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

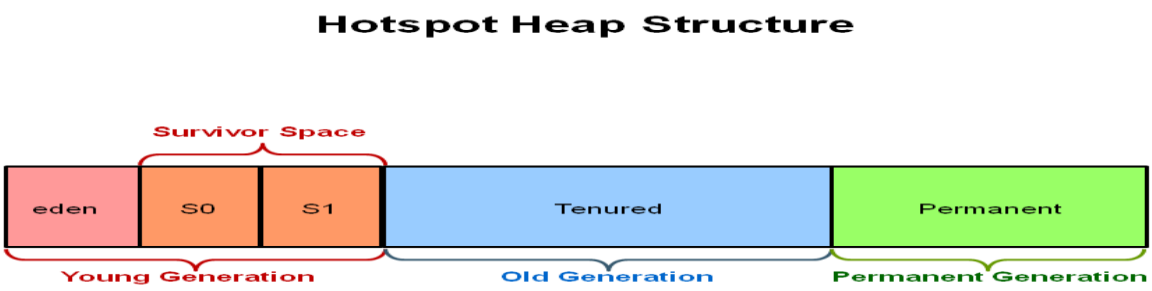
Normal deletion removes unreferenced objects leaving referenced objects and pointers to free space. The memory allocator holds references to blocks of free space where new object can be allocated.

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. 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 **the*remark 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
    a. let the VM select the collector, or
    b. 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
    a. let the VM select the collector, or
    b. 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 object, the following two conditions need to hold at some point during tracing:

1. The mutator stores a reference to a white object into a black object.
2. All paths from any 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 to another.
white : In a tri-color marking scheme, white objects are objects that were condemned at the beginning of the collection cycle and have not been shown to be reachable. When tracing is complete, white objects will be subject to reclamation.
Unreachable objects are white.
tricolour marking
Tri-color marking is a tracing garbage collection algorithm that assigns a color (black, white, or gray) to each node in the graph. It is basic to incremental garbage collection.
Initially all nodes are colored white. The distinguished root set is colored gray. The collector[(2)]proceeds to discover the reachable nodes by finding an 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.

The mutator is free to access any part of the graph and allocate new nodes while thecollector[(2)] is determining the reachable nodes, provided the tri-color invariant is maintained, by changing the colors of the nodes affected, if necessary.

Black : In a tri-color marking scheme, black objects are objects that have been scanned. **Scanned and reachable objects are black.**
Grey : In a tri-color marking scheme, gray objects are objects that are proved or assumed (seegenerational and condemn) to be reachable, but have not yet been scanned. **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

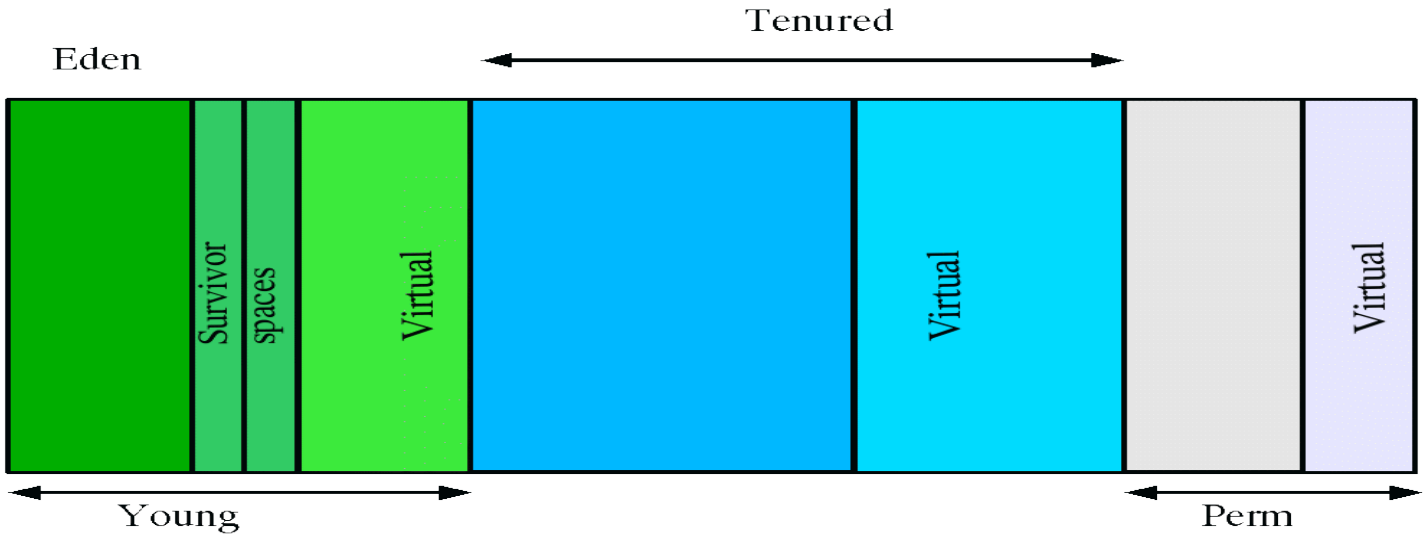
All the Garbage Collections are "Stop the World" events because all application threads are stopped until the operation completes.

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.



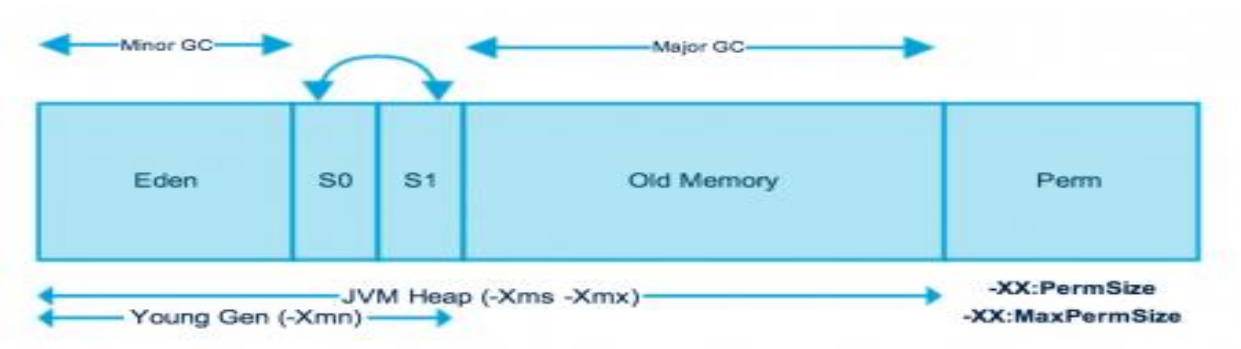
At initialization, a maximum address space is virtually reserved but not allocated to physical memory unless it is needed. The complete address space reserved for object memory can be divided into the young and tenured generations.

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



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, 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.

How Garbage Collection works in Java

http://javarevisited.blogspot.com/2011/04/garbage-collection-in-java.html

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. You can call it a tutorial about garbage collection in simple word, which would be easy to understand and have sufficient information to understand how garbage collection works in Java. Garbage collection works by employing several GC algorithm e.g. Mark and Sweep. There are different kinds of garbage collector available in Java to collect different area of heap memory e.g. you have serial, parallel and concurrent garbage collector in Java. A new collector called G1 (Garbage first) are also introduced in JDK 1.7. First step to learn about GC is to understand when an object becomes eligible to garbage collection? Since `JVM` provides memory management, Java developers only care about creating object, they don't care about cleaning up, that is done by garbage collector, but it can only collect objects which has no live strong reference or it's not reachable from any thread. If an object, which is suppose to be collected but still live in memory due to unintentional strong reference then it's known as memory leak in Java. ThreadLocal variables in Java web application can easily cause memory leak.

Important points about Garbage Collection in Java

This article is in continuation of my previous articles How Classpath works in Java and How to write Equals method in Java 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 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.

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

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 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.

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 weak references via `WeakHashMap` it will be eligible for garbage collection.

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. 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 runtime via JVM 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 PermGen area of Heap, garbage collection can occur in perm space but depends upon JVM to JVM. By the way from JDK 1.7 update, String pool is moved to heap area where objects are created.

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.

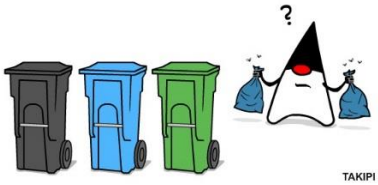
7) JVM command line options `-Xmx` and `-Xms` is used to setup starting and max size for Java Heap. Ideal ratio of this parameter is either 1:1 or 1:1.5 based upon my experience for example you can have either both `-Xmx` and `-Xms` as 1GB or `-Xms` 1.2 GB and 1.8 GB.

8) There is no manual way of doing garbage collection in Java.

That's all about **garbage collection in Java**. In this tutorial we learn how heap is divided into different regions e.g. eden, survivor spaces and perm gen space. An object become eligible to garbage collection when there is no strong reference pointing to it or it is not reachable form any thread. When garbage collector realize need of garbage collection it trigger minor collection and some time stop-the-world major collection. It's all automatic as you cannot force garbage collection in Java.

Garbage Collectors – Serial vs. Parallel vs. CMS vs. G1 (and what's new in Java 8)

http://blog.takipi.com/garbage-collectors-serial-vs-parallel-vs-cms-vs-the-g1-and-whats-new-in-java-8/



The 4 Java Garbage Collectors – How the Wrong Choice Dramatically Impacts Performance

The year is 2014 and there are two things that still remain a mystery to most developers – Garbage collection and understanding the opposite sex. Since I don't know much about the latter, I thought I'd take a whack at the former, especially as this is an area that has seen some major changes and improvements with Java 8, especially with the removal of the PermGen and some new and exciting optimizations (more on this towards the end).

When we speak about garbage collection, the vast majority of us know the concept and employ it in our everyday programming. Even so, there's much about it we don't understand, and that's when things get painful. One of the biggest misconceptions about the JVM is that it has one garbage collector, where in fact it provides **four different ones**, each with its own unique advantages and disadvantages. The choice of which one to use isn't automatic and lies on your shoulders and the differences in throughput and application pauses can be dramatic.

What's common about these four garbage collection algorithms is that they are generational, which means they split the managed heap into different segments, using the age-old assumptions that most objects in the heap are short lived and should be recycled quickly. As this too is a well-covered area, I'm going to jump directly into the different algorithms, along with their pros and their cons.

1. The Serial Collector

The serial collector is the simplest one, and the one you probably won't be using, as it's mainly designed for single-threaded environments (e.g. 32 bit or Windows) and for small heaps. 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.

How to use it: You can use it by turning on the *-XX:+UseSerialGC* JVM argument,

2. The Parallel / Throughput collector

Next off is the Parallel collector. This is the JVM's default collector. Much like its name, its biggest advantage is that is uses multiple threads to scan through and compact the heap. 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.

3. The CMS Collector

Following up on the parallel collector is the CMS collector ("*concurrent-mark-sweep*"). This algorithm uses multiple threads ("concurrent") to scan through the heap ("mark") for unused objects that can be recycled ("sweep"). 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. For most long-running server applications which are adverse to application freezes, that's usually a good trade off to make. Even so, this algorithm is **not on by default**. You have to specify *XX:+USeParNewGC* to actually enable it. If you're willing to allocate more CPU resources to avoid application pauses this is the collector you'll probably want to use, assuming that your heap is less than 4Gb in size. However, if it's greater than 4GB, you'll probably want to use the last algorithm – the G1 Collector.

4. The G1 Collector

The Garbage first collector (G1) introduced in JDK 7 update 4 was designed to better support heaps larger than 4GB. The G1 collector utilizes multiple background threads to scan through the heap that it divides into regions, spanning from 1MB to 32MB (depending on the size of your heap). G1 collector is geared towards scanning those regions that contain the most garbage objects first, giving it its name (Garbage first). This collector is turned on using the *–XX:+UseG1GC*flag.

This strategy the chance of the heap being depleted before background threads have finished scanning for unused objects, in which case the collector will have to stop the application which will result in a

STW collection. The G1 also has another advantage that is that it compacts the heap on-the-go, something the CMS collector only does during full STW collections.

Large heaps have been a fairly contentious area over the past few years with many developers moving away from the single JVM per machine model to more micro-service, componentized architectures with multiple JVMs per machine. This has been driven by many factors including the desire to isolate different application parts, simplifying deployment and avoiding the cost which would usually come with reloading application classes into memory (something which has actually been improved in Java 8).

Even so, one of the biggest drivers to do this when it comes to the JVM stems from the desire to avoid those long “stop the world” pauses (which can take many seconds in a large collection) that occur with large heaps. This has also been accelerated by container technologies like Docker that enable you to deploy multiple apps on the same physical machine with relative ease.

Java 8 and the G1 Collector

Another beautiful optimization which was just out with Java 8 update 20 for is the G1 Collector **String deduplication**. Since strings (and their internal char[] arrays) takes much of our heap, a new optimization has been made that enables the G1 collector to identify strings which are duplicated more than once across your heap and correct them to point into the same internal char[] array, to avoid multiple copies of the same string from residing inefficiently within the heap. You can use the *-XX:+UseStringDeduplication*JVM argument to try this out.

Java 8 and PermGen

One of the biggest changes made in Java 8 was removing the permgen part of the heap that was traditionally allocated for class meta-data, interned strings and static variables. This would traditionally require developers with applications that would load significant amount of classes (something common with apps using enterprise containers) to optimize and tune for this portion of the heap specifically. This has over the years become the source of many OutOfMemory exceptions, so having the JVM (mostly) take care if it is a very nice addition. Even so, that in itself will probably not reduce the tide of developers decoupling their apps into multiple JVMs.

Each of these collectors is configured and tuned differently with a slew of toggles and switches, each with the potential to increase or decrease throughput, all based on the specific behavior of your app. We’ll delve into the key strategies of configuring each of these in our next posts.

In the meanwhile, what are the things you’re most interested in learning about regarding the differences between the different collectors? Hit me up in the comments section

Java is the new C

High performance Java, realtime distributed computing, other stuff.

Due to trouble in a project with long GC pauses, I just had myself a deeper look into GC details. There is not that much acessible information/benchmarks on the Web, so I thought I might share my tests and enlightments ^^. Last time I tested GC some years ago I just came to the conclusion, that allocation of any form is evil in Java and avoided it as far as possible in my code.

I will not describe the exact algorithms here as there is plenty of material regarding this. Understanding the algorithms in detail does not necessary help on predicting actual behaviour in real systems, so I will do some tests and benchmarks.

The concepts of Garbage Collection in Hotspot are explained e.g. here.
In depth coverage of algorithms and parameters can be found here. I will cover GC from an empirical point of view here.

The basic idea of multi generational Garbage Collection is, to collect newer ("younger") Objects more frequent using a "minor" (short duration/pause) Collection algorithm. Objects which survived one or more minor collections then move to the "OldSpace". The OldSpace is garbage collected by the "major" Garbage Collector. I will name them NewSpace and NewGC, OldSpace and OldGC.

The NewGC Algorithm is pretty much the same amongst the 3 Garbage Collectors HotSpot provides. The Old Generation Collector ("OldGC") makes the difference.

- In the **Default Collector**, OldSpace is cleaned up using a "Stop the World" Mark-Sweep.
- **The Concurrent Mark&Sweep** Collector (**CMS**) uses (surprise!) a concurrent implementation of the Mark & Sweep Collection. This means the running application is not stopped during major GC. However it falls back to Stop-The-World GC if it can't keep up with applications allocation (promotion, tenuring) rate.
- **The G1** is also concurrent. It segments memory into equal chunks, trying to split the Garbage Collection Process into smaller pieces by collecting segments which are likely to contain a lot of garbage first. A more detailed description can be found here. It also falls back to Full-Stop-GC in case it can't keep up with application allocation rate. However the duration of those Full-GC pauses are of shorter duration compared to the other OldSpace collectors.

Note that the term "parallel GC" refers to collection algorithms which run multithreaded, not necessary in parallel to your application.

The Benchmark

I wrote a small program which emulates most of the stuff Garbage Collectors have problems with:

- 4GB of statically allocated Objects which are rarely freed.

Problem for GC: with each major GC objects must be traversed, so the larger your reference data, cache's etc., the longer major GC will need for a full traversal collection.

- A lot of temporary Object allocation of various size and age. "Age" refers to the amount of time the Object is referenced by the application.
- Intentional partial replacements of pseudo-static data by new objects. This way I enforce "promotion" of objects to OldSpace, as they are long lived.

This is achieved by putting a lot of objects into a HashMap, then replace a fraction of it. Additionally the latency of a ~0,5 ms operation is measured and memorized to simulate processing of e.g. Requests. This way I get a distribution of VM/GC-related pauses. The benchmark runs for 5 minutes, so long term effects like heap fragmentation are **not** covered by the tests.

Note that this benchmark has an insane allocation rate and object age distribution. So results and VM tuning evaluated in this post illustrate the effects of some GC settings, its not cut & paste stuff, most of the sizings used to get this allocation greedy benchmark to work are way to big for real world applications.

If OldSpace (big green one) is full, the application gets a full-stop GC. In order to avoid Full GC, we need to reduce the promotion rate to OldSpace. An object gets promoted if it is alive (aka referenced) for a longer time than NewSpace (=Eden+Survivor Spaces) holds it.

So time for

Finding #1:

It is key to reduce the promotion rate from young gen to OldSpace. The promotion rate to OldSpace needs to be lowered in order to avoid Full-GC's. In case of concurrent OldSpace GC (CMS,G1), this will enable collectors to keep up with allocation rate.

Tuning the Young Generation

All 3 Collectors avaiable in Java 7 will profit of a proper NewGC setup.

Promotion happens if:
- The "survivor space" (S0, S1) is full.

The size of Survivor vs Eden is specified with the -XX:SurvivorRatio=N. A large Eden will increase throughput (usually) and will catch ultra-short lived temporary Objects better. However large Eden means small Survivor Spaces, so middle aged Objects might get promoted too quickly to OldSpace then, putting load on OldSpace GC.
- Survivors have survived more than `-XX:MaxTenuringThreshold=N (Default 15)` minor collections. Unfortunately I did not find an option to give a lower bound for this value, so one can specify a maximum here only. -XX:InitialTenuringThreshold might help, however I found the VM will choose lower values anyway in case.

The following actions will reduce promotion (by encouraging survivors to live longer in young generation)

- Decreasing -XX:SurvivorRatio=N to lower values than 8 (this actually increases the size of survivor spaces and decreases Eden size).

Effect is that survivors will stay for a longer time in young gen (if there is sufficient size)
This will reduce throughput as survivors are copied with each minor GC between S0,S1.

- Increase the overall size of young generation with -XX:NewRatio=N.

"1" means, young generation will use 50% of your heap, 2 means it will use 33% etc. A larger young gen reduces heap size for long-lived objects but will reduce the number of minor GCs and increase the size avaiable for survivors.

- Increase -XX:MaxTenuringThreshold=N to values > 15.

Of course this only reduces promotion, if the survivor space is large enough. Additonally this is only an upper

bound, so the VM might choose a lower value regardless of your setting (you can also try -XX:InitialTenuringThreshold).

- Increasing overall VM heap will help (in fact more GB always help :-)), as this will increase young generation (Eden+Survivor) and OldSpace size. An increase in OldSpace size reduces the number of required major GC's (or give concurrent OldSpace collectors more headroom to complete a major GC concurrent, pauseless).

It depends on the allocation behaviour of your application which of this actions will have effect.

Finding #2:

When adjusting Survivor Ratio and/or MaxTenuringThreshold manually, always switch off auto adjustment with

`-XX:-UseAdaptiveSizePolicy`

Note: In practice one would evaluate the required size of NewSpace and specify it with `-XX:NewSize=X -XX:MaxNewSize=X` absolutely. This way changing -Xmx will affect OldSpace size only and will not mess up absolute Survivor Space sizes by applying ratios.

Actually the Default GC is the most useful collector to use in order to profile NewSpace setup, since there is no 2cnd background collector bluring OldSpace growth.

Survivor Sizing

The most important thing is, to figure out a good sizing (absolute, not ratio) for the survivor spaces and the promotion counter (MaxTenuringThreshold). Unfortunately there is a strong interaction between Eden size and MaxTenuringThreshold: If Eden is small, then Objects are put into survivor spaces faster, additional the tenuring counter is incremented more quickly. This means if Eden is doubled in size, you probably want to decrease your MaxTenuringThreshold and vice versa. This gets even more complicated as the number of Eden GC (=minor GC) also depends on application allocation rate.

The optimal survivor size is large enough to hold middle lived Objects under full application load without promoting them due to size shortage.

- If survivor space is too large, its just a waste of memory which could be given to Eden instead. Additionally there is a correlation between survivor size and minor GC pauses (throughput degradation, jitter) [Fixme: to be proven].
- If survivor space is too small, Objects will be promoted to OldSpace too early even if TenuringThreshold is not reached yet.

MaxTenuringThreshold

This defines how many minor GC's an Object may survive in SurvivorSpace before getting tenured to OldSpace. Again you have to optimize this under max application load, as without load there are fewer minor GC's so the "survived"-counters will be lower. Another issue to think of is that Eden size also affects the frequency of minor GC's. The VM will handle your value as an upper bound only and will automatically use lower values if it thinks these are more appropriate.

- If MaxTenuringThreshold is too high, throughput might suffer, as non-temporary Objects will be subject to minor collection which slows down application. As said, the VM automatically corrects that.
- If MaxTenuringThreshold is too low, temporary Objects might get promoted to OldSpace too early. This can hamper the OldSpace collector and increase the number of OldSpace GC's. If the promotion rate gets too high, even concurrent Collecors (CMS, G1) will do a full-stop-GC.

If in doubt, set MaxTenuringThreshold too high, this won't have a significant impact on application performance in practice.
It also strongly depends on the coding quality of the application: if you are the kind of guy preferring zero allocation programming, even a MaxTenuringThreshold=0 might be adequate (there is also kind of "alwaysTenure" option). The other extreme is "return new HashBuilder().computeHash(this);"-style (some alternative VM language produce lots of short to mid-lived temporary Garbage) where a settings like '30' or higher (which most often means: keep survivors as long there is room in SurvivorSpace) might be required.
Initially it looks like there are no Objects promoted to OldSpace, as they actually "sit" in Survivor Space. Once the survivor spaces get filled, survivors are tenured to old Space resulting in a sudden increase of promotion rate. (5 minute chart of benchmark running with default GC, actually the application does not allocate more memory, it just constantly replaces small fractions of initially allocated pseudo-static Objects). This will probably confuse concurrent OldSpace Collectors, as they will start concurrent collection too late. Beware: Clever project manager's might bug you to look for memory leaks in your application or to plow through the logs to find out "what happened 14:36 when memory consumption all over a sudden starts to rise".

The promotion rate now reflects the actual allocation rate of long-lived objects. Since there is no concurrent OldSpace Collector in the default GC, it looks like a permanent growth. Once the limit is reached, a Full-GC will be triggered and will clean up unused long lived Objects. A concurrent collector like CMS, G1 will now be able to detect promotion rate and keep up with it.

Eden Size

Eden Size directly correlates with throughput as most java applications will create a lot of temporary objects

Finding #3:

- ***Eden size strongly correlates with throughput/performance for common java applications (with common allocation patterns). The difference can be massive.***
- ***Eden size, Allocation rate, Survivor size and TenuringThreshold are interdependent. If one of those factors is modified, the others need readjustment***
- ***Ratio-based options can be dangerous and lead to false assumptions. NewSpace size should be specified using absolute settings (-XX:NewSize=)***
- ***Wrong sizing can lead to strange allocation and memory consumption patterns***

Tuning OldSpace Garbage Collectors

Default GC

Default GC has a Full-Stop-GC (>15 seconds with the benchmark), so there is not much to do. If you want to run your application with Default GC (e.g. because of high throughput requirements), your only choice is to tune

NewSpace GC very agressively, then throw tons of Heap to your application in order to avoid Full-GC during the day. If your system is 24x7 consider triggering a full GC using System.gc() at night if you expect the system load to be low.
Another possibility would be to even size the VM bigger than your physical RAM, so tenured Objects are written to swap disk. However you have to be sure then no Full-GC is triggered ever, because duration of Full-GC will go into the minutes then. I have not tried this.

Ofc one can improve things always by coding less memory intensive, however this is not the topic of this post.

Concurrent Mark & Sweep (CMS)

The CMS Collector does a pretty good job as long your promotion rate is not too high (which should not be the case if you optimized that as described above).
One Key setting of CMS Collector is, when to trigger a concurrent full GC. If it is triggered too late, it might not be able to finish in time and a Full-Stop-GC will happen. If you know your application has like 30% statically allocated data you might want to set this to 30% like -XX:+UseCMSInitiatingOccupancyOnly
-XX:CMSInitiatingOccupancyFraction=30. In practice I always start with a value of 0, then experiment with higher values once everything (NewSpace, OldSpace) is calibrated to operate without triggering FullGC under load.

When i copy the settings evaluated in the "NewGen Tuning" settings straight forward, the result will be a permanent Full-GC. Why ?
Because CMS requires more heap than the default GC. It seems like the same data structures just require ~20-30% more memory. So we just have to multiply NewGC settings evaluated from Default GC with 1.3.
Additionally, a good start is to let the concurrent Mark & Sweep run all the time.

So I go with (copied 2cnd NewSpace config from above and multiplied):

-XX:+UseConcMarkSweepGC -XX:+UseCMSInitiatingOccupancyOnly
-XX:CMSInitiatingOccupancyFraction=10 -Xmx12g -Xms12g -XX:-UseAdaptiveSizePolicy
-XX:SurvivorRatio=3 -XX:NewSize=4173m -XX:MaxNewSize=4173m
-XX:MaxTenuringThreshold=15

The CMS is barely able to keep up with promotion rate, throughput is 300.000 which is acceptable given that cost of (in contradiction to tests above) Full-GC is included.

We can see from Visual GC (an excellent jVisualVM plugin), that OldSpace size is on the edge of triggering a full GC. In order to improve throughput, we would like to increase eden. Unfortunately there is not enough headroom in OldSpace, as a reduction of OldSpace would trigger Full-Stop-GC's.
Reducing the size of Surviver Spaces is not an option, as this would result in a higher tenured Object rate and again trigger Full GC's. The only solution is: More Memory.
Comparing throughput with the Default GC test above is not fair, as the Default GC would run into Full-Stop-GC's for sure, if the test would run for a longer time than 5 minutes.
On a side note: the memory chart of jVisual VM's (and printouts by -verbose:gc) does not tell you the full story as you cannot see the fraction of used memory in OldSpace.

Ok, so lets add 2 more Gb to be able to increase eden resulting in

```
-XX:+UseConcMarkSweepGC -XX:+UseCMSInitiatingOccupancyOnly
-XX:CMSInitiatingOccupancyFraction=10 -Xmx14g -Xms14g -XX:-UseAdaptiveSizePolicy
```

```
-XX:SurvivorRatio=4 -XX:NewSize=5004m -XX:MaxNewSize=5004m
-XX:MaxTenuringThreshold=15
```

Conclusion

Disclaimer:

- The benchmark used is worst case ever regarding allocation rate and memory waste. So take any finding with a grain of salt, when applying GC optimizations to your program.
- I did not drive long term tests. All tests ran for 5 minutes only. Due to the extreme allocation rate of the benchmark, 5 minute benchmark is likely aequivalent to an hour operation of a "real" program. Anyway in an application with lower allocation rate, concurrent collectors will have more time to complete GC's concurrent, so you probably never will need an eden size of 4Gb in practice :).
I will provide long term runs in a separate post (maybe :)).

Default GC (Serial Mark&Sweep, Serial NewGC) shows highest throughput as long no Full GC is triggered. If your system has to run for a limited amount of time (like 12 hours) and you are willing to invest into a very careful programming style regarding allocation; keep large datasets Off-Heap, DefaultGC can be the best choice. Of course there are applications which are ok with some long GC pauses here and there.

CMS does best in pause-free low latency operation as long you are willing to throw memory at it. Throughput is pretty good. Unfortunately it does not compact the heap, so fragmentation can be an issue over time. This is not covered here as it would require to run real application tests with many different Object sizes for several days. CMS is still way behind commercial low-latency solutions such as Azul's Zing VM.

G1 excels in robustness, memory efficiency with acceptable throughput. While CMS and DefaultGC react to OldSpace overflow with Full-Stop-GC of several seconds up to minutes (depends on Heap size and Object graph complexity), G1 is more robust in handling those situations. Taking into account the benchmark represents a worst case scenario in allocation rate and programming style, the results are encouraging.

Types of Garbage Collectors in Java

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

Java Garbage Collectors

You now know the basics of garbage collection and have observed the garbage collector in action on a sample application. In this section, you will learn about the garbage collectors available for Java and the command line switches you need to select them.

Common Heap Related Switches

There are many different command line switches that can be used with Java. This section describes some of the more commonly used switches that are also used in this OBE.

Switch	Description
`-Xms`	Sets the initial heap size for when the JVM starts.
`-Xmx`	Sets the maximum heap size.
`-Xmn`	Sets the size of the Young Generation.
-XX:PermSize	Sets the starting size of the Permanent Generation.
-XX:MaxPermSize	Sets the maximum size of the Permanent Generation

The Serial GC

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.

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). Still, on today's hardware, the Serial GC can efficiently manage a lot of non-trivial applications with a few hundred MBs of Java heap, with relatively short worst-case pauses (around a couple of seconds for full garbage collections).

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. And the Serial GC fits this trade-off nicely.

Finally, with the proliferation of embedded hardware with minimal memory and few cores, the Serial GC could make a comeback.

Command Line Switches

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
```

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
```

`-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
```


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 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`:

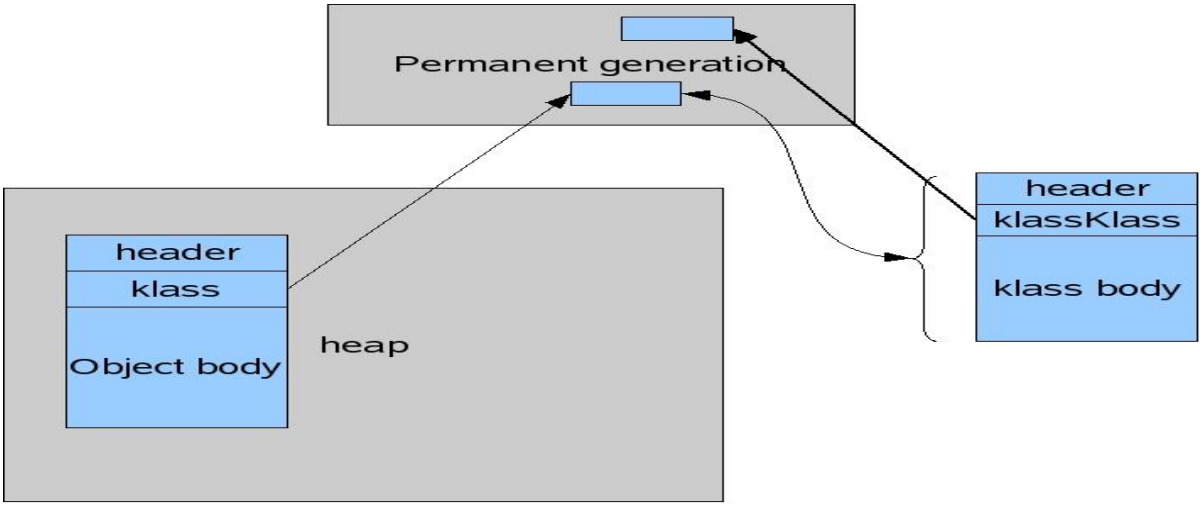
```
java -Xmx12m -Xms3m -XX:+UseG1GC -jar c:\javademos\demo\jfc\Java2D\Java2demo.jar
```

Presenting the Permanent Generation

https://blogs.oracle.com/jonthecollector/entry/presenting_the_permanent_generation

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

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



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

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

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

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

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

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

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

- Methods of a class (including the bytecodes)
- Names of the classes (in the form of an object that points to a string also in the permanent generation)
- Constant pool information (data read from the class file, see chapter 4 of the JVM specification for all the details).

- Object arrays and type arrays associated with a class (e.g., an object array containing references to methods).
- Internal objects created by the JVM (java/lang/Object or java/lang/exception for instance)
- Information used for optimization by the compilers (JITs)

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

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

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

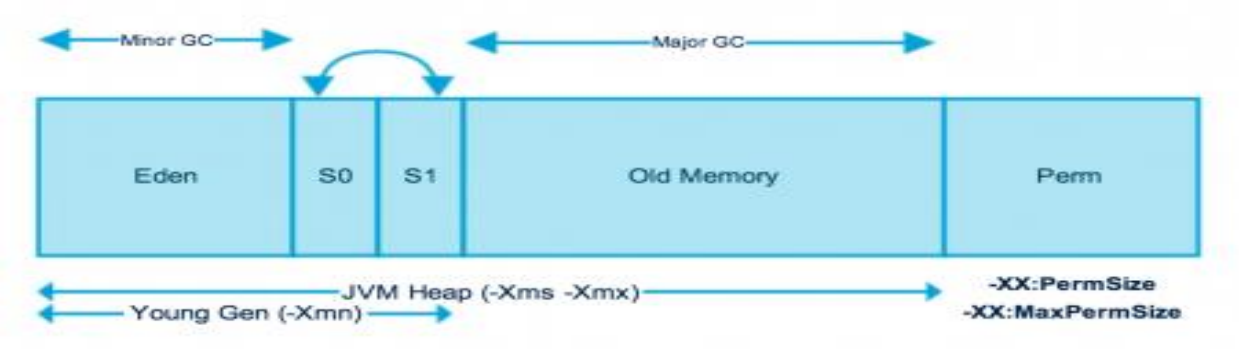
And why not use the same knowledge about allocation order to eliminate the permanent generations even in the parallel old collector case? The parallel old collector does maintain allocation order of objects, but objects are moved in parallel. When the collection gets to X, we no longer know if K has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

Java (JVM) Memory Model and Garbage Collection

http://www.journaldev.com/2856/java-jvm-memory-model-and-garbage-collection-monitoring-tuning

Understanding **JVM Memory Model** is very important if you want to understand the working of **Java Garbage Collection**. Today we will look into different parts of JVM memory and how to monitor and perform garbage collection tuning.

Java (JVM) Memory Model



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.

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.

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.

Perm Gen is populated by JVM at runtime based on the classes used by the application. Perm Gen also contains Java SE library classes and methods. Perm Gen objects are garbage collected in a full garbage collection.

Method Area

Method Area is part of space in the Perm Gen and used to store class structure (runtime constants and static variables) and code for methods and constructors.

Memory Pool

Memory Pools are created by JVM memory managers to create a pool of immutable objects, if implementation supports it. String Pool is a good example of this kind of memory pool. Memory Pool can belong to Heap or Perm Gen, depending on the JVM memory manager implementation.

Runtime Constant Pool

Runtime constant pool is per-class runtime representation of constant pool in a class. It contains class runtime constants and static methods. Runtime constant pool is the part of method area.

Java Stack Memory

Java Stack memory is used for execution of a thread. They contain method specific values that are short-lived and references to other objects in the heap that are getting referred from the method. You should read Difference between Stack and Heap Memory.

Java Heap Memory Switches

Java provides a lot of memory switches that we can use to set the memory sizes and their ratios. Some of the commonly used memory switches are:

VM Switch	VM Switch Description
-Xms	For setting the initial heap size when JVM starts
-Xmx	For setting the maximum heap size.
-Xmn	For setting the size of the Young Generation, rest of the space goes for Old Generation.
-XX:PermGen	For setting the initial size of the Permanent Generation memory
-XX:MaxPermGen	For setting the maximum size of Perm Gen
-XX:SurvivorRatio	For providing ratio of Eden space and Survivor Space, for example if Young Generation size is 10m and VM switch is -XX:SurvivorRatio=2 then 5m will be reserved for Eden Space and 2.5m each for both the Survivor spaces. The default value is 8.
-XX:NewRatio	For providing ratio of old/new generation sizes. The default value is 2.

Most of the times, above options are sufficient, but if you want to check out other options too then please check JVM Options Official Page.

Java Garbage Collection

Java Garbage Collection is the process to identify and remove the unused objects from the memory and free space to be allocated to objects created in the future processing. One of the best feature of java programming language is the **automatic garbage collection**, unlike other programming languages such as C where memory allocation and deallocation is a manual process.

Garbage Collector is the program running in the background that looks into all the objects in the memory and find out objects that are not referenced by any part of the program. All these unreferenced objects are deleted and space is reclaimed for allocation to other objects.

One of the basic way of garbage collection involves three steps:

1. **Marking**: This is the first step where garbage collector identifies which objects are in use and which ones are not in use.
2. **Normal Deletion**: Garbage Collector removes the unused objects and reclaim the free space to be allocated to other objects.

3. **Deletion with Compacting**: For better performance, after deleting unused objects, all the survived objects can be moved to be together. This will increase the performance of allocation of memory to newer objects.

There are two problems with simple mark and delete approach.

1. First one is that it's not efficient because most of the newly created objects will become unused
2. Secondly objects that are in-use for multiple garbage collection cycle are most likely to be in-use for future cycles too.

The above shortcomings with the simple approach is the reason that **Java Garbage Collection is Generational** and we have **Young Generation** and **Old Generation** spaces in the heap memory. I have already explained above how objects are scanned and moved from one generational space to another based on the Minor GC and Major GC.

Java Garbage Collection Types

There are five types of garbage collection types that we can use in our applications. We just need to use JVM switch to enable the garbage collection strategy for the application. Let's look at each of them one by one.

1. **Serial GC (-XX:+UseSerialGC)**: Serial GC uses the simple **mark-sweep-compact** approach for young and old generations garbage collection i.e Minor and Major GC.
   Serial GC is useful in client-machines such as our simple stand alone applications and machines with smaller CPU. It is good for small applications with low memory footprint.

2. **Parallel GC (-XX:+UseParallelGC)**: Parallel GC is same as Serial GC except that is spawns N threads for young generation garbage collection where N is the number of CPU cores in the system. We can control the number of threads using `-XX:ParallelGCThreads=n` JVM option.
   Parallel Garbage Collector is also called throughput collector because it uses multiple CPUs to speed up the GC performance. Parallel GC uses single thread for Old Generation garbage collection.

3. **Parallel Old GC (-XX:+UseParallelOldGC)**: This is same as Parallel GC except that it uses multiple threads for both Young Generation and Old Generation garbage collection.
4. **Concurrent Mark Sweep (CMS) Collector (-XX:+UseConcMarkSweepGC)**: CMS Collector is also referred as concurrent low pause collector. It does the garbage collection for Old generation. CMS collector tries to minimize the pauses due to garbage collection by doing most of the garbage collection work concurrently with the application threads.
   CMS collector on young generation uses the same algorithm as that of the parallel collector. This garbage collector is suitable for responsive applications where we can't afford longer pause times. We can limit the number of threads in CMS collector using `-XX:ParallelCMSThreads=n` JVM option.
5. **G1 Garbage Collector (-XX:+UseG1GC)**: The Garbage First or G1 garbage collector is available from Java 7 and it's long term goal is to replace the CMS collector. The G1 collector is a parallel, concurrent, and incrementally compacting low-pause garbage collector.
   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”. You can find more details about it at Garbage-First Collector Oracle Documentation.

Java Garbage Collection Monitoring

We can use Java command line as well as UI tools for monitoring garbage collection activities of an application. For my example, I am using one of the demo application provided by Java SE downloads.

If you want to use the same application, go to Java SE Downloads page and download **JDK 7 and JavaFX Demos and Samples**. The sample application I am using is **Java2Demo.jar** and it’s present in`jdk1.7.0_55/demo/jfc/Java2D` directory. However this is an optional step and you can run the GC monitoring commands for any java application.

FINALIZE METHOD IN JAVA - WHY NOT TO USE, LIMITED USE CASES AND ALTERNATIVES

http://javajee.com/finalize-method-in-java-why-not-to-use-limited-use-cases-and-alternatives

Why using finalize is not a good idea

Finalizers are bad for functional reasons as well as for performance reasons. We will first see functional reasons and then the performance reasons.

Functional reasons not to use finalize

You should never depend on a finalizer to update critical data or do anything time-critical in a finalizer, or assume that finalize will work the same on another JVM, because:

1. You can resurrect your object from a finalize method (creating a new strong reference to the referent object), but Java won't call finalize again when the object is freed again, and the expected cleanup of the referent won't happen.
2. Java language specification does not guarantee that finalize will always be executed; specification only guarantees that finalize() will not be called twice. Even System.gc and System.runFinalization (though may increase the chances of finalizers getting executed,) don't guarantee it either.
3. There is also no guarantee finalizers will be executed promptly; even after an object becomes eligible for garbage collection, it can take any time before finalizer is executed.
4. When to execute finalizer may also dependent on the GC algorithm in use in a particular JVM, thus also affecting portability.
5. If a runtime exception is thrown within the finalize method, finalization of that object terminates and the exception is ignored. Therefore there is a change for some activities to be missed due to an exception.

Performance impact of finalize

Finalizers also has great impact on performance. The JVM uses a special private reference object class java.lang.ref.Finalizer to keep track of objects that have defined a finalize() method. The java.lang.ref.Finalizer in turn is a java.lang.ref.FinalReference. This is a special reference similar to other public reference object classes available in the java.lang.ref package. You can learn more about reference objects @ http://javajee.com/soft-and-week-reference-object-classes-in-java and for further reference you can refer to Oracle documentation for java.lang.ref package @ http://docs.oracle.com/javase/7/docs/api/java/lang/ref/package-summary.html.

When an object that has a finalize() method is allocated, the JVM allocates two objects: the object itself, and a Finalizer reference that also refer to this object. As with other less strong references, it takes at least two GC cycles before this less strong reference object can be freed. However, the performance penalty here is greater than with other less strong reference types. When the referent of a soft or weak reference is eligible for GC, the referent object is freed first, and the soft or weak reference is placed on the reference queue and is freed in next GC cycle; the two-cycle penalty for GC applies only to the reference object itself (and not the referent). However in case of finalizers, referent object is not freed immediately when the finalizer reference is placed on its reference queue, as the implementation of the Finalizer class must have access to the referent in order to call the referent's finalize() method. Only when the reference queue processes the finalizer, the Finalizer object will be removed from the queue and then eligible for collection. Unlike other less strong references, here the memory of the referent object is also retained, which can be usually much bigger than the reference object.

Unavoidable cases

However in some cases, finalize might be unavoidable. For instance, if you have some native resource to be freed, then you can have an explicit method to free that resource, but also can have finalize as a backup plan in case the developer forget to call that explicit method. For instance, classes that handle zip files in JDK uses this technique as opening a ZIP file uses some native code that allocates native memory and it needs to be closed even if developer forget to call close. Other JDK classes such as FileInputStream, FileOutputStream, Timer, Connection etc. also have finalizers. The book Effective Java suggests that the finalizer should log a warning in these cases if it finds that the resource has not been terminated, so that the client code can be fixed (however JDK classes don't do that). You should also make sure that the memory accessed by the object is kept to a minimum in such cases when it is unavoidable to use finalizers.

Finalizer Guardian idiom

Unlike constructors, finalize won't automatically call its parent. So you need to call the super finalize explicitly (like any other method), and there is a change that you may miss that. The section 'Item 7: Avoid finalizers' in Effective Java talks about a solution to this problem: Put the finalizer on an anonymous inner class (called a finalizer guardian), and a single instance of the anonymous class is stored in a private instance field so the finalizer guardian becomes eligible for finalization at the same time as the enclosing instance. When the guardian is finalized, it performs the finalization activity desired for the enclosing instance, just as if its finalizer were a method on the enclosing class.

Alternatives to finalize

You can overcome some of the disadvantages of finalizers by using PhantomReference class (along with ReferenceQueue) rather than implicitly using a Finalizer reference. This can overcomes two limitations of the traditional finalizer: the memory associated with the referent object is released as soon as the referent is collected (rather than doing that in the finalizer() method and giving a chance to resurrect). There is also no way for the referent object to be resurrected in the cleanup code, since it has already been collected. Other limitations like no guarantee of timing, no guarantee that it will be actually cleared etc. can happen here also.

I will not explain the approach here, but just wanted to tell that there is one such approach possible. You can search online or refer to a book that explains the approach. I recommend you look at the book 'Java Performance: The Definitive Guide' as that is my primary reference book for learning java performance.

Clearing and monitoring the Finalizer Queue

The finalizer queue is the reference queue used to process the Finalizer references when the referent is eligible for GC.

You can cause the finalizer queue to be processed by executing GC.run_finalization of jcmd command:

jcmd <process_id> GC.run_finalization

You can monitor the finalizer queue using jmap as:

jmap -finalizerinfo <process_id>

You can also monitor the finalizer queue using jconsole.

Replacing Finalizers With Phantom References

http://resources.ej-technologies.com/jprofiler/help/doc/index.html

Why finalizers are bad

Sometimes one must perform pre-garbage collection actions such as freeing resources. In a JDBC driver, for example, a database connection may be held by a connection object. Before the connection object is garbage collected, the actual database connection must be closed. In such a case, one typically cannot rely on the `close()` method being called by the user application code.

Most often, **finalizers** are used to solve this problem. A finalizer is created by overriding the `finalize()` method of `java.lang.Object`. In that case, before the object is garbage collected, this finalize method will be called. Unfortunately, there are severe problems with the design of this finalizer mechanism. Using finalizers has a negative impact on the performance of the garbage collector and can break data integrity of your application if you're not very careful since the "finalizer" is invoked in a random thread, at a random time. If you use a lot of finalizers, the finalizer system may be completely overwhelmed which can lead to `OutOfMemoryError`s. In addition, you have no control about when a finalizer will be run, so it can create problems with locking, the shutdown of the JVM and other exceptional circumstances.

Because the random execution of the finalizers break the call tree, JProfiler eliminates them from the profiling results.

The solution for all these problems is to **eliminate finalizers** where they are not strictly required and **replace the necessary ones with phantom references**.

What are phantom references?

Phantom references can be used to perform actions before an object is garbage collected in a safe way. In the constructor of a `java.lang.ref.PhantomReference`, you specify a `java.lang.ref.ReferenceQueue` where the phantom reference will be enqueued once the referenced object becomes "phantom reachable". Phantom reachable means unreachable other than through the phantom reference. The initially confusing thing is that although the phantom reference continues to hold the referenced object in a private field (unlike soft or weak references), its `getReference()` method always returns `null`. This is so that you cannot make the object strongly reachable again.

From time to time, you can poll the reference queue and check if there are any new phantom references whose referenced objects have become phantom reachable. In order to be able to do anything useful, one can for example derive a class from `java.lang.ref.PhantomReference` that references resources that should be freed before garbage collection. The referenced object is only garbage collected once the phantom reference becomes unreachable itself.

How to replace finalizers with phantom references

Let's continue with the example of the JDBC driver above: Before a connection object is garbage collected, the actual database connection must be closed. The following steps are necessary to achieve this with phantom references:

- **Add data structure that holds phantom references**
  The JDBC driver class gets a data structure that holds phantom references to the connection objects. A private field

  ```
  private LinkedList phantomReferences = new LinkedList();
  ```

  would be appropriate. This is necessary to ensure that phantom references are not garbage collected as long as they have not been handled by the reference queue.

- **Create reference queue**
  Before a connection object will be garbage collected, its phantom reference will be enqueued into the associated reference queue. The JDBC driver thus gets an additional private field

  ```
  private ReferenceQueue queue = new ReferenceQueue();
  ```

- **Derive a class from PhantomReference that references resources**
  You will not be able to access the original object from a phantom reference. Therefore, you have to add the resources that must be freed to the phantom reference itself. In our example JDBC driver this could be a class named `DatabaseConnection`. The phantom reference class will thus look like:

```
public class ConnectionPhantomReference extends PhantomReference {
    private DatabaseConnection databaseConnection;

    public MyPhantomReference(ConnectionImpl connection, ReferenceQueue queue) {
        super(connection, queue);
        databaseConnection = connection.getDatabaseConnection();
    }

    public void cleanup() {
        databaseConnection.close();
    }
}
```

  The custom phantom reference extracts the resource object from the implementation class of the connection and saves it in a private field. It additionally provides a `cleanup()` method that can be invoked once after the phantom reference is taken out of the reference queue.

- **Create and remember phantom references when objects are created**
  When a connection object is created, a corresponding `ConnectionPhantomReference` must be created as well and added to the `phantomReferences` list:

```
phantomReferences.add(new ConnectionPhantomReference(connection, queue));
```

- **Create reference queue handler thread**
  When a phantom reference is added to the queue by the garbage collector, no further action is taken. You have to handle and empty the reference queue yourself. It's best to create a separate daemon thread that removes phantom references from the queue and invokes the cleanup method:

```
Thread referenceThread = new Thread() {
    public void run() {
        while (true) {
            try {
                ConnectionPhantomReference ref =
(ConnectionPhantomReference)queue.remove();
                ref.close();
                phantomReferences.remove(ref);
            } catch (Exception ex) {
              // log exception, continue
            }
        }
    }
};
referenceThread.setDaemon(true);
```

```
referenceThread.start();
```

  The phantom reference is removed from the `phantomReferences` list. Now the phantom reference is unreferenced itself and the referenced object can be garbage collected.

10 Garbage Collection Interview Questions and Answers in Java Programming

http://javarevisited.blogspot.in/2012/10/10-garbage-collection-interview-question-answer.html

GC Interview Questions Answer in Java

Garbage collection interview questions are very popular in both core Java and advanced Java Interviews. Apart from Java Collection and Thread many tricky Java questions stems Garbage collections which are tough to answer. In this Java Interview article I will share some questions from GC which is asked in various core Java interviews.These questions are based upon concept of How Garbage collection works, Different kinds of Garbage collector and JVM parameters used for garbage collection monitoring and tuning. As I said GC is an important part of any Java interview so make sure you have good command in GC. One more thing which is getting very important is ability to comprehend and **understand Garbage collection Output**, more and more interviewer are checking whether candidate can understand GC output or not. During Java interview they may provide a snippet of GC output and ask various questions based on that e.g. Which Garbage collector is used, whether output is from major collection or minor collection, How much memory is free from GC, What is size of new generation and old generation after GC etc. I have included few Garbage collection interview questions and answers from GC output to help with that. It's recommended to prepare questions from Java collection, multithreading and programmingalong with Garbage collection to do well in Java interviews at any stage.

Interview questions on Java Garbage collection

 Here is some Garbage collection Interview questions from my personal collection, which I have created from my experience and with the help of various friends and colleagues which has shared *GC interview questions* with me. Actually there are lot many questions than What I am sharing here but to keep this post small I thought to only share some questions, I can think of second part of GC interview question if you guys find this useful.

Question 1 - What is structure of Java Heap ? What is Perm Gen space in Heap ?

Answer : In order to better perform in Garbage collection questions in any Java interview, It's important to have basic understanding of Java Heap space. To learn more about heap, see my post 10 points on Java heap space. By the way Heap is divided into different generation e.g. new generation, old generation and `PermGen` space.PermGen space is used to store class's metadata and filling of PermGen space can causejava.lang.OutOfMemory:PermGen space. Its also worth noting to remember JVM option to configure PermGen space in Java.

Question 2 - How do you identify minor and major garbage collection in Java?

Answer: Minor collection prints `"GC"` if garbage collection logging is enable using `-verbose:gc` or `-XX:PrintGCDetails`, while Major collection prints `"Full GC"`. This Garbage collection interview question is based on understanding of Garbage collection output. As more and more Interviewer are asking question to check candidate's ability to understand GC output, this topic become even more important.

Question 3 - What is difference between `ParNew` and `DefNew` Young Generation Garbage collector?

Answer : This *Garbage Collection interview questions* is recently asked to one of my friend. It require more than average knowledge on GC to answer this question. By the way `ParNew` and `DefNew` is two young generation garbage collector. `ParNew` is a multi-threaded GC used along with concurrent Mark Sweep while `DefNew` is single threaded GC used along with Serial Garbage Collector.

Question 4 - How do you find GC resulted due to calling `System.gc()`?

Answer : Another GC interview question which is based on GC output. Similar to major and minor collection, there will be a word `"System"` included in Garbage collection output.

Question 5 - What is difference between Serial and Throughput Garbage collector?

Answer : Serial Garbage collector is a stop the world GC which stops application thread from running during both minor and major collection. Serial Garbage collector can be enabled using JVM option `-XX:UseSerialGC` and it's designed for Java application which doesn't have pause time requirement and have client configuration. **Serial Garbage collector** was also default GC in JDK 1.4 before ergonomics was introduced in JDK 1.5. Serial GC is most suited for small application with less number of thread while throughput GG is more suited for large applications. On the other hand Throughput garbage collector is parallel collector where minor and major collection happens in parallel taking full advantage of all the system resources available like multiple processor. Though both major and minor collection runs on stop-the-world fashion and introduced pause in application. Throughput Garbage collector can be enable using `-XX:UseParallelGC` or `-XX:UseOldParallelGC`. It increases overall throughput of application my minimizing time spent in Garbage

collection but still has long pauses during full GC.This is a kind of *Garbage collection interview questions* which gives you an opportunity to show your knowledge in detail while answering. I always suggest to answer these kind of questions in detail.

Question 6 – When does an Object becomes eligible for Garbage collection in Java ?

Answer : An object becomes eligible for garbage collection when there is no live reference for that object or it can not be reached by any live thread. Cyclic reference doesn't count as live reference and if two objects are pointing to each other and there is no live reference for any of them, than both are eligible for GC. Also Garbage collection thread is a daemon thread which will run by JVM based upon GC algorithm and when runs it collects all objects which are eligible for GC.

Question 7 - What is finalize method in Java ? When does Garbage collector calls finalize method in Java ?

Answer : Finalize method in Java also called finalizer is a method defined in `java.lang.Object` and called by Garbage collector before collecting any object which is eligible for GC. `Finalize()` method provides last chance to object to do cleanup and free any remaining resource, to learn more about finalizers, read What is finalize method in Java.

Question 8 - If Object A has reference to Object B and Object B refer to Object A, apart from that there is no live reference to either object A or B, Does they are eligible to Garbage collection ?

This Garbage collection interview questions is related question 5 "When object become eligible for Garbage collection". An object becomes eligible for Garbage collection if there is no live reference for it. It can not be accessible from any Thread and cyclic dependency doesn't prevent Object from being Garbage collected. Which means in this case both Object A and Object B are eligible of Garbage collection. See How Garbage collection works in Java for more details.

Question 9 -Can we force Garbage collector to run at any time ?

Answer : No, you can not force Garbage collection in Java. Though you can request it by calling `Sytem.gc()` or its cousin `Runtime.getRuntime().gc()`. It's not guaranteed that GC will run immediately as result of calling these method.

Question 10 - Does Garbage collection occur in permanent generation space in JVM?

Answer : This is a tricky Garbage collection interview question as many programmers are not sure whether `PermGen` space is part of Java heap space or not and since it maintains class Meta data and String pool, whether its eligible for garbage collection or not. By the way Garbage Collection does occur in `PermGen` space and if PermGen space is full or cross a threshold, it can trigger Full GC. If you look at output of GC you will find that PermGen space is also garbage collected. This is why correct sizing of PermGen space is important to avoid frequent full GC. You can control size of `PermGen` space by JVM options `-XX:PermGenSize` and `-XX:MaxPermGenSize`.

Question 11 : How to you monitor garbage collection activities?

Answer : One of my favorite interview questions on Garbage collection, just to check whether candidate has ever monitored GC activities or not. You can monitor garbage collection activities either offline or real-time. You can use tools like **JConsole** and **VisualVM** VM with its Visual GC plug-in to monitor real time garbage collection activities and memory status of JVM or you can redirect Garbage collection output to a log file for offline analysis by using `-XlogGC=<PATH>` JVM parameter. Anyway you should always enable GC options like `-XX:PrintGCDetails -X:verboseGC` and `-XX:PrintGCTimeStamps` as it doesn't impact application performance much but provide useful states for performance monitoring.

Question 12: Look at below Garbage collection output and answer following question :

```
[GC
        [ParNew: 1512K->64K(1512K), 0.0635032 secs]
        15604K->13569K(600345K), 0.0636056 secs]
        [Times: user=0.03 sys=0.00, real=0.06 secs]
```

1. Is this output of Major Collection or Minor Collection ?
2. Which young Generation Garbage collector is used ?

3. What is size of Young Generation, Old Generation and total Heap Size?
4. How much memory is freed from Garbage collection ?
5. How much time is taken for Garbage collection ?
6. What is current Occupancy of Young Generation ?

This Garbage collection Interview questions is completely based on GC output. Following are answers of above GC questions which will not only help you to answer these question but also help you to understand and interpret GC output.

Answer 1: It's Minor collection because of `"GC"` word, In case of Major collection, you would see `"Full GC"`.

Answer 2: This output is of multi-threaded Young Generation Garbage collector `"ParNew"`, which is used along with CMS concurrent Garbage collector.

Answer 3: [1512K] which is written in bracket is total size of Young Generation, which include Eden and two survivor space. 1512K on left of arrow is occupancy of Yong Generation before GC and 64K is occupancy after GC. On the next line value if bracket is total heap size which is `(600345K)`. If we subtract size of young generation to total heap size we can calculate size of Old Generation. This line also shows occupancy of heap before and after Garbage collection.

Answer 4: As answered in previous garbage collection interview question, second line shows heap occupancy before and after Garbage collection. If we subtract value of right side 13569K, to value on left side 15604K, we can get total memory freed by GC.

Answer 5: `0.0636056 secs` on second line denotes total time it took to collect dead objects during Garbage collection. It also include time taken to GC young generation which is shown in first line `(0635032 secs)`.

Answer 6: 64K

Here are few more interesting *Garbage collection Interview question* for your practice, I haven't provided answers of all garbage collection interview questions. If you know the answer than you can post via comments.

Question - What is difference between `-XX:ParallelGC` and `-XX:ParallelOldGC`?

Question - When do you `ConcurrentMarkSweep` Garbage collector and `Throughput` GC?

Question - What is difference between `ConcurrentMarkSweep` and G1 garbage collector?

Question - Have you done any garbage collection tuning? What was your approach**?**

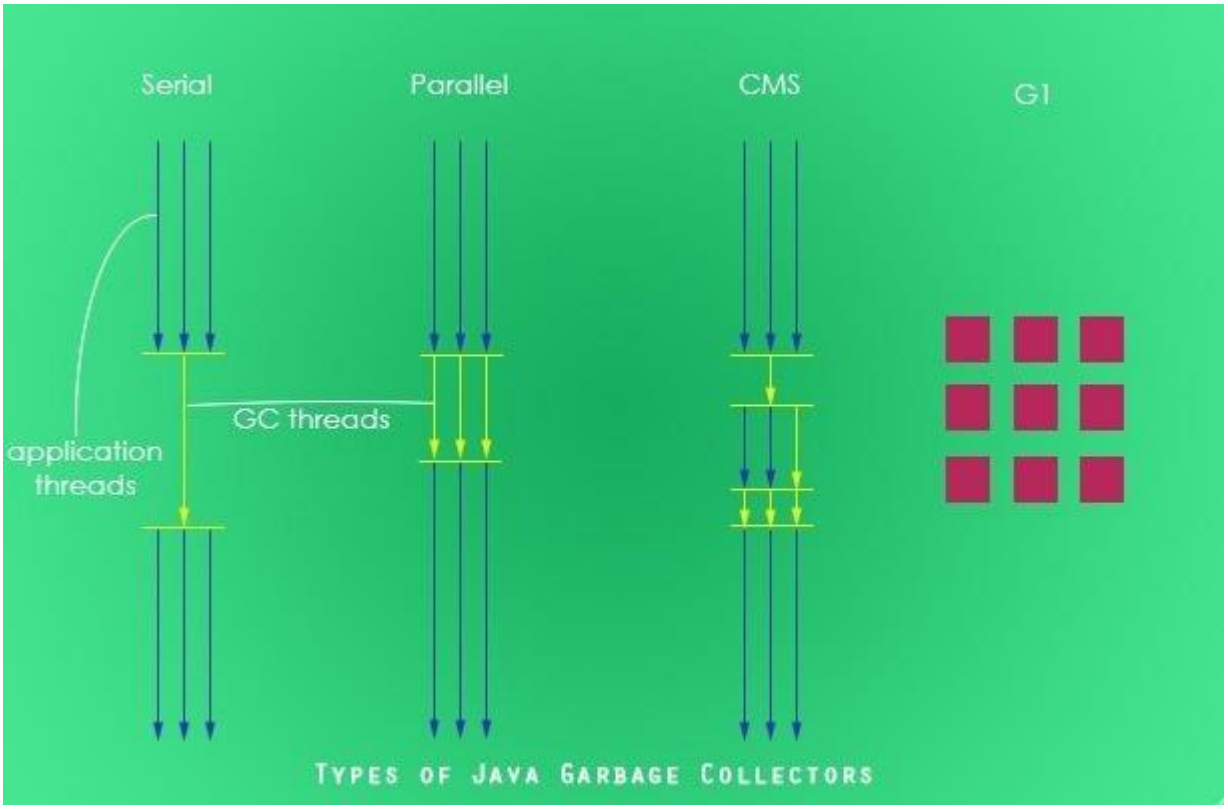
These were some Garbage collection interview questions and answers, may help on your Java Interview preparation. If you have got any interesting interview questions related to GC than don't forget to share with us.

Types of Java Garbage Collectors

In this tutorial we will go through the various type of Java garbage collectors available. Garbage collection is an automatic process in Java which relieves the programmer of object memory allocation and de-allocation chores. This is the third part in the garbage collectison tutorial series. In the previous part 2 we saw about how garbage collection works in Java, it is an interesting read and I recommend you to go through it. In the part 1 introduction to Java garbage collection, we saw about the JVM architecture, heap memory model and surrounding Java terminologies.

Java has **four types of garbage collectors**,

1. Serial Garbage Collector
2. Parallel Garbage Collector
3. CMS Garbage Collector
4. G1 Garbage Collector



Each of these four types has its own advantages and disadvantages. Most importantly, we the programmers can choose the type of garbage collector to be used by the JVM. We can choose them by passing the choice as JVM argument. Each of these types differ largely and can provide completely different application performance. It is critical to understand each of these types of garbage collectors and use it rightly based on the application.

1. Serial Garbage Collector

Serial garbage collector works by holding all the application threads. It is designed for the single-threaded environments. It uses just a single thread for garbage collection. The way it works by freezing all the application threads while doing garbage collection may not be suitable for a server environment. It is best suited for simple command-line programs.

Turn on the `-XX:+UseSerialGC` JVM argument to use the serial garbage collector.

2. Parallel Garbage Collector

Parallel garbage collector is also called as throughput collector. It is the default garbage collector of the JVM. Unlike serial garbage collector, this uses multiple threads for garbage collection. Similar to serial garbage collector this also freezes all the application threads while performing garbage collection.

3. CMS Garbage Collector

Concurrent Mark Sweep (CMS) garbage collector uses multiple threads to scan the heap memory to mark instances for eviction and then sweep the marked instances. CMS garbage collector holds all the application threads in the following two scenarios only,

1. while marking the referenced objects in the tenured generation space.
2. if there is a change in heap memory in parallel while doing the garbage collection.

In comparison with parallel garbage collector, CMS collector uses more CPU to ensure better application throughput. If we can allocate more CPU for better performance then CMS garbage collector is the preferred choice over the parallel collector.

Turn on the `XX:+USeParNewGC` JVM argument to use the CMS garbage collector.

4. G1 Garbage Collector

G1 garbage collector is used for large heap memory areas. It separates the heap memory into regions and does collection within them in parallel. G1 also does compacts the free heap space on the go just after reclaiming the memory. But CMS garbage collector compacts the memory on stop the world (STW) situations. G1 collector prioritizes the region based on most garbage first.

Turn on the `-XX:+UseG1GC` JVM argument to use the G1 garbage collector.

Java 8 Improvement

Turn on the `-XX:+UseStringDeduplication` JVM argument while using G1 garbage collector. This optimizes the heap memory by removing duplicate String values to a single char[] array. This option is introduced in Java 8 u 20.

Given all the above four types of Java garbage collectors, which one to use depends on the application scenario, hardware available and the throughput requirements.

Garbage Collection JVM Options

Following are the key JVM options that are related to Java garbage collection.

Type of Garbage Collector to run

Option	Description
-XX:+UseSerialGC	Serial Garbage Collector

-XX:+UseParallelGC	Parallel Garbage Collector
-XX:+UseConcMarkSweepGC	CMS Garbage Collector
-XX:ParallelCMSThreads=	CMS Collector – number of threads to use
-XX:+UseG1GC	G1 Gargbage Collector

GC Optimization Options

Option	Description
-Xms	Initial heap memory size
-Xmx	Maximum heap memory size
-Xmn	Size of Young Generation
-XX:PermSize	Initial Permanent Generation size
-XX:MaxPermSize	Maximum Permanent Generation size

Example Usage of JVM GC Options

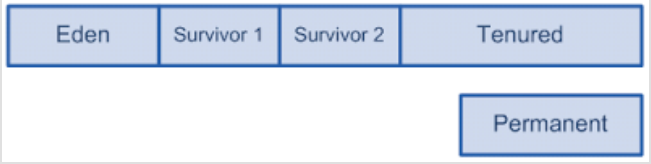
```
java -Xmx12m -Xms3m -Xmn1m -XX:PermSize=20m -XX:MaxPermSize=20m -XX:+UseSerialGC -jar java-application.jar
```

In the next part of this Java garbage collection tutorial series, we will see about how to monitor and analyze the garbage collection with an example Java application.

Stop the world pauses of JVM due to work of garbage collector are known foes of java based application. HotSpot JVM has a set of very advanced and tunable garbage collectors, but to find optimal configuration it is very important to understand an exact mechanics garbage collection algorithms. This article one of series explaining how exactly GC spends our precious CPU cycle during stop-the-world pauses. An algorithm for young space garbage collection in HotSpot is explained in this issue.

Structure of heap

Most of modern GCs are generational. That means java heap memory is separated into few spaces. Spaces are usually distinguished by "age" of objects. Objects are allocated in young space, then, if they survive long enough, eventually promoted to old (tenured) space. That approach rely on hypothesis that most object "die young", i.e. majority of objects are becoming garbage shortly after being allocated. All HotSpot garbage collectors are separating memory into 5 spaces (though for G1 collector spaces may not be continuous).



- Eden are space there objects are allocated,
- Survivor spaces are used to receive object during young (or minor GC),
- Tenured space is for long lived objects,
- Permanent space is for JVM own objects (like classes and JITed code), it is behaves very like tenured space so we will ignore it for rest of article.

Eden and 2 survivor spaces together are called young space.

HotSpot GC algorithms overview

HotSpot JVM is implementing few algorithms for GC which are combined in few possible GC profiles.

- Serial generational collector (`-XX:+UseSerialGC`).
- Parallel for young space, serial for old space generational collector (`-XX:+UseParallelGC`).
- Parallel young and old space generational collector (`-XX:+UseParallelOldGC`).
- Concurrent mark sweep with serial young space collector (`-XX:+UseConcMarkSweepGC -XX:-UseParNewGC`).
- Concurrent mark sweep with parallel young space collector (`-XX:+UseConcMarkSweepGC -XX:+UseParNewGC`).
- G1 garbage collector (`-XX:+UseG1GC`).

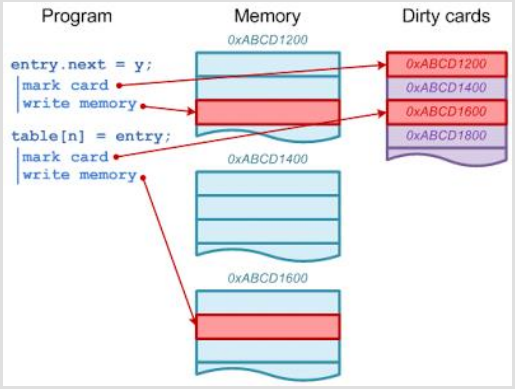
All profiles except G1 are using almost same young space collection algorithms (with serial vs parallel variations).

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.

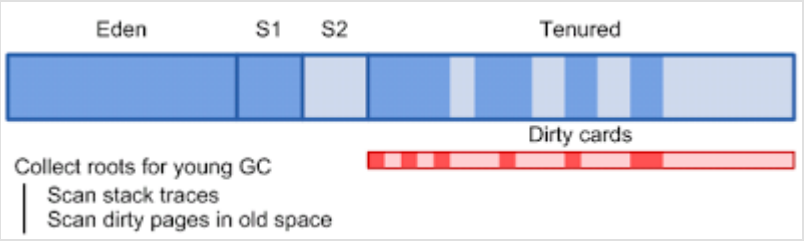
Dirty cards write-barrier (card marking)

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.

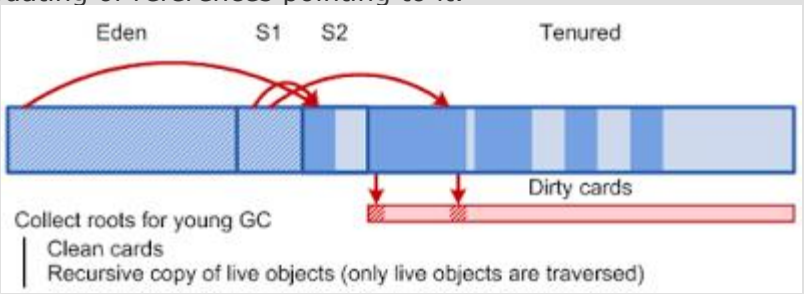


Young space collection algorithm

Almost all new objects (there are few exception when new object can be allocated directly in old space) are allocated in Eden space. To be more effective HotSpot is using thread local allocation blocks (TLAB) for allocation of new objects, but TLAB themselves are allocated in Eden. Once Eden becomes full minor GC is triggered. Goal of minor GC is to clear fresh garbage in Eden space. Copy-collection algorithm is used (live objects are copied to another space, and then whole space is marked as free memory). But before start collecting live objects, JVM should find all root references. Root references for minor GC are references from stack and all references from old space.
Normally collection of all reference from old space will require scanning through all objects in old space. That is why we need write-barrier. All objects in young space have been created (or relocated) since last reset of write-barrier, so non-dirty pages cannot have references into young space. This means we can scan only object in dirty pages.



Once initial reference set is collected, dirty cards are reset and JVM starts coping of live objects from Eden and one of survivor spaces into other survivor space. JVM only need to spend time on live objects. Relocating of object also requires updating of references pointing to it.



While JVM is updating references to relocated object, memory pages get marked again, so we can be sure what on next young GC only dirty pages has references to young space.

Finally we have Eden and one survivor space clean (and ready for allocation) and one survivor space filled with objects.

Object promotion

If object is not cleared during young GC it will be eventually copied (promoted) to old space. Promotion occurs in following situations:

- `-XX:+AlwaysTenure` makes JVM to promote objects directly to old space instead of survivor space (survivor spaces are not used in this case).
- once survivor space is full, all remaining live object are relocated directly to old space.
- If object has survived certain number of young space collections, it will be promoted to old space (required number of collections can be adjusted using `-XX:MaxTenuringThreshold` option and `-XX:TargetSurvivorRatio` JVM options).

Allocation of new objects in old space

It would be beneficial if we could possibly allocate long lived objects directly in old space. Unfortunately there is no way to instruct JVM to do this for particular object. But there are few cases when object can be allocated directly in old space.

- Option `-XX:PretenureSizeThreshold=<n>` instructs JVM what all objects larger than `<n>` bytes should be allocated directly in old space (though if object size fits TLAB, JVM will allocate it in TLAB and thus young space, so you should also limit TLAB size).
- If object is larger than size of Eden space it also will be allocated in old space.

Unlike application objects, system objects are always allocated by JVM directly in permanent space.

Parallel execution

Most of task during young space collection can be done in parallel. If there are several CPUs available, JVM can utilize them to compress duration of stop-the-world pause during collection. Number of threads can be configured in HotSpot JVM by `-XX:ParallelGCTreads=` parameter. By default JVM will choose number of thread by number of available CPU. As expected, serial version of collector will ignore this parameter because it can use only one CPU. Using parallel collection reduces time of stop-the-world pause by factor close to number of physical cores.

Time of young GC

Young space collection happens during stop-the-world pause (all non-GC-related threads in JVM are suspended). Wall clock time of stop-the-world pause is very important factor for applications (especially applications requiring fast response time). Parallel execution affects wall clock time of pause but not work effort to be done.
Let's summarize components of young GC pause. Total pause time can be written as:
$T_{young} = T_{stack\_scan} + T_{card\_scan} + T_{old\_scan} + T_{copy}$; there T_{young} is total time of young GC pause, T_{stack_scan} is time to scan root in stacks, T_{card_scan} is time to scan card table to find dirty pages, T_{old_scan} is time to scan roots in old space and T_{copy} is time to copy live objects (1).
Thread stack are usually very small, so major factors affecting time of young GC is T_{card_scan} , T_{old_scan} and T_{copy}.
Another important parameter is frequency of young GC. Period between young collections is mainly determined by application allocation rate (bytes per second) and size of Eden space.

$P_{young} = S_{eden} / R_{alloc}$; there P_{young} is period between young GC, S_{eden} is size of Eden and R_{alloc} is rate of memory allocations (bytes per second) (2).
T_{stack_scan} – can be considered application specific constant.
T_{card_scan} – is proportional to size of old space. Literally, JVM have to check single byte of table for each 512 bytes of heap (e.g. 8G of heap -> 16m to scan).
T_{old_scan} – is proportional to number of dirty cards in old space at the moment of young GC. If we assume that references to young space are distributed randomly in old space, then we can provide following formula for time of old space scanning.

$$T_{old\_scan} = D \cdot \left(1 - \frac{k_{card}}{S_{old} + n_{card}}\right)$$

; there S_{old} is size of old space and D, k_{card} and n_{card} are coefficients specific for application (3).

T_{copy} – is proportional to number of live objects in heap. We can approximate it by formula:

$$T_{copy} = k_{copy} \cdot P_{young} \cdot R_{long\_live} = k_{copy} \cdot \left(\frac{S_{eden}}{R_{alloc}}\right) \cdot \left(k_{survive} \cdot k_{tenure} \cdot R_{alloc}\right)$$

$$T_{copy} = k_{copy} \cdot k_{survive} \cdot k_{tenure} \cdot S_{eden}$$

There k_{copy} is effort to copy object, R_{long_live} is rate of allocation of long lived objects,

$$k_{survive} = \frac{R_{long\_live}}{R_{alloc}}$$

($k_{survive}$ usually very small), and k_{tenure} is a coefficient to approximate aging of object in young space before tenuring ($k_{tenure} \geq 1$) (4).
Now we can analyze how various JVM options may affect time and frequency of young GC.

Size of old space.
Size of old space is affecting T_{card_scan} and T_{old_scan} part of young GC pause time according to formulas above. So we as we are increasing size of old space (read total heap size) time of young GC pauses will grow and it cannot be helped. After certain size of heap (usually 4-8 Gb) time of young collection is dominated by T_{card_scan} (technically T_{copy} can be even greater than T_{card_scan} , but it usually can be controlled by tuning of other GC options).
HotSpot JVM options:

```
-Xmx=<n> -Xms=<n>
```

Size of Eden space
Period between young GC is proportional to size of Eden. T_{copy} is also proportional to size of eden but in practice $k_{survive}$ can be so small that for some applications we can forget about Tcopy. Unfortunately time between young GC will also affect coefficient D in equation (4). Though dependency between D and P_{young} is very application specific, increasing P_{young} will increase D and as a consequence T_{scan_old}.
HotSpot JVM options:

```
-XX:NewSize=<n> -XX:MaxNewSize=<n>
```

Size of survivor space.

Size of survivor space puts hard limit of how much objects can stay in young space between collections. Changing size of survivor space may affect k_{tenure} (or mat not, e.g. if k_{tenure} is already 1).

HotSpot JVM options:

```
-XX:SurviorRatio=<n>
```

Max tenuring threshold and **target survivor ratio**

These two JVM options also allow artificially adjust k_{tenure}.

HotSpot JVM options:

```
-XX:TargetSurviorRatio=<n>
-XX:MaxTenuringThreshold=<n>
-XX:+AlwaysTenure
-XX:+NeverTenure
```

Pretenuring threshold

For certain applications using pretenuring threshold could reduce$k_{survive}$ due to allocation of long lived object directly in old space.

HotSpot JVM options:

```
-XX:PretenureThreshold=<n>
```

Understanding Java Garbage Collection

What are the benefits of knowing how garbage collection (GC) works in Java? Satisfying the intellectual curiosity as a software engineer would be a valid cause, but also, understanding how GC works can help you write much better Java applications.

This is a very personal and subjective opinion of mine, but I believe that a person well versed in GC tends to be a better Java developer. If you are interested in the GC process, that means you have experience in developing applications of certain size. If you have thought carefully about choosing the right GC algorithm, that means you completely understand the features of the application you have developed. Of course, this may not be common standards for a good developer. However, few would object when I say that understanding GC is a requirement for being a great Java developer.

This is the first of a series of "*Become a Java GC Expert*" articles. I will cover the *GC introduction* this time, and in the next article, I will talk about analyzing GC status and GC tuning examples from NHN.

The purpose of this article is to introduce GC to you in an easy way. I hope this article proves to be very helpful. Actually, my colleagues have already published a few great articles on Java Internals which became quite popular on Twitter. You may refer to them as well.

Returning back to Garbage Collection, there is a term that you should know before learning about GC. The term is "**stop-the-world**." Stop-the-world will occur no matter which GC algorithm you choose. *Stop-the-world* means that the JVM is stopping the application from running to execute a GC. When stop-the-world occurs, every thread except for the threads needed for the GC will stop their tasks. The interrupted tasks will resume only after the GC task has completed. GC tuning often means reducing this stop-the-world time.

Generational Garbage Collection

Java does not explicitly specify a memory and remove it in the program code. Some people sets the relevant object to null or use System.gc() method to remove the memory explicitly. Setting it to null is not a big deal, but calling System.gc() method will affect the system performance drastically, and must not be carried out. (Thankfully, I have not yet seen any developer in NHN calling this method.)

In Java, as the developer does not explicitly remove the memory in the program code, the garbage collector finds the unnecessary (garbage) objects and removes them. This garbage collector was created based on the following two hypotheses. (It is more correct to call them suppositions or preconditions, rather than hypotheses.)

Most objects soon become unreachable.

References from old objects to young objects only exist in small numbers.

These hypotheses are called the **weak generational hypothesis**. So in order to preserve the strengths of this hypothesis, it is physically divided into two - **young generation** and **old generation** - in HotSpot VM.

Young generation: Most of the newly created objects are located here. Since most objects soon become unreachable, many objects are created in the young generation, then disappear. When objects disappear from this area, we say a "**minor GC**" has occurred.

Old generation: The objects that did not become unreachable and survived from the young generation are copied here. It is generally larger than the young generation. As it is bigger in size, the GC occurs less frequently than in the young generation. When objects disappear from the old generation, we say a "**major GC**" (or a "**full GC**") has occurred.

Let's look at this in a chart.

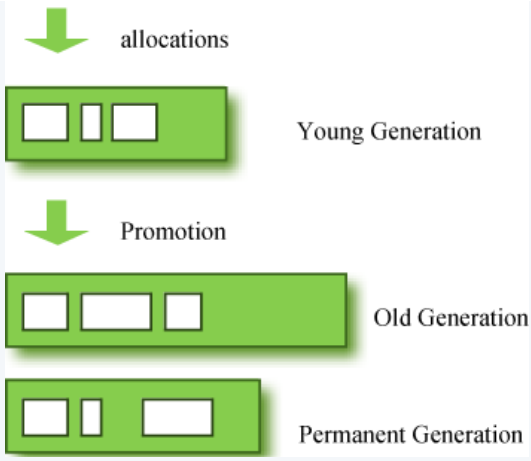


Figure 1: GC Area & Data Flow.

The **permanent generation** from the chart above is also called the "**method area,**" and it stores classes or interned character strings. So, this area is definitely not for objects that survived from the old generation to stay permanently. A GC may occur in this area. The GC that took place here is still counted as a major GC.

Some people may wonder:

What if an object in the old generation need to reference an object in the young generation?

To handle these cases, there is something called the a "card table" in the old generation, which is a *512 byte chunk*. Whenever an object in the old generation references an object in the young generation, it is recorded in this table. When a GC is executed for the young generation, only this card table is searched to determine whether or not it is subject for GC, instead of checking the reference of all the objects in the old generation. This card table is managed with write barrier. This *write barrier* is a device that allows a faster performance for minor GC. Though a bit of overhead occurs because of this, the overall GC time is reduced.

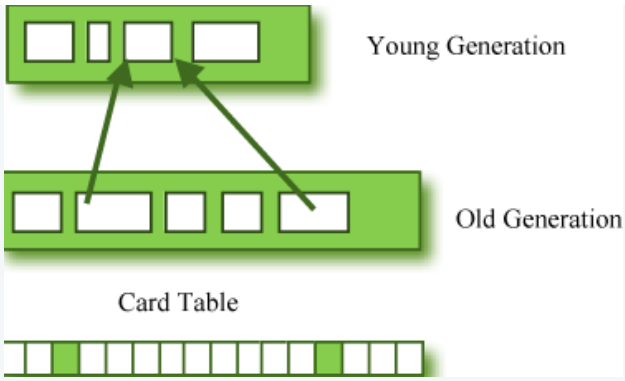


Figure 2: Card Table Structure.

Composition of the Young Generation

In order to understand GC, let's learn about the young generation, where the objects are created for the first time. The young generation is divided into 3 spaces.
One **Eden** space
Two **Survivor** spaces

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.

As you can see by checking these steps, one of the Survivor spaces must remain empty. If *data exists in both Survivor spaces, or the usage is 0 for both spaces*, then take that as a sign that **something is wrong with your system**.
The process of data piling up into the old generation through minor GCs can be shown as in the below chart:

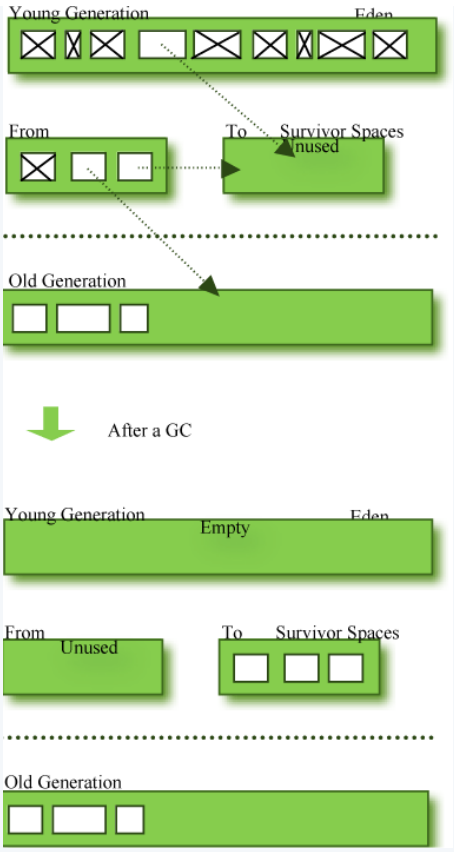


Figure 3: Before & After a GC.

Note that in HotSpot VM, two techniques are used for faster memory allocations. One is called "**bump-the-pointer**," and the other is called "**TLABs (Thread-Local Allocation Buffers)**."

Bump-the-pointer technique tracks the last object allocated to the Eden space. That object will be located on top of the Eden space. And if there is an object created afterwards, it checks only if the size of the object is suitable for the Eden space. If the said object seems right, it will be placed in the Eden space, and the new object goes on top. So, when new objects are created, only the lastly added object needs to be checked, which allows much faster memory allocations. However, it is a different story if we consider a multithreaded environment. To save objects used by multiple threads in the Eden space for Thread-Safe, an inevitable lock will occur and the performance will drop due to the lock-contention. **TLABs** is the solution to this problem in HotSpot VM. This allows each thread to have a small portion of its Eden space that corresponds to its own share. As each thread can only access to their own TLAB, even the bump-the-pointer technique will allow memory allocations without a lock.
This has been a quick overview of the GC in the young generation. You do not necessarily have to remember the two techniques that I have just mentioned. You will not go to jail for not knowing them. But please remember that after the objects are first created in the Eden space, and the long-surviving objects are moved to the old generation through the Survivor space.

GC for the Old Generation

The old generation basically performs a GC when the data is full. The execution procedure varies by the GC type, so it would be easier to understand if you know different types of GC.
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

Among these, the **serial GC must not be used on an operating server**. This GC type was created when there was only one CPU core on desktop computers. Using this serial GC will drop the application performance significantly.

Now let's learn about each GC type.

Serial GC (-XX:+UseSerialGC)

The GC in the young generation uses the type we explained in the previous paragraph. The GC in the old generation uses an algorithm called "**mark-sweep-compact**."

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.

Parallel GC (-XX:+UseParallelGC)

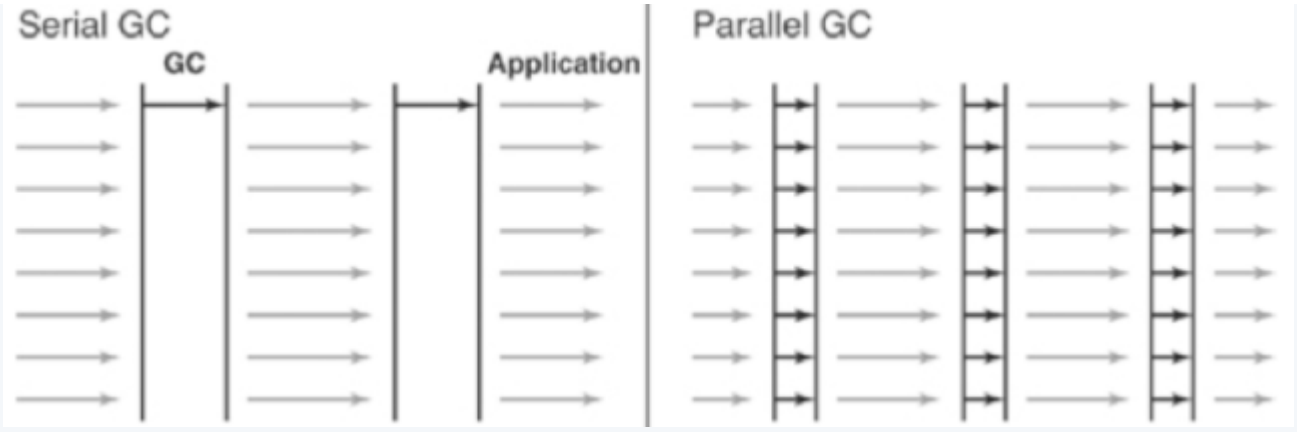


Figure 4: Difference between the Serial GC and Parallel GC.

From the picture, you can easily see the difference between the serial GC and parallel GC. 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**."

Parallel Old GC(-XX:+UseParallelOldGC)

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.

CMS GC (-XX:+UseConcMarkSweepGC)

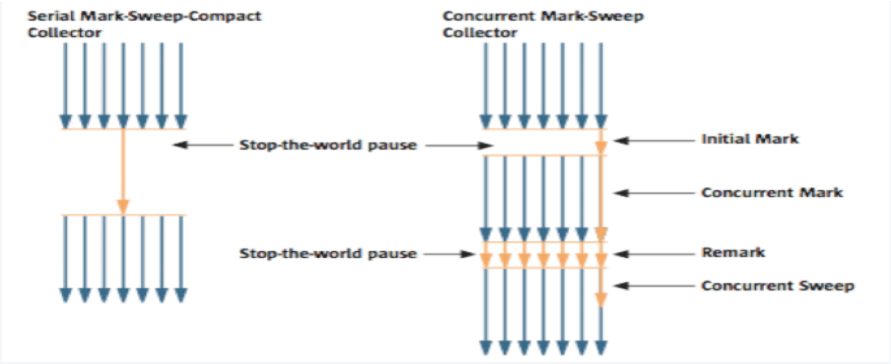


Figure 5: Serial GC & CMS GC.

As you can see from the picture, the Concurrent Mark-Sweep GC is much more complicated than any other GC types that I have explained so far. The early *initial mark* step is simple. The surviving objects among the objects the closest to the

classloader are searched. So, the pausing time is very short. In the *concurrent mark* step, the objects referenced by the surviving objects that have just been confirmed are tracked and checked. The difference of this step is that it proceeds while other threads are processed at the same time. In the *remark*step, the objects that were newly added or stopped being referenced in the concurrent mark step are checked. Lastly, in the *concurrent sweep* step, the garbage collection procedure takes place. The garbage collection is carried out while other threads are still being processed. Since this GC type is performed in this manner, the pausing time for GC is very short. 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.

You need to carefully review before using this type. Also, if the compaction task needs to be carried out because of the many memory fragments, the stop-the-world time can be longer than any other GC types. You need to check how often and how long the compaction task is carried out.

G1 GC

Finally, let's learn about the garbage first (G1) GC.

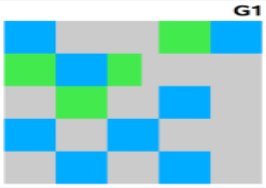


Figure 6: Layout of G1 GC.

If you want to understand G1 GC, forget everything you know about the young generation and the old generation. As you can see in the picture, one object is allocated to each grid, and then a GC is executed. Then, once one area is full, the objects are allocated to another area, and then a GC is executed. The steps where the data moves from the three spaces of the young generation to the old generation cannot be found in this GC type. This type was created to replace the CMS GC, which has causes a lot of issues and complaints in the long term.
The biggest advantage of the G1 GC is its **performance**. It is faster than any other GC types that we have discussed so far. But in JDK 6, this is called an *early access* and can be used only for a test. It is officially included in JDK 7. In my personal opinion, we need to go through a long test period (at least 1 year) before NHN can use JDK7 in actual services, so you probably should wait a while. Also, I heard a few times that a JVM crash occurred after applying the G1 in JDK 6. Please wait until it is more stable.
I will talk about the **GC tuning** in the next issue, but I would like to ask you one thing in advance. If the size and the type of all objects created in the application are identical, all the GC options for WAS used in our company can be the same. But the size and the lifespan of the objects created by WAS vary depending on the service, and the type of equipment varies as well. In other words, just because a certain service uses the GC option "A," it does not mean that the same option will bring the best results for a different service. It is necessary to find the best values for the WAS threads, WAS instances for each equipment and each GC option by constant tuning and monitoring. This did not come from my personal experience, but from the discussion of the engineers making Oracle JVM for JavaOne 2010.
In this issue, we have only glanced at the GC for Java. Please look forward to our next issue, where I will talk about **how to monitor the Java GC status and tune GC**.

The Principles of Java Application Performance Tuning

This is the fifth article in the series of "Become a Java GC Expert". In the first issue Understanding Java Garbage Collection we have learned about the processes for different GC algorithms, about how GC works, what Young and Old Generation is, what you should know about the 5 types of GC in the new JDK 7, and what the performance implications are for each of these GC types.

In the second article How to Monitor Java Garbage Collection we have explained how JVM actually runs the Garbage Collection in the real time, how we can monitor GC, and which tools we can use to make this process faster and more effective.

In the third article How to Tune Java Garbage Collection we have shown some of the best options based on real cases as our examples that you can use for GC tuning. Also we have explained how to minimize the number of objects passed to Old Area, decreasing Full GC time, as well as how to set GC type and the memory size.

In the fourth article MaxClients in Apache and its effect on Tomcat during Full GC we have explained the importance of `MaxClients` parameter in Apache that significantly affects the overall system performance when GC occurs.

In this fifth article I will explain about the principles of Java application performance tuning. Specifically, I will explain what is required in order to tune the performance of Java application, the steps you need to perform to identify whether your application needs tuning. I will also explain the problems you may encounter during performance tuning. The article will be finalized with the recommendations you need to follow to make better decisions when tuning Java applications.

Overview

Not every application requires tuning. If an application performs as well as expected, you don't need to exert additional efforts to enhance its performance. However, it would be difficult to expect an application would reach its target performance as soon as it finishes debugging. This is when tuning is required. Regardless of the implementation language, tuning an application requires high expertise and concentration. Also, you may not use the same method for tuning a certain application to tune another application. This is because each application has its unique action and a different type of resource usage. For this reason, tuning an application requires more basic knowledge compared to the knowledge required to write an application. For example, you need knowledge on virtual machines, operating systems and computer architectures. When you focus on an application domain based on such knowledge, you can successfully tune an application.

Sometimes Java application tuning requires only changing JVM options, such as Garbage Collector, but sometimes it requires changing the application source code. Whichever method you choose, you need to monitor the process of executing the Java application first. For this reason, the issues this article will deal with are as follows:

How can I monitor a Java application?

What JVM options should I give?

How can I know if modifying source codes is required or not?

Knowledge Required to Tune the Performance of Java Applications

Java applications operate inside Java Virtual Machine (JVM). Therefore, to tune a Java application, you need to understand the JVM operation process. I have previously blogged about Understanding JVM Internals where you can find great insights about JVM.

The knowledge regarding the process of the operation of JVM in this article mainly refers to the knowledge of Garbage Collection (GC) and Hotspot. Although you may not be able to tune the performance of all kinds of Java applications only with the knowledge on GC or Hotspot, these two factors influence the performance of Java applications in most cases.

It is noted that from the perspective of an operating system JVM is also an application process. To make an environment in which a JVM can operate well, you should understand how an OS allocates resources to processes. This means, to tune the performance of Java applications, you should have an understanding of OS or hardware as well as JVM itself.

Another aspect is that knowledge of Java language domain is also important. It is also important to understand lock or concurrency and to be familiar with class loading or object creation.

When you carry out Java application performance tuning, you should approach it by integrating all this knowledge.

The Process of Java Application Performance Tuning

Figure 1 shows a flow chart from the book <Java Performance> co-authored by Charlie Hunt and Binu John. This chart shows the process of Java application performance tuning.

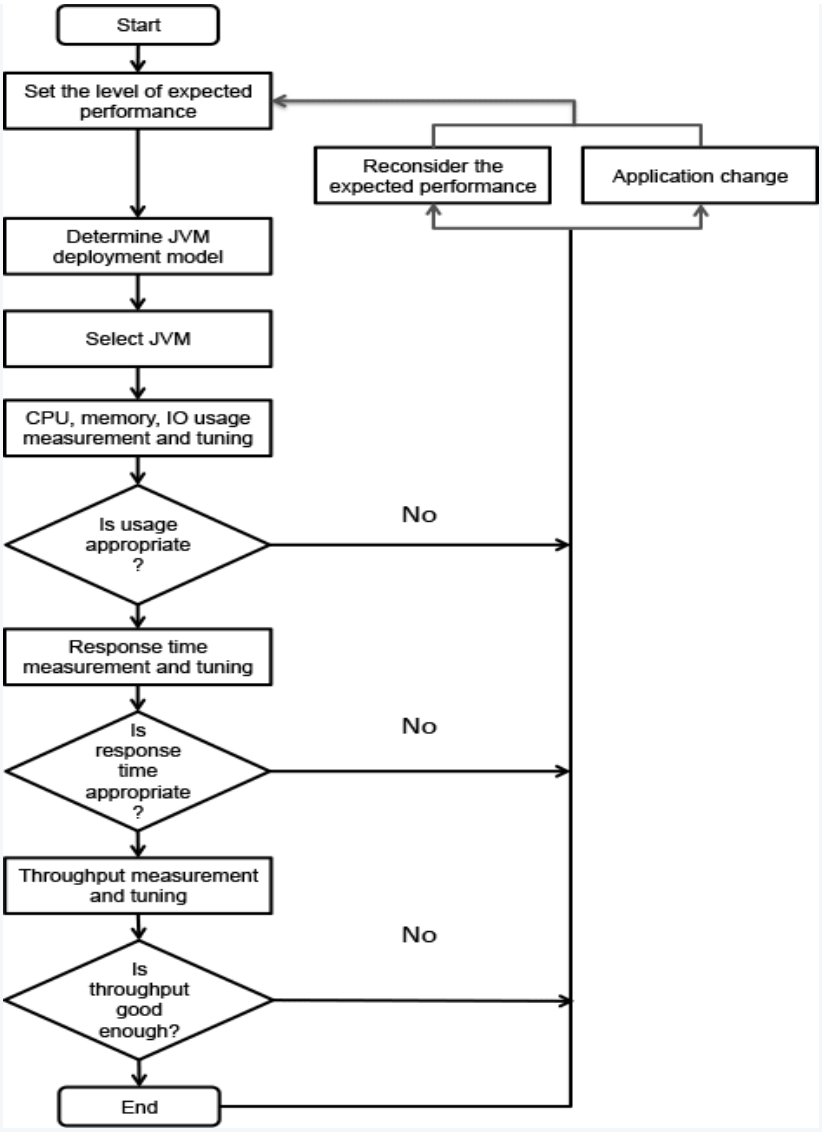


Figure 1: The Process of Tuning the Performance of Java Applications.

The above process is not a one-time process. You may need to repeat it until the tuning is completed. This also applies to determining an expected performance value. In the process of tuning, sometimes you should lower the expected performance value, and sometimes raise it.

JVM distribution model

A **JVM distribution model** is related with making a decision on whether to operate Java applications on a single JVM or to operate them on multiple JVMs. You can decide it according to its availability, responsiveness and maintainability. When operating JVM on multiple servers, you can also decide whether to run multiple JVMs on a single server or to run a single JVM per server. For example, for each server, you can decide whether to run a single JVM using a heap of 8 GB, or to use four JVMs each using a heap of 2 GB. Of course, you can decide the number of JVMs running on a single server depending on the number of cores and the characteristics of the application. When comparing the two settings in terms of responsiveness, it might be more advantageous to use a heap of 2 GB rather than 8 GB for the same application, for it takes shorter to perform a full garbage collection when using a heap of 2 GB. If you use a heap of 8 GB, however, you can reduce the frequency of full GCs. You can also improve responsiveness by increasing the hit rate if the application uses internal cache. Therefore, you can choose a suitable distribution model by taking into account the characteristics of the application and the method to overcome the disadvantage of the model you chose for some advantages.

JVM architecture

Selecting a JVM means whether to use a **32-bit JVM** or a **64-bit JVM**. Under the same conditions, you had better choose a 32-bit JVM. This is because a 32-bit JVM performs better than a 64-bit JVM. However, the maximum logical heap size of a 32-bit JVM is 4 GB. (However, actual allocatable size for both 32-bit OS and 64-bit OS is 2-3 GB.) It is appropriate to use a 64-bit JVM when a heap size larger than this is required.

Table 1: Performance Comparison (source).		
Benchmark	Time (sec)	Factor
C++ Opt	23	1.0x
C++ Dbg	197	8.6x
Java 64-bit	134	5.8x
Java 32-bit	290	12.6x
Java 32-bit GC*	106	4.6x
Java 32-bit SPEC GC*	89	3.7x
Scala	82	3.6x
Scala low-level*	67	2.9x
Scala low-level GC*	58	2.5x
Go 6g	161	7.0x
Go Pro*	126	5.5x

The next step is to run the application and to measure its performance. This process includes tuning GC, changing OS settings and modifying codes. For these tasks, you can use a system monitoring tool or a profiling tool.
It should be noted that tuning for responsiveness and tuning for throughput could be different approaches. Responsiveness will be reduced if stop-the-world occurs from time to time, for example, for a full garbage collection despite a large amount of throughput per unit time. You also need to consider that a trade-off could occur. Such trade-off could occur not only between responsiveness and throughput. You may need to use more CPU resources to reduce memory usage or put up with reduction in responsiveness or throughput. As opposite cases could likewise occur, you need to approach it according to the priority.
The flow chart of **Figure 1** above shows the performance tuning approach for almost all kinds of Java applications, including Swing applications. However, this chart is somewhat unsuitable for writing a server application for Internet service as our company NHN does. The flow chart in **Figure 2** below is a simpler procedure designed based on **Figure 1** to be more suitable for NHN.

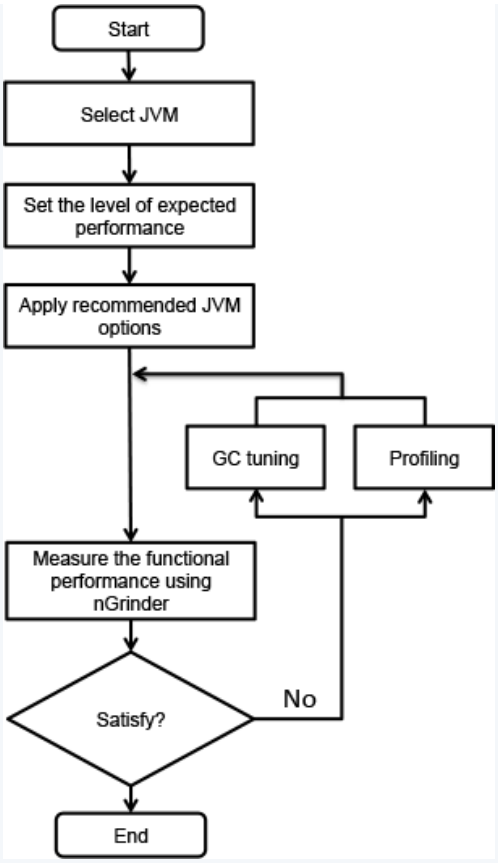


Figure 2: A Recommended Procedure for Tuning NHN's Java Applications.

Select JVM in the above flow chart means using a 32-bit JVM as much as possible except when you need to use a 64-bit JVM to maintain cache of several GB.
Now, based on the flow chart in **Figure 2**, you will learn about things to do to execute each of the steps.q

JVM Options

I will explain how to specify suitable JVM options mainly for a web application server. Despite not being applied to every case, the **best GC algorithm**, especially for web server applications, is the Concurrent Mark Sweep GC. This is because what matters is **low latency**. Of course, when using the Concurrent Mark Sweep, sometimes a very long stop-the-world phenomenon could take place due to fractions. Nevertheless, this problem is likely to be resolved by adjusting the new area size or the fraction ratio.
Specifying the **new area size** is as important as specifying the **entire heap size**. You had better specify the ratio of the new area size to the entire heap size by using `-XX:NewRatio` or specify the desired new area size by using the `-XX:NewSize` option. Specifying a new area size is important because most objects cannot survive long. In web applications, most objects, except cache data, are generated when `HttpResponse` to `HttpRequest` is created. This time hardly exceeds a second. This means the life of objects does not exceed a second, either. If the new area size is not large, it should be moved to the old area to make space for newly created objects. The cost for GC for the old area is much bigger than that for the new area; therefore, it is good to set the size of the new area sufficiently.
If the new area size exceeds a certain level, however, responsiveness will be reduced. This is because the garbage collection for the new area is basically to copy data from one survivor area to another survivor area. Also, the stop-the-world phenomenon will occur even when performing GC for the new area as well as the old area. If the new area becomes bigger, the survivor area size will increase, and thus the size of the data to copy will increase as well. Given such characteristics, it is good to set a suitable new area size by referring to the `NewRatio` of HotSpot JVM by OS.

Table 2: NewRatio by OS and option.	
OS and option	Default -XX:NewRatio
Sparc -server	2
Sparc -client	8
x86 -server	8
x86 -client	12

If the `NewRatio` is specified, `1/(NewRatio +1)` of the entire heap size becomes the new area size. You will find the `NewRatio` of **Sparc -server** is very small. This is because the Sparc system was used for more high-end use than x86 when default values were specified. Now it is common to use the x86 server and its performance has also been improved. Thus it is better to specify 2 or 3, which is the value similar to that of the**Sparc -server**.

You can also specify `NewSize` and `MaxNewSize` instead of `NewRatio`. The new area is created as much as the value specified for `NewSize` and the size increments as much as the value specified for `MaxNewSize`. The Eden or Survivor area also increases according to the (specified or default) ratio. As you specify the same size for `-Xs` and `-Xmx`, it is a very good choice to specify the same size for `MaxSize` and `MaxNewSize`.

If you have specified both `NewRatio` and `NewSize`, you should use the bigger one. Therefore, when a heap has been created, you can express the initial New area size as follows:

```
1    min(MaxNewSize, max(NewSize, heap/(NewRatio+1)))
```

However, it is impossible to determine the appropriate entire heap size and New area size in a single attempt. Based on my experience running Web server applications at NHN, I recommend to run Java applications with the following JVM options. After monitoring the performance of the application with these options, you can use a more suitable GC algorithm or options.

Table 3: Recommended JVM options.	
Type	Option
Operation mode	`-sever`
Entire heap size	Specify the same value for `-Xms` and `-Xmx`.
New area size	`-XX:NewRatio`: value of 2 to 4 `-XX:NewSize=?` `-XX:MaxNewSize=?`. Also good to specify `NewSize` instead of `NewRatio`.
Perm size	`-XX:PermSize=256 m` `-XX:MaxPermSize=256 m`. Specify the value to an extent not to cause any trouble operation because it does not affect the performance.
GC log	`-Xloggc:$CATALINA_BASE/logs/gc.log` `-XX:+PrintGCDetails` `-XX:+PrintGCDateStamps`. Leaving a not particularly affect the performance of Java applications. You are recommended to leave a GC log as m possible.
GC algorithm	`-XX:+UseParNewGC` `-XX:+CMSParallelRemarkEnabled` `-XX:+UseConcMarkSweepGC-` `XX:CMSInitiatingOccupancyFraction=75`.This is only a generally recommendable configuration. Other could be better depending on the characteristics of the application.
Creating a heap dump when an OOM error occurs	`-XX:+HeapDumpOnOutOfMemoryError` `-XX:HeapDumpPath=$CATALINA_BASE/logs`

Table 3: Recommended JVM options.	
Type	Option
Actions after an OOM occurs	`-XX:OnOutOfMemoryError=$CATALINA_HOME/bin/stop.sh` or `-XX:OnOutOfMemoryError=$CATALINA_HOME/bin/restart.sh`. After leaving a heap dump, take a proper according to a management policy.

Measuring the Performance of Applications

The information to acquire to grasp the performance of an application is as follows:

TPS (OPS): The information required to understand the performance of an application conceptually.

Request Per Second (RPS): Strictly speaking, RPS is different from responsiveness, but you can understand it as responsiveness. Through RPS, you can check the time it takes for the user to see the result.

RPS Standard Deviation: It is necessary to induce even RPS if possible. If a deviation occurs, you need to check GC tuning or interworking systems.

To obtain a more accurate performance result, you should measure it after warming up the application sufficiently. This is because byte code is expected to be compiled by HotSpot JIT. In general, you can measure actual performance values after applying load to a certain feature for at least 10 minutes by using nGrinderload testing tool.

Tuning in Earnest

You don't need to tune the performance of an application if the result of the execution of nGrinder meets the expectation. If the performance does not meet the expectation, you need to carry out tuning to resolve problems. Now you will see the approach by case.

In the event the Stop-the-World takes long

Long **stop-the-world** time could result from inappropriate GC options or incorrect implementation. You can decide the cause according to the result of a profiler or a heap dump. This means you can judge the cause after checking the type and number of objects of a heap. If you find many unnecessary objects, you had better modify source codes. If you find no particular problem in the process of creating objects, you had better simply change GC options.

To adjust GC options appropriately, you need to have GC log secured for a sufficient period of time. You need to understand in which situation the stop-the-world takes a long time. For more information on the selection of appropriate GC options, read my colleague's blog about How to Monitor Java Garbage Collection.

In the event CPU usage rate is low

When blocking time occurs, both TPS and CPU usage rate will decrease. This might result from the problem of interworking systems or concurrency. To analyze this, you can use an analysis on the result of thread dump or a profiler. For more information on thread dump analysis, read How to Analyze Java Thread Dumps.

You can conduct a very accurate lock analysis by using a commercial profiler. In most cases, however, you can obtain a satisfactory result with only the CPU analyzer in **jvisualvm**.

In the event CPU usage rate is high

If TPS is low but CPU usage rate is high, this is likely to result from inefficient implementation. In this case, you should find out the location of bottlenecks by using a profiler. You can analyze this by using **jvisuavm**, **TPTP** of Eclipse or **JProbe**.

Approach for Tuning

You are advised to use the following approach to tune applications.

First, you should check whether performance tuning is necessary. The process of performance measuring is not easy work. You are also not guaranteed to obtain a satisfactory result all the time. Therefore, if the application already meets its target performance, you don't need to invest additionally in performance.

The problem lies in only a single place. All you have to do is to fix it. The Pareto principle applies to performance tuning as well. This does not mean to emphasize that the low performance of a certain feature results necessarily from a single problem. Rather, this emphasizes that we should focus on one factor that has the biggest influence on the performance when approaching performance tuning. Thus, you could handle another problem after fixing the most important one. You are advised to try to fix just one problem at a time.

You should consider the balloon effect. You should decide what to give up to get something. You can improve responsiveness by applying cache but if the cache size increases, the time it takes to carry out a full GC will increase as well. In general, if you want a small amount of memory usage, throughput or responsiveness could be deteriorated. Thus, you need to consider what is most important and what is less important.

So far, you have read the method for Java application performance tuning. To introduce a concrete procedure for performance measurement, I had to omit some details. Nevertheless, I think this could satisfy most of the cases for tuning Java web server applications.

How to Monitor Java Garbage Collection

This is the second article in the series of "*Become a Java GC Expert*". In the first issue Understanding Java Garbage Collection we have learned about the processes for different GC algorithms, about how GC works, what Young and Old Generation is, what you should know about the 5 types of GC in the new JDK 7, and what the performance implications are for each of these GC types.

In this article, I will explain **how JVM is actually running Garbage Collection in the real time**.

What is GC Monitoring?

Garbage Collection Monitoring refers to the *process of figuring out how JVM is running GC*. For example, we can find out:
when an object in young has moved to old and by how much,
or when stop-the-world has occurred and for how long.

GC monitoring is carried out *to see if JVM is running GC efficiently*, and *to check if additional GC tuning is necessary*. Based on this information, the application can be edited or GC method can be changed (**GC tuning**).

How to Monitor GC?

There are different ways to monitor GC, but the only difference is how the GC operation information is shown. GC is done by JVM, and since the GC monitoring tools disclose the GC information provided by JVM, you will get the same results no matter how you monitor GC. Therefore, you do not need to learn all methods to monitor GC, but since it only requires a little amount of time to learn each GC monitoring method, knowing a few of them can help you use the right one for different situations and environments.

The tools or JVM options listed below cannot be used universally regardless of the HVM vendor. This is because there is no need for a "standard" for disclosing GC information. In this example we will use **HotSpot JVM**(Oracle JVM). Since NHN is using Oracle (Sun) JVM, there should be no difficulties in applying the tools or JVM options that we are explaining here.

First, the GC monitoring methods can be separated into **CUI** and **GUI** depending on the access interface. The typical CUI GC monitoring method involves using a separate CUI application called "**jstat**", or selecting a JVM option called "**verbosegc**" when running JVM.

GUI GC monitoring is done by using a separate GUI application, and three most commonly used applications would be "jconsole", "jvisualvm" and "Visual GC".

Let's learn more about each method.

jstat

jstat is a monitoring tool in HotSpot JVM. Other monitoring tools for HotSpot JVM are **jps** and **jstatd**. Sometimes, you need all three tools to monitor a Java application.

jstat does not provide only the GC operation information display. It also provides class loader operation information or Just-in-Time compiler operation information. Among all the information jstat can provide, in this article we will only cover its functionality to *monitor* GC operating information.

jstat is located in $JDK_HOME/bin, so if *java* or *javac* can run without setting a separate directory from the command line, so can jstat.

You can try running the following in the command line.

```
$> jstat -gc  $<vmid$> 1000

S0C        S1C        S0U     S1U      EC          EU          OC          OU          PC          PU
3008.0    3072.0     0.0      1511.1   343360.0    46383.0     699072.0    283690.2    75392.0     410
3008.0    3072.0     0.0      1511.1   343360.0    47530.9     699072.0    283690.2    75392.0     410
3008.0    3072.0     0.0      1511.1   343360.0    47793.0     699072.0    283690.2    75392.0     410

$>
```

Just like in the example, the real type data will be output along with the following columns: **S0C S1C S0U S1U EC EU OC OU PC**.
vmid (Virtual Machine ID), as its name implies, is the **ID** for the VM. Java applications running either on a local machine or on a remote machine can be specified using vmid. The vmid for Java application running on a local machine is called **lvmid** (Local vmid), and usually is PID. To find out the lvmid, you can write the PID value using a **ps** command or Windows task manager, but we suggest **jps** because PID and lvmid does not always match.**jps** stands for Java PS. jps shows *vmids* and main method information. Just like ps shows PIDs and process names.
Find out the vmid of the Java application that you want to monitor by using jps, then use it as a parameter in jstat. If you use jps alone, only bootstrap information will show when several WAS instances are running in one equipment. We suggest that you use **ps -ef | grep java** command along with **jps**.
GC performance data needs constant observation, therefore when running jstat, try to output the GC monitoring information on a regular basis.
For example, running "**jstat –gc <vmid> 1000**" (or 1s) will display the GC monitoring data on the console every 1 second. "**jstat –gc <vmid> 1000 10**" will display the GC monitoring information once every 1 second for 10 times in total.

There are many options other than **-gc**, among which GC related ones are listed below.

Option Name	Description
gc	It shows the current size for each heap area and its current usage (Ede, survivor, old, etc.), total number of GC performed, and the accumulated time for GC operations.
gccapactiy	It shows the minimum size (ms) and maximum size (mx) of each heap area, current size, and the number of GC performed for each area. (Does not show current usage and accumulated time for GC operations.)
gccause	It shows the "information provided by -gcutil" + reason for the last GC and the reason for the current GC.
gcnew	Shows the GC performance data for the new area.
gcnewcapacity	Shows statistics for the size of new area.
gcold	Shows the GC performance data for the old area.
gcoldcapacity	Shows statistics for the size of old area.
gcpermcapacity	Shows statistics for the permanent area.
gcutil	Shows the usage for each heap area in percentage. Also shows the total number of GC performed and the accumulated time for GC operations.

Only looking at frequency, you will probably use **-gcutil** (or -gccause), **-gc** and **-gccapacity** the most in that order.

- **-gcutil** is used to check the usage of heap areas, the number of GC performed, and the total accumulated time for GC operations,
- while **-gccapacity** option and others can be used to check the actual size allocated.

You can see the following output by using the **-gc** option:

```
S0C       S1C     …    GCT
1248.0    896.0   …    1.246
```

```
3     1248.0    896.0   …    1.246
4     …         …       …    …
```

Different jstat options show different types of columns, which are listed below. Each column information will be displayed when you use the "jstat option" listed on the right.

Column	Description	Jstat Option
S0C	Displays the current size of Survivor0 area in KB	-gc -gccapacity -gcnew -gcnewcapacity
S1C	Displays the current size of Survivor1 area in KB	-gc -gccapacity -gcnew -gcnewcapacity
S0U	Displays the current usage of Survivor0 area in KB	-gc -gcnew
S1U	Displays the current usage of Survivor1 area in KB	-gc -gcnew
EC	Displays the current size of Eden area in KB	-gc -gccapacity -gcnew -gcnewcapacity
EU	Displays the current usage of Eden area in KB	-gc -gcnew
OC	Displays the current size of old area in KB	-gc -gccapacity -gcold -gcoldcapacity
OU	Displays the current usage of old area in KB	-gc -gcold
PC	Displays the current size of permanent area in KB	-gc -gccapacity -gcold -gcoldcapacity - gcpermcapacity
PU	Displays the current usage of permanent area in KB	-gc -gcold
YGC	The number of GC event occurred in young area	-gc -gccapacity -gcnew -gcnewcapacity -gcold -gcoldcapacity - gcpermcapacity -gcutil -gccause
YGCT	The accumulated time for GC operations for Yong area	-gc -gcnew -gcutil -gccause

Column	Description	Jstat Option
FGC	The number of full GC event occurred	-gc -gccapacity -gcnew -gcnewcapacity -gcold -gcoldcapacity - gcpermcapacity -gcutil -gccause
FGCT	The accumulated time for full GC operations	-gc -gcold -gcoldcapacity - gcpermcapacity -gcutil -gccause
GCT	The total accumulated time for GC operations	-gc -gcold -gcoldcapacity - gcpermcapacity -gcutil -gccause
NGCMN	The minimum size of new area in KB	-gccapacity -gcnewcapacity
NGCMX	The maximum size of max area in KB	-gccapacity -gcnewcapacity
NGC	The current size of new area in KB	-gccapacity -gcnewcapacity
OGCMN	The minimum size of old area in KB	-gccapacity -gcoldcapacity
OGCMX	The maximum size of old area in KB	-gccapacity -gcoldcapacity
OGC	The current size of old area in KB	-gccapacity -gcoldcapacity
PGCMN	The minimum size of permanent area in KB	-gccapacity - gcpermcapacity
PGCMX	The maximum size of permanent area in KB	-gccapacity - gcpermcapacity
PGC	The current size of permanent generation area in KB	-gccapacity - gcpermcapacity
PC	The current size of permanent area in KB	-gccapacity - gcpermcapacity
PU	The current usage of permanent area in KB	-gc -gcold
LGCC	The cause for the last GC occurrence	-gccause
GCC	The cause for the current GC occurrence	-gccause

Column	Description	Jstat Option
TT	Tenuring threshold. If copied this amount of times in young area (S0 ->S1, S1->S0), they are then moved to old area.	-gcnew
MTT	Maximum Tenuring threshold. If copied this amount of times inside young arae, then they are moved to old area.	-gcnew
DSS	Adequate size of survivor in KB	-gcnew

The advantage of **jstat** is that it can always monitor the GC operation data of Java applications running on local/remote machine, as long as a console can be used. From these items, the following result is output when **–gcutil** is used. At the time of GC tuning, pay careful attention to **YGC, YGCT, FGC, FGCT** and **GCT**.

```
1   S0      S1      E       O       P       YGC     YGCT    FGC     FGCT    GCT
2   0.00    66.44   54.12   10.58   86.63   217     0.928   2       0.067   0.995
3   0.00    66.44   54.12   10.58   86.63   217     0.928   2       0.067   0.995
4   0.00    66.44   54.12   10.58   86.63   217     0.928   2       0.067   0.995
```

These items are important because they show how much time was spent in running GC.
In this example, **YGC** is 217 and **YGCT** is 0.928. So, after calculating the arithmetical average, you can see that it required about *4 ms* (0.004 seconds) for each young GC. Likewise, the average full GC time us *33ms*.
But the arithmetical average often does not help analyzing the actual GC problem. This is due to the severe deviations in GC operation time. (In other words, if the average time is *0.067 seconds* for a full GC, one GC may have lasted 1 ms while the other one lasted *57 ms*.) In order to check the individual GC time instead of the arithmetical average time, it is better to use **-verbosegc**.

-verbosegc

-verbosegc is one of the JVM options specified when running a Java application. While *jstat* can monitor any JVM application that has not specified any options, **-verbosegc** needs to be specified in the beginning, so it could be seen as an unnecessary option (since jstat can be used instead). However, as **-verbosegc** displays easy to understand output results whenever a GC occurs, it is very helpful for monitoring rough GC information.

	jstat	-verbosegc
Monitoring Target	Java application running on a machine that can log in to a terminal, or a remote Java application that can connect to the network by using jstatd	Only when -verbogc was specified as a JVM starting option
Output information	Heap status (usage, maximum size, number of times for GC/time, etc.)	Size of ew and old area before/after GC, and GC operation time
Output Time	Every designated time	Whenever GC occurs
Whenever useful	When trying to observe the changes of the size of heap area	When trying to see the effect of a single GC

The followings are other options that can be used with **-verbosegc**.

- -XX:+PrintGCDetails
- -XX:+PrintGCTimeStamps
- -XX:+PrintHeapAtGC
- -XX:+PrintGCDateStamps (from JDK 6 update 4)

If only **-verbosegc** is used, then **-XX:+PrintGCDetails** is applied by default. Additional options for **–verbosgc** are not exclusive and can be mixed and used together.
When using **-verbosegc**, you can see the results in the following format whenever a minor GC occurs.

```
[GC [<collector>: <starting occupancy1> -> <ending occupancy1>, <pause time1> secs] <sta
occupancy3>, <pause time3> secs]
```

Collector	Name of Collector Used for minor gc
starting occupancy1	The size of young area before GC
ending occupancy1	The size of young area after GC
pause time1	The time when the Java application stopped running for minor GC
starting occupancy3	The total size of heap area before GC
ending occupancy3	The total size of heap area after GC
pause time3	The time when the Java application stopped running for overall heap GC, including major GC

This is an example of **-verbosegc** output for **minor GC**:

```
1   S0     S1     E      O      P         YGC     YGCT    FGC    FGCT     GCT
2   0.00   66.44  54.12  10.58  86.63     217     0.928     2    0.067    0.995
3   0.00   66.44  54.12  10.58  86.63     217     0.928     2    0.067    0.995
4   0.00   66.44  54.12  10.58  86.63     217     0.928     2    0.067    0.995
```

This is the example of output results after an **Full GC** occurred.

```
1   [Full GC [Tenured: 3485K->4095K(4096K), 0.1745373 secs] 61244K->7418K(63104K), [Perm : 10756
sys=0.00, real=0.19 secs]
```

If a CMS collector is used, then the following CMS information can be provided as well.

As **-verbosegc** option outputs a log every time a GC event occurs, it is easy to see the changes of the heap usage rates caused by GC operation.

(Java) VisualVM + Visual GC

Java Visual VM is a GUI profiling/monitoring tool provided by Oracle JDK.

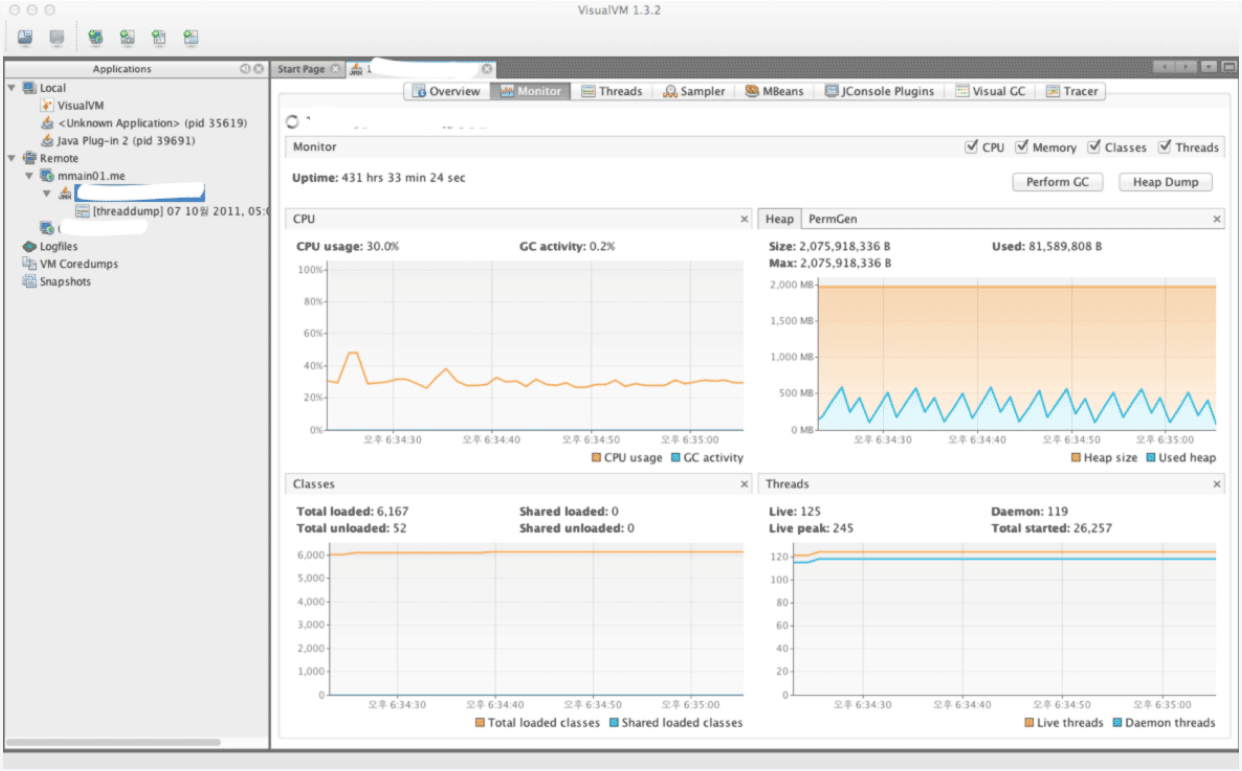


Figure 1: VisualVM Screenshot.

Instead of the version that is included with JDK, you can download Visual VM directly from its website. For the sake of convenience, the version included with JDK will be referred to as Java VisualVM (jvisualvm), and the version available from the website will be referred to as Visual VM (visualvm). The features of the two are not exactly identical, as there are slight differences, such as when installing plug-ins. Personally, I prefer the Visual VM version, which can be downloaded from the website.

After running Visual VM, if you select the application that you wish to monitor from the window on the left side, you can find the "*Monitoring*" tab there. You can get the basic information about GC and Heap from this Monitoring tab.

Though the basic GC status is also available through the basic features of VisualVM, you cannot access detailed information that is available from either **jstat** or **-verbosegc** option.

If you want the detailed information provided by jstat, then it is recommended to install the Visual GC plug-in.

Visual GC can be accessed in real time from the *Tools* menu.

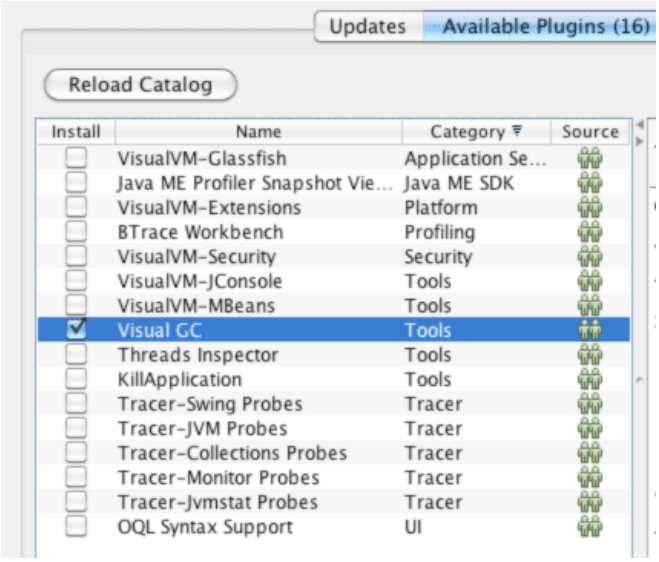


Figure 2: Viusal GC Installation Screenshot.

By using Visual GC, you can see the information provided by running **jstatd** in a more intuitive way.

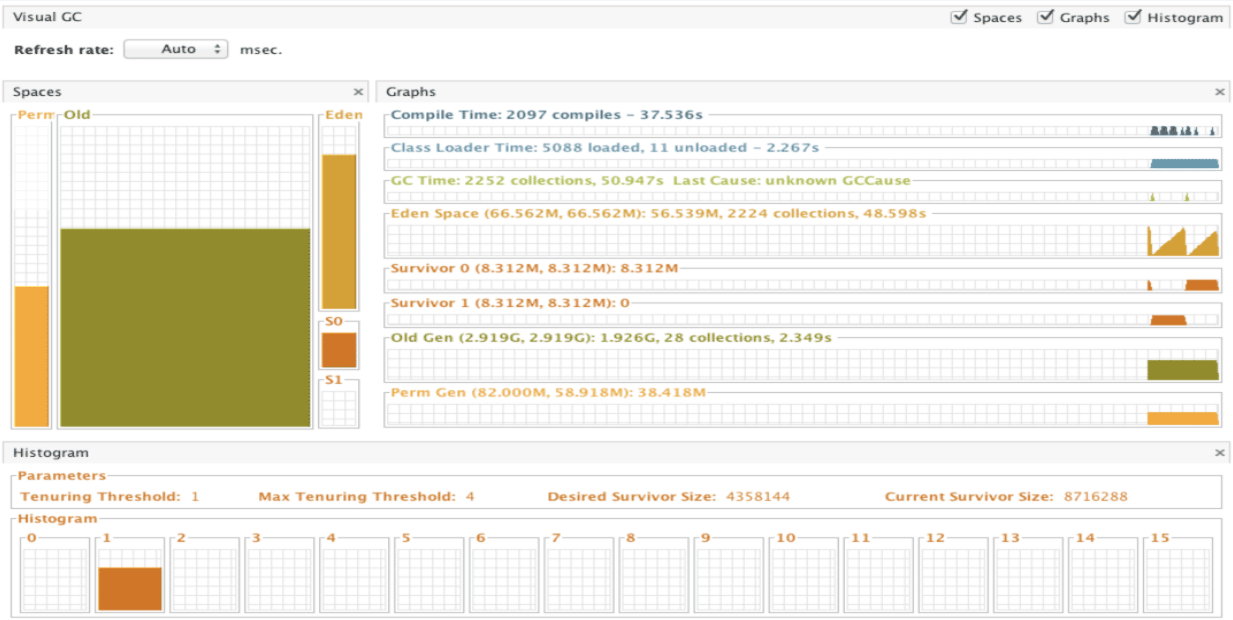


Figure 3: Visual GC execution screenshot.

HPJMeter

HPJMeter is convenient for analyzing **-verbosegc** output results. If Visual GC can be considered as the GUI equivalent of *jstat*, then HPJMeter would be the GUI equivalent of *-verbosgc*. Of course, GC analysis is just one of the many features provided by HPJMeter. HPJMeter is a performance monitoring tool developed by HP. It can be used in HP-UX, as well as Linux and MS Windows.

Originally, a tool called **HPTune** used to provide the GUI analysis feature for **-verbosegc**. However, since the HPTune feature has been integrated into HPJMeter since version 3.0, there is no need to download HPTune separately.

When executing an application, the **-verbosegc** output results will be redirected to a separate file.

You can open the redirected file with HPJMeter, which allows faster and easier GC performance data analysis through the intuitive GUI.

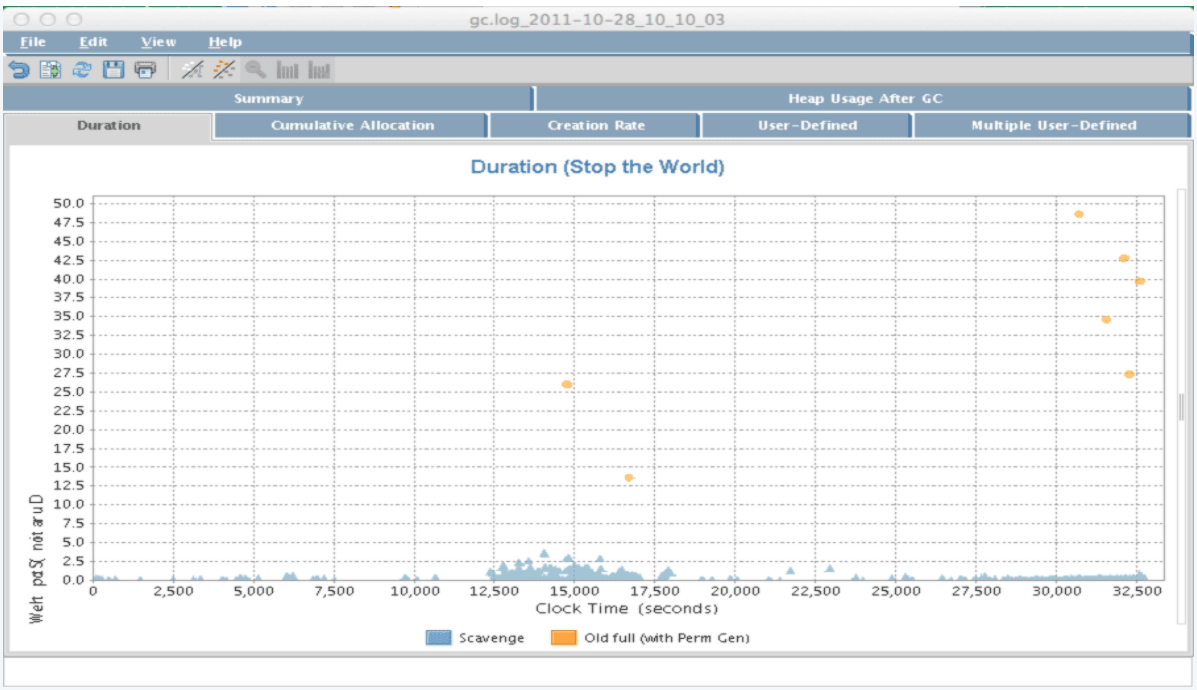


Figure 4: HPJMeter.

What is the Next Article About?

In this article I focused on *how to monitor GC operation information*, as the preparation stage for GC tuning. From my personal experience, I suggest using **jstat** to monitor GC operation, and if you feel that it takes too lmuch time to execute GC, then try **-verbosegc** option to analyze GC. The general GC tuning process is *to analyze the results after applying the changed GC options* after the **-verbosegc** option has been applied based on the analysis. In the next article, we will see the best options for executing GC tuning by using real cases as our examples.