**Java Garbage Collection**

**Factors affecting GC**

1. **Memory- Amount of heap assigned**
2. **Latency - Number of pauses for GC**
3. **Throughput - Time spent running the code vs time spent on GC**

1. Marking

Objects which are no longer referenced are marked

2. Deletion with compaction

The marked objects are deleted and remaining objects are compacted together

The heap is divided into

**Young Generation**

**Tenured Generation**

Permanent generation **(No perm gen from Java 8 Metaspace will be present)**

When young generation space is full Minor GC runs.

Objects in young gen which reach particular age are moved to tenured gen

The collection in tenured gen is Major GC

In Minor GC and Major GC all application threads are stopped until operation completes

Permanent gen contains meta-data required by JVM to describe classes and methods, Library classes and methods may also reside here

It is also included in full garbage collection

**Young Generation (Mark and Copy)**

New objects are allocated to Eden space, S0 and S1 are empty

When Eden space is full Minor GC runs

Referenced objects are moved to S0 and unreferenced objects are deleted

At the next minor GC unreferenced objects are deleted and referenced objects are moved to S1, objects in S0 are also moved to S1, S0 and Eden space are cleared

At next GC same thing happens but instead of S1 objects are moved to S0 and S1 and Eden space are cleared

After minor GC when objects reach certain age they are moved to tenured gen

**Old generation (Mark and Sweep)**

When then tenured space becomes full major GC will be performed on tenured gen

All live objects are marked

The unreferenced objects are reclaimed or swept

The live objects are moved together so the new objects are allocated continuous memory locations (compaction)

**Metaspace**

GC is triggered in Metaspace when it reaches the MaxMetaspaceSize, if this parameter is not specified then there is dynamic resizing

|  |  |
| --- | --- |
| -Xmx | Max heap size |
| -Xms | Initial heap size when JVM starts |
| -Xmn | Young gen size |
| -XX:PermiSize | Initial Perm gen size |
| -XX:MaxPermSize | Maximum Perm gen size |
| -XX:MaxMetaspaceSize | Set maximum meta space for a class |

1. **Serial GC** Minor and Major GC runs sequentially, uses marked compact method (old memory are moved to beginning of head so new memory can be allocated in

continuous chunks)

Applicable for small memory, and where CPU is shared

Application threads are paused when GC is running

Single thread for garbage collection

High pause time

-XX:+UseSerialGC

2. **Parallel GC (Default in Java8)**

Uses multiple threads for GC

Application threads are paused when GC is running

High pause time

High CPU utilization

-XX:+UseParallelGC

Multithreaded young gen collector and single threaded old gen collector

-XX:+UseParallelOldGC (**default**)

Multithreaded young gen and old gen collector

It also performs compacting

Application is paused during GC's

Can specify maximum GC threads, max GC pause times (between 2 GC's), throughput (time spent GC vs time spent other than GC)

3. **CMS (Concurrent Mark Sweep) garbage collector**

-XX:+UseParNewGC

Uses multiple threads for GC by sharing processor, shorter GC pauses, application are slower

It will cause the application threads to stop (STW) when the application has changed the heap or collection is happening in old generation

Does compaction during STW (Stop the World)

Low pause time

High CPU utilization

 Calling System.gc() may cause Concurrent Mode failure

4. **The G1(Garbage First) Garbage Collector** **(Default Java9)**

-XX:+UseG1GC

Applicable for heap size > 4GB and multiprocessor env

Heap size is divided into equal sized heap regions

Performs concurrent marking across the heap regions (Phase-1 marking), scans regions having most garbage objects

It performs sweep in all the marked areas (Phase-2 sweeping)

Compaction is on the go

Possible to specify the maximum pause time

Low pause time

High CPU utilization

**5. Epsilon garbage collector**

Passive garbage collector, does not deallocate memory

If process runs out of heap memory JVM exit

Use to monitor application performance

No pause time and No CPU utilization

In order to reduce the memory use by creating multiple copies of same string

-XX:+UseStringDeduplication

This removes the multiple copies of same string and maintains global single char[]

**finalize ()**

Method is called before object is ready for garbage collection. May/may not get called

Called is only once for an object

Separate daemon thread(finalizer) queues finalization queue which contains objects having finalize () method and calls the finalize () method on the object

JVM makes entry of objects having finalize method

When the object is ready for garbage collection then GC will remove the entry and add it to the finalization queue otherwise object is garbage collected

Finalizer thread will poll the queue and call the finalize () method, if object still has no reference then GC is done

**OutOfMemoryError**

1. Java heap space: Less heap size for the process (Xmx) or excessive use of finalize () method

Jconsole used to monitor objects pending for finalization

1. GC overhead limit exceeded - Java processes is doing GC 98% of time and recovering only 2% heap space for 5 continuous GC cycles
2. Permgen space is thrown - Perm gen space is exhausted Not possible from Java 8
3. Metaspace - If meta space for class is exhausted
4. Requested Array size exceeds VM limit - Allocate array size exceed heap size