**10 Points about Java heap memory**

When I started java programming I didn't know **what is java heap**or**heap space in Java**, I was even not aware of where does object in Java gets created, it’s when I started doing professional programming I came across error [java.lang.OutOfMemoryError in Tomcat](http://javarevisited.blogspot.sg/2012/01/tomcat-javalangoutofmemoryerror-permgen.html) then I realized What is Heap in Java or Java Heap Space. Its happens with most of programmer because learning language is easy but learning basics is difficult since there is no formal process which can teach you every basics of programming its experience and work which reveals the secret of programming. For Java developer knowledge of Heap in Java, setting size of java heap space, [dealing with java heap space OutOfMemoryError](http://javarevisited.blogspot.sg/2011/09/javalangoutofmemoryerror-permgen-space.html), analyzing heap dumps is very important. This Java Heap tutorial is for my beginner brothers who are new in programming and learning it. It makes too much difference if you know the basics and underlying, until you know that object is created in heap, you won't be able to think why OutOfMemoryError occurs in Heap. I am trying to provide as much information about *Heap in Java* as I know but would like you guys to contribute and share your knowledge about Heap in Java to benefit all. By the way if you are confused between heap and stack, which is where your local variables get created, then, you can also check [difference between heap and stack memory in Java](http://javarevisited.blogspot.com.au/2013/01/difference-between-stack-and-heap-java.html).

**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.

Read more: <http://javarevisited.blogspot.com/2011/05/java-heap-space-memory-size-jvm.html#ixzz3Jolv3C8Y>

Difference between stack and heap memory is common [programming question](http://javarevisited.blogspot.com/2011/06/top-programming-interview-questions.html) asked by beginners learning Java or any other programming language. Stack and heap memory are two terms programmers starts hearing once they started programming but without any clear and definite explanation. Lack of knowledge on [what is heap in Java](http://javarevisited.blogspot.com/2011/05/java-heap-space-memory-size-jvm.html) and what is stack memory in Java, results in misconcepts related to stack and heap. To add to this confusion, stack is also a data structure which is used to store elements in LIFO(Last In First out) order and available in Java API as java.util.Stack. In general both stack and heap are part of memory, a program is allocated and used for different purposes. Java program runs inside JVM which is launched as a process by "java" command. Java also uses both stack and heap memory for different needs. In our last article [10 points on Java heap space](http://javarevisited.blogspot.sg/2011/05/java-heap-space-memory-size-jvm.html) I have touched base on Java heap space and in this article we will see difference between stack and heap memory in Java.

## Difference between Stack vs Heap in Java

[](http://1.bp.blogspot.com/-p7EVk3HnwGw/UCJuX-7mWpI/AAAAAAAAAb0/rTwwhrgC3Dc/s1600/di-logo-java-orange.png)

Here are few differences between stack and heap memory in Java:

1) Main difference between heap and stack is that stack memory is used to store [local variables](http://javarevisited.blogspot.com/2012/02/difference-between-instance-class-and.html) and function call, while heap memory is used to store objects in Java. No matter, where object is created in code e.g. as member variable, local variable or class variable,  they are always created inside heap space in Java.

2) Each [Thread in Java](http://javarevisited.blogspot.com/2011/02/how-to-implement-thread-in-java.html) has there own stack which can be specified using -Xss JVM parameter, similarly you can also specify heap size of Java program using JVM option -Xms and -Xmx where -Xms is starting size of heap and -Xmx is maximum size of java heap. to learn more about JVM options see my post [10 JVM option Java programmer should know](http://javarevisited.blogspot.com/2011/11/hotspot-jvm-options-java-examples.html).

3) If there is no memory left in stack for storing function call or local variable, JVM will throw java.lang.StackOverFlowError, while if there is no more heap space for creating object, JVM will throw java.lang.OutOfMemoryError: Java Heap Space. Read more about how to deal with java.lang.OutOfMemoryError  in my post [2 ways to solve OutOfMemoryError in Java](http://javarevisited.blogspot.com/2011/09/javalangoutofmemoryerror-permgen-space.html).

4) If you are using [Recursion](http://javarevisited.blogspot.com/2012/12/recursion-in-java-with-example-programming.html), on which method calls itself, You can quickly fill up stack memory. Another difference between stack and heap is that size of stack memory is lot lesser than size of  heap memory in Java.

5) Variables stored in stacks are only visible to the owner Thread, while objects created in heap are visible to all thread. In other words stack memory is kind of private memory of Java Threads, while heap memory is shared among all threads.

That's all on **difference between Stack and Heap memory in Java**. As I said, It’s important to understand what is heap and what is stack in Java and which kind of variables goes where, how you can run out of stack and heap memory in Java etc. Let us know if you are familiar with any other difference between stack and heap memory in java.

Read more: <http://javarevisited.blogspot.com/2013/01/difference-between-stack-and-heap-java.html#ixzz3Joma7Wp5>

I have read many articles on **Garbage Collection in Java**, some of them are too complex to understand and some of them don’t contain enough information required to understand*garbage collection in Java*. Then I decided to write my own experience as an article or you call tutorial about How **Garbage Collection works in Java** or **what is Garbage collection in Java** in simple word which would be easy to understand and have sufficient information to understand *how garbage collection works in Java.*  
  
[Garbage collection in Java Tutorial](http://javarevisited.blogspot.com/2011/04/garbage-collection-in-java.html)This article is  in continuation of my previous articles [How Classpath works in Java](http://javarevisited.blogspot.com/2011/01/how-classpath-work-in-java.html) and [How to write Equals method in java](http://javarevisited.blogspot.com/2011/02/how-to-write-equals-method-in-java.html) and  before moving ahead let's recall few important points about garbage collection in java:  
  
1) **objects are created on heap in Java**  irrespective of there scope e.g. local or member variable. while its worth noting that class variables or static members are created in method area of [Java memory space](http://javarevisited.blogspot.com/2011/05/java-heap-space-memory-size-jvm.html) and both heap and method area is shared between different thread.  
2) Garbage collection is a mechanism provided by Java Virtual Machine to **reclaim heap space** from objects which are**eligible for Garbage collection**.  
3) **Garbage collection**relieves java programmer from**memory management** which is essential part of C++ programming and gives more time to focus on business logic.  
4) **Garbage Collection in Java** is carried by a daemon thread called ***Garbage Collector***.  
5) Before removing an object from memory **Garbage collection thread invokes finalize () method**of that object and gives an opportunity to perform any sort of cleanup required.  
6) You as Java programmer **can not force Garbage collection in Java**; it will only **trigger** if JVM thinks it needs a garbage collection **based on**[**Java heap size**](http://javarevisited.blogspot.com/2011/05/java-heap-space-memory-size-jvm.html).  
7) There are methods like **System.gc ()** and **Runtime.gc** () which **is used to send request of Garbage collection to JVM** but it’s *not guaranteed that garbage collection will happen*.  
8) If there is no memory space for creating new object in Heap **Java Virtual Machine** throws **OutOfMemoryError**or [**java.lang.OutOfMemoryError heap space**](http://javarevisited.blogspot.com/2011/05/java-heap-space-memory-size-jvm.html)  
9) J2SE 5(Java 2 Standard Edition) adds a new feature called **Ergonomics** goal of ergonomics is to provide *good performance from the****JVM****with minimum of command line****tuning***.

### When an Object becomes Eligible for Garbage Collection

An Object becomes **eligible for Garbage collection or GC** if **its not reachable from any live threads or any static references** in other words you can say that an object becomes eligible for garbage collection if its *all references are null*. **Cyclic dependencies** are not counted as reference so if Object A has reference of object B and object B has reference of Object A and they don't have any other live reference then both Objects A and B will be ***eligible for Garbage collectio***n.   
Generally an object becomes *eligible for garbage collection in Java* on following cases:  
1) All references of that object explicitly set to null e.g. object = null  
2) Object is created inside a block and reference goes out scope once control exit that block.  
3) Parent object set to null, if an object holds reference of another object and when you set container object's reference null, child or contained object automatically becomes eligible for garbage collection.  
4) If an object has only live references via **WeakHashMap** it will be eligible for garbage collection. To learn more about HashMap see here [How HashMap works in Java](http://javarevisited.blogspot.com/2011/02/how-hashmap-works-in-java.html).

### Heap Generations for Garbage Collection in Java

Java objects are created in Heap and *Heap is divided into three parts or generations for sake of garbage collection in Java*, these are called as **Young generation, Tenured or Old Generation** and Perm **Area of heap**.   
New Generation is further divided into three parts known as **Eden space**, **Survivor 1** and **Survivor 2** space*. When an object first created in heap its gets created in new generation inside Eden space* and after subsequent **Minor Garbage collection** if object survives its gets moved to survivor 1 and then Survivor 2 before **Major Garbage collection** moved that object to **Old or tenured generation**.  
  
*Permanent generation of Heap or Perm Area of Heap* is somewhat special and it is used to store Meta data related to classes and method in JVM, it also hosts **String pool provided by JVM** as discussed in my string tutorial [**why String is immutable in Java**](http://javarevisited.blogspot.com/2010/10/why-string-is-immutable-in-java.html). There are many opinions around *whether garbage collection in Java happens in perm area of java heap or not*, as per my knowledge this is something which is JVM dependent and happens at least in Sun's implementation of JVM. You can also try this by just creating millions of String and watching *for Garbage collection or OutOfMemoryError*.

### Types of Garbage Collector in Java

Java Runtime (J2SE 5) provides various **types of Garbage collection in Java** which you can choose based upon your application's performance requirement. Java 5 adds three additional **garbage collectors** except **serial garbage collector**. Each is **generational garbage collector** which has been implemented to increase throughput of the application or to reduce **garbage collection pause times**.  
  
1) **Throughput Garbage Collector**: This garbage*collector* in Java uses a parallel version of the *young generation collector*. It is used if the *-XX:+UseParallelGC* option is passed to *the JVM via command line options* . The tenured generation collector is same as the serial collector.  
  
2) **Concurrent low pause Collector**: This Collector is used if the -Xingc or **-XX:+UseConcMarkSweepGC** is passed on the command line. This is also referred as **Concurrent Mark Sweep Garbage collector**. The concurrent collector is used to collect the tenured generation and does most of the collection concurrently with the execution of the application. The application is paused for short periods during the collection. A parallel version of the *young generation* *copying collector* is sued with the concurrent collector. Concurrent Mark Sweep Garbage collector is most widely used garbage collector in java and it uses algorithm to first mark object which needs to collected when garbage collection triggers.  
  
3) **The Incremental (Sometimes called train) low pause collector**: This collector is used only if -**XX:+UseTrainGC** is passed on the command line. This garbage collector has not changed since the java 1.4.2 and is currently not under active development. It will not be supported in future releases so avoid using this and please see 1.4.2 GC Tuning document for information on this collector.  
Important point to not is that **-XX:+UseParallelGC** should not be used with **-XX:+UseConcMarkSweepGC**. The argument passing in the J2SE platform starting with version 1.4.2 should only allow legal combination of command line options for garbage collector but earlier releases may not find or detect all illegal combination and the results for illegal combination are unpredictable. It’s not recommended to use this garbage collector in java.

### JVM Parameters for garbage collection in Java

**Garbage collection tuning**is a long exercise and requires lot of profiling of application and patience to get it right. *While working with High volume low latency Electronic trading system*I have worked with some of the project where we need to increase the performance of Java application by profiling and finding what causing full GC and I found that*Garbage collection tuning* largely depends on application profile, what kind of object application has and what are there average lifetime etc. for example if an application has too many short lived object then making Eden space wide enough or larger will reduces number of minor collections. you can also control size of both young and Tenured generation using **JVM** **parameters**for example setting -XX:NewRatio=3 means that the ratio among the young and tenured generation is 1:3 , you got to be careful on sizing these generation. As **making young generation larger will reduce size of tenured generation which will force Major collection** to occur more frequently which pauses application thread during that duration results in degraded or reduced throughput. The parameters **NewSize** and **MaxNewSize** are used to specify the young generation size from below and above. Setting these equal to one another fixes the *young generation*. In my opinion before doing garbage collection tuning detailed understanding of garbage collection in java is must and I would recommend reading Garbage collection document provided by Sun Microsystems for detail knowledge of *garbage collection in Java*. Also to get a full list of JVM parameters for a particular Java Virtual machine please refer official documents on garbage collection in Java. I found this link quite helpful though http://www.oracle.com/technetwork/java/gc-tuning-5-138395.html

### Full GC and Concurrent Garbage Collection in Java

**Concurrent garbage collector** in java uses a single **garbage collector thread** that ***runs concurrently****with the application threads* with the goal of completing the collection of the tenured generation before it becomes full. In normal operation, the concurrent garbage collector is able to do most of its work with the application threads still running, so only brief pauses are seen by the application threads. As a fall back, if the **concurrent garbage collector** is unable to finish before the tenured generation fill up, the application is paused and the collection is completed with all the application threads stopped. Such Collections with the application stopped are referred as **full garbage collections** or **full GC** and are a sign that some adjustments need to be made to the concurrent collection parameters. Always try to avoid or minimize **full garbage collection** or **Full GC**because it affects **performance of Java application**. When you work in finance domain for electronic trading platform and with high volume low latency systems performance of java application becomes extremely critical an you definitely like to avoid full GC during trading period.

## Summary on Garbage collection in Java

1) Java Heap is divided into *three generation for sake of garbage collection*. These are **young generation**, **tenured or old generation** and **Perm area*.***  
2) New objects are created into ***young generation*** and subsequently moved to **old generation**.  
*3) String pool is created in*[***Perm***](http://www.blogger.com/goog_720812146)[***area of Heap***](http://javarevisited.blogspot.com/2011/05/java-heap-space-memory-size-jvm.html)***, garbage collection can occur in perm space****but depends upon JVM to JVM.*  
4) **Minor garbage collection** is used to ***move object from Eden space to Survivor 1 and Survivor 2*** space and **Major collection**is used to move object from***young to tenured generation***.  
5) Whenever ***Major garbage collection*** occurs application threads stops during that period which will reduce application’s **performance** and **throughput**.  
6) There are few performance improvement has been applied in **garbage collection in java 6** and we usually use JRE 1.6.20 for running our application.  
7) **JVM command line options –Xmx and -Xms i**s used to [setup starting and max size for **Java Heap**.](http://javarevisited.blogspot.com/2011/05/java-heap-space-memory-size-jvm.html) Ideal ratio of this parameter is either **1:1** or 1:1.5 based upon my experience for example you can have either both **–Xmx and –Xms as 1GB or –Xms 1.2 GB and 1.8 GB**.  
8) There is no manual way of doing garbage collection in Java.  
  
Read more: <http://javarevisited.blogspot.com/2011/04/garbage-collection-in-java.html#ixzz3Jomo2AhL>

# [10 Examples of HotSpot JVM Options in Java](http://javarevisited.blogspot.sg/2011/11/hotspot-jvm-options-java-examples.html)

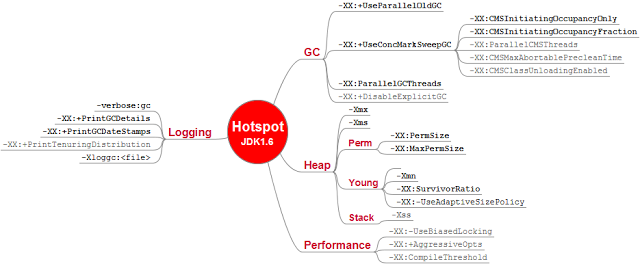
There are hundreds of **JVM parameters** or **JVM Options** exists inside sun JDK and its virtually impossible to keep track of every single [JVM](http://javarevisited.blogspot.com/2011/12/jre-jvm-jdk-jit-in-java-programming.html)option and based on my experience we don't even use most of JVM flags except couple of important JVM option related to java heap size, java options for printing garbage collection details and most likely JVM switches for setting up remote debugging in Java. but there are many other useful category of JVM parameters which you at least like to be familiar even if not intending to use it more frequently. In this article we will see examples of 10 different categories of **JVM parameter** which I found useful and use more frequently than other. I would recommend to get a full knowledge of what does a particular JVM options does by referring official list of JVM options.

## JVM parameters in Java

On the basis of how we specify **JVM option it can be divided into two parts**, JVM Options which starts with –X and those which starts with -XX:

1)    *JVM Options that begin with -X* are non-standard (thy are not guaranteed to be supported on all JVM implementations), and are subject to change without notice in subsequent releases of the JDK.

2)    *JVM Options or parameters which are specified with -XX* are not stable and are not recommended for casual use. These options are subject to change without notice also.

[](http://4.bp.blogspot.com/-KMPS7glbJhY/T8cc_8QtQqI/AAAAAAAAAX8/hmMqLBx7m4k/s1600/Hotspot2_JVM_Parameters_GC_Heap.png)

I was thinking about writing post on JVM options when I completed my post on [**Java Heap Size**](http://javarevisited.blogspot.com/2011/05/java-heap-space-memory-size-jvm.html) and [**Java Garbage Collection**](http://javarevisited.blogspot.com/2011/04/garbage-collection-in-java.html) because these are two main area where we see usages of various JVM flags. But it didn’t happened even after I covered OutOfMemoryError post which has some[**JVM option to solve OutOfMemoryError in Java**](http://javarevisited.blogspot.com/2011/09/javalangoutofmemoryerror-permgen-space.html). Now I am happy that I have completed this piece of information and its ready to be published. As always I look for your feedback, suggestions and any other JVM flags which I have missed and you guys find useful to share.   
  
Good knowledge of JVM options specially related to GC tuning is important for time critical application e.g. **high volume low latency electronic trading platform** where every micro seconds matter. though getting right combination requires lot of profiling and trial and error and depends heavily on nature of trading application.

### Important Points about JVM Options:

1)    Boolean JVM options can be  turned on with -XX:+ and can be turned off with -XX:-.

2)    Numeric JVM Options can be set with -XX:=. Numbers can include 'm' or 'M' for megabytes, 'k' or 'K' for kilobytes, and 'g' or 'G' for gigabytes (for example, 32k is the same as 32768).

3)    String JVM options can be set by using -XX:=, and usually used to specify a file, a path, or a list of commands.

The command **java -help** lists the standard options (standard across different JVM implementations) for the Java application launcher. The **command java -X can be used to see the Java application launcher's non-standard** (X for extension specific to that JVM) arguments.The -X options are non-standard and subject to change without notice. If you wish to detect which JVM arguments your currently running Java application is using, you can use the ManagementFactory.getRuntimeMXBean().getInputArguments()

Now here is my list of important JVM flags, switches, options or parameters which is most commonly used while running Java applications:

**1) JVM memory options related to java heap size**

Following three JVM options are used to specify initial and max heap size and thread stack size while running Java programs.

**-Xms**        set initial Java heap size

**-Xmx**        set maximum Java heap size

**-Xss**>         set java thread stack size

**2) JVM option to print gc details**

**-verbose:gc** logs garbage collector runs and how long they're taking. I generally use this as my first tool to investigate if GC is a bottleneck for a given application.

**-XX:+PrintGCDetails** includes the data from -verbose:gc but also adds information about the size of the new generation and more accurate timings.

**-XX:-PrintGCTimeStamps**  Print timestamps at garbage collection.

**3) JVM parameters to specify Java Garbage collector**

**-XX:+UseParallelGC**      Use parallel garbage collection for scavenges

**-XX:-UseConcMarkSweepGC** Use concurrent mark-sweep collection for the old generation. (Introduced in 1.4.1)

**-XX:-UseSerialGC**        Use serial garbage collection. (Introduced in 5.0.)  
  
beware when you use GC Parameters if you are working on time critical application e.g. high frequency trading application. As  GC is time consuming operation and its desired to create a balance.

**4) JVM debug options JVM options for remote debugging**

-Xdebug -Xnoagent -Xrunjdwp:transport=dt\_socket,server=y,suspend=n,address=8000  
to read more about remote debugging check [How to Setup Java remote debugging in Eclipse](http://javarevisited.blogspot.com/2011/02/how-to-setup-remote-debugging-in.html) and [10 Java debugging tips in Eclipse](http://javarevisited.blogspot.com/2011/07/java-debugging-tutorial-example-tips.html)

**5) JVM options related to profiling**

-Xprof

-Xrunhprof

**6) JVM options related to java classpath**

**Xbootclasspath** specifies classpath entries you want loaded without verification. The JVM verifies all classes it loads to ensure they don't try to dereference an object with an int, pop extra entries off the stack or push too many, and so on. This verification is part of the reason why the JVM is very stable, but it's also rather costly, and responsible for a large part of start up delay. Putting classes on the bootclasspath skips this cost, but should only be used when you know the classes have been verified many times before. In JRuby, this reduced startup time by half or more for a simple script. The -**Xbootclasspath** option can be used to either prepend (/p) or append (/a) resources to the bootstrap classpath. You Can read more about Java Classpath in my articles [How Classpath Works in Java](http://javarevisited.blogspot.com/2011/01/how-classpath-work-in-java.html) and [How to Solve ClassNotFoundException in Java](http://javarevisited.blogspot.com/2011/08/classnotfoundexception-in-java-example.html)

**7) JVM options to change  Perm Gen Size**

These JVM optiosn are quite useful to solve [java.lang.OutOfMemoryError:Perm Gen Space](http://javarevisited.blogspot.com/2011/09/javalangoutofmemoryerror-permgen-space.html).

-XX:PermSize and MaxPermSize

-XX:NewRatio=2  Ratio of new/old generation sizes.

-XX:MaxPermSize=64m     Size of the Permanent Generation.

**8) JVM parameters to trace classloading and unloading**

**-XX:+TraceClassLoading** and **-XX:+TraceClassUnloading** are two JVM options which we use to print logging information whenever classes loads into JVM or unloads from JVM. These JVM flags are extremely useful if you have any memory leak related to classloader and or suspecting that classes are not unloading or garbage collected.

**9) JVM switches related to logging**

-XX:+TraceClassLoading and -XX:+TraceClassUnloading print information class loads and unloads. Useful for investigating if you have a class leak or if old classes (like JITed Ruby methods in JRuby) are getting collected or not. You can read more about logging in Java on my post [10 Tips while logging in Java](http://javarevisited.blogspot.com/2011/05/top-10-tips-on-logging-in-java.html)

**-XX:+PrintCompilation** prints out the name of each Java method Hotspot decides to JIT compile. The list will usually show a bunch of core Java class methods initially, and then turn to methods in your application. In JRuby, it eventually starts to show Ruby methods as well

**10) JVM Switches for debugging purpose**

-XX:HeapDumpPath=./java\_pid.hprof  Path to directory or file name for heap dump.

-XX:-PrintConcurrentLocks       Print java.util.concurrent locks in Ctrl-Break thread dump.

-XX:-PrintCommandLineFlags   Print flags that appeared on the command line.

That’s all on JVM Options, I understand its not possible to remember all JVM flags but at-least having an idea of what kind of JVM flags are available is good asset. Image for JVM parameters is from Java tuning and Nutshell.  For full list of JVM options you can refer these link from Oracle Java site: [Java Hotspot VM Options](http://www.oracle.com/technetwork/java/javase/tech/vmoptions-jsp-140102.html)

Read more: <http://javarevisited.blogspot.com/2011/11/hotspot-jvm-options-java-examples.html#ixzz3Jzcm8BT0>

# [2 solution of java.lang.OutOfMemoryError in Java](http://javarevisited.blogspot.sg/2011/09/javalangoutofmemoryerror-permgen-space.html)

Every one in java development face **java.lang.OutOfMemoryError** now and then, OutOfMemoryError in Java is one problem which is more due to system's limitation (memory) rather than due to programming mistakes in most cases though in certain cases you could have **memory leak** which causing **OutOfMemoryError**. I have found that even though java.lang.OutOfMemoryError is quite common basic knowledge of its cause and solution is largely unknown among junior developers. In this article we will explore *what is java.lang.OutOfMemoryError*; Why OutOfMemoryError comes in Java application, different type of OutOfMemoryError and *How to fix OutOfMemoryError in Java*. This article is purely meant to provide basic knowledge of java.lang.OutMemoryError and won't discuss profiling in detail.

### What is java.lang.OutOfMemoryError in Java

[java.lang.OutOfMemoryError in Java, PermGen space or heap space](http://javarevisited.blogspot.com/2011/07/javalangunsupportedclassversionerror.html)OutOfMemoryError in Java is a subclass **of java.lang.VirtualMachineError** and JVM throws java.lang.OutOfMemoryError when it ran *out of memory in heap*. OutOfMemoryError in Java can come any time in heap mostly while you try to create an object and there is not enough space in heap to allocate that object. [javavdoc of OutOfMemoryError](http://download.oracle.com/javase/6/docs/api/) is not very informative about this though.

### Types of OutOfMemoryError in Java

I have seen mainly two types of OutOfMemoryError in Java:  
  
1) **Java.lang.OutOfMemoryError: Java heap space**  
2) **Java.lang.OutOfMemoryError: PermGen space**  
  
Though both of them occur because JVM ran out of memory they are quite different to each other and there solutions are independent to each other.

### Difference between "java.lang.OutOfMemoryError: Java heap space" and "java.lang.OutOfMemoryError: PermGen space"

If you are familiar with different generations on heap and [How garbage collection works in java](http://javarevisited.blogspot.com/2011/04/garbage-collection-in-java.html) and aware of new, old and permanent generation of heap space then you would have easily figured out this OutOfMemoryError in Java. Permanent generation of heap is used to store String pool and various Meta data required by JVM related to Class, method and other java primitives. Since **in most of JVM default size of Perm Space is around "64MB"** you can easily ran out of memory if you have too many classes or huge number of Strings in your project. Important point to remember is that it doesn't depends on **–Xmx**value so no matter how big your total heap size you can ran OutOfMemory in perm space. Good think is you can specify**size of permanent generation** using JVM options **"-XX:PermSize"** and  **"-XX:MaxPermSize"** based on your project need.  
  
One small thing to remember is that "=" is used to separate parameter and value while specifying **size of perm space in heap** while "=" is not required while [**setting maximum heap size in java**](http://javarevisited.blogspot.com/2011/05/java-heap-space-memory-size-jvm.html), as shown in below example.  
  
**export JVM\_ARGS="-Xmx1024m -XX:MaxPermSize=256m"**

Another reason of "**java.lang.OutOfMemoryError: PermGen**" is memory leak through Classloaders and it’s very often surfaced in WebServer and application server like tomcat, webshere, glassfish or weblogic. In Application server different classloaders are used to load different web application so that you can deploy and undeploy one application without affecting other application on same server, but while undeploying if container some how keeps reference of any class loaded by application class loader than that class and all other related class will not be garbage collected and can quickly fill the PermGen space if you deploy and undeploy your application many times. "*java.lang.OutOfMemoryError: PermGen*” has been observed many times in tomcat in our last project but solution of this problem are really tricky because first you need to know which class is causing memory leak and then you need to fix that. Another reason of OutOfMemoryError in PermGen space is if any thread started by application doesn't exit when you undeploy your application.

These are just some example of infamous classloader leaks, anybody who is writing code for loading and unloading classes have to be very careful to avoid this. You can also use **visualgc**for monitoring PermGen space, this tool will show graph of PermGen space and you can see how and when Permanent space getting increased. I suggest using this tool before reaching to any conclusion.

Another rather unknown but interesting cause of "java.lang.OutOfMemoryError: PermGen" we found is introduction of JVM options **"-Xnoclassgc**". This option sometime used to avoid loading and unloading of classes when there is no further live references of it just to avoid performance hit due to frequent loading and unloading, but using this option is J2EE environment can be very dangerous because many framework e.g. Struts, spring etc uses reflection to create classes and with frequent deployment and undeployment you can easily ran out of space in **PermGen**if earlier references was not cleaned up. This instance also points out that some time bad JVM arguments or configuration can cause OutOfMemoryError in Java.

So conclusion is avoid using ***"-Xnoclassgc*"** in J2EE environment especially with AppServer.

### Tomcat to Solve OutOfMemoryError in PermGen Space

From tomcat > 6.0 onward tomcat provides memory leak detection feature which can detect many common memory leaks on web-app perspective e.g ThreadLocal memory leaks, JDBC driver registration, RMI targes, LogFactory and Thread spawned by web-apps. You can check complete details on htp://wiki.apache.org/tomcat/MemoryLeakProtection you can also detect memory leak by accessing manager application which comes with tomcat, in case you are experiencing memory leak on any java web-app its good idea to run it on tomcat.

### How to solve java.lang.OutOfMemoryError: Java heap space

1) Easy way to solve OutOfMemoryError in java is to [*increase the maximum heap size*](http://javarevisited.blogspot.com/2011/08/increase-heap-size-maven-ant.html) by using JVM options "-Xmx512M", this will immediately solve your OutOfMemoryError. This is my preferred solution when I get OutOfMemoryError in Eclipse, Maven or ANT while building project because based upon size of project you can easily ran out of Memory.here is **an example of increasing maximum heap size of JVM**, Also its better to keep **-Xmx to -Xms**ration either 1:1 or 1:1.5 if you are setting heap size in your java application  
  
**export JVM\_ARGS="-Xms1024m -Xmx1024m"**  
  
2) Second way to resolve OutOfMemoryError in Java is rather hard and  comes when you don't have much memory and even after increase maximum heap size you are still getting java.lang.OutOfMemoryError, in this case you probably want to profile your application and look for any memory leak. You can use [**Eclipse Memory Analyzer**](http://www.eclipse.org/mat/) to examine your heap dump or you can use any profiler like Netbeans or JProbe. This is tough solution and requires some time to analyze and **find memory leaks**.

### How to solve java.lang.OutOfMemoryError: PermGen space

As explained in above paragraph this OutOfMemory error in java comes when Permanent generation of heap filled up. To fix this OutOfMemoryError in Java you need to *increase heap size of Perm space* by using JVM option   **"-XX:MaxPermSize".**You can also specify initial size of Perm space by using    **"-XX:PermSize"** and keeping both initial **and maximum Perm Space** you can prevent some full garbage collection which may occur when Perm Space gets re-sized. Here is **how you can specify initial and maximu Perm size in Java**:  
  
**export JVM\_ARGS="-XX:PermSize=64M -XX:MaxPermSize=256m"**  
  
Some time java.lang.OutOfMemoryError  in Java gets tricky and on those cases profiling remain ultimate solution.Though you have freedom to increase heap size in java, it’s recommended that to follow memory management practices while coding and setting null to any unused references.  
That’s all from me on **OutOfMemoryError in Java** I will try to write more about finding memory leak in java and using profiler in some other post. Please share what is your approach to solve *java.lang.OutOfMemoryError in Java*.

**Important Note:** From Tomcat > 6.0 onward tomcat provides memory leak detection feature which can detect many common memory leaks on Java application e.g ThreadLocal memory leaks, JDBC driver registration, RMI targes, LogFactory and Thread spawned by webapps. You can check complete details on htp://wiki.apache.org/tomcat/MemoryLeakProtection. You can also detect memoy leak by accessing manager application which comes with tomcat, in case you are experiencing memory leak on any java webapp its good idea to run it on tomcat to find out reason of OutOfMemoryError in PermGen space.

## 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.

Here are some of my other post on Java you may find interesting:

Read more: <http://javarevisited.blogspot.com/2011/09/javalangoutofmemoryerror-permgen-space.html#ixzz3JzwIGaGf>

# [How to get max memory, free memory and total memory in Java](http://javarevisited.blogspot.sg/2012/01/find-max-free-total-memory-in-java.html)

Getting **free memory**, **total memory** and **max memory** on [JVM](http://javarevisited.blogspot.com/2011/12/jre-jvm-jdk-jit-in-java-programming.html) is using by **Runtime** class in Java. and many java programmer is interested to know whether they have any relationship with JVM argument **-Xms** and **-Xmx** which is used to specify starting heap size and maximum heap size of JVM. I have touched this on [10 Points on Java heap](http://javarevisited.blogspot.com/2011/05/java-heap-space-memory-size-jvm.html) and [how to set heap size in ANT and Maven](http://javarevisited.blogspot.com/2011/08/increase-heap-size-maven-ant.html) but here we will see some way to find out starting and maximum heap size from Java program.

## How to find max memory, free memory and total memory in Java

[how to find free memory, total memory and max memory in java](http://javarevisited.blogspot.com/2011/11/static-keyword-method-variable-java.html)As per Javadoc freeMemory is currently available memory which can be allocated to future objects and **totalMemory**is the total amount of memory in the Java virtual. Since we know that **JVM expands heap** whenever it needs so if we start our JVM with -Xms10m and -Xmx120m you should expect that initial **freeMemory**and totalMemory should be same with starting heap size of JVM as Virtual machine

has not been expanded yet and that's the case exactly. even value returned by Runtime.maxMemory() will be close to value of -Xmx though little less. In this article we will see *how to get approximate value of inital and maximum heap size*, free memory available in JVM and used memory or memory currently occupied by objects in heap.

### How to get free Memory in Java

In order to get currently *free Memory available*for creating object use Runtime.getRuntime().freeMemory() method, this will return **size in bytes**, which you convert in Mega Bytes for better readability. we will see an example of getting initial heap and free memory in code example section. Calling Garbage collector by either System.gc() or Runtime.gc() may results in slightly higher free memory reclaimed by dead objects.

### How to get total Memory in Java

You can use Runtime.getRuntime.totalMemory() to **get total memory from JVM** which represent current heap size of JVM which is combination of used memory currently occupied by objects and free memory available for new objects. As per javadoc value returned by totalMemory() may vary over time depending upon environment. see code **example for getting total memory in Java** in next code example section.

### How to get initial Heap Size from Java Program

We specify ***initial heap space*** by using -Xms and JVM creates initial heap with this much size. in order to get this size from Java program call Runtime.getRuntime.totalMemory() before creating any object. See code example of getting initial heap size from java program in next section. Apart from –Xms and –Xmx there are lot of other useful JVM Options I have shared on my post [10 useful JVM parameters Java Programmer should know](http://javarevisited.blogspot.com/2011/11/hotspot-jvm-options-java-examples.html).

### How to get maximum Heap Size from Java Program

This is relatively easy as ***maximum heap space*** is not going to change over JVM life cycle and call to Runtime.getRuntime.maxMemory() will return value close to -Xmx but keep in mind exact value will be little less than what have you set.

### How to get Used Memory in JVM

by using Runtime.getRuntime.totalMemory() and Runtime.getRuntime.freeMemory() we can calculate how much space has been currently occupied by Java object or you say used memory in JVM as show in below code example of getting memory sizes in Java:

### Code Example of getting heap memory in Java program:

In below example we get **initial size of heap** by calling ***freeMemory, totalMemory and max memory*** at start of program and then we create thousands of object which occupy space in heap and not eligible for garbage collection which forces JVM to extend heap. now call to total memory, free memory will return different value based on current heap size but max memory will still return same. try creating some more object and you will be greeted with [java.lang.OutOfMemoryError](http://javarevisited.blogspot.com/2011/09/javalangoutofmemoryerror-permgen-space.html) :)

**public** **class** MemoryUtil{

**private** **static** **final** **int** *MegaBytes* = 10241024;

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

**long** freeMemory = Runtime.*getRuntime*().freeMemory()/*MegaBytes*;

**long** totalMemory = Runtime.*getRuntime*().totalMemory()/*MegaBytes*;

**long** maxMemory = Runtime.*getRuntime*().maxMemory()/*MegaBytes*;

              System.*out*.println("JVM freeMemory: " + freeMemory);

              System.*out*.println("JVM totalMemory also equals to initial heap size of JVM : "

                                         + totalMemory);

              System.*out*.println("JVM maxMemory also equals to maximum heap size of JVM: "

                                         + maxMemory);

              ArrayList objects = **new** ArrayList();

**for** (**int** i = 0; i < 10000000; i++) {

                     objects.add(("" + 10 \* 2710));

              }

              freeMemory = Runtime.*getRuntime*().freeMemory() / *MegaBytes*;

              totalMemory = Runtime.*getRuntime*().totalMemory() / *MegaBytes*;

              maxMemory = Runtime.*getRuntime*().maxMemory() / *MegaBytes*;

              System.*out*.println("Used Memory in JVM: " + (maxMemory - freeMemory));

              System.*out*.println("freeMemory in JVM: " + freeMemory);

              System.*out*.println("totalMemory in JVM shows current size of java heap : "

                                         + totalMemory);

              System.*out*.println("maxMemory in JVM: " + maxMemory);

       }

}

**Output:**

JVM freeMemory: 9

JVM totalMemory also equals to initial heap size of JVM : 9

JVM maxMemory also equals to maximum heap size of JVM: 116

Used Memory in JVM: 81

freeMemory in JVM: 35

totalMemory in JVM shows current size of java heap : 108

maxMemory in JVM: 116

That’s all on **how to get free, total and max memory from JVM** using Java programming and Runtime class.This is not the best way to know the sizes and in practice it will report less size  that what have you specified in –Xmx and –Xms but still its working solution for most of needs.

Read more: <http://javarevisited.blogspot.com/2012/01/find-max-free-total-memory-in-java.html#ixzz3JzzYQyjR>