# [How Garbage Collection works in Java](http://javarevisited.blogspot.com/2011/04/garbage-collection-in-java.html" \o "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](http://javarevisited.blogspot.sg/2013/01/threadlocal-memory-leak-in-java-web.html" \t "_blank) 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](http://javarevisited.blogspot.com/2011/05/java-heap-space-memory-size-jvm.html) and both heap and method area is shared between different thread.  
  
2) Garbage collection is a mechanism provided by Java Virtual Machine to reclaim heap space from objects which are eligible for Garbage collection.  
  
3) Garbage collection relieves Java programmer from memory management which is essential part of C++ programming and gives more time to focus on business logic.  
  
4) Garbage Collection in Java is carried by a daemon thread called Garbage Collector.  
  
5) Before removing an object from memory garbage collection thread invokes finalize() method of that object and gives an opportunity to perform any sort of cleanup required.  
  
6) You as Java programmer can not force garbage collection in Java; it will only trigger if JVM thinks it needs a garbage collection based on Java heap size.  
  
7) There are methods like System.gc() and Runtime.gc() which is used to send request of Garbage collection to JVM but it’s not guaranteed that garbage collection will happen.  
  
8) If there is no memory space for creating new object in Heap Java Virtual Machine throws OutOfMemoryError or [java.lang.OutOfMemoryError heap space](http://javarevisited.blogspot.com/2011/05/java-heap-space-memory-size-jvm.html)  
  
9) J2SE 5(Java 2 Standard Edition) adds a new feature called Ergonomics goal of ergonomics is to provide good performance from the JVM with minimum of command line tuning.

## When an Object becomes Eligible for Garbage Collection

An object becomes eligible for Garbage collection or GC if its not reachable from any live threads or 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](http://javarevisited.blogspot.sg/2014/03/difference-between-weakreference-vs-softreference-phantom-strong-reference-java.html) 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](http://javarevisited.blogspot.com/2010/10/why-string-is-immutable-in-java.html). There are many opinions around whether garbage collection in Java happens in perm area of Java heap or not, as per my knowledge this is something which is JVM dependent and happens at least in Sun's implementation of JVM. You can also try this by just creating millions of String and watching for Garbage collection or OutOfMemoryError.

### Types of Garbage Collector in Java

Java Runtime (J2SE 5) provides various types of Garbage collection in Java which you can choose based upon your application's performance requirement. Java 5 adds three additional garbage collectors except serial garbage collector. Each is generational garbage collector which has been implemented to increase throughput of the application or to reduce garbage collection pause times.  
  
1) Throughput Garbage Collector: This garbage collector in Java uses a parallel version of the young generation collector. It is used if the -XX:+UseParallelGC option is passed to the runtime via [JVM command line options](http://javarevisited.blogspot.sg/2011/11/hotspot-jvm-options-java-examples.html) . 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 parametersNewSize 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.

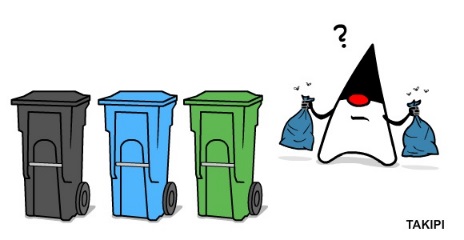
### Summary on Garbage collection in Java

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/>

[](http://www.takipiblog.com/wp-content/uploads/2014/09/Blog_Trash-1.jpg)

**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](http://java.dzone.com/articles/java-8-permgen-metaspace) 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

Additional reading –

1. A really great in-depth review of the G1 Collector on [InfoQ](http://www.infoq.com/articles/G1-One-Garbage-Collector-To-Rule-Them-All" \t "_blank).

2. Java performance – The definitive guide. My favorite [book](http://www.amazon.com/Java-Performance-The-Definitive-Guide/dp/1449358454) on Java performance.

3. More about String deduplication on the CodeCentric [blog](https://blog.codecentric.de/en/2014/08/string-deduplication-new-feature-java-8-update-20-2/).

 Takipi shows you when and why your code breaks in production. It detects caught and uncaught exceptions, HTTP and log errors, and gives you the code and variable state when they happened. Get actionable information, solve complex bugs in minutes. Installs in 5-min. Built for production.

<http://java-is-the-new-c.blogspot.com/2013/07/tuning-and-benchmarking-java-7s-garbage.html>

# [Java is the new C](http://java-is-the-new-c.blogspot.com/)

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](http://www.oracle.com/webfolder/technetwork/tutorials/obe/java/gc01/index.html).  
In depth coverage of algorithms and parameters can be found [here](http://blog.ragozin.info/2011/06/understanding-gc-pauses-in-jvm-hotspots.html). 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](http://docs.oracle.com/javase/7/docs/technotes/guides/vm/G1.html). 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.