# Java (JVM) Memory Types

Java has only two types of memory when it comes to JVM. Heap memory and Non-heap memory. All the other memory jargons you hear are logical part of either of these two.

## Heap Memory

Class instances and arrays are stored in heap memory. Heap memory is also called as shared memory. As this is the place where multiple threads will share the same data.

## Non-heap Memory

It comprises of ‘Method Area’ and other memory required for internal processing. So here the major player is ‘Method Area’.

### Method Area

As given in the last line, method area is part of non-heap memory. It stores per-class structures, code for methods and constructors. Per-class structure means runtime constants and [static fields](http://javapapers.com/core-java/explain-the-java-static-modifier/).

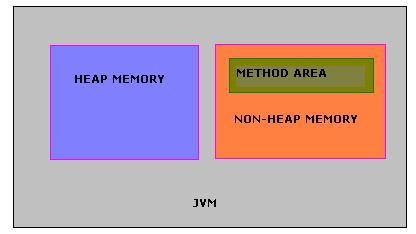
The above three (heap memory, non-heap memory and method area) are the main jargon when it comes to memory and JVM. There are some other technical jargon you might have heard and I will summarize them below.

## Memory Pool

Memory pools are created by JVM memory managers during runtime. Memory pool may belong to either heap or non-heap memory.

## Runtime Constant Pool

A run time constant pool is a per-class or per-interface run time representation of the constant\_pool table in a class file. Each runtime constant pool is allocated from the Java virtual machine’s method area.



## Java Stacks or Frames

Java stacks are created private to a thread. Every thread will have a program counter (PC) and a java stack. PC will use the java stack to store the intermediate values, dynamic linking, return values for methods and dispatch exceptions. This is used in the place of registers.

## Memory Generations

HotSpot VM’s garbage collector uses generational garbage collection. It separates the JVM’s memory into and they are called young generation and old generation.

### Young Generation

Young generation memory consists of two parts, Eden space and survivor space. Shortlived objects will be available in Eden space. Every object starts its life from Eden space. When GC happens, if an object is still alive and it will be moved to survivor space and other dereferenced objects will be removed.

### Old Generation – Tenured and PermGen

Old generation memory has two parts, tenured generation and permanent generation (PermGen). PermGen is a popular term. We used to error like PermGen space not sufficient.

GC moves live objects from survivor space to tenured generation. The permanent generation contains meta data of the virtual machine, class and method objects.

## Discussion:

Java specification doesn’t give hard and fast rules about the design of JVM with respect to memory. So it is completely left to the JVM implementers. The types of memory and which kind of variable / objects and where they will be stored is specific to the JVM implementation.

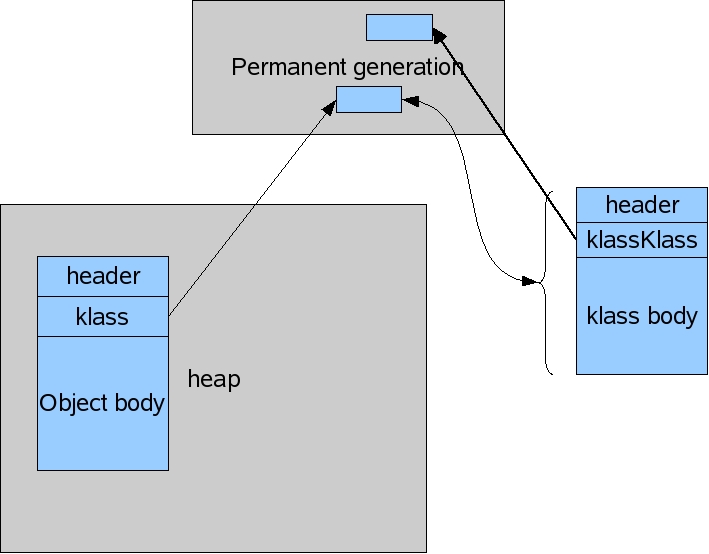
## Key Takeaways

* Local Variables are stored in Frames during runtime.
* Static Variables are stored in Method Area.
* Arrays are stored in heap memory.

**Presenting the Permanent Generation**

Have you ever wondered how the permanent generation fits into our generational system? Ever been curious about what's in the permanent generation. Are objects ever promoted into it? Ever promoted out? We'll you're not alone. Here are some of the answers.

Java objects are instantiations of Java classes. Our JVM has an internal representation of those Java objects and those internal representations are stored in the heap (in the young generation or the tenured generation). Our JVM also has an internal representation of the Java classes and those are stored in the permanent generation. That relationship is shown in the figure below.



The internal representation of a Java object and an internal representation of a Java class are very similar. From this point on let me just call them Java objects and Java classes and you'll understand that I'm referring to their internal representation. The Java objects and Java classes are similar to the extent that during a garbage collection both are viewed just as objects and are collected in exactly the same way. So why store the Java objects in a separate permanent generation? Why not just store the Java classes in the heap along with the Java objects?

Well, there is a philosophical reason and a technical reason. The philosophical reason is that the classes are part of our JVM implementation and we should not fill up the Java heap with our data structures. The application writer has a hard enough time understanding the amount of live data the application needs and we shouldn't confuse the issue with the JVM's needs.

The technical reason comes in parts. Firstly the origins of the permanent generation predate my joining the team so I had to do some code archaeology to get the story straight (thanks Steffen for the history lesson).

Originally there was no permanent generation. Objects and classes were just stored together.

Back in those days classes were mostly static. Custom class loaders were not widely used and so it was observed that not much class unloading occurred. As a performance optimization the permanent generation was created and classes were put into it. The performance improvement was significant back then. With the amount of class unloading that occur with some applications, it's not clear that it's always a win today.

It might be a nice simplification to not have a permanent generation, but the recent implementation of the parallel collector for the tenured generation (aka parallel old collector) has made a separate permanent generation again desirable. The issue with the parallel old collector has to do with the order in which objects and classes are moved. If you're interested, I describe this at the end.

So the Java classes are stored in the permanent generation. What all does that entail? Besides the basic fields of a Java class there are

* Methods of a class (including the bytecodes)
* Names of the classes (in the form of an object that points to a string also in the permanent generation)
* Constant pool information (data read from the class file, see chapter 4 of the JVM specification for all the details).
* Object arrays and type arrays associated with a class (e.g., an object array containing references to methods).
* Internal objects created by the JVM (java/lang/Object or java/lang/exception for instance)
* Information used for optimization by the compilers (JITs)

That's it for the most part. There are a few other bits of information that end up in the permanent generation but nothing of consequence in terms of size. All these are allocated in the permanent generation and stay in the permanent generation. So now you know.

This last part is really, really extra credit. During a collection the garbage collector needs to have a description of a Java object (i.e., how big is it and what does it contain). Say I have an object X and X has a class K. I get to X in the collection and I need K to tell me what X looks like. Where's K? Has it been moved already? With a permanent generation during a collection we move the permanent generation first so we know that all the K's are in their new location by the time we're looking at any X's.

How do the classes in the permanent generation get collected while the classes are moving? Classes also have classes that describe their content. To distinguish these classes from those classes we spell the former klasses. The classes of klasses we spell klassKlasses. Yes, conversations around the office can be confusing. Klasses are instantiation of klassKlasses so the klassKlass KZ of klass Z has already been allocated before Z can be allocated. Garbage collections in the permanent generation visit objects in allocation order and that allocation order is always maintained during the collection. That is, if A is allocated before B then A always comes before B in the generation. Therefore if a Z is being moved it's always the case that KZ has already been moved.

And why not use the same knowledge about allocation order to eliminate the permanent generations even in the parallel old collector case? The parallel old collector does maintain allocation order of objects, but objects are moved in parallel. When the collection gets to X, we no longer know if K has been moved. It might be in its new location (which is known) or it might be in its old location (which is also known) or part of it might have been moved (but not all of it). It is possible to keep track of where K is exactly, but it would complicate the collector and the extra work of keeping track of K might make it a performance loser. So we take advantage of the fact that classes are kept in the permanent generation by collecting the permanent generation before collecting the tenured generation. And the permanent generation is currently collected serially.

**Busting java.lang.String.intern() Myths**

**(Or: String.intern() is dangerous unless you know what you're doing!)**

***Update:*** *The subject discussed in this article was true up to Java 6. In Java 7, interned strings are no longer allocated in the permanent generation, but in the main Java Heap. I've decided to leave this article here for historical purposes, but keep in mind that the allocation of interned strings in the main heap makes the intern() method an appealing feature to prevent string explosion in the heap.*   
  
If you ever peeked through the Javadoc or the source code of the Java String class, you probably noticed a misterious native method called intern(). The [javadoc](http://java.sun.com/javase/6/docs/api/java/lang/String.html#intern%28%29) is very concise. It basically says that the method returns a representation of the String that is guaranteed to be unique through the JVM. If two String objects are internalized, they can be safely compared with == instead of equals. This description gives two reasons to use intern(): because comparision becomes faster and because there can be potential memory usage improvements because you wouldn't waste the heap with lots of equivalent strings.  
  
The two reasons above are closer to myth than reality. The performance myth doesn't cause any harm, it is just that the gain is not that big as one would think it would be. But the memory usage improvement myth is where the danger lies: by trying to improve memory usage, one can actually end up causing OOM errors in the application.  
  
Let's look at those myths with more detail.  
  
**Myth 1: Comparing strings with == is much faster than with equals()**  
  
An industrious developer could think of internalizing strings for performance reasons: you call intern() once, even if though it can be a costly operation, but then after that you can always compare the strings with ==. What a performance improvement it must be!  
  
I wrote a quick benchmark to compare both approaches. It turns out that comparing strings with average length of 16 characters using equals is approximately only 5 times slower than comparing with ==. Even though a 5 times difference is still a large number, you may be surprised that the gap isn't bigger. There are two reasons for this. First, String.equals() only compares the characters as a last effort: it first compares the length of the strings, which is stored in a separate field, and only if the lengths are the same it starts comparing the characters, but it halts as soon as it finds the first non matching character.  
  
Another reason for the relatively small difference between the two approaches is that the HotSpot optimizer does a very good job of optimizing method calls, and String.equals() is a very good candidate for inlining since it is a small method that belongs to a final class. That removes any overhead related to method calls.  
  
Now, == provides a 5-fold improvement over equals(). But since String comparision usually represents only a small percentage of the total execution time of an application, the overall gain is much smaller than that, and the final gain will be diluted to a few percent.  
  
So Myth 1: **busted!** Yes, == is faster than String.equals(), but in general it isn't near a performance improvement as it is cracked up to be.  
  
**Myth 2: String.intern() saves a lot of memory**  
  
This myth is where the danger lies. On one hand, it is true that you can remove String duplicates by internalizing them. The problem is that the internalized strings go to the Permanent Generation, which is an area of the JVM that is reserved for non-user objects, like Classes, Methods and other internal JVM objects. The size of this area is limited, and is usually much smaller than the heap. Calling intern() on a String has the effect of moving it out from the heap into the permanent generation, and you risk running out of PermGen space.  
  
I wrote a small test program that confirm this (see below). The call to Thread.sleep(1000) is so that you can see the permanent generation going up in a profiler. You can check it yourself by running this program and then running [jconsole](http://java.sun.com/developer/technicalArticles/J2SE/jconsole.html) which is available in the JDK distribution. Go to jconsole's *Memory* tab and select *Memory Pool "Perm Gen"* in the dropdown box. You will see the permanent generation going up steadly until the process terminates with a java.lang.OutOfMemoryError: PermGen space.

**import** java.util.ArrayList;

**import** java.util.List;

**public** **class** Main {

**public** **static** void **main**(String[] args) **throws** Exception {

int steps = 1000;

String base = **getBaseString**();

List strings = **new** **ArrayList**();

int i = 0;

**while** (**true**) {

String str = base + i;

str = str.**intern**();

strings.**add**(str);

i++;

**if** (i % steps == 0) {

Thread.**sleep**(1000);

}

}

}

**private** **static** String **getBaseString**() {

StringBuilder builder = **new** **StringBuilder**();

**for** (int i = 0; i < 1000; i++) {

builder.**append**("a");

}

**return** builder.**toString**();

}

}

So Myth 2: **busted!** String.intern() saves heap space, but at the expense of using up the more precious PermGen space.  
  
**Myth 3: internalized strings stay in the memory forever**  
  
This myth goes in the opposite direction of myth 2. Some people belive that internalized strings stay in the memory until the JVM ends. It may have been true a long time ago, but today the internalized strings are garbage collected if there are no more references to them. See below a slightly modified version of the program above. It clears the references to internalized strings from time to time. If you follow the program execution from jconsole, you will see that the PermGen space usage goes up and down, as the Garbage Collector reclaims the memory used by the unreferenced internalized strings.

**import** java.util.ArrayList;

**import** java.util.List;

**public** **class** Main {

**public** **static** void **main**(String[] args) **throws** Exception {

int steps = 1000;

String base = **getBaseString**();

List strings = **new** **ArrayList**();

int i = 0;

**while** (**true**) {

String str = base + i;

str = str.**intern**();

strings.**add**(str);

i++;

**if** (i % steps == 0) {

Thread.**sleep**(1000);

}

**if** (i % (steps \* 4) == 0) {

strings = **new** **ArrayList**();

}

}

}

**private** **static** String **getBaseString**() {

StringBuilder builder = **new** **StringBuilder**();

**for** (int i = 0; i < 1000; i++) {

builder.**append**("a");

}

**return** builder.**toString**();

}

}

Myth 3: **Busted!** Internalized strings are released if they are no longer referenced.  
  
**Note: when == is worth over equals()**  
  
If you are doing heavy text processing you may want to internalize strings. But in this case, you are probably better off using an approach that I outlined here: [Weak Object Pools With WeakHashMap](http://www.codeinstructions.com/2008/09/instance-pools-with-weakhashmap.html). With this approach, you get the benefit of having unique Strings, but without the penalty of using up the PermGen space.

**Conclusion: always know what you are doing**  
  
As I said in the subtitle of this article, String.intern() is dangerous **if you don't know what you are doing**. Now you know the risks of using String.intern(), and you will be able to make a more informed decision about whether to use it or not.

Comments Got:

1. Dont forget Matt Quigley, String.equals short-cuts the comparison with a this==that, then tries a length comparison, then finally tries a == comparison on the char array elements.  
     
   William Louth, looking up your cost table for method calls means absolutely nothing when you're running code in a JVM that auto-inlines methods and compiles to native code. Unless you're comparing a large number of identical length strings it would be very rare to enter the loop that iterates the char array.
2. In your loop you are constantly interning new strings. Of course that's going to run out of memory. base + i is going to result in the value of base concatenated with the number - "aaa...aa1", "aaa...aa2", etc. You are interning each of those unique strings. The code sample is pretty irrelevant. Try taking the "intern()" out. Do you eventually get an out of heap space error?   
     
   There are a number of places that interned strings make sense. If you are pulling data from a database and you have perhaps 100 or 200 companies in all the records, interning them ensures you are not duplicating the strings for each and every record.  
     
   This can't be done with public static final String unless you are going to manually enter every company into the class, recompile everything and then redeploy.  
     
   A 5\* speed up can be quite significant and I'm willing to bet it gets to be much higher than that.
3. If you are running Java 7, internalized Strings are stored in the heap (assuming you are using the HotSpot JVM). So that removes the problem of running out of PermGen space.

**Area:** HotSpot  
**Synopsis:** In JDK 7, interned strings are no longer allocated in the permanent generation of the Java heap, but are instead allocated in the main part of the Java heap (known as the young and old generations), along with the other objects created by the application. This change will result in more data residing in the main Java heap, and less data in the permanent generation, and thus may require heap sizes to be adjusted. Most applications will see only relatively small differences in heap usage due to this change, but larger applications that load many classes or make heavy use of the String.intern() method will see more significant differences.  
**RFE:** [6962931](http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=6962931)

**[WeakHashMap is not a cache! Understanding WeakReference and SoftReference](http://www.codeinstructions.com/2008/09/weakhashmap-is-not-cache-understanding.html)**

If you ever found yourself in the need to implement a simple caching functionality in your Java programs, chances are that you at least considered using the WeakHashMap class as the cache.  
  
It turns out that the WeakHashMap makes for a terrible cache, and for two reasons. The first reason is that it uses weak references as the underlying memory management mechanism. The second reason is that the weak references are used for the keys and not for the values, which is what you would want.  
  
**Reference classes and reachability**  
  
To understand what the WeakHashMap is good for, we need to understand the WeakReference and SoftReference classes and what is the difference between them. Both extend from the Reference class, which resides, along with its children, in the java.lang.ref package. The Reference classes are used to represent object references that are weaker than regular java references, which are called *strong references* Objects that can be reached by a chain of only strong references never get garbage collected. The weaker the references to an object, the more likely the object will be reclaimed by the garbage collector.  
  
The stronger type of reference is the strong reference, like when you declare String name = "John Doe";. The name variable is a *strong reference* to the "John Doe" String object. SoftReferences are weaker than strong references, and WeakReferences are weaker than SoftReferences. There is also an even weaker type of reference, the PhantonReference, of which I'm not going to talk about here.   
  
The type of references involved in the reference chain that starts from a local or a static variable and ends in an object defines the type of object's reachability. The Java API explains the different categories of object reachability this way:

* *An object is strongly reachable if it can be reached by some thread without traversing any reference objects. A newly-created object is strongly reachable by the thread that created it.*
* *An object is softly reachable if it is not strongly reachable but can be reached by traversing a soft reference.*
* *An object is weakly reachable if it is neither strongly nor softly reachable but can be reached by traversing a weak reference. When the weak references to a weakly-reachable object are cleared, the object becomes eligible for finalization.*
* *An object is phantom reachable if it is neither strongly, softly, nor weakly reachable, it has been finalized, and some phantom reference refers to it.*
* *Finally, an object is unreachable, and therefore eligible for reclamation, when it is not reachable in any of the above ways.*

if the garbage collector determines that an object is strongly reachable, it will not reclaim the object. This is what we would expect. Nobody wants to have an object garbage collected when it can still be reached by a chain of strong references. Now, here is the important point that is not written in the explanation above: If the garbage collector determines that an object is softly reachable, it **may** clear atomically all soft references to the object, in the case that it finds that memory is running low, or at its own discretion. But if the garbage collector determines that an object is weakly reachable, it **will** clear atomically all weak references to the object. This is the major difference between weak and soft references and the reason that makes the WeakReference ill-suited for caching.  
  
**What is the WeakHashMap good for?**  
  
Now it is easy to understand why the WeakHashMap doesn't work for caching. First of all it wouldn't work anyway because it uses soft references for the keys and not for the map values. But additional to that, the garbage collector aggressively reclaims the memory that is referenced only by weak references. It means that once you lose the last strong reference to an object that is working as a key in a WeakHashMap, the garbage collector will soon reclaim that map entry.  
  
If the WeakHashMap is no good for caching, then what is it good for? It is good to implement canonical maps. Lets say you want to associate some extra information to an object that you have a strong reference to. You put an entry in a WeakHashMap with the object as the key, and the extra information as the map value. Then, as long as you keep a strong reference to the object, you will be able to check the map to retrieve the extra information. And once you release the object, the map entry will be cleared and the memory used by the extra information will be released.  
  
**Can I just copy and paste the WeakHashMap class to write my cache?**  
  
No. Please, **don't** copy and paste the WeakHashMap source code replacing WeakReference with SoftReference. This won't be effective. To understand why, look at this example:

SoftHashMap cache = **new** **SoftHashMap**(); *// A copy and paste from WeakHashMap*  
SomeExpensiveClass myReference1 = .... *// get expensive class instance*  
cache.**put**(**new** **Long**(10), myReference1); *// put the expensive object in the cache*  
... *// do some stuff, but keep the myReference1 variable around!*  
SomeExpensiveClass myReference2 = cache.**get**(**new** **Long**(10)); *// query the cache*  
**if** (myReference2 == **null**) {  
 *// Uh-oh, the cache got rid of the object, even though I*  
 *// still had a reference to it in the myReference1 variable!*  
}

You would expect the cache to keep the reference to the object, since the myReference1 variable was still around. This may happen, but it may also happen that the cache will have been cleared. Why is this? Because at the "do some stuff" block, the garbage collector may have kicked in, noticed that the map's **key** was softly reachable, and chosen to garbage collect it. Remember, the WeakHashMap uses WeakReference for the keys, so your copy-paste implementation would use SoftReference for the keys. This reduces the effectiveness of the cache because you would expect the object to remain in the cache since the strong reference to it is still around. But the problem is that there is no strong reference to the key in the example above.  
  
**So how the hell do I implement a cache in Java?**  
  
My suggestion is to use one of the freely available Cache implementations, like JCS, OSCache and others. Those libraries provide better memory management with LRU and FIFO policies for instance, disk overflow, data expiration and many other optional advanced features.  
  
If you still want to implement a cache class that takes advantage of SoftReferences yourself, implement your own Map class that extends AbstractMap and internally wrap the map values in SoftReferences. Then you need to implement an internal mechanism to remove the stale entries from the map. For this, you may want to use a ReferenceQueue, and poll the ReferenceQueue for collected entries. Then, for each collected entry, you will need to remove it from the map. Make sure you do this in an efficient way! Or you may end up with some nested loops and O(n2) operations and your cache performance will suck.

## Tomcat – java.lang.OutOfMemoryError: PermGen space Cause and Solution

### Cause of OutOfMemoryError in PermGen space in Tomcat:

PermGen Space of heap is used to store classes and Meta data about [classes in Java](http://javarevisited.blogspot.com/2011/10/class-in-java-programming-general.html). When a class is loaded by a classloader it got stored in PermGen space, it gets unloaded only when the classloader which loaded this class got garbage collected. If any object retains reference of classloader than its not garbage collected and Perm Gen Space is not freed up. This causes memory leak in PermGen Space and eventually cause java.lang.OutOfMemoryError**: PermGen space.** Another important point is that when you deploy your web application a new Clasloader gets created and it loads the classes used by web application. So if Classloader doesn't get garbage collected when your web application stops you will have memoery leak in tomcat.

### Solution of Tomcat: OutOfMemroyError:

1) Find the offending classes which are retaining reference of Classloader and prventing it from being garbage collected. Tomcat provides memory leak detection functionality *after tomcat 6* onwards which can help you to find when particular library, framework or class is causing memory leak in tomcat. Here are some of the common causes of **java.lang.OutOfMemoryError: PermGen space in tomcat server**:

1) **JDBC Drivers:**

JDBC drivers are most common cause of java.lang.OutOfMemoryError: PermGen space in tomcat if web app doesn't unregister during stop. One hack to get around this problem is that JDBC driver to be loaded by common class loader than application classloader and you can do this by transferring driver's jar into tomcat lib instead of bundling it on web application's war file.

2) **Logging framework:**

Similar solution can be applied to prevent logging libraries like Log4j causing java.lang.OutOfMemoryError: PermGen space in tomcat.

3) **Application Threads which have not stopped.**

Check your code carefully if you are leaving your [thread](http://javarevisited.blogspot.com/2011/02/how-to-implement-thread-in-java.html) unattended and running in while loop that can retain classloader's reference and cause java.lang.OutOfMemoryError: PermGen space in tomcat web server. Another common culprint is ThreadLocal, avoid using it untily you need it absolutely, if do you make sure to set them null or free any object ThreadLocal's are holding.

Another Simple Solution is to increase PermGen [heap size](http://javarevisited.blogspot.com/2011/05/java-heap-space-memory-size-jvm.html) in catalina.bat or catalina.sh of tomcat server; this can give you some breathing space but eventually this will also return in *java.lang.OutOfMemoryError: PermGen space* after some time.

### Steps to increase PermGen Heap Space in Tomcat:

1) Go to Tomcat installation directory i.e C:\Program Files\Apache Software Foundation\Apache Tomcat 7.0.14\bin in Windows and something similar in linux.

2) Add JAVA\_OPTS in your catalina.bat or Catalina.sh

In Windows:

set JAVA\_OPTS="-Xms1024m -Xmx10246m -XX:NewSize=256m -XX:MaxNewSize=356m -XX:PermSize=256m -XX:MaxPermSize=356m"

In linux:

export JAVA\_OPTS="-Xms1024m -Xmx10246m -XX:NewSize=256m -XX:MaxNewSize=356m -XX:PermSize=256m -XX:MaxPermSize=356m"

You can change the actual heap size and PermGen Space as per your requirement.

3) Restart Tomcat.

As I said earlier **increasing PermGen space** can prevent **java.lang.OutOfMemoryError: PermGen** in tomcat only for some time and it will eventually occur based on how many times you redeploy your web application, its best to find the offending class which is causing memory leak in tomcat and fix it.

**Tools to investigate and fix OutOfMemoryError in Java**

Java.lang.OutOfMemoryError is a kind of error which needs lot of investigation to find out root cause of problem, which object is taking memory, how much memory it is taking or finding dreaded memory leak and you can't do this without having knowledge of available tools in java space. Here I am listing out some free tools which can be used to analyze heap and will help you to find culprit of OutOfMemoryError

**1) Visualgc**

Visualgc stands for Visual Garbage Collection Monitoring Tool and you can attach it to your instrumented hostspot JVM. Main strength of visualgc is that it displays all key data graphically including class loader, garbage collection and JVM compiler performance data.

The target JVM is identified by its virtual machine identifier also called as vmid. You can read more about visualgc and vmid options here.

**2) Jmap**

Jmap is a command line utility comes with JDK6 and allows you to take a memory dump of heap in a file. It’s easy to use as shwon below:

jmap -dump:format=b,file=heapdump 6054

Here file specifies name of memory dump file which is "heapdump" and 6054 is PID of your Java progress. You can find the PDI by using "ps -ef” or windows task manager or by using tool called "jps"(Java Virtual Machine Process Status Tool).

**3) Jhat**

Jhat was earlier known as hat (heap analyzer tool) but it is now part of JDK6. You can use jhat to analyze heap dump file created by using "jmap". Jhat is also a command line utility and you can rum it from cmd window as shown below:

jhat -J-Xmx256m heapdump

Here it will analyze memory-dump contained in file "heapdump". When you start jhat it will read this heap dump file and then start listening on http port, just point your browser into port where jhat is listening by default 7000 and then you can start analyzing objects present in heap dump.

**4) Eclipse memory analyzer**

Eclipse memory analyzer (MAT) is a tool from eclipse foundation to analyze java heap dump. It helps to find classloader leaks and memory leaks and helps to minimize memory consumption.you can use MAT to analyze heap dump carrying millions of object and it also helps you to extract suspect of memory leak. See here for more information.

**What is Heap space in Java?**

When a Java program started Java Virtual Machine gets some memory from Operating System. Java Virtual Machine or JVM uses this memory for all its need and part of this memory is call **java heap memory**. Heap in Java generally located at bottom of address space and move upwards. whenever we create object using new operator or by any another means object is allocated memory from Heap and When object dies or garbage collected ,memory goes back to **Heap space** in Java, to learn more about garbage collection see [how garbage collection works in Java](http://javarevisited.blogspot.com/2011/04/garbage-collection-in-java.html).

**How to increase size of Java Heap**

*Default size of Heap space  in Java is 128MB* on most of 32 bit Sun's [JVM](http://javarevisited.blogspot.sg/2011/12/jre-jvm-jdk-jit-in-java-programming.html) but its highly varies from JVM to JVM  e.g. default maximum and start heap size for the 32-bit Solaris Operating System (SPARC Platform Edition) is -Xms=3670K and -Xmx=64M and Default values of heap size parameters on 64-bit systems have been increased up by approximately 30%. Also if you are using throughput garbage collector in Java 1.5 default maximum heap size of JVM would be Physical Memory/4 and  default initial heap size would be Physical Memory/16. Another way to find default heap size of JVM is to start an application with default heap parameters and monitor in using JConsole which is available on JDK 1.5 onwards, on VMSummary tab you will be able to see maximum heap size.

By the way you can **increase size of java heap space** based on your application need and I always recommend this to avoid using default JVM heap values. if your application is large and lots of object created you can change size of heap space by using JVM options **-Xms and -Xmx**.  Xms denotes starting size of Heap while -Xmx denotes maximum size of Heap in Java. There is another parameter called -Xmn which denotes Size of new generation of **Java Heap Space**. Only thing is you can not change the size of Heap in Java dynamically, you can only provide Java Heap Size parameter while starting JVM. I have shared some more useful JVM options related to Java Heap space and Garbage collection on my post [10 JVM options Java programmer must know](http://javarevisited.blogspot.sg/2011/11/hotspot-jvm-options-java-examples.html), you may find useful.

**Update:**

Regarding default heap size in Java, from Java 6 update 18 there are significant changes in how JVM calculates default heap size in 32 and 64 bit machine and on client and server JVM mode:

1) initial heap space and maximum heap space is larger for improved performance.

2) default maximum heap space is 1/2 of physical memory of size upto 192 bytes and 1/4th of physical memory for size upto 1G. So for 1G machine maximum heap size is 256MB 2.maximum heap size will not be used until program creates enough object to fill initial heap space which will be much lesser but at-least 8 MB or 1/64th part of Physical memory upto 1G.

3) for Server Java virtual machine default maximum heap space is 1G for 4GB of physical memory on a 32 bit JVM. for 64 bit JVM its 32G for a physical memory of 128GB. Reference : <http://www.oracle.com/technetwork/java/javase/6u18-142093.html>

**Java Heap and Garbage Collection**

As we know **objects are created inside heap memory** and Garbage collection is a process which removes dead objects from Java Heap space and returns memory back to Heap in Java. For the sake of Garbage collection Heap is divided into three main regions named as New Generation, Old or Tenured Generation and Perm space. New Generation of Java Heap is part of Java Heap memory where newly created object are stored, During the course of application many objects created and died but those remain live they got moved to Old or Tenured Generation by Java Garbage collector thread on [Major or full garbage collection](http://javarevisited.blogspot.sg/2011/04/garbage-collection-in-java.html). Perm space of Java Heap is where JVM stores Meta data about classes and methods, String pool and Class level details. You can see How Garbage collection works in Java for more information on Heap in Java and Garbage collection.

**OutOfMemoryError in Java Heap**

When JVM starts JVM heap space is equal to the **initial size of Heap** specified by -Xms parameter, as application progress more objects get created and heap space is expanded to accommodate new objects. JVM also run garbage collector periodically to reclaim memory back from dead objects. JVM expands Heap in Java some where near to Maximum Heap Size specified by -Xmx and if there is no more memory left for creating new object in java heap , *JVM throws  java.lang.OutOfMemoryError* and  your application dies. Before throwing [OutOfMemoryError No Space in Java Heap](http://javarevisited.blogspot.sg/2011/09/javalangoutofmemoryerror-permgen-space.html), JVM tries to run garbage collector to free any available space but even after that not much space available on Heap in Java it results into OutOfMemoryError. To resolve this error you need to understand your application object profile i.e. what kind of object you are creating, which objects are taking how much memory etc. you can use profiler or heap analyzer to troubleshoot OutOfMemoryError in Java. **"**java.lang.OutOfMemoryError: Java heap space" error messages denotes that Java heap does not have sufficient space and cannot be expanded further while "java.lang.OutOfMemoryError: PermGen space**"** error message comes when the permanent generation of Java Heap is full, the application will [fail to load a class](http://javarevisited.blogspot.sg/2011/08/classnotfoundexception-in-java-example.html) or to allocate an interned string.

**Java Heap dump**

**Java Heap dump** is a snapshot of Java Heap Memory at a particular time. This is very useful to analyze or troubleshoot any memory leak in Java or any Java.lang.OutOfMemoryError. There are tools available inside JDK which helps you to take heap dump and there are heap analyzer available tool which helps you to analyze java heap dump. You can use "jmap" command to get java heap dump, this will create heap dump file and then you can use *"jhat - Java Heap Analysis Tool"* to analyze those heap dumps.

**How to increase Java heap space on Maven and ANT**

Many times we need to increase heap size of Maven or ANT because once number of classes increases build tool requires more memory to process and build and often throw OutOfMemoryError which we can avoid by changing or increase heap memory of JVM. For details see my post [How to increase java heap memory for Ant or Maven](http://javarevisited.blogspot.com/2011/05/java-heap-space-memory-size-jvm.html)

**10 Points about Java Heap Space**

1. Java Heap Memory is part of memory allocated to JVM by Operating System.

2. Whenever we create objects they are created inside Heap in Java.

3. Java Heap space is divided into three regions or generation for sake of garbage collection called New Generation, Old or tenured Generation or Perm Space. Permanent generation is garbage collected during full gc in hotspot JVM.

4. You can increase or change size of Java Heap space by using JVM command line option -Xms, -Xmx and -Xmn. don't forget to add word "M" or "G" after specifying size to indicate Mega or Gig. for example you can set java heap size to 258MB by executing following command java -Xmx256m HelloWord.

5. You can use either JConsole or Runtime.maxMemory(), Runtime.totalMemory(), Runtime.freeMemory() to query about Heap size programmatic in Java. See my post [How to find memory usage in Java program](http://javarevisited.blogspot.sg/2012/01/find-max-free-total-memory-in-java.html) for more details.

6. You can use command **"jmap"** to take Heap dump in Java and **"jhat"** to analyze that heap dump.

7. Java Heap space is different than Stack which is used to store call hierarchy and local variables.

8. [Java Garbage collector](http://javarevisited.blogspot.com/2011/04/garbage-collection-in-java.html) is responsible for reclaiming memory from dead object and returning to Java Heap space.

9. Don’t panic when you get java.lang.OutOfMemoryError, sometimes its just matter of increasing heap size but if it’s recurrent then look for [memory leak in Java](http://javarevisited.blogspot.sg/2012/01/tomcat-javalangoutofmemoryerror-permgen.html).

10. Use Profiler and Heap dump Analyzer tool to understand Java Heap space and how much memory is allocated to each object.

**Question: How much memory does the String “Hello World” consume?**  
Answer: 62/86 Bytes (32/64 bit Java)!  
This breaks down into 8/16 (Object Header for String) + 11 \* 2 (characters) + [8/16 (Object Header char Array) + 4 (array length) padded to 16/24] + 4 (Offset) + 4 (Count) + 4 (HashCode) + 4/8 (Reference to char Array). [On 64Bit the size of String Object is padded to 40].

## The Problem

Imagine you have a lot of Locations attached to tweets in your data store. The implementation of the location as a Java class could look like this

|  |
| --- |
| **class** Location {  **String** city;  **String** region;  **String** countryCode;  **double** **long**;  **double** lat;  } |

So if you load all the locations of tweets ever made, it is quite obvious that you load a lot of String objects, and at the scale Twitter has, there are for sure a lot of duplicate Strings. Attila said that this data did not fit into a 32 GB heap. So the question is: can we reduce memory consumption, so that all Locations fit into memory?  
  
Let us have a look at two possible solutions, which can even augment each other.

## Attilas Solution

There is a, more or less, hidden dependency between the data stored in the Location class, which, once realized, will solve the problem in an elegant way, non-technical way. We can just apply normalization to the Object and split it into two:

|  |
| --- |
| **class** SharedLocation {  **String** city;  **String** region;  **String** countryCode;  }  **class** Location {  SharedLocation sharedLocation;  **double** **long**;  **double** lat;  } |

This is actually a neat solution, because cities rarely change the region and country they reside in. The combination of those Strings is unique. And it is flexible, so that even violations of those uniqueness can be handled. This makes always sense for data which could be user input. By doing so, multiple Tweets from “Solingen, NRW, DE” just consume one SharedLocation.  
But still “Ratingen, NRW, DE” would store 3 additional String in memory, rather than just the new one “Ratingen”. By refactoring the data model like that the amount of data for that Twitter research project dropped to about 20GB.

## String interning

But what to do when you do not want to, or simply cannot refactor the data model? Or the researcher at twitter would not have had a 20GB Heap?  
The answer is String interning, which keeps every String only once in memory. But there is huge confusion about String interning. Many people ask if it speeds up equals checks, because equal strings are actually identical Strings when using interning. Yes, it might do that (like all objects should)

// java.lang.String

public boolean equals(Object anObject) {

if (this == anObject) {

return true;

}

//...

}

But equals performance is not the reason you should do interning. String interning is intended to reuse String objects to save memory.

Only use String.intern() on Strings you know are occurring multiple times, and only do it to save memory

How effective interning is, is determined by the ratio of duplicate/unique strings. And it depends on whether it is easy to change code at string generating places.

### So how does it work?

String interning takes a String instance (so it already exists in the Heap) and checks if an identical copy exists already in a StringTable.  
That StringTable is basically a HashSet that stores the String in the Permanent Generation. The only purpose of that Table is to keep a single instance of the String alive. If it is in there, the instance is returned. If its not, its added to the String Table:

|  |
| --- |
| *// OpenJDK 6 code*  JVM\_ENTRY(jstring, JVM\_InternString(JNIEnv \*env, jstring str))  JVMWrapper("JVM\_InternString");  JvmtiVMObjectAllocEventCollector oam;  if (str == NULL) return NULL;  oop string = JNIHandles::resolve\_non\_null(str);  oop result = StringTable::intern(string, CHECK\_NULL);  return (jstring) JNIHandles::make\_local(env, result);  JVM\_END    oop StringTable::intern(Handle string\_or\_null, jchar\* name,  int len, TRAPS) {  unsigned int hashValue = hash\_string(name, len);  int index = the\_table()->hash\_to\_index(hashValue);  oop string = the\_table()->lookup(index, name, len, hashValue);    *// Found*  if (string != NULL) return string;    *// Otherwise, add to symbol to table*  return the\_table()->basic\_add(index, string\_or\_null, name, len,  hashValue, CHECK\_NULL);  } |

As a result, this specific instance of a String only exists once.

### How to use interning

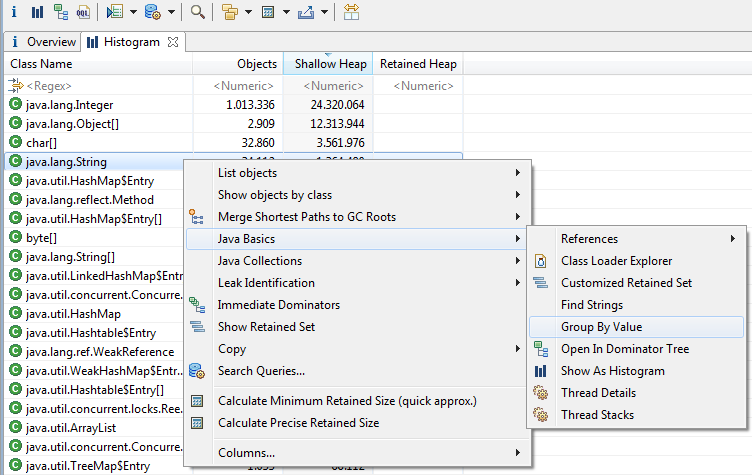
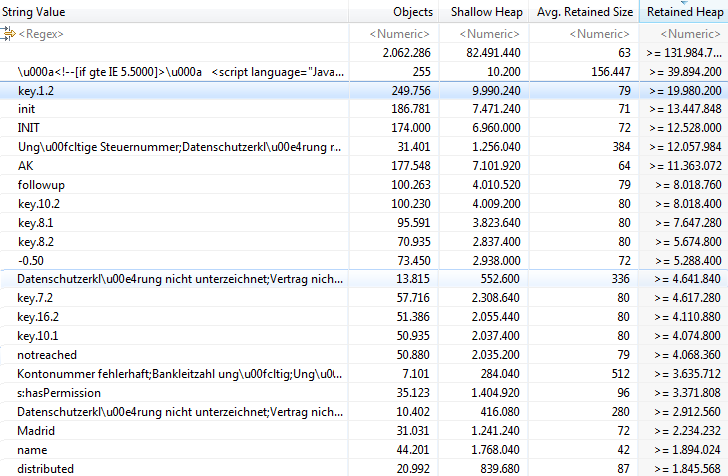
The right place to use String interning is where you read from data store and add the Objects/String to a larger scope. Note that ALL Strings which are hardcoded (as constant or anywhere in code) are automatically interned by the compiler.  
An example would be:

|  |
| --- |
| String city = resultSet.getString(1);  String region = resultSet.getString(2);  String countryCode = resultSet.getString(3);  **double** city = resultSet.getDouble(4);  **double** city = resultSet.getDouble(5);    Location location = **new** Location(city.intern(), region.intern(), countryCode.intern(), **long**, lat);  allLocations.add(location); |

All newly created location objects will use the interned string. The temporary strings read from the database will be garbage collected.

### How to find out how effective would be string interning

You are best off taking a heap dump of a quite full heap. It can even be collected at an OutOfMemoryError.  
Open it in MAT and select the java.lang.String from the histogram. On that one use “Java Basics” and “Group By Value”

  
Depending on the heap size, this may take a long time. In the end the result will be like this. Sorting either along retained heap or number of objects will reveal interesting things:  
  
From this screenshot we can see that empty Strings take a lot of memory! 2 million empty Strings take a total of 130MB. Then we see some amount of JavaScript that is loaded, a few more technical strings like the keys, which are used for localization. And we can see some Strings related to business logic.  
These business logic strings are probably the easiest to intern, because we might know where they are loaded into memory.  
For the other ones we need to use “Merge shortest Path to GC Root” to identify where they are stored, which may or may not reveal to use where it is created.

### Tradeoffs

So why not always do String interning? Because it slows your code down!  
Here a small example:

|  |
| --- |
| **private** **static** **final** **int** MAX = 40000000;  **public** **static** **void** main(**String**[] args) **throws** **Exception** {  **long** t = **System**.currentTimeMillis();  **String**[] arr = **new** **String**[MAX];  **for** (**int** i = 0; i < MAX; i++) {  arr[i] = **new** **String**(DB\_DATA[i % 10]);  *// and: arr[i] = new String(DB\_DATA[i % 10]).intern();*  }  **System**.out.println((**System**.currentTimeMillis() - t) + "ms");  **System**.gc();  **System**.out.println(arr[0]);  } |

The code uses a String array to keep a strong reference to the String objects. and we print the first element in the end to avoid removal of the structure due to optimization. Then we load 10 different Strings from the database. I used new String() here to illustrate the temp String allocation you always do when you read from storage. At the end I do a GC, so that the results are correct and no leftovers are included.  
This was run on a 64bit Windows, JDK 1.6.0\_27, i5-2520M with 8GB Ram. Run with -XX:+PrintGCDetails -Xmx6G -Xmn3G to log all GC activity. Here is the output:  
**Without intern()**

|  |
| --- |
| 1519ms  **[**GC **[**PSYoungGen: 2359296K-**>**393210K**(**2752512K**)]** 2359296K-**>**2348002K**(**4707456K**)**, 5.4071058 secs**]** **[**Times: user=8.84 sys=1.00, real=5.40 secs**]**  **[**Full GC **(**System**)** **[**PSYoungGen: 393210K-**>**392902K**(**2752512K**)]** **[**PSOldGen: 1954792K-**>**1954823K**(**1954944K**)]** 2348002K-**>**2347726K**(**4707456K**)** **[**PSPermGen: 2707K-**>**2707K**(**21248K**)]**, 5.3242785 secs**]** **[**Times: user=3.71 sys=0.20, real=5.32 secs**]**  DE  Heap  PSYoungGen total 2752512K, used 440088K **[**0x0000000740000000, 0x0000000800000000, 0x0000000800000000**)**  eden space 2359296K, 18**%** used **[**0x0000000740000000,0x000000075adc6360,0x00000007d0000000**)**  from space 393216K, 0**%** used **[**0x00000007d0000000,0x00000007d0000000,0x00000007e8000000**)**  to space 393216K, 0**%** used **[**0x00000007e8000000,0x00000007e8000000,0x0000000800000000**)**  PSOldGen total 1954944K, used 1954823K **[**0x0000000680000000, 0x00000006f7520000, 0x0000000740000000**)**  object space 1954944K, 99**%** used **[**0x0000000680000000,0x00000006f7501fd8,0x00000006f7520000**)**  PSPermGen total 21248K, used 2724K **[**0x000000067ae00000, 0x000000067c2c0000, 0x0000000680000000**)**  object space 21248K, 12**%** used **[**0x000000067ae00000,0x000000067b0a93e0,0x000000067c2c0000**)** |

**With intern()**

|  |
| --- |
| 4838ms  **[**GC **[**PSYoungGen: 2359296K-**>**156506K**(**2752512K**)]** 2359296K-**>**156506K**(**2757888K**)**, 0.1962062 secs**]** **[**Times: user=0.69 sys=0.01, real=0.20 secs**]**  **[**Full GC **(**System**)** **[**PSYoungGen: 156506K-**>**156357K**(**2752512K**)]** **[**PSOldGen: 0K-**>**18K**(**5376K**)]** 156506K-**>**156376K**(**2757888K**)** **[**PSPermGen: 2708K-**>**2708K**(**21248K**)]**, 0.2576126 secs**]** **[**Times: user=0.25 sys=0.00, real=0.26 secs**]**  DE  Heap  PSYoungGen total 2752512K, used 250729K **[**0x0000000740000000, 0x0000000800000000, 0x0000000800000000**)**  eden space 2359296K, 10**%** used **[**0x0000000740000000,0x000000074f4da6f8,0x00000007d0000000**)**  from space 393216K, 0**%** used **[**0x00000007d0000000,0x00000007d0000000,0x00000007e8000000**)**  to space 393216K, 0**%** used **[**0x00000007e8000000,0x00000007e8000000,0x0000000800000000**)**  PSOldGen total 5376K, used 18K **[**0x0000000680000000, 0x0000000680540000, 0x0000000740000000**)**  object space 5376K, 0**%** used **[**0x0000000680000000,0x0000000680004b30,0x0000000680540000**)**  PSPermGen total 21248K, used 2725K **[**0x000000067ae00000, 0x000000067c2c0000, 0x0000000680000000**)**  object space 21248K, 12**%** used **[**0x000000067ae00000,0x000000067b0a95d0,0x000000067c2c0000**)** |

We can see that the difference is significant. It took 3.3 seconds longer to run with interning. But the memory saved was enormous. When the code finished, the program using interning used 253472K(250M) of Memory. The other one used 2397635K (2.4G). That is quite a difference and illustrates nicely the tradeoffs when using String interning.