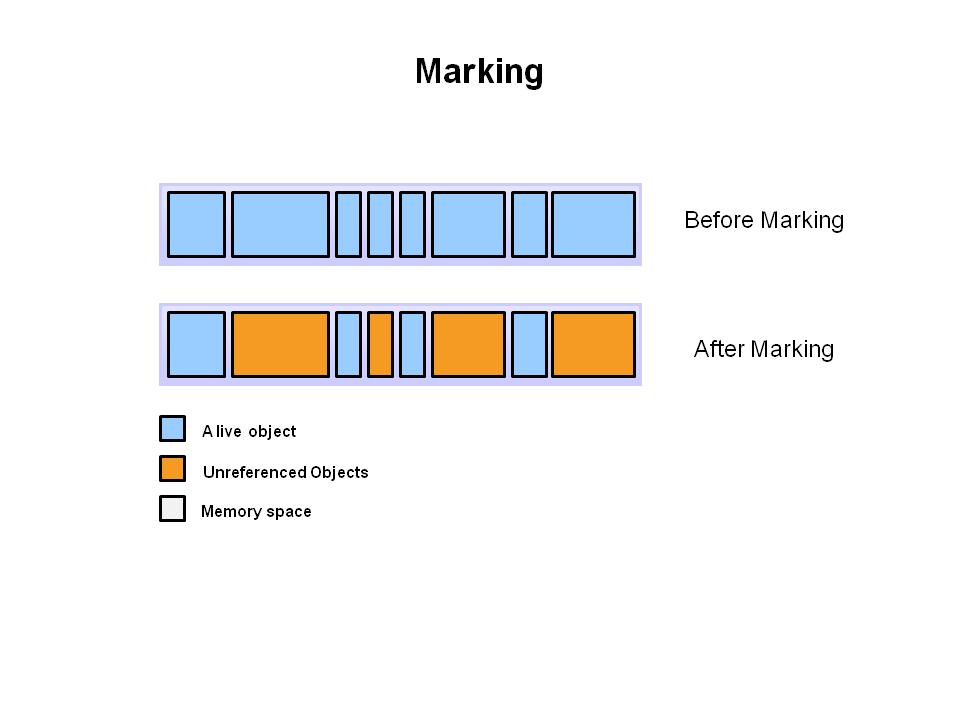
1. **What is GC?**

Automatic garbage collection is the process of looking at heap memory, identifying which objects are in use and which are not, and deleting the unused objects. An in use object, or a referenced object, means that some part of your program still maintains a pointer to that object. An unused object, or unreferenced object, is no longer referenced by any part of your program. So the memory used by an unreferenced object can be reclaimed.

In a programming language like C, allocating and deallocating memory is a manual process. In Java, process of deallocating memory is handled automatically by the garbage collector. The basic process can be described as follows.

#### Step 1: Marking

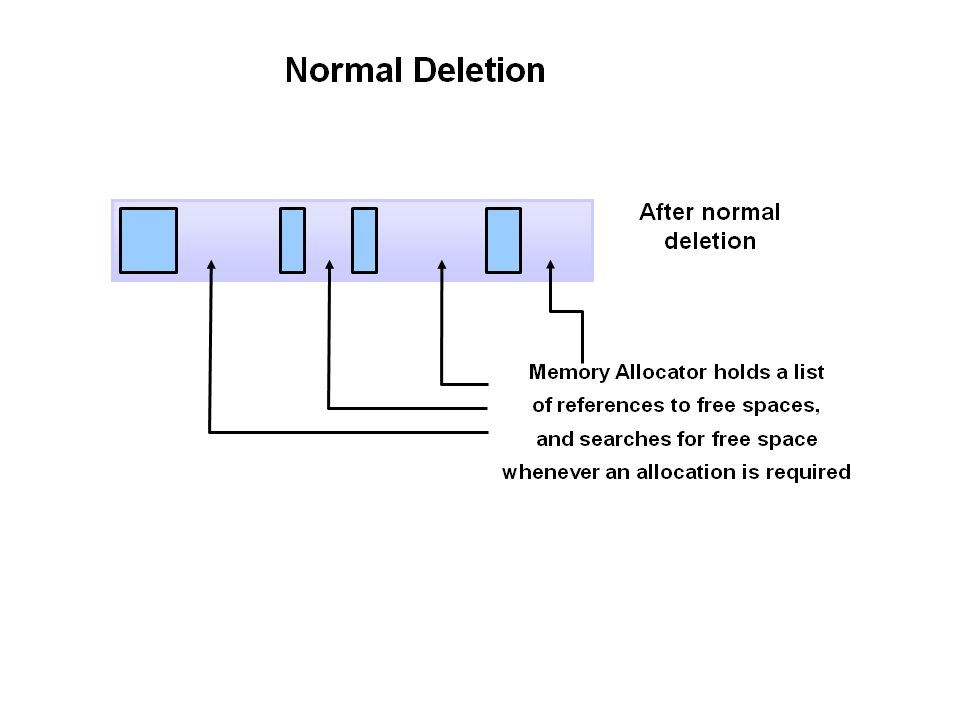
The first step in the process is called marking. This is where the garbage collector identifies which pieces of memory are in use and which are not.



Referenced objects are shown in blue. Unreferenced objects are shown in gold. 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

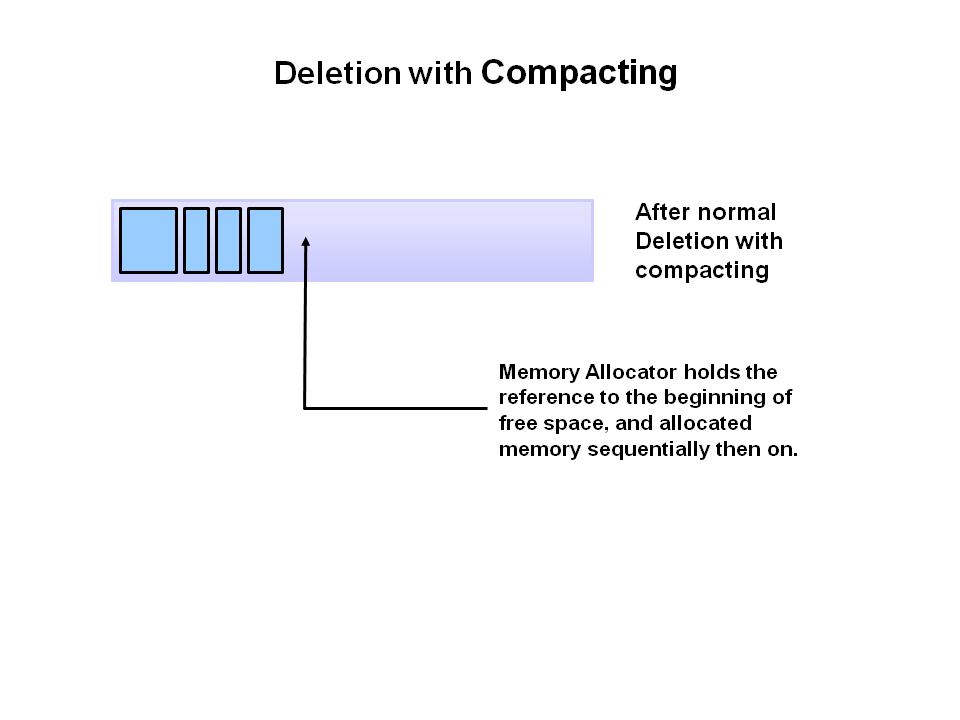
Normal deletion removes unreferenced objects leaving referenced objects and pointers to free space.



The memory allocator holds references to blocks of free space where new object can be allocated.

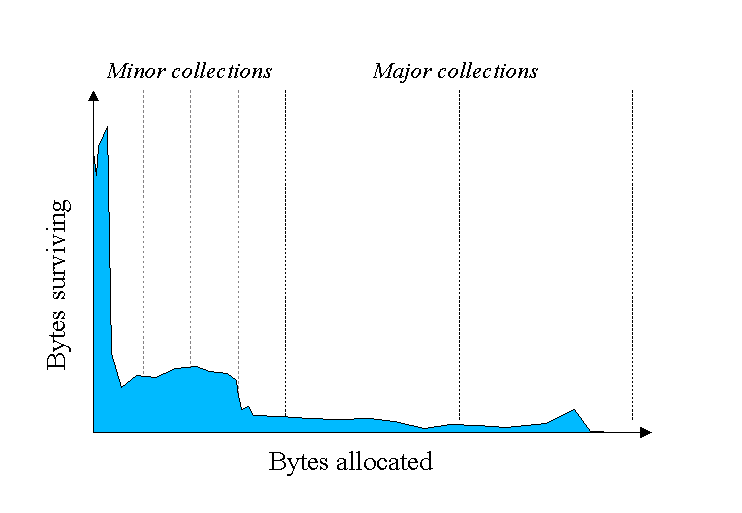
#### Step 2a: Deletion with Compacting

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.



As stated earlier, having to mark and compact all the objects in a JVM is inefficient. As more and more objects are allocated, the list of objects grows and grows leading to longer and longer garbage collection time. However, empirical analysis of applications has shown that most objects are short lived.

Here is an example of such data. The Y axis shows the number of bytes allocated and the X access shows the number of bytes allocated over time.



As you can see, fewer and fewer objects remain allocated over time. In fact most objects have a very short life as shown by the higher values on the left side of the graph.

#### JVM Generations

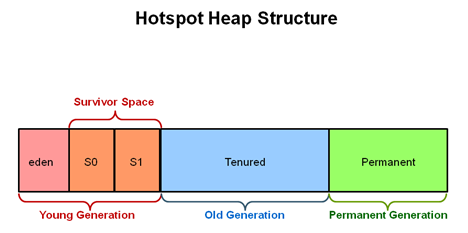
As explained above the basic way of garbage collection involves three steps:

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.

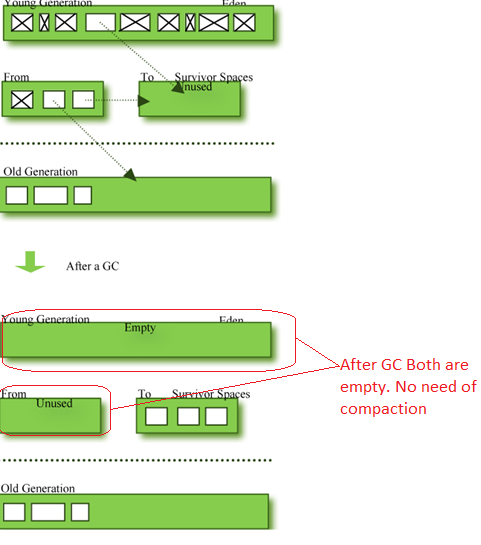
The above shortcomings with the simple approach are the reason that **Java Garbage Collection is Generational** and we have Young Generation, Old or Tenured Generation, and Permanent Generation spaces in the heap memory.



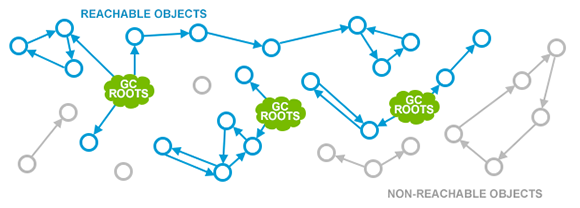
**Young Generation** Young generation is the place where all the new objects are created. When young generation is filled, garbage collection is performed. This garbage collection is called **Minor GC**. Young Generation is divided into three parts – **Eden Memory** and two **Survivor Memory** spaces.

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.
* After minor GC both eden and one of the survivor space is empty.



* The advantage here is that no fragmentation occurs, and thus there is no need for free lists and compaction. Allocations are always fast and the GC algorithm is simple. This strategy is effective only if most objects die, which is the default in the young generation. However, this can lead to a problem when the JVM is executing a high number of concurrent transactions.
* If the young generation is too small, objects are tenured prematurely to the old generation. If the young generation is too large, too many objects are alive (undead) and the GC cycle will take too long.
* 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.



*Live objects are represented as blue on the picture above. When the marking phase finishes, every live object is marked. All other objects (grey data structures on the picture above) are thus unreachable from the GC roots, implying that your application cannot use the unreachable objects anymore. Such objects are considered garbage and GC should get rid of them in the following phases*

**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.

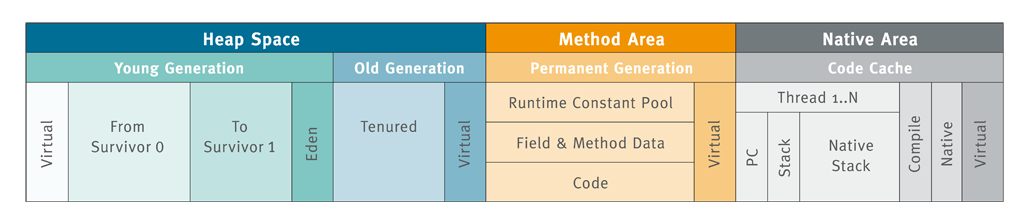
The duration taken by garbage collector depends on the strategy used for garbage collection. That’s why it’s necessary to monitor and tune the garbage collector to avoid timeouts in the highly responsive applications.

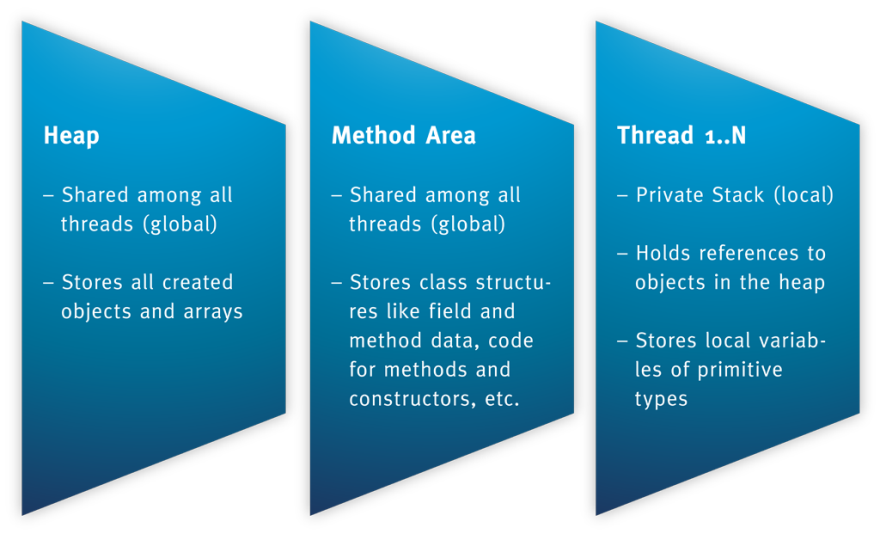
**Old Generation** Old Generation memory contains the objects that are long lived and survived after many rounds of Minor GC. Typically, a threshold is set for young generation object and when that age is met, the object gets moved to the old generation. Usually garbage collection is performed in Old Generation memory when it’s full. Old Generation Garbage Collection is called **Major GC** and usually takes longer time.

***Major garbage collection*** 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.

**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.**





### 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](http://www.oracle.com/technetwork/java/javase/tech/vmoptions-jsp-140102.html).

### 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.

**Serial GC (-XX:+UseSerialGC)**: This collection of garbage collectors uses [**mark-copy**](https://plumbr.eu/handbook/garbage-collection-algorithms/removing-unused-objects/copy) for the Young Generation and [**mark-sweep-compact**](https://plumbr.eu/handbook/garbage-collection-algorithms/removing-unused-objects/compact) for the Old Generation. As the name implies – both of these collectors are single-threaded collectors, incapable of parallelizing the task at hand. Both collectors also trigger stop-the-world pauses, stopping all application threads.

This GC algorithm cannot thus take advantage of multiple CPU cores commonly found in modern hardware. Independent of the number of cores available, just one is used by the JVM during garbage collection.

Enabling this collector for both the Young and Old Generation is done via specifying a single parameter in the JVM startup script:

java -XX:+UseSerialGC com.mypackages.MyExecutableClass

This option makes sense and is recommended only for the JVM with a couple of hundreds megabytes heap size, running in an environment with a single CPU. For the majority of server-side deployments this is a rare combination. Most server-side deployments are done on platforms with multiple cores, essentially meaning that by choosing Serial GC you are setting artificial limits on the use of system resources. This results in idle resources which otherwise could be used to reduce latency or increase throughput.

Serial GC is useful in client-machines such as our simple stand alone applications and machines with smaller CPU. It is good fr somall applications with low memory footprint.

**Parallel GC (-XX:+UseParallelGC)**: This combination of Garbage Collectors uses [**mark-copy**](https://plumbr.eu/handbook/garbage-collection-algorithms/removing-unused-objects/copy) in the Young Generation and [**mark-sweep-compact**](https://plumbr.eu/handbook/garbage-collection-algorithms/removing-unused-objects/compact) in the Old Generation. Both Young and Old collections trigger stop-the-world events, stopping all application threads to perform garbage collection. Both collectors run marking and copying / compacting phases using multiple threads, hence the name ‘Parallel’. Using this approach, collection times can be considerably reduced.

The number of threads used during garbage collection is configurable via the command line parameter *-XX:ParallelGCThreads=NNN*. The default value is equal to the number of cores in your machine.

Selection of Parallel GC is done via the specification of any of the following combinations of parameters in the JVM startup script:

java -XX:+UseParallelGC com.mypackages.MyExecutableClass

java -XX:+UseParallelOldGC com.mypackages.MyExecutableClass

java -XX:+UseParallelGC -XX:+UseParallelOldGC com.mypackages.MyExecutableClass

Parallel Garbage Collector is suitable on multi-core machines in cases where your primary goal is to increase throughput. Higher throughput is achieved due to more efficient usage of system resources:

* during collection, all cores are cleaning the garbage in parallel, resulting in shorter pauses
* between garbage collection cycles neither of the collectors is consuming any resources

On the other hand, as all phases of the collection have to happen without any interruptions, these collectors are still susceptible to long pauses during which your application threads are stopped. So if latency is your primary goal, you should check the next combinations of garbage collectors.

1. **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.

**Concurrent Mark Sweep (CMS) Collector (-XX:+UseConcMarkSweepGC)**: The official name for this collection of garbage collectors is “Mostly Concurrent Mark and Sweep Garbage Collector”. It uses the parallel stop-the-world [mark-copy](https://plumbr.eu/handbook/garbage-collection-algorithms/removing-unused-objects/copy) algorithm in the Young Generation and the mostly concurrent [mark-sweep](https://plumbr.eu/handbook/garbage-collection-algorithms/removing-unused-objects/sweep) algorithm in the Old Generation.

This collector was designed to avoid long pauses while collecting in the Old Generation. It achieves this by two means. Firstly, it does not compact the Old Generation but uses free-lists to manage reclaimed space. Secondly, it does most of the job in the mark-and-sweep phases concurrently with the application. This means that garbage collection is not explicitly stopping the application threads to perform these phases. It should be noted, however, that it still competes for CPU time with the application threads. By default, the number of threads used by this GC algorithm equals to ¼ of the number of physical cores of your machine.

This garbage collector can be chosen by specifying the following option on your command line:

java -XX:+UseConcMarkSweepGC com.mypackages.MyExecutableClass

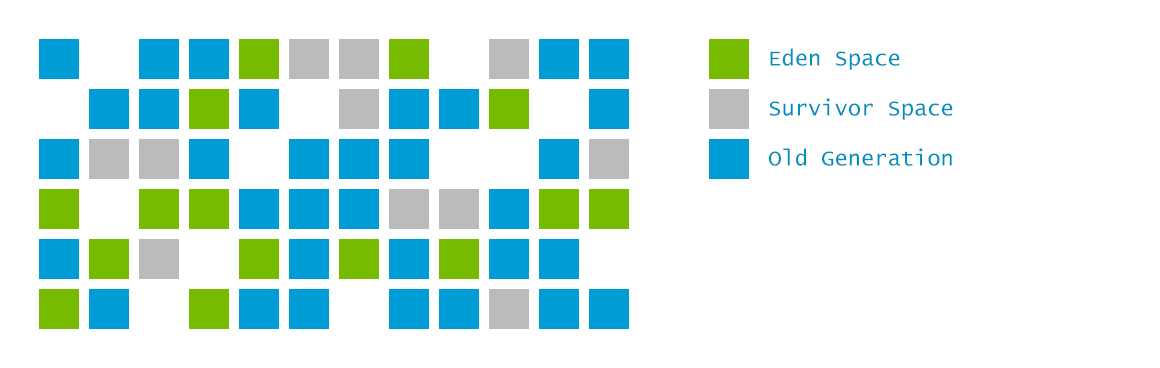
This combination is a good choice on multi-core machines if your primary target is latency. Decreasing the duration of an individual GC pause directly affects the way your application is perceived by end-users, giving them a feel of a more responsive application. As most of the time at least some CPU resources are consumed by the GC and not executing your application’s code, CMS generally often worse throughput than Parallel GC in CPU-bound applications.We can limit the number of threads in CMS collector using -XX:ParallelCMSThreads=n JVM option.

1. **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.

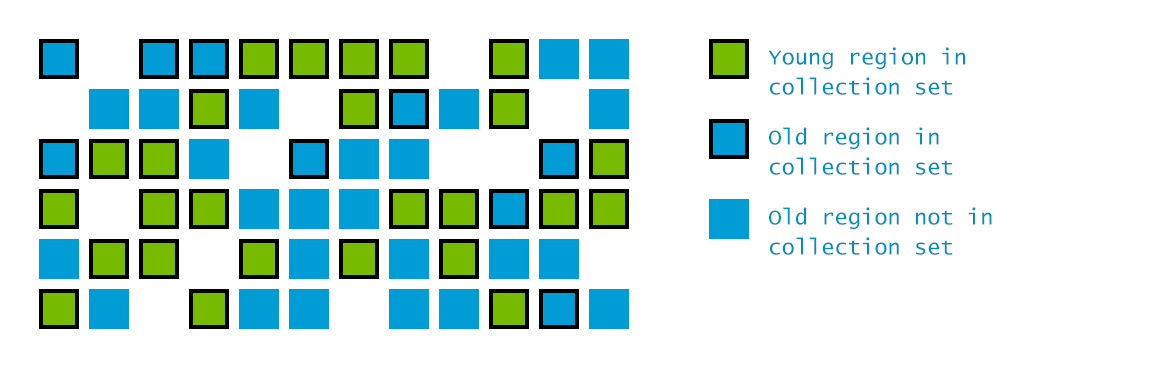
Garbage First Collector doesn’t work like other collectors and there is no concept of Young and Old generation space. It divides the heap space into multiple equal-sized heap regions. When a garbage collection is invoked, it first collects the region with lesser live data, hence “Garbage First”. You can find more details about it at [Garbage-First Collector Oracle Documentation](http://docs.oracle.com/javase/7/docs/technotes/guides/vm/G1.html).

One of the key design goals of G1 was to make the duration and distribution of stop-the-world pauses due to garbage collection predictable and configurable. In fact, Garbage-First is a soft real-time garbage collector, meaning that you can set specific performance goals to it. You can request the stop-the-world pauses to be no longer than x milliseconds within any given y-millisecond long time range, e.g. no more than 5 milliseconds in any given second. Garbage-First GC will do its best to meet this goal with high probability (but not with certainty, that would be hard real-time).

To achieve this, G1 builds upon a number of insights. First, the heap does not have to be split into contiguous Young and Old generation. Instead, the heap is split into a number (typically about 2048) smaller heap regions that can house objects. Each region may be an Eden region, a Survivor region or an Old region. The logical union of all Eden and Survivor regions is the Young Generation, and all the Old regions put together is the Old Generation:



This allows the GC to avoid collecting the entire heap at once, and instead approach the problemincrementally: only a subset of the regions, called the collection set will be considered at a time. All the Young regions are collected during each pause, but some Old regions may be included as well:



Another novelty of G1 is that during the concurrent phase it estimates the amount of live data that each region contains. This is used in building the collection set: the regions that contain the most garbage are collected first. Hence the name: garbage-first collection.

To run the JVM with the G1 collector enabled, run your application as

java -XX:+UseG1GC com.mypackages.MyExecutableClass

|  |  |  |
| --- | --- | --- |
| **Young** | **Tenured** | **JVM options** |
| Serial | Serial | -XX:+UseSerialGC |
| Parallel Scavenge | Parallel Old | -XX:+UseParallelGC -XX:+UseParallelOldGC |
| Parallel New | CMS | -XX:+UseParNewGC -XX:+UseConcMarkSweepGC |
| G1 |  | -XX:+UseG1GC |
| mark-copy | mark-sweep-compact | Serial |
| mark-copy | mark-sweep-compact | Parallel |
| mark-copy | mark-sweep | Concurrent Mark and Sweep |

* Serial GC for both the Young and Old generations
* Parallel GC for both the Young and Old generations
* Parallel New for Young + Concurrent Mark and Sweep (CMS) for the Old Generation
* G1, which encompasses collection of both Young and Old generations

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

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

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

<http://www.dynatrace.com/en/javabook/making-garbage-collection-faster.html>

<https://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/sizing.html>

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

<http://www.dynatrace.com/en/javabook/making-garbage-collection-faster.html>

<https://plumbr.eu/handbook/garbage-collection-algorithms-implementations>

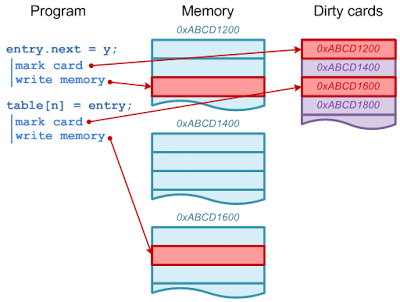
# <http://blog.ragozin.info/2011/06/understanding-gc-pauses-in-jvm-hotspots.html>

# 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)

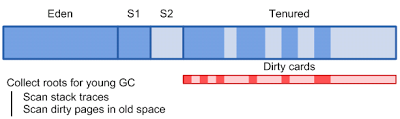
Principle of dirty card write-barrier is very simple. Each time when program modifies reference in memory, it should mark modified memory page as dirty. There is a special card table in JVM and each 512 byte page of memory has associated byte in card table.

[](http://4.bp.blogspot.com/-4BSXuryn7Ss/TeYHvOqwltI/AAAAAAAAI0g/kfH9vN0KgrI/s1600/blog-7.png)

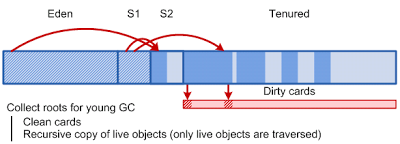
# Young space collection algorithm

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.

[](http://4.bp.blogspot.com/-JeoPvpns3II/TeYILJO5mnI/AAAAAAAAI0k/rIuloiOSZ-I/s1600/blog-8.png)

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.

[](http://3.bp.blogspot.com/-ooT_iuZ694I/TeYIcAQrbgI/AAAAAAAAI0o/x4dy6cb2dyA/s1600/blog-9.png)

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.

[http://3.bp.blogspot.com/-EDGMs03LT80/TpYFZaL3PrI/AAAAAAAAKG8/COdI9A2niNc/s400/blog-9.png](http://3.bp.blogspot.com/-EDGMs03LT80/TpYFZaL3PrI/AAAAAAAAKG8/COdI9A2niNc/s1600/blog-9.png)

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

1. **What is daemon thread?**
2. GC is a Daemon thread. Daemon thread is a low priority thread, which runs intermittently in the back ground doing the garbage collection operation for the java runtime system.
3. **Which method is used to create the daemon thread?**
4. setDaemon method is used to create a daemon thread.

* You can make any java thread as daemon thread. Daemon threads acts like service providers for other threads running in the same process.
* Daemon threads will be terminated by the JVM when there are none of the other threads running, it includs main thread of execution as well.
* To specify that a thread is a daemon thread, call the setDaemon method with the argument true.
* To determine if a thread is a daemon thread, use the accessor method isDaemon.

package com.myjava.threads;

public class DaemonThread extends Thread{

    public DaemonThread(){

        setDaemon(true);

    }

    public void run(){

        System.out.println("Is this thread Daemon? - "+isDaemon());

    }

    public static void main(String a[]){

        DaemonThread dt = new DaemonThread();

        // even you can set daemon constrain here also

        // it is like dt.setDeamon(true)

        dt.start();

    }

}

<http://www.java2novice.com/java_thread_examples/daemon_threads/#sthash.gwM2ZwYK.dpuf>

**What is Heap space in Java?**

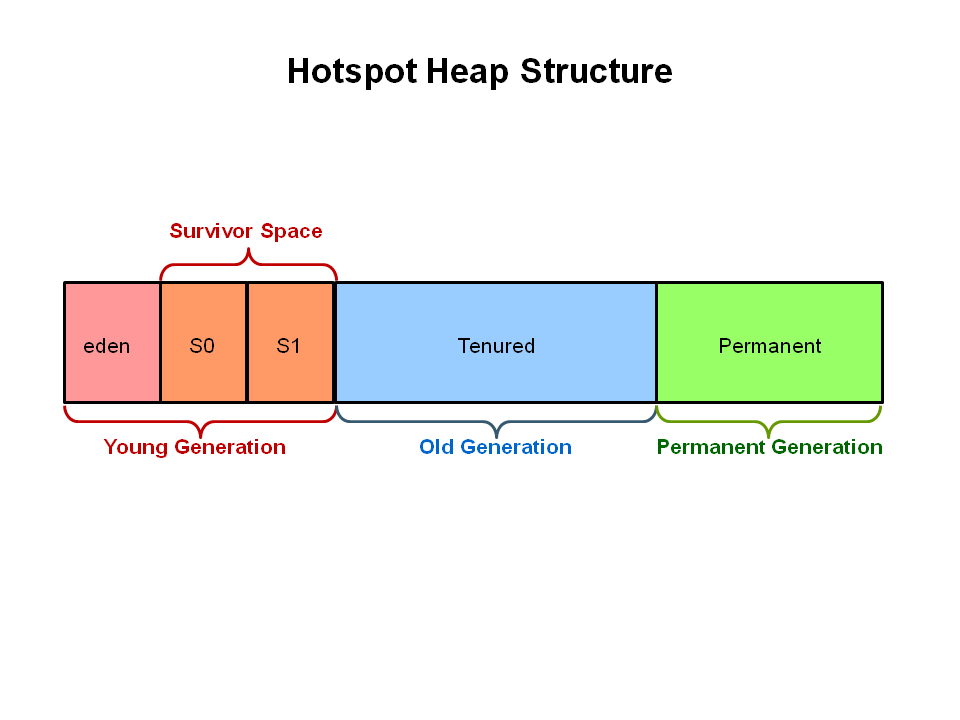
When a Java program started Java Virtual Machine gets some memory from Operating System. Java Virtual Machine or JVM uses this memory for all its need and part of this memory is call java heap memory. Heap in Java generally located at bottom of address space and move upwards. whenever we create object using new operator or by any another means object is allocated memory from Heap and When object dies or garbage collected ,memory goes back to Heap space in Java, to learn more about garbage collection see [how garbage collection works in Java](http://javarevisited.blogspot.com/2011/04/garbage-collection-in-java.html).

**How to increase heap size in Java**

Default size of Heap space in Java is 128MB on most of 32 bit Sun's [JVM](http://javarevisited.blogspot.sg/2011/12/jre-jvm-jdk-jit-in-java-programming.html) but its highly varies from JVM to JVM e.g. default maximum and start heap size for the 32-bit Solaris Operating System (SPARC Platform Edition) is -Xms=3670K and -Xmx=64M and Default values of heap size parameters on 64-bit systems have been increased up by approximately 30%. Also if you are using throughput garbage collector in Java 1.5 default maximum heap size of JVM would be Physical Memory/4 and default initial heap size would be Physical Memory/16. Another way to find default heap size of JVM is to start an application with default heap parameters and monitor in using JConsole which is available on JDK 1.5 onwards, on VMSummary tab you will be able to see maximum heap size.  
  
By the way you can increase size of java heap space based on your application need and I always recommend this to avoid using default JVM heap values. if your application is large and lots of object created you can change size of heap space by using JVM options -Xms and -Xmx. Xms denotes starting size of Heap while -Xmx denotes maximum size of Heap in Java. There is another parameter called -Xmn which denotes Size of new generation of Java Heap Space. Only thing is you can not change the size of Heap in Java dynamically, you can only provide Java Heap Size parameter while starting JVM. I have shared some more useful JVM options related to Java Heap space and Garbage collection on my post [10 JVM options Java programmer must know](http://javarevisited.blogspot.sg/2011/11/hotspot-jvm-options-java-examples.html), you may find useful.  
  
  
**Update:**  
Regarding default heap size in Java, from Java 6 update 18 there are significant changes in how JVM calculates default heap size in 32 and 64 bit machine and on client and server JVM mode:  
  
1) Initial heap space and maximum heap space is larger for improved performance.  
  
2) Default maximum heap space is 1/2 of physical memory of size upto 192 bytes and 1/4th of physical memory for size upto 1Gig. So for 1Gig machine maximum heap size is 256MB 2.maximum heap size will not be used until program creates enough object to fill initial heap space which will be much lesser but at-least 8 MB or 1/64th part of Physical memory upto 1GB.  
  
  
3) for Server Java virtual machine default maximum heap space is 1G for 4GB of physical memory on a 32 bit JVM. for 64 bit JVM its 32G for a physical memory of 128GB.  To learn more about [how much memory you can set in 32-bit and 64-bit JVM](http://javarevisited.blogspot.sg/2013/04/what-is-maximum-heap-size-for-32-bit-64-JVM-Java-memory.html) in various operating system e.g. Windows 8, Linux, or Solaris, see here.

**Java Heap and Garbage Collection**

As we know objects are created inside heap memory and Garbage collection is a process which removes dead objects from Java Heap space and returns memory back to Heap in Java. For the sake of Garbage collection Heap is divided into three main regions named as New Generation, Old or Tenured Generation and Perm space. New Generation of Java Heap is part of Java Heap memory where newly created object are stored, During the course of application many objects created and died but those remain live they got moved to Old or Tenured Generation by Java Garbage collector thread on [Major or full garbage collection](http://javarevisited.blogspot.sg/2011/04/garbage-collection-in-java.html). Perm space of Java Heap is where JVM stores Meta data about classes and methods, String pool and Class level details. You can see How Garbage collection works in Java for more information on Heap in Java and Garbage collection.

[](http://1.bp.blogspot.com/-MoOVJsk6fEc/VJmLWYxfgEI/AAAAAAAACQE/jggByJMXWhw/s1600/Java+Heap.PNG)

**OutOfMemoryError in Java Heap**

When JVM starts JVM heap space is equal to the initial size of Heap specified by -Xms parameter, as application progress more objects get created and heap space is expanded to accommodate new objects. JVM also run garbage collector periodically to reclaim memory back from dead objects. JVM expands Heap in Java some where near to Maximum Heap Size specified by -Xmx and if there is no more memory left for creating new object in java heap , JVM throws java.lang.OutOfMemoryError and your application dies. Before throwing [OutOfMemoryError No Space in Java Heap](http://javarevisited.blogspot.sg/2011/09/javalangoutofmemoryerror-permgen-space.html), JVM tries to run garbage collector to free any available space but even after that not much space available on Heap in Java it results into OutOfMemoryError. To resolve this error you need to understand your application object profile i.e. what kind of object you are creating, which objects are taking how much memory etc. you can use profiler or heap analyzer to troubleshoot OutOfMemoryError in Java. "java.lang.OutOfMemoryError: Java heap space" error messages denotes that Java heap does not have sufficient space and cannot be expanded further while "java.lang.OutOfMemoryError: PermGen space" error message comes when the permanent generation of Java Heap is full, the application will [fail to load a class](http://javarevisited.blogspot.sg/2011/08/classnotfoundexception-in-java-example.html) or to allocate an interned string.

**Java Heap dump**

Java Heap dump is a snapshot of Java Heap Memory at a particular time. This is very useful to analyze or troubleshoot any memory leak in Java or any java.lang.OutOfMemoryError. There are tools available inside JDK which helps you to take heap dump and there are heap analyzer available tool which helps you to analyze java heap dump. You can use "jmap" command to get java heap dump, this will create heap dump file and then you can use "jhat - Java Heap Analysis Tool" to analyze those heap dumps. You should also read [Java Performance The Definitive Guide By Scott Oaks](http://www.amazon.com/Java-Performance-The-Definitive-Guide/dp/1449358454?tag=javamysqlanta-20) to learn more about Java performance tuning and profiling. It is one of the must read Java performance book for any senior Java developers.

**How to increase Java heap space on Maven and ANT**

Many times we need to increase heap size of Maven or ANT because once number of classes increases build tool requires more memory to process and build and often throw OutOfMemoryError which we can avoid by changing or increase heap memory of JVM. For details see my post [How to increase java heap memory for Ant or Maven](http://javarevisited.blogspot.com/2011/05/java-heap-space-memory-size-jvm.html)

**10 Points about Java Heap Space**

1. Java Heap Memory is part of memory allocated to JVM by Operating System.  
     
   2. Whenever we create objects they are created inside Heap in Java.  
     
   3. Java Heap space is divided into three regions or generation for sake of garbage collection called New Generation, Old or tenured Generation or Perm Space. Permanent generation is garbage collected during full gc in hotspot JVM.  
     
   4. You can increase or change size of Java Heap space by using JVM command line option -Xms, -Xmx and -Xmn. don't forget to add word "M" or "G" after specifying size to indicate Mega or Gig. for example you can set java heap size to 258MB by executing following command java -Xmx256m HelloWord.  
     
   5. You can use either JConsole or Runtime.maxMemory(), Runtime.totalMemory(), Runtime.freeMemory() to query about Heap size programmatic in Java. See my post [How to find memory usage in Java program](http://javarevisited.blogspot.sg/2012/01/find-max-free-total-memory-in-java.html) for more details.  
     
   6. You can use command "jmap" to take Heap dump in Java and "jhat" to analyze that heap dump.  
     
   7. Java Heap space is different than Stack which is used to store call hierarchy and local variables.  
     
   8. Java Garbage collector is responsible for reclaiming memory from dead object and returning to Java Heap space.  
     
   9. Don’t panic when you get java.lang.OutOfMemoryError, sometimes its just matter of increasing heap size but if it’s recurrent then look for [memory leak in Java](http://javarevisited.blogspot.sg/2012/01/tomcat-javalangoutofmemoryerror-permgen.html).  
     
   10. Use Profiler and Heap dump Analyzer tool to understand Java Heap space and how much memory is allocated to each object.

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

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

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

### How to get free Memory in Java

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

### How to get total Memory in Java

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

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

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

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

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

### How to get Used Memory in JVM

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

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

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

**public** **class** MemoryUtil{

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

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

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

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

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

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

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

                                         + totalMemory);

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

                                         + maxMemory);

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

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

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

              }

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

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

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

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

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

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

                                         + totalMemory);

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

       }

}

**Output:**

JVM freeMemory: 9

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

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

Used Memory in JVM: 81

freeMemory in JVM: 35

totalMemory in JVM shows current size of java heap : 108

maxMemory in JVM: 116

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

Read more: <http://javarevisited.blogspot.com/2012/01/find-max-free-total-memory-in-java.html#ixzz442gebitD>