JVM → Memory Management:

Forms of Garbage Collection:

(1) Do nothing

(2) Reference Counting → Calls two functions AddRef & release
↳ when cnt=0 it would be freed. object & release decrements

(3) Mark & Sweep

(4) Copying on object

(5) Generational → Survives a garbage collect, it would be around for long time

(6) Incremental → it doesn't look at memory during garbage Collect

Mark phase → identifies objects that still in use

Sweep → remove unused objects

Compact phase → to compact the memory

Copying → objects moved from ~~from space~~ → to space

Generational Collectors: Maintain ~~different~~ generations for memory

↳ Long lived objects promoted to different generation.

↳ objects stored in memory under younger Generation undergoes Garbage collect, in which it is Survived it moves into Older Generation.

→ Younger Generation → two Survivor Spaces

- objects Survive GC → moved to Survivor space

↳ only one Survivor Space can be used at a time

→ objects Copied b/w Survivor spaces

Minor Garbage Collection → objects collected from Younger Generation

Major Garbage Collection → when younger & older Generations → full then it happens

↳ Slower than minor

=> Triggered when tenured space is full
1 -> No JVM option to force objects to old space
↳ PretenureSize Threshold = <n>
↳ all objects larger than <n> bytes should be allocated directly in old space
-> Jvm use Thread Local Allocation Buffers
↳ Each thread has its own buffer in Edenspace
↳ No locking Required
live roots -> objects from stack frames, static variables
=> Each write to reference to young object goes through write barrier
-> one Entry per 512 bytes of memory
↳ Minor GC scans cardtable

G1 Collector -> Compacting Collector
↳ Meant for Server applications

Concurrent Mark & Sweep
↳ only Collects old space; causes heap fragmentation
EE -> Eden; O -> old; S1C -> Capacity for Survivor S0U, S1U
OU -> Old Capacity
Utilization
References; Strong -> Soft -> Weak -> phantom
↳ Soft References Collected if there is memory pressure
-> weak will be Collected immediately.
phantom -> refers two different things.
Person person = new Person
WeakReference<Person> wr = new WeakReference<Person>(person);
↳ associate metadata with another type
↳ Use WeakHashMap -> Store weakReference
SoftReference
↳ used for caching
PhantomReference -> Interaction with GC.
-> when Strong reference is cleared then softreference is still available
=> Reference types -> Enqueued to ReferenceQueue.
↳ Poll & remove method
Phantom reference -> least used type. -> Used Instead of Initializers.