KEY TERMS IN GARBAGE COLLECTION

→ Factors to consider :-

*. Memory :- → HEAP SIZE
→ GC footprint

*. Throughput :- How much time actual app runs
vs.
How much time GC runs.

*. Latency :- Application Pause Time when GC runs.
→ STW (STOP-THE-WORLD Event)*

⇒ GC Process :- ① Mark :- → Traverse object graph from root and mark live objects (Reachable)

② Deletion (Sweep) :- Delete all the unused/unreachable objects

③ Compaction :- After deletion, survived objects are moved together
→ heap traversed to find empty spaces between live objects & to be made available for memory allocation.

⇒ Generational Hypothesis :- (Infant Mortality)
→ YOUNG objects much likely to die than OLD Gen.
↳ Monitor Young gen frequently *(Minor GC)

→ Objects survived many round of minor GC moved to OLD gen.
↳ After certain threshold, GC happened in OLD gen. ⇒ *(Major GC)

GARBAGE COLLECTION

→ Mark & Copy :- Young Gen
- EDEN
- Survivor spaces
  - S0
  - S1

→ Live objects during minor GC copied from Eden space to one of the survivor space.

→ Now Eden space has :-
- copied objects
- Unused / Unreferenced objects

→ Cleanup happened at Eden space

→ COMPACTION :- To move all the live objects together to order/organize the free-up space.

↳ copy all the objects and updating the memory references

↳ Increase the App. Pause Time*

↳ Survivor spaces meant to be used for compaction.

→ Lost Object Problem :-

→ If App. thread & GC thread running side-by-side there may be objects identified for GC but when GC pauses, app. thread may create a new pointer from GC identified object.

↳ We will loose the latest object reference to be processed.

GARBAGE COLLECTION

→ Tri-Color Algorithm :- Objects inside heap can be in 3 states.

→ BLACK (completed Processing)
→ No pointers to any objects in White Set

→ GREY (Processing)
→ May have Pointers to White Set

→ WHITE (Not Touched)
→ Possible Accessible from Roots (candidate for collection)

* Constraint :-

IF (object.isBlack() &&
object.pointingReference.isWhite())
{
objectpointingReference.setGrey(true);
to make it a processing node
}

Known as WRITE BARRIER

→ Problem Stmt. :- OLD objects may point to Young Gen objects.

→ Solved through :- Code Injection By JIT (write Barrier)
→ Set Bits to say which part of the OLD Regions are dirty.
→ happens during write operation.

→ Data Structure :- CARD TABLE

IF (object.pointingReference.isYoungGen())
{ cardTable [current object Ref] = 1;
↳ Find old Gen Regions
}

GARBAGE COLLECTION

① Serial GC :- → GC uses single thread for both minor and major GC
→ uses simple mark-compact-sweep approach
→ -XX : + UseSerialGC

② Parallel GC :- spawns 'N' threads for young gen GC
→ Doesnot run concurrently with application
→ when GC runs, the GC threads run parallely.
→ -XX : UseParallelGC
-XX : ParallelGCThreads = n

③ Parallel Old GC :- Multiple Threads for both minor & major GC
→ -XX : UseParallelOldGC

④ Concurrent Mark & Sweep GC :-
→ collector runs concurrently with application to mark live objects
→ Less Latency because less pause time
→ Less Throughput than Parallel GC

⑤ G1 (Garbage first GC) :- → Default in JDK-9
→ Parallel, concurrent & incrementally compact low-pause GC

(... continued)

GARBAGE COLLECTION

→ G1GC :- → Divides whole heap-space into multiple equal-sized heap regions
→ First collects the region with lesser live data
→ -XX:+UseG1GC
→ *maxTargetPauseTime (G1GC try to ensure GC work done only for this much time in one cycle)

$$GC\text{-}Priority = \frac{1}{Object\text{-}Ref\text{-}CNT}$$

→ Data structure :- Remembered set
(Tells about the incoming pointers towards a region/partition)

→ 2048 Regions :- From 1MB to 32MB size

#. STRING DEDUPLICATION :-
→ Enables GC to identify strings which have multiple footprint in heap and modify them to point same internal char[] array.
(-XX:+UseStringDeDuplication)

GARBAGE COLLECTION

⑥ ZGC (Z-Garbage collector) :-

→ Low Latency GC
→ All concurrent operation
→ use to handle heap size from KBs to TB
→ 10 ms App. Pause time (latency)
→ Experimental GC in Java-11 (Linux)
→ Load Barriers with colored pointers to perform concurrent operations.
→ Uses metadata Bits to mark the state of an object
→ divides memory into dynamic regions.
(ZPages)

→ Load Barrier :- Runs when the thread loads an object reference from Heap.
↳ to check whether the loaded object ref. has a bad color.
↳ JIT code Injection.

⑦ Shenandoah GC :- → Ultra Low-Pause GC
→ Runs Garbage collection work concurrently.
→ adds concurrent compaction.
→ use to avoid GC Nepotism **
(Dead Parents (Old gen) can cause children (dead young gen) to get promoted.

** Epsilon GC → only allocates memory and release *.