MaxPermSize
OOM observed after a certain period of time	- OOM observed after a longer but consistent period of time (days) - PermGen space monitoring will show hourly or daily increase during your application business hours	- There are many possible causes of PermGen space memory leak. The most common is a class loader leak: increasing number of Class objects overtime - Improper JVM arguments like usage of the Xnoclassgc flag (*turn OFF Class garbage collection*)	- Review your JVM HotSpot start-up arguments for any obvious problem like Xnoclassgc flag - Analyse the JVM HotSpot Heap Dump as it can provides some hints on the source of a class loader leak - Investigate any third party API you are using for any potential class loader leak defect - Investigate your application code for any improper use of

			Reflection API and / or dynamic class loading
OOM observed following a redeploy of your application code (*EAR, WAR files...*)	- OOM may be observed during or shortly after your application redeploy process	- Unloading and reloading of your application code can lead to PermGen leak (*class loader leak*) and deplete your PermGen space fairly quickly	- Open a ticket with your Java EE vendor for any known class loader leak issue - Shutdown and restart your server (JVM) post deployment to cleanup any class loader leak

OutOfMemoryError: Java heap space - what is it?

This error message is typically what you will see your middleware server logs (Weblogic, WAS, JBoss etc.) following a JVM OutOfMemoryError condition:

- It is generated from the actual Java HotSpot VM native code
- It is triggered as a result of Java Heap (Young Gen / Old Gen spaces) memory allocation failure (due to Java Heap exhaustion)

Ok, so my application Java Heap is exhausted...how can I monitor and track my application Java Heap?

The simplest way to properly monitor and track the memory footprint of your Java Heap spaces (Young Gen & Old Gen spaces) is to enable verbose GC from your HotSpot VM. Please simply add the following parameters within your JVM start-up arguments:

```
-verbose:gc -XX:+PrintGCDetails -XX:+PrintGCTimeStamps -Xloggc:<app
path>/gc.log
```

You can then follow my tutorial here in order to understand how to read and analyze your HotSpot Java Heap footprint.

Ok thanks, now I can see that I have a big Java Heap memory problem...but how can I fix it?

There are multiple scenarios which can lead to Java Heap depletion such as:

- Java Heap space too small vs. your application traffic & footprint
- Java Heap memory leak (OldGen space slowly growing over time...)
- Sudden and / or rogue Thread(s) consuming large amount of memory in short amount of time etc.

Find below a list of high level steps you can follow in order to further troubleshoot:

- If not done already, enabled verbose GC >> -verbose:gc
- Analyze the verbose GC output and determine the memory footprint of the Java Heap for each of the Java Heap spaces (YoungGen & OldGen)
- Analyze the verbose GC output or use a tool like JConsole to determine if your Java Heap is leaking over time. This can be observed via monitoring of the HotSpot old gen space.
- Monitor your middleware Threads and generate JVM Thread Dumps on a regular basis, especially when a sudden increase of Java Heap utilization is observed. Thread Dump analysis will allow you to pinpoint potential long running Thread(s) allocating a lot of objects on your Java Heap in a short amount of time; if any
- Add the following parameter within your JVM start-up arguments: -XX:HeapDumpOnOutOfMemoryError This will allow your HotSpot VM to generate a binary Heap Dump (HPROF) format. A binary Heap Dump is a critical data allowing to analyze your application memory footprint and / or source(s) of Java Heap memory leak

From a resolution perspective, my recommendation to you is to analyze your Java Heap memory footprint using the generated Heap Dump. The binary Heap Dump (HPROF format) can be analyzed using the free Memory Analyzer tool (MAT). This will allow you to understand your java application footprint and / or pinpoint source(s) of possible memory leak. Once you have a clear picture of the situation, you will be able to resolve your problem by increasing your Java Heap capacity (via -Xms & Xmx arguments) or reducing your application memory footprint and / or eliminating the memory leaks from your application code. Please note that memory leaks are often found in middleware server code and JDK as well.

OutOfMemoryError: Out of swap space - Problem Patterns

In this section we will revisit a common Java HotSpot VM problem that you probably already experienced at some point in your JVM troubleshooting experience on Solaris OS; especially on a 32-bit JVM.

We will provide you with a description of this particular type of OutOfMemoryError, the common problem patterns and the recommended resolution approach.

java.lang.OutOfMemoryError: Out of swap space? - what is it?

This error message is thrown by the Java HotSpot VM (native code) following a failure to allocate native memory from the OS to the HotSpot C-Heap or dynamically expand the Java Heap etc... This problem is very different than a standard OutOfMemoryError (normally due to an exhaustion of the Java Heap or PermGen space).

A typically error found in your application / server logs is:

```
Exception in thread "main" java.lang.OutOfMemoryError: requested 53459 bytes
for ChunkPool::allocate. Out of swap space?
```

Also, please note that depending of the OS that you use (Windows, AIX, Solaris etc.) some OutOfMemoryError due to C-Heap exhaustion may not give you detail such as "Out of swap space". In this case, you will need to review the OOM error Stack Trace and determine if the computing task that triggered the OOM and determine which OutOfMemoryError problem pattern your problem is related to (Java Heap, PermGen or Native Heap exhaustion).

Ok so can I increase the Java Heap via -Xms & -Xmx to fix it?

Definitely not! This is the last thing you want to do as it will make the problem worse. As you learned, the Java HotSpot VM is split between 3 memory spaces (Java Heap, PermGen, C-Heap). For a 32-bit VM, all these memory spaces compete between each other for memory. Increasing the Java Heap space will further reduce capacity of the C-Heap and reserve more memory from the OS.

Your first task is to determine if you are dealing with a C-Heap depletion or OS physical / virtual memory depletion.

Now let's see the most common patterns of this problem.

Common problem patterns

There are multiple scenarios which can lead to a native OutOfMemoryError. I will share with you what I have seen in my past experience as the most common patterns.

- Native Heap (C-Heap) depletion due to too many Java EE applications deployed on a single 32-bit JVM (combined with large Java Heap e.g. 2 GB) * *most common problem* *
- Native Heap (C-Heap) depletion due to a non-optimal Java Heap size e.g. Java Heap too large for the application(s) needs on a single 32-bit JVM
- Native Heap (C-Heap) depletion due to too many created Java Threads e.g. allowing the Java EE container to create too many Threads on a single 32-bit JVM
- OS physical / virtual memory depletion preventing the HotSpot VM to allocate native memory to the C-Heap (32-bit or 64-bit VM)
- OS physical / virtual memory depletion preventing the HotSpot VM to expand its Java Heap or PermGen space at runtime (32-bit or 64-bit VM)
- C-Heap / native memory leak (third party monitoring agent / library, JVM bug etc.)

Troubleshooting and resolution approach

Please keep in mind that each HotSpot native memory problem can be unique and requires its own troubleshooting & resolution approach.

Find below a list of high level steps you can follow in order to further troubleshoot:

- First, determine if the OOM is due to C-Heap exhaustion or OS physical / virtual memory. For this task, you will need to perform close monitoring of your OS memory utilization and Java process size. For example on Solaris, a 32-bit JVM process size can go to about 3.5 GB (technically 4 GB limit) then you can expect some native memory allocation failures. The Java process size monitoring will also allow you to determine if you are dealing with a native memory leak (growing overtime / several days...).

- The OS vendor and version that you use is important as well. For example, some versions of Windows (32-bit) by default support a process size up to 2 GB only (leaving you with minimal flexibility for Java Heap and Native Heap allocations). Please review your OS and determine what is the maximum process size e.g. 2 GB, 3 GB or 4 GB or more (64-bit OS).
- Like the OS, it is also important that you review and determine if you are using a 32-bit VM or 64-bit VM. Native memory depletion for a 64-bit VM typically means that your OS is running out of physical / virtual memory.
- Review your JVM memory settings. For a 32-bit VM, a Java Heap of 2 GB+ can really start to add pressure point on the C-Heap; depending how many applications you have deployed, Java Threads etc. In that case, please determine if you can safely reduce your Java Heap by about 256 MB (as a starting point) and see if it helps improve your JVM memory "balance".
- Analyze the verbose GC output or use a tool like JConsole to determine your Java Heap footprint. This will allow you to determine if you can reduce your Java Heap in a safe manner or not.
- When OutOfMemoryError is observed. Generate a JVM Thread Dump and determine how many Threads are active in your JVM; the more Threads, the more native memory your JVM will use. You will then be able to combine this data with OS, Java process size and verbose GC; allowing to determine where the problem is.

Once you have a clear view of the situation in your environment and root cause, you will be in a better position to explore potential solutions as per below:

- Reduce the Java Heap (if possible / after close monitoring of the Java Heap) in order to give that memory back to the C-Heap / OS.
- Increase the physical RAM / virtual memory of your OS (only applicable if depletion of the OS memory is observed; especially for a 64-bit OS & VM).
- Upgrade your HotSpot VM to 64-bit (for some Java EE applications, a 64-bit VM is more appropriate) or segregate your applications to different JVM's (increase demand on your hardware but reduce utilization of C-Heap per JVM).
- Native memory leak are trickier and requires deeper dive analysis such as analysis of the Solaris pmap / AIX svmon data and review of any third party library (e.g. monitoring agents). Please also review the Oracle Sun Bug database and determine if your HotSpot version you use is exposed to known native memory problems.

OutOfMemoryError: unable to create new native thread

One of the common problems I have observed from Java EE production systems is OutOfMemoryError: unable to create new native thread; error thrown when the HotSpot JVM is unable to further create a new Java thread.

This section will revisit this HotSpot VM error and provide you with recommendations and resolution strategies.

OutOfMemoryError: unable to create new native thread - what is it?

Let's start with a basic explanation. This HotSpot JVM error is thrown when the internal JVM native

code is unable to create a new Java thread. More precisely, it means that the JVM native code was unable to create a new "native" thread from the OS (Solaris, Linux, MAC, Windows...). Unfortunately at this point you won't get more detail than this error, with no indication of why the JVM is unable to create a new thread from the OS.

HotSpot JVM: 32-bit or 64-bit?

Before you go any further in the analysis, one fundamental fact that you must determine from your Java or Java EE environment is which version of HotSpot VM you are using e.g. 32-bit or 64-bit.

Why is it so important? What you will learn shortly is that this JVM problem is very often related to native memory depletion; either at the JVM process or OS level. For now please keep in mind that:

- A 32-bit JVM process is in theory allowed to grow up to 4 GB (even much lower on some older 32-bit Windows versions).
- For a 32-bit JVM process, the C-Heap is in a race with the Java Heap and PermGen space e.g. C-Heap capacity = 2-4 GB - Java Heap size (-Xms, -Xmx) - PermGen size (-XX:MaxPermSize)
- A 64-bit JVM process is in theory allowed to use most of the OS virtual memory available or up to 16 EB (16 million TB)

As you can see, if you allocate a large Java Heap (2 GB+) for a 32-bit JVM process, the native memory space capacity will be reduced automatically, opening the door for JVM native memory allocation failures.

For a 64-bit JVM process, your main concern, from a JVM C-Heap perspective, is the capacity and availability of the OS physical, virtual and swap memory.

OK great but how does native memory affect Java threads creation?

Now back to our primary problem. Another fundamental JVM aspect to understand is that Java threads created from the JVM requires native memory from the OS. You should now start to understand the source of your problem.

The high level thread creation process is as per below:

- A new Java thread is requested from the Java program & JDK.
- The JVM native code then attempt to create a new native thread from the OS.
- The OS then attempts to create a new native thread as per attributes which include the thread stack size. Native memory is then allocated (reserved) from the OS to the Java process native memory space; assuming the process has enough address space (e.g. 32-bit process) to honour the request.
- The OS will refuse any further native thread & memory allocation if the 32-bit Java process size has depleted its memory address space e.g. 2 GB, 3 GB or 4 GB process size limit.
- The OS will also refuse any further Thread & native memory allocation if the virtual memory of the OS is depleted (including Solaris swap space depletion since thread access to the stack can generate a SIGBUS error, crashing the JVM (also see here).

In summary:

- Java threads creation require native memory available from the OS; for both 32-bit & 64-bit JVM processes
- For a 32-bit JVM, Java thread creation also requires memory available from the C-Heap or process address space

Problem diagnostic

Now that you understand native memory and JVM thread creation a little better, is it now time to look at your problem. As a starting point, I suggest that your follow the analysis approach below:

- Determine if you are using HotSpot 32-bit or 64-bit JVM
- When problem is observed, take a JVM Thread Dump and determine how many Threads are active
- Monitor closely the Java process size utilization before and during the OOM problem replication
- Monitor closely the OS virtual memory utilization before and during the OOM problem replication; including the swap memory space utilization if using Solaris OS

Proper data gathering as per above will allow you to collect the proper data points, allowing you to perform the first level of investigation. The next step will be to look at the possible problem patterns and determine which one is applicable for your problem case.

Problem pattern #1 - C-Heap depletion (32-bit JVM)

From my experience, OutOfMemoryError: unable to create new native thread is quite common for 32-bit JVM processes. This problem is often observed when too many threads are created vs. C-Heap capacity.
JVM Thread Dump analysis and Java process size monitoring will allow you to determine if this is the cause.

Problem pattern #2 - OS virtual memory depletion (64-bit JVM)

In this scenario, the OS virtual memory is fully depleted. This could be due to a few 64-bit JVM processes taking lot memory e.g. 10 GB+ and / or other high memory footprint rogue processes. Again, Java process size & OS virtual memory monitoring will allow you to determine if this is the cause.

Problem pattern #3 - OS virtual memory depletion (32-bit JVM)

The third scenario is less frequent but can still be observed. The diagnostic can be a bit more complex but the key analysis point will be to determine which processes are causing a full OS virtual memory depletion. Your 32-bit JVM processes could be either the source or the victim such as rogue processes using most of the OS virtual memory and preventing your 32-bit JVM processes to reserve more native memory for its thread creation process.

Please note that this problem can also manifest itself as a full JVM crash (as per below sample) when

running out of OS virtual memory or swap space on Solaris.

```
#
# A fatal error has been detected by the Java Runtime Environment:
#
# java.lang.OutOfMemoryError: requested 32756 bytes for ChunkPool::allocate.
Out of swap space?
#
#  Internal Error (allocation.cpp:166), pid=2290, tid=27
#  Error: ChunkPool::allocate
#
# JRE version: 6.0_24-b07
# Java VM: Java HotSpot(TM) Server VM (19.1-b02 mixed mode solaris-sparc )
# If you would like to submit a bug report, please visit:
#   http://java.sun.com/webapps/bugreport/crash.jsp
#

---------------  T H R E A D  ---------------

Current thread (0x003fa800):  JavaThread "CompilerThread1" daemon
[_thread_in_native, id=27, stack(0x65380000,0x65400000)]

Stack: [0x65380000,0x65400000],  sp=0x653fd758,  free space=501k
Native frames: (J=compiled Java code, j=interpreted, Vv=VM code, C=native code)
```

Native memory depletion: symptom or root cause?

You now understand your problem and know which problem pattern you are dealing with. You are now ready to provide recommendations to address the problem... are you?

Your work is not done yet, please keep in mind that this JVM OOM event is often just a "symptom" of the actual root cause of the problem. The root cause is typically much deeper so before providing recommendations to your client I recommend that you really perform deeper analysis. The last thing you want to do is to simply address and mask the symptoms. Solutions such as increasing OS physical / virtual memory or upgrading all your JVM processes to 64-bit should only be considered once you have a good view on the root cause and production environment capacity requirements.

The next fundamental question to answer is how many threads were active at the time of the OutOfMemoryError? In my experience with Java EE production systems, the most common root cause is actually the application and / or Java EE container attempting to create too many threads at a given time when facing non happy paths such as thread stuck in a remote IO call, thread race conditions etc. In this scenario, the Java EE container can start creating too many threads when attempting to honour incoming client requests, leading to increase pressure point on the C-Heap and native memory allocation. Bottom line, before blaming the JVM, please perform your due diligence and determine if you are dealing with an application or Java EE container thread tuning problem as the root cause.

Once you understand and address the root cause (source of thread creations), you can then work on tuning your JVM and OS memory capacity in order to make it more fault tolerant and better "survive"

these sudden thread surge scenarios.

Recommendations:

- First, quickly rule out any obvious OS memory (physical & virtual memory) & process capacity (e.g. ulimit) problem.
- Perform a JVM Thread Dump analysis and determine the source of all the active threads vs. an established baseline. Determine what is causing your Java application or Java EE container to create so many threads at the time of the failure.
- Please ensure that your monitoring tools closely monitor both your Java VM processes size & OS virtual memory. This crucial data will be required in order to perform a full root cause analysis. Please remember that a 32-bit Java process size is limited between 2 GB - 4 GB depending of your OS.
- Look at all running processes and determine if your JVM processes are actually the source of the problem or victim of other processes consuming all the virtual memory.
- Revisit your Java EE container thread configuration & JVM thread stack size. Determine if the Java EE container is allowed to create more threads than your JVM process and / or OS can handle.
- Determine if the Java Heap size of your 32-bit JVM is too large, preventing the JVM to create enough threads to fulfill your client requests. In this scenario, you will have to consider reducing your Java Heap size (if possible), vertical scaling or upgrade to a 64-bit JVM.

Capacity planning analysis to the rescue

Lack of capacity planning analysis is often the source of the problem. Any comprehensive load and performance testing exercise should also properly determine the Java EE container threads, JVM & OS native memory requirement for your production environment; including impact measurements of "non-happy" paths. This approach will allow your production environment to stay away from this type of problem and lead to better system scalability and stability in the long run.

ClassNotFoundException: How to resolve

This section will provide you with an overview of this common Java exception, a sample Java program to support your learning process and resolution strategies.

As per the Oracle documentation, ClassNotFoundException is thrown following the failure of a class loading call, using its string name, as per below:

- The Class.forName method
- The ClassLoader.findSystemClass method
- The ClassLoader.loadClass method

In other words, it means that one particular Java class was not found or could not be loaded at "runtime" from your application current context class loader.

This problem can be particularly confusing for Java beginners. This is why I always recommend to

Java developers to learn and refine their knowledge on Java class loaders. Unless you are involved in dynamic class loading and using the Java Reflection API, chances are that the ClassNotFoundException error you are getting is not from your application code but from a referencing API. Another common problem pattern is a wrong packaging of your application code. We will get back to the resolution strategies at the end of the section.

java.lang.ClassNotFoundException: Sample Java program

Now find below a very simple Java program which simulates the 2 most common ClassNotFoundException scenarios via Class.forName() & ClassLoader.loadClass(). Please simply copy/paste and run the program with the IDE of your choice (Eclipse IDE was used for this example).

The Java program allows you to choose between problem scenario #1 or problem scenario #2 as per below. Simply change to 1 or 2 depending of the scenario you want to study.

```
# Class.forName()
private static final int PROBLEM_SCENARIO = 1;

# ClassLoader.loadClass()
private static final int PROBLEM_SCENARIO = 2;
```

```
package org.ph.javaee.training5;

public class ClassNotFoundExceptionSimulator {

 private static final String CLASS_TO_LOAD = "org.ph.javaee.training5.ClassA";
 private static final int PROBLEM_SCENARIO = 1;

 public static void main(String[] args) {

 System.out.println("java.lang.ClassNotFoundException Simulator - Training 5");
 System.out.println("Author: Pierre-Hugues Charbonneau");
 System.out.println("http://javaeesupportpatterns.blogspot.com");

 switch(PROBLEM_SCENARIO) {

 // Scenario #1 - Class.forName()
 case 1:

 System.out.println("\n** Problem scenario #1: Class.forName() **\n");
 try {
 Class<?> newClass = Class.forName(CLASS_TO_LOAD);

 System.out.println("Class "+newClass+" found successfully!");

 } catch (ClassNotFoundException ex) {

 ex.printStackTrace();

 System.out.println("Class "+CLASS_TO_LOAD+" not found!");

 } catch (Throwable any) {
 System.out.println("Unexpected error! "+any);
 }

 break;

 // Scenario #2 - ClassLoader.loadClass()
 case 2:

 System.out.println("\n** Problem scenario #2: ClassLoader.loadClass() **\n");
 try {
 ClassLoader classLoader = Thread.currentThread().getContextClassLoader();
 Class<?> callerClass = classLoader.loadClass(CLASS_TO_LOAD);

 Object newClassAInstance = callerClass.newInstance();

 System.out.println("SUCCESS!: "+newClassAInstance);
 } catch (ClassNotFoundException ex) {

 ex.printStackTrace();

 System.out.println("Class "+CLASS_TO_LOAD+" not found!");

 } catch (Throwable any) {
 System.out.println("Unexpected error! "+any);
 }

 break;
 }

 System.out.println("\nSimulator done!");
 }
}
```

```
package org.ph.javaee.training5;

/**
 * ClassA
 * @author Pierre-Hugues Charbonneau
 *
 */
public class ClassA {

private final static Class<ClassA> CLAZZ = ClassA.class;

   static {
      System.out.println("Class loading of "+CLAZZ+" from ClassLoader
'"+CLAZZ.getClassLoader()+"' in progress...");
   }

   public ClassA() {
      System.out.println("Creating a new instance of "+ClassA.class.getName()+"...");

      doSomething();
   }

   private void doSomething() {
      // Nothing to do...
   }
}
```

If you run the program as is, you will see the output as per below for each scenario:

#Scenario 1 output (baseline)

```
java.lang.ClassNotFoundException Simulator - Training 5
Author: Pierre-Hugues Charbonneau
http://javaeesupportpatterns.blogspot.com

** Problem scenario #1: Class.forName() **

Class loading of class org.ph.javaee.training5.ClassA from ClassLoader
'sun.misc.Launcher$AppClassLoader@bfbdb0' in progress...
Class class org.ph.javaee.training5.ClassA found successfully!

Simulator done!
```

#Scenario 2 output (baseline)

```
java.lang.ClassNotFoundException Simulator - Training 5
Author: Pierre-Hugues Charbonneau
http://javaeesupportpatterns.blogspot.com

** Problem scenario #2: ClassLoader.loadClass() **

Class loading of class org.ph.javaee.training5.ClassA from ClassLoader
'sun.misc.Launcher$AppClassLoader@2a340e' in progress...
Creating a new instance of org.ph.javaee.training5.ClassA...
SUCCESS!: org.ph.javaee.training5.ClassA@6eb38a

Simulator done!
```

For the “baseline” run, the Java program is able to load `ClassA` successfully.

Now let’s voluntary change the full name of `ClassA` and re-run the program for each scenario. The following output can be observed:

#ClassA changed to ClassB

#Scenario 1 output (problem replication)

```
java.lang.ClassNotFoundException Simulator - Training 5
Author: Pierre-Hugues Charbonneau
http://javaeesupportpatterns.blogspot.com

** Problem scenario #1: Class.forName() **

java.lang.ClassNotFoundException: org.ph.javaee.training5.ClassB
        at java.net.URLClassLoader$1.run(URLClassLoader.java:366)
        at java.net.URLClassLoader$1.run(URLClassLoader.java:355)
        at java.security.AccessController.doPrivileged(Native Method)
        at java.net.URLClassLoader.findClass(URLClassLoader.java:354)
        at java.lang.ClassLoader.loadClass(ClassLoader.java:423)
        at sun.misc.Launcher$AppClassLoader.loadClass(Launcher.java:308)
        at java.lang.ClassLoader.loadClass(ClassLoader.java:356)
        at java.lang.Class.forName0(Native Method)
        at java.lang.Class.forName(Class.java:186)
        at
org.ph.javaee.training5.ClassNotFoundExceptionSimulator.main(ClassNotFoundExcep
tionSimulator.java:29)
Class org.ph.javaee.training5.ClassB not found!

Simulator done!
```

#Scenario 2 output (problem replication)

```
java.lang.ClassNotFoundException Simulator - Training 5
Author: Pierre-Hugues Charbonneau
http://javaeesupportpatterns.blogspot.com

** Problem scenario #2: ClassLoader.loadClass() **

java.lang.ClassNotFoundException: org.ph.javaee.training5.ClassB
        at java.net.URLClassLoader$1.run(URLClassLoader.java:366)
        at java.net.URLClassLoader$1.run(URLClassLoader.java:355)
        at java.security.AccessController.doPrivileged(Native Method)
        at java.net.URLClassLoader.findClass(URLClassLoader.java:354)
        at java.lang.ClassLoader.loadClass(ClassLoader.java:423)
        at sun.misc.Launcher$AppClassLoader.loadClass(Launcher.java:308)
        at java.lang.ClassLoader.loadClass(ClassLoader.java:356)
        at
org.ph.javaee.training5.ClassNotFoundExceptionSimulator.main(ClassNotFoundExcep
tionSimulator.java:51)
Class org.ph.javaee.training5.ClassB not found!

Simulator done!
```

What happened? Well since we changed the full class name to org.ph.javaee.training5.ClassB, such class was not found at runtime (does not exist), causing both Class.forName() and ClassLoader.loadClass() calls to fail.

You can also replicate this problem by packaging each class of this program to its own JAR file and then omit ting the jar file containing ClassA.class from the main class path Please try this and see the results for yourself... (hint: NoClassDefFoundError)

Now let's jump to the resolution strategies.

java.lang.ClassNotFoundException: Resolution strategies

Now that you understand this problem, it is now time to resolve it. Resolution can be fairly simple or very complex depending of the root cause.

- Don't jump on complex root causes too quickly, rule out the simplest causes first.
- First review the java.lang.ClassNotFoundException stack trace as per the above and determine which Java class was not loaded properly at runtime e.g. application code, third party API, Java EE container itself etc.
- Identify the caller e.g. Java class you see from the stack trace just before the Class.forName() or ClassLoader.loadClass() calls. This will help you understand if your application code is at fault vs. a third party API.
- Determine if your application code is not packaged properly e.g. missing JAR file(s) from your classpath.
- If the missing Java class is not from your application code, then identify if it belongs to a third party API you are using as per of your Java application. Once you identify it, you will need to

add the missing JAR file(s) to your runtime classpath or web application WAR/EAR file.
- If still struggling after multiple resolution attempts, this could means a more complex class loader hierarchy problem. In this case, please review the NoClassDefFoundError section below for more examples and resolution strategies.

NoClassDefFoundError Problem patterns

Getting a java.lang.NoClassDefFoundError when supporting a Java EE application is quite common and at the same time complicated to resolve.

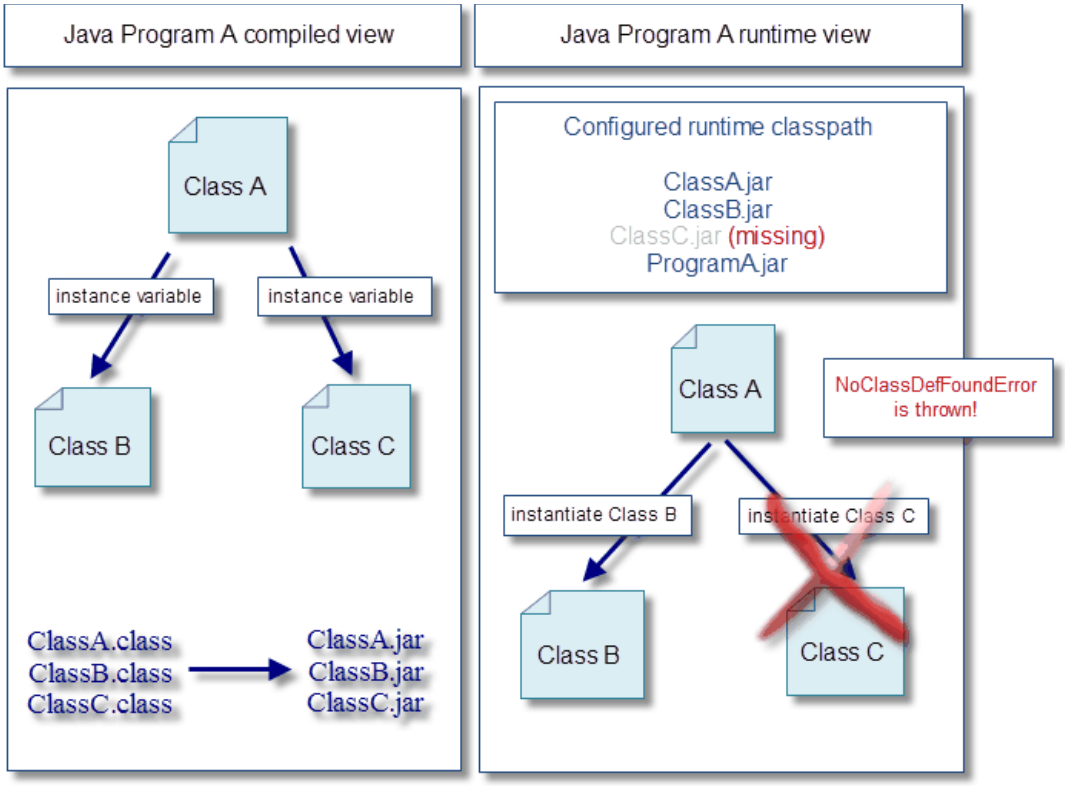
The section will provide you with the common problem patterns responsible for java.lang.NoClassDefFoundError problems.

java.lang.NoClassDefFoundError- what is it?

This runtime error is thrown by the JVM when it tries to load the definition of a Class and when such Class definition could not be found in the current Class loader tree.

This normally means that the compiled version of the reference to this Class was done successfully but that such reference at runtime can not be found.

Sound confusing? Let's have a look at the visual diagram below so you can better understand this fundamental problem.



Now if you are interested, find below the source code of our sample program along with java.lang.NoClassDefFoundError error.

```
package com.cgi.tools.java;

public class ClassA {
	private ClassB instanceB = null;
	private ClassC instanceC = null;

	public ClassA() {
		instanceB = new ClassB();
		instanceC = new ClassC();
	}
}
```

```
// ClassB.java
package com.cgi.tools.java;

public class ClassB {

}
```

```
// ClassC.java
package com.cgi.tools.java;

public class ClassC {

}
```

```
package com.cgi.tools.java;

public class ProgramA {

 /**
 * @param args
 */
 public static void main(String[] args) {

 try {
 ClassA instanceA = new ClassA();

 System.out.println("ClassA instance created properly!");
 }
 catch (Throwable any) {
 System.out.println("Unexpected problem! "+any.getMessage()+" ["+any+"]");
 }
 }

}
```

ProgramA runtime classpath and output – with ClassC.jar

```
java -classpath ClassA.jar;ClassB.jar;ClassC.jar;ProgramA.jar
com.cgi.tools.java.ProgramA

ClassA instance created properly!
```

ProgramA runtime classpath and output – without ClassC.jar

// We voluntarily omitted to add ClassC.jar in the System classpath

```
java -classpath ClassA.jar;ClassB.jar;ProgramA.jar
com.cgi.tools.java.ProgramA

Unexpected problem! com/cgi/tools/java/ClassC

[java.lang.NoClassDefFoundError: com/cgi/tools/java/ClassC]
```

What are the most common scenarios causing NoClassDefFoundError?

There are a few common scenarios which can lead to NoClassDefFoundError in your Java EE environment or standalone Java program.

Problem pattern #1 - Missing vendor or third party library in System classpath or Java EE App classloader

A missing Java library of your Java EE server itself (Weblogic, WAS, JBoss etc.) or third party (Apache, Spring, Hibernate etc.) is the most common program; exactly like our above sample program.

Solution

Resolution requires proper root cause analysis as per below recommended steps:

1. Review the NoClassDefFoundError error and identify the missing Java Class
2. Search through your local development and / or build environment and identify which Jar file contains the missing Java Class
3. Once jar file(s) is / are identified, compare your local / build classpath with your production / problematic environment
4. Resolution may include adding the missing JAR file(s) to the System class path or to your application EAR file for example

Problem pattern #2 - Vendor or third party library version mismatch in System classpath or Java EE App classloader

This problem pattern is less common but trickier to pinpoint the root cause. This is a normally caused by using wrong version of a shared third party library like Apache commons logging etc.

Solution

The resolution is quite similar to pattern #1:

1. Review the NoClassDefFoundError error and identify the missing Java Class along with the referrer (very important)
2. Search through your local development and / or build environment and identify which Jar file contains the missing Java Class
3. Search through your local development and / or build environment and identify which Jar file contains the referrer Java Class
4. Once jar file(s) is / are identified, compare your local / build classpath with your production / problematic environment
5. Resolution may include replacing the problematic JAR file(s) with the right version as per the third party API documentation; this might include replacement of the JAR file referrer depending on your root cause analysis results

Problem pattern #3 - static{} block code failure

This problem pattern is also quite common and can take some time to pinpoint. Java offers the capability to write some code to be executed once in life time of the JVM / Class loader. This is achieved via a static{} block, called static initializer, normally located right after the class instance variables.

Unfortunately, proper error handling and "non happy paths" for static initializer code blocks are often overlooked which opens the door for problems.

Any failure such as an uncaught Exception will prevent such Java class to be loaded to its class loader. The pattern is as per below:

- the first attempt to load the class will generate a java.lang.ExceptionInInitializerError; preventing the class loader to load the referenced class
- subsequent calls will then generate a java.lang.NoClassDefFoundError from any other referencing classes in a consistent manner until the problem is resolved and the JVM restarted (or live redeploy via your Java EE server redeploy task)

Solution

Resolution requires proper root cause analysis as per below recommended steps:

1. Review the NoClassDefFoundError error and identify the affected Java Class
2. Perform a code review of the affected Java class and see if any static{} initializer block can be found
3. If found, review the error handling and add proper try{} catch{} along with proper logging in order to understand the root cause of the static block code failure
4. Compile, redeploy, retest and confirm problem resolution

NoClassDefFoundError – How to resolve

Exception in thread "main" java.lang.NoClassDefFoundError is one of the most common and difficult problems that you can face when developing Java EE enterprise or standalone Java applications. The complexity of the root cause analysis and resolution process mainly depend of the size of your Java EE middleware environment, especially given the high number of class loaders present across the various Java EE applications.

As mentioned before, this runtime error is thrown by the JVM when there is an attempt by a ClassLoader to load the definition of a Class (Class referenced in your application code etc.) and when such Class definition could not be found within the current ClassLoader tree.

Basically, this means that such Class definition was found at compiled time but is not found at runtime.

Java ClassLoader overview

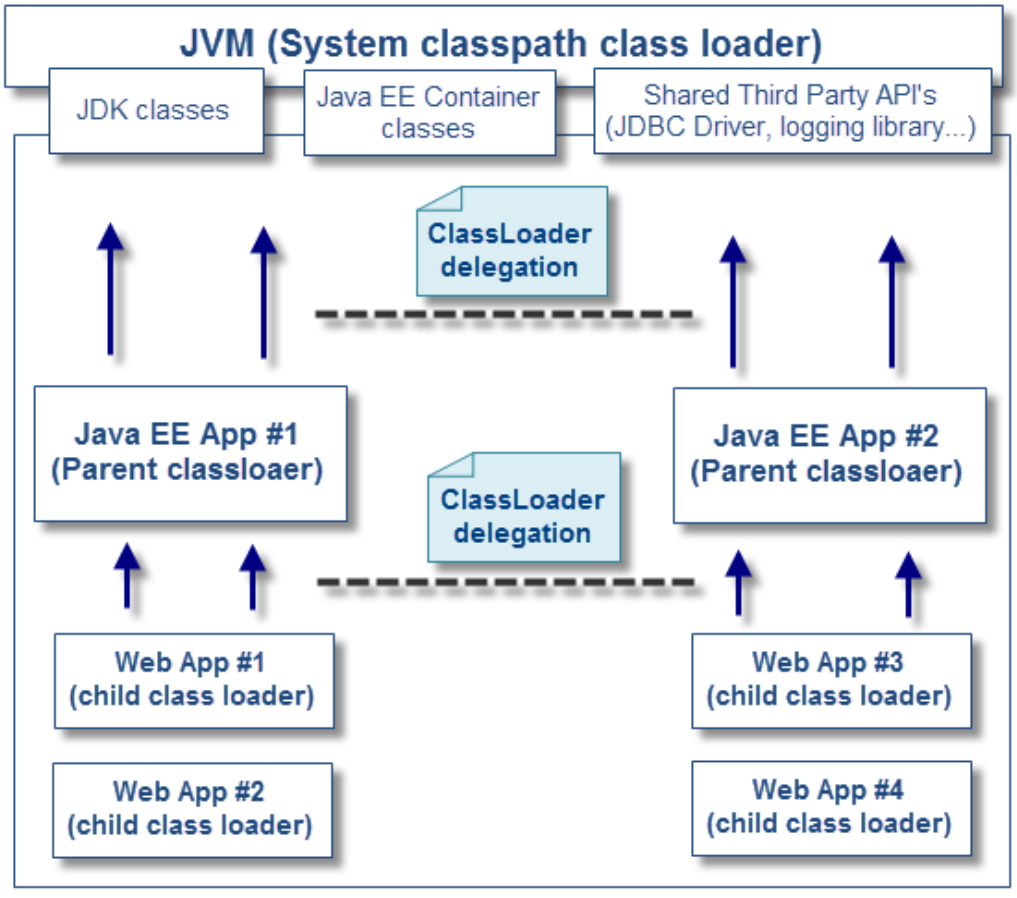
Before going any further, it is very important that you have a high level of understanding of the Java ClassLoader principles. Quite often individuals debugging NoClassDefFoundError problems are struggling because they are lacking proper knowledge and understanding of Java ClassLoader

principles; preventing them to pinpoint the root cause.

A class loader is a Java object responsible for loading classes. Basically a class loader attempts to locate or generate data that constitutes a definition for the class. One of the key points to understand is that Java class loaders by default use a delegation model to search for classes. Each instance of ClassLoader has an associated parent class loader. So let's say that your application class loader needs to load class A. The first thing that it will attempt to do is to delegate the search for Class A to its parent class loader before attempting to find the Class A itself. You can end up with a large class loader chain with many parent class loaders up to the JVM system classpath bootstrap class loader.

What is the problem? Well if Class A is found from a particular parent class loader then it will be loaded by such parent which open the doors for NoClassDefFoundError if you are expecting Class A to be loaded by your application (child) class loader. For example, third part JAR file dependencies could only be present to your application child class loader.

Now let's visualize this whole process in the context of a Java EE enterprise environment so you can better understand.



As you can see, any code loaded by the child class loader (Web application) will first delegate to the parent class loader (Java EE App). Such parent class loader will then delegate to the JVM system class path class loader. If no such class is found from any parent class loader then the Class will be

loaded by the child class loader (assuming that the class was found). Please note that Java EE containers such as Oracle Weblogic have mechanisms to override this default class loader delegation behavior.

NoClassDefFoundError problem case 1 - missing JAR file

The first problem case we will cover is related to a Java program packaging and / or classpath problem. A typical Java program can include one or many JAR files created at compile time. NoClassDefFoundError can often be observed when you forget to add JAR file(s) containing Java classes referenced by your Java or Java EE application.

This type of problem is normally not hard to resolve once you analyze the Java Exception and missing Java class name.

Sample Java program

The following simple Java program is split as per below:

- The main Java program `NoClassDefFoundErrorSimulator`
- The caller Java class `CallerClassA`
- The referencing Java class `ReferencingClassA`
- A util class for ClassLoader and logging related facilities `JavaEETrainingUtil`

This program is simple attempting to create a new instance and execute a method of the Java class `CallerClassA` which is referencing the class `ReferencingClassA`. It will demonstrate how a simple classpath problem can trigger NoClassDefFoundError. The program is also displaying detail on the current class loader chain at class loading time in order to help you keep track of this process. This will be especially useful for future and more complex problem cases when dealing with larger class loader chains.

```
#### NoClassDefFoundErrorSimulator.java
package org.ph.javaee.training1;

import org.ph.javaee.training.util.JavaEETrainingUtil;

/**
 * NoClassDefFoundErrorTraining1
 * @author Pierre-Hugues Charbonneau
 *
 */
public class NoClassDefFoundErrorSimulator {


	/**
	 * @param args
	 */
	public static void main(String[] args) {
		System.out.println("java.lang.NoClassDefFoundError Simulator - Training
1");
		System.out.println("Author: Pierre-Hugues Charbonneau");
		System.out.println("http://javaeesupportpatterns.blogspot.com");

		// Print current Classloader context
		System.out.println("\nCurrent ClassLoader chain:
"+JavaEETrainingUtil.getCurrentClassloaderDetail());

		// 1. Create a new instance of CallerClassA
		CallerClassA caller = new CallerClassA();

		// 2. Execute method of the caller
		caller.doSomething();

		System.out.println("done!");
	}
}
```

```
#### CallerClassA.java
package org.ph.javaee.training1;

import org.ph.javaee.training.util.JavaEETrainingUtil;

/**
 * CallerClassA
 * @author Pierre-Hugues Charbonneau
 *
 */
public class CallerClassA {

	private final static String CLAZZ = CallerClassA.class.getName();

	static {
		System.out.println("Classloading of "+CLAZZ+" in
progress..."+JavaEETrainingUtil.getCurrentClassloaderDetail());
	}

	public CallerClassA() {
		System.out.println("Creating a new instance of
"+CallerClassA.class.getName()+"...");
	}

	public void doSomething() {

		// Create a new instance of ReferencingClassA
		ReferencingClassA referencingClass = new ReferencingClassA();
	}
}
```

```
#### ReferencingClassA.java
package org.ph.javaee.training1;

import org.ph.javaee.training.util.JavaEETrainingUtil;

/**
 * ReferencingClassA
 * @author Pierre-Hugues Charbonneau
 *
 */
public class ReferencingClassA {

	private final static String CLAZZ = ReferencingClassA.class.getName();

	static {
		System.out.println("Classloading of "+CLAZZ+" in
progress..."+JavaEETrainingUtil.getCurrentClassloaderDetail());
	}

	public ReferencingClassA() {
		System.out.println("Creating a new instance of
"+ReferencingClassA.class.getName()+"...");
	}

	public void doSomething() {
		//nothing to do...
	}
}
```

```
#### JavaEETrainingUtil.java
package org.ph.javaee.training.util;

import java.util.Stack;
import java.lang.ClassLoader;

public class JavaEETrainingUtil {

        public static String getCurrentClassloaderDetail() {

                StringBuffer classLoaderDetail = new StringBuffer();
                Stack<ClassLoader> classLoaderStack = new Stack<ClassLoader>();

                ClassLoader currentClassLoader =
Thread.currentThread().getContextClassLoader();


classLoaderDetail.append("\n-------------------------------------------------------------------
--\n");

                // Build a Stack of the current ClassLoader chain
                while (currentClassLoader != null) {

                        classLoaderStack.push(currentClassLoader);

                        currentClassLoader = currentClassLoader.getParent();
                }

                // Print ClassLoader parent chain
                while(classLoaderStack.size() > 0) {

                        ClassLoader classLoader = classLoaderStack.pop();

                        // Print current
                        classLoaderDetail.append(classLoader);

                        if (classLoaderStack.size() > 0) {
                               classLoaderDetail.append("\n--- delegation ---\n");
                        } else {
                               classLoaderDetail.append(" **Current ClassLoader**");
                        }
                }


classLoaderDetail.append("\n-------------------------------------------------------------------
--\n");

                return classLoaderDetail.toString();
        }
}
```

Problem reproduction

In order to replicate the problem, we will simply “voluntary” omit one of the JAR files from the classpath that contains the referencing Java class `ReferencingClassA`.

The Java program is packaged as per below:

- MainProgram.jar (contains `NoClassDefFoundErrorSimulator.class` and `JavaEETrainingUtil.class`)
- CallerClassA.jar (contains `CallerClassA.class`)
- ReferencingClassA.jar (contains `ReferencingClassA.class`)

Now, let’s run the program as is:

Baseline (normal execution)

```
..\bin>java -classpath CallerClassA.jar;ReferencingClassA.jar;MainProgram.jar
org.ph.javaee.training1.NoClassDefFoundErrorSimulator

java.lang.NoClassDefFoundError Simulator - Training 1
Author: Pierre-Hugues Charbonneau
http://javaeesupportpatterns.blogspot.com

Current ClassLoader chain:
---------------------------------------------------------------
sun.misc.Launcher$ExtClassLoader@17c1e333
--- delegation ---
sun.misc.Launcher$AppClassLoader@214c4ac9 **Current ClassLoader**
---------------------------------------------------------------

Classloading of org.ph.javaee.training1.CallerClassA in progress...
---------------------------------------------------------------
sun.misc.Launcher$ExtClassLoader@17c1e333
--- delegation ---
sun.misc.Launcher$AppClassLoader@214c4ac9 **Current ClassLoader**
---------------------------------------------------------------

Creating a new instance of org.ph.javaee.training1.CallerClassA...
Classloading of org.ph.javaee.training1.ReferencingClassA in progress...
---------------------------------------------------------------
sun.misc.Launcher$ExtClassLoader@17c1e333
--- delegation ---
sun.misc.Launcher$AppClassLoader@214c4ac9 **Current ClassLoader**
---------------------------------------------------------------

Creating a new instance of org.ph.javaee.training1.ReferencingClassA...
done!
```

For the initial run (baseline), the main program was able to create a new instance of `CallerClassA`

and execute its method successfully; including successful class loading of the referencing class `ReferencingClassA`.

Problem reproduction run (with removal of ReferencingClassA.jar)

```
../bin>java -classpath CallerClassA.jar;MainProgram.jar
org.ph.javaee.training1.NoClassDefFoundErrorSimulator

java.lang.NoClassDefFoundError Simulator - Training 1
Author: Pierre-Hugues Charbonneau
http://javaeesupportpatterns.blogspot.com

Current ClassLoader chain:
-----------------------------------------------------------------
sun.misc.Launcher$ExtClassLoader@17c1e333
--- delegation ---
sun.misc.Launcher$AppClassLoader@214c4ac9 **Current ClassLoader**
-----------------------------------------------------------------

Classloading of org.ph.javaee.training1.CallerClassA in progress...
-----------------------------------------------------------------
sun.misc.Launcher$ExtClassLoader@17c1e333
--- delegation ---
sun.misc.Launcher$AppClassLoader@214c4ac9 **Current ClassLoader**
-----------------------------------------------------------------

Creating a new instance of org.ph.javaee.training1.CallerClassA...
Exception in thread "main" java.lang.NoClassDefFoundError:
org/ph/javaee/training1/ReferencingClassA
        at
org.ph.javaee.training1.CallerClassA.doSomething(CallerClassA.java:25)
        at
org.ph.javaee.training1.NoClassDefFoundErrorSimulator.main(NoClassDefFoundError
Simulator.java:28)
Caused by: java.lang.ClassNotFoundException:
org.ph.javaee.training1.ReferencingClassA
        at java.net.URLClassLoader$1.run(Unknown Source)
        at java.net.URLClassLoader$1.run(Unknown Source)
        at java.security.AccessController.doPrivileged(Native Method)
        at java.net.URLClassLoader.findClass(Unknown Source)
        at java.lang.ClassLoader.loadClass(Unknown Source)
        at sun.misc.Launcher$AppClassLoader.loadClass(Unknown Source)
        at java.lang.ClassLoader.loadClass(Unknown Source)
        ... 2 more
```

What happened? The removal of the ReferencingClassA.jar, containing `ReferencingClassA`, did prevent the current class loader to locate this referencing Java class at runtime leading to ClassNotFoundException and NoClassDefFoundError.

This is the typical Exception that you will get if you omit JAR file(s) from your Java start-up classpath or within an EAR / WAR for Java EE related applications.

ClassLoader view

Now let's review the ClassLoader chain so you can properly understand this problem case. As you saw from the Java program output logging, the following Java ClassLoaders were found:

```
Classloading of org.ph.javaee.training1.CallerClassA in progress...
-----------------------------------------------------------------
sun.misc.Launcher$ExtClassLoader@17c1e333
--- delegation ---
sun.misc.Launcher$AppClassLoader@214c4ac9 **Current ClassLoader**
-----------------------------------------------------------------
```

*** Please note that the Java bootstrap class loader is responsible to load the core JDK classes and is written in native code ***

sun.misc.Launcher$AppClassLoader
This is the system class loader responsible to load our application code found from the Java classpath specified at start-up.

##sun.misc.Launcher$ExtClassLoader
This is the extension class loader responsible to load code in the extensions directories (<JAVA_HOME>/lib/ext, or any other directory specified by the java.ext.dirs system property).

As you can see from the Java program logging output, the extension class loader is the actual super parent of the system class loader. Our sample Java program was loaded at the system class loader level. Please note that this class loader chain is very simple for this problem case since we did not create child class loaders at this point.

Recommendations and resolution strategies

Now find below my recommendations and resolution strategies for NoClassDefFoundError problem case 1:

- Review the java.lang.NoClassDefFoundError error and identify the missing Java class
- Verify and locate the missing Java class from your compile / build environment
- Determine if the missing Java class is from your application code, third part API or even the Java EE container itself. Verify where the missing JAR file(s) is / are expected to be found
- Once found, verify your runtime environment Java classpath for any typo or missing JAR file(s)
- If the problem is triggered from a Java EE application, perform the same above steps but verify the packaging of your EAR / WAR file for missing JAR and other library file dependencies such as MANIFEST

Java static initializer revisited

The Java programming language provides you with the capability to "statically" initialize variables or a block of code. This is achieved via the "static" variable identifier or the usage of a static {} block at the header of a Java class. Static initializers are guaranteed to be executed only once in the JVM life cycle

and are Thread safe by design which make their usage quite appealing for static data initialization such as internal object caches, loggers etc.

What is the problem? I will repeat again, static initializers are guaranteed to be executed only once in the JVM life cycle...This means that such code is executed at the class loading time and never executed again until you restart your JVM. Now what happens if the code executed at that time (@Class loading time) terminates with an unhandled Exception?

Welcome to the java.lang.NoClassDefFoundError problem case #2!

NoClassDefFoundError problem case 2 - static initializer failure

This type of problem is occurring following the failure of static initializer code combined with successive attempts to create a new instance of the affected (non-loaded) class.

Sample Java program

The following simple Java program is split as per below:

- The main Java program `NoClassDefFoundErrorSimulator`
- The affected Java class `ClassA`
- `ClassA` provides you with a ON/OFF switch allowing you the replicate the type of problem that you want to study

This program is simply attempting to create a new instance of `ClassA` 3 times (one after each other). It will demonstrate that an initial failure of either a static variable or static block initializer combined with successive attempt to create a new instance of the affected class triggers java.lang.NoClassDefFoundError.

```
#### NoClassDefFoundErrorSimulator.java
package org.ph.javaee.tools.jdk7.training2;

public class NoClassDefFoundErrorSimulator {

	/**
	 * @param args
	 */
	public static void main(String[] args) {
		System.out.println("java.lang.NoClassDefFoundError Simulator - Training
2");
		System.out.println("Author: Pierre-Hugues Charbonneau");
		System.out.println("http://javaeesupportpatterns.blogspot.com\n\n");

		try {
			// Create a new instance of ClassA (attempt #1)
			System.out.println("FIRST attempt to create a new instance of
ClassA...\n");
			ClassA classA = new ClassA();

		} catch (Throwable any) {
			any.printStackTrace();
		}

		try {
			// Create a new instance of ClassA (attempt #2)
			System.out.println("\nSECOND attempt to create a new instance
of ClassA...\n");
			ClassA classA = new ClassA();

		} catch (Throwable any) {
			any.printStackTrace();
		}

		try {
			// Create a new instance of ClassA (attempt #3)
			System.out.println("\nTHIRD attempt to create a new instance of
ClassA...\n");
			ClassA classA = new ClassA();

		} catch (Throwable any) {
			any.printStackTrace();
		}

		System.out.println("\n\ndone!");
	}
}
```

```
#### ClassA.java
package org.ph.javaee.tools.jdk7.training2;

/**
 * ClassA
 * @author Pierre-Hugues Charbonneau
 *
 */
public class ClassA {

	private final static String CLAZZ = ClassA.class.getName();
	// Problem replication switch ON/OFF
	private final static boolean REPLICATE_PROBLEM1 = true; // static variable
initializer
	private final static boolean REPLICATE_PROBLEM2 = false; // static block{}
initializer

	// Static variable executed at Class loading time
	private static String staticVariable = initStaticVariable();

	// Static initializer block executed at Class loading time
	static {

		// Static block code execution...
		if (REPLICATE_PROBLEM2) throw new
IllegalStateException("ClassA.static{}: Internal Error!");
	}

	public ClassA() {
		System.out.println("Creating a new instance of "+ClassA.class.getName()
+"...");
	}

	/**
	 *
	 * @return
	 */
	private static String initStaticVariable() {

		String stringData = "";

		if (REPLICATE_PROBLEM1) throw new
IllegalStateException("ClassA.initStaticVariable(): Internal Error!");

		return stringData;
	}
}
```

Problem reproduction

In order to replicate the problem, we will simply "voluntary" trigger a failure of the static initializer code. Please simply enable the problem type that you want to study e.g. either static variable or static block initializer failure:

```
// Problem replication switch ON (true) / OFF (false)
private final static boolean REPLICATE_PROBLEM1 = true; // static variable initializer
private final static boolean REPLICATE_PROBLEM2 = false; // static block{} initializer
```

Now, let's run the program with both switch at OFF (both boolean values at false)

Baseline (normal execution)

```
java.lang.NoClassDefFoundError Simulator - Training 2
Author: Pierre-Hugues Charbonneau
http://javaeesupportpatterns.blogspot.com


FIRST attempt to create a new instance of ClassA...

Creating a new instance of org.ph.javaee.tools.jdk7.training2.ClassA...

SECOND attempt to create a new instance of ClassA...

Creating a new instance of org.ph.javaee.tools.jdk7.training2.ClassA...

THIRD attempt to create a new instance of ClassA...

Creating a new instance of org.ph.javaee.tools.jdk7.training2.ClassA...

done!
```

For the initial run (baseline), the main program was able to create 3 instances of `ClassA` successfully with no problem.

Problem reproduction run (static variable initializer failure)

```
java.lang.NoClassDefFoundError Simulator - Training 2
Author: Pierre-Hugues Charbonneau
http://javaeesupportpatterns.blogspot.com


FIRST attempt to create a new instance of ClassA...

java.lang.ExceptionInInitializerError
        at
org.ph.javaee.tools.jdk7.training2.NoClassDefFoundErrorSimulator.main(NoClassDe
fFoundErrorSimulator.java:21)
Caused by: java.lang.IllegalStateException: ClassA.initStaticVariable():
Internal Error!
        at
org.ph.javaee.tools.jdk7.training2.ClassA.initStaticVariable(ClassA.java:37)
        at org.ph.javaee.tools.jdk7.training2.ClassA.<clinit>(ClassA.java:16)
        ... 1 more

SECOND attempt to create a new instance of ClassA...

java.lang.NoClassDefFoundError: Could not initialize class
org.ph.javaee.tools.jdk7.training2.ClassA
        at
org.ph.javaee.tools.jdk7.training2.NoClassDefFoundErrorSimulator.main(NoClassDe
fFoundErrorSimulator.java:30)

THIRD attempt to create a new instance of ClassA...

java.lang.NoClassDefFoundError: Could not initialize class
org.ph.javaee.tools.jdk7.training2.ClassA
        at
org.ph.javaee.tools.jdk7.training2.NoClassDefFoundErrorSimulator.main(NoClassDe
fFoundErrorSimulator.java:39)


done!
```

Problem reproduction run (static block initializer failure)

```
 java.lang.NoClassDefFoundError Simulator - Training 2
Author: Pierre-Hugues Charbonneau
http://javaeesupportpatterns.blogspot.com



FIRST attempt to create a new instance of ClassA...


java.lang.ExceptionInInitializerError
       at
org.ph.javaee.tools.jdk7.training2.NoClassDefFoundErrorSimulator.main(NoClassDe
fFoundErrorSimulator.java:21)
Caused by: java.lang.IllegalStateException: ClassA.static{}: Internal Error!
       at org.ph.javaee.tools.jdk7.training2.ClassA.<clinit>(ClassA.java:22)
       ... 1 more


SECOND attempt to create a new instance of ClassA...


java.lang.NoClassDefFoundError: Could not initialize class
org.ph.javaee.tools.jdk7.training2.ClassA
       at
org.ph.javaee.tools.jdk7.training2.NoClassDefFoundErrorSimulator.main(NoClassDe
fFoundErrorSimulator.java:30)


THIRD attempt to create a new instance of ClassA...


java.lang.NoClassDefFoundError: Could not initialize class
org.ph.javaee.tools.jdk7.training2.ClassA
       at
org.ph.javaee.tools.jdk7.training2.NoClassDefFoundErrorSimulator.main(NoClassDe
fFoundErrorSimulator.java:39)


done!
```

What happened? As you can see, the first attempt to create a new instance of ClassA did trigger a `java.lang.ExceptionInInitializerError`. This exception indicates the failure of our static initializer for our static variable & bloc which is exactly what we wanted to achieve.

The key point to understand at this point is that this failure did prevent the whole class loading of `ClassA`. As you can see, attempt #2 and attempt #3 both generated a

java.lang.NoClassDefFoundError, why? Well since the first attempt failed, class loading of `ClassA` was prevented. Successive attempts to create a new instance of `ClassA` within the current ClassLoader did generate java.lang.NoClassDefFoundError over and over since `ClassA` was not found within current ClassLoader.

As you can see, in this problem context, the NoClassDefFoundError is just a *symptom* or *consequence* of another problem. The original problem is the ExceptionInInitializerError triggered following the failure of the static initializer code. This clearly demonstrates the importance of proper error handling and logging when using Java static initializers.

Recommendations and resolution strategies

Now find below my recommendations and resolution strategies for NoClassDefFoundError problem case 2:

- Review the java.lang.NoClassDefFoundError error and identify the missing Java class
- Perform a code walkthrough of the affected class and determine if it contains static initializer code (variables & static block)
- Review your server and application logs and determine if any error (e.g. ExceptionInInitializerError) originates from the static initializer code
- Once confirmed, analyze the code further and determine the root cause of the initializer code failure. You may need to add some extra logging along with proper error handling to prevent and better handle future failures of your static initializer code going forward

Parent first Classloader

The following section will describe one common problem pattern when using the default class loader delegation model.

A simple Java program will again be provided in order to help you understand this problem pattern.

Default JVM Classloader delegation model

As we saw before, the default class loader delegation model is from bottom-up e.g. parent first. This means that the JVM is going up to the class loader chain in order to find and load each of your application Java classes. If the class is not found from the parent class loaders, the JVM will then attempt to load it from the current Thread context class loader; typically a child class loader.

NoClassDefFoundError problems can occur, for example, when you wrongly package your application (or third part API's) between the parent and child class loaders. Another example is code / JAR files injection by the container itself or third party API's deployed at a higher level in the class loader chain.

In the above scenarios:

- The JVM loads one part of the affected code to a parent class loader (SYSTEM or parent class loaders)
- The JVM loads the other parts of the affected code to a child class loader (Java EE container

or application defined class loader)

Now what happens when Java classes loaded from the parent attempt to load reference classes deployed only to the child classloader? NoClassDefFoundError!
Please remember that a parent class loader has no visibility or access to child class loaders. This means that any referencing code must be found either from the parent class loader chain (bottom-up) or at the current Thread context class loader level; otherwise java.lang.NoClassDefFoundError is thrown by the JVM.

This is exactly what the following Java program will demonstrate.

Sample Java program

The following simple Java program is split as per below:

- The main Java program `NoClassDefFoundErrorSimulator` is packaged in MainProgram.jar
- A logging utility class `JavaEETrainingUtil` is packaged in MainProgram.jar
- The Java class caller `CallerClassA` is packaged in caller.jar
- The referencing Java class `ReferencingClassA` is packaged in referencer.jar

These following tasks are performed:

- Create a child class loader (java.net.URLClassLoader)
- Assign the caller and referencing Java class jar files to the child class loader
- Change the current Thread context ClassLoader to the child ClassLoader
- Attempt to load and create a new instance of CallerClassA from the current Thread context class loader e.g. child
- Proper logging was added in order to help you understand the class loader tree and Thread context class loader state

It will demonstrate that a wrong packaging of the application code is leading to a NoClassDefFoundError as per the default class loader delegation model.

```
#### NoClassDefFoundErrorSimulator.java
package org.ph.javaee.training3;

import java.net.URL;
import java.net.URLClassLoader;

import org.ph.javaee.training.util.JavaEETrainingUtil;

public class NoClassDefFoundErrorSimulator {

	/**
	 * @param args
	 */
	public static void main(String[] args) {

		System.out.println("java.lang.NoClassDefFoundError Simulator - Training 3");
		System.out.println("Author: Pierre-Hugues Charbonneau");
		System.out.println("http://javaeesupportpatterns.blogspot.com");

		// Local variables
		String currentThreadName = Thread.currentThread().getName();
		String callerFullClassName = "org.ph.javaee.training3.CallerClassA";

		// Print current ClassLoader context & Thread
		System.out.println("\nCurrent Thread name: '"+currentThreadName+"'");
		System.out.println("Initial ClassLoader chain:
"+JavaEETrainingUtil.getCurrentClassLoaderDetail());

		try {
			// Location of the application code for our child ClassLoader
			URL[] webAppLibURL = new URL[] {new URL("file:caller.jar"),new
URL("file:referencer.jar")};

			// Child ClassLoader instance creation
			URLClassLoader childClassLoader = new URLClassLoader(webAppLibURL);
			/*** Application code execution... ***/
			// 1. Change the current Thread ClassLoader to the child ClassLoader
			Thread.currentThread().setContextClassLoader(childClassLoader);
			System.out.println(">> Thread '"+currentThreadName+"' Context ClassLoader now
changed to '"+childClassLoader+"'");
			System.out.println("\nNew ClassLoader chain:
"+JavaEETrainingUtil.getCurrentClassLoaderDetail());

			// 2. Load the caller Class within the child ClassLoader...
			System.out.println(">> Loading '"+callerFullClassName+"' to child ClassLoader
'"+childClassLoader+"'...");
			Class<?> callerClass = childClassLoader.loadClass(callerFullClassName);

			// 3. Create a new instance of CallerClassA
			Object callerClassInstance = callerClass.newInstance();

		} catch (Throwable any) {
			System.out.println("Throwable: "+any);
			any.printStackTrace();
		}

		System.out.println("\nSimulator completed!");
	}
}
```

```
#### CallerClassA.java
package org.ph.javaee.training3;

import org.ph.javaee.training3.ReferencingClassA;

/**
 * CallerClassA
 * @author Pierre-Hugues Charbonneau
 *
 */
public class CallerClassA {

	private final static Class<CallerClassA> CLAZZ = CallerClassA.class;

	static {
		System.out.println("Class loading of "+CLAZZ+" from ClassLoader
'"+CLAZZ.getClassLoader()+"' in progress...");
	}

	public CallerClassA() {
		System.out.println("Creating a new instance of
"+CallerClassA.class.getName()+"...");

		doSomething();
	}

	private void doSomething() {

		// Create a new instance of ReferencingClassA
		ReferencingClassA referencingClass = new ReferencingClassA();
	}
}
```

```
#### ReferencingClassA.java
package org.ph.javaee.training3;

/**
 * ReferencingClassA
 * @author Pierre-Hugues Charbonneau
 *
 */
public class ReferencingClassA {

	private final static Class<ReferencingClassA> CLAZZ = ReferencingClassA.class;

	static {
		System.out.println("Class loading of "+CLAZZ+" from ClassLoader
'"+CLAZZ.getClassLoader()+"' in progress...");
	}

	public ReferencingClassA() {
		System.out.println("Creating a new instance of
"+ReferencingClassA.class.getName()+"...");
	}

	public void doSomething() {
		//nothing to do...
	}
}
```

Problem reproduction

In order to replicate the problem, we will simply “voluntary” split the packaging of the application code (caller & referencing class) between the parent and child class loader.

For now, let’s run the program with the right JAR files deployment and class loader chain:

- The main program and utility class are deployed at the parent class loader (SYSTEM classpath)
- `CallerClassA` and `ReferencingClassA` and both deployed at the child class loader level

Baseline (normal execution)

```
<JDK_HOME>\bin>java -classpath MainProgram.jar
org.ph.javaee.training3.NoClassDefFoundErrorSimulator

java.lang.NoClassDefFoundError Simulator - Training 3
Author: Pierre-Hugues Charbonneau
http://javaeesupportpatterns.blogspot.com

Current Thread name: 'main'
Initial ClassLoader chain:
---------------------------------------------------------------
sun.misc.Launcher$ExtClassLoader@17c1e333
--- delegation ---
sun.misc.Launcher$AppClassLoader@214c4ac9 **Current Thread 'main' Context
ClassLoader**
---------------------------------------------------------------

>> Thread 'main' Context ClassLoader now changed to
'java.net.URLClassLoader@6a4d37e5'

New ClassLoader chain:
---------------------------------------------------------------
sun.misc.Launcher$ExtClassLoader@17c1e333
--- delegation ---
sun.misc.Launcher$AppClassLoader@214c4ac9
--- delegation ---
java.net.URLClassLoader@6a4d37e5 **Current Thread 'main' Context ClassLoader**
---------------------------------------------------------------

>> Loading 'org.ph.javaee.training3.CallerClassA' to child ClassLoader
'java.net.URLClassLoader@6a4d37e5'...
Class loading of class org.ph.javaee.training3.CallerClassA from ClassLoader
'java.net.URLClassLoader@6a4d37e5' in progress...
Creating a new instance of org.ph.javaee.training3.CallerClassA...
Class loading of class org.ph.javaee.training3.ReferencingClassA from
ClassLoader 'java.net.URLClassLoader@6a4d37e5' in progress...
Creating a new instance of org.ph.javaee.training3.ReferencingClassA...

Simulator completed!
```

For the initial run (baseline), the main program was able to create successfully a new instance of `CallerClassA` from the child class loader (java.net.URLClassLoader) along with its referencing class with no problem.

Now let's run the program with the wrong application packaging and class loader chain:

- The main program and utility class are deployed at the parent class loader (SYSTEM classpath)
- `CallerClassA` and `ReferencingClassA` and both deployed at the child class loader level
- `CallerClassA` (caller.jar) is also deployed at the parent class loader level

Problem reproduction run (static variable initializer failure)

```
<JDK_HOME>\bin>java -classpath MainProgram.jar;caller.jar
org.ph.javaee.training3.NoClassDefFoundErrorSimulator

java.lang.NoClassDefFoundError Simulator - Training 3
Author: Pierre-Hugues Charbonneau
http://javaeesupportpatterns.blogspot.com

Current Thread name: 'main'
Initial ClassLoader chain:
------------------------------------------------------------------
sun.misc.Launcher$ExtClassLoader@17c1e333
--- delegation ---
sun.misc.Launcher$AppClassLoader@214c4ac9 **Current Thread 'main' Context ClassLoader**
------------------------------------------------------------------

>> Thread 'main' Context ClassLoader now changed to 'java.net.URLClassLoader@6a4d37e5'

New ClassLoader chain:
------------------------------------------------------------------
sun.misc.Launcher$ExtClassLoader@17c1e333
--- delegation ---
sun.misc.Launcher$AppClassLoader@214c4ac9
--- delegation ---
java.net.URLClassLoader@6a4d37e5 **Current Thread 'main' Context ClassLoader**
------------------------------------------------------------------

>> Loading 'org.ph.javaee.training3.CallerClassA' to child ClassLoader
'java.net.URLClassLoader@6a4d37e5'...
Class loading of class org.ph.javaee.training3.CallerClassA from ClassLoader
'sun.misc.Launcher$AppClassLoader@214c4ac9' in progress...// Caller is loaded from the
parent class loader, why???
Creating a new instance of org.ph.javaee.training3.CallerClassA...
Throwable: java.lang.NoClassDefFoundError: org/ph/javaee/training3/ReferencingClassA
java.lang.NoClassDefFoundError: org/ph/javaee/training3/ReferencingClassA
        at org.ph.javaee.training3.CallerClassA.doSomething(CallerClassA.java:27)
        at org.ph.javaee.training3.CallerClassA.<init>(CallerClassA.java:21)
        at sun.reflect.NativeConstructorAccessorImpl.newInstance0(Native Method)
        at sun.reflect.NativeConstructorAccessorImpl.newInstance(Unknown Source)
        at sun.reflect.DelegatingConstructorAccessorImpl.newInstance(Unknown Source)
        at java.lang.reflect.Constructor.newInstance(Unknown Source)
        at java.lang.Class.newInstance0(Unknown Source)
        at java.lang.Class.newInstance(Unknown Source)
        at
org.ph.javaee.training3.NoClassDefFoundErrorSimulator.main(NoClassDefFoundErrorSimulator
.java:51)
Caused by: java.lang.ClassNotFoundException: org.ph.javaee.training3.ReferencingClassA
        at java.net.URLClassLoader$1.run(Unknown Source)
        at java.net.URLClassLoader$1.run(Unknown Source)
        at java.security.AccessController.doPrivileged(Native Method)
        at java.net.URLClassLoader.findClass(Unknown Source)
        at java.lang.ClassLoader.loadClass(Unknown Source)
        at sun.misc.Launcher$AppClassLoader.loadClass(Unknown Source)
        at java.lang.ClassLoader.loadClass(Unknown Source)
        ... 9 more

Simulator completed!
```

What happened?

- The main program and utility classes were loaded as expected from the parent class loader (`sun.misc.Launcher$AppClassLoader`)
- The Thread context class loader was changed to child class loader as expected which includes both caller and referencing jar files
- However, we can see that `CallerClassA` was actually loaded by the parent class loader (`sun.misc.Launcher$AppClassLoader`) instead of the child class loader
- Since `ReferencingClassA` was not deployed to the parent class loader, the class cannot be found from the current class loader chain since the parent class loader has no visibility on the child class loader, NoClassDefFoundError is thrown

The key point to understand at this point is why `CallerClassA` was loaded by the parent class loader. The answer is with the default class loader delegation model. Both child and parent class loaders contain the caller JAR files. However, the default delegation model is always parent first which is why it was loaded at that level. The problem is that the caller contains a class reference to ReferencingClassA which is only deployed to the child class loader; java.lang.NoClassDefFoundError condition is met.

As you can see, a packaging problem of your code or third part API can easily lead to this problem due to the default class loader delegation behaviour. It is very important that you review your class loader chain and determine if you are at risk of duplicate code or libraries across your parent and child class loaders.

Recommendations and resolution strategies

Now find below my recommendations and resolution strategies for this problem pattern:

- Review the java.lang.NoClassDefFoundError error and identify the Java class that the JVM is complaining about
- Review the packaging of the affected application(s), including your Java EE container and third part API's used. The goal is to identify duplicate or wrong deployments of the affected Java class at runtime (SYSTEM class path, EAR file, Java EE container itself etc.).
- Once identified, you will need to remove and / or move the affected library/libraries from the affected class loader (complexity of resolution will depend of the root cause).
- Enable JVM class verbose e.g. –verbose:class. This JVM debug flag is very useful to monitor the loading of the Java classes and libraries from the Java EE container your applications. It can really help you pinpoint duplicate Java class loading across various applications and class loaders at runtime

ABOUT THE AUTHOR

Pierre-Hugues Charbonneau (nickname P-H) is working for CGI Inc. Canada for the last 10 years as a senior IT consultant. His primary area of expertise is Java EE, middleware & JVM technologies. He is a specialist in production system troubleshooting, root cause analysis, middleware, JVM tuning, scalability and capacity improvement; including internal processes improvement for IT support teams. P-H is the principal author at Java EE Support Patterns.

ABOUT THE EDITOR

Ilias is a senior software engineer working in the telecom domain. He is an applications developer in a wide variety of applications/services, currently the technical lead in a in-house PCRF solution. Particularly interested in multi-tier architecture, middleware services and mobile development (contact). Ilias Tsagklis is co-founder and Executive Editor at Java Code Geeks.