THE PERFORMANCE ENGINEER'S GUIDE TO OPENJDK HOTSPOT GARBAGE COLLECTION

Monica Beckwith
President & CTO
Code Karam LLC

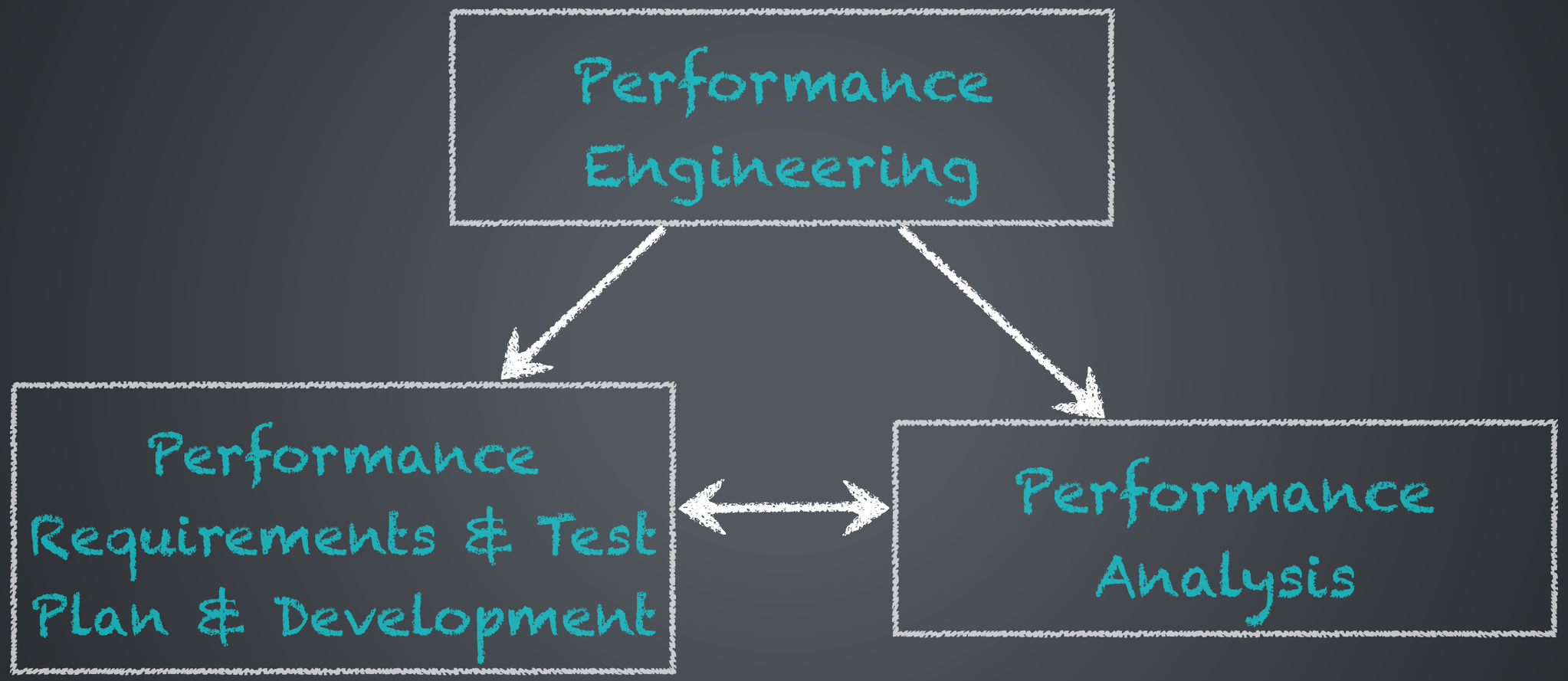
monica@codekaram.com

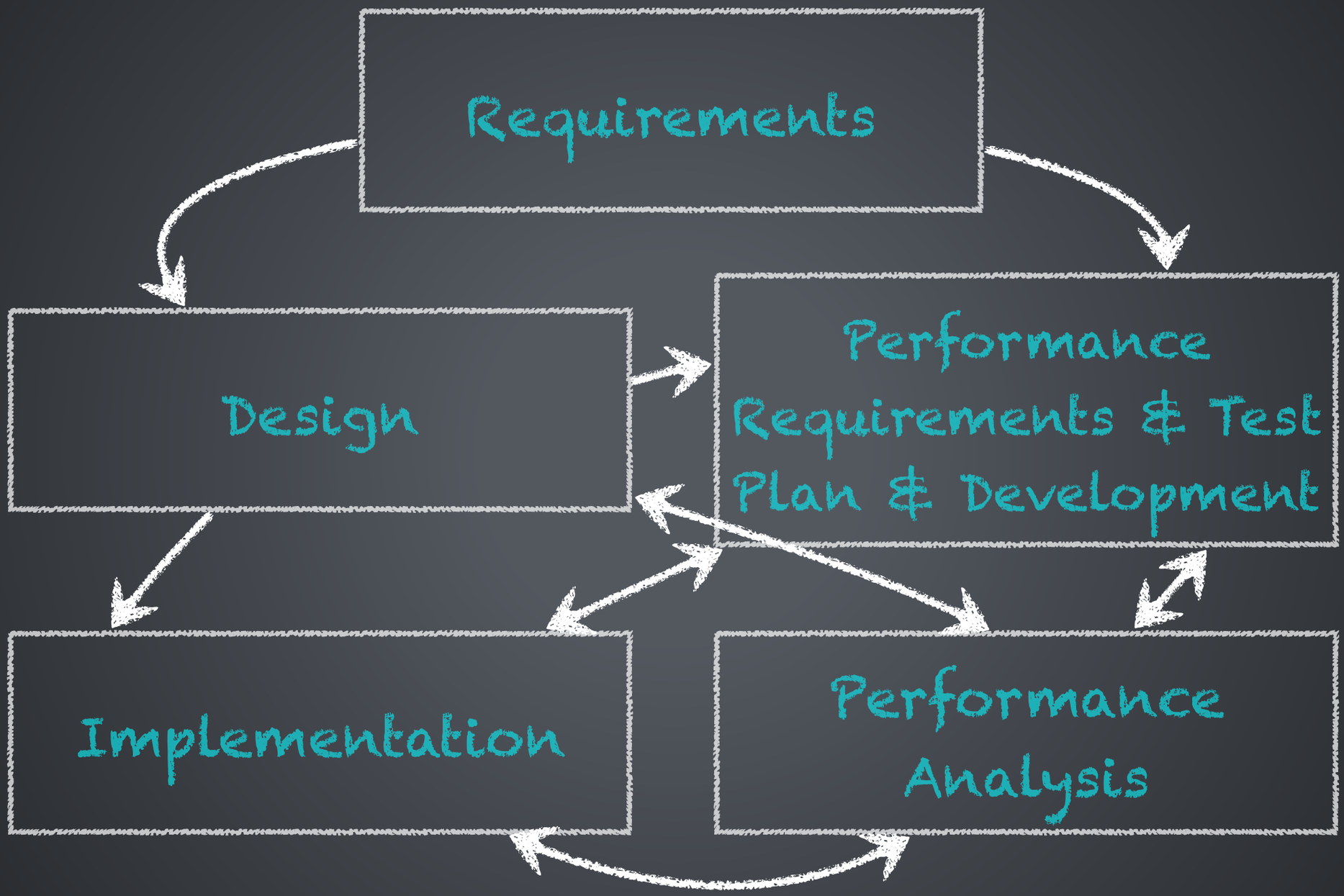
https://www.linkedin.com/in/monicabeckwith

www.codekaram.com

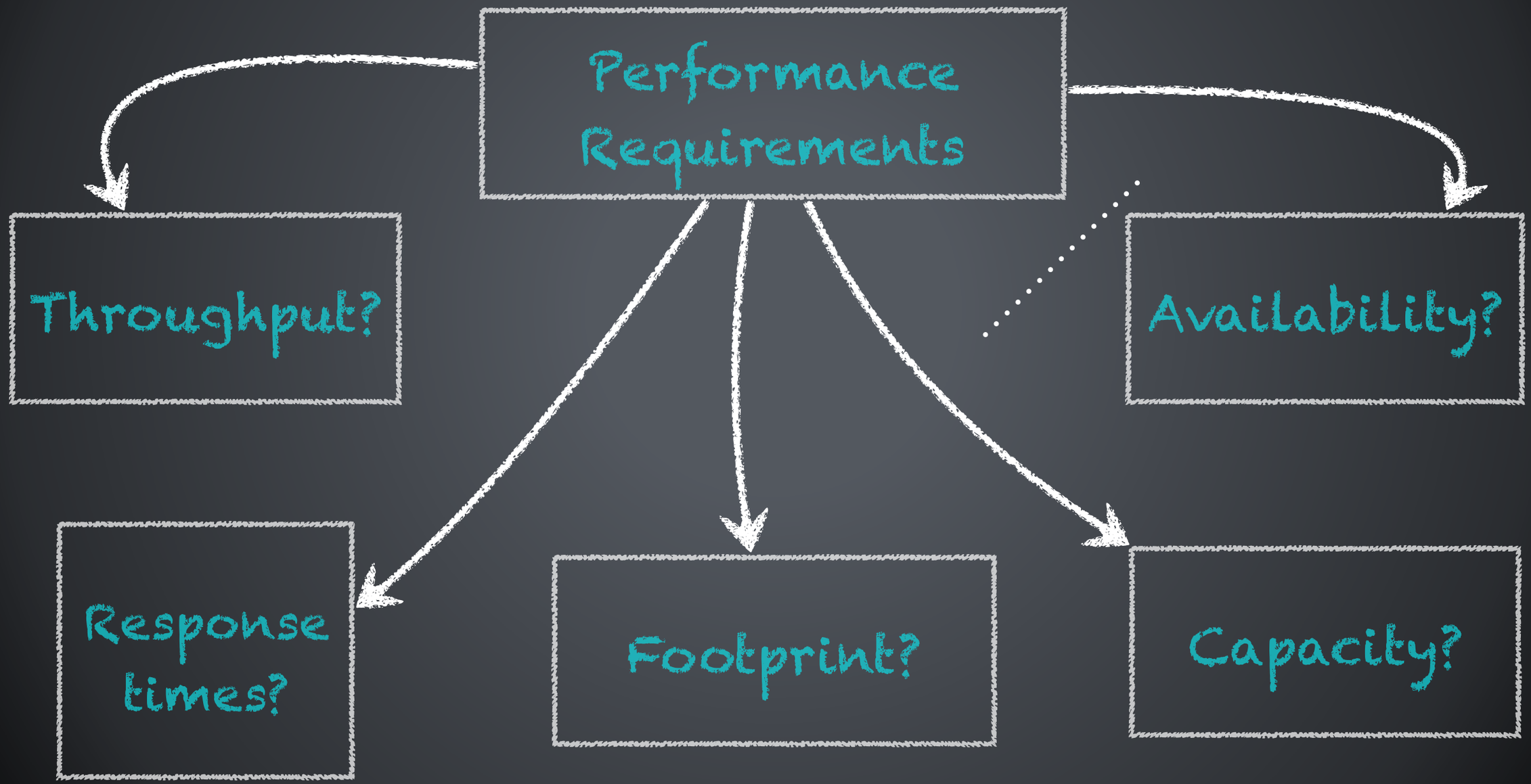
QCon Beijing
2016

Performance Engineering

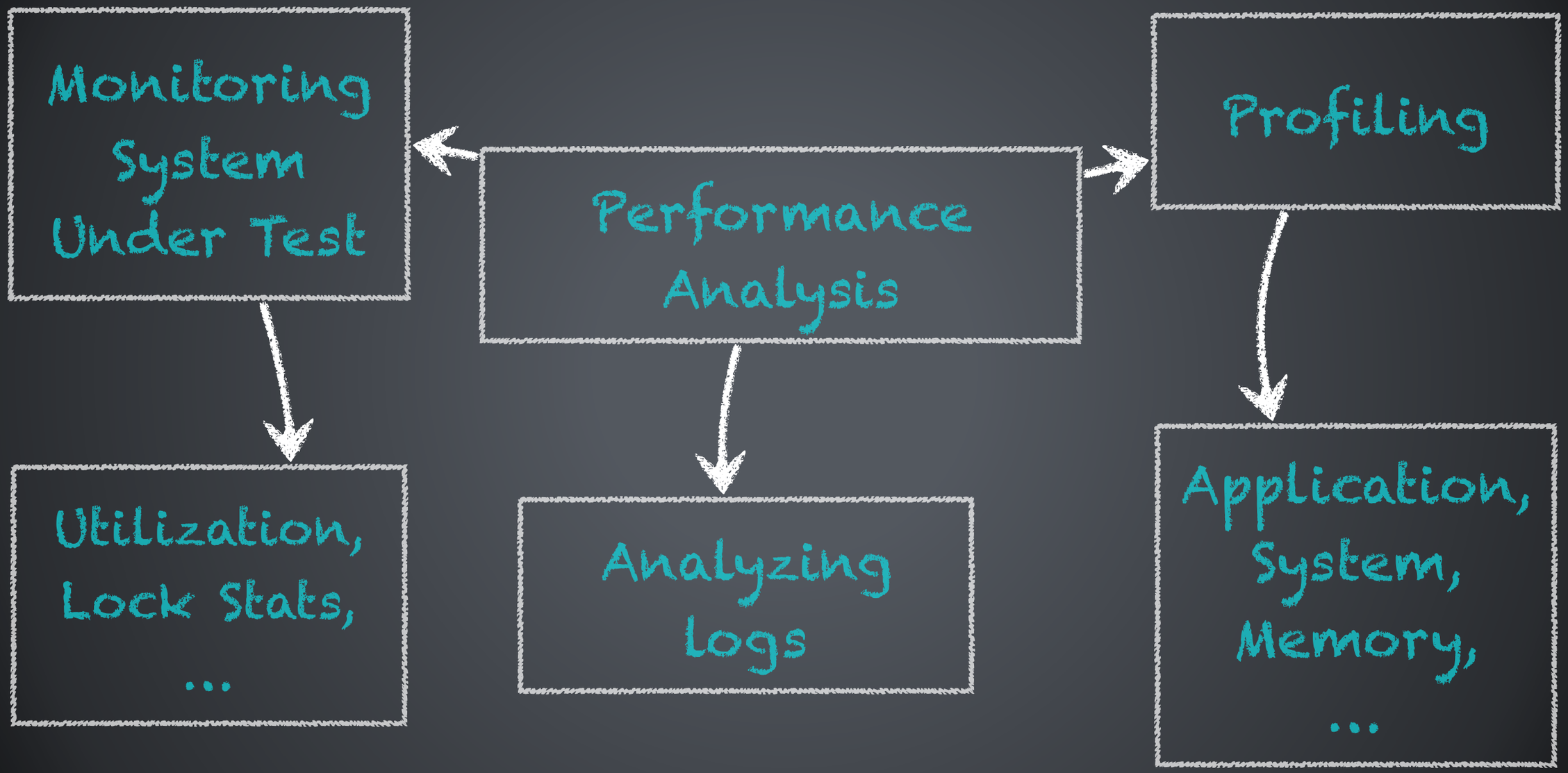




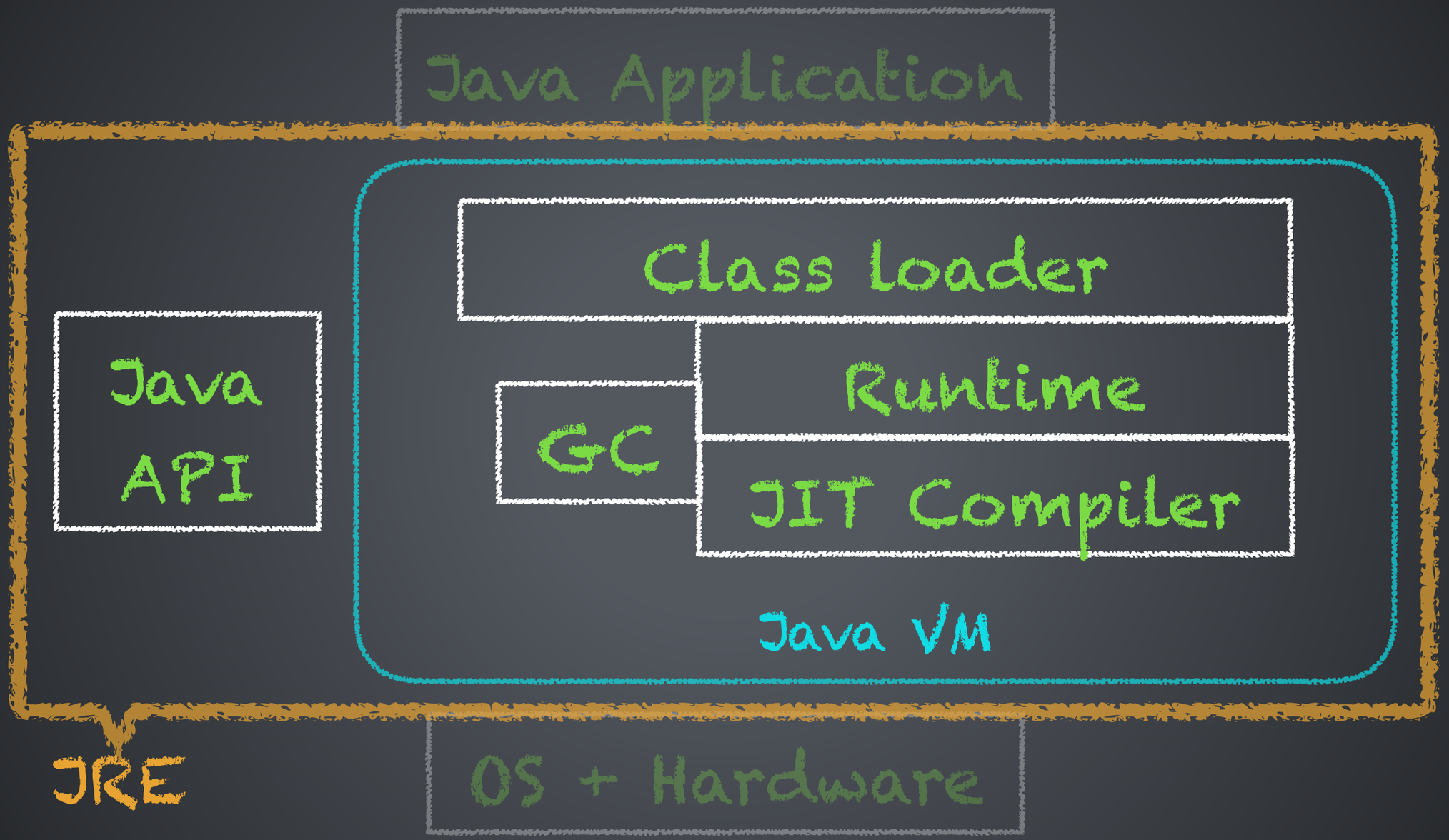
Performance Requirements

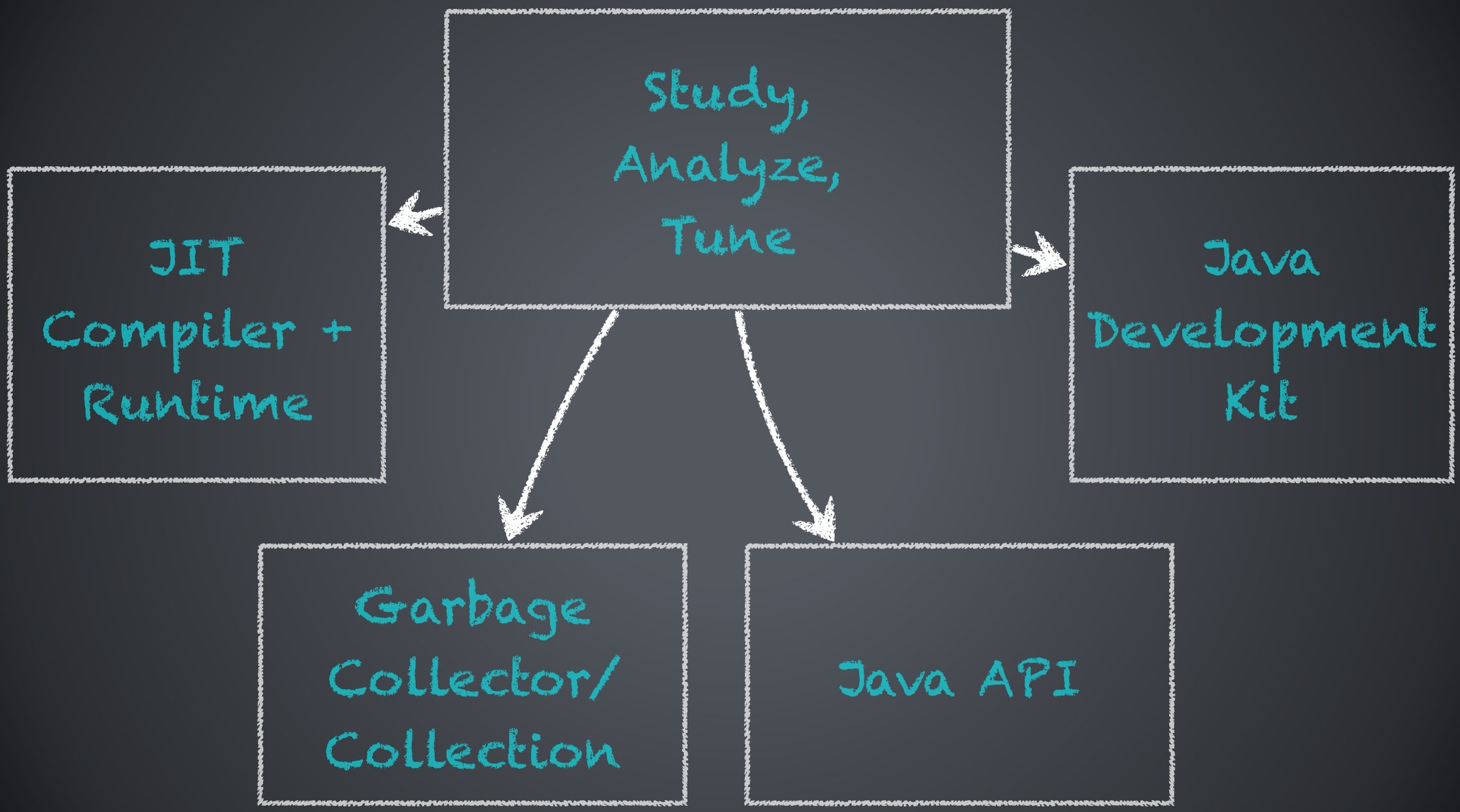


Performance Analysis

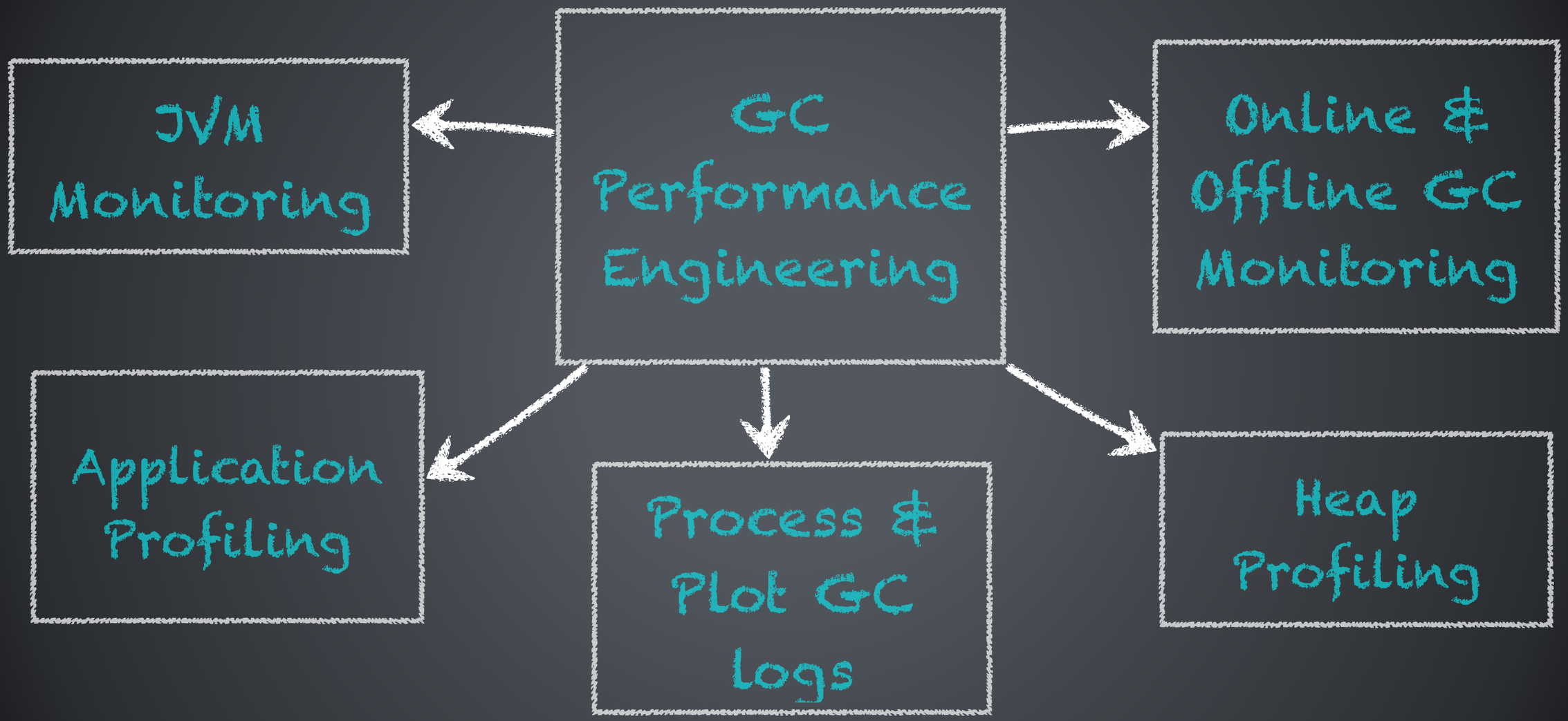


JVM Performance Engineering





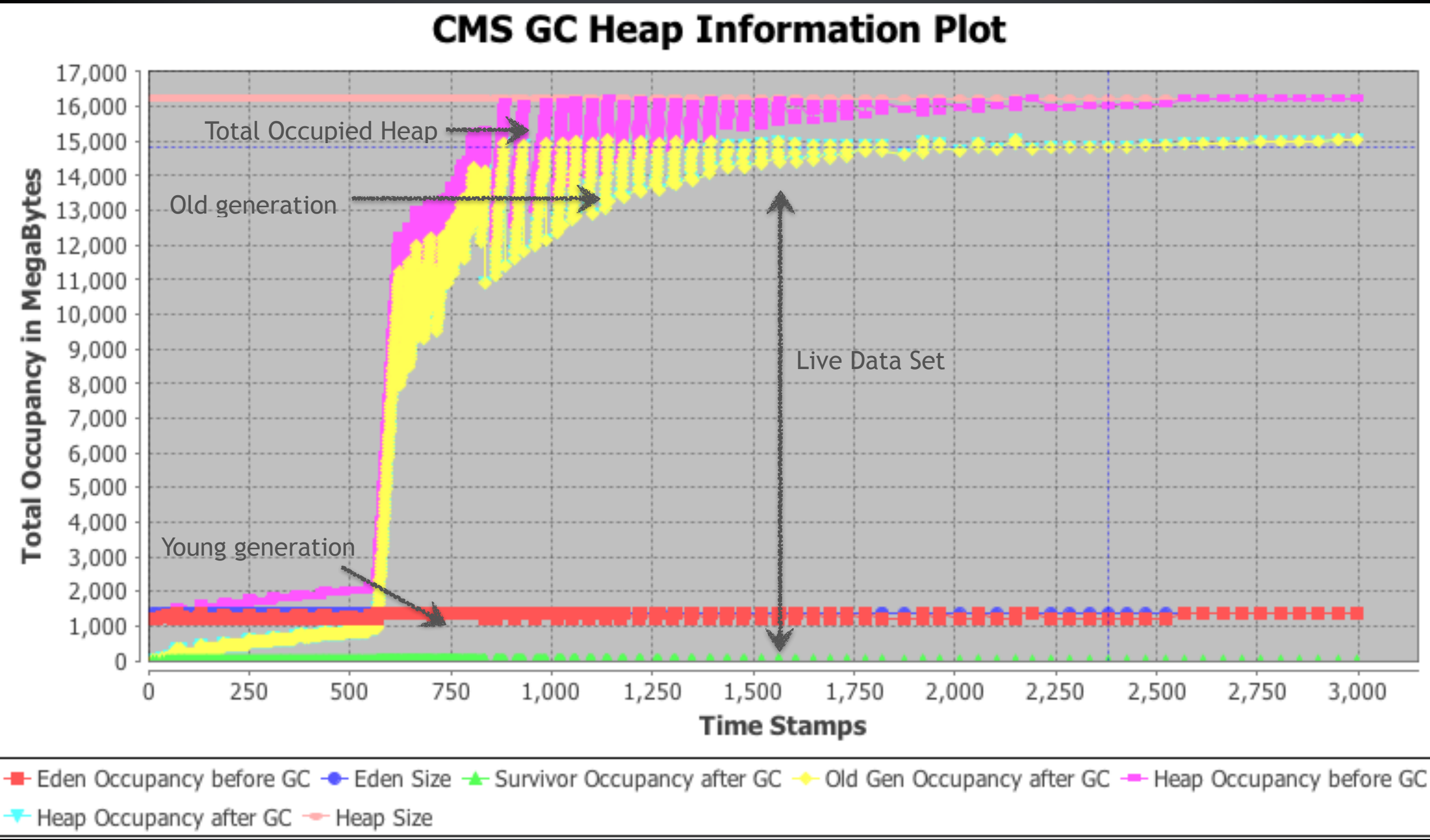
Garbage Collection Performance Engineering



Garbage Collection - Facts, Trade-Offs And Algorithms

GC Fact!

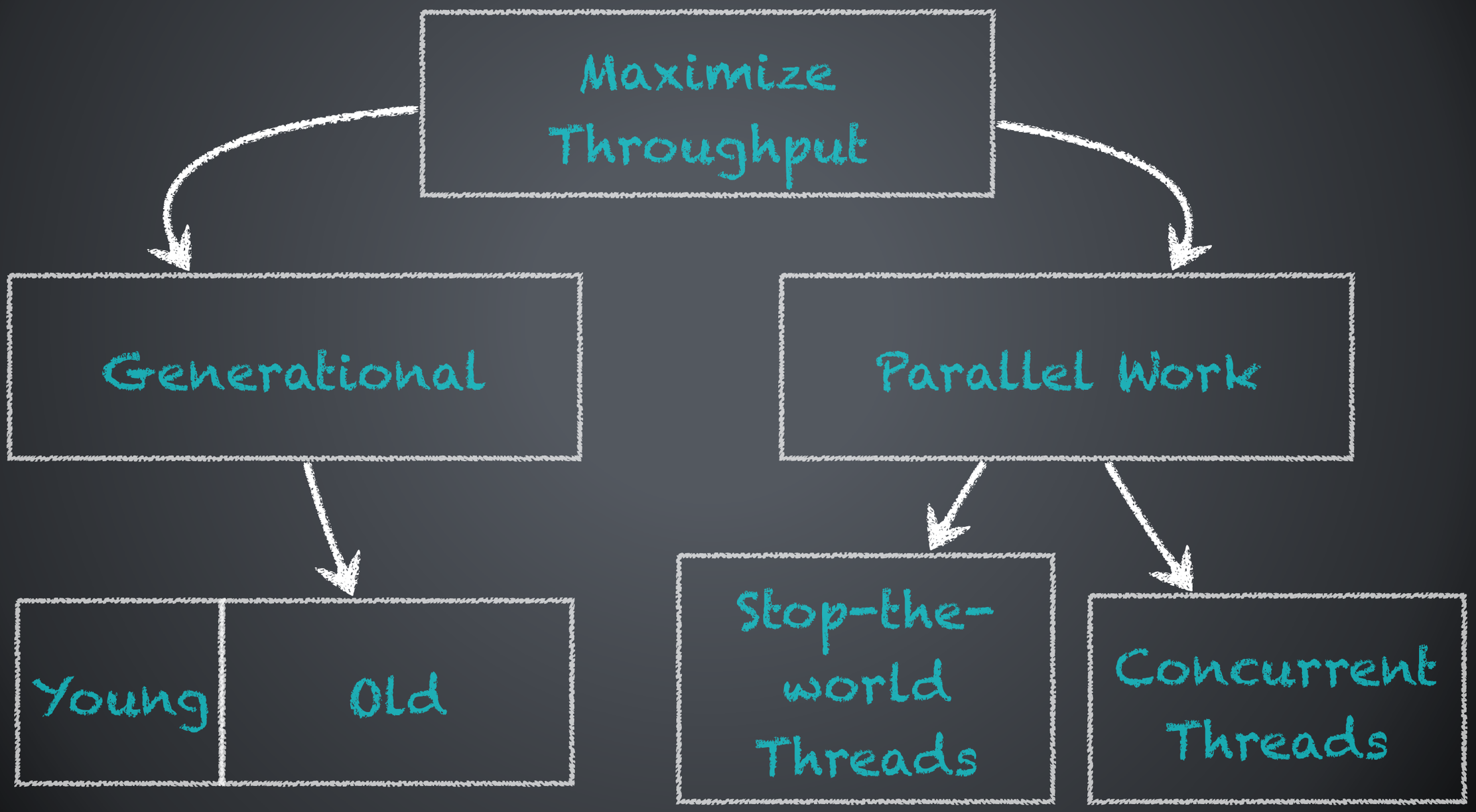
- GC can NOT eliminate your memory leaks!
- GC (and heap dump) can provide an insight into your application.



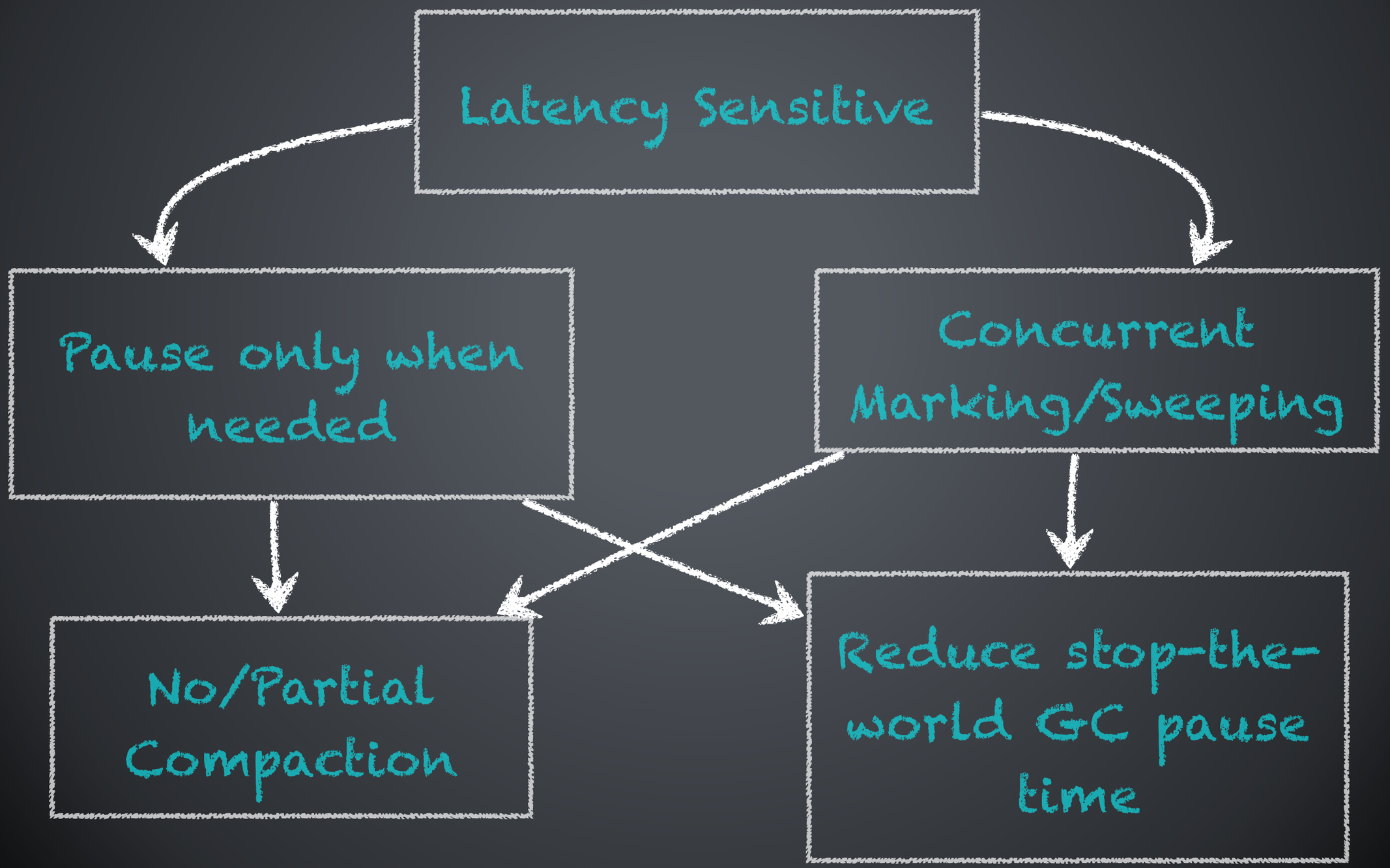
GC Fact!

Throughput and latency are the two main drivers towards refinement of GC algorithms.

The Throughput Maximizer



Mr. Latency Sensitive



Let's look at OpenJDK HotSpot GCs :)

Throughput Maximizer

CMS GC

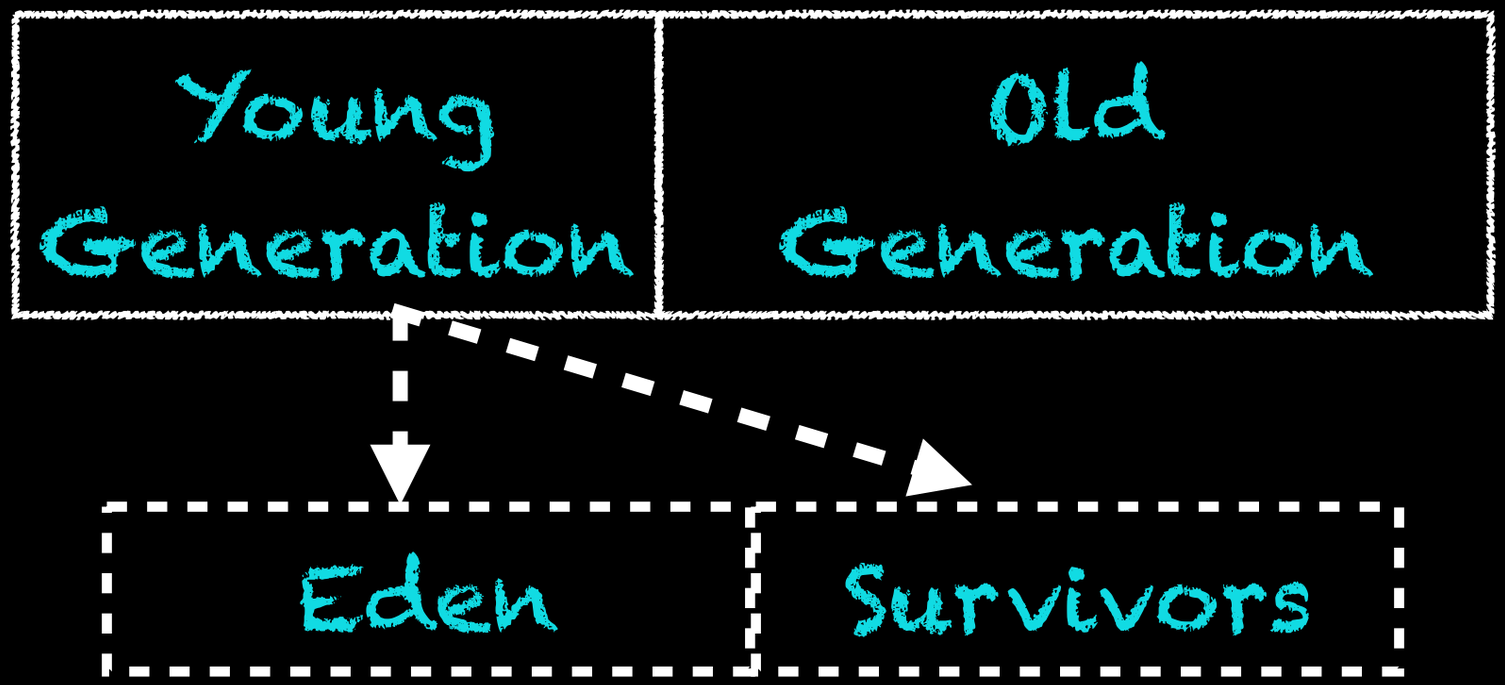
G1 GC

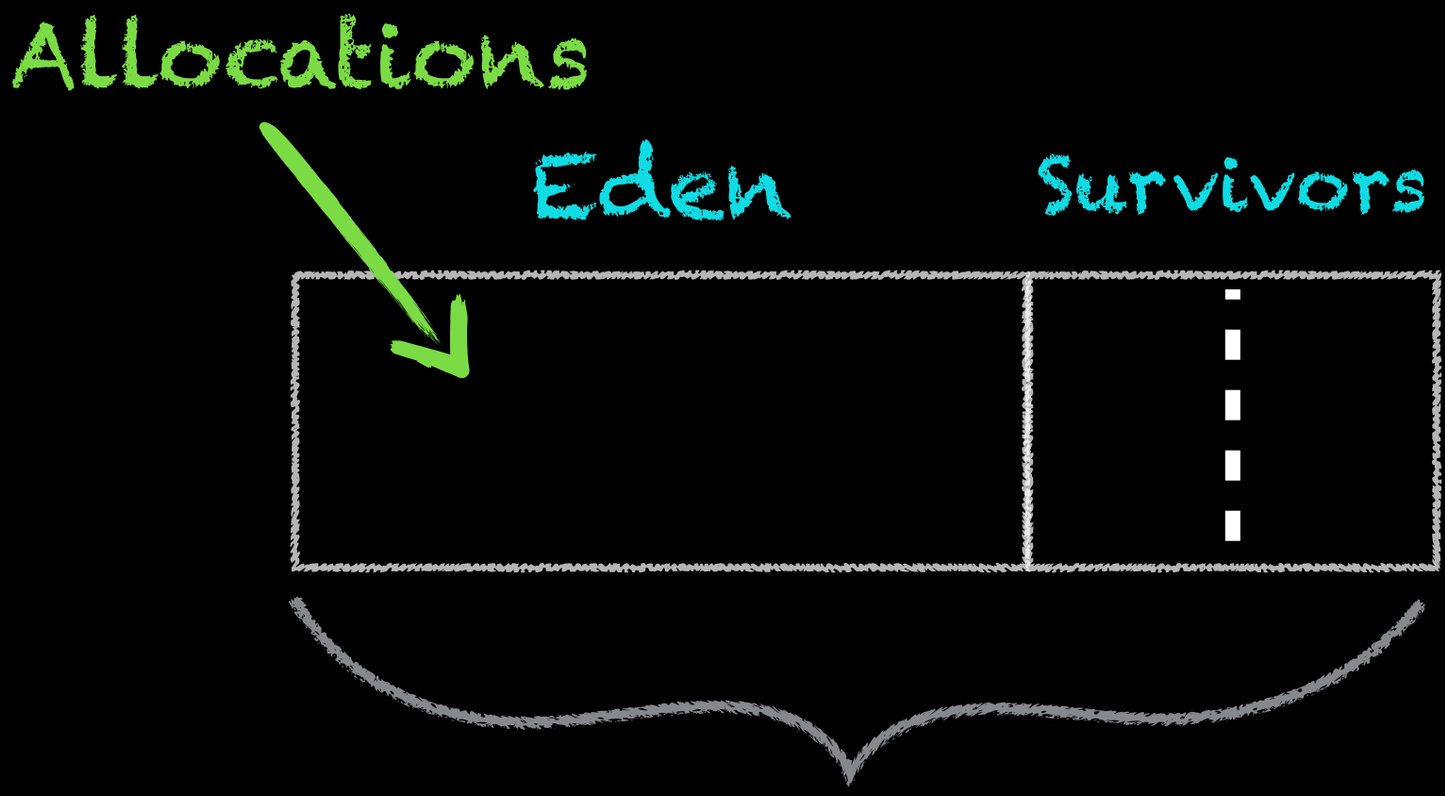
Parallel GC

Latency Sensitive

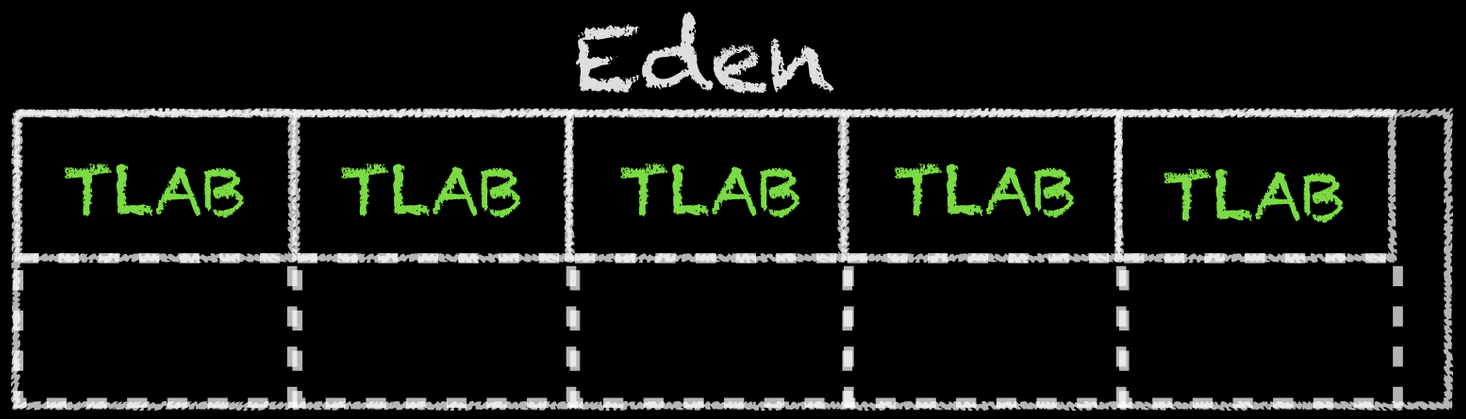
GC Fact!

All GCs in OpenJDK HotSpot are generational.

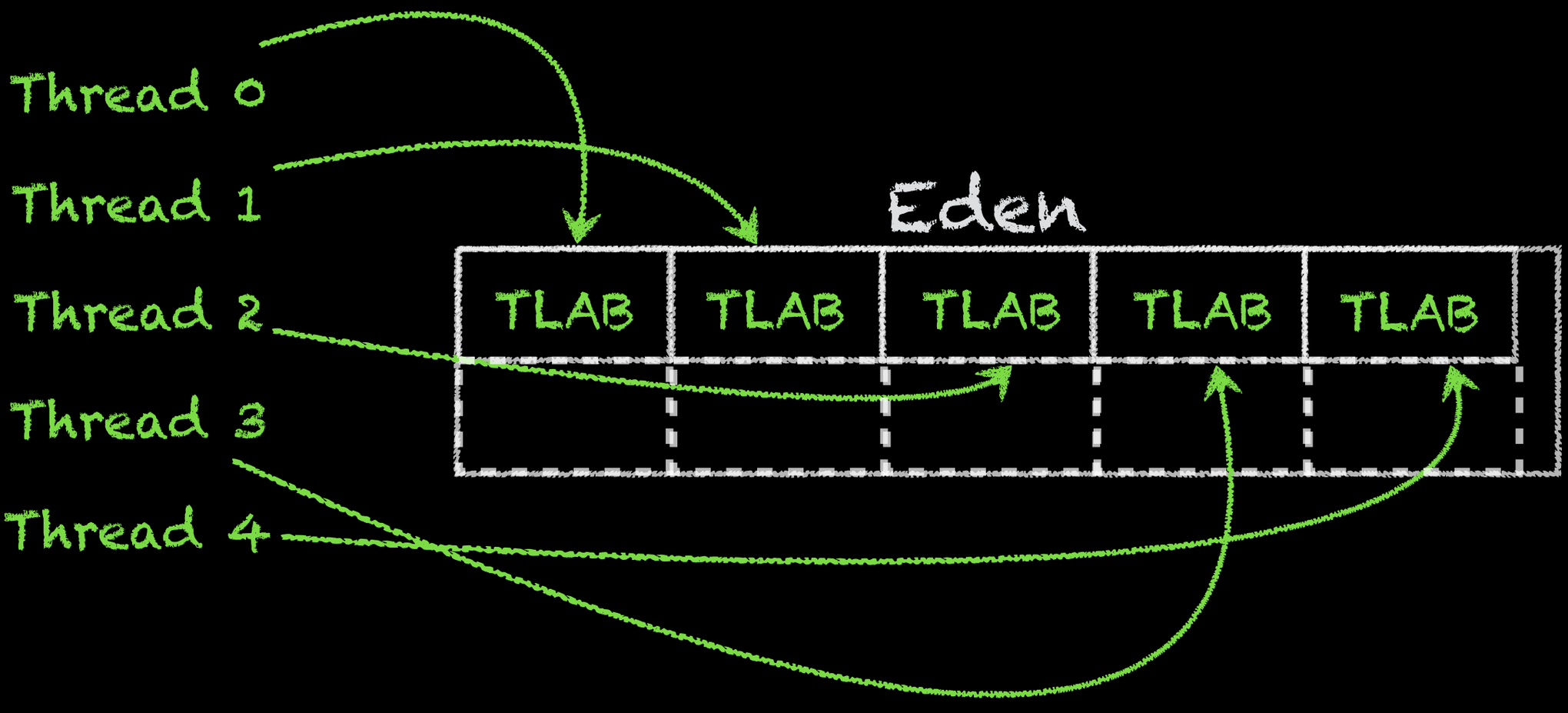


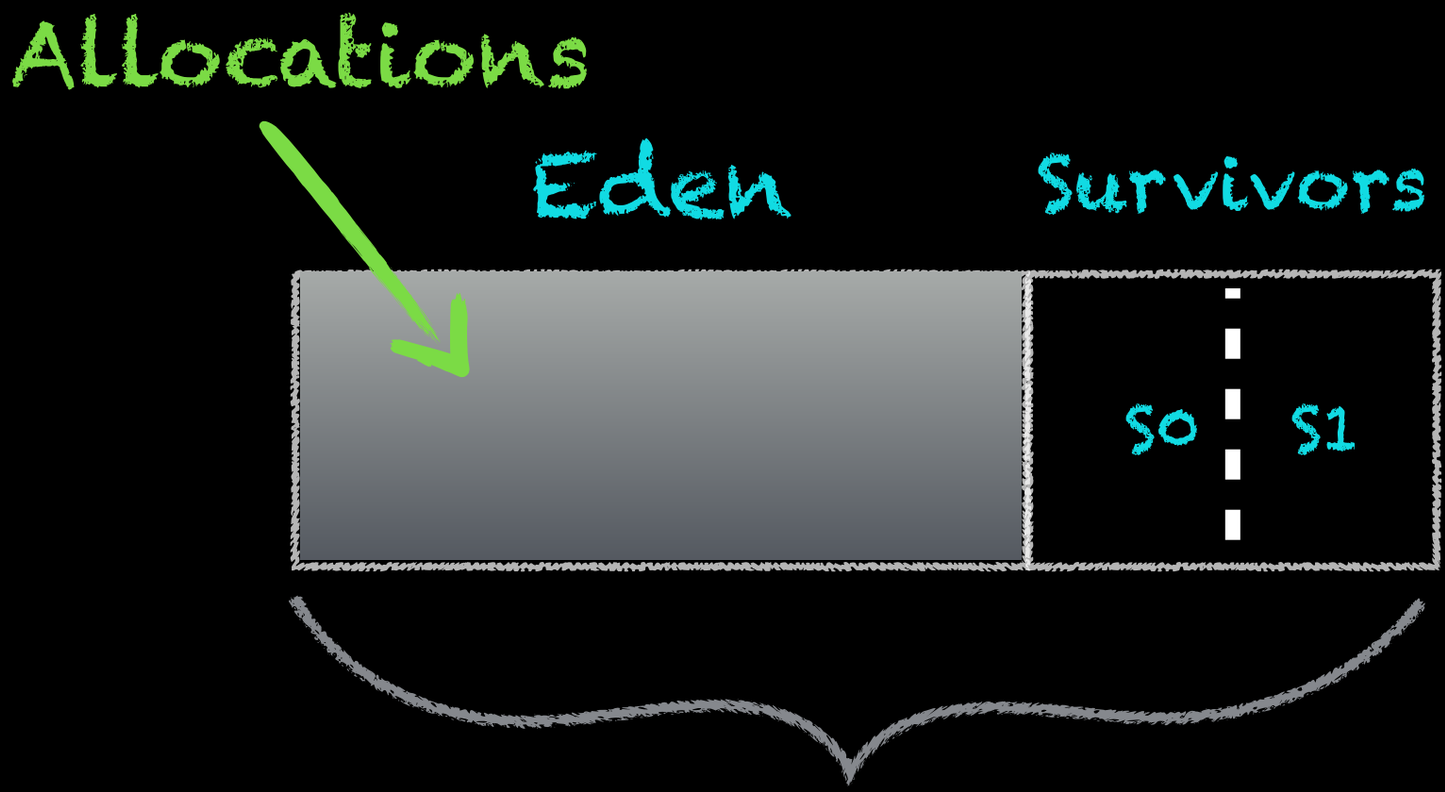


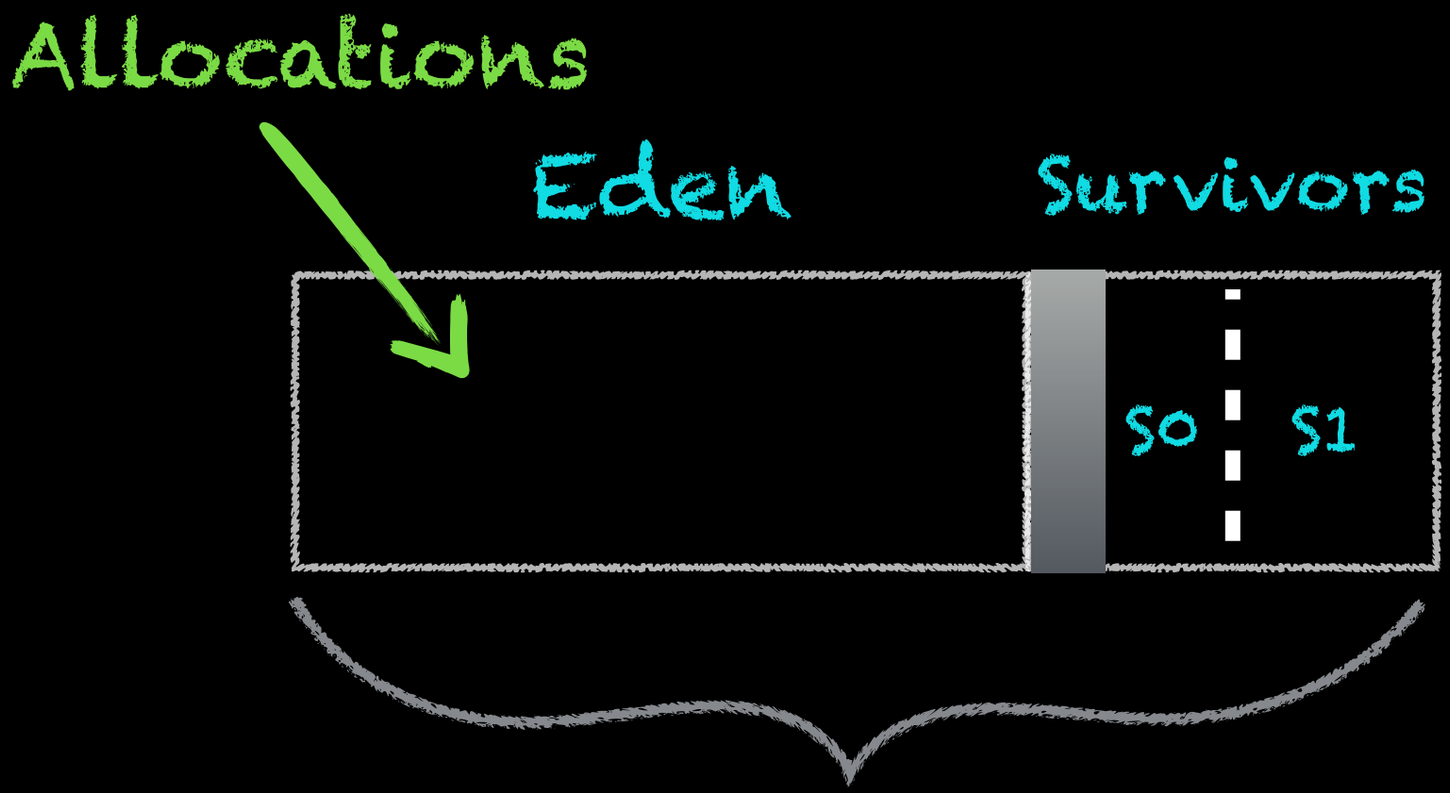
Fast Path Allocation == Lock-Free Allocation == Threads Allocate Into Their Local Allocation Buffer (LAB)s



TLAB = Thread Local Allocation Buffer







Garbage Collection - Reclamation.

Young Generation

Old Generation

Young Generation

Old Generation

*Similar GC Algorithms for OpenJDK HotSpot

Different GC Algorithms for OpenJDK Hotspot

Always collected in its entirety

Young Generation

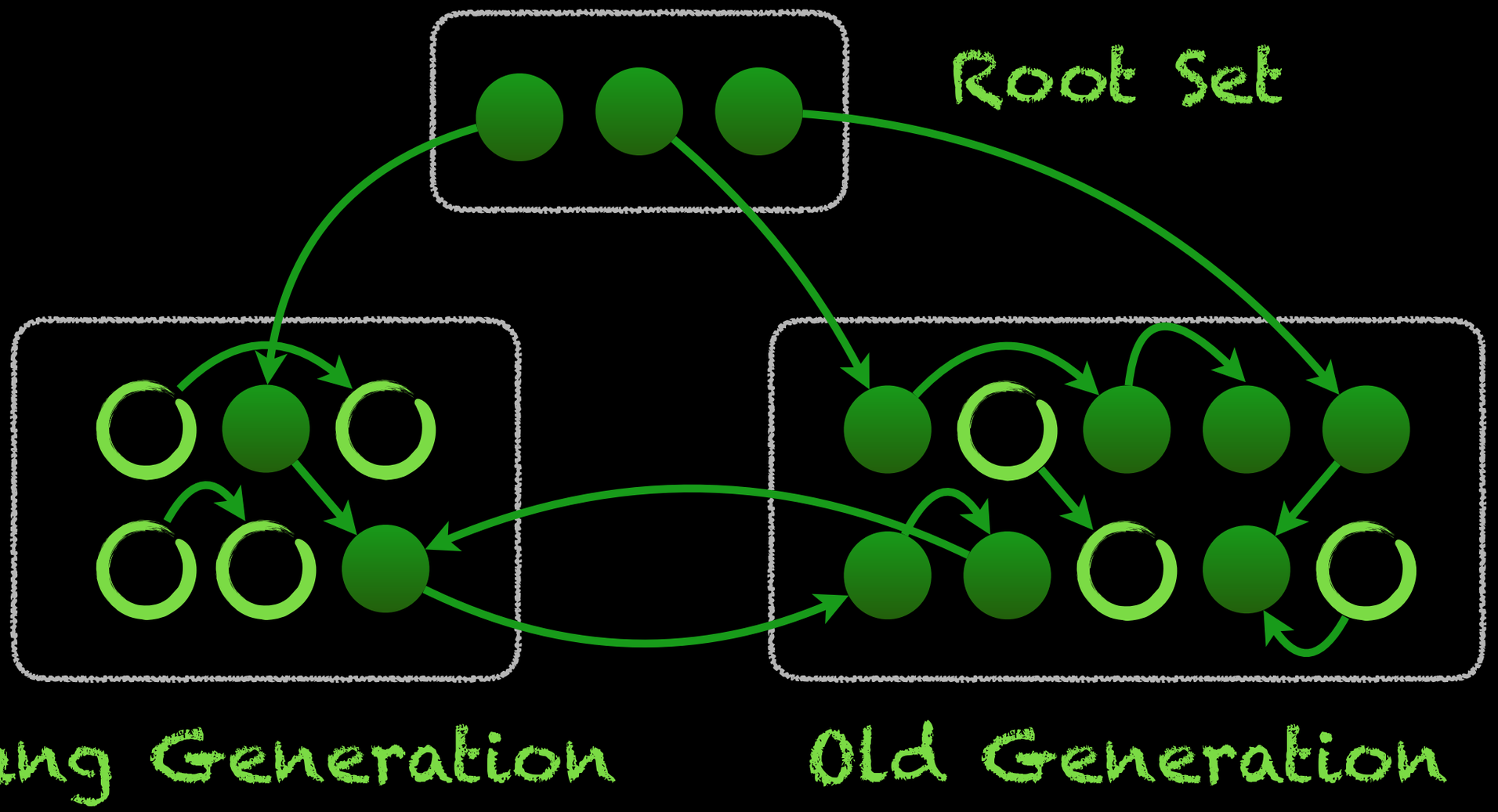
Old Generation

*Similar GC Algorithms for OpenJDK HotSpot

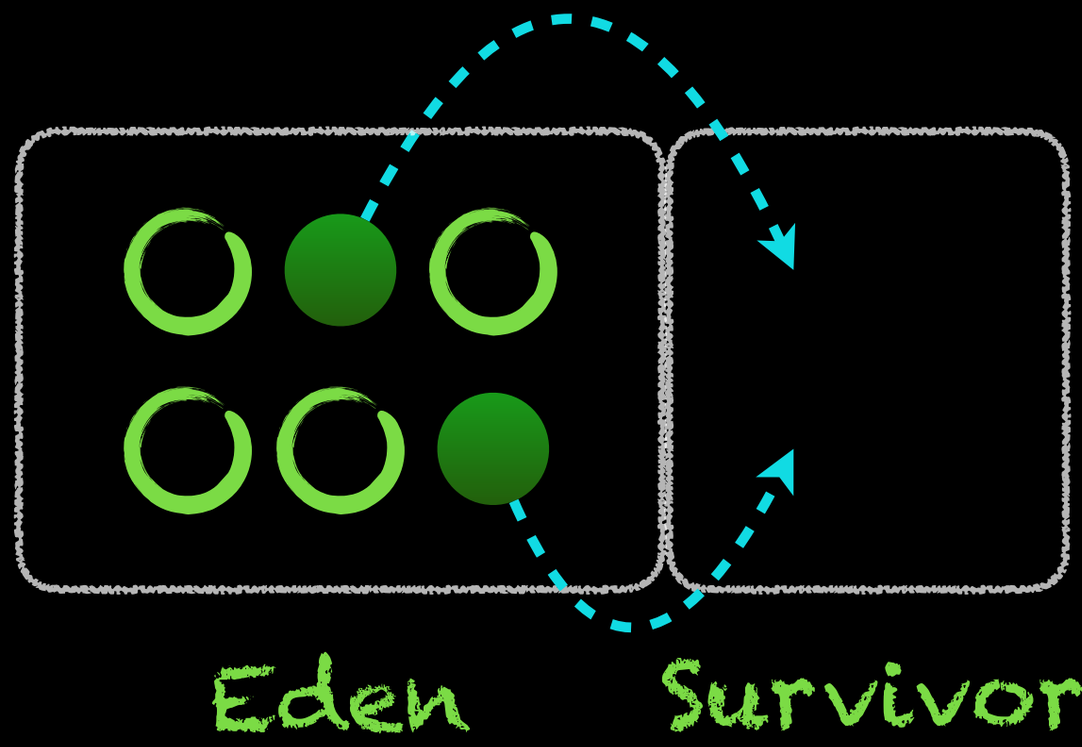
Different GC Algorithms for OpenJDK Hotspot

Always collected in its entirety

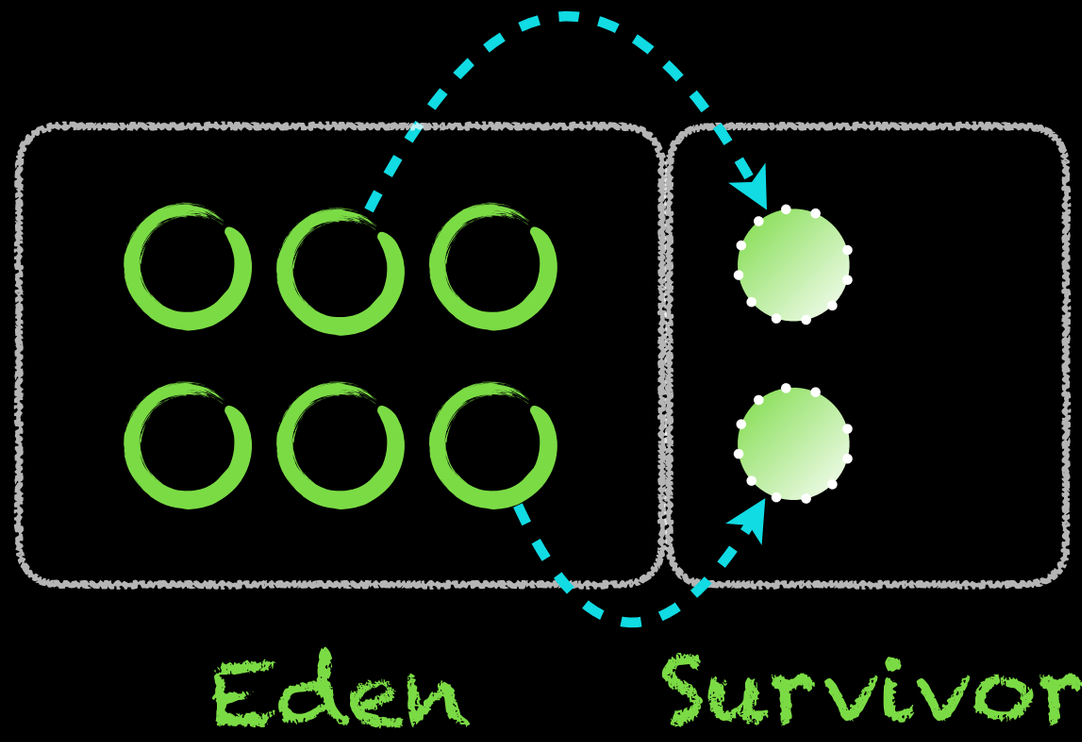
Young Garbage Collection == Reclamation Via Scavenging



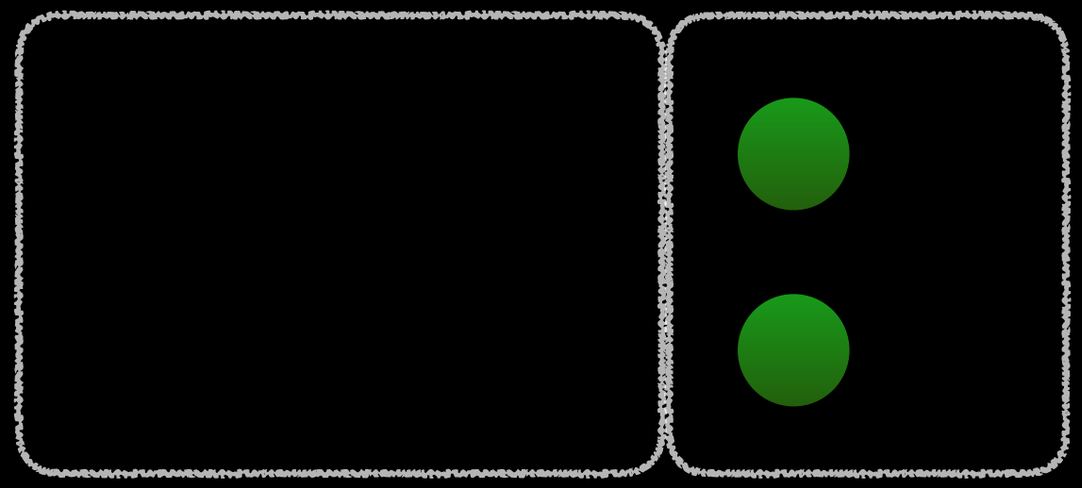
Young Generation



Young Generation



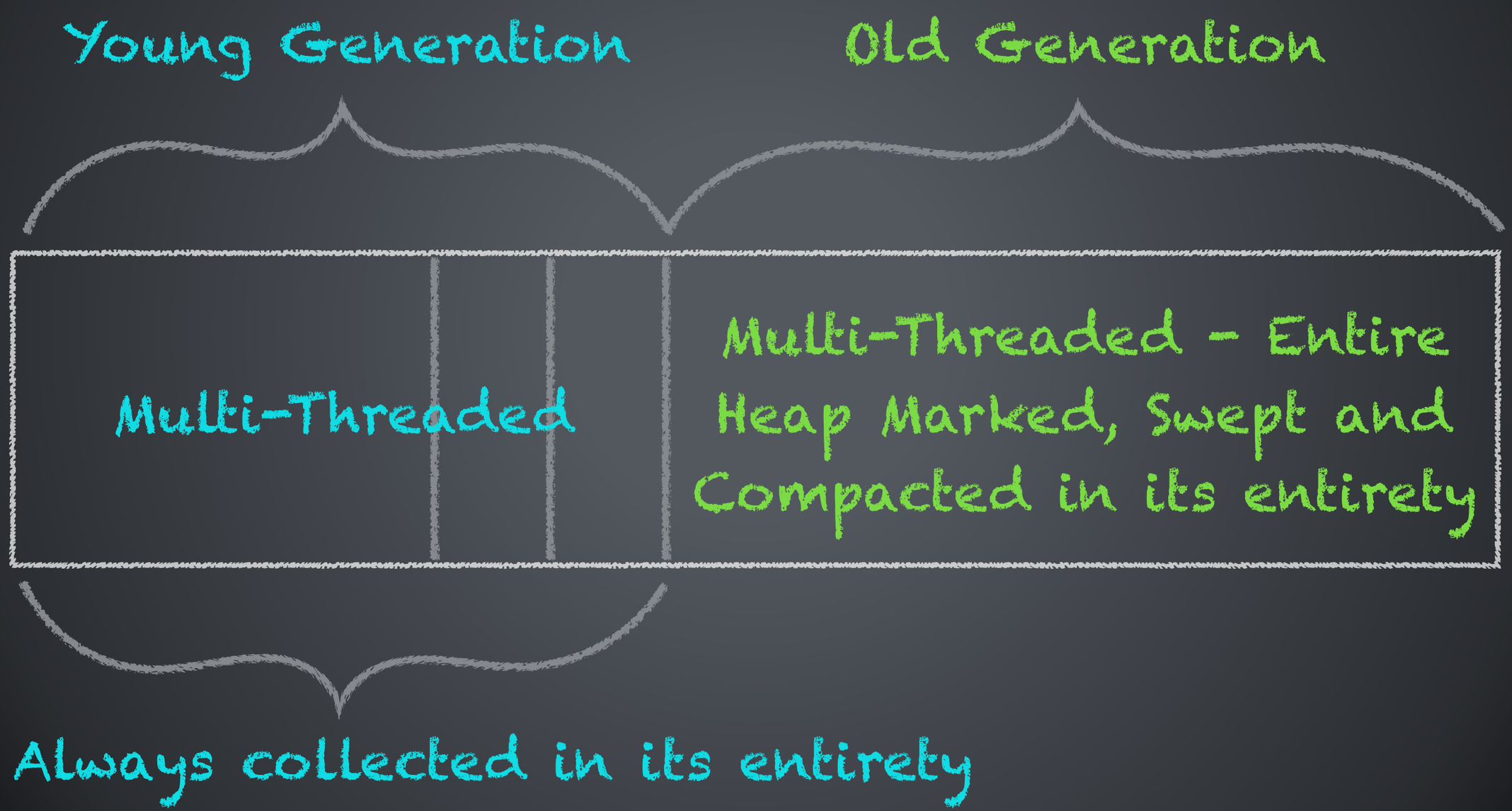
Young Generation



Eden Survivor

Old Garbage Collection == Different GC Algorithms

The Throughput Collector



Young Generation

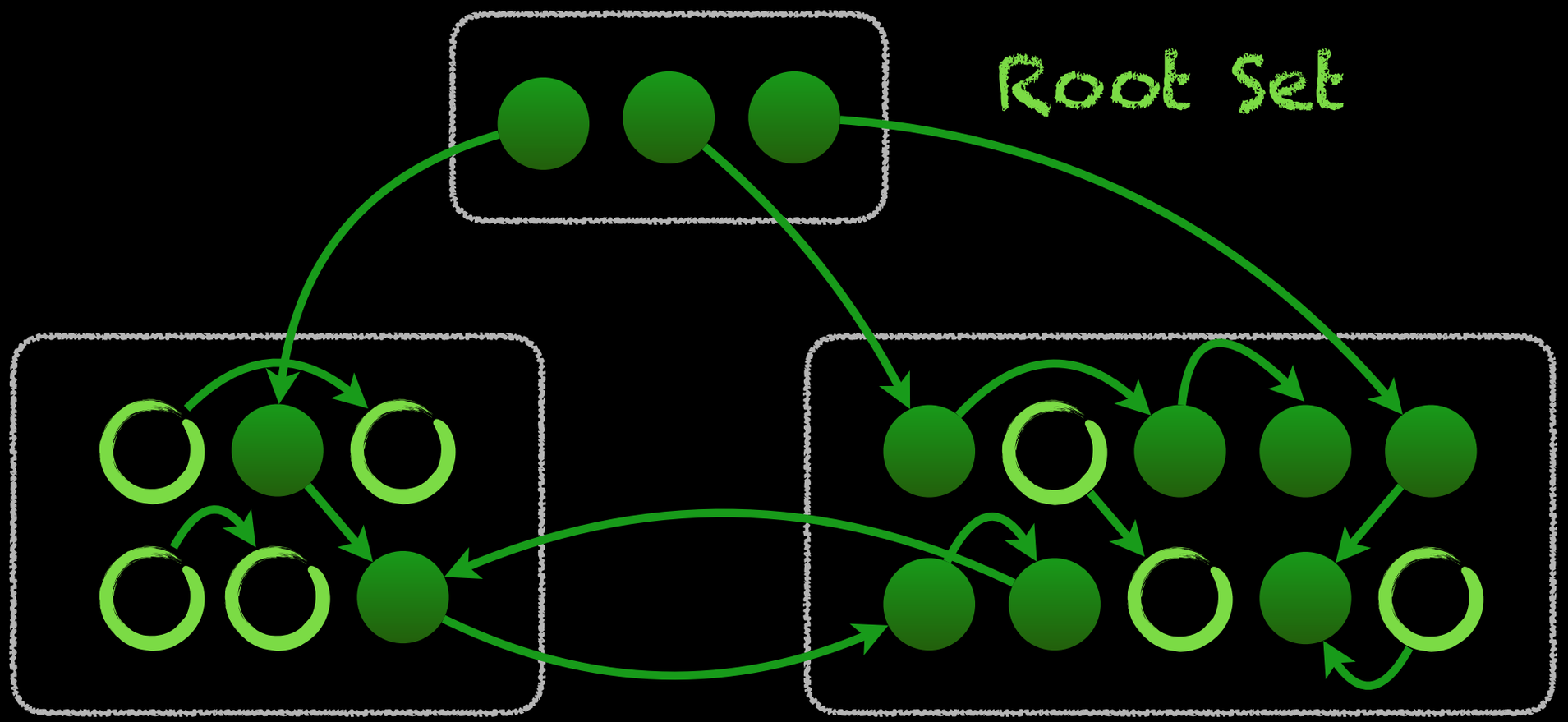
Old Generation

Multi-Threaded

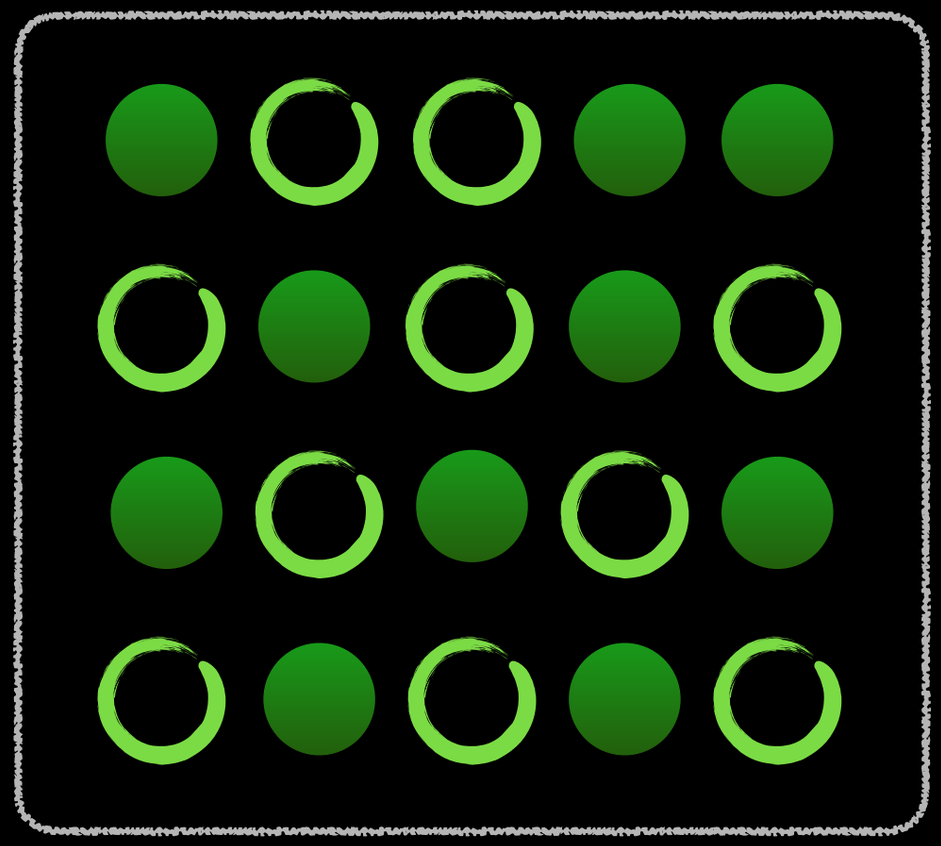
Multi-Threaded - Entire Heap Marked, Swept and Compacted in its entirety

Always collected in its entirety

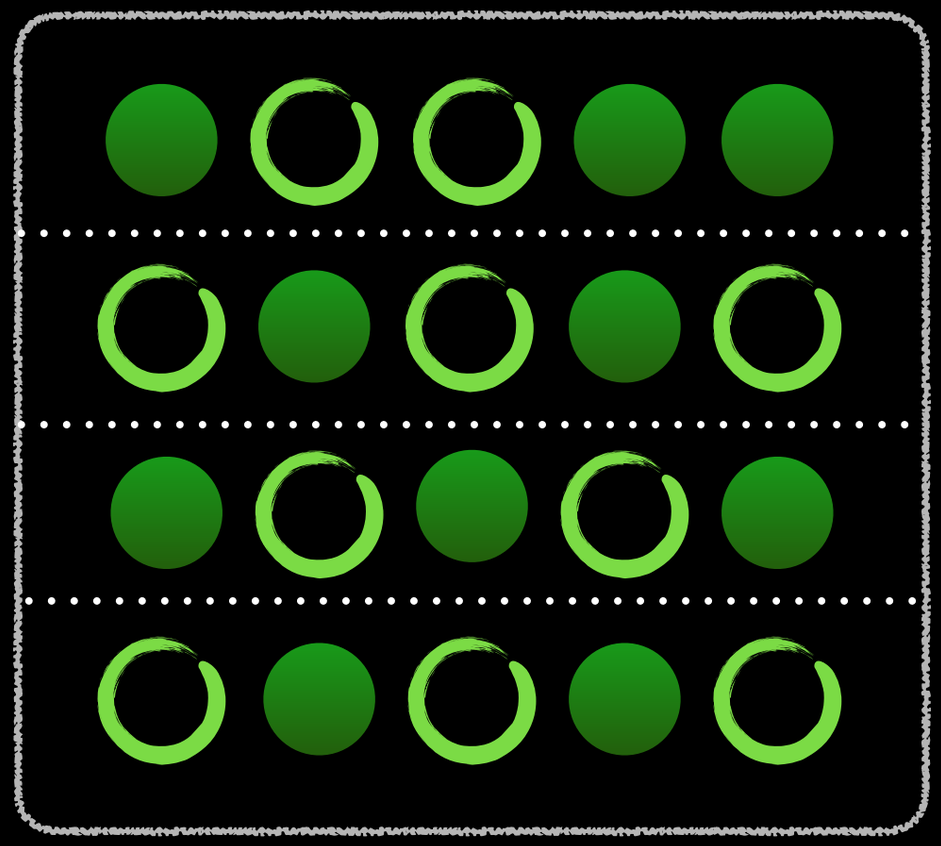
Garbage Collection -Reclamation via Parallel Mark-Compact



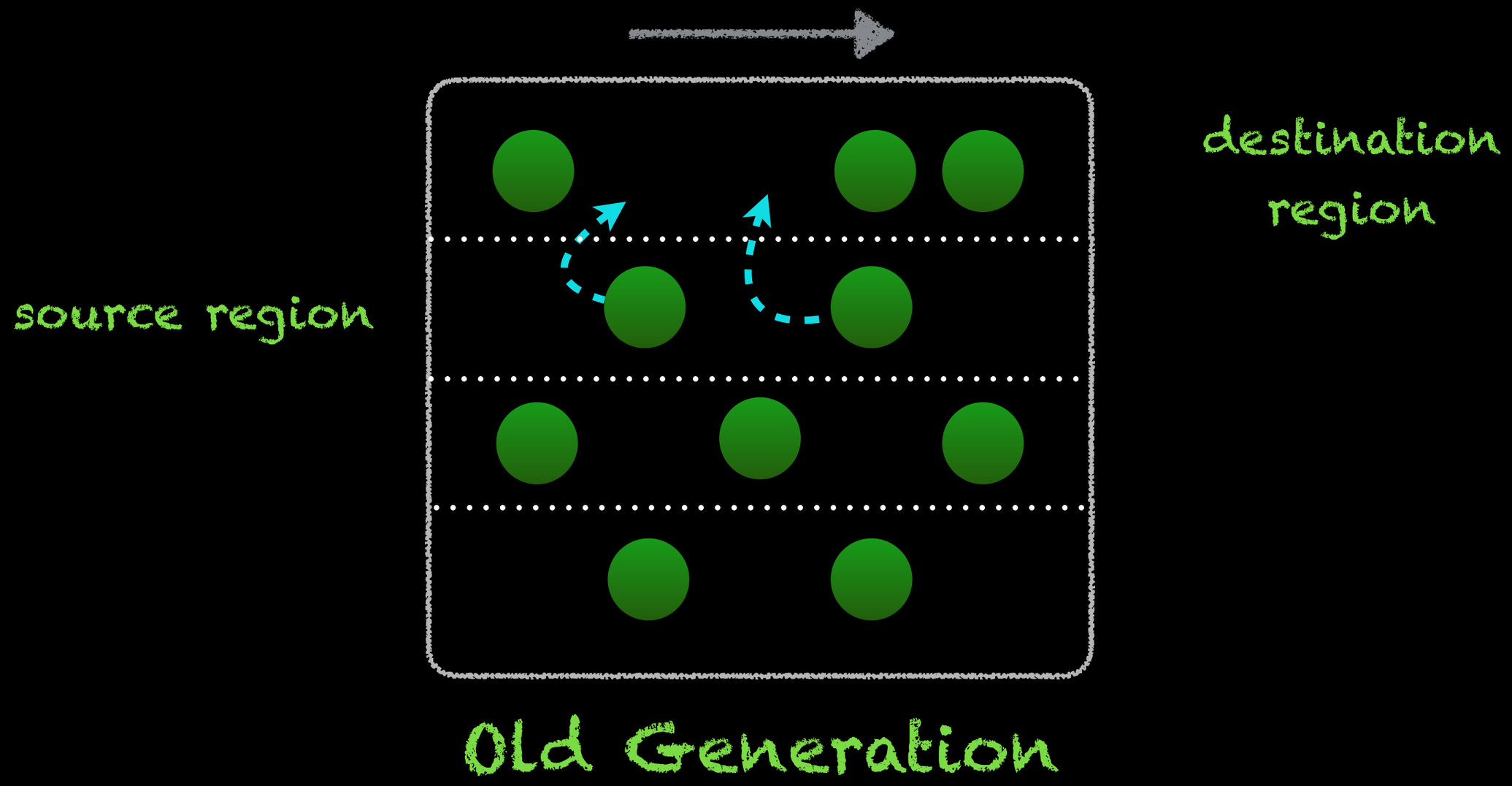
Garbage Collection -Reclamation via Parallel Mark-Compact



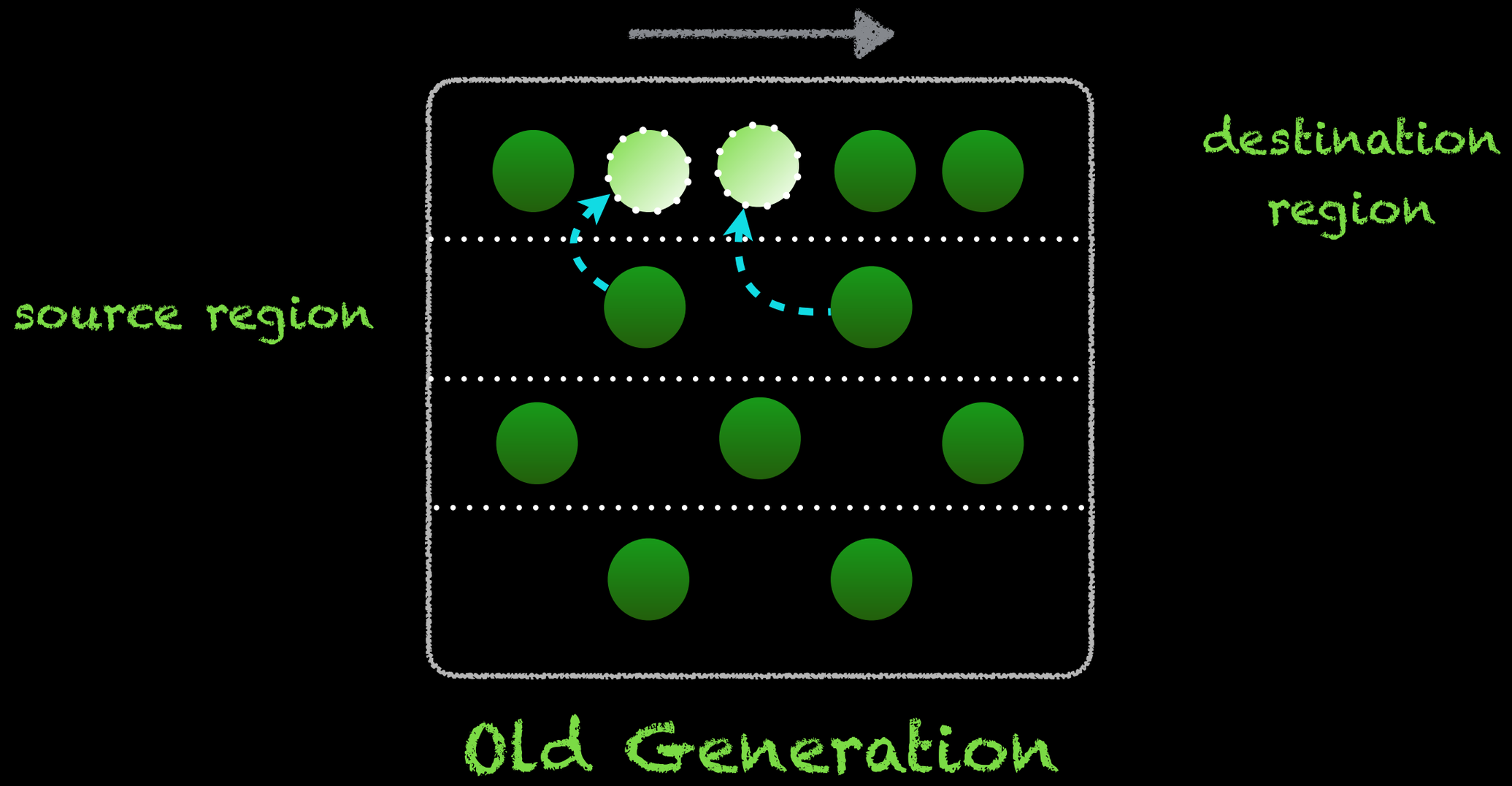
Garbage Collection -Reclamation via Parallel Mark-Compact



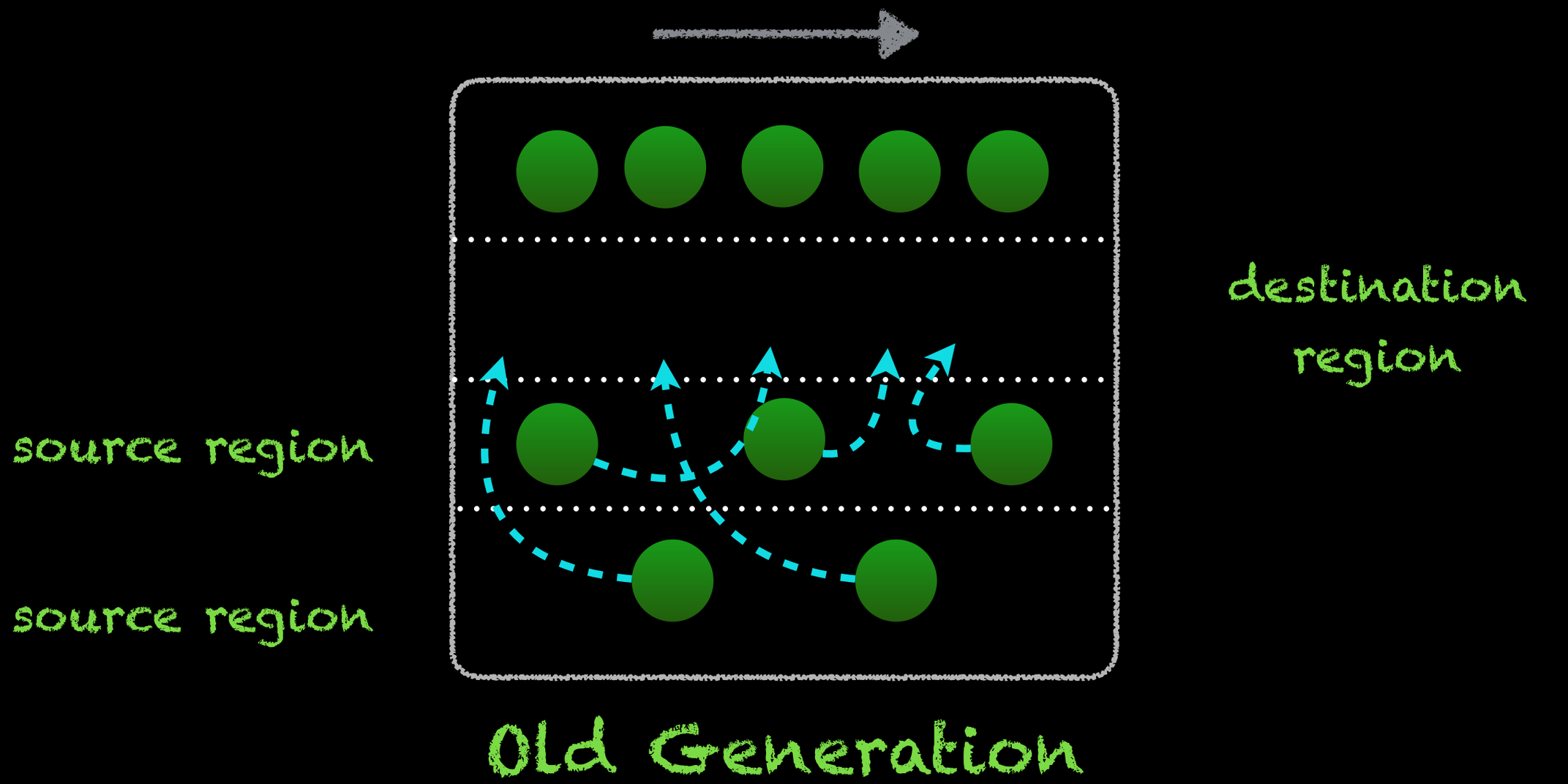
Garbage Collection -Reclamation via Parallel Mark-Compact



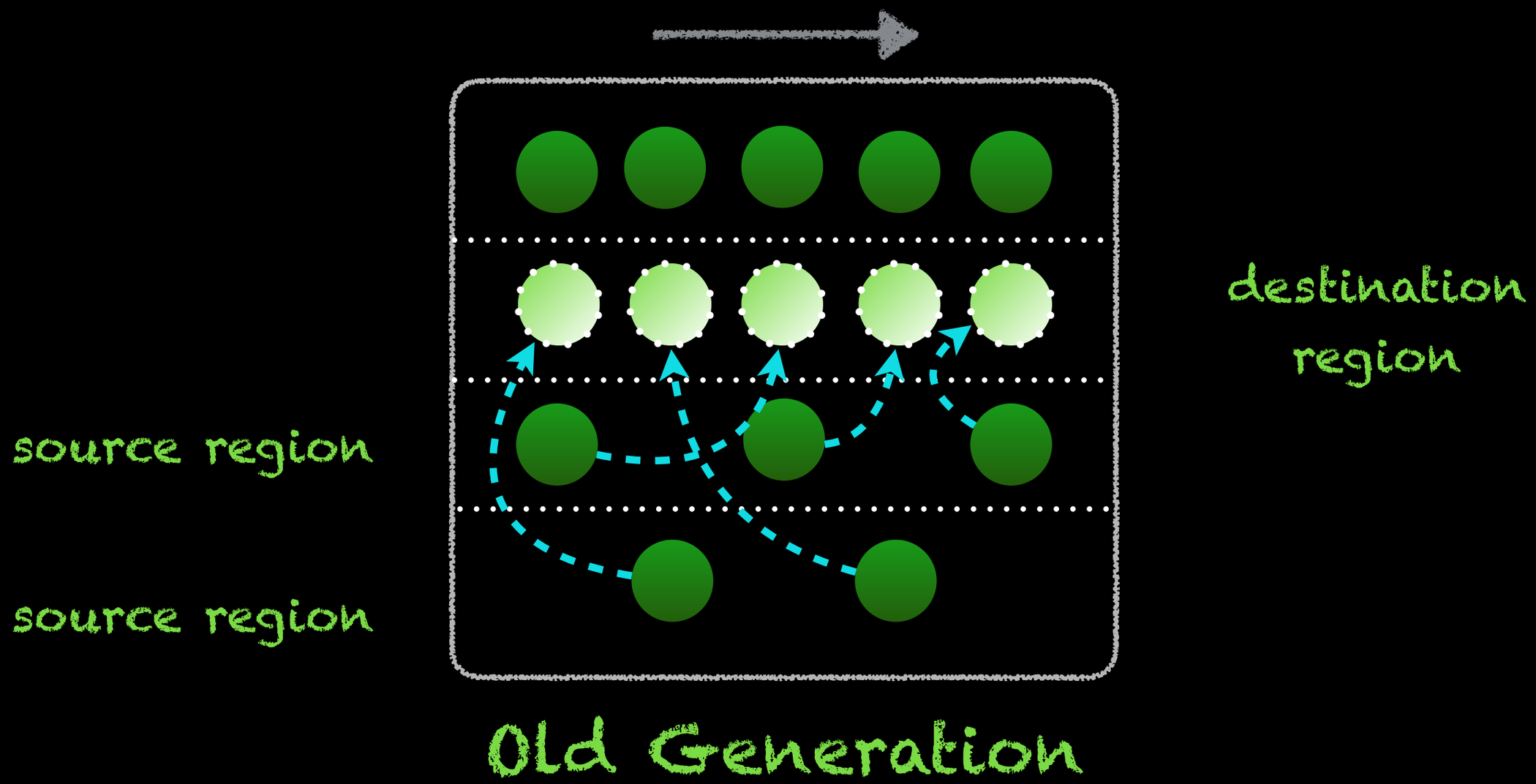
Garbage Collection -Reclamation via Parallel Mark-Compact



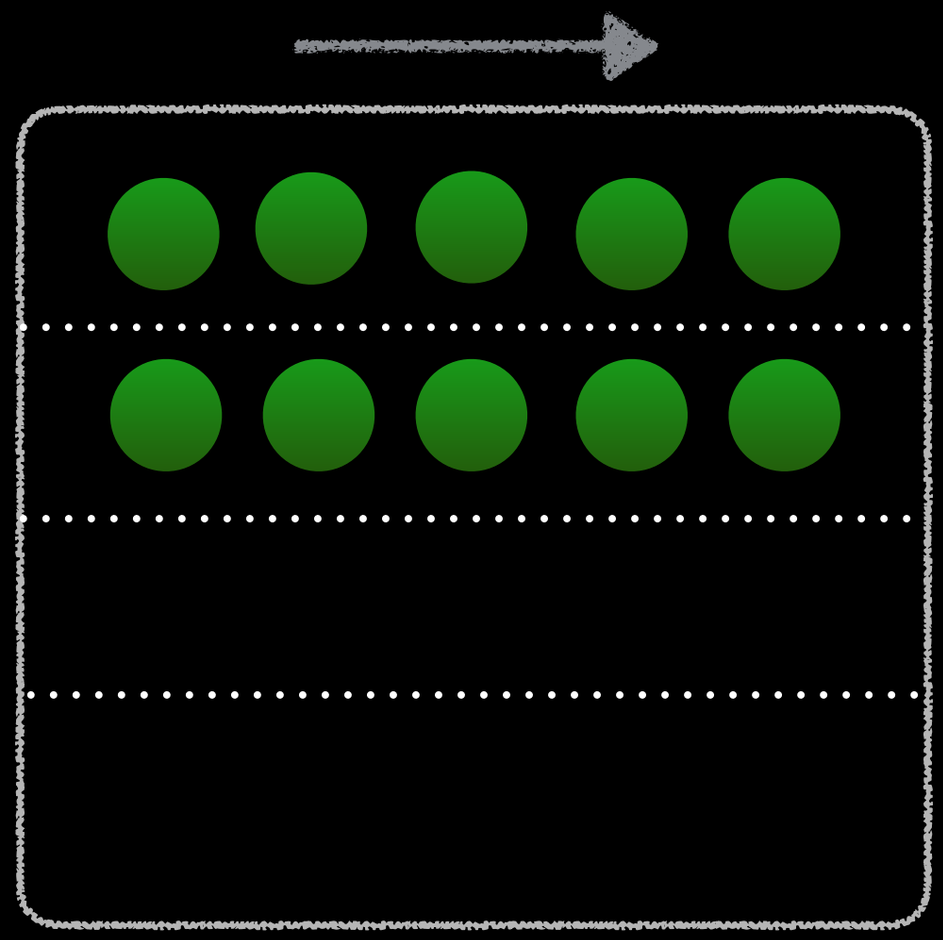
Garbage Collection -Reclamation via Parallel Mark-Compact



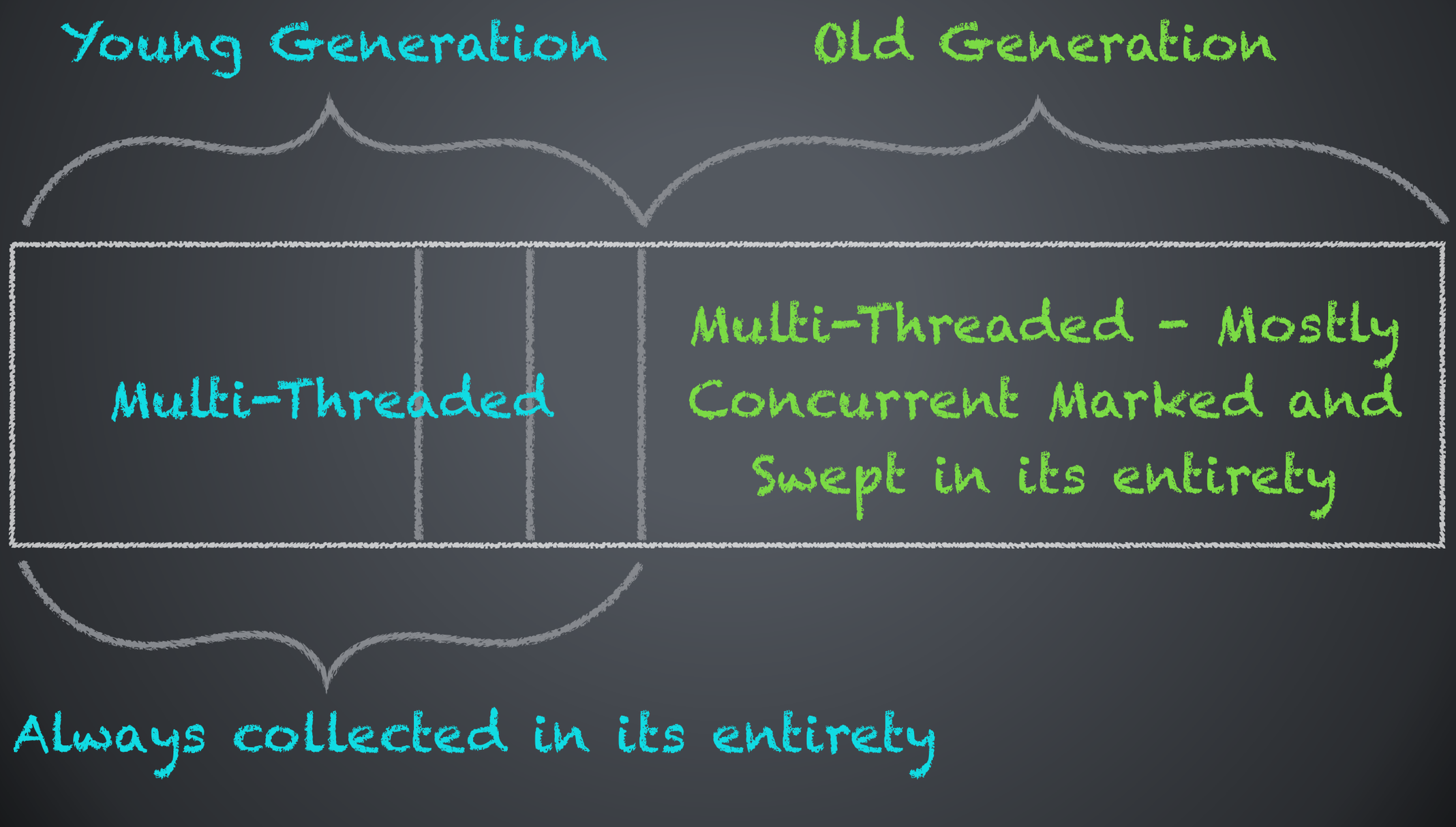
Garbage Collection -Reclamation via Parallel Mark-Compact



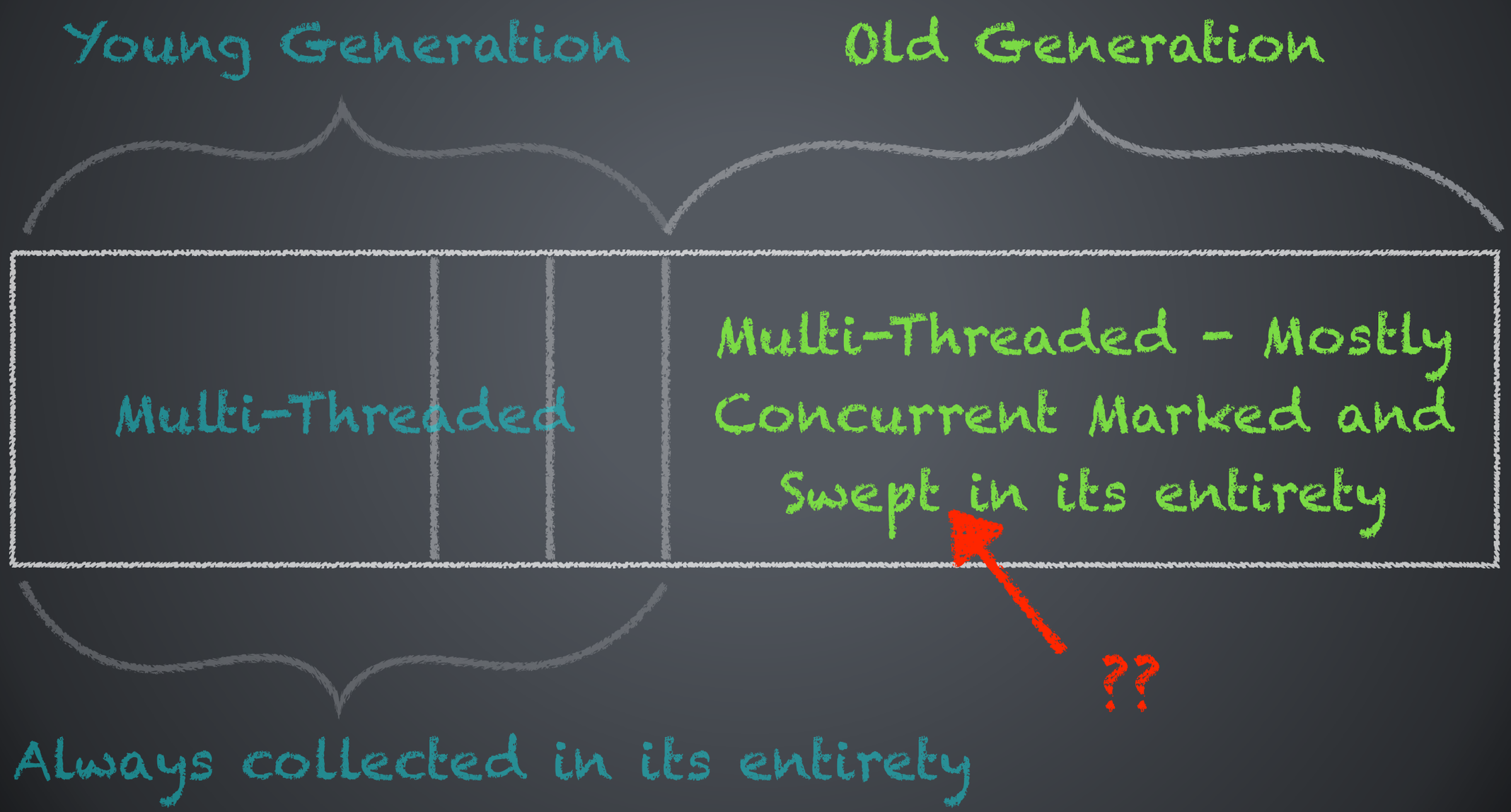
Garbage Collection -Reclamation via Parallel Mark-Compact



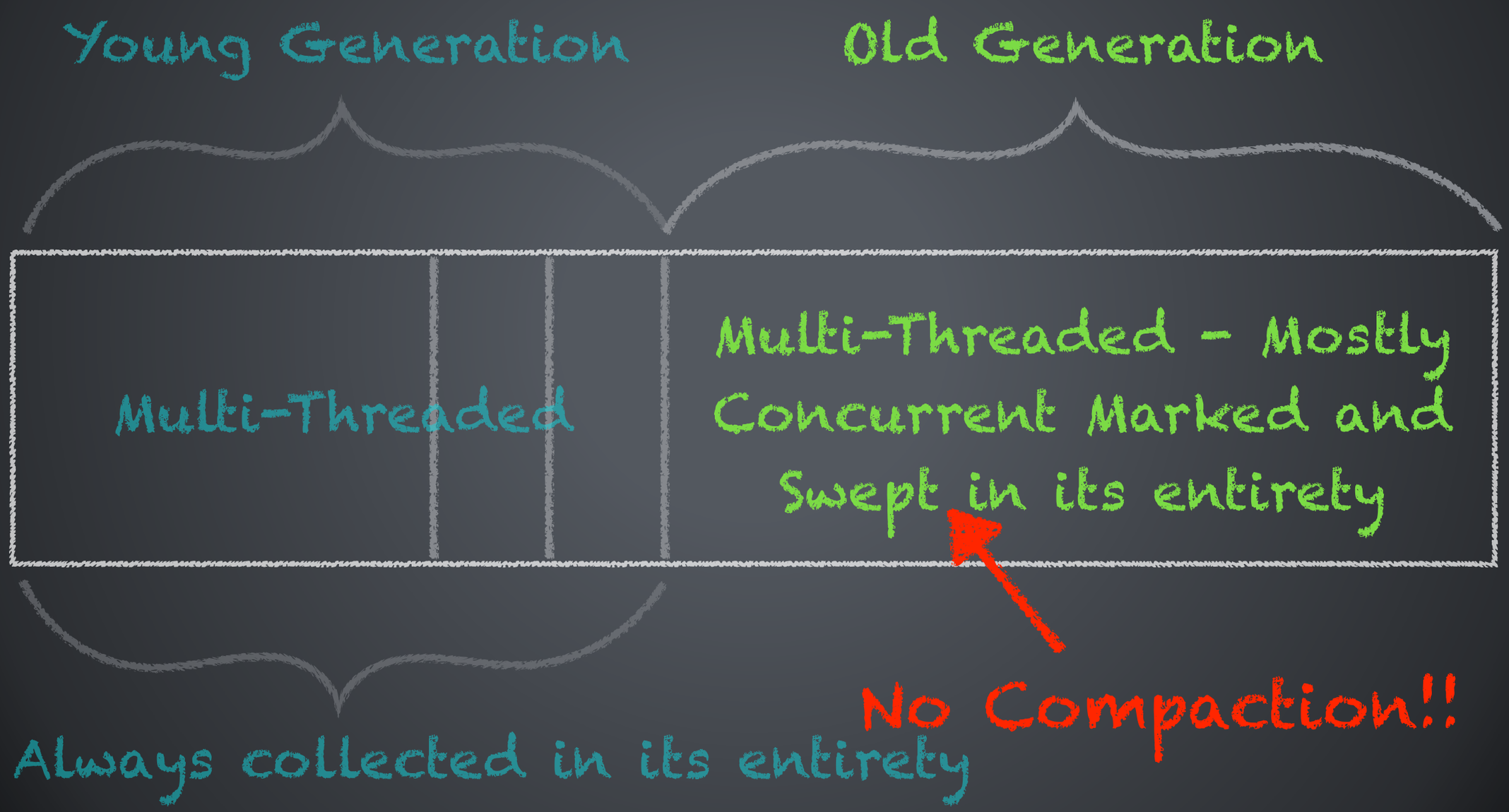
The CMS Collector



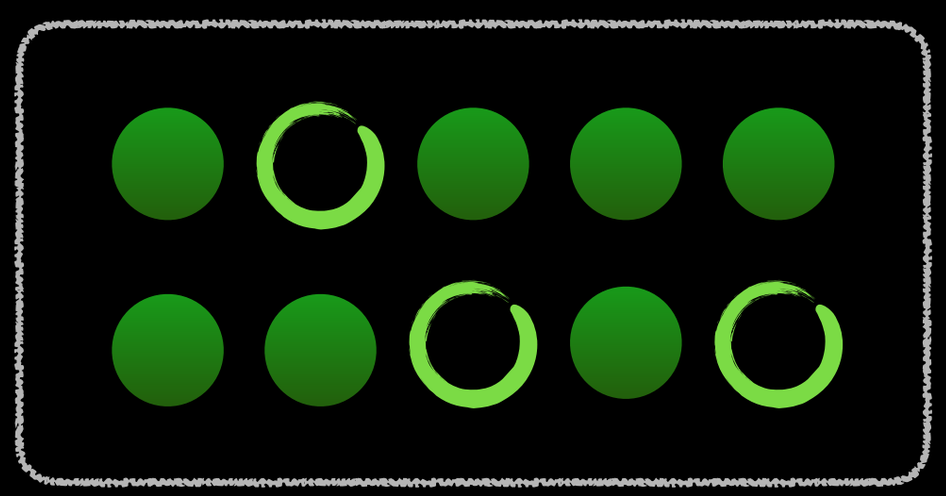
The CMS Collector



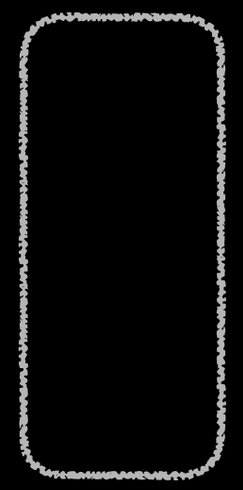
The CMS Collector



Garbage Collection - Reclamation via Mark-Sweep

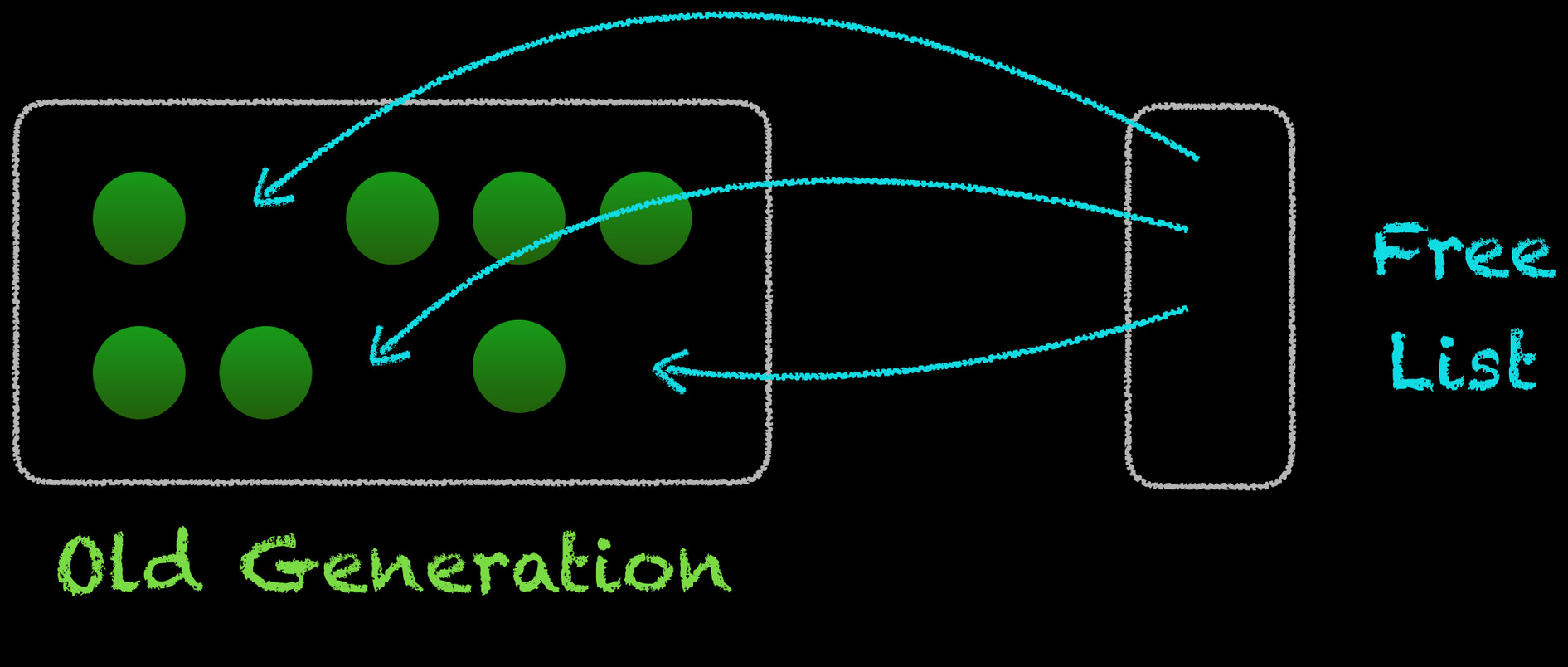


Old Generation



Free List

Garbage Collection - Reclamation via Mark-Sweep

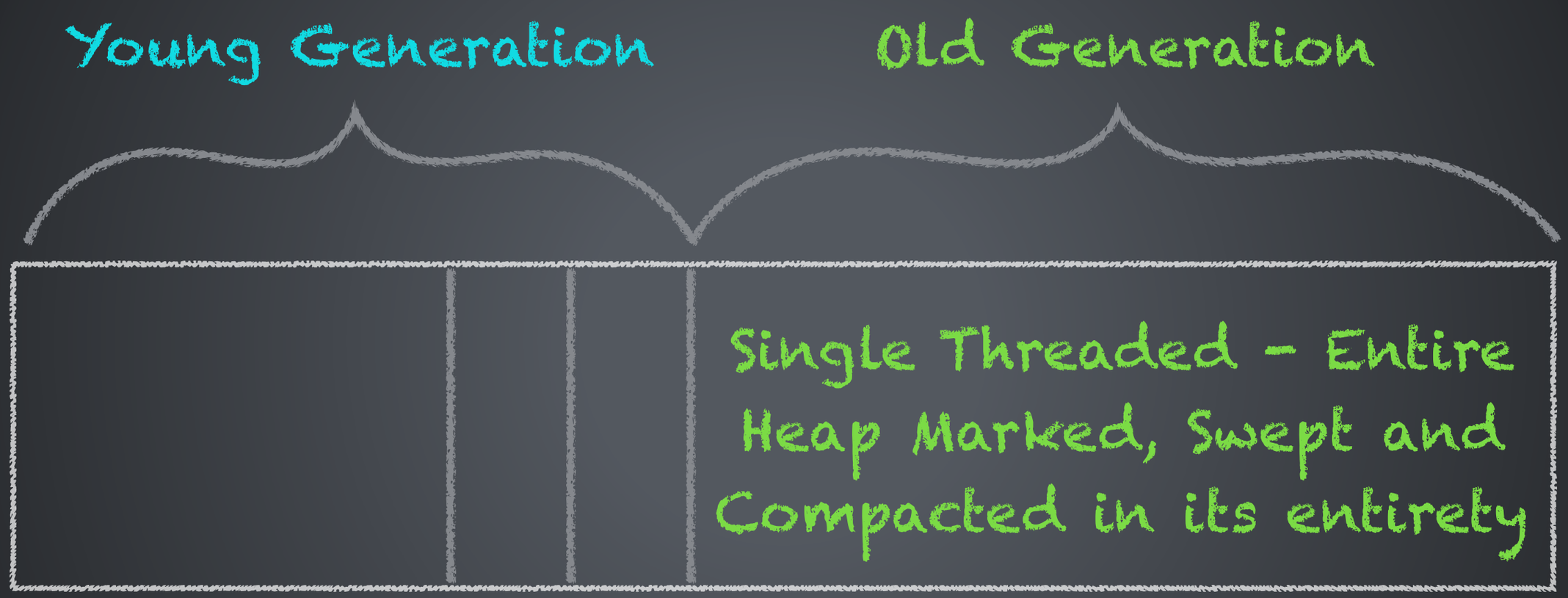


GC Fact!

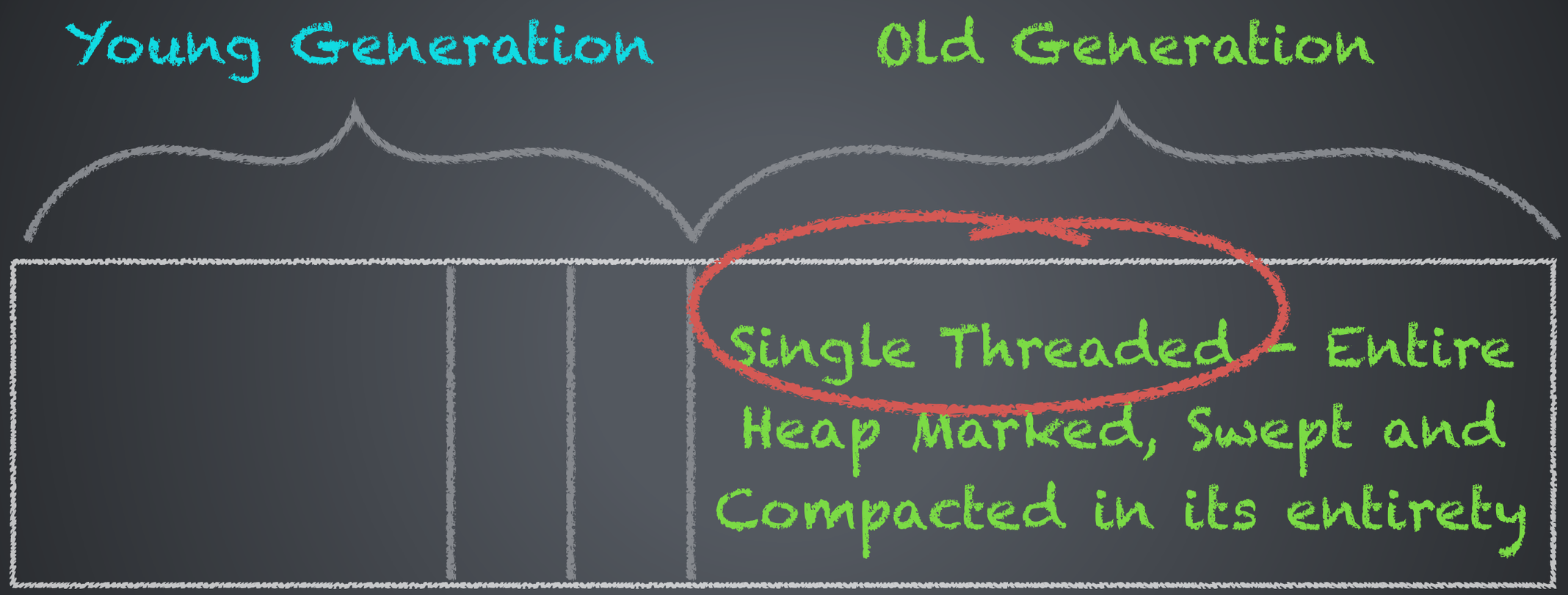
All non/partial compacting GCs in OpenJDK HotSpot fallback* to a fully compacting stop-the-world garbage collection called the "full" GC.

*Tuning can help avoid or postpone full GCs in many cases.

Garbage Collection -Reclamation via Mark-Sweep-Compact

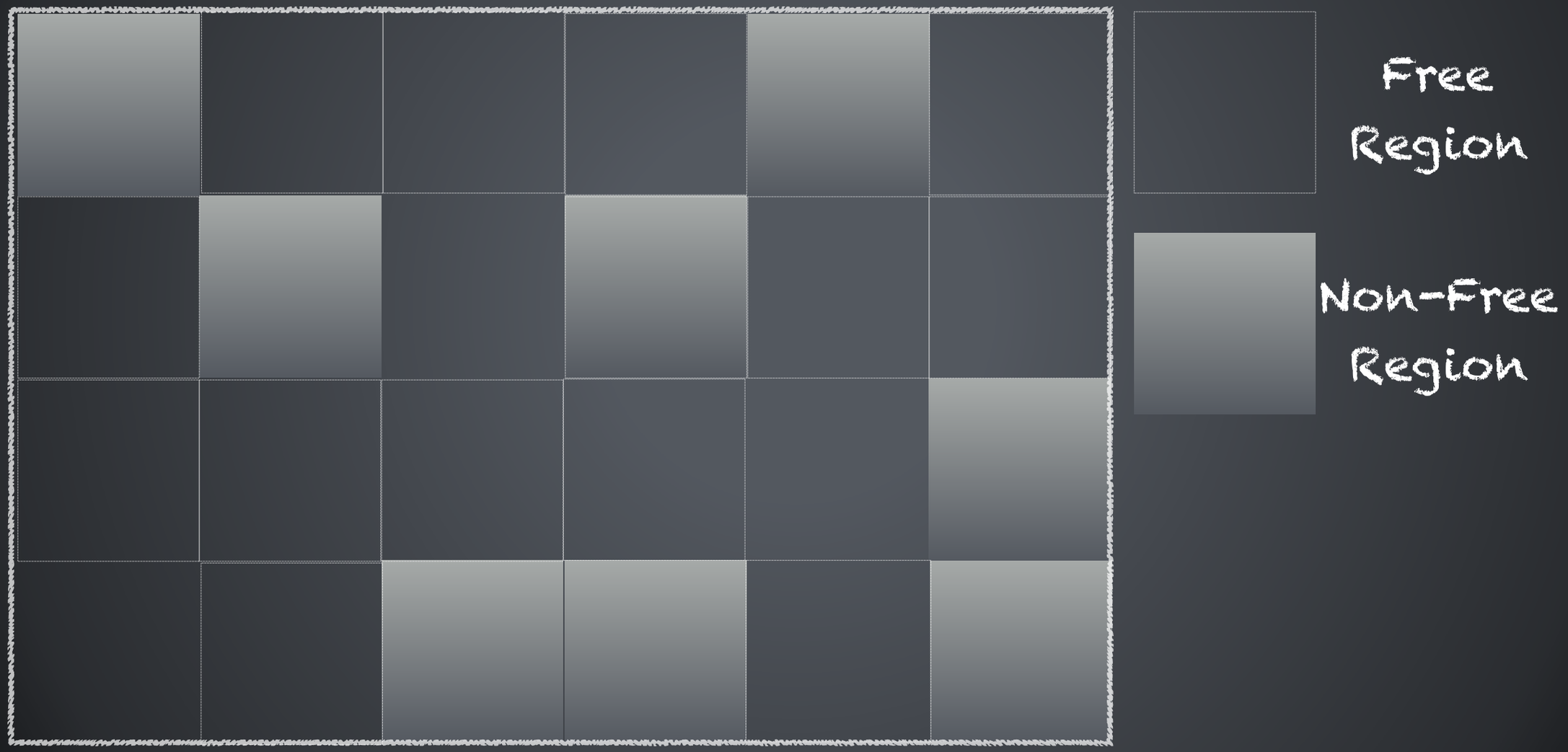


Garbage Collection -Reclamation via Mark-Sweep-Compact

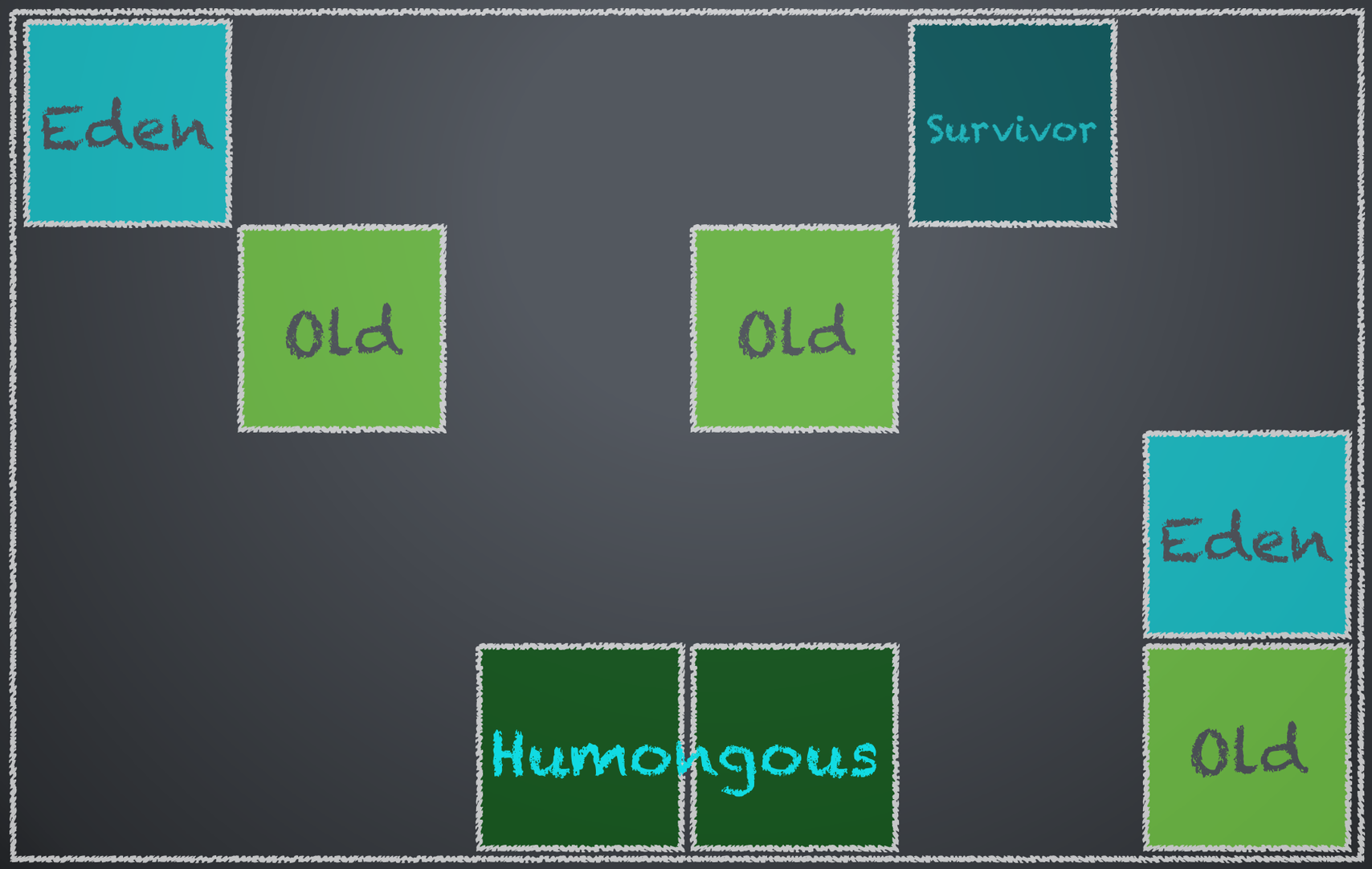


What About Garbage First GC?

Heap Regions

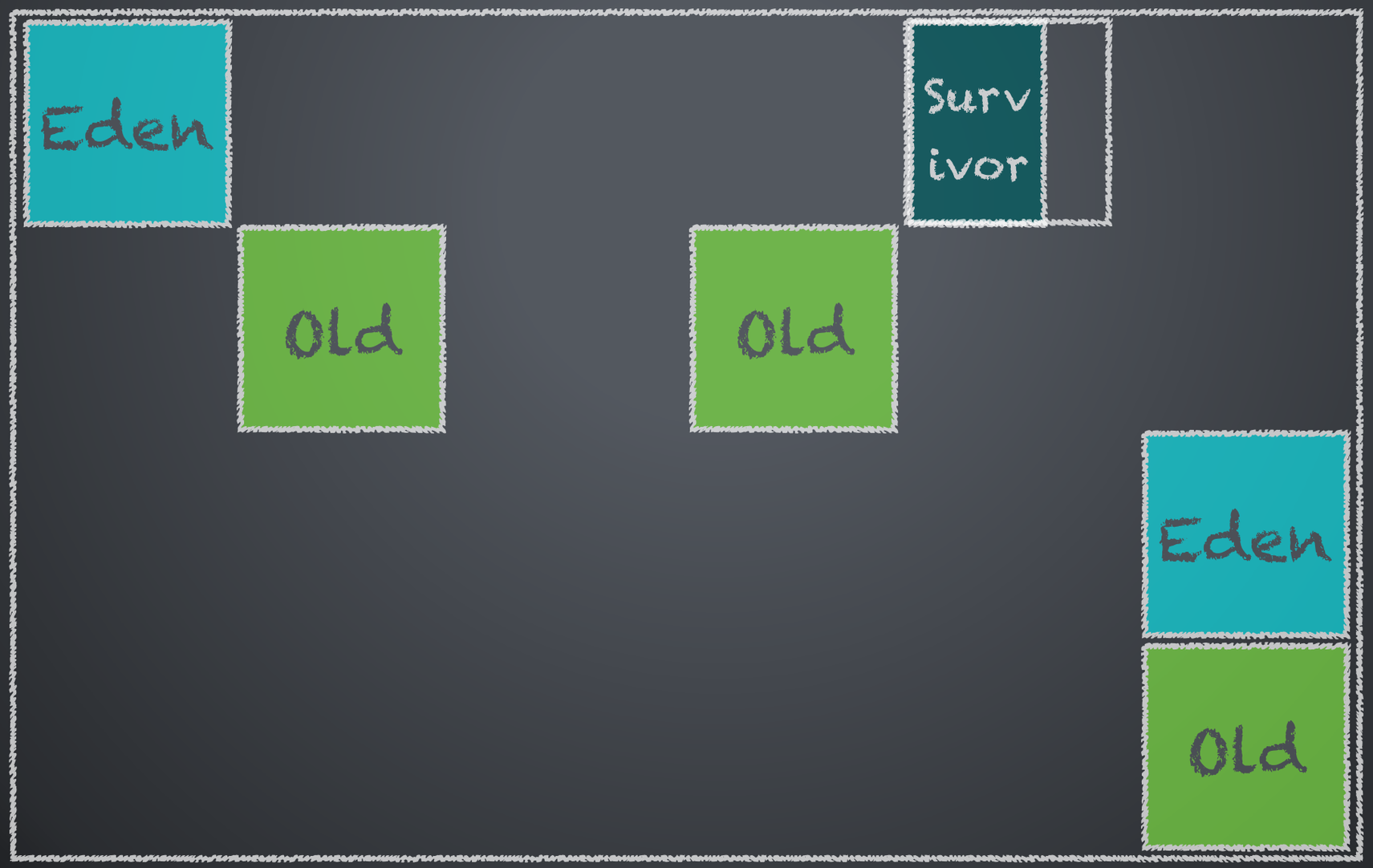


The Garbage First Collector - Regionalized Heap

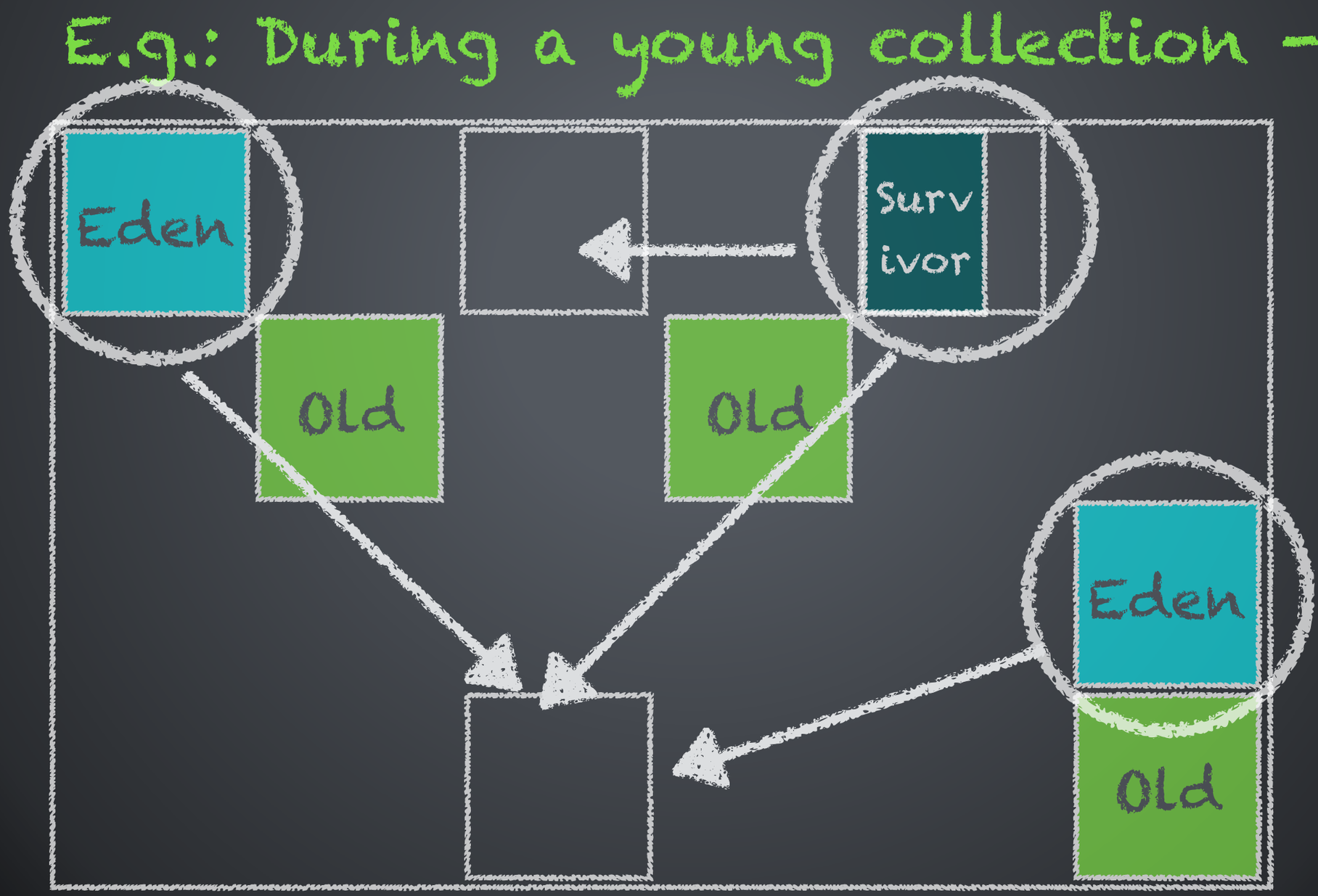


The Garbage First Collector

E.g.: Current heap configuration -

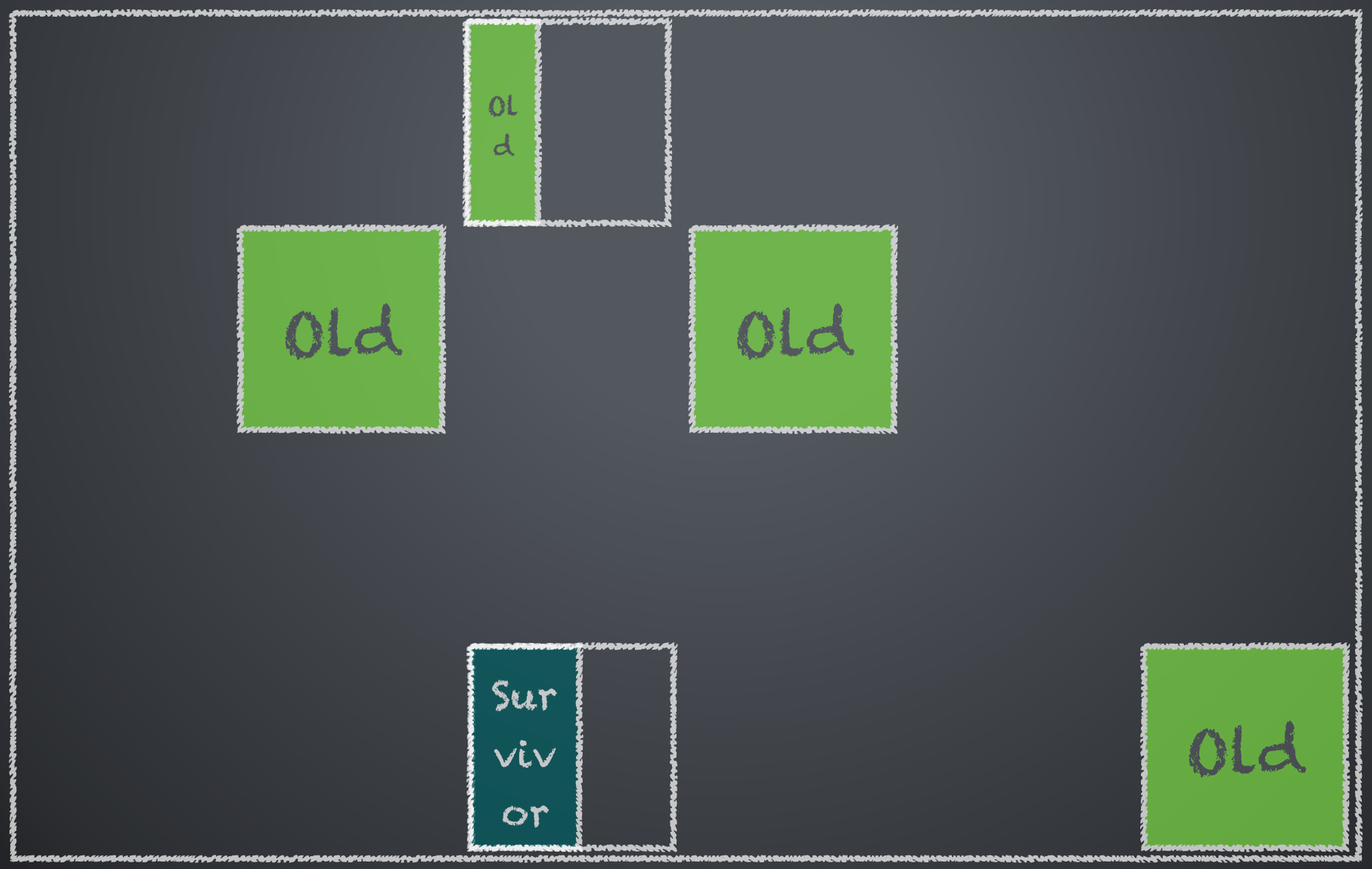


The Garbage First Collector



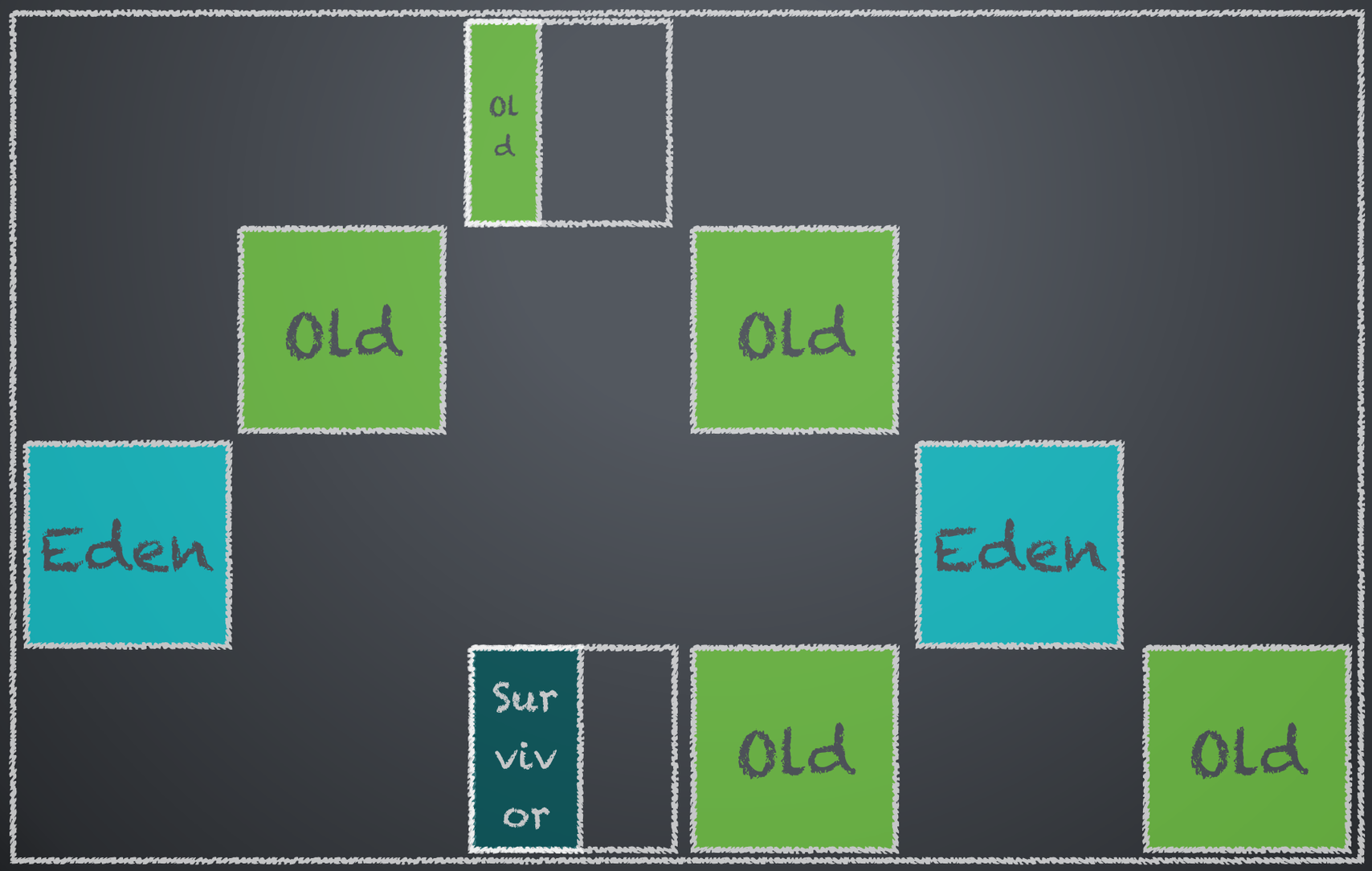
The Garbage First Collector

E.g.: After a young collection -



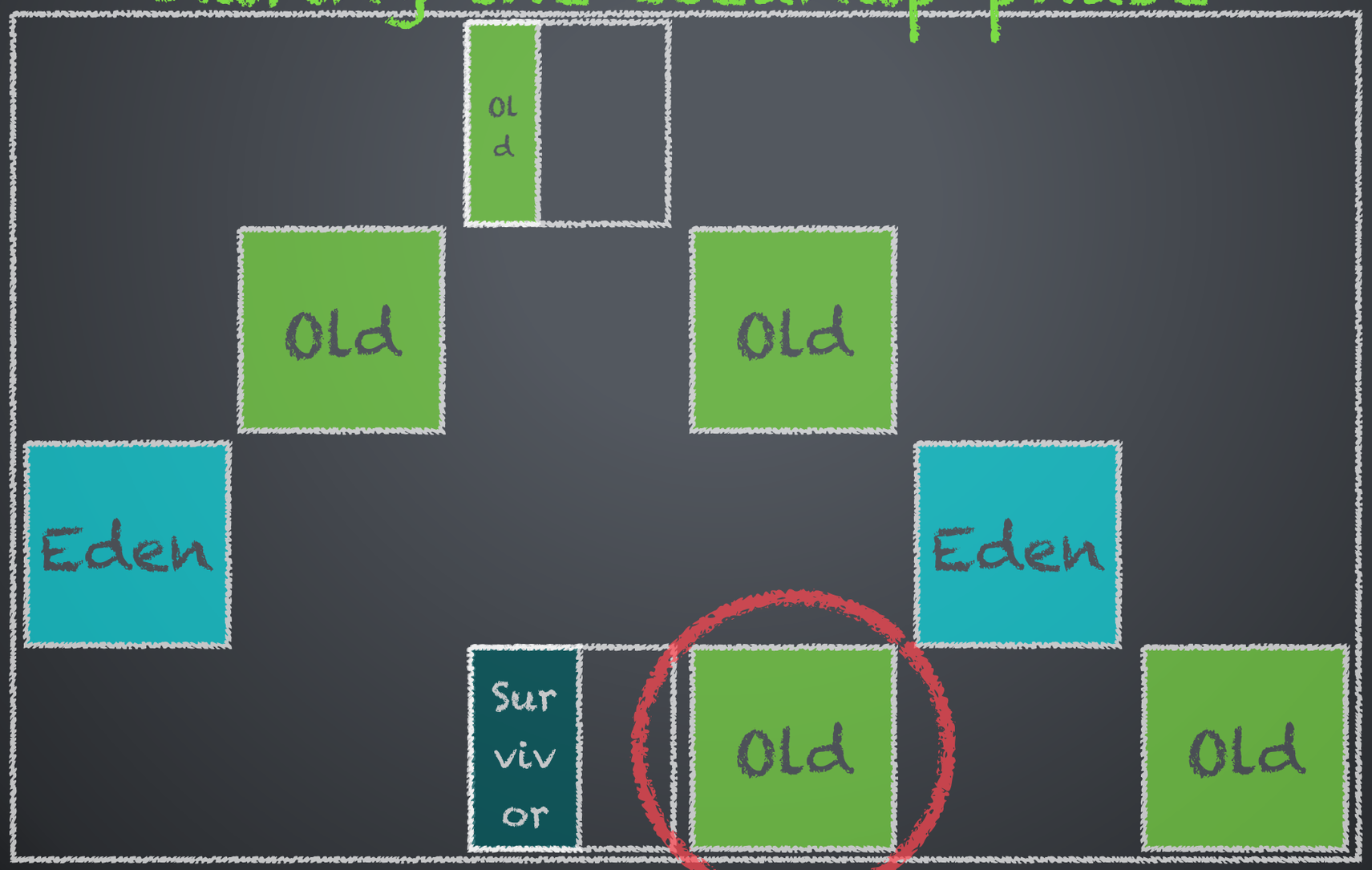
The Garbage First Collector

E.g.: Current heap configuration -



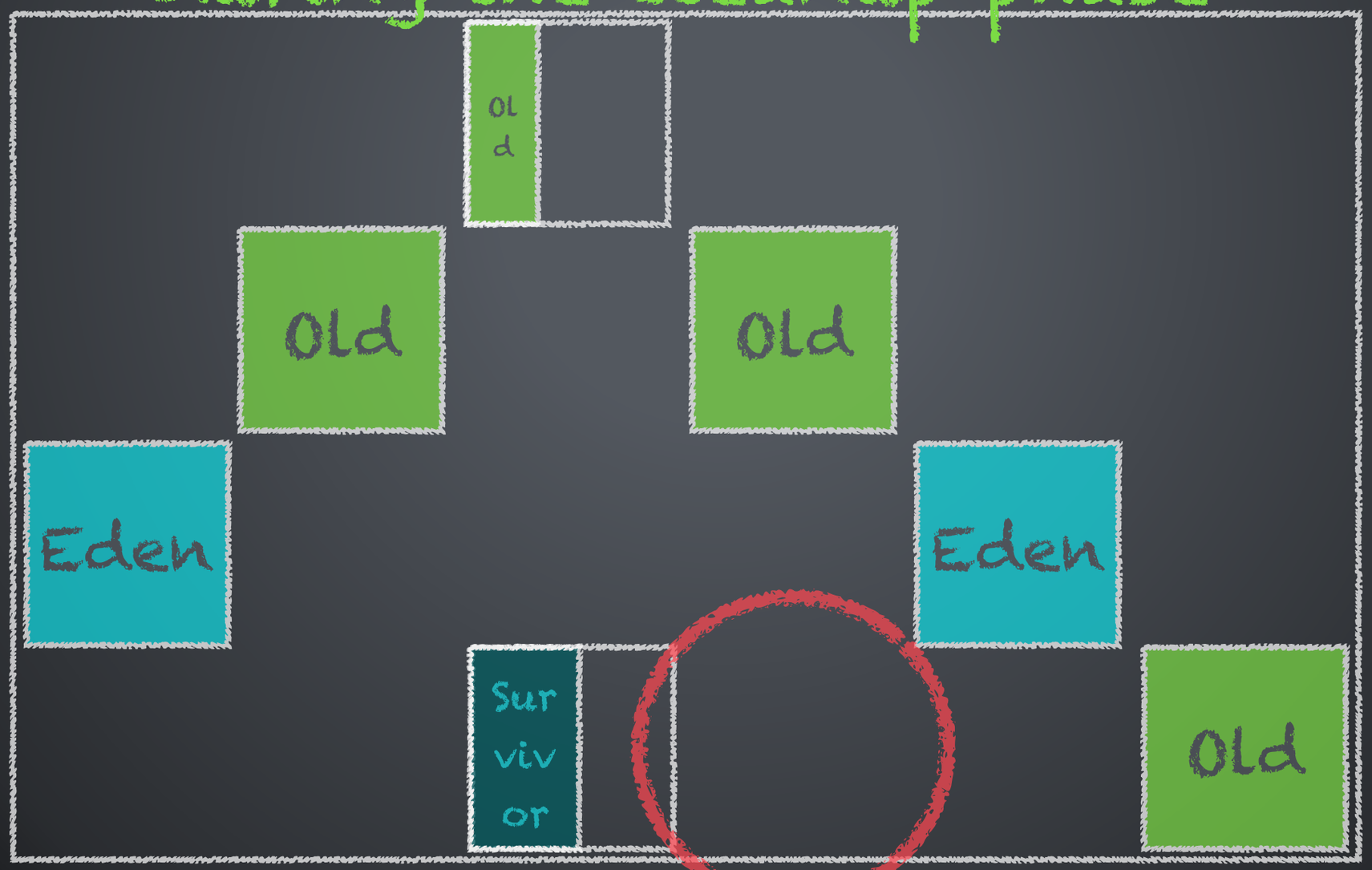
The Garbage First Collector

E.g.: Reclamation of a garbage-filled region during the cleanup phase -



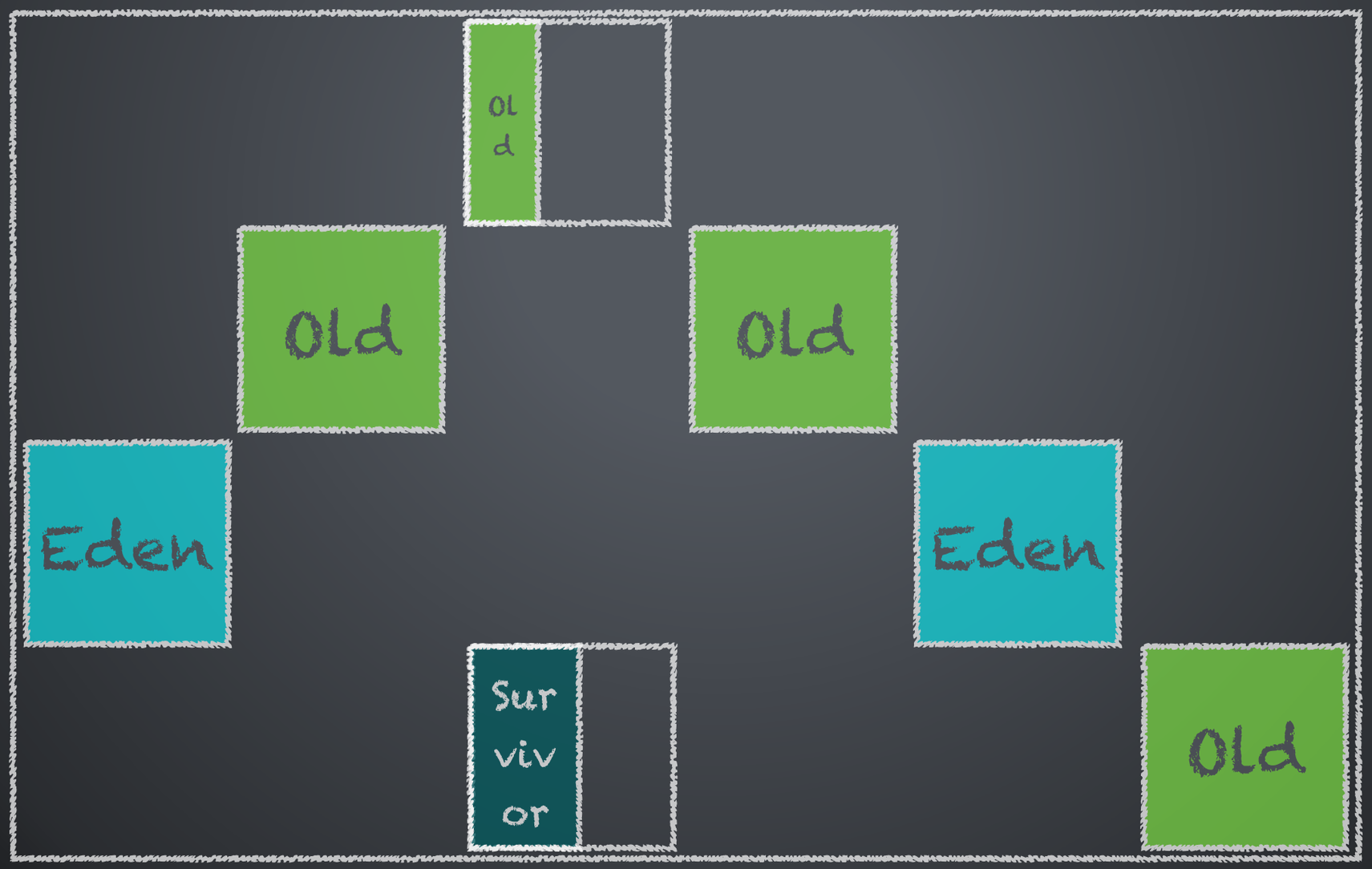
The Garbage First Collector

E.g.: Reclamation of a garbage-filled region during the cleanup phase -



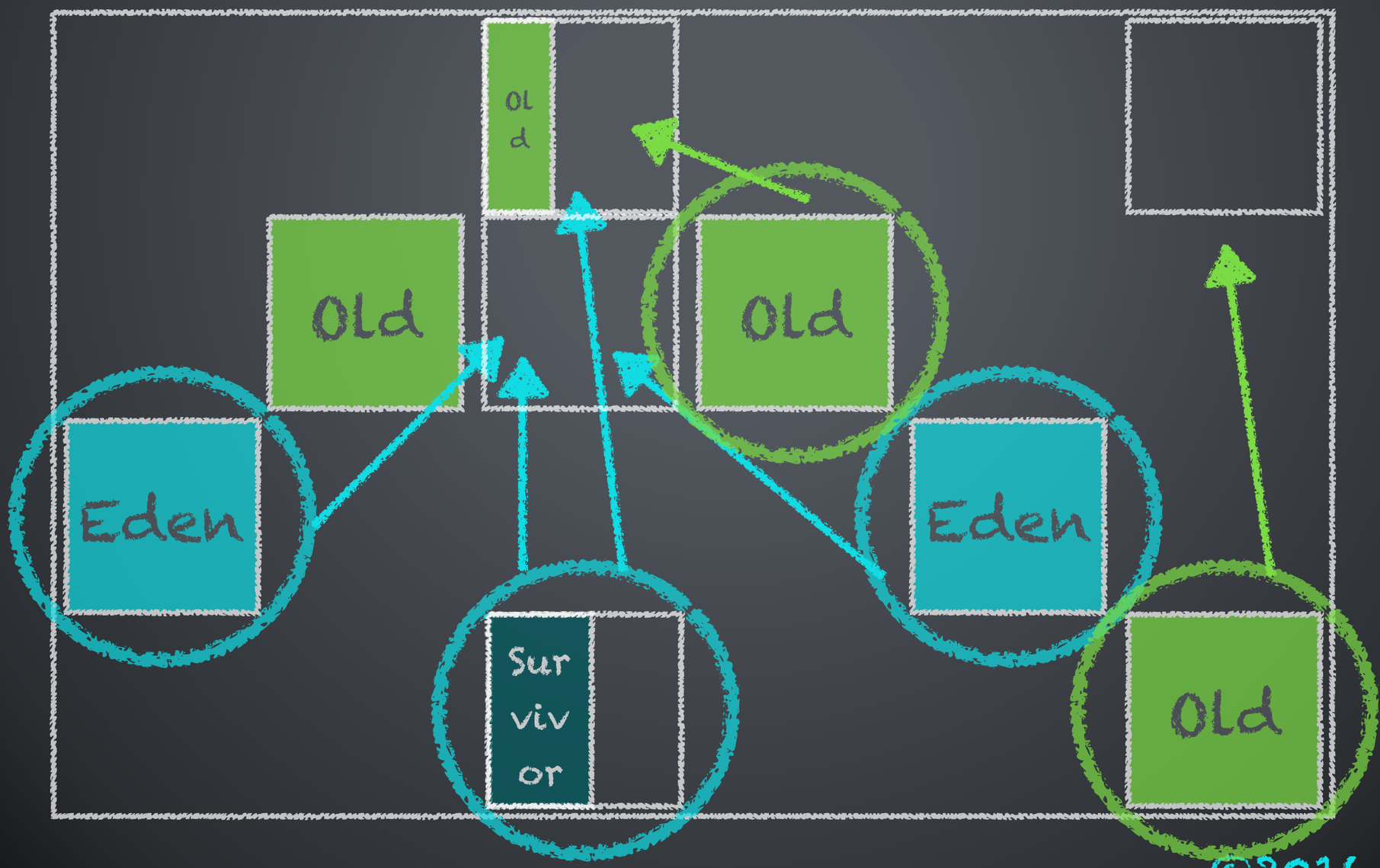
The Garbage First Collector

E.g.: Current heap configuration -



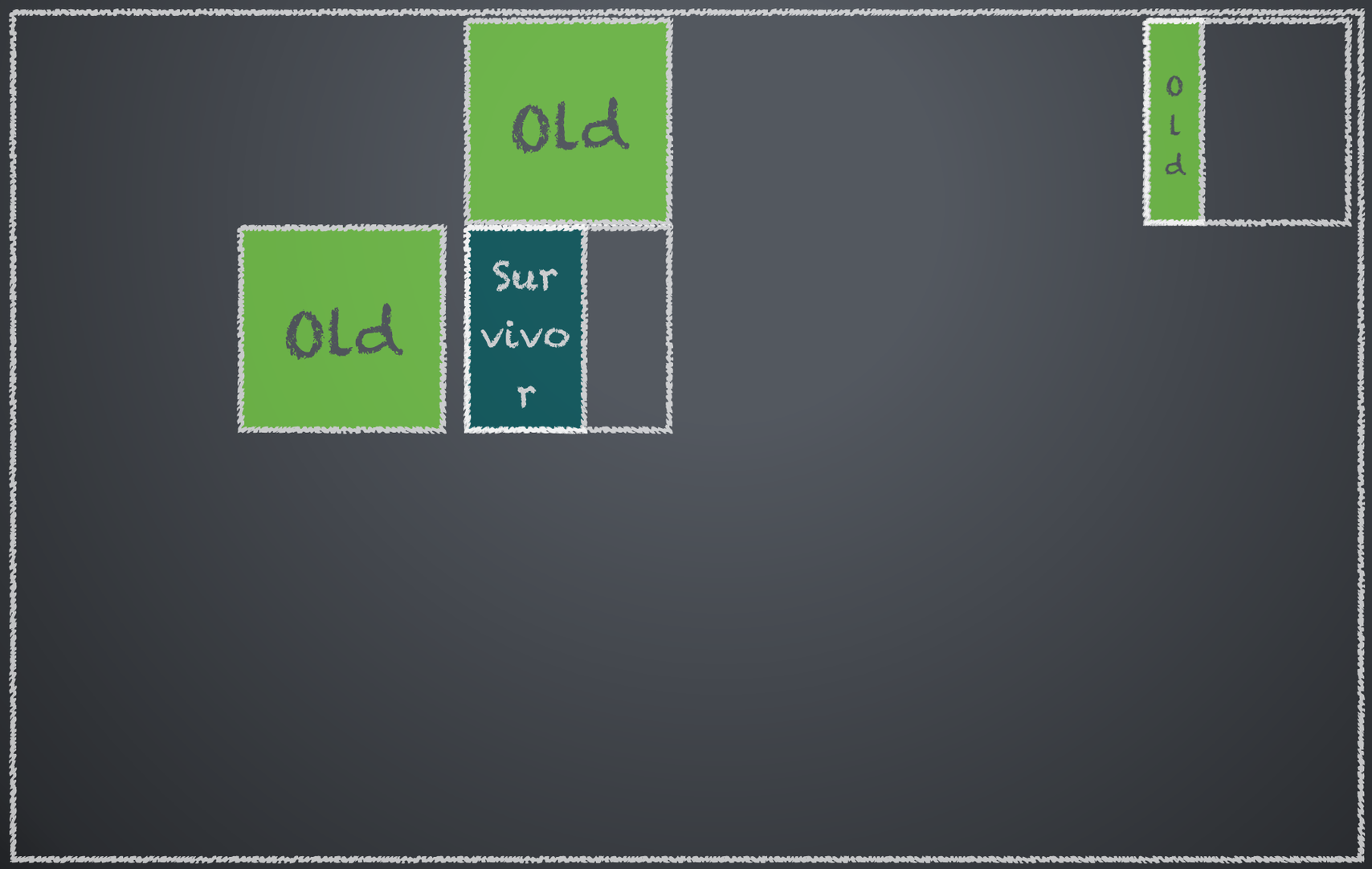
The Garbage First Collector

E.g.: During a mixed collection -

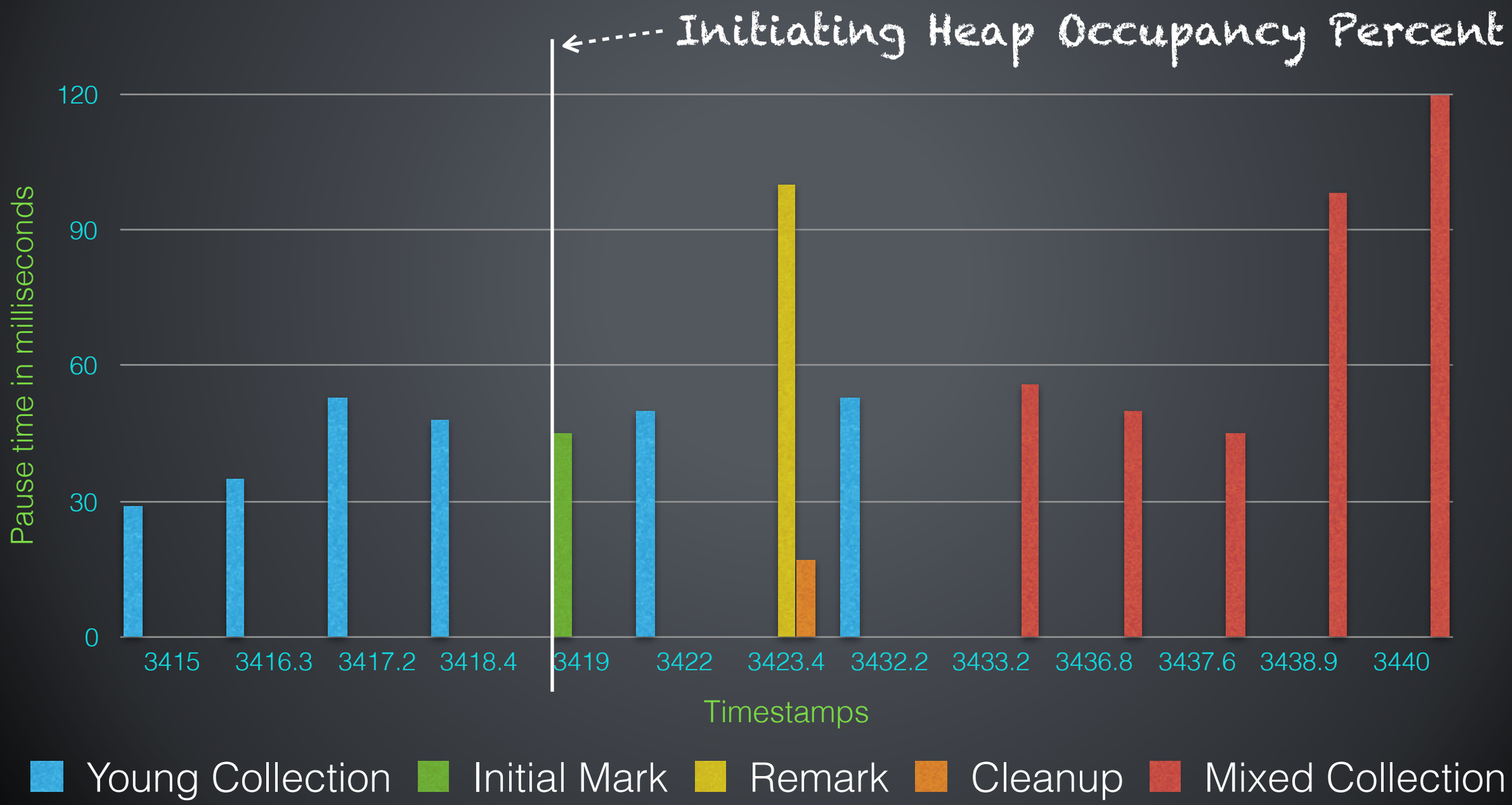


The Garbage First Collector

E.g.: After a mixed collection -



The Garbage First Collector - Pause Histogram



Finally, Let's Talk About Tuning! :)

Tuning GC For Throughput Or For Latency??

GC Elapsed Time Or GC Overhead

GC Elapsed Time indicated the amount of time it takes to execute stop the world GC events

The higher the GC elapsed time - the lower the application responsiveness due to the GC induced latencies

Overhead is an indication of the frequency of stop the world GC events.

The more frequent the GC events - The more likely it is to negatively impact application throughput

What's The #1 Contributor To A GC Elapsed Time?

Copying Costs!

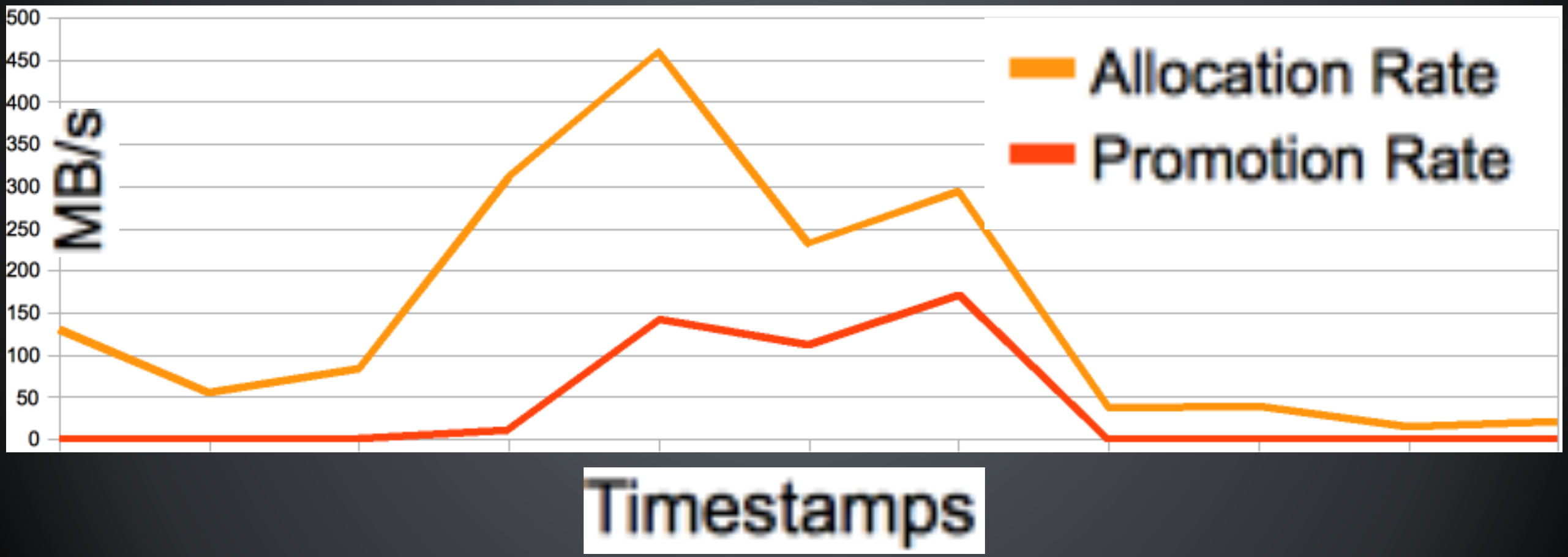
What Are The Contributors To GC Overhead?

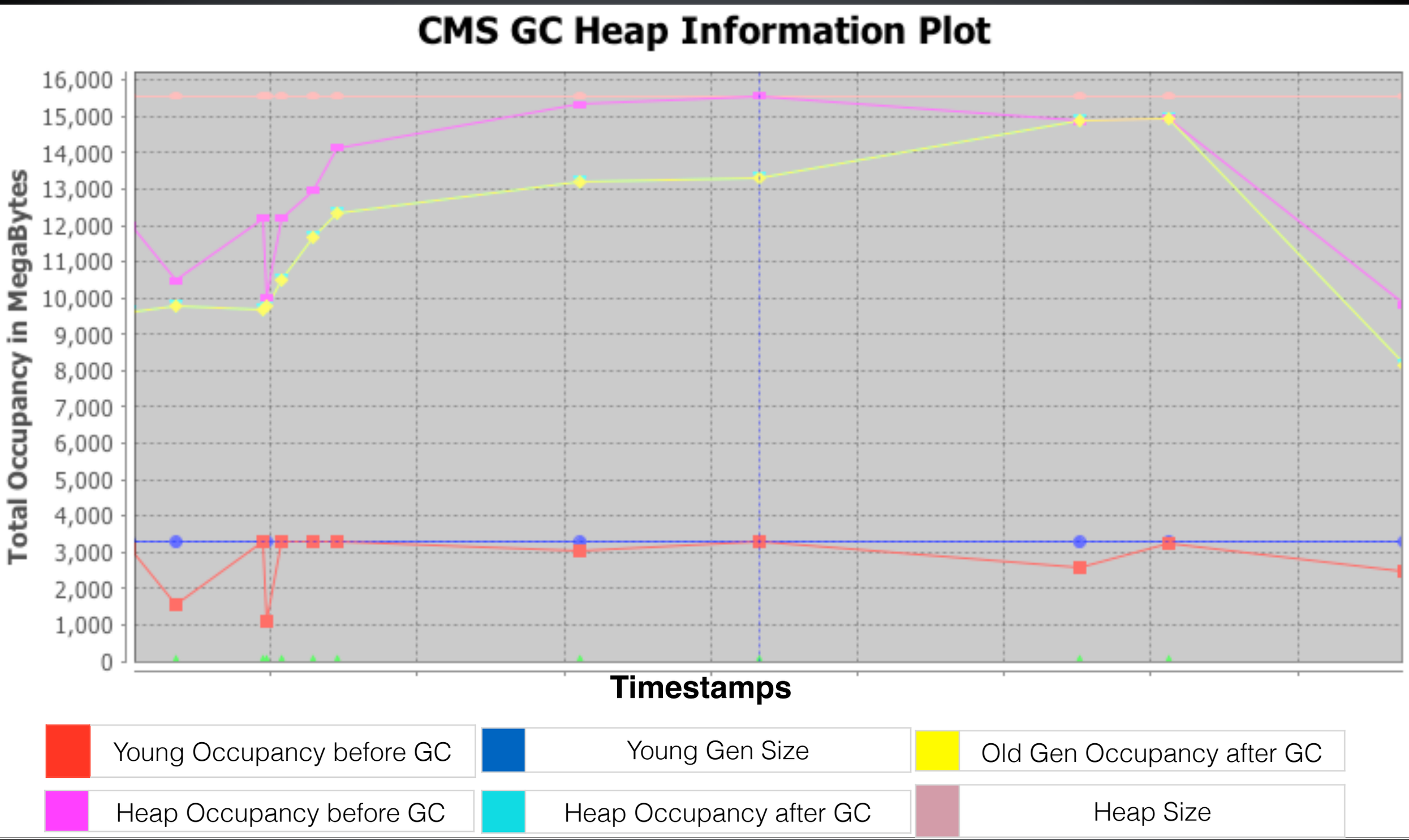
Allocation Rate and Promotion Rate

Tuning Recommendations

- Size generations keeping your application's object longevity and size in mind.
  - The faster the generation gets "filled"; the sooner a GC is triggered.
- Size your generations and age your objects appropriately.
  - The higher the amount of live data to be copied, the longer the GC pause.
- **Premature promotions are a big problem!**

Plot Allocation & Promotion Rates





Questions?

hotspot-gc-use@openjdk.java.net

hotspot-gc-dev@openjdk.java.net

monica@codekaram.com

www.codekaram.com