Memory management in the JVM

- The JVM manages memory through a process called garbage collection, which continuously identifies and eliminates unused memory in Java programs.
- Garbage collection happens inside a running JVM.

Garbage Collection

- Garbage collection (GC) is the process that aims to free up occupied memory that is no longer referenced by any reachable Java object, and is an essential part of the Java virtual machine's (JVM's) dynamic memory management system.
- The space occupied by previously referenced objects is freed and reclaimed to enable new object allocation.
- Quickly free unreferenced memory in order to satisfy an application's allocation rate so that it doesn't run out of memory.
- Reclaim memory while minimally impacting the performance (e.g., latency and throughput) of a running application.

Two kinds of garbage collection

1. Reference counting
2. Tracing collectors

Reference counting

- Reference counting collectors keep track of how many references are pointing to each Java object.
- Once the count for an object becomes zero, the memory can be immediately reclaimed.
- This immediate access to reclaimed memory is the major advantage of the reference-counting approach to garbage collection.
- Keeping all reference counts up to date can be quite costly

Challenges

- Keeping the reference counts accurate.
- The complexity associated with handling circular structures.

Tracing collectors

- All live objects are found by iteratively tracing all references and subsequent references from an initial set of known to be live objects.
- The initial set of live objects (also called *root objects*) are located by analysing the registers, global fields, and stack frames at the moment when a garbage collection is triggered.
- After that, the tracing collector follows references from these objects and queues them up.

Contd.

- Marking all found referenced objects live means that the known live set increases over time.
- .Once the tracing collector has found all live objects, it will reclaim the remaining memory.
- Tracing collectors differ from reference-counting collectors in that they can handle circular structures.
- The main attraction of most tracing collectors is the marking phase, which entails a wait before being able to reclaim non-referenced memory.

Tracing collector algorithms

- Tracing collectors are most commonly used for memory management in dynamic languages.
- They're still the two most common algorithms that implement tracing garbage collection
  - Copying
  - mark-and-sweep garbage collection

Copying collectors

- Traditional copying collectors use a **from-space** and a **to-space** i.e. two separately defined address spaces of the heap.
- At the point of garbage collection, the live objects within the area defined as **from-space** are copied into the next available space within the area defined as **to-space**. When all the live objects within the **from-space** are moved out, the entire **from-space** can be reclaimed.

Pros & Cons

- Main advantage of copying collectors is that objects are allocated together tightly in the **to-space**, completely eliminating fragmentation.
- Copying collectors are also known as *stop-the-world collectors*, which means that no application work can be executed for as long as the garbage collection is in cycle.
- It’s a time consuming algorithm and also not suitable where a large set of live objects are concerned because we have to make enough space to keep all the live objects. So, Its slightly memory inefficient.

Mark-and-sweep collectors

- A *mark-and-sweep collector* traces references and marks each found object with a "live" bit.
- Once the marking process is completed, in the sweep phase, a collector will traverse the entire heap to locate all unmarked locations of consecutive memory address spaces.
- Unmarked memory is free and reclaimable.
- The collector then links together these unmarked addresses into organized free lists.
- After, the sweep phase is complete allocation will begin again.
- Most commercial JVMs deployed in enterprise production environments run mark-and-sweep (or marking) collectors

Pros and Cons

- It optimizes the memory allocation and reduces fragmentation.
- The mark phase is dependent on the amount of live data on the heap and the sweep phase is dependent on the heap size. Since we have to wait until both the *mark* and *sweep* phases are complete to reclaim memory.
- this algorithm causes pause-time challenges for larger heaps and larger live data sets.
- Through GC-Tuning options we can accommodate various application scenarios and needs.
- Tuning for every load change and application modification is a repetitive task, however, as the tuning is only valid for a specific workload and allocation rate.

Implementations of mark-and-sweep

- Parallel approach
- Concurrent (or mostly concurrent) approach.

Parallel collectors

- *Parallel collection* means that resources assigned to the process are used in parallel for the purpose of garbage collection.
- Most commercially implemented parallel collectors are monolithic stop-the-world collectors -- all application threads are stopped until the entire garbage collection cycle is complete.
- In production environments, application threads cannot do any work during a GC
- It's having a major impact on response-time sensitive applications, especially if there are lot of references to trace.
- It will happen when many live or complex data structures on the heap.
- For a monolithic parallel approach using all resources in parallel, this entire time will be a pause, and that pause corresponds to the entire GC cycle.

Concurrent Collectors

- *Concurrent* means that some (or most) garbage collection work is performed concurrently with the running application threads.
- It is needed enough time to trace the live set and reclaim the memory before the application runs out of memory.
- Heuristics have been designed to determine when to start garbage collection and when to do various GC optimizing tasks and how much at a time, etc.

Why tuning doesn't replace garbage collection

- Most tuning parameters -- such as allocation rate, object sizes, timing of response-time sensitive tasks, and how fast objects die -- are tuned specifically for the application's allocation rate, such as the test workload at hand. The end result could be either (or both) of these:
  - Things that worked during testing fail in production.
  - Workload or application changes requires to re-tune the application entirely.
- Tuning is something that will always need to be repeated!
- Concurrent garbage collectors in particular can require a lot of tuning, especially in production environments. Heuristics are needed to match the specific application's needs for the expected worst-case load.
- The end result becomes a very rigid configuration, leading to a lot of resource waste.

Summary

- Different garbage collection algorithms and approaches will meet different application needs. Tracing collectors are most commonly used in commercial Java environments.
- Parallel garbage collection uses available resources in parallel to perform GC. This tactic is usually implemented as a monolithic, stop-the-world collector, using all available system resources for a fast GC. Parallel GC thus provides higher throughput, but all application threads must wait until it's finished, which impacts latency.
- Concurrent GC does its work while application threads are still running. The timing of concurrent GC is tricky because it needs to be finished before your application requires memory.

Contd.

- Generational garbage collection helps postpone fragmentation, but does not eliminate it. Generational GC divides the heap into two spaces, one for allocating young objects and one for objects that (being still referenced) have survived young-space GC. Use a generational collector for any Java application that has many short-lived small objects that will die within their first collection cycle.
- Compaction is the only way to handle fragmentation completely. Most collectors have to perform compaction as a stop-the-world operation. Applications having long running time with more reference complexity having more heterogeneous object-size distribution. These factors will result in longer pauses to complete compaction. Larger heap size also impacts the compaction pause because there will likely be more live data and more references to update.

- Tuning can help postpone OutOfMemoryErrors but the trade-off of too much tuning is rigidity. Be sure about the consequences of a non-dynamic approach before setting it as, A too-rigid configuration will most likely break under dynamic production loads.