**[How Garbage Collection works in Java](http://javarevisited.blogspot.in/2011/04/garbage-collection-in-java.html" \o "How Garbage Collection works in Java)**

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**](http://javarevisited.blogspot.com/2011/05/java-heap-space-memory-size-jvm.html) 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**](http://javarevisited.blogspot.com/2011/05/java-heap-space-memory-size-jvm.html) 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 refrences** 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.