Garbage Collection in Java

*Garbage collection is a mechanism provided by Java Virtual Machine to reclaim heap space from objects which are eligible for Garbage collection.*

## 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 by 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 collection.   
  
**Garbage collection Basic Algorithm**

<http://www.oracle.com/webfolder/technetwork/tutorials/obe/java/gc01/index.html>

The basic process can be described as follows.

**Step 1: Marking**

The first step in the process is called marking. This is where the garbage collector identifies which pieces of memory are in use and which are not.  All objects are scanned in the marking phase to make this determination. This can be a very time consuming process if all objects in a system must be scanned.

#### Step 2: Normal Deletion

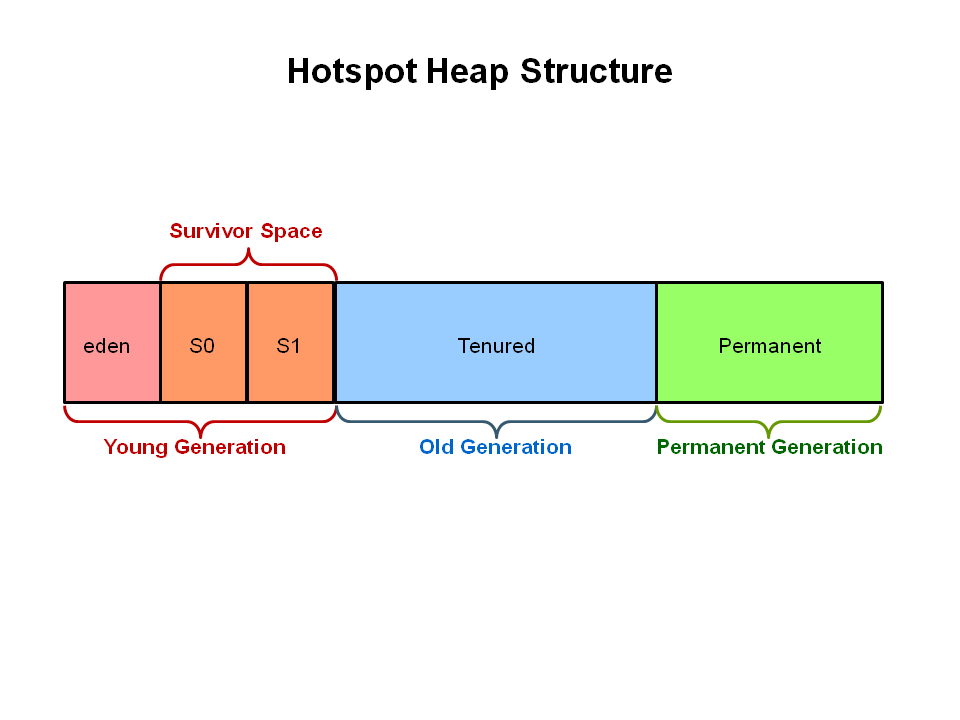
Normal deletion removes unreferenced objects leaving referenced objects and pointers to free space. The memory allocator holds references to blocks of free space where new object can be allocated.

#### Step 2a: Deletion with Compacting

To further improve performance, in addition to deleting unreferenced objects, you can also compact the remaining referenced objects. By moving referenced object together, this makes new memory allocation much easier and faster.

**JVM Generations**

The information learned from the object allocation behavior can be used to enhance the performance of the JVM. Therefore, the heap is broken up into smaller parts or generations. The heap parts are: Young Generation, Old or Tenured Generation, and Permanent Generation



The **Young Generation** is where all new objects are allocated and aged. When the young generation fills up, this causes a **minor garbage collection**. Minor collections can be optimized assuming a high object mortality rate. A young generation full of dead objects is collected very quickly. Some surviving objects are aged and eventually move to the old generation.

**Stop the World Event** - All minor garbage collections are "Stop the World" events. This means that all application threads are stopped until the operation completes. Minor garbage collections are always Stop the World events.

The **Old Generation** is used to store long surviving objects. Typically, a threshold is set for young generation object and when that age is met, the object gets moved to the old generation. Eventually the old generation needs to be collected. This event is called a **major garbage collection**.

Major garbage collection are also Stop the World events. Often a major collection is much slower because it involves all live objects. So for Responsive applications, major garbage collections should be minimized. Also note, that the length of the Stop the World event for a major garbage collection is affected by the kind of garbage collector that is used for the old generation space.

The **Permanent generation** contains metadata required by the JVM to describe the classes and methods used in the application. The permanent generation is populated by the JVM at runtime based on classes in use by the application. In addition, Java SE library classes and methods may be stored here.

Classes may get collected (unloaded) if the JVM finds they are no longer needed and space may be needed for other classes. The permanent generation is included in a full garbage collection.

<http://www.cubrid.org/blog/dev-platform/understanding-java-garbage-collection/>

According to JDK 7, there are 5 GC types.

1. Serial GC
2. Parallel GC
3. Parallel Old GC (Parallel Compacting GC)
4. Concurrent Mark & Sweep GC  (or "CMS")
5. Garbage First (G1) GC

**Serial GC (-XX:+UseSerialGC)**

The serial collector is the default for client style machines in Java SE 5 and 6. With the serial collector, both minor and major garbage collections are done serially (using a single virtual CPU). In addition, it uses a mark-compact collection method. This method moves older memory to the beginning of the heap so that new memory allocations are made into a single continuous chunk of memory at the end of the heap. This compacting of memory makes it faster to allocate new chunks of memory to the heap.

1. The first step of this algorithm is to mark the surviving objects in the old generation.
2. Then, it checks the heap from the front and leaves only the surviving ones behind (sweep).
3. In the last step, it fills up the heap from the front with the objects so that the objects are piled up consecutively, and divides the heap into two parts: one with objects and one without objects (compact).

The serial GC is suitable for a small memory and a small number of CPU cores. The serial collector uses a single thread to perform all garbage collection work, which makes it relatively efficient since there is no communication overhead between threads. It is best-suited to single processor machines, since it cannot take advantage of multiprocessor hardware, although it can be useful on multiprocessors for applications with small data sets (up to approximately 100MB).

#### Usage Cases

The Serial GC is the garbage collector of choice for most applications that do not have low pause time requirements and run on client-style machines. It takes advantage of only a single virtual processor for garbage collection work (therefore, its name).

Another popular use for the Serial GC is in environments where a high number of JVMs are run on the same machine (in some cases, more JVMs than available processors!). In such environments when a JVM does a garbage collection it is better to use only one processor to minimize the interference on the remaining JVMs, even if the garbage collection might last longer.

To enable the Serial Collector use:  
-XX:+UseSerialGC

Here is a sample command line for starting the Java2Demo:  
java -Xmx12m -Xms3m -Xmn1m -XX:PermSize=20m -XX:MaxPermSize=20m -XX:+UseSerialGC -jar c:\javademos\demo\jfc\Java2D\Java2demo.jar

This collector freezes all application threads whenever it’s working, which disqualifies it for all intents and purposes from being used in a server environment.

**Parallel GC (-XX:+UseParallelGC)**

The parallel garbage collector uses multiple threads to perform the young genertion garbage collection. By default on a host with N CPUs, the parallel garbage collector uses N garbage collector threads in the collection. The number of garbage collector threads can be controlled with command-line options:  
-XX:ParallelGCThreads=<desired number>

On a host with a single CPU the default garbage collector is used even if the parallel garbage collector has been requested. On a host with two CPUs the parallel garbage collector generally performs as well as the default garbage collector and a reduction in the young generationgarbage collector pause times can be expected on hosts with more than two CPUs. The Parallel GC comes in two flavors.

**Usage Cases**

The Parallel collector is also called a throughput collector. Since it can use multilple CPUs to speed up application throughput. This collector should be used when a lot of work need to be done and long pauses are acceptable. For example, batch processing like printing reports or bills or performing a large number of database queries.

**-XX:+UseParallelGC**

With this command line option you get a multi-thread young generation collector with a single-threaded old generation collector. The option also does single-threaded compaction of old generation.

Here is a sample command line for starting the Java2Demo:  
java -Xmx12m -Xms3m -Xmn1m -XX:PermSize=20m -XX:MaxPermSize=20m -XX:+UseParallelGC -jar c:\javademos\demo\jfc\Java2D\Java2demo.jar

While the serial GC uses only one thread to process a GC, the parallel GC uses several threads to process a GC, and therefore, faster. This GC is useful when there is enough memory and a large number of cores. It is also called the "**throughput GC**."

The *parallel collector* (also known as the *throughput collector*) performs minor collections in parallel, which can significantly reduce garbage collection overhead. It is intended for applications with medium- to large-sized data sets that are run on multiprocessor or multi-threaded hardware. The parallel collector is selected by default on certain hardware and operating system configurations, or can be explicitly enabled with the option -XX:+UseParallelGC.

The downside to the parallel collector is that it will stop application threads when performing either a minor or full GC collection. The parallel collector is best suited for apps that can tolerate application pauses and are trying to optimize for lower CPU overhead caused by the collector.

**Parallel Old GC(-XX:+UseParallelOldGC)**

With the -XX:+UseParallelOldGC option, the GC is both a multithreaded young generation collector and multithreaded old generation collector. It is also a multithreaded compacting collector. HotSpot does compaction only in the old generation. Young generation in HotSpot is considered a copy collector; therefore, there is no need for compaction.

Compacting describes the act of moving objects in a way that there are no holes between objects. After a garbage collection sweep, there may be holes left between live objects. Compacting moves objects so that there are no remaining holes. It is possible that a garbage collector be a non-compacting collector. Therefore, the difference between a parallel collector and a parallel compacting collector could be the latter compacts the space after a garbage collection sweep. The former would not.

Here is a sample command line for starting the Java2Demo:  
java -Xmx12m -Xms3m -Xmn1m -XX:PermSize=20m -XX:MaxPermSize=20m -XX:+UseParallelOldGC -jar c:\javademos\demo\jfc\Java2D\Java2demo.jar

Parallel Old GC was supported since JDK 5 update. Compared to the parallel GC, the only difference is the GC algorithm for the old generation. It goes through three steps: *mark – summary – compaction*. The summary step identifies the surviving objects separately for the areas that the GC have previously performed, and thus different from the sweep step of the mark-sweep-compact algorithm. It goes through a little more complicated steps.

*parallel compaction* is a feature introduced in J2SE 5.0 update 6 and enhanced in Java SE 6 that allows the parallel collector to perform major collections in parallel. Without parallel compaction, major collections are performed using a single thread, which can significantly limit scalability. Parallel compaction is enabled by adding the option -XX:+UseParallelOldGC to the command line.

The parallel collector (also referred to here as the throughput collector) is a generational collector similar to the serial collector; the primary difference is that multiple threads are used to speed up garbage collection. By default, only minor collections are executed in parallel; major collections are performed with a single thread. However, parallel compaction can be enabled with the option -XX:+UseParallelOldGC so that both minor and major collections are executed in parallel, to further reduce garbage collection overhead. On a machine with N processors the parallel collector uses N garbage collector threads; However, when running applications with medium- to large-sized heaps, it generally outperforms the serial collector by a modest amount on machines with two processors, and usually performs significantly better than the serial collector when more than two processors are available.

**CMS GC (-XX:+UseConcMarkSweepGC) - The Concurrent Mark Sweep (CMS) Collector**

The Concurrent Mark Sweep (CMS) collector (also referred to as the concurrent low pause collector) collects the tenured generation. It attempts to minimize the pauses due to garbage collection by doing most of the garbage collection work concurrently with the application threads. Normally the concurrent low pause collector does not copy or compact the live objects. A garbage collection is done without moving the live objects. If fragmentation becomes a problem, allocate a larger heap.

**Note:** CMS collector on young generation uses the same algorithm as that of the parallel collector.

**Usage Cases**

The CMS collector should be used for applications that require low pause times and can share resources with the garbage collector. Examples include desktop UI application that respond to events, a webserver responding to a request or a database responding to queries.

**Command Line Switches**

To enable the CMS Collector use:  
**-XX:+UseConcMarkSweepGC** and to set the number of threads use**:-XX:ParallelCMSThreads=<n>**

Here is a sample command line for starting the Java2Demo:  
java -Xmx12m -Xms3m -Xmn1m -XX:PermSize=20m -XX:MaxPermSize=20m -XX:+UseConcMarkSweepGC -XX:ParallelCMSThreads=2 -jar c:\javademos\demo\jfc\Java2D\Java2demo.jar

The CMS GC is also called the low latency GC, and is **used when the response time from all applications is crucial**.

**While this GC type has the advantage of short stop-the-world time, it also has the following disadvantages.**

* **It uses more memory and CPU than other GC types.**
* **The compaction step is not provided by default.**

The *concurrent collector* performs most of its work concurrently (i.e., while the application is still running) to keep garbage collection pauses short. It is designed for applications with medium- to large-sized data sets for which response time is more important than overall throughput, since the techniques used to minimize pauses can reduce application performance.

The concurrent collector is designed for applications that prefer shorter garbage collection pauses and that can afford to share processor resources with the garbage collector while the application is running. Typically applications which have a relatively large set of long-lived data (a large tenured generation), and run on machines with two or more processors tend to benefit from the use of this collector. However, this collector should be considered for any application with a low pause time requirement; for example, good results have been observed for interactive applications with tenured generations of a modest size on a single processor, especially if using [incremental mode](http://www.oracle.com/technetwork/java/javase/gc-tuning-6-140523.html#icms). The concurrent collector is enabled with the command line option -XX:+UseConcMarkSweepGC.

Since at least one processor is utilized for garbage collection during the concurrent phases, the concurrent collector does not normally provide any benefit on a uniprocessor (single-core) machine.

**Concurrent Mode failure** : if the concurrent collector is unable to finish reclaiming the unreachable objects before the tenured generation fills up, or if an allocation cannot be satisfied with the available free space blocks in the tenured generation, then the application is paused and the collection is completed with all the application threads stopped. The inability to complete a collection concurrently is referred to as concurrent mode failure and indicates the need to adjust the concurrent collector parameters. The concurrent collector will throw an OutOfMemoryError if too much time is being spent in garbage collection.

The concurrent collector pauses an application twice during a concurrent collection cycle. The first pause is to mark as live the objects directly reachable from the roots (e.g., object references from application thread stacks and registers, static objects and so on) and from elsewhere in the heap (e.g., the young generation). This first pause is referred to as the initial mark pause. The second pause comes at the end of the concurrent tracing phase and finds objects that were missed by the concurrent tracing due to updates by the application threads of references in an object after the concurrent collector had finished tracing that object. This second pause is referred to as **theremark pause**.

The concurrent collection cycle typically includes the following steps:

1. stop all application threads and identify the set of objects reachable from roots, then resume all application threads
2. concurrently trace the reachable object graph, using one or more processors, while the application threads are executing
3. concurrently retrace sections of the object graph that were modified since the tracing in the previous step, using one processor
4. stop all application threads and retrace sections of the roots and object graph that may have been modified since they were last examined, then resume all application threads
5. concurrently sweep up the unreachable objects to the free lists used for allocation, using one processor
6. concurrently resize the heap and prepare the support data structures for the next collection cycle, using one processor

This algorithm will enter “stop the world” (STW) mode in two cases: when initializing the initial marking of roots (objects in the old generation that are reachable from thread entry points or static variables) and when the application has changed the state of the heap while the algorithm was running concurrently, forcing it to go back and do some final touches to make sure it has the right objects marked.

The biggest concern when using this collector is encountering **promotion failures** which are instances where a race condition occurs between collecting the young and old generations. If the collector needs to promote young objects to the old generation, but hasn’t had enough time to make space clear it,  it will have to do so first which will result in a full STW collection – the very thing this CMS collector was meant to prevent. To make sure this doesn’t happen you would either increase the size of the old generation (or the entire heap for that matter) or allocate more background threads to the collector for him to compete with the rate of object allocation.

Another downside to this algorithm in comparison to the parallel collector is that it uses more CPU in order to provide the application with higher levels of continuous throughput, by using multiple threads to perform scanning and collection.

**Garbage First (G1) GC - The G1 Garbage Collector**

The Garbage First or G1 garbage collector is available in Java 7 and is designed to be the long term replacement for the CMS collector. The G1 collector is a parallel, concurrent, and incrementally compacting low-pause garbage collector that has quite a different layout from the other garbage collectors described previously. However, detailed discussion is beyond the scope of this OBE.

#### Command Line Switches

To enable the G1 Collector use:  
-XX:+UseG1GC

Here is a sample command line for starting the Java2Demo:  
java -Xmx12m -Xms3m -XX:+UseG1GC -jar c:\javademos\demo\jfc\Java2D\Java2demo.jar

Garbage First Collector doesn’t work like other collectors and there is no concept of Young and Old generation space. It divides the heap space into multiple equal-sized heap regions. When a garbage collection is invoked, it first collects the region with lesser live data, hence “Garbage First”.

<http://docs.oracle.com/javase/7/docs/technotes/guides/vm/G1.html>

## Technical description

The G1 collector achieves high performance and pause time goals through several techniques.

The heap is partitioned into a set of equal-sized heap regions, each a contiguous range of virtual memory. G1 performs a concurrent global marking phase to determine the liveness of objects throughout the heap. After the mark phase completes, G1 knows which regions are mostly empty. It collects in these regions first, which usually yields a large amount of free space. This is why this method of garbage collection is called Garbage-First. As the name suggests, G1 concentrates its collection and compaction activity on the areas of the heap that are likely to be full of reclaimable objects, that is, garbage. G1 uses a pause prediction model to meet a user-defined pause time target and selects the number of regions to collect based on the specified pause time target.

The regions identified by G1 as ripe for reclamation are garbage collected using evacuation. G1 copies objects from one or more regions of the heap to a single region on the heap, and in the process both compacts and frees up memory. This evacuation is performed in parallel on multi-processors, to decrease pause times and increase throughput. Thus, with each garbage collection, G1 continuously works to reduce fragmentation, working within the user defined pause times. This is beyond the capability of both the previous methods. CMS (Concurrent Mark Sweep ) garbage collection does not do compaction. ParallelOld garbage collection performs only whole-heap compaction, which results in considerable pause times.

It is important to note that G1 is not a real-time collector. It meets the set pause time target with high probability but not absolute certainty. Based on data from previous collections, G1 does an estimate of how many regions can be collected within the user specified target time. Thus, the collector has a reasonably accurate model of the cost of collecting the regions, and it uses this model to determine which and how many regions to collect while staying within the pause time target.

## **Selecting a Collector**

1. If the application has a small data set (up to approximately 100MB), then select the serial collector with -XX:+UseSerialGC.
2. If the application will be run on a single processor and there are no pause time requirements, then
   1. let the VM select the collector, or
   2. select the serial collector with -XX:+UseSerialGC.
3. If (a) peak application performance is the first priority and (b) there are no pause time requirements or pauses of one second or longer are acceptable, then
   1. let the VM select the collector, or
   2. select the parallel collector with -XX:+UseParallelGC and (optionally) enable parallel compaction with -XX:+UseParallelOldGC.
4. If response time is more important than overall throughput and garbage collection pauses must be kept shorter than approximately one second, then select the concurrent collector with -XX:+UseConcMarkSweepGC.

If the recommended collector does not achieve the desired performance, first attempt to adjust the heap and generation sizes to meet the desired goals. If still unsuccessful, then try a different collector: use the concurrent collector to reduce pause times and use the parallel collector to increase overall throughput on multiprocessor hardware.

**Extra and Miscellaneous**

**Generational garbage collection**

Java uses generational garbage collection. This means that if you have an object foo (which is an instance of some class), the more garbage collection events it survives (if there are still references to it), the further it gets promoted. It starts in the young generation (which itself is divided into multiple spaces - Eden and Survivor) and would eventually end up in the tenured generation if it survived long enough. The partition of objects into different *generations* (time intervals) based on time of allocation, and giving them different GC policies depending on age. Based on the heuristic (often true in practice) that *most* objects are discarded shortly after being used--hence the GC is tuned to get rid of those first.

**Write barrier**

Key point of generational GC is what it does need to collect entire heap each time, but just portion of it (e.g. young space). But to achieve this JVM have to implement special machinery called “write barrier”. There 2 types of write barriers implemented in HotSpot: dirty cards and snapshot-at-the-beginning (SATB). SATB write barrier is used in G1 algorithms (which is not covered in this article). All other algorithms are using dirty cards.

***Dirty cards write barrier***

Principle of dirty card write-barrier is very simple. Each time when program modifies reference in memory, it should mark modified memory page as dirty. There is a special card table in JVM and each 512 byte page of memory has associated byte in card table.

**Snapshot at the beginning(SATB)**

In order for the collector to miss a [reachable](http://www.memorymanagement.org/glossary/r.html#term-reachable) [object](http://www.memorymanagement.org/glossary/o.html#term-object), the following two conditions need to hold at some point during tracing:

1. The mutator stores a [reference](http://www.memorymanagement.org/glossary/r.html#term-reference) to a [white](http://www.memorymanagement.org/glossary/w.html#term-white) object into a [black](http://www.memorymanagement.org/glossary/b.html#term-black) object.
2. All paths from any [gray](http://www.memorymanagement.org/glossary/g.html#term-gray) objects to that white object are destroyed.

Snapshot-at-the-beginning algorithms ensure the second condition cannot occur, by causing the collector to process any reference that the mutator overwrites and that might be part of such a path.

**reference :** In memory management, *a reference* is the general term for a link from one [object](http://www.memorymanagement.org/glossary/o.html#term-object) to another.

**white :** In a [tri-color marking](http://www.memorymanagement.org/glossary/t.html#term-tri-color-marking) scheme, white [objects](http://www.memorymanagement.org/glossary/o.html#term-object) are objects that were [condemned](http://www.memorymanagement.org/glossary/c.html#term-condemned-set) at the beginning of the [collection cycle](http://www.memorymanagement.org/glossary/c.html#term-collection-cycle) and have not been shown to be [reachable](http://www.memorymanagement.org/glossary/r.html#term-reachable). When [tracing](http://www.memorymanagement.org/glossary/t.html#term-trace) is complete, white objects will be subject to [reclamation](http://www.memorymanagement.org/glossary/r.html#term-reclaim).

**Unreachable objects are white.**

**tricolour marking**

Tri-color marking is a [tracing garbage collection](http://www.memorymanagement.org/glossary/t.html#term-tracing-garbage-collection) algorithm that assigns a [color](http://www.memorymanagement.org/glossary/c.html#term-color) ([black](http://www.memorymanagement.org/glossary/b.html#term-black), [white](http://www.memorymanagement.org/glossary/w.html#term-white), or [gray](http://www.memorymanagement.org/glossary/g.html#term-gray)) to each [node](http://www.memorymanagement.org/glossary/n.html#term-node) in the [graph](http://www.memorymanagement.org/glossary/g.html#term-graph). It is basic to [incremental garbage collection](http://www.memorymanagement.org/glossary/i.html#term-incremental-garbage-collection).

Initially all nodes are colored white. The distinguished [root set](http://www.memorymanagement.org/glossary/r.html#term-root-set) is colored gray. The [collector(2)](http://www.memorymanagement.org/glossary/c.html#term-collector-2)proceeds to discover the [reachable](http://www.memorymanagement.org/glossary/r.html#term-reachable) nodes by finding an [edge](http://www.memorymanagement.org/glossary/e.html#term-edge) from a gray node to a white node and coloring the white node gray. Hence each tracing step involves choosing a gray node and graying its white children.

When all the edges from a gray node lead only to other gray (or black) nodes, the node is colored black. When no gray nodes remain, the reachable part of the graph has been discovered and any nodes that are still white may be [recycled](http://www.memorymanagement.org/glossary/r.html#term-recycle).

The [mutator](http://www.memorymanagement.org/glossary/m.html" \l "term-mutator) is free to access any part of the graph and allocate new nodes while the[collector(2)](http://www.memorymanagement.org/glossary/c.html#term-collector-2) is determining the reachable nodes, provided the [tri-color invariant](http://www.memorymanagement.org/glossary/t.html#term-tri-color-invariant) is maintained, by changing the colors of the nodes affected, if necessary.

**Black :** In a [tri-color marking](http://www.memorymanagement.org/glossary/t.html#term-tri-color-marking) scheme, black [objects](http://www.memorymanagement.org/glossary/o.html#term-object) are objects that have been [scanned](http://www.memorymanagement.org/glossary/s.html#term-scan). **Scanned and reachable objects are black.**

**Grey :** In a [tri-color marking](http://www.memorymanagement.org/glossary/t.html#term-tri-color-marking) scheme, gray [objects](http://www.memorymanagement.org/glossary/o.html#term-object) are objects that are proved or assumed (see[generational](http://www.memorymanagement.org/glossary/g.html#term-generational-garbage-collection) and [condemn](http://www.memorymanagement.org/glossary/c.html#term-condemned-set)) to be [reachable](http://www.memorymanagement.org/glossary/r.html#term-reachable), but have not yet been [scanned](http://www.memorymanagement.org/glossary/s.html#term-scan). **Root is gray.**

## JVM Parameters for Garbage Collection in Java

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.

Garbage collection occurs in each generation when the generation fills up. The vast majority of objects are allocated in a pool dedicated to young objects (the young generation), and most objects die there. When the young generation fills up it causes a minor collection in which only the young generation is collected;

There are two primary measures of garbage collection performance:

1. Throughput is the percentage of total time not spent in garbage collection, considered over long periods of time. Throughput includes time spent in allocation (but tuning for speed of allocation is generally not needed).
2. Pauses are the times when an application appears unresponsive because garbage collection is occurring.

Footprint is the working set of a process, measured in pages and cache lines. On systems with limited physical memory or many processes, footprint may dictate scalability. Promptness is the time between when an object becomes dead and when the memory becomes available, an important consideration for distributed systems, including remote method invocation (RMI).

**What is Ergonomics ? :** Sun refers to automatic selection of default options based on hardware and OS characteristics as “Ergonomics”. Ergonomics selects the garbage collector dynamically in order to provide good performance on a variety of applications.

**What is throughput ?** Throughput is the percentage of total time not spent in garbage collection, considered over long periods of time. Throughput includes time spent in allocation (but tuning for speed of allocation is generally not needed).

**Permanent Generation**

Permanent Generation or “Perm Gen” contains the application metadata required by the JVM to describe the classes and methods used in the application. Note that Perm Gen is not part of Java Heap memory.

**Stop the World Event**

All the Garbage Collections are “Stop the World” events because all application threads are stopped until the operation completes.

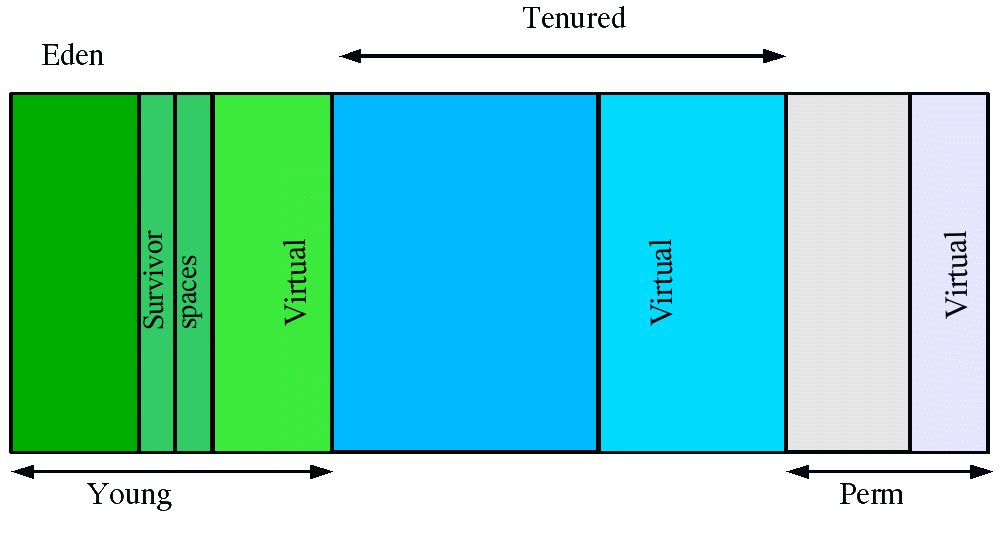
Since Young generation keeps short-lived objects, Minor GC is very fast and the application doesn’t get affected by this.

However Major GC takes longer time because it checks all the live objects. Major GC should be minimized because it will make your application unresponsive for the garbage collection duration. So if you have a responsive application and there are a lot of Major Garbage Collection happening, you will notice timeout errors.

The duration taken by garbage collector depends on the strategy used for garbage collection. That’s why it’s necessary to monitor and tune the garbage collector to avoid timeouts in the highly responsive applications.

**Java Heap**

The default arrangement of generations (for all collectors with the exception of the parallel collector) looks something like this.



At initialization, a maximum address space is virtually reserved but not allocated to physical memory unless it is needed. The complete address space reserved for object memory can be divided into the young and tenured generations.

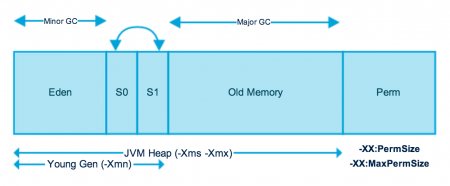
The young generation consists of eden and two survivor spaces. Most objects are initially allocated in eden. One survivor space is empty at any time, and serves as the destination of any live objects in eden and the other survivor space during the next copying collection. Objects are copied between survivor spaces in this way until they are old enough to be tenured (copied to the tenured generation).

A third generation closely related to the tenured generation is the permanent generation which holds data needed by the virtual machine to describe objects that do not have an equivalence at the Java language level.

There are 3 spaces in total, two of which are Survivor spaces. The order of execution process of each space is as below:

1. The majority of newly created objects are located in the Eden space.
2. After one GC in the Eden space, the surviving objects are moved to one of the Survivor spaces.
3. After a GC in the Eden space, the objects are piled up into the Survivor space, where other surviving objects already exist.
4. Once a Survivor space is full, surviving objects are moved to the other Survivor space. Then, the Survivor space that is full will be changed to a state where there is no data at all.
5. The objects that survived these steps that have been repeated a number of times are moved to the old generation.

### Java (JVM) Memory Model

[](http://www.journaldev.com/wp-content/uploads/2014/05/Java-Memory-Model.png)

As you can see in the above image, JVM memory is divided into separate parts. At broad level, JVM Heap memory is physically divided into two parts – **Young Generation** and **Old Generation**.

### Young Generation

Young generation is the place where all the new objects are created. When young generation is filled, garbage collection is performed. This garbage collection is called **Minor GC**. Young Generation is divided into three parts – **Eden Memory** and two **Survivor Memory** spaces.

Important Points about Young Generation Spaces:

* Most of the newly created objects are located in the Eden memory space.
* When Eden space is filled with objects, Minor GC is performed and all the survivor objects are moved to one of the survivor spaces.
* Minor GC also checks the survivor objects and move them to the other survivor space. So at a time, one of the survivor space is always empty.
* Objects that are survived after many cycles of GC, are moved to the Old generation memory space. Usually it’s done by setting a threshold for the age of the young generation objects before they become eligible to promote to Old generation.

### Old Generation

Old Generation memory contains the objects that are long lived and survived after many rounds of Minor GC. Usually garbage collection is performed in Old Generation memory when it’s full. Old Generation Garbage Collection is called **Major GC** and usually takes longer time.

**What is memory leak ?** : A memory leak occurs when object references that are no longer needed are unnecessarily maintained.  They put unnecessary pressure on your machine as your programs consume more and more resources. For starters, think of memory leakage as a disease and Java’s OutOfMemoryError ([OOM](http://docs.oracle.com/javase/7/docs/api/java/lang/OutOfMemoryError.html), for brevity) as a symptom. But as with any disease, **not all OOMs necessarily imply memory leaks**: an OOM can occur due to the generation of a large number of local variables or other such events. On the other hand, **not all memory leaks necessarily manifest themselves as OOMs**, especially in the case of desktop applications or client applications. **Memory leak in Java is a situation where some objects are not used by application any more, but GC fails to recognize them as unused**. As a result, these objects remain in memory indefinitely reducing the amount of memory available to the application.