

Content

- **OutOfMemoryError**

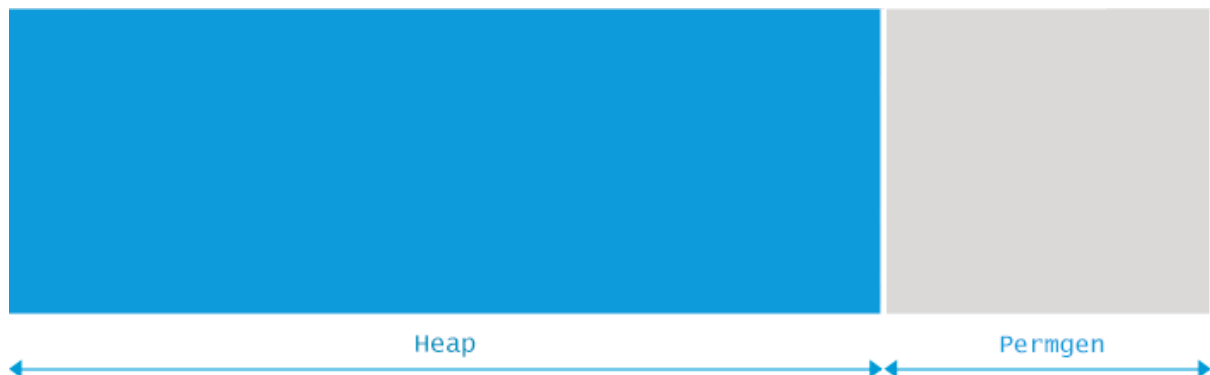
1. [java.lang.OutOfMemoryError: Java heap space](#)
2. [java.lang.OutOfMemoryError: PermGen space](#)
3. [java.lang.OutOfMemoryError: GC overhead limit exceeded](#)
4. [java.lang.OutOfMemoryError: Unable to create new native thread](#)
5. [java.lang.OutOfMemoryError: Requested array size exceeds VM limit](#)
6. [java.lang.OutOfMemoryError: Out of swap space?](#)
7. [java.lang.OutOfMemoryError: Metaspace](#)
8. [java.lang.OutOfMemoryError: Kill process or sacrifice child](#)

OutOfMemoryError

OutOfMemoryError occurs if your application involves a large amount of data processing, or is long lived - such as a server that responds to requests and is expected to run indefinitely. In technical terms, this error is thrown when the Java Virtual Machine cannot allocate an object because it is out of memory, and no more memory could be made available by the garbage collector. There can be 8 different types of OutOfMemoryError as mentioned below.

1. java.lang.OutOfMemoryError: Java heap space

Java applications are only allowed to use a limited amount of memory. This limit is specified during application startup. To make things more complex, Java memory is separated into two different regions. These regions are called **Heap Space** and **Permgen** (for Permanent Generation):



The size of those regions is set during the Java Virtual Machine (JVM) launch and can be customized by specifying JVM parameters `-Xmx` and `-XX:MaxPermSize`. If you do not explicitly set the sizes, platform-specific defaults will be used.

The *java.lang.OutOfMemoryError: Java heap space* error will be triggered when the application **attempts to add more data into the heap space area, but there is not enough room for it**.

Note that there might be plenty of physical memory available, but the *java.lang.OutOfMemoryError: Java heap space* error is thrown whenever the JVM reaches the heap size limit.

1.1 What's the cause?

The most common reason for the *java.lang.OutOfMemoryError: Java heap space* error is simple – you try to fit an XXL application into an S-sized Java heap space. That is – the application just requires more Java heap space than available to it to operate normally. Other causes for this OutOfMemoryError message are more complex and are caused by a programming error:

- **Spikes in usage/data volume.** The application was designed to handle a certain amount of users or a certain amount of data. When the number of users or the volume of data suddenly spikes and crosses that expected threshold, the operation which functioned normally before the spike ceases to operate and triggers the *java.lang.OutOfMemoryError: Java heap space* error.
- **Memory leaks.** A particular type of programming error will lead your application to constantly consume more memory. Every time the leaking functionality of the application is used it leaves some objects behind into the Java heap space. Over time the leaked objects consume all of the available Java heap space and trigger the already familiar *java.lang.OutOfMemoryError: Java heap space* error.

1.2 Example

Trivial example

The first example is truly simple – the following Java code tries to allocate an array of 2M integers. When you compile it and launch with 12MB of Java heap space (*java -Xmx12m OOM*), it fails with the *java.lang.OutOfMemoryError: Java heap space* message. With 13MB Java heap space the program runs just fine.

```
class OOM {
    static final int SIZE=2*1024*1024;
    public static void main(String[] a) {
        int[] i = new int[SIZE];
    }
}
```

Memory leak example

The second and a more realistic example is of a memory leak. In Java, when developers create and use new objects e.g. *new Integer(5)*, they don't have to allocate memory themselves – this is being taken care of by the Java Virtual Machine (JVM). During the life of the application the JVM periodically checks which objects in memory are still being used and which are not. Unused objects can be discarded and the memory reclaimed and reused again. This process is called Garbage Collection. The corresponding module in JVM taking care of the collection is called the Garbage Collector (GC).

Java's automatic memory management relies on GC to periodically look for unused objects and remove them. Simplifying a bit we can say that a memory leak in Java is a situation where some objects are no longer used by the application but Garbage Collection fails to recognize it. As a result these unused objects remain in Java heap space indefinitely. This pileup will eventually trigger the *java.lang.OutOfMemoryError: Java heap space* error.

It is fairly easy to construct a Java program that satisfies the definition of a memory leak:

```
class KeylessEntry {

    static class Key {
        Integer id;

        Key(Integer id) {
            this.id = id;
        }

        @Override
        public int hashCode() {
            return id.hashCode();
        }
    }

    public static void main(String[] args) {
        Map m = new HashMap();
        while (true)
            for (int i = 0; i < 10000; i++)
                if (!m.containsKey(new Key(i)))
                    m.put(new Key(i), "Number:" + i);
    }
}
```

When you execute the above code above you might expect it to run forever without any problems, assuming that the naive caching solution only expands the underlying Map to 10,000 elements, as beyond that all the keys will already be present in the HashMap. However, in reality the elements will keep being added as the Key class does not contain a proper *equals()* implementation next to its *hashCode()*.

As a result, over time, with the leaking code constantly used, the “cached” results end up consuming a lot of Java heap space. And when the leaked memory fills all of the available memory in the heap region and Garbage Collection is not able to clean it, the *java.lang.OutOfMemoryError:Java heap space* is thrown.

The solution would be easy – add the implementation for the *equals()* method similar to the one below and you will be good to go. But before you manage to find the cause, you will definitely have lose some precious brain cells.

```
@Override
public boolean equals(Object o) {
    boolean response = false;
    if (o instanceof Key) {
        response = (((Key)o).id).equals(this.id);
    }
    return response;
}
```

1.2 What's the solution?

In some cases, the amount of heap you have allocated to your JVM is just not enough to accommodate the needs of your applications running on that JVM. In that case, you should just allocate more heap.

In many cases however, providing more Java heap space will not solve the problem. For example, if your application contains a memory leak, adding more heap will just postpone the *java.lang.OutOfMemoryError: Java heap space* error. Additionally, increasing the amount of Java heap space also tends to increase the length of GC pauses affecting your application's throughput or latency.

If you wish to solve the underlying problem with the Java heap space instead of masking the symptoms, you need to figure out which part of your code is responsible for allocating the most memory. In other words, you need to answer these questions:

1. Which objects occupy large portions of heap
2. Where these objects are being allocated in source code

Here is a rough process outline that will help you answer the above questions:

- Get security clearance in order to perform a heap dump from your JVM. “Dumps” are basically snapshots of heap contents that you can analyze. These snapshot can thus contain confidential information, such as passwords, credit card numbers etc, so acquiring such a dump might not even be possible for security reasons.
- Get the dump at the right moment. Be prepared to get a few dumps, as when taken at a wrong time, heap dumps contain a significant amount of noise and can be practically useless. On the other hand, every heap dump “freezes” the JVM entirely, so don't take too many of them or your end users start facing performance issues.
- Find a machine that can load the dump. When your JVM-to-troubleshoot uses for example 8GB of heap, you need a machine with more than 8GB to be able to analyze heap contents. Fire up dump analysis software.

- Detect the paths to GC roots of the biggest consumers of heap.
- Figure out where in your source code the potentially hazardous large amount of objects is being allocated.

However, when your conclusion from memory analysis is that memory use is legal and there is nothing to change in the source code, you need to allow your JVM more Java heap space to run properly. In this case, alter your JVM launch configuration and add (or increase the value if present) the following:

```
-Xmx1024m
```

The above configuration would give the application 1024MB of Java heap space. You can use g or G for GB, m or M for MB, k or K for KB. For example all of the following are equivalent to saying that the maximum Java heap space is 1GB:

```
java -Xmx1073741824 com.mycompany.MyClass
java -Xmx1048576k com.mycompany.MyClass
java -Xmx1024m com.mycompany.MyClass
java -Xmx1g com.mycompany.MyClass
```

2. java.lang.OutOfMemoryError: PermGen space

The PermGen or permanent generation is a part of memory which holds meta-data describing user classes (classes that are not part of the Java language). Examples of such metadata are objects describing classes and methods and they are stored in the Permanent Generation. Applications with large code-base can quickly fill up this segment of the heap which will cause *java.lang.OutOfMemoryError: PermGen* no matter how high your -Xmx and how much memory you have on the machine.

Java applications are only allowed to use a limited amount of memory. The exact amount of memory your particular application can use is specified during application startup. If you do not set the sizes yourself, platform-specific defaults will be used.

The *java.lang.OutOfMemoryError: PermGen space* message indicates that the Permanent Generation's area in memory is exhausted.

2.1 What's the cause?

To understand the cause for the *java.lang.OutOfMemoryError: PermGen space*, we would need to understand what this specific memory area is used for.

For practical purposes, the permanent generation consists mostly of class declarations loaded and stored into PermGen. This includes the name and fields of the class, methods with the method bytecode, constant pool information, object arrays and type arrays associated with a class and Just In Time compiler optimizations.

From the above definition you can deduce that the PermGen size requirements depend both on the number of classes loaded as well as the size of such class declarations. Therefore we can say that **the main cause for the *java.lang.OutOfMemoryError: PermGen space* is that either too many classes or too big classes are loaded to the permanent generation.**

2.2 Example

Minimalistic example

As we described above, PermGen space usage is strongly correlated with the number of classes loaded into the JVM. The following code serves as the most straightforward example:

```
import javassist.ClassPool;
```

```

public class MicroGenerator {
    public static void main(String[] args) throws Exception {
        for (int i = 0; i < 100_000_000; i++) {
            generate("eu.plumbr.demo.Generated" + i);
        }
    }

    public static Class generate(String name) throws Exception {
        ClassPool pool = ClassPool.getDefault();
        return pool.makeClass(name).toClass();
    }
}

```

In this example the source code iterates over a loop and generates classes at runtime. Class generation complexity is being taken care of by the [javassist](#) library.

Launching the code above will keep generating new classes and loading their definitions into Permgen space until the space is fully utilized and the *java.lang.OutOfMemoryError: Permgen space* is thrown.

Redeploy-time example

When you redeploy an application, you would expect that Garbage Collection will get rid of the previous classloader referencing all the previously loaded classes and it gets replaced with a classloader loading new versions of the classes.

Unfortunately many 3rd party libraries and poor handling of resources such as threads, JDBC drivers or file system handles makes unloading the previously used classloader impossible. This in turn means that **during each redeploy all the previous versions of your classes will still reside in PermGen generating tens of megabytes of garbage during each redeploy.**

Let's imagine an example application that connects to a relational database using JDBC drivers. When the application is started, the initializing code loads the JDBC driver to connect to the database. Corresponding to the specification, the JDBC driver registers itself with *java.sql.DriverManager*. This registration includes storing a reference to an instance of the driver inside a static field of *DriverManager*.

Now, when the application is undeployed from the application server, *java.sql.DriverManager* will still hold that reference. We end up having a live reference to the driver class which in turn holds reference to the instance of *java.lang.ClassLoader* used to load the application. This in turn means that the Garbage Collection Algorithms are not able to reclaim the space.

And that instance of *java.lang.ClassLoader* still references all classes of the application, usually occupying **tens of megabytes in PermGen**. Which means that it would take just a handful of deploys to fill a typically sized PermGen and get the *java.lang.OutOfMemoryError: Permgen space* error message in your logs.

2.2 What's the solution?

1. Solving initialization-time OutOfMemoryError

When the OutOfMemoryError due to PermGen exhaustion is triggered during the application launch, the solution is simple. The application just needs more room to load all the classes to the PermGen area so we just need to increase its size. To do so, alter your application launch configuration and add (or increase if present) the `-XX:MaxPermSize` parameter similar to the following example:

```
java -XX:MaxPermSize=512m com.yourcompany.YourClass
```

The above configuration will tell the JVM that PermGen is allowed to grow up to 512MB before it can start complaining in the form of `OutOfMemoryError`.

2. Solving redeploy-time `OutOfMemoryError`

For this, proceed with heap dump analysis – take the heap dump after a redeploy with a command similar to this one:

```
jmap -dump:format=b,file=dump.hprof <process-id>
```

Then open the dump with heap dump analyzer. In the analyzer, you can look for duplicate classes, especially those loading your application classes. From there, you need to progress to all classloaders to find the currently active classloader.

For the inactive classloaders, you need to determine the reference blocking them from being Garbage Collected via harvesting the shortest path to GC root from the inactive classloaders. Equipped with this information you will have found the root cause. In case the root cause was in a 3rd party library, you can proceed to Google/StackOverflow to see if this is a known issue to get a patch/workaround. If this was your own code, you need to get rid of the offending reference.

3. Solving run-time `OutOfMemoryError`

First step in such case is to check whether the GC is allowed to unload classes from PermGen. The standard JVM is rather conservative in this regard – classes are born to live forever. So once loaded, classes stay in memory even if no code is using them anymore. This can become a problem when the application creates lots of classes dynamically and the generated classes are not needed for longer periods. In such a case, allowing the JVM to unload class definitions can be helpful. This can be achieved by adding just one configuration parameter to your startup scripts:

```
-XX:+CMSClassUnloadingEnabled
```

By default this is set to false and so to enable this you need to explicitly set the following option in Java options. If you enable `CMSClassUnloadingEnabled`, GC will sweep PermGen too and remove classes which are no longer used. Keep in mind that this option will work only when `UseConcMarkSweepGC` is also enabled using the below option. So when running [Parallel GC](#) or, God forbid, [Serial GC](#), make sure you have set your GC to [CMS](#) by specifying:

```
-XX:+UseConcMarkSweepGC
```

After making sure classes can be unloaded and the issue still persists, you should proceed with heap dump analysis – taking the heap dump with a command similar to following:

```
jmap -dump:file=dump.hprof,format=b <process-id>
```

Then opening the dump with heap dump analyzer and progressing to find the most expensive classloaders by the number of classes loaded. From such classloaders, you can proceed to extract the loaded classes and sort such classes by the instances to have the top list of suspects.

For each suspect, you then need to manually trace the root cause back to your application code that generates such classes.

3. java.lang.OutOfMemoryError: GC overhead limit exceeded

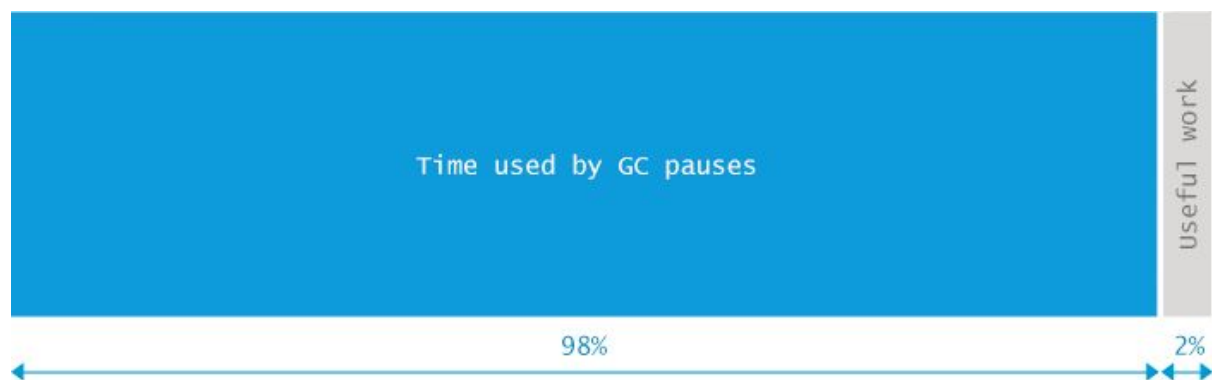
Java runtime environment contains a built-in Garbage Collection (GC) process. In many other programming languages, the developers need to manually allocate and free memory regions so that the freed memory can be reused.

Java applications on the other hand only need to allocate memory. Whenever a particular space in memory is no longer used, a separate process called Garbage Collection clears the memory for them.

The *java.lang.OutOfMemoryError: GC overhead limit exceeded* error is displayed when **your application has exhausted pretty much all the available memory and GC has repeatedly failed to clean it.**

3.1 What's the cause?

The *java.lang.OutOfMemoryError: GC overhead limit exceeded* error is the JVM's way of signalling that your application spends too much time doing garbage collection with too little result. By default the JVM is configured to throw this error if it spends more than **98% of the total time doing garbage collection and when after the garbage collection, only less than 2% of the heap is recovered.**



What would happen if this GC overhead limit would not exist? Note that the *java.lang.OutOfMemoryError: GC overhead limit exceeded* error is only thrown when 2% of the memory is freed after several GC cycles. This means that the small amount of heap the GC is able to clean will likely be quickly filled again, forcing the GC to restart the cleaning process again. This forms a vicious cycle where the CPU is 100% busy with GC and no actual work can be done. End users of the application face extreme slowdowns – operations which normally complete in milliseconds take minutes to finish.

3.2 Example

In the following example we create a “GC overhead limit exceeded” error by initializing a Map and adding key-value pairs into the map in an unterminated loop:

```
class Wrapper {  
    public static void main(String args[]) throws Exception {  
        Map map = System.getProperties();
```



```

Random r = new Random();
while (true) {
    map.put(r.nextInt(), "value");
} } }

```

As you might guess this cannot end well. And, indeed, when we launch the above program with:

```
java -Xmx100m -XX:+UseParallelGC Wrapper
```

we soon face the *java.lang.OutOfMemoryError: GC overhead limit exceeded* message. But the above example is tricky. When launched with different Java heap size or a different [GC algorithm](#), the Mac OS X 10.9.2 with Hotspot 1.7.0_45 will choose to die differently. For example, when the program is run with smaller Java heap size like this:

```
java -Xmx10m -XX:+UseParallelGC Wrapper
```

the application will die with a more common *java.lang.OutOfMemoryError: Java heap space* message that is thrown on Map resize. And when I run it with other garbage collection algorithms besides [ParallelGC](#), such as `-XX:+UseConcMarkSweepGC` or `-XX:+UseG1GC`, the error is caught by the default exception handler and is without stacktrace as the heap is exhausted to the extent where the stack trace cannot even be filled on Exception creation.

These variations are truly good examples that demonstrate that in resource-constrained situations you cannot predict the way your application is going to die so do not base your expectations on a specific sequence of actions to be completed.

3.2 What's the solution?

As a tongue-in-cheek solution, if you just wished to get rid of the “*java.lang.OutOfMemoryError: GC overhead limit exceeded*” message, adding the following to your startup scripts would achieve just that:

```
-XX:-UseGCOverheadLimit
```

It is **strongly suggested NOT to use this option** though – instead of fixing the problem you just postpone the inevitable: the application running out of memory and needing to be fixed. Specifying this option will just mask the original *java.lang.OutOfMemoryError: GC overhead limit exceeded* error with a more familiar message *java.lang.OutOfMemoryError: Java heap space*.

Also sometimes the GC overhead limit error is triggered because the amount of heap you have allocated to your JVM is just not enough to accommodate the needs of your applications running on that JVM. In that case, you should just allocate more heap. For the solution of this problem, refer to the **Section 1.3**.

4. java.lang.OutOfMemoryError: Unable to create new native thread

Java applications are multi-threaded by nature. What this means is that the programs written in Java can do several things (seemingly) at once. For example – even on machines with just one processor – while you drag content from one window to another, the movie played in the background does not stop just because you carry out several operations at once. Threads within the JVM need some room to carry out the work they are summoned to deal with. When there are more threads than there is room in memory we have built a foundation for a problem. The message *java.lang.OutOfMemoryError: Unable to create new native thread* means that the **Java application has hit the limit of how many Threads it can launch**.

4.1 What's the cause?

You have a chance to face the *java.lang.OutOfMemoryError: Unable to create new native thread* whenever the JVM asks for a new thread from the OS. Whenever the underlying OS cannot allocate a new native thread, this *OutOfMemoryError* will be thrown. This error can surface because of following two reasons:

1. There is no room in the memory to accommodate new threads.
2. The number of threads exceeds the Operating System limit.

But, in general, the situation causing *java.lang.OutOfMemoryError: Unable to create new native thread* goes through the following phases:

1. A new Java thread is requested by an application running inside the JVM
2. JVM native code proxies the request to create a new native thread to the OS
3. The OS tries to create a new native thread which requires memory to be allocated to the thread
4. The OS will refuse native memory allocation either because the 32-bit Java process size has depleted its memory address space – e.g. (2-4) GB process size limit has been hit – or the virtual memory of the OS has been fully depleted
5. The *java.lang.OutOfMemoryError: Unable to create new native thread* error is thrown.

4.2 Example

The following example creates and starts new threads in a loop. When running the code, operating system limits are reached fast and *java.lang.OutOfMemoryError: Unable to create new native thread* message is displayed.

```
while(true) {
    new Thread(new Runnable() {
        public void run() {
            try {
                Thread.sleep(10000000);
            } catch (InterruptedException e) { }
        }
    }).start();
}
```

The exact native thread limit is platform-dependent, for example tests on Windows, Linux and Mac OS X reveal that:

- 64-bit Mac OS X 10.9, Java 1.7.0_45 – JVM dies after #2031 threads have been created
- 64-bit Ubuntu Linux, Java 1.7.0_45 – JVM dies after #31893 threads have been created
- 64-bit Windows 7, Java 1.7.0_45 – due to a different thread model used by the OS, this error seems not to be thrown on this particular platform. On thread #250,000 the process was still alive, even though the swap file had grown to 10GB and the application was facing extreme performance issues.

So make sure you know your limits by invoking a small test and find out when the *java.lang.OutOfMemoryError: Unable to create new native thread* will be triggered

4.2 What's the solution?

There are 6 potential solutions to address this *java.lang.OutOfMemoryError: unable to create new native thread* error. Depending on what event is triggering this error, either one or a combination of the below-mentioned solutions can be applied to resolve the problem

1. Fix Thread Creation Rate

When you see *java.lang.OutOfMemoryError: unable to create new native thread*, you should diagnose whether the application has started to create more threads. You can use thread dump analyzer tool to see how many threads are created? What is the stack trace of those excessively created threads? Who is creating them? Once you know to these questions, it's easy to solve them as the solution lies in the code of your application itself.

2. Increase the Thread Limits Set at Operating System

The Operating System has limits for the number of threads that can be created. The limit can be found by issuing **ulimit -u** command. In certain servers, it is seen this value set to a low value such as 1024. It means totally only 1024 threads can be created in this machine. So if your application is creating more than 1024 threads, it's going to run into *java.lang.OutOfMemoryError: unable to create new native thread*. In such circumstances increase this limit.

3. Allocate More Memory to the Machine

If you don't see a high number of threads created and "ulimit -u" value is well ahead then it's indicative that your application has grown organically and needs more memory to create threads. In such circumstance, allocate more memory to the machine. It should solve the problem.

4. Reduce Heap Space

One very important point that even seasoned engineers forget is: threads are not created within the JVM heap. They are created outside the JVM heap. So if there is less room left in the RAM, after the JVM heap allocation, application will run into *java.lang.OutOfMemoryError: unable to create new native thread*. So let's consider this example:

Overall RAM size	6 GB
Heap size (i.e. -Xms and -Xmx)	5 GB
Perm Gen size (i.e. -XX:MaxPermSize and -XX:MaxPermSize)	512 MB

As per this configuration 5.5 GB (i.e. 5 GB heap + 512 MB Perm Gen) is used by the JVM Heap and it leave only 0.5GB (i.e. 6 GB – 5.5GB) space. Note in this 0.5 GB space - kernel processes, other user processes and threads has to run. It may not be sufficient, and most likely the application will start to experience *java.lang.OutOfMemoryError: unable to create new native thread*. To mitigate this problem, you can consider reducing the Heap Size from 5GB to 4GB (if your application can accommodate it without running into other memory bottlenecks).

5. Reduce Number of Processes

This solution is quite similar to 'Reduce Heap Space'. Let's looks into this scenario, where you are running multiple processes in a server which is constrained by memory. Say:

Overall RAM size	32 GB
Number of Java Processes in the server	5
Heap size of each Java process	6 GB

It means in total all of the java processes heap is occupying 30 GB (i.e. 5 processes X 6 GB) of memory. It leaves only 2 GB for kernel processes, other user processes and threads to run. It may not be sufficient, and most likely the application will start to experience *java.lang.OutOfMemoryError: unable to create new native thread*. In this circumstance, it's better to run only 4 java processes on one server. So that only 24 GB is occupied (4 processes X 6GB) and it leaves 8 GB (i.e. 32 GB – 24 GB) of memory. It might leave enough room

for threads and run other processes to run.

6. Reduce Thread Stack Size (-Xss)

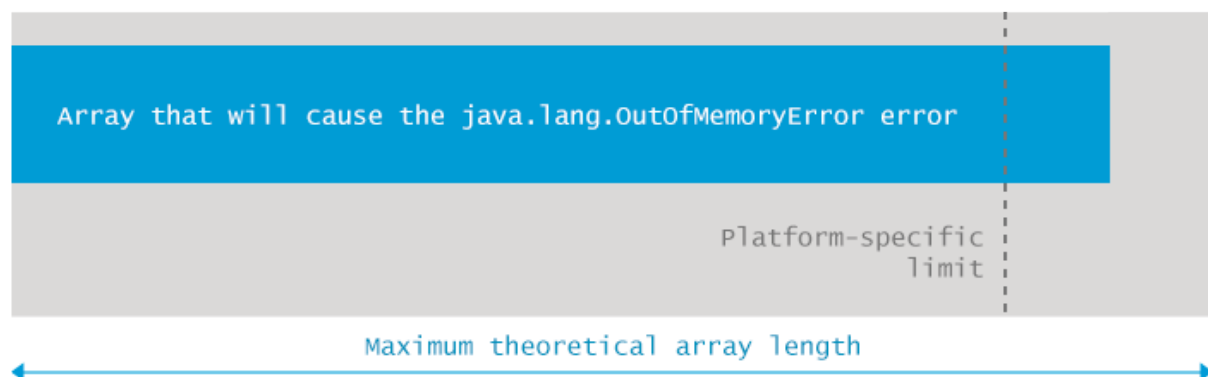
A thread occupies memory in RAM. So if each thread has high memory allocation then overall memory consumption will also go higher. The default value of a thread's memory size depends on the JVM provider. In some cases it's 1mb. So if your application has 500 threads then threads alone is going to occupy 500mb of space.

However, you can use the java system property `-Xss` to set the thread's memory size. Using this property you can throttle down the memory size. Example if you configure `-Xss256k`, your threads will only consume 125mb of space (i.e. 500 threads X 256k). So by lowering `-Xss` size also, you might be able to eliminate the error..

CAUTION: However if you configure `-Xss` to a very low value, you will start to experience `java.lang.StackOverflowError`. If you configure to even lower value, JVM will not even start.

5. java.lang.OutOfMemoryError: Requested array size exceeds VM limit

Java has got a limit on the maximum array size your program can allocate. The exact limit is platform-specific but is generally somewhere between 1 and 2.1 billion elements.



When you face the `java.lang.OutOfMemoryError: Requested array size exceeds VM limit`, this means that the application that crashes with the error is trying to allocate an array larger than the Java Virtual Machine can support.

5.1 What's the cause?

The error is thrown by the native code within the JVM. It happens before allocating memory for an array when the JVM performs a platform-specific check: whether the allocated data structure is addressable in this platform. This error is less common than you might initially think.

The reason you only seldom face this error is that Java arrays are indexed by `int`. The maximum positive `int` in Java is $2^{31} - 1 = 2,147,483,647$. And the platform-specific limits can be really close to this number – for example the 64bit MB Pro on Java 1.7 is able initialize arrays with up to 2,147,483,645 or `Integer.MAX_VALUE-2` elements.

Increasing the length of the array by one to `Integer.MAX_VALUE-1` results in the familiar `OutOfMemoryError`:

```
Exception in thread "main" java.lang.OutOfMemoryError: Requested array size exceeds VM limit
```

But the limit might not be that high – on 32-bit Linux with OpenJDK 6, you will hit the “*java.lang.OutOfMemoryError: Requested array size exceeds VM limit*” already when allocating an array with ~1.1 billion elements.

5.2 Example

When trying to recreate the *java.lang.OutOfMemoryError: Requested array size exceeds VM limit* error, let's look at the following code:

```
for (int i = 3; i >= 0; i--) {
    try {
        int[] arr = new int[Integer.MAX_VALUE-i];
        System.out.format("Successfully initialized an array with %,d
elements.\n", Integer.MAX_VALUE-i);
    } catch (Throwable t) {
        t.printStackTrace();
    }
}
```

The example iterates four times and initializes an array of long primitives on each turn. The size of the array this program is trying to initialize grows by one with every iteration and finally reaches `Integer.MAX_VALUE`. Now, when launching the code snippet on 64-bit Mac OS X with Hotspot 7, you should get the output similar to the following:

```
java.lang.OutOfMemoryError: Java heap space
    at eu.plumbr.demo.ArraySize.main(ArraySize.java:8)
java.lang.OutOfMemoryError: Java heap space
    at eu.plumbr.demo.ArraySize.main(ArraySize.java:8)
java.lang.OutOfMemoryError: Requested array size exceeds VM limit
    at eu.plumbr.demo.ArraySize.main(ArraySize.java:8)
java.lang.OutOfMemoryError: Requested array size exceeds VM limit
    at eu.plumbr.demo.ArraySize.main(ArraySize.java:8)
```

Note that before facing *java.lang.OutOfMemoryError: Requested array size exceeds VM limit* on the last two attempts, the allocations failed with a lot more familiar *java.lang.OutOfMemoryError: Java heap space* message. It happens because the $2^{31}-1$ int primitives you are trying to make room for require 8G of memory which is less than the defaults used by the JVM.

This example also demonstrates why the error is so rare – in order to see the VM limit on array size being hit, you need to allocate an array with the size right in between the platform limit and `Integer.MAX_INT`. When our example is run on 64bit Mac OS X with Hotspot 7, there are only two such array lengths: `Integer.MAX_INT-1` and `Integer.MAX_INT`.

5.2 What's the solution?

The *java.lang.OutOfMemoryError: Requested array size exceeds VM limit* can appear as a result of either of the following situations:

- Your arrays grow too big and end up having a size between the platform limit and the *Integer.MAX_INT*
- You deliberately try to allocate arrays larger than $2^{31}-1$ elements to experiment with the limits.

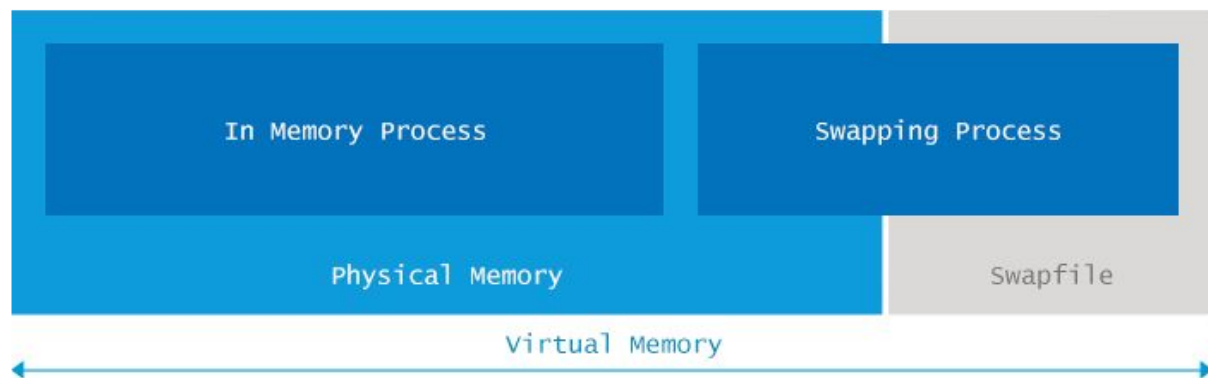
In the first case, check your code base to see whether you really need arrays that large. Maybe you could reduce the size of the arrays and be done with it. Or divide the array into smaller bulks and load the data you need to work with in batches fitting into your platform limit.

In the second case – remember that Java arrays are indexed by int. So you cannot go beyond $2^{31}-1$ elements in your arrays when using the standard data structures within the platform. In fact, in this case you are already blocked by the compiler announcing “*error: integer number too large*” during compilation.

But if you really work with truly large data sets, you need to rethink your options. You can load the data you need to work with in smaller batches and still use standard Java tools, or you might go beyond the standard utilities. One way to achieve this is to look into the *sun.misc.Unsafe* class. This allows you to allocate memory directly.

6. java.lang.OutOfMemoryError: Out of swap space?

Java applications are given limited amount of memory during the startup. This limit is specified via the -Xmx and other similar startup parameters. In situations where the total memory requested by the JVM is larger than the available physical memory, operating system starts swapping out the content from memory to hard drive.



The *java.lang.OutOfMemoryError: Out of swap space?* error indicates that the swap space is also exhausted and the new attempted allocation fails due to the lack of both physical memory and swap space.

6.1 What's the cause?

The *java.lang.OutOfMemoryError: Out of swap space?* is thrown by JVM when an allocation request for bytes from the native heap fails and the native heap is close to exhaustion. The message indicates the size (in bytes) of the allocation which failed and the reason for the memory request.

The problem occurs in situations where the Java processes have started swapping, which, recalling that Java is a garbage collected language is already not a good situation. Modern GC algorithms do a good job, but when faced with latency issues caused by swapping, the GC pauses tend to increase to levels not tolerable by most applications.

java.lang.OutOfMemoryError: Out of swap space? is often caused by operating system level issues, such as:

- The operating system is configured with insufficient swap space.
- Another process on the system is consuming all memory resources.

It is also possible that the application fails due to a native leak, for example, if application or library code continuously allocates memory but does not release it to the operating system.

6.2 What's the solution?

To overcome this issue, there are several possibilities. First and often the easiest workaround is to increase swap space. The means for this are platform specific, for example in Linux you can achieve this with the following example sequence of commands, which create and attach a new swapfile sized at 640MB:

```
swapoff -a
dd if=/dev/zero of=swapfile bs=1024 count=655360
mkswap swapfile
swapon swapfile
```

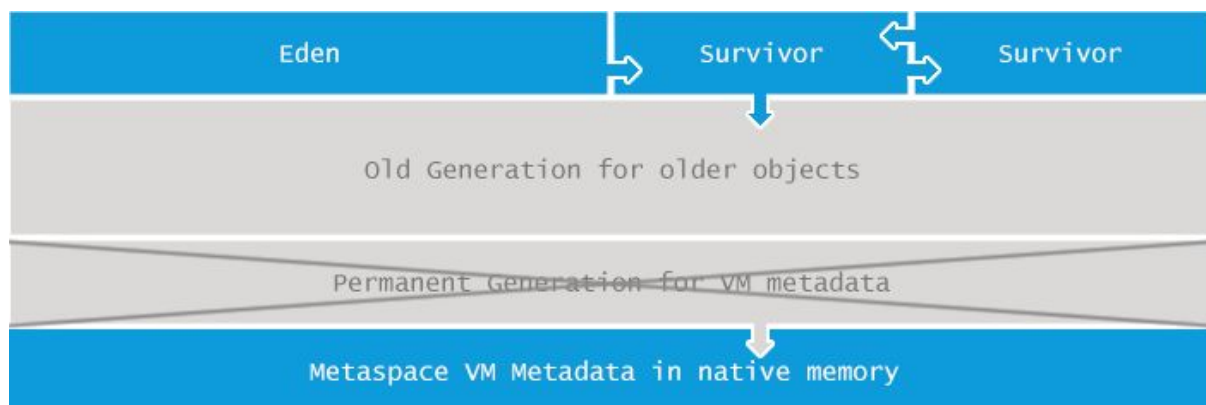
Now, you should recall that due to garbage collection sweeping the memory content, swapping is undesirable for Java processes in general. Running garbage collection algorithms on swapped allocations can increase the length of GC pauses by several orders of magnitude, so one should think twice before jumping to the easy solution bandwagon.

If the application is deployed next to a “noisy neighbor” with whom the JVM needs to compete for resources, the services should be isolated to separate (virtual) machines .

And in many cases, the only truly viable alternative is to either upgrade the machine to contain more memory or optimize the application to reduce its memory footprint. When you turn to the optimization path, a good way to start is by using memory dump analyzers to detect large allocations in memory.

7. java.lang.OutOfMemoryError: Metaspace

Java applications are allowed to use only a limited amount of memory. The exact amount of memory your particular application can use is specified during application startup. To make things more complex, Java memory is separated into different regions, as seen in the following figure:



The size of all those regions, including the metaspace area, can be specified during the JVM launch. If you do not determine the sizes yourself, platform-specific defaults will be used.

The *java.lang.OutOfMemoryError: Metaspace* message indicates that the Metaspace area in memory is exhausted.

7.1 What's the cause?

If you are not a newcomer to the Java landscape, you might be familiar with another concept in Java memory management called PermGen. Starting from Java 8, the memory model in Java was significantly changed. A new memory area called Metaspace was introduced and Permgen was removed. This change was made due to variety of reasons, including but not limited to:

- The required size of permgen was hard to predict. It resulted in either under-provisioning triggering `java.lang.OutOfMemoryError: Permgen space` errors or over-provisioning resulting in wasted resources.
- GC performance improvements, enabling concurrent class data de-allocation without GC pauses and specific iterators on metadata
- Support for further optimizations such as [G1](#) concurrent class unloading.

So if you were familiar with PermGen then all you need to know as background is that – whatever was in PermGen before Java 8 (name and fields of the class, methods of a class with the bytecode of the methods, constant pool, JIT optimizations etc) – is now located in Metaspace.

As you can see, Metaspace size requirements depend both upon the number of classes loaded as well as the size of such class declarations. So it is easy to see the **main cause for the `java.lang.OutOfMemoryError: Metaspace` is either too many classes or too big classes being loaded to the Metaspace.**

7.2 Example

Metaspace usage is strongly correlated with the number of classes loaded into the JVM. The following code serves as the most straightforward example:

```
public class Metaspace {
    static javassist.ClassPool cp = javassist.ClassPool.getDefault();

    public static void main(String[] args) throws Exception{
        for (int i = 0; ; i++) {
            Class c = cp.makeClass("eu.plumbr.demo.Generated" +
i).toClass();
        }
    }
}
```

In this example the source code iterates over a loop and generates classes at the runtime. All those generated class definitions end up consuming Metaspace. Class generation complexity is taken care of by the [javassist](#) library.

The code will keep generating new classes and loading their definitions to Metaspace until the space is fully utilized and the `java.lang.OutOfMemoryError: Metaspace` is thrown. When launched with `-XX:MaxMetaspaceSize=64m` then on Mac OS X my Java 1.8.0_05 dies at around 70,000 classes loaded.

7.3 What's the solution?

The first solution when facing the `OutOfMemoryError` due to Metaspace should be obvious. If the application exhausts the Metaspace area in the memory you should increase the size of Metaspace. Alter your application launch configuration and increase the following:

```
-XX:MaxMetaspaceSize=512m
```


The above configuration example tells the JVM that Metaspace is allowed to grow up to 512MB before it can start complaining in the form of *OutOfMemoryError*.

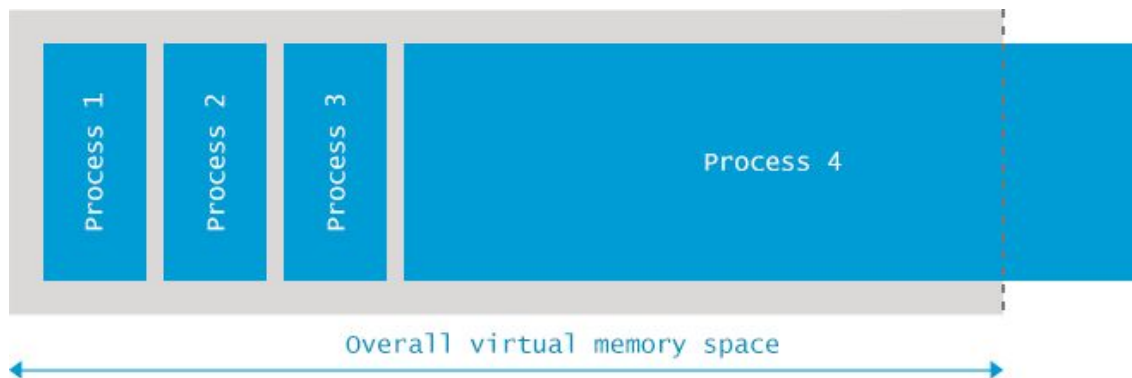
Another solution is even simpler at first sight. You can remove the limit on Metaspace size altogether by deleting this parameter. But pay attention to the fact that by doing so you can introduce heavy swapping and/or reach native allocation failures instead.

More often than not it can happen that by using the above recommended “quick fixes” you end up masking the symptoms by hiding the *java.lang.OutOfMemoryError: Metaspace* and not tackling the underlying problem. If your application leaks memory or just loads something unreasonable into Metaspace the above solution will not actually improve anything, it will just postpone the problem.

8. *java.lang.OutOfMemoryError*: Kill process or sacrifice child

In order to understand this error, we need to recoup the operating system basics. As you know, operating systems are built on the concept of processes. Those processes are shepherded by several kernel jobs, one of which, named “Out of memory killer” is of interest to us in this particular case.

This kernel job can annihilate your processes under extremely low memory conditions. When such a condition is detected, the Out of memory killer is activated and picks a process to kill. The target is picked using a set of heuristics scoring all processes and selecting the one with the worst score to kill. The *Out of memory: Kill process or sacrifice child* is thus different from other errors as it is neither triggered nor proxied by the JVM but is a safety net built into the operating system kernels.



The *Out of memory: kill process or sacrifice child* error is generated when the available virtual memory (including swap) is consumed to the extent where the overall operating system stability is put to risk. In such case the Out of memory killer picks the rogue process and kills it.

8.1 What's the cause?

By default, Linux kernels allow processes to request more memory than currently available in the system. This makes all the sense in the world, considering that most of the processes never actually use all of the memory they allocate. The easiest comparison to this approach would be the broadband operators. They sell all the consumers a 100Mbit download promise, far exceeding the actual bandwidth present in their network. The bet is

again on the fact that the users will not all simultaneously use their allocated download limit. Thus one 10Gbit link can successfully serve way more than the 100 users our simple math would permit.

A side effect of such an approach is visible in case some of your programs are on the path of depleting the system's memory. This can lead to extremely low memory conditions, where no pages can be allocated to process. You might have faced such situation, where not even a root account cannot kill the offending task. To prevent such situations, the killer activates, and identifies the rogue process to be the killed.

One common trigger for the activation is hidden in the operating system configuration. When you check the configuration in `/proc/sys/vm/overcommit_memory`, you have the first hint – the value specified here indicates whether all `malloc()` calls are allowed to succeed. Note that the path to the parameter in the `proc` file system varies depending on the system affected by the change.

Overcommitting configuration allows to allocate more and more memory for this rogue process which can eventually trigger the “*Out of memory killer*” to do exactly what it is meant to do.

8.2 Example

When you compile and launch the following Java code snippet on Linux:

```
package eu.plumbr.demo;
public class OOM {
public static void main(String[] args){
    java.util.List<int[]> l = new java.util.ArrayList();
    for (int i = 10000; i < 100000; i++) {
        try {
            l.add(new int[100_000_000]);
        } catch (Throwable t) {
            t.printStackTrace();
        }
    }
}
```

then you will face an error similar to the following in the system logs (`/var/log/kern.log` in our example):

```
Jun  4 07:41:59 plumbr kernel: [70667120.897649] Out of memory: Kill process 29957
(java) score 366 or sacrifice child
Jun  4 07:41:59 plumbr kernel: [70667120.897701] Killed process 29957 (java)
total-vm:2532680kB, anon-rss:1416508kB, file-rss:0kB
```

Note that you might need to tweak the swapfile and heap sizes, in our testcase we used a 2GB heap specified by `-Xmx2g` and had the following swap configuration:

```
swapoff -a
dd if=/dev/zero of=swapfile bs=1024 count=655360
mkswap swapfile
swapon swapfile
```

8.3 What's the solution?

There are several ways to handle such situation. The first and most straightforward way to overcome the issue is to migrate the system to an instance with more memory.

Other possibilities would involve fine-tuning the `OutOfMemory` killer, scaling the load horizontally across several small instances or reducing the memory requirements of the application.

One solution which we are not keen to recommend involves increasing swap space. When you recall that Java is a garbage collected language, then this solution already seems less lucrative. Modern GC algorithms are efficient when running in physical memory, but when dealing with swapped allocations the efficiency is hammered. Swapping can increase the length of GC pauses in several orders of magnitude, so you should think twice before jumping to this solution.