Hansraj Choudhary · Follow
Sep 9, 2019 . 8 min read

Evolution of Garbage Collection on Java: Garbage First Garbage Collection

Garbage collector is a form of *automatic memory management in Java*. In very basic terms, its like a deconstructor in C++.

Understanding the strategies and parameters for GC-tuning will help in controlling unexpected downtimes in critical applications on JVM.

In this post I'll touch upon old java GC architecture, explore G1 GC in details, how the G1 works compared to other collectors and why it can so easily outperform other state-of-the-art GCs on large heaps.

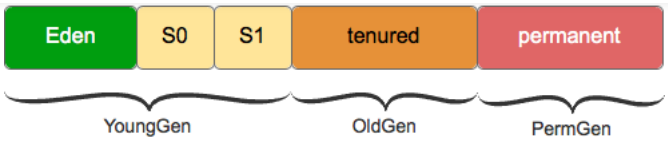
Understanding GC

Take a look at the following code:

```
for (Employee e : employees) {
    String printStr = "Printing the employee name: "+e.getName();
    System.out.println(printStr);
}
```

In the above code, the String *printStr* is being created on each iteration of the **for** loop. The string object printStr is considered "*garbage*" after every iteration. Eventually, when a lot of garbage is getting accumulated and the Java Virtual Machine started getting out of space to make new objects, that's where the garbage collector steps in.

The below diagram seems to be familiar to almost everyone and it's been a hot topic of discussion since last many years in favour & against Java. Garbage collection strategies before Java 7 relied on the definition of fixed regions of memory named generations



Eden is the space where the newly created objects start and moved to *Survivor* space after one GC cycle. When objects are garbage collected from *Young* generation space and survived objects are moved to *Old* Generation space, is called minor GC event. When objects from Old generation are garbage collected are the major GC event. Classes which are being used are loaded in *Permanent Generation* during runtime.

The String objects being formed inside for-loop in the above example would not be survived from Eden space and died young. The objects like 'employees' usually survive for a longer time if being used in the code at multiple places.

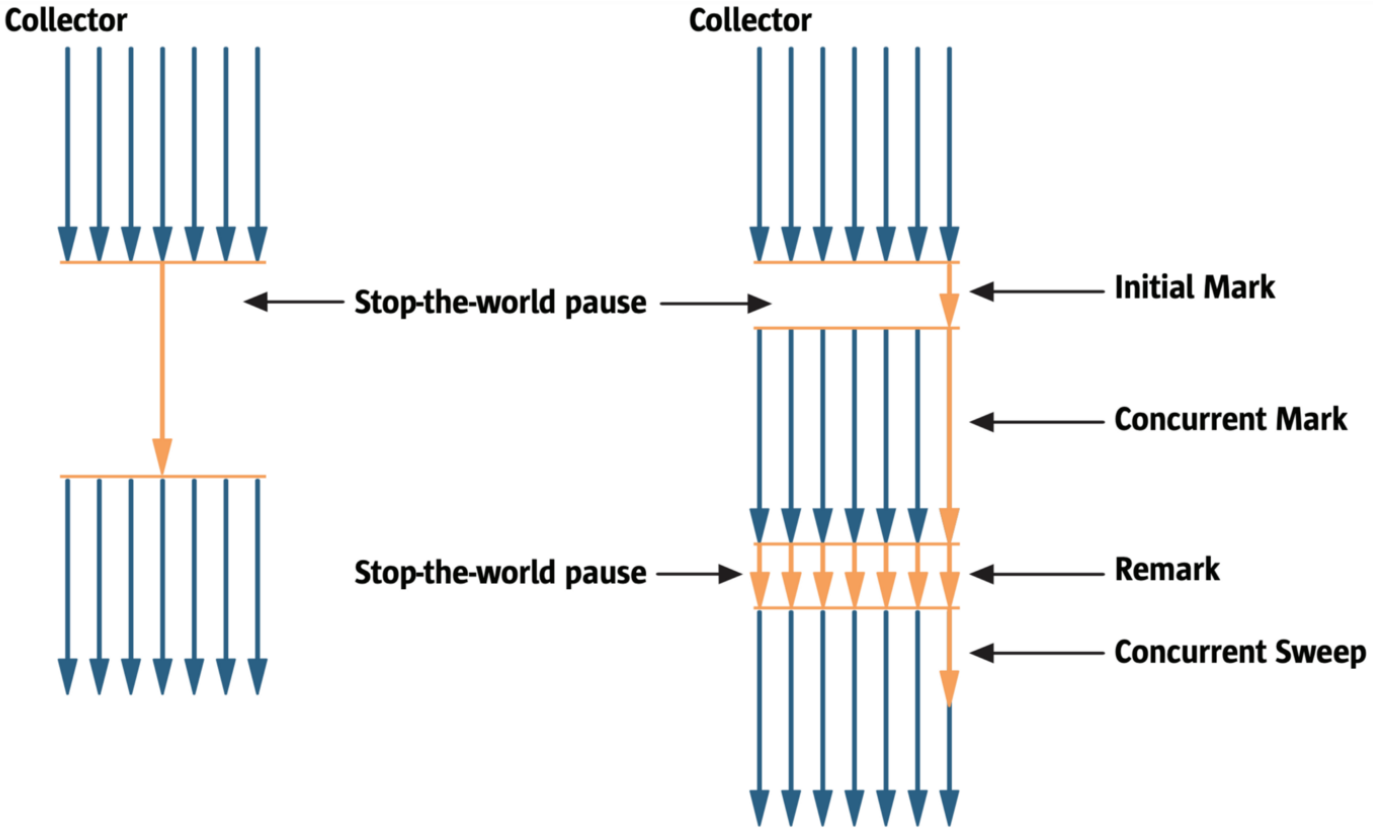
GC algorithms

The event in which Garbage Collectors are doing their job is called "Stop the world *(StW)*" event which means all of your application threads are put on hold until the garbage is collected.

Serial Garbage Collector: *GC events are performed serially in one thread & works by holding all the application threads. Used only for small applications or for local testing purposes [+UseSerialGC]*

Parallel Garbage Collector: *Serial GC in multiple threads. [+UseParallelGC]*

CMS Garbage Collector: *Runs concurrently with the application & hold application threads during marking process [+USeParNewGC]*



Comparison between serial and CMS collection

> *A collection cycle for the CMS collector starts with a short pause, called the* ***initial mark****, that identifies the initial set of live objects directly*

G1 (Garbage First): It is *used for large heap memory areas and default garbage collector from JDK 9 onwards, offering a balance between latency and throughput for most use cases [+UseG1GC]*

Shenandoah: *Enhancements to G1GC, from java 12 on [+UseShenandoahGC]*

> One disadvantage of these garbage collection algorithms is that they require copying and deleting objects during promotions, temporarily using more memory and in some cases causing an overflow that could crash an application.

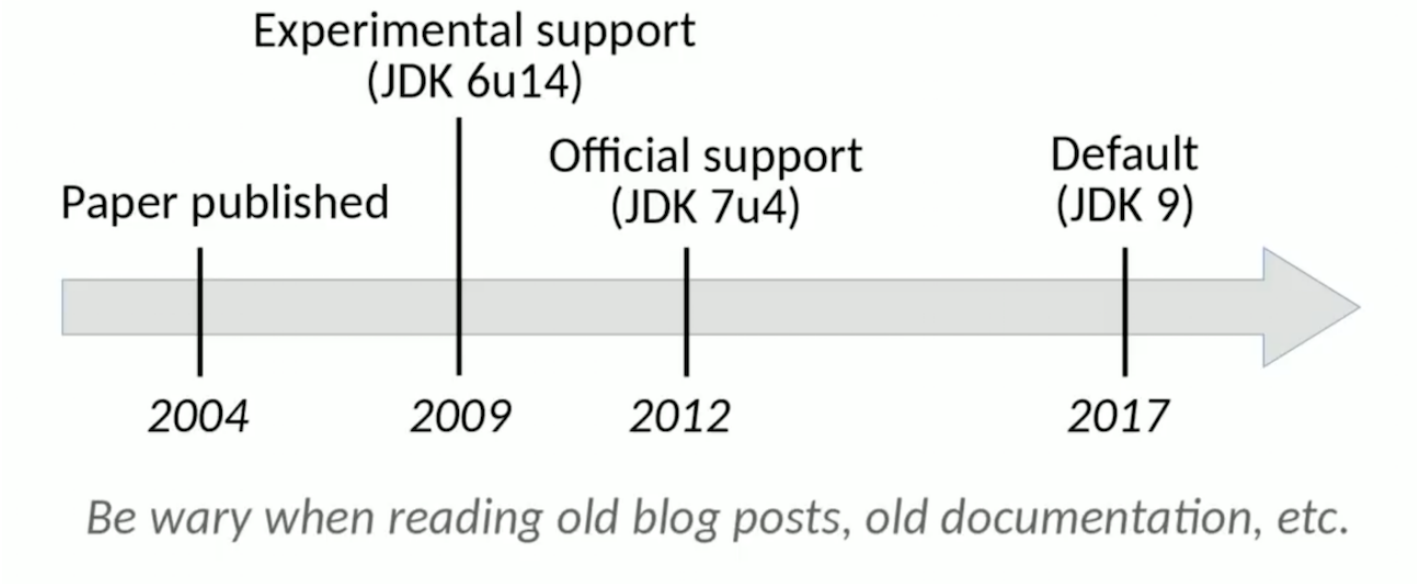
Depending on the heap size and the number of live objects, StW can take plenty of time — up to several minutes. Of course, that always happens exactly when you don't want it to happen! G1GC has addressed this issue till a lot of extent. Lets understand G1GC in detail.

G1GC (Garbage First)

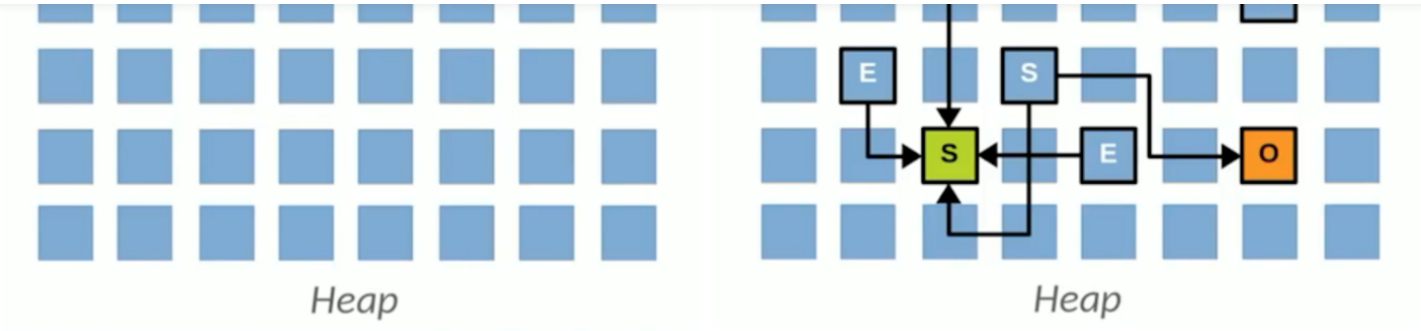
From wikipedia

> *Garbage-first (G1) collector is a server-style garbage collector, targeted for multiprocessors with large memories, that meets a soft real-time goal with high probability, while achieving high throughput.*

There are a lot of old blog posts and documentation available with details about the old GC type & algorithms. As G1GC become default collector, understanding the algorithm and tuning params is critical.



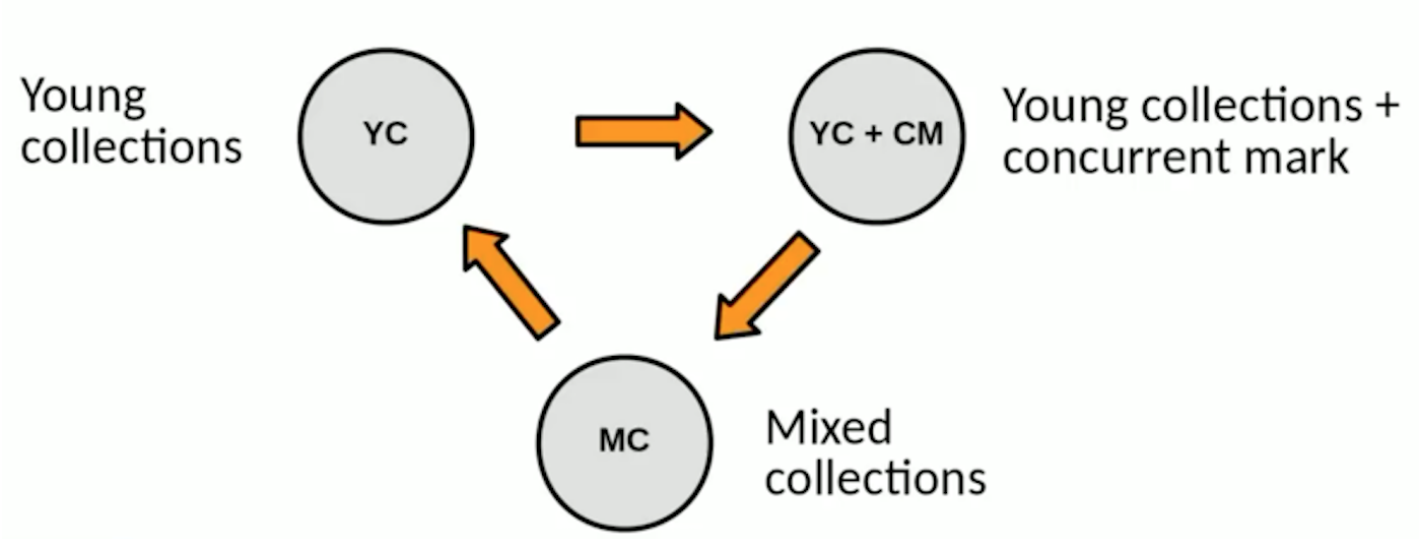
G1 divides memory into heap regions of the same size instead of splitting the heap into 3 big regions. The heap does not have to be split into contiguous Young and Old generation. Instead, it is split into a number smaller heap regions that can store objects. Each region may be an Eden (E) region, a Survivor (S) region or an Old (O) region



The above picture shows that heap having empty regions during start of the application which are called ‘free list’. When object production begins, a region is allocated from the free list and marked as (E). When the region has been exhausted of space, a new region is selected, allocated and filled.

First, being true to its name, G1 collects regions with the least amount of live data (Garbage First!) and compacts/evacuates live data into new regions. Secondly, it uses a series of incremental, parallel and multi-phased cycles to achieve its soft pause time target. This allows G1 to do what’s necessary, in the time defined, irrespective of the overall heap size.

> *On a high level, the G1 collector alternates between two phases. The young-only phase contains garbage collections that fill up the currently available memory with objects in the old generation gradually. The space-reclamation phase is where G1 reclaims space in the old generation incrementally, in addition to handling the young generation. Then the cycle restarts with a young-only phase.*



This diagram represents a mixed collection. All Eden regions are collected and evacuated to Survivor regions and, depending on age, all survivor regions are collected and sufficiently tenured live objects are promoted to new Old regions. The process of compaction and evacuation allows for a significant reduction in fragmentation and ensures adequate free regions are maintained.

Collection Set(CSet):- The set of regions to be collected in a gc pause. The number of young/survivor/old regions in CSet is decided dynamically according to pause time goal and other heuristics. G1 can therefore collect a few regions at a time. Fewer objects to collect results in shorter pause interval and the pause interval could be tuned easily.

> *If the size of an object is greater than 50% of a single region size, it considered as humongous object in g1 and directly allocated one or more continuous regions depending on size. It’s treated as an old region from the start to avoid copying huge amount of data during promotions.*

Humongous allocation represents a single object, and as such, must be allocated into contiguous space. This can lead to significant

Understanding GC Logs

Enable gc logs by passing the below-mentioned system properties to your JVM
-XX:+PrintGCDetails -XX:+PrintGCDateStamps -Xloggc:<file-path>

Garbage Collection log file would look like during a minor GC event:

```
1  2015-08-12T10:48:00.827-0400: 0.356: [GC pause (G1 Evacuation Pause) (young), 0.0758520 secs]
2     [Parallel Time: 22.8 ms, GC Workers: 8]
3        [GC Worker Start (ms): Min: 2518335564.5, Avg: 2518335564.5, Max: 2518335564.6, Diff: 0.1]
4        [Ext Root Scanning (ms): Min: 11.6, Avg: 13.4, Max: 15.3, Diff: 3.7, Sum: 53.6]
5        [Update RS (ms): Min: 1.5, Avg: 3.4, Max: 4.8, Diff: 3.3, Sum: 13.5]
6           [Processed Buffers: Min: 117, Avg: 163.5, Max: 192, Diff: 75, Sum: 654]
7        [Scan RS (ms): Min: 0.7, Avg: 0.8, Max: 0.8, Diff: 0.2, Sum: 3.1]
8        [Object Copy (ms): Min: 4.6, Avg: 4.9, Max: 5.3, Diff: 0.7, Sum: 19.6]
9        [Termination (ms): Min: 0.0, Avg: 0.0, Max: 0.0, Diff: 0.0, Sum: 0.0]
10       [GC Worker Other (ms): Min: 0.0, Avg: 0.1, Max: 0.1, Diff: 0.1, Sum: 0.3]
11       [GC Worker Total (ms): Min: 22.5, Avg: 22.5, Max: 22.6, Diff: 0.1, Sum: 90.2]
12       [GC Worker End (ms): Min: 2518335587.0, Avg: 2518335587.1, Max: 2518335587.1, Diff: 0.1]
13    [Code Root Fixup: 0.0 ms]
14    [Clear CT: 0.7 ms]
15    [Other: 52.4 ms]
16       [Choose CSet: 0.0 ms]
17       [Ref Proc: 48.6 ms]
18       [Ref Enq: 0.1 ms]
19       [Free CSet: 2.4 ms]
20    [Eden: 12.0M(12.0M)-&gt;0.0B(14.0M) Survivors: 0.0B-&gt;2.0M Heap: 12.6M(252.0M)->7848.3K(252.0M)]
21  [Times: user=0.08, sys=0.00, real=0.02 secs]
```

Here **0.356** *<in line-1>* indicates that 356 milliseconds after the Java process was started this GC event was fired. **(young)** — indicates that this is a Young GC event. **GC Workers: 8** — indicates the number of GC worker threads.

<line-20> indicates the heap size changes after minor GC event.

- **Eden: 12.0M(12.0M)->0.0B(14.0M) —** indicates that Eden generation's capacity was 12mb and all of the 12mb was occupied. After this GC event, young generation occupied size came down to 0. Target Capacity of Eden generation has been increased to 14mb, but not yet committed. Additional regions are added to Eden generation, as demands are made.
- **Survivors: 0.0B->2048.0K —** indicates that Survivor space was 0 bytes before this GC event. But after the even Survivor size increased to 2048kb. It indicates that objects are promoted from Young Generation to Survivor space.
- **Heap: 12.6M(252.0M)->7848.3K(252.0M)** — indicates that capacity of heap size was 252mb, in that 12.6mb was utilized. After this GC event, heap utilization dropped to 7848.3kb (i.e. 5mb (i.e. 12.6mb — 7848.3kb) of objects has been garbage collected in this event). And heap capacity remained at 252mb.

This GC event took **0.02** seconds *<in line-21>* to complete. When a Full GC event happens, **Full GC (Allocation Failure)** log statement will be printed in the GC log file in the 1st line. Subsequent lines will have details about memory allocations and time taken in GC process.

Conclusion

One of G1's main advantages is the ability to compact free memory space without lengthy pause times and uncommit unused heaps. I found this GC to be the best option for vertical scaling of Java applications running on OpenJDK or HotSpot JDK. By introducing a parallel, concurrent and multi-phased marking cycle, G1 GC can work with much larger heaps while providing reasonable worst-case pause times.

The basic idea with G1 GC is to set your heap ranges and a realistic (soft real time) pause time goal and then let the GC do its job. One thing to note is that for G1 GC, neither the young nor the old generation has to be contiguous. This is a handy feature since the sizing of the generation is now more dynamic.