# Understanding OutOfMemoryError

In Java, all objects are stored in the heap. They are allocated by the `new` operator, and discarded when the JVM determines that no program thread can access them. Most of the time, this happens quietly, and programmers don't give it a second thought. And then, usually a day or so before a deadline, the program dies.

```
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
```

`OutOfMemoryError` is a frustrating exception. It usually means that you're doing something wrong: either holding onto objects too long, or trying to process too much data at a time. Sometimes, it indicates a problem that's out of your control, such as a third-party library that caches strings, or an application server that doesn't clean up after deploys. And sometimes, it has nothing to do with objects on the heap.

This article looks at the different causes of `OutOfMemoryError`, and what you can do about them. It focuses on the Sun JVM, which is the one that I use. However, much of the material applies to any JVM implementation. It is written based on literature available online and my own experience. I don't work on the JVM internals, so can't speak from a position of authority. But I have faced — and fixed — a lot of memory problems.

## Garbage Collection, an Overview

I've written a detailed account of the garbage collection process elsewhere. To summarize, the mark-sweep collector starts at "garbage collection roots" and walks the entire object graph, marking every object that it touches. Objects that aren't touched are garbage, and will be collected.

Java's approach to garbage collection means that the *only* way you can run out of memory is if you keep adding objects into that graph but don't remove them. Most often, this is because you add objects to a collection — usually a `Map` — that's referenced from a static variable. Or, less often, a collection held by a `ThreadLocal` or the `Runnable` of a long-lived thread.

This is very different from memory leaks in C and C++ programs. In those languages, leaks happen when a method calls `malloc()` or `new` and then returns without calling the corresponding `free()` or `delete`. These are true leaks: without using tools to match each allocation with its corresponding deallocation, you'll never be able to recover the memory.

In Java, "leaked" memory is merely misplaced. As far as the JVM is concerned, it's all accounted for. The problem is that you, the programmer, don't know where. Fortunately, there are ways to find it.

Before diving into those techniques, there's one last thing to know about the garbage collector: it will make a best effort to free memory *before* the JVM throws `OutOfMemoryError`. Which means that calling `System.gc()` *will not* solve your problem. You have to find the leak and plug it yourself.

## Setting the Heap Size

As pedants are fond of pointing out, the Java Language Specification doesn't say anything about garbage collection: you could implement a JVM that never frees memory (not that it would be terribly useful). The Java Virtual Machine Specification notes that the heap is managed by a garbage collector, but explicitly leaves the implementation details open. The only guarantee regarding garbage collection is the one I stated above: the collector (if it exists) will run before the JVM throws `OutOfMemoryError`.

In practice, the Sun JVM uses a fixed-size heap that's allowed to grow as needed between minimum and maximum bounds. If you don't specify these bounds, they default to a minimum of 2Mb and a maximum of 64Mb for the "client" JVM; the "server" JVM uses defaults based on available memory. While 64Mb must have seemed enormous when it became the default in 2000 (earlier, the default was 16Mb), modern applications can easily exhaust it.

> As far as I can tell, the JVM is unique among modern programming environments in that it has a fixed-size heap. I've never figured out the reason for this, but suspect that it was meant to ensure a contiguous heap for JVMs running within a browser. The only other place that I've seen explicit memory sizing was in the Macintosh (up through at least System 7), which ran multiple applications without hardware memory management.

Which means that you usually have to size the heap explicitly, using the `-Xms` and `-Xmx` command-line parameters:

```
java -Xms256m -Xmx512m MyClass
```

There are many rules-of-thumb for setting the minimum and maximum heap sizes. Obviously, the maximum has to be high enough to hold all of the objects that the program needs. However, setting it "just large enough" is not a good idea, because that increases the workload of the garbage collector. Instead, you should plan to keep 20–25% of the heap empty for a long-running application (although your particular application may need different settings; GC tuning is an art that's beyond the scope of this article).

Surprisingly, the minimum heap size is often more important than the maximum. The garbage collector will try to pre... the *current* heap size, whatever it is, rather than growing the heap. This can lead to a situation where a program crea... and discards a lot of objects, but never requires memory greater than the initial (minimum) heap size. The heap will ...y at th... size, but the garbage collector will be running constantly to keep it there. In a production environment, I think it ...akes ...nse to set the minimum and maximum bounds to the same value.



You may wonder why I bother with limiting the maximum size: after all, the operating system won't allocate physica... unless it's actually used. Part of the answer is that the virtual address space has to hold more than just the Java heap. If you're running on a 32-bit system, a large maximum heap size may limit the number of JARs on the classpath, or the number of threads you can create.

Another reason to limit the maximum heap size is that it helps you to discover any memory leaks in your code. Development environments tend not to stress applications. If you use a huge maximum heap size during development, you may never realize that you have a memory leak until you hit production.

## Watching the Garbage Collector at Work

All JVMs provide the `-verbose:gc` option, which tells the garbage collector to write a log message to the console whenever it runs:

```
java -verbose:gc com.kdgregory.example.memory.SimpleAllocator
[GC 1201K->1127K(1984K), 0.0020460 secs]
[Full GC 1127K->103K(1984K), 0.0196060 secs]
[GC 1127K->1127K(1984K), 0.0006680 secs]
[Full GC 1127K->103K(1984K), 0.0180800 secs]
[GC 1127K->1127K(1984K), 0.0001970 secs]
...
```

The Sun JVMs provide two additional options that show you the breakdown by generation, as well as the time that collection started:

```
java -XX:+PrintGCDetails -XX:+PrintGCTimeStamps com.kdgregory.example.memory.SimpleAll
0.095: [GC 0.095: [DefNew: 177K->64K(576K), 0.0020030 secs]0.097: [Tenured: 1063K->103
0.117: [GC 0.118: [DefNew: 0K->0K(576K), 0.0007670 secs]0.119: [Tenured: 1127K->103K(1
0.164: [GC 0.164: [DefNew: 0K->0K(576K), 0.0001990 secs]0.164: [Tenured: 1127K->103K(1
0.183: [GC 0.184: [DefNew: 0K->0K(576K), 0.0003400 secs]0.184: [Tenured: 1127K->103K(1
```

So what does this tell us? Well, first off that collections are happening very frequently. The first field in each line is a count of seconds since the JVM started, and we're seeing collections every few hundredths of a second. In fact, adding the start time of a collection to its execution time (shown at the end of the line), it appears that the collector is running constantly.

In a real application, this would be a problem, as the collector is taking CPU cycles away from the program. As I noted above, this might mean a too-small initial heap size, and the log confirms that is the case: every time the heap reaches 1.1 Mb, it gets collected. If you see this happening, increase the `-Xms` value before making changes to your application.

There's something else that's interesting about this log: other than the first collection, no objects are being stored in the young generation ("DefNew"). This indicates that the program is allocating large arrays and nothing else — which no real-world programs should do. If I saw this in a real application, my first question would be "what are these arrays used for?"

## Heap Dumps

A heap dump shows you the objects that the application is using. At its most basic, it's just a count of instances and bytes by class. You can also get dumps that show the code that allocated the memory, and compare historical counts to live counts. However, the more information that you collect, the more overhead you add to the running JVM, so some of these techniques are only applicable in a development environment.

### How to get a heap dump

The `-XX:+HeapDumpOnOutOfMemoryError` command-line argument is the simplest way to generate a heap dump. As its name implies, it will generate the dump only if the program runs out of memory, which makes it appropriate for production use. However, because it's a post-mortem dump, the only information it can provide is a histogram of objects. Furthermore, it creates a binary file, and you must use the jhat tool to examine that file (this tool is part of the JDK 1.6 distribution, but will read files produced by JDK 1.5 JVMs).

The jmap command (available since 1.5) lets you produce a heap dump from a running JVM: either a dump file for *jhat*, or a simple text histogram. A histogram is a good first line of analysis, particularly when you run it several times over an extended period, or when you compare live object counts to historical allocation counts.

At the far end of the scale, both in terms of information and overhead, are profilers. Profilers use the JVM's debugging interface to collect detailed information about object allocations, including line of code and the call stack. This can be incredibly useful: rather than just knowing that you've allocated a gigabyte of arrays, you can see that you've allocated 950 MB in one place, and can ignore the others. Of course, that information comes at a cost, both in terms of CPU consumption and memory to store the raw data. You won't be allowed to run a profiler in a production environment.

## Heap dump analysis: live objects

The definition of a Java memory leak is that you allocate objects and don't clear all references to them, meaning that the garbage collector can't reclaim them. A heap histogram is an easy way to start looking for such leaks: it shows not only the objects that are in your heap, but also the amount of memory they consume. The main drawback to a simple histogram is that all objects of the same class are grouped together, so you have to do some detective work to figure out where allocated.

Invoking *jmap* with the `-histo` option gives you a histogram that shows counts and memory consumption of all objects created since the program started, including those that have already been collected. Invoking it with `-histo:live` shows the counts for objects that are still on the heap, *whether or not they are eligible for collection*.

A histogram that shows all objects, even those that have been collected, is also useful to find "hot" objects: those that are frequently allocated and discarded. If you create temporary objects in an $O(N^2)$ algorithm, it will be immediately apparent from the histogram.

Which means that, to get accurate counts, you need to force the garbage collector to run before invoking *jmap*. If you're running the application locally, the easiest way to do this is with jconsole: at the top of the "Memory" tab, there's a button labeled "Perform GC." If you're running in a server environment, and have JMX beans exposed, the `MemoryMXBean` ("Memory" in the "java.lang" group) has a `gc()` operation. If neither of these options are available to you, you can always wait for a normal garbage collection. However, if you have a bad leak, the first major collection is likely to be an immediate precursor to `OutOfMemoryError`.

There are two ways to use a *jmap*-generated histograms. The most useful technique, particularly with long-running server applications, is to invoke the "live" option several times over an extended time period, and investigate those objects whose counts keep increasing. However, depending on the server load it might take an hour or more to get good information.

A quicker approach is to compare the live objects to the total. Those objects where the live count is a significant fraction of the total are possible leaks. Here's an example that shows the top dozen entries (out of nearly 2,500) for a repository manager that's been serving 100+ users for several weeks. This program does not, as far as I know, have a memory leak, but its normal operation results in heap dumps similar to programs that do.

```
~, 510> jmap -histo 7626 | more

 num     #instances         #bytes  class name
----------------------------------------------
   1:        339186       63440816  [C
   2:         84847       18748496  [I
   3:         69678       15370640  [Ljava.util.HashMap$Entry;
   4:        381901       15276040  java.lang.String
   5:         30508       13137904  [B
   6:        182713       10231928  java.lang.ThreadLocal$ThreadLocalMap$Entry
   7:         63450        8789976  <constMethodKlass>
   8:        181133        8694384  java.lang.ref.WeakReference
   9:         43675        7651848  [Ljava.lang.Object;
  10:         63450        7621520  <methodKlass>
  11:          6729        7040104  <constantPoolKlass>
  12:        134146        6439008  java.util.HashMap$Entry

~, 511> jmap -histo:live 7626 | more

 num     #instances         #bytes  class name
----------------------------------------------
   1:        200381       35692400  [C
   2:         22804       12168040  [I
   3:         15673       10506504  [Ljava.util.HashMap$Entry;
   4:         17959        9848496  [B
   5:         63208        8766744  <constMethodKlass>
   6:        199878        7995120  java.lang.String
   7:         63208        7592480  <methodKlass>
   8:          6608        6920072  <constantPoolKlass>
   9:         93830        5254480  java.lang.ThreadLocal$ThreadLocalMap$Entry
  10:        107128        5142144  java.lang.ref.WeakReference
  11:         93462        5135952  <symbolKlass>
  12:          6608        4880592  <instanceKlassKlass>
```

When looking for a memory leak, start with the objects the consume the most memory. That sounds obvious, but sometimes they *aren't* the source of the leak. Still, it's the best place to start, and in this case, the most memory is consumed by `char[]` instances (although the total here is 60Mb, hardly a problem). What is worrying is that the "live" count is almost two thirds of the "ever allocated" count.

Heap dump output uses the "internal form" of the class name. For the most part the names are what you'd expect, with the exception of arrays and classes internal to the JVM. The latter start with "<", while the former start with "`[`". Primitive arrays follow the bracket with a single uppercase character (you'll find the complete list in the JavaDoc for Class.getName()). Arrays of objects follow this bracket with an uppercase "`L`", the component classname, and a semi-colon. Multi-dimensional arrays are indicated by multiple brackets.

A normal program allocates objects and then releases them; if it holds onto objects for a long time, that's a possible leak. But having said that, the amount of "churn" that a particular program experiences depends on what the program is doing. Character arrays are almost always associated with strings, and some programs simply have a lot of strings that they hold for the life of the program. JSP-based app-servers, for example, define string literals for each block of HTML in the JSP. This particular program does serve HTML, but its need to hold strings is not so clear-cut: it serves directory listings, not a lot of static text. If I was

running out of memory, I'd try to find out where those strings were being allocated and why they weren't discarded.

Another area for concern are the byte arrays ("`[B`"). There are a lot of classes in the JDK that use them (for example, `BufferedInputStream`), but it's rare to see them in application code. Usually they're used as buffers, but buffers should be short-lived. Here we see that more than half of the byte arrays ever allocated are still considered live objects. This is worrisome, and it highlights one of the problems with a simple histogram: all of the objects for a single class are grouped together. For application objects, this isn't necessarily a problem, because they're typically allocated in one section of the program. But byte arrays are allocated everywhere, and most of those allocations are hidden within a library. Should we search for code that calls `new byte[]`, or code that calls `new ByteArrayOutputStream()`?

## Heap dump analysis: associating cause and effect

To find the ultimate cause of your leak, counts of bytes by class may not be enough. Instead, you have to associate the objects that are being leaked with the objects that your program is allocating. One way to do that is to look more closely at instance counts, in order to associate objects that are allocated together. Here's an (anonymized) excerpt of a heap dump from a program that's suffering memory problems:

```
 num     #instances         #bytes  class name
----------------------------------------------
   1:       1362278      140032936  [Ljava.lang.Object;
   2:         12624      135469922  [B
  ...
   5:        352166       45077248  com.example.ItemDetails
  ...
   9:       1360742       21771872  java.util.ArrayList
  ...
  41:          6254         200128  java.net.DatagramPacket
```

If you just looked at the top lines of this heap dump, you might start a fruitless search for code that allocated `Object[]` and `byte[]`. The actual culprits were `ItemDetails` and `DatagramPacket`: the former allocated multiple `ArrayList` instances, each of which in turn allocated an `Object[]`, while the latter uses `byte[]` to hold data retrieved from the network.

The first problem, allocating too-large arrays, wasn't actually a leak. The default `ArrayList` constructor allocates an array with ten elements, and the program was only using one or two; that's 62 bytes wasted per instance on a 64-bit JVM. A smarter design for the class would use a `List` only when one was needed, saving an additional 48 bytes per instance. Changes like that take effort, however, and adding memory is often cheaper.

The datagram leak was more troubling (as well as harder to fix): it indicated that received data was not being processed quickly enough.

In order to trace such chains of cause and effect, you need to know how your application uses objects. Not many programs allocate an `Object[]`: if they use arrays at all, they'll normally use a typed array. `ArrayList`, on the other hand, uses object arrays internally. But knowing that the memory is consumed by `ArrayList` instances isn't enough. You need to move one step up the chain, to find the objects that hold those lists.

One way to do this is to look for correlated instance counts. In the example above, the relationship of `byte[]` to `DatagramPacket` is obvious: one is almost exactly double the other. The relationship between `ArrayList` and `ItemDetails`, however, isn't immediately obvious (in fact, each `ItemDetails` instance held several lists).

The trick in this case is to focus on any other class that has high instance counts. We have a million `ArrayList` instances; either they're spread out between different classes or concentrated in a few. Regardless, a million references is hard to hide. Even if there were a dozen classes that held an `ArrayList`, those classes would still have 100,000 instances apiece.

Tracing such chains from a histogram is a lot of work. Fortunately, *jmap* isn't limited to histograms, it will also produce browsable heap dumps.

## Heap dump analysis: following reference chains

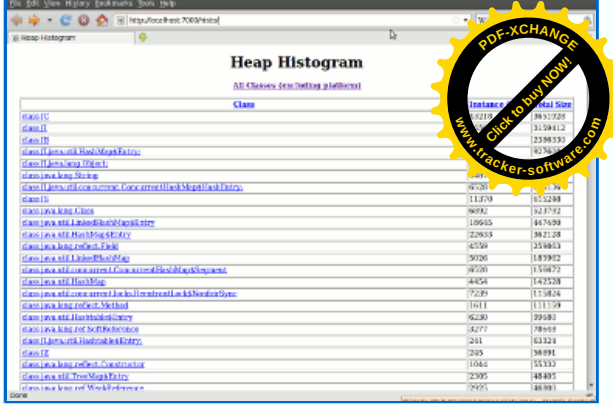
Browsing a heap dump is a two-step process: first, you invoke *jmap* with the `-dump` option, then you invoke *jhat* on the resulting file. If you need to go down this route, however, make sure that you have plenty of memory available: a dump file can easily be hundreds of megabytes, and *jhat* may need several gigabytes to process that file (I have occasionally resorted to Amazon EC2 quad-extra-large instances to be able to browse a heap dump).

```
tmp, 517> jmap -dump:live,file=heapdump.06180803 7626
Dumping heap to /home/kgregory/tmp/heapdump.06180803 ...
Heap dump file created

tmp, 518> jhat -J-Xmx8192m heapdump.06180803
Reading from heapdump.06180803...
Dump file created Sat Jun 18 08:04:22 EDT 2011
Snapshot read, resolving...
Resolving 335643 objects...
Chasing references, expect 67 dots.....................................................
Eliminating duplicate references.....................................................
Snapshot resolved.
Started HTTP server on port 7000
Server is ready.
```

The default URL gives you a listing of all classes loaded in the

system, which I rarely find useful. Instead, I start at `http://localhost:7000/histo/`, which is a histogram view that's sortable by instance count or total bytes (click on the image at right to see a full-size version).




Each of the class names in this histogram is a link, that takes you to a details page for that class. There you'll see the class' position in the class hierarchy, its member variables, and a link for every reference to an instance of the class. I don't find this page terribly useful either, and the list of references will spike your browser's memory usage.

For tracking down memory problems, the most useful page is the reference summary. This page has two tables: referrers (incoming) and referees (outgoing), both of which are pre-sorted by reference count. Clicking on a classname takes you to the reference summary for that class. Using the `ArrayList` example from the previous section, it's a matter of six clicks to follow the chain from `[Ljava.lang.Object;` to `com.example.ItemDetails`.

You can get to the reference summary page from the class details page. But I find it easier to construct the link directly: take the class link from the histogram page (eg: `http://localhost:7000/class/0xa5408348`) and replace “class” by “refsByType” (eg: `http://localhost:7000/refsByType/0xa5408228`).

## Heap dump analysis: allocation sites

Most of the time, knowing *what* objects are consuming memory is sufficient to figure out *why* they're being leaked. You can use *jhat* to find all references to those objects, and chances are good that you'll see the code that's holding objects too long. But sometimes that's not enough.

For example, if your code is leaking `Strings`, it may take you days to walk through all the string manipulation code. To solve problems like this, you need a heap dump that actually shows the places where memory is allocated. Be aware, however, that this type of profiling adds extreme overhead to your application, because the profiling agent must record every invocation of the `new` operator.

Most interactive profiles can produce this level of data, but I find the easiest way is to start the JVM with the built-in *hprof* agent:

```
java -Xrunhprof:heap=sites,depth=2 com.kdgregory.example.memory.Gobbler
```

*hprof* has a lot of options: not only can it profile memory usage in a variety of ways, it will also track CPU consumption. You'll find a link to a Sun technical article describing those options below. For this run, I specified a post-mortem dump of allocated objects along with the locations they were allocated, with a resolution of two stack frames. The output is written to the file `java.hprof.txt`, and the heap dump portion looks like this:

```
SITES BEGIN (ordered by live bytes) Tue Sep 29 10:43:34 2009
          percent          live          alloc'ed  stack class
 rank   self  accum     bytes objs     bytes  objs trace name
    1 99.77% 99.77%  66497808 2059  66497808  2059 300157 byte[]
    2  0.01% 99.78%      9192    1     27512    13 300158 java.lang.Object[]
    3  0.01% 99.80%      8520    1      8520     1 300085 byte[]
SITES END
```

This particular program doesn't allocate a lot of different object types, or allocate them in many different places. A normal dump is hundreds or thousands of lines long, showing each site where a particular object type was allocated. Fortunately, most problems appear in the first few lines of a dump. In this case, the thing that jumps out is the 64 Mb of live byte arrays, particularly since they average about 32k apiece.

Most programs don't need to hold onto that much data, so this is an indication that the program is not properly extracting and summarizing the data that it processes. You'll often see this happening with programs that read large strings and then hold onto substrings: a little-known implementation detail of `String.substring()` is that it shares a character array with the original string. If you read a file line-by-line, yet only use the first five characters of each line, you'll still hold the entire file in memory.

This dump also shows that the count of allocations for these arrays is equal to the count of live objects. This is a classic leak, and we can find the actual code by searching for the “trace” number:

```
TRACE 300157:
        com.kdgregory.example.memory.Gobbler.main(Gobbler.java:22)
```

Well, that was simple enough: when I go to that line in my code, I see that I'm storing the arrays in an `ArrayList` that never goes out of scope. Sometimes, however, the stack trace has no connection to code that you wrote:

```
TRACE 300085:
        java.util.zip.InflaterInputStream.<init>(InflaterInputStream.java:71)
        java.util.zip.ZipFile$2.<init>(ZipFile.java:348)
```

In this case, you need to increase the depth of the stack trace and rerun your application. There's a tradeoff here: as you

capture more stack frames you increase the overhead of profiling. The default, if you don't specify a `depth` value, is 4. I find that most problems in my code can be discovered with a depth of 2, although I have run with depths as high as 12 (on a machine that had a half gigabyte of physical memory; it was thrashing, and the profiling run took close to an hour, but I found the problem).

One other benefit of increasing the stack depth is that the report will be more granular: you might discover that you have leaking objects from two or three places, all of which use a common method.

#### Heap dump analysis: locality

Generational garbage collectors work because most objects are discarded shortly after they're allocated. You can use the same principle to find memory leaks: using a debugger, set a breakpoint at the allocation site and then walk through the code. In almost all cases, you'll see it added to a long-lived collection shortly after allocation.

## The Permanent Generation

In addition to the young and tenured generations of the application heap, the JVM also manages an area called the "permanent generation," where it stores objects such as classes and string literals. Normally, you don't see the garbage collector work on the permanent generation; most of the action happens in the application heap. But despite its name, objects in the permgen aren't always there permanently. Classes loaded by an app-server classloader, for example, are cleaned up once there are no longer any references to the classloader. And in an application server doing hot deploys, this may happen very frequently ... except when it doesn't:

```
Exception in thread "main" java.lang.OutOfMemoryError: PermGen space
```

Note the message: it's not referring to the application heap. You can run out of permgen space even when there's plenty of space in the application heap. Usually, it happens when you're redeploying an EAR or WAR, and simply indicates that the permgen isn't large enough to hold both the old classes and the new (the old classes are retained until all requests that used them have completed). This happens a lot when running a development app-server.

The first step to resolving permgen errors is to increase the permgen space, which you do with the `-XX:MaxPermSize` command-line argument. The default is 64M, but 256M is usually needed for an app-server or IDE.

```
java -XX:MaxPermSize=256m
```

Sometimes, however, it's not so simple. Leaks in the permgen happen for the same reason as leaks in the application heap: somewhere there's an object that's holding a reference that it shouldn't. In my experience, these tend to be references to actual `Class` objects, or objects in the `java.lang.reflect` package, rather than instances of a class. And because of the way that app-server classloaders are structured, the usual culprit is found in the server configuration.

For example, if you're using Tomcat, you have a directory for shared JARs: `shared/lib`. If you're running several web-apps on the same server, it makes sense to put some JARs in that directory, because their classes will be loaded only once, and you'll reduce the overall memory footprint of the server. However, what happens if one of those libraries has an object cache?

The answer is that, until the cache releases its reference to an application object, that application's classes won't get unloaded. The soluton in this case is to move the library into the WAR or EAR. But sometimes it's not so simple: the JDKs bean introspector caches `BeanInfo` instances and is loaded by the boot classloader. Any library using introspection will also cache those instances, and you may not know where the real responsibility falls.

Solving permgen problems is painful. The way to start is to track class loading and unloading with the `-XX:+TraceClassLoading` and `-XX:+TraceClassUnloading` command line options, Look for the classes that are loaded but not unloaded. If you add the `-XX:+TraceClassResolution` flag, you can see where one class accesses another, which may help you to find who is holding onto classes that aren't unloaded.

Here's a excerpt of output with all three options. The first line shows `MyClassLoader` being loaded from the classpath. Then, since it extends `URLClassLoader`, we see a "RESOLVE" message, followed in turn by another resolve message as `UrlClassLoader` references `Class`.

```
[Loaded com.kdgregory.example.memory.PermgenExhaustion$MyClassLoader from file:/home/k
RESOLVE com.kdgregory.example.memory.PermgenExhaustion$MyClassLoader java.net.URLClass
RESOLVE java.net.URLClassLoader java.lang.Class URLClassLoader.java:188
```

The information is all there, but it's usually faster just to move shared libraries into the WAR/EAR until the problem goes away.

## `OutOfMemoryError` When There's Still Heap Memory Available

As you saw with the permgen message, `OutOfMemoryError` gets thrown for reasons other than the application heap getting full. Here are a few cases:

#### Contiguous allocation

When I was describing the generational heap, I said that objects would be allocated in the young generation, and eventually

moved into the tenured generation. This isn't quite true: if your object is large enough, it will be created directly in the tenured generation. User-defined objects won't (shouldn't!) have anywhere near the number of members needed to trigger this behavior, but arrays will: with JDK 1.5, arrays larger than a half megabyte go directly into the tenured generation.

A half megabyte translates into an `Object[]` containing 131,072 elements on a 32-bit JVM. Big, but within the realm possibility for an enterprise scale application. Particularly one that uses a `HashMap` that needs to resize itself. And some applications need to work with much larger arrays.

The problem happens when there isn't a contiguous block of memory to hold the array, even after the garbage collector compacts the heap. This is rare, and the only time you'll run into it is when you're close to the limits of heap memory anyway. Increasing the maximum heap size is the best solution, but you can also try pre-sizing your large collections.

### Threads

The JavaDoc for `OutOfMemoryError` indicates that it is thrown when the garbage collector is unable to make memory available. This is only half true: the JVM also throws `OutOfMemoryError` when its internal code receives an `ENOMEM` error from the operating system. As Unix programmers know, there are a lot of places where you get `ENOMEM`, one of which is thread creation:

```
Exception in thread "main" java.lang.OutOfMemoryError: unable to create new native thr
```

On my 32-bit Linux box, running JDK 1.5, I can start 5,550 threads before getting this error. But there's still plenty of memory in the application heap (see this example). So what's happening?

Behind the scenes, threads are managed by the operating system, not the JVM, and thread creation can fail for any number of reasons. In my case, each thread takes about a half megabyte of virtual memory for its stack, and after 5,000 threads the 2G process memory space is used up (more on this below). Some operating systems might also impose a hard limit on the number of threads that can be created for a single process.

Again, there really isn't a solution to this problem other than changing your program. Most programs have no reason to create that many threads; they'll be spending all of their time waiting for the operating system to schedule them. But some server applications could reasonably create thousands of threads to service requests, knowing that most will be waiting for data; in this case, the solution might be NIO channels and selectors.

### Direct ByteBuffers

Java has allowed applications to access memory outside of the heap (in a limited way) since JDK 1.4, using bytebuffers. And while the `ByteBuffer` object itself is very small, the off-heap memory that it controlls may not be.

```
Exception in thread "main" java.lang.OutOfMemoryError: Direct buffer memory
```

There are several reasons for bytebuffer allocations to fail, which I've described in the linked article. Usually, however, it's because you've either exceeded the virtual memory space (only relevant on a 32-bit JVM) or placed a claim on all of physical memory and swap. But unless you're simply dealing with data that's too large for your machine, you run out of memory with direct buffers for the same reason that you run out of heap memory: you're holding a reference that you don't think you are. The heap analysis techniques described above can help you to find that leak.

### Commit charge exceeds physical memory

As I noted earlier, the JVM is unique in that you specify the minimum and maximum size of its heap on startup. This means that the JVM will change its demand for virtual memory while it's running. On a memory-constrained machine, you might be able to start multiple JVMs, even though their combined maximum heaps exceed available physical memory and swap. The result will be an `OutOfMemoryError` even though the total size of your live objects is less than the specified heap size.

This situation is no different from running multiple C++ programs that use all available memory. The JVM simply tricked you into thinking you could get away with it. The only solution is to buy more memory or run fewer programs. There's no way to make the JVM “fail fast” in this situation; at least on Linux, you can make a claim (via the `-Xms` and `-Xmx` parameters) on far more memory than can be supported.

## Looking Outside the Heap

That last situation brings up an important point: the Java heap is only part of the story. The JVM also uses memory for threads, its internal code and work space, shared libraries, direct buffers, and memory-mapped files. On a 32-bit JVM, these all have to fit into a 2GB virtual address space, which is a scarce resource (at least in server and back-end applications). On a 64-bit JVM, the virtual address space is effectively unlimited; physical memory is the scarce resource.

There aren't a lot of things that go wrong with virtual memory; the OS and JVM are pretty smart about managing it. Usually, the only reason that you'd look at a virtual memory map is to identify large chunks of memory used by direct buffers and memory-mapped files. But it's useful to know what the virtual memory map looks like, if only for personal education.

To dump the virtual memory map on Linux, you use pmap; for Windows, use VMMap (there may be other tools; I don't do much development on Windows). The following is an excerpt of the *pmap* output for a Tomcat server. The actual dump file is hundreds of lines long; this excerpt just shows the interesting bits.

```
08048000     60K r-x--  /usr/local/java/jdk-1.5/bin/java
```

```
08057000      8K rwx--  /usr/local/java/jdk-1.5/bin/java
081e5000   6268K rwx--    [ anon ]
889b0000    896K rwx--    [ anon ]
88a90000   4096K rwx--    [ anon ]
88e90000  10056K rwx--    [ anon ]
89862000  50488K rwx--    [ anon ]
8c9b0000   9216K rwx--    [ anon ]
8d2b0000  56320K rwx--    [ anon ]
...
afd70000    504K rwx--    [ anon ]
afdee000     12K -----    [ anon ]
afdf1000    504K rwx--    [ anon ]
afe6f000     12K -----    [ anon ]
afe72000    504K rwx--    [ anon ]
...
b0cba000     24K r-xs-  /usr/local/java/netbeans-5.5/enterprise3/apache-tomcat-5.5.17/
b0cc0000     64K r-xs-  /usr/local/java/netbeans-5.5/enterprise3/apache-tomcat-5.5.17/
b0cd0000    632K r-xs-  /usr/local/java/netbeans-5.5/enterprise3/apache-tomcat-5.5.17/
b0d6e000    164K r-xs-  /usr/local/java/netbeans-5.5/enterprise3/apache-tomcat-5.5.17/
b0d97000     88K r-xs-  /usr/local/java/netbeans-5.5/enterprise3/apache-tomcat-5.5.17/
...
b6ee3000   3520K r-x--  /usr/local/java/jdk-1.5/jre/lib/i386/client/libjvm.so
b7253000    120K rwx--  /usr/local/java/jdk-1.5/jre/lib/i386/client/libjvm.so
b7271000   4192K rwx--    [ anon ]
b7689000   1356K r-x--  /lib/tls/i686/cmov/libc-2.11.1.so
```

The dump output gives you four pieces of information for each segment in the memory map: its virtual address, its size, its permissions, and its source (for segments that are loaded from a file). Of these, the most interesting in terms of resource management are the permissions flags, which indicate whether the segment is read-only ("r-") or read-write ("rw").

I'll start with the read-write segments. Most of these have the name "`[ anon ]`", which is what Linux gives to everything that isn't associated with a file. There are also a few named read-write segments here, associated with shared libraries; I believe that these are the per-process relocation tables for those libraries.

Since most of the writable segments have the same name, it can take a bit of detective work to figure out what you're looking at. The Java heap will be four relatively large chunks that are allocated together (two for the young generation, one for the tenured generation, and one for the permgen); their total and relative sizes will depend on the GC and heap configuration. Each Java thread requires its own stack and scratch space; these are allocated in pairs, and you'll see the same pairs repeated through the map (in this map, the pairs are 504k/12k). Everything else corresponds to direct buffers (which may belong to your code, the JDK, or a third-party library) and runtime scratch spaces (which can belong to the JVM or third-party libraries).

The main issue with read-write segments is that they add to the process' "commit charge": the amount of physical memory and/or swap that is required for the process to run. The total commit charge for all of the programs running on a machine cannot exceed the physical RAM and swap of that machine. Modern operating systems deal with this gracefully: they return an `ENOMEM` when the program tries to allocate (older systems preferred to crash). This results in an `OutOfMemoryError` for your program, a situation that I described in the previous section.

I said that there isn't a lot that can go wrong with the virtual memory map, but the first time that I saw an "unable to create thread" error it was coming from an application server that had only a few dozen active threads and a reasonable application heap. It wasn't until I looked at the *pmap* output that I discovered the reason: the JVM (1.2 on Solaris) was allocating virtual address space for threads in strictly descending order based on the outstanding threads. In normal operation, as threads finished, their virtual address space would be reused. In this case, an occasional thread would stick around, and eventually that area of memory was fully used.

I never learned whether this was a bug with the JVM or with Solaris. We simply changed the way that we used threads. But it was a reminder to keep *pmap* and *strace* in my toolbox.

Read-only memory segments can be shared between processes, and give the operating system flexibility when managing physical memory: it can simply drop read-only pages, knowing that the can be reloaded from the source as needed. They're used for standard system libraries (*eg*, `libc`), the application code (`libjvm`), and memory-mapped files — including JARs from the classpath.

Memory-mapped JARs are an optimization: the JAR directory is at the end of the file, and contains offsets to the individual entries, so you can quickly access the entry with a "bulk get" operation. For classpath JARs this is a big win: you don't have to scan a `JarInputStream` (or use a caching scheme) every time you need to load a class. But there's a cost: memory-mapped files decrease the space available for the application heap and thread stacks.

With typical library JARs this isn't an issue, becasue they tend to be small (the JDK's `rt.jar` is large at 1.6 MB). Files that contain large resources (eg images) are another matter, although the JVM seems to be smart about only mapping portions of the file. But the your program can explicitly map any JAR or ZIP. As always, with great power comes great responsibility; give thought to why you want to memory-map the file. On a 32-bit JVM it's almost always better to unzip the JAR's contents into a scratch directory.

## Closing Comments

This section contains bits and pieces that didn't quite fit anywhere else. Most are tips from my personal experience in debugging memory problems.

### Don't be misled by virtual memory statistics

There's a common complaint that Java is a "memory hog," often evidenced by pointing at the "VIRT" column of *top* or the "Mem Usage" column on Windows Task Manager. As the last section should have made clear, there's a lot that goes into this number, and some of it is shared with other programs (the C library, for example). There is also a lot of "empty" space in the virtual memory map: if you invoke the JVM with `-Xms1000m`, the virtual size will start out over a gigabyte before your program starts allocating objects.

A much better measurement is the resident set size: the number of pages of physical memory that are actually used by your program, excluding shared pages. This is the "RES" column in *top* (I don't think it was available in Task Manger prior to Windows 7). However, resident set isn't a very good measure of your program's actual memory usage either. The operating system will leave physical pages in a process' memory space until it needs them elsewhere, and unless your system is heavily loaded that may take a long time.

Bottom line: always diagnose Java memory problems using tools that provide the detail needed to solve those problems. And don't jump to conclusions until you see an actual `OutOfMemoryError`.

### OutOfMemoryError is often thrown close to the source

Earlier I mentioned "locality": that memory leaks usually happen soon after the memory is allocated. A similar observation is that the `OutOfMemoryError` is often thrown close to the leak, and the stack trace (if it's available) can be a first step in diagnosis. The rationale behind this observation is that, to be noticeable, a leak must involve a lot of memory. Which means that, either due to the frequency that allocation code is called, or the volume of memory allocated on each pass, the leaky code will get a higher volume of failures. Not an ironclad rule, but it works well enough that I pay attention to the stack traces.

### Always suspect caches

I've mentioned caches several times in this article, and with good reason: most of the memory leaks that I've seen, in over a dozen years working with Java, are found in a class with "Cache" in its name. And I wrote a few of them. The trouble is that caches aren't easy to get right.

That said, there are a lot of good reasons to use a cache, and a few good reasons to write your own (although in most cases, an off-the-shelf cache like EHCache is a better choice). If you do choose to use a cache, make sure that you have answers to the following questions:

- *What objects go into the cache?*
  If you limit your cache to a single object type (or inheritance tree), then it's much easier to trace cause and effect versus a generic cache that holds many different types of objects.
- *How many objects should be in the cache at any point in time?*
  Again, if you know that `ProductCache` should only hold 1,000 entries, but see 10,000 instances of `Product` in the histogram, the link between cause and effect is easier to trace. More important, if you go in with a maximum size in mind, then you can easily calculate how much memory you'll need for the cached objects.
- *What is the eviction policy?*
  Every cache should have an explicit eviction policy that controls how long objects will remain in the cache. If you don't have a policy in place, then objects may remain in the cache long after the program needs them — using memory and adding load to the garbage collector (remember: the mark phase requires time proportional to *live* objects).
- *Will I hold other long-lived references to cached objects?*
  Caches work best in code that's executed frequently, doesn't take a lot of time to run, but needs objects that are expensive to retrieve from source. If you create an object and need a reference to it as long as your program runs, there's no need to put that object in a cache (although a pool might be a good idea, to control the number of instances).

### Pay attention to object lifetime

Objects generally fall into two classes: those that live as long as the program is running, and those that live only long enough to service a single request. Keeping the two separate and knowing which is which is the key to easy debugging: you only need to look at the objects you know are long-lived.

One way to do this is to explicitly initialize all of your long-lived objects when the program starts, whether or not they're needed right away. Another, better, approach is to use a dependency injection framework like Spring. Not only does it mean that all long-lived objects will be found in the bean configuration (never use classpath scanning!), but it's easy to find all of the places where those beans are used.

### Look out for instance variables misused as method parameters

In almost all cases, the objects allocated within a method should be discarded by the end of that method (excluding return values, of course). As long as you use local variables to hold all these objects, this rule is easy to follow. However, it's tempting to use instance variables to hold method data, particularly for methods that call a lot of other methods, and thereby avoiding the need for large parameter lists.

Doing this won't necessarily create a leak. Subsequent method calls should re-assign the variables, allowing the objects to be collected. But it raises your memory requirements unnecessarily (sometimes enormously) and makes debugging more difficult (because you have unexpected objects in memory). And from a general design perspective, when I've seen code like this it has always indicated that the method really wants to be its own class.

### J2EE: don't abuse the session

The session object exists to hold user-specific data between web requests; it's a way around the stateless nature of HTTP. Far

too often it ends up as a makeshift cache. For example, an eCommerce programmer might store a `Product` object in the session, rationalizing that the user will be browsing several pages for the same product.

This won't turn into a true leak, because the servlet container will time-out the user's session. But it needlessly increases the memory footprint of the application, which is just as bad (it doesn't matter what causes the `OutOfMemoryError` the application is just as dead). And it's hard to debug: as I noted above with mixed-content caches, there's no clear indication of where the objects are being held.

### Beware excessive garbage collection

While `OutOfMemoryError` is bad, a constantly running garbage collector is far worse: it takes CPU cycles that should be going to your application. Tuning the garbage collector is an art, and one that I haven't spent a lot of time doing. So I've included some references at the end of this article.

### 64-bit processors and cheap RAM won't make your problems go away

This article was originally written when 32-bit processors were mainstream, and so contains a lot of caveats about how pieces of the JVM have to share a 2 Gb virtual address space. Most of those caveats will soon be obsolete: when you can go to an office supply store and spend $500 (in early 2011) for a PC with 6 gig of RAM and effectively unlimited virtual address space, there doesn't seem to be a lot gained by fussing over every last byte.

But that doesn't mean that you shouldn't pay attention to your memory footprint. Large heaps and lots of live objects can turn into a “GC timebomb,” in which the garbage collector spends an inordinate amount of time doing its job. Even with a concurrent GC, your application may be blocked waiting for memory to be freed so that it can allocate more.

### Sometimes you just need more memory

As I said back at the beginning, the JVM is the only modern programming environment that I know of that forces you to set a maximum size for your data. Because of this, there are times when you think you have a leak but in fact you just have to increase the heap size. The first step in resolving a memory problem should always be increasing the available memory. If you have an actual leak, you'll get an `OutOfMemoryError` no matter how much memory you provide.

## For More Information

Sun's tuning whitepaper (JDK 1.6 edition) describes how the Sun JVM manages its heap, and how you can adjust the behavior of the garbage collector. There's also a FAQ on features of the various garbage collectors.

This document lists the Sun-only JVM configuration options. There are a number of useful diagnostic options that I haven't covered here.

This Sun technote describes the built-in “hprof” profiling tool.

The chapter on memory management from O'Reilly's System Performance Tuning, 2nd Edition gives a nice overview of how Unix manages process memory (this book is 10 years old, but the concepts haven't changed much in the last 30+ years, so it's still relevant).

My article on Java reference objects will help you manage memory so that you can avoid running out.

I've created several example programs that consume resources:

- Gobbler grabs as much memory as it can, and never lets go.
- SimpleAllocator grabs memory and then immediately releases it. Its lets you watch the GC in action.
- PermgenExhaustion creates classloader instances, and uses them to load a class. It can be difficult to follow, but will also give you an idea of how classloaders work.
- ThreadExhaustion fires up as many threads as it can. On both Linux and Windows XP, the maximum number of threads is limited by virtual memory. Other systems may impose lower limits.