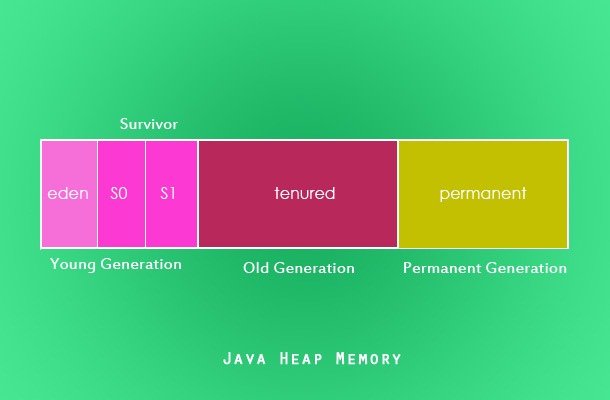
**Java Heap Memory**

[**https://plumbr.io/handbook/what-is-garbage-collection**](https://plumbr.io/handbook/what-is-garbage-collection) **- Good site for GC**

It is essential to understand the role of heap memory in JVM memory model. At runtime the Java instances are stored in the heap memory area. When an object is not referenced anymore it becomes eligible for garbage collector. During garbage collection process, those objects are evicted / removed from heap memory and the space is reclaimed / released. Heap memory has three major areas,

1. Young Generation
   1. Eden Space (any instance enters the runtime memory area through eden)
   2. S0 Survivor Space (older instances moved from eden to S0)
   3. S1 Survivor Space (older instances moved from S0 to S1)
2. Old Generation (instances promoted from S1 to tenured)
3. Permanent Generation (contains meta information like class, method detail)





**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 moves them to the other survivor space. So at a time, one of the survivor spaces 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.

**Memory Management in Java – 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 a 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.

**Java Memory Model – 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](https://www.journaldev.com/4098/java-heap-space-vs-stack-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.

**Java Memory Model – 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.

**Java Memory Model – Memory Pool**

Memory Pools are created by JVM memory managers to create a pool of [immutable](https://www.journaldev.com/129/how-to-create-immutable-class-in-java) objects if the 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.

**Java Memory Model – 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 part of the method area.

**Java Memory Model – 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 is getting referred from the method. You should read [Difference between Stack and Heap Memory](https://www.journaldev.com/4098/java-heap-space-vs-stack-memory).

**Memory Management in Java – 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).

**Memory Management in Java – Java Garbage Collection Type**

Java has **four types of garbage collectors**, According to JDK 7, there are 5 GC types.

1. [Serial Garbage Collector](https://javapapers.com/java/types-of-java-garbage-collectors/#serial-garbage-collector)
2. [Parallel Garbage Collector](https://javapapers.com/java/types-of-java-garbage-collectors/#parallel-garbage-collector)
3. Parallel Old GC (Parallel Compacting GC)
4. [CMS Garbage Collector](https://javapapers.com/java/types-of-java-garbage-collectors/#cms-garbage-collector)
5. [G1 Garbage Collector](https://javapapers.com/java/types-of-java-garbage-collectors/#g1-garbage-collector)

Serial collector / Serial GC (Garbage collector) in java

Features of Serial GC (Garbage collector) in java >

* Serial collector is also called Serial GC (Garbage collector) in java.
* Serial collector is simply also called Serial collector in java.
* Serial GC (Garbage collector) is rarely used in java.
* Serial GC (Garbage collector) is designed for the single threaded environments in java.
* In Serial GC (Garbage collector), both minor and major garbage collections are done serially by one thread (using a single virtual CPU) in java.
* Serial GC 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 in java.

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

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).
4. All garbage collection events are conducted serially in one thread. Compaction is executed after each garbage collection.

Vm (JVM) option for enabling serial GC (garbage Collector) in java

-XX:+UseSerialGC

Example of Passing Serial GC in Command Line for starting jar>

Java -Xms256m -Xms512m  -XX:+UseSerialGC -jar d:\MyJar.jar

When to Use the Serial GC (garbage Collector) in java >

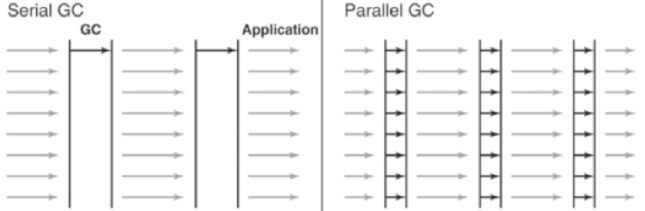
* The Serial GC is the garbage collector of choice for most applications that do not have low pause time requirements and run on client-style machines. It takes advantage of only a single virtual processor for garbage collection work in java.
* Serial GC (garbage collector) is also popular in environments where a high number of JVMs are run on the same machine. 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 in java.

**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 ways 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 reclaims 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.

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



**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**."

What’s the difference between the Serial and the Parallel collector?

Both the serial and parallel collectors cause a stop-the-world during the GC.  
So what's the difference between them?  
A serial collector is a default copying collector which uses only one GC thread for the GC operation, while a parallel collector uses multiple GC threads for the GC operation.

**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 little more complicated steps.

**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.

<https://www.javamadesoeasy.com/2017/03/top-50-garbage-collection-interview.html>

**Concurrent Mark Sweep (CMS) Collector/concurrent low pause collector in java?**

[**Concurrent Mark Sweep Collector**](http://www.javamadesoeasy.com/2016/10/concurrent-mark-sweep-cms-collector.html) **is also called**

* + concurrent low pause collector
  + concurrent low pause GC (garbage collector)
  + CMS GC (garbage Collector)
  + CMS Collector
  + concurrent low pause collector
  + concurrent low pause GC (garbage collector)
* Concurrent Mark Sweep (CMS) collector collects the old/tenured generation in java.
* Concurrent Mark Sweep (CMS) Collector minimizes the pauses by doing most of the garbage collection work concurrently with the application threads in java.
* Concurrent Mark Sweep (CMS) Collector on live objects >

Concurrent Mark Sweep (CMS) 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 in java.

### When to Use the Concurrent Low Pause Collector in java

* Concurrent Low Pause Collector should be used if your applications that require low garbage collection pause times in java.
* Concurrent Low Pause Collector should be used when your application can afford to share processor resources with the garbage collector while the application is running in java.
* Concurrent Low Pause Collector is beneficial to applications which have a relatively large set of long-lived data (a large tenured generation) and run on machines with two or more processors in java.

Examples when to use Concurrent Mark Sweep (CMS) collector / concurrent low pause collector should be used for >

Example 1 - Desktop UI application that respond to events,

Example 2 - Web server responding to a request and

Example 3 - Database responding to queries.

Vm (JVM) option for enabling Concurrent Mark Sweep (CMS) Collector in java >

-XX:+UseConcMarkSweepGC

Example of using Concurrent Mark Sweep (CMS) collector / concurrent low pause collector in Command Line for starting jar>

java -Xms256m -Xms512m  -XX:+UseConcMarkSweepGC -jar d:\MyJar.jar

G1 **Garbage Collector (-XX:+UseG1GC)**:

The Garbage First or G1 garbage collector is available from Java 7 and its 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”.

We can find more details about it at [Garbage-First Collector Oracle Documentation](http://docs.oracle.com/javase/7/docs/technotes/guides/vm/G1.html).

### Memory Management in Java – Java Garbage Collection Monitoring

### Java Garbage Collection Best Practices

For many simple applications, Java garbage collection is not something that a programmer needs to consciously consider. However, for programmers who want to advance their Java skills, it is important to understand how Java garbage collection works and the ways in which it can be tuned.

Besides the basic mechanisms of garbage collection, one of the most important points to understand about garbage collection in Java is that it is non-deterministic, and there is no way to predict when garbage collection will occur at run time. It is possible to include a hint in the code to run the garbage collector with the System.gc() or Runtime.gc() methods, but they provide no guarantee that the garbage collector will actually run.

The best approach to tuning Java garbage collection is setting flags on the JVM. Flags can adjust the garbage collector to be used (e.g. Serial, G1, etc.), the initial and maximum size of the heap, the size of the heap sections (e.g. Young Generation, Old Generation), and more. The nature of the application being tuned is a good initial guide to settings. For example, the Parallel garbage collector is efficient but will frequently cause “stop the world” events, making it better suited for backend processing where long pauses for garbage collection are acceptable.

On the other hand, the CMS garbage collector is designed to minimize pauses, making it ideal for GUI applications where responsiveness is important. Additional fine-tuning can be accomplished by changing the size of the heap or its sections and measuring garbage collection efficiency using a tool like jstat.

### 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 🙂

<https://blog.overops.com/garbage-collectors-serial-vs-parallel-vs-cms-vs-the-g1-and-whats-new-in-java-8/>

<https://www.baeldung.com/java-permgen-metaspace>