GC Tuning Confessions Of A Performance Engineer

Monica Beckwith
monica@codekaram.com
@mon_beck
www.linkedin.com/in/monicabeckwith

JavaOne Conference
Oct. 26th, 2015

About Me

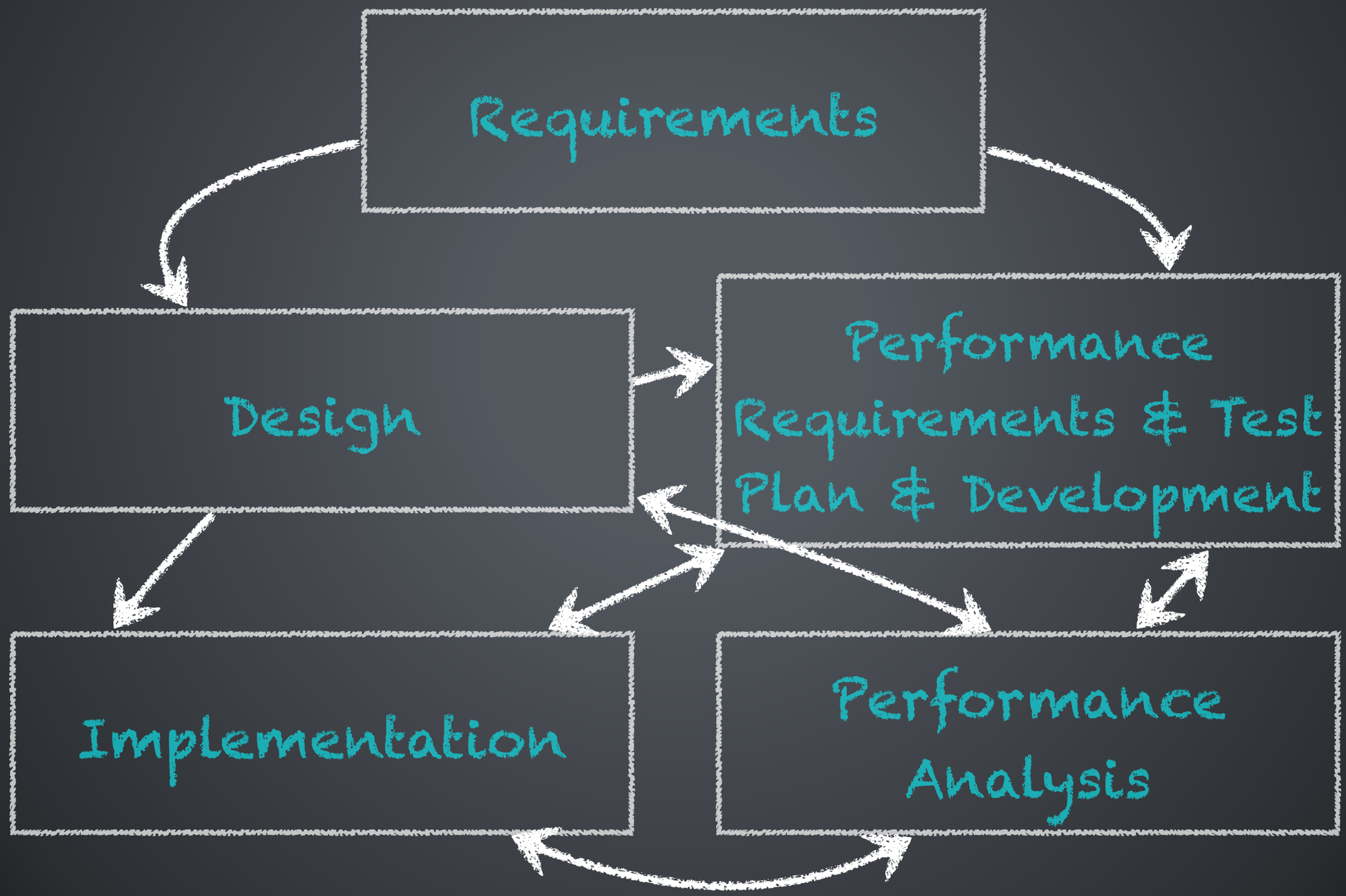
- JVM/GC Performance Engineer/Consultant
- Worked at AMD, Sun, Oracle…
- Worked with HotSpot JVM
  - JVM heuristics, JIT compiler, GCs: Parallel(Old) GC, G1 GC, CMS GC

About Today's Talk

- A little bit about Performance Engineering
- Insight into Garbage Collectors
- OpenJDK HotSpot GCs
    - The Tradeoffs
    - GC Algorithms
    - Key Topics
- Summary
    - GC Tuneables

Performance Engineering

Performance Engineering



Performance Requirements

- Service level agreements (SLAs) for:
- throughput,
- latency and other response time related metrics E.g. Response time (RT) metrics - Average (RT), max or worst-case RT, 99th percentile RT…

Response Time Metrics

	Number of GC events	Minimum (ms)
System1	37353	7.622
System2	34920	7.258
System3	36270	6.432
System4	40636	7.353

Response Time Metrics

	Number of GC events	Minimum (ms)	Average (ms)
System1	37353	7.622	307.741
System2	34920	7.258	320.778
System3	36270	6.432	321.483
System4	40636	7.353	323.143

Response Time Metrics

	Number of GC events	Minimum (ms)	Average (ms)	99th Percentile (ms)
System1	37353	7.622	307.741	940.901
System2	34920	7.258	320.778	1006.607
System3	36270	6.432	321.483	1004.018
System4	40636	7.353	323.143	1041.225

Response Time Metrics

	Number of GC events	Minimum (ms)	Average (ms)	99th Percentile (ms)	Maximum (ms)
System 1	37353	7.622	307.741	940.901	3131.331
System 2	34920	7.258	320.778	1006.607	2744.588
System 3	36270	6.432	321.483	1004.018	1681.308
System 4	40636	7.353	323.143	1041.225	**20699.505**

Response Time Metrics

	Average (ms)	99th Percentile (ms)	Maximum (ms)
System1	307.741	940.901	3131.331
System2	320.778	1006.607	2744.588
System3	321.483	1004.018	1681.308
System4	323.143	1041.225	**20699.505**

5 full GCs and 10 evacuation failures

Insight into GCs

Fun Facts!

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

Ideal GC?

Maximize Throughput ✓

Ideal GC?

Maximize Throughput ✓

Minimize Latency ✓

Ideal GC?

Maximize Throughput ✓

Minimize Latency ✓

Minimize Footprint ✓

The Reality!

Pick Any Two! :(

OpenJDK HotSpot

GCs: The TRADE/OFF

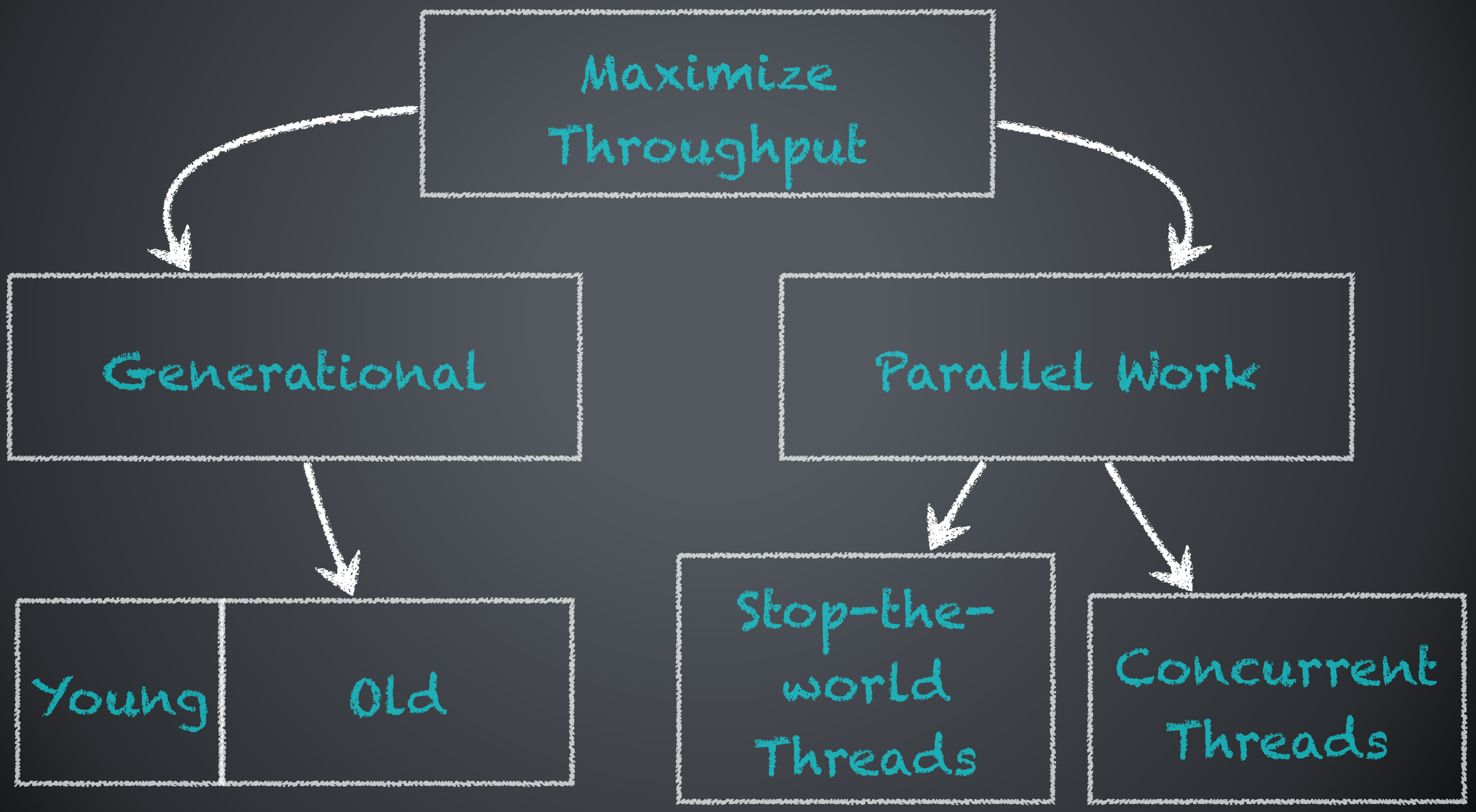
Fun Fact!

Most OpenJDK HotSpot users would like to increase their (Java) heap space but they fear full garbage collections.

The Tradeoff -

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

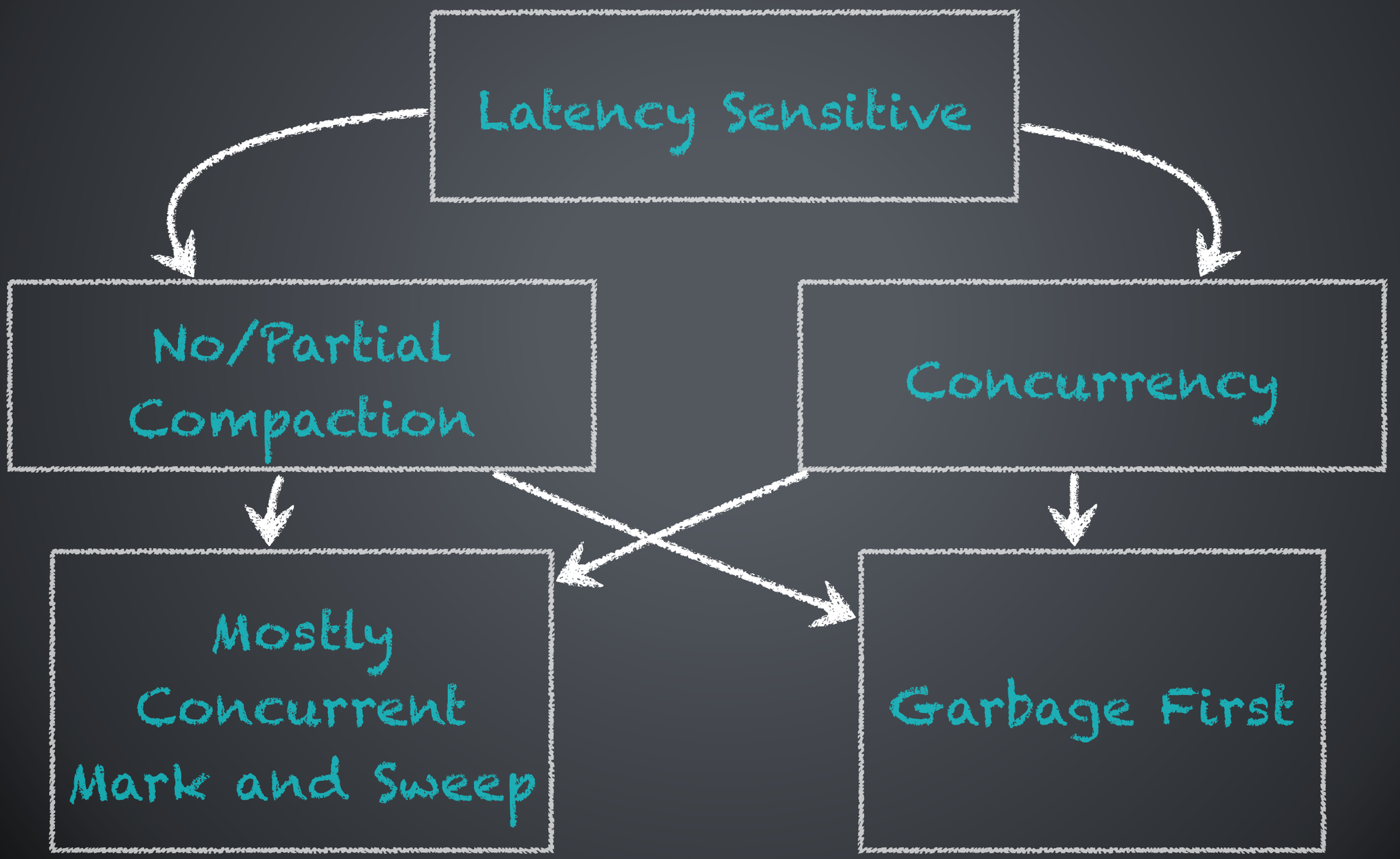
The Throughput Maximizer



Fun Fact!

All GCs in OpenJDK HotSpot are generational.

Mr. Latency Sensitive



Fun Fact!

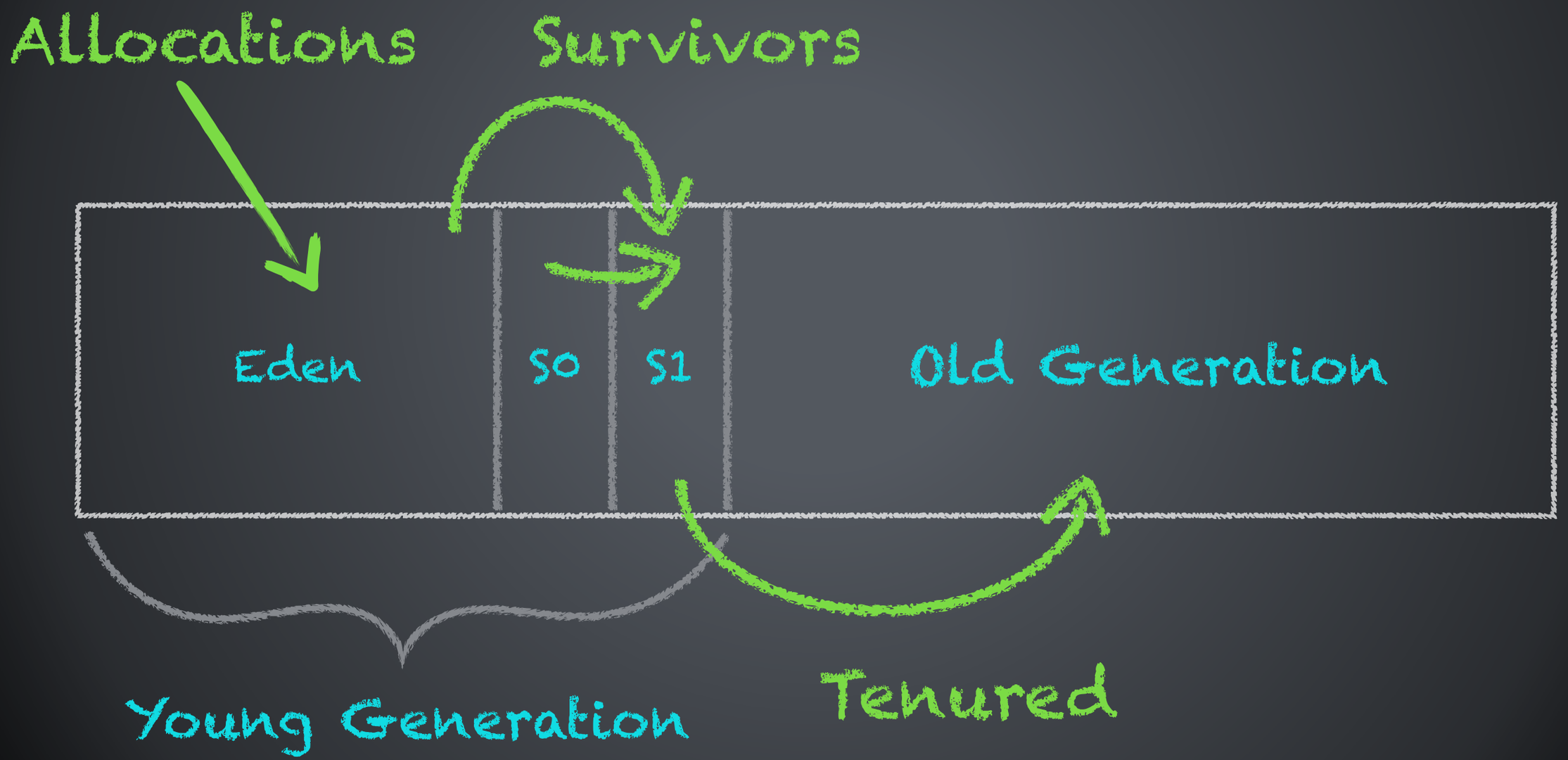
All GCs in OpenJDK HotSpot fallback to a fully compacting stop-the-world garbage collection called the "full" GC.

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

OpenJDK HotSpot

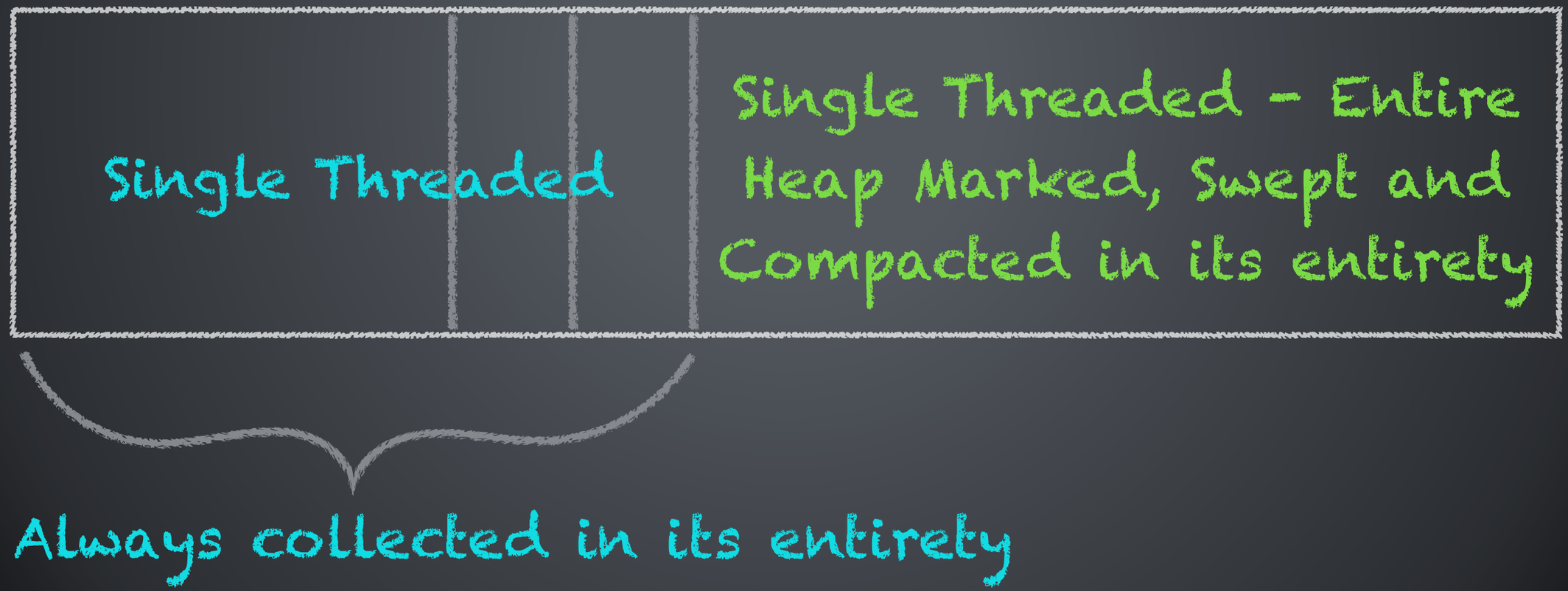
GCs: Algorithms

The Generational Java Heap

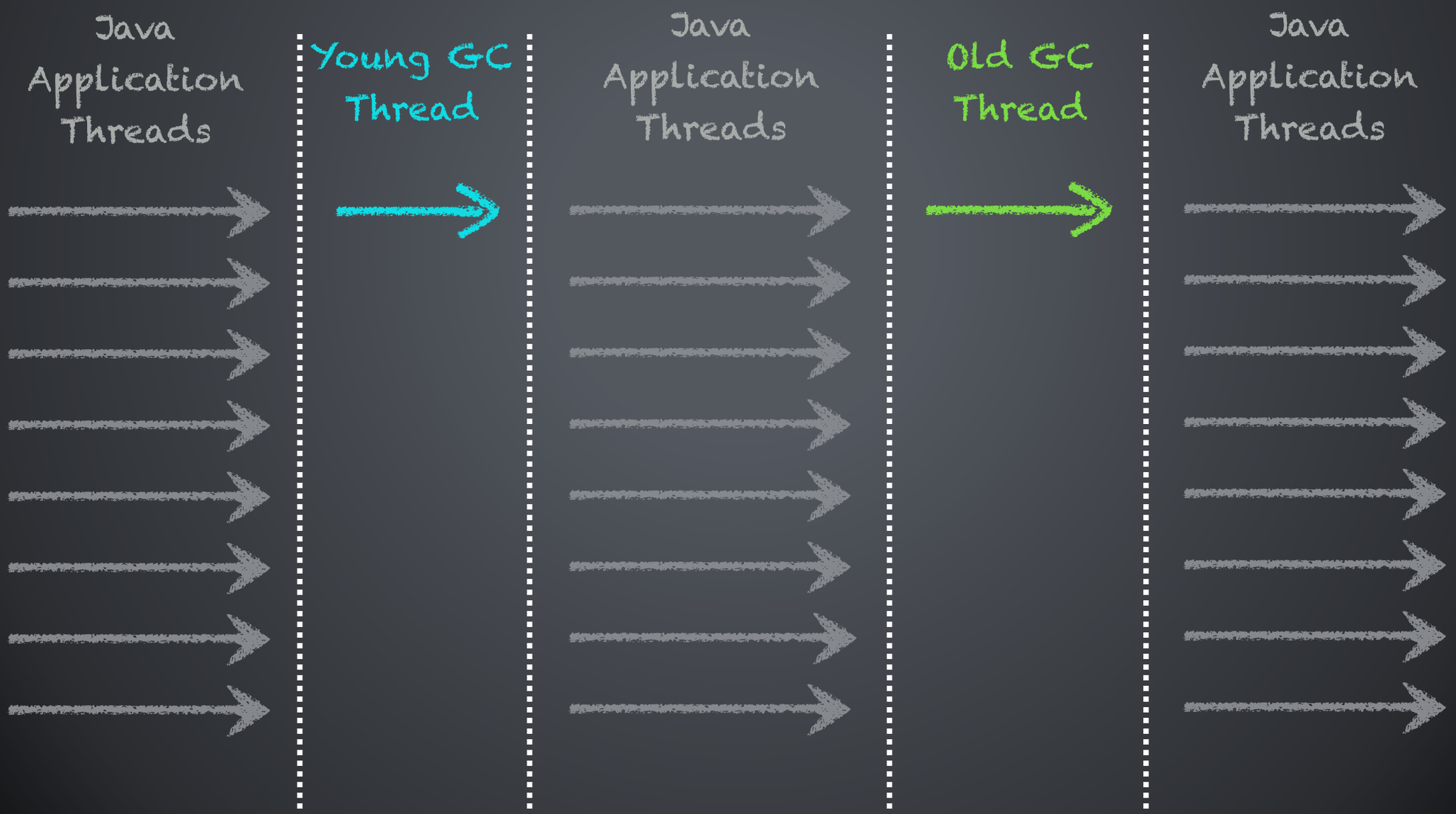


OpenJDK HotSpot Collectors

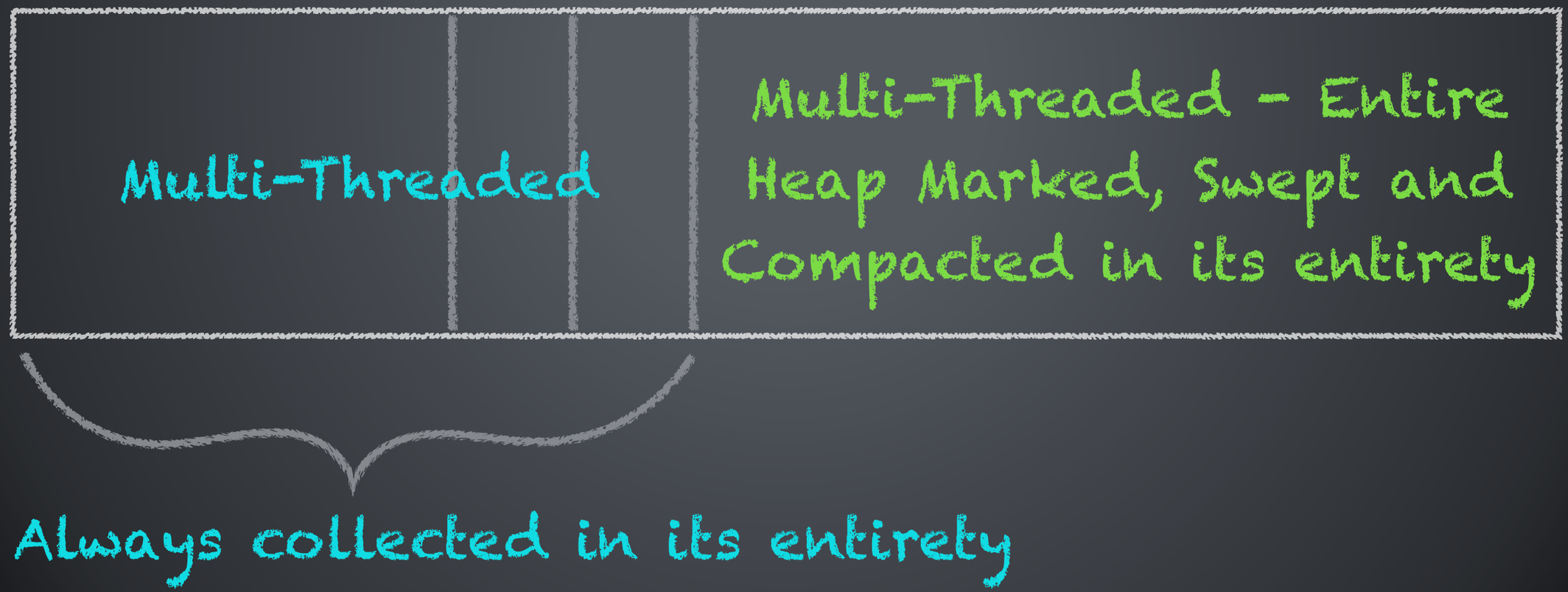
The Serial Collector



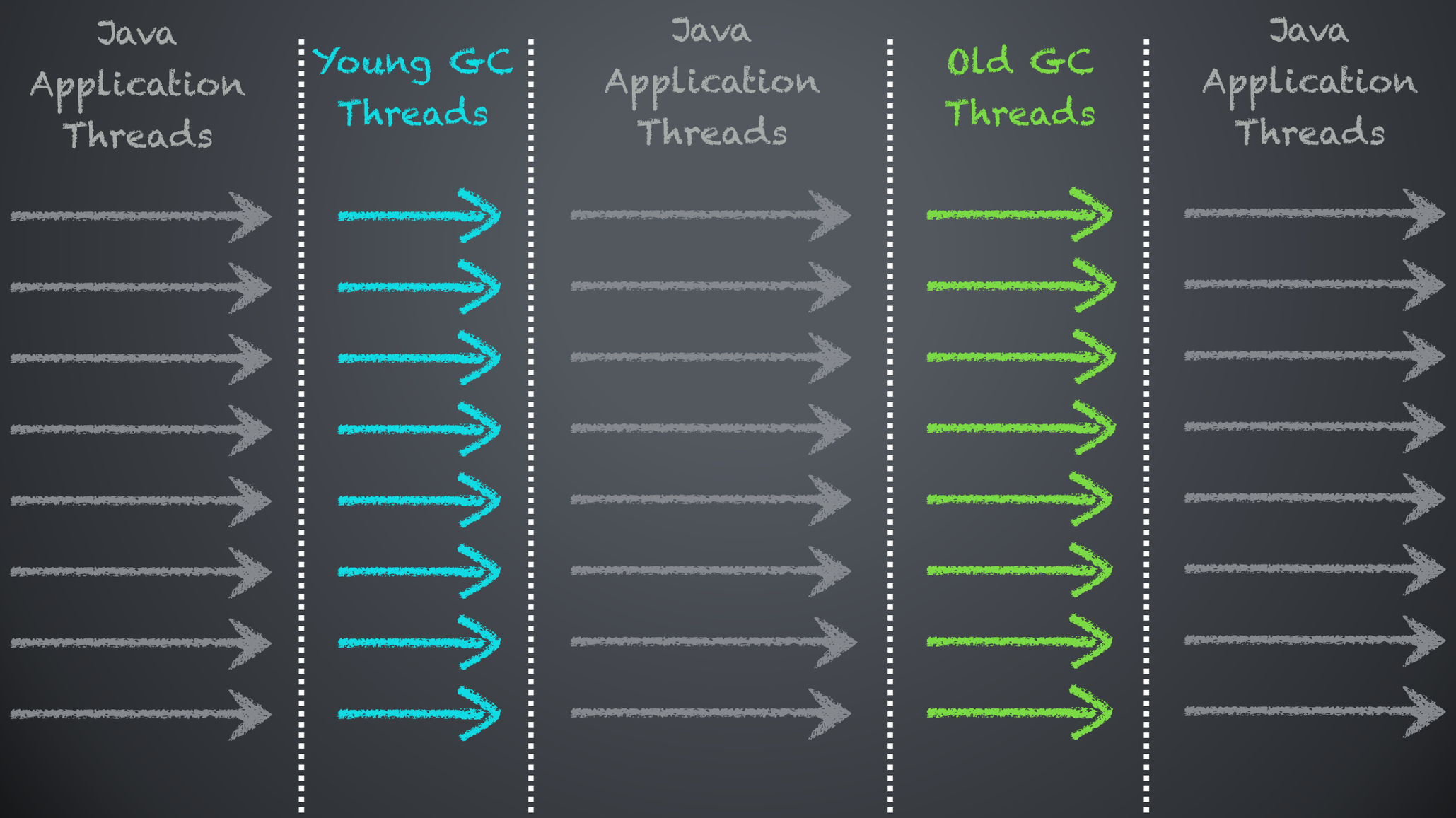
The Serial Collector



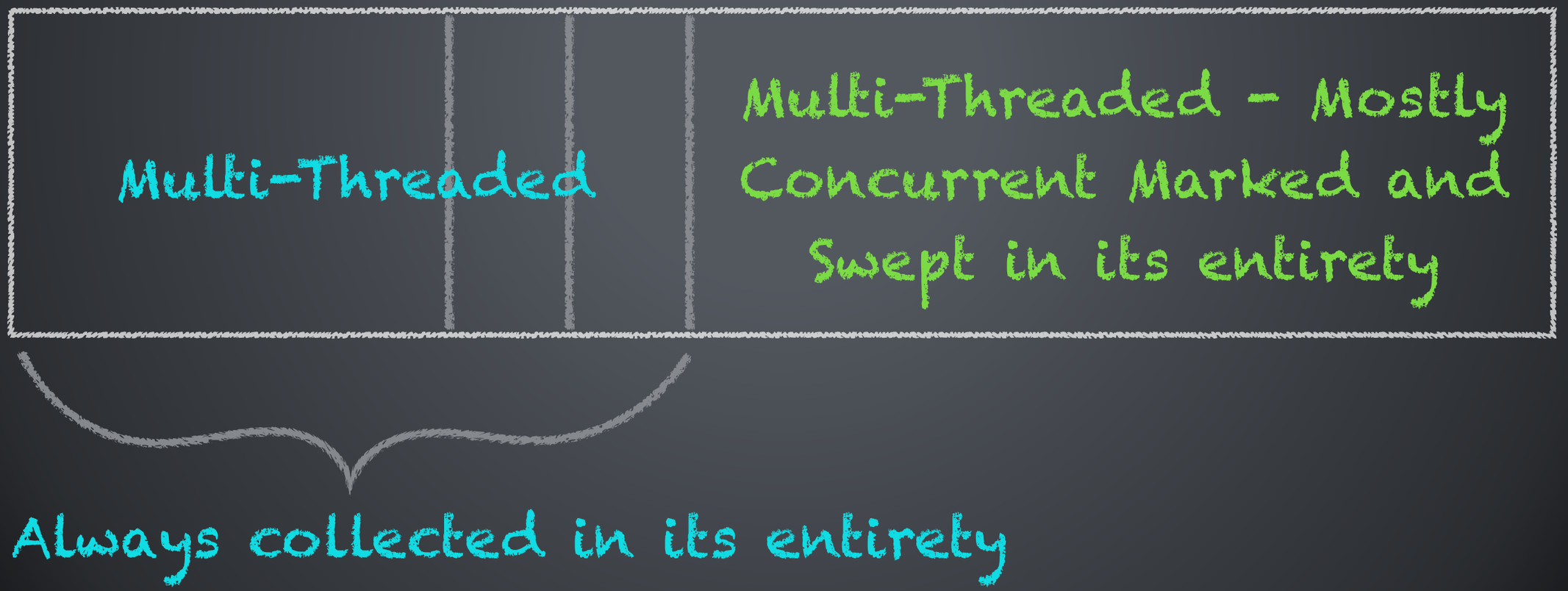
The Throughput Collector



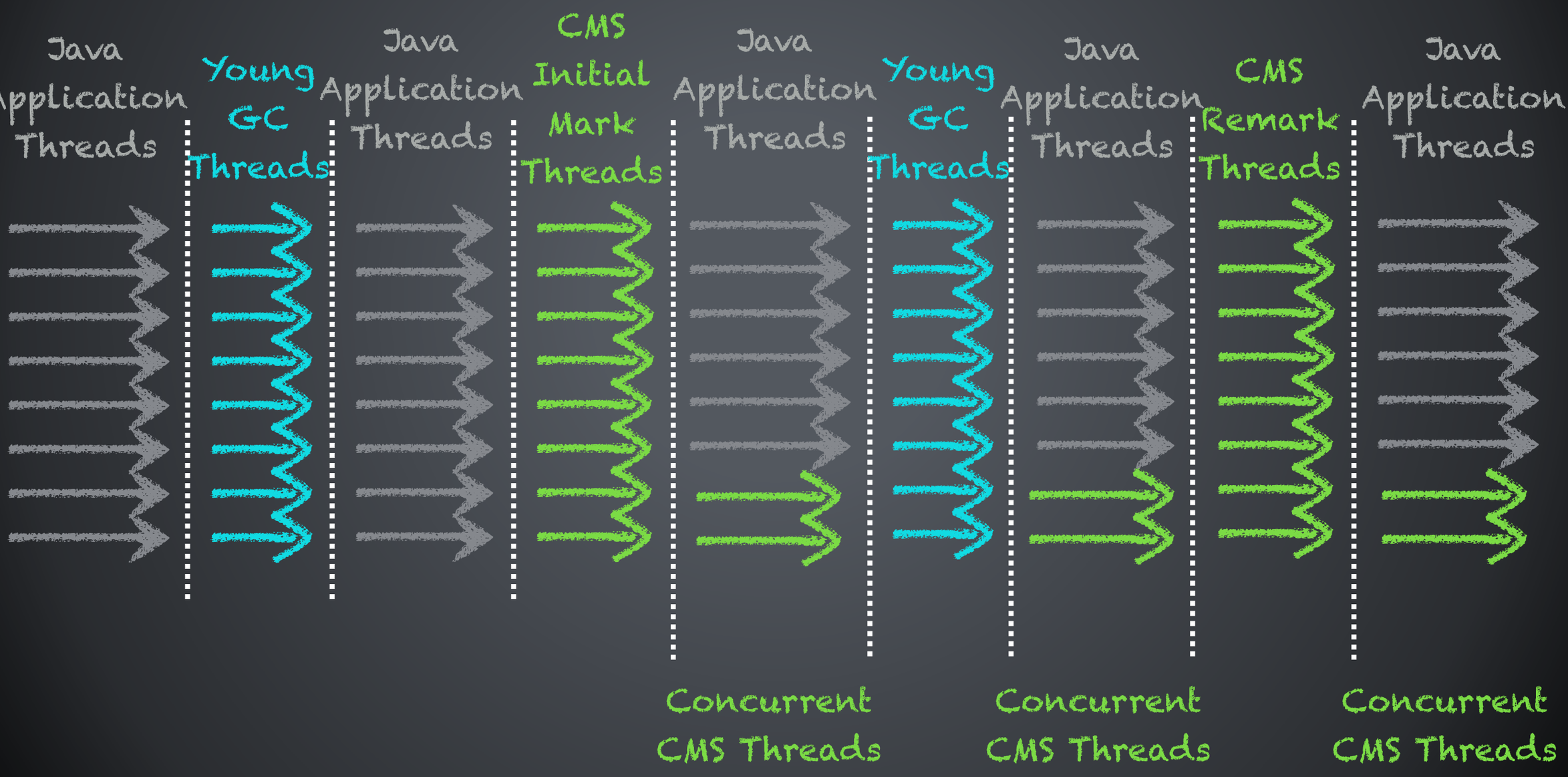
The Throughput Collector



The CMS Collector



The CMS Collector



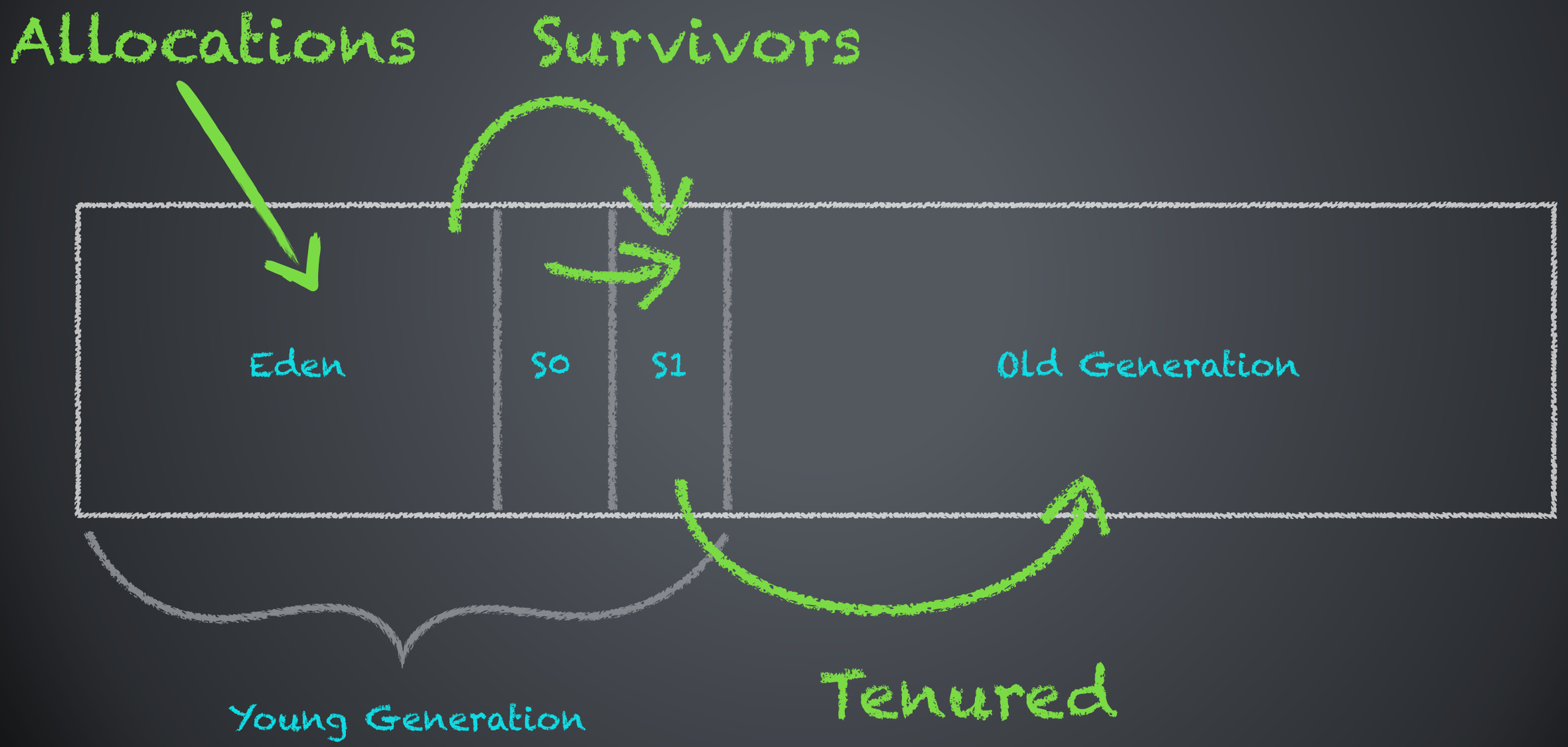
GC Algorithms - Key Topics

What Triggers Full (Fail-Safe) Collections?

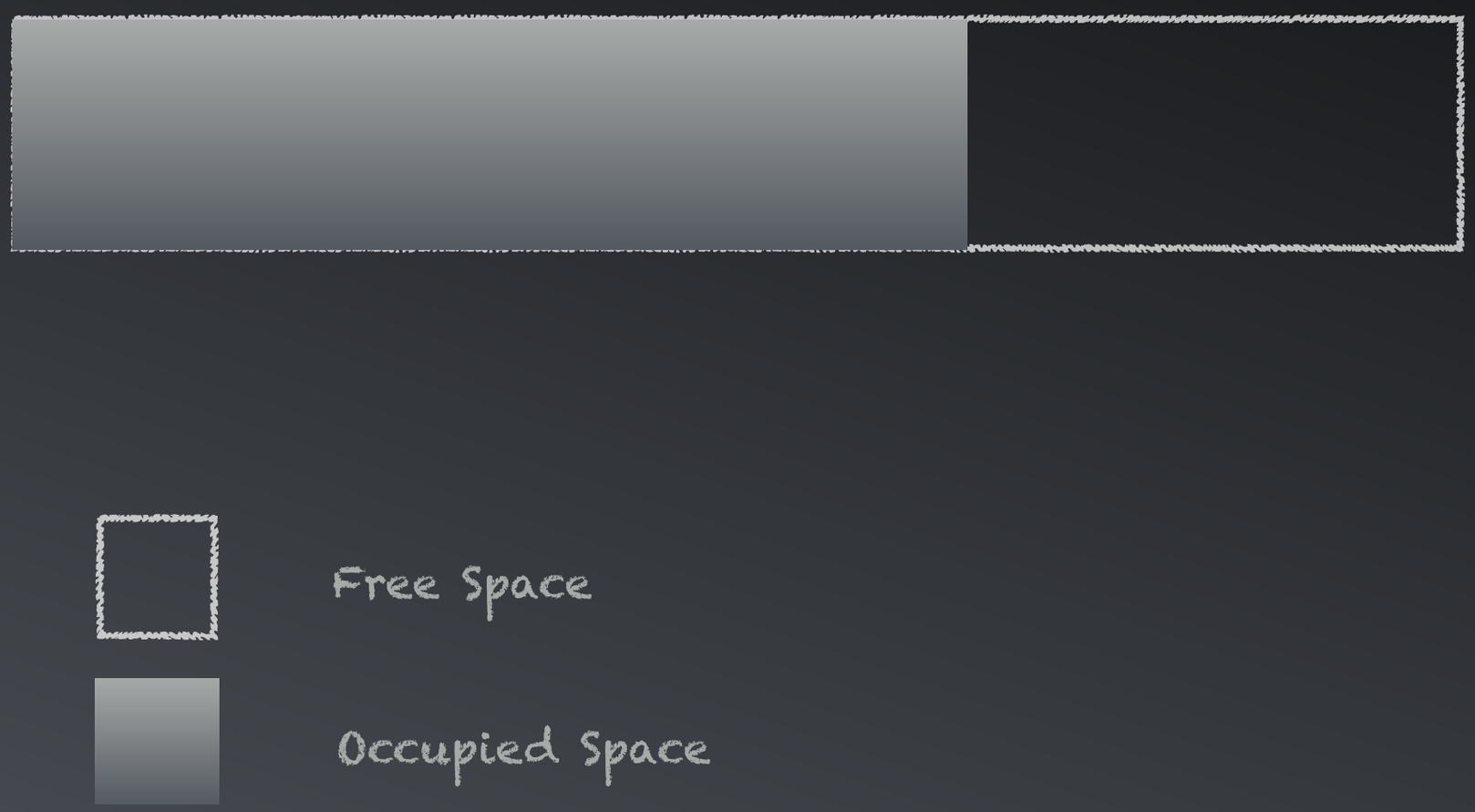
Promotion Failures!

Promotion Failures In The Throughput Collector

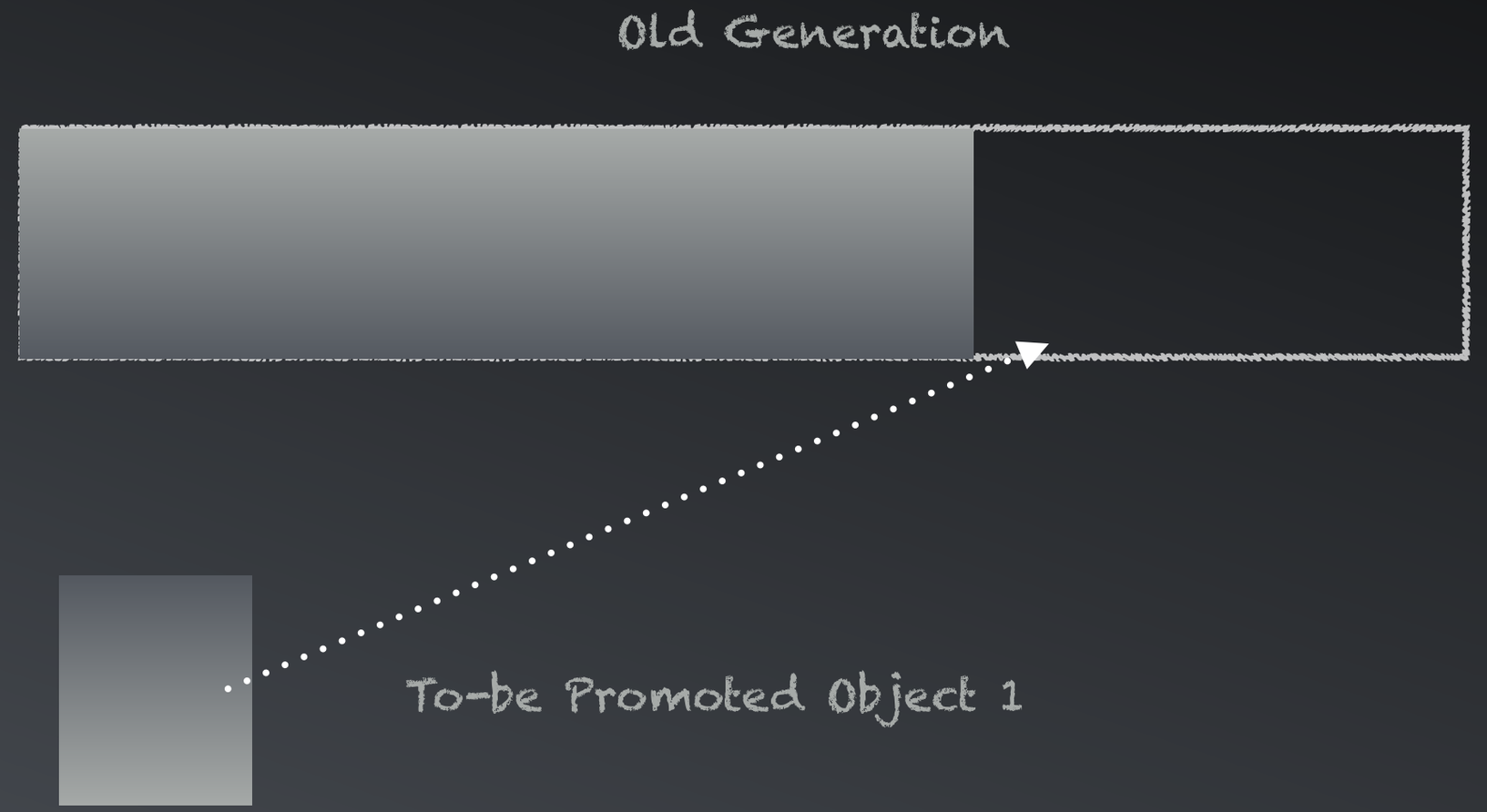
The Throughput Collector - Java Heap



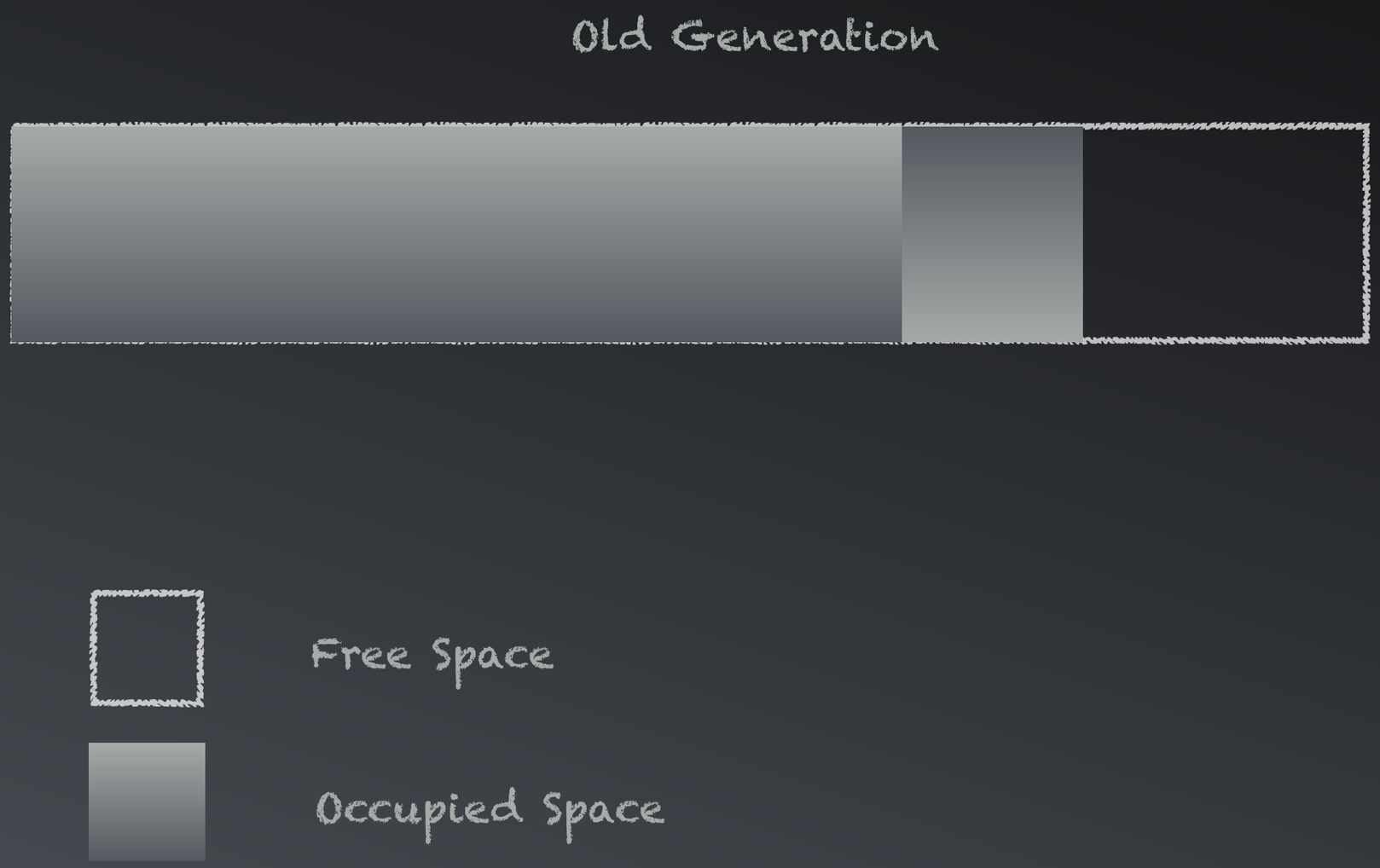
The Throughput Collector - Contiguous Old Generation



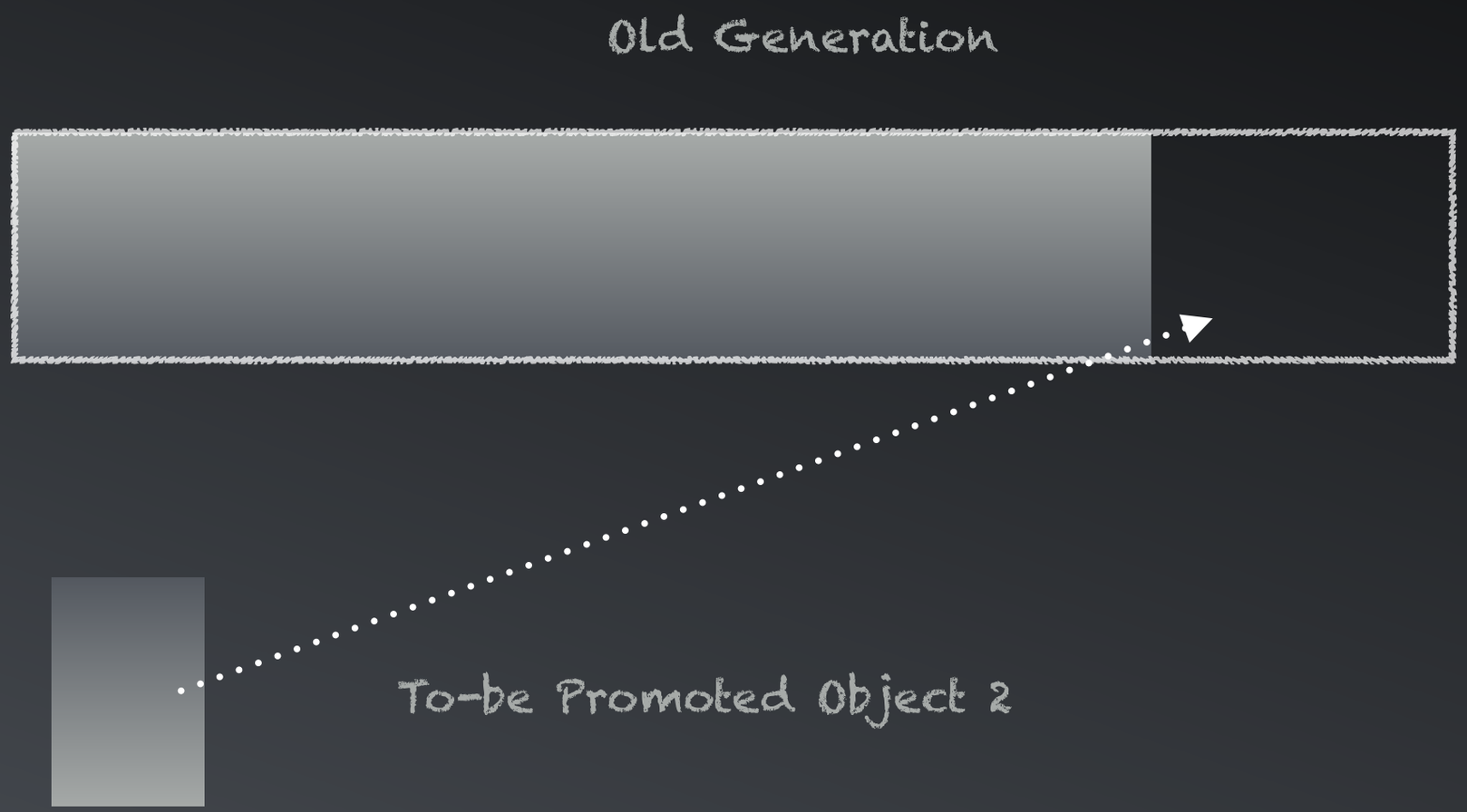
The Throughput Collector - Contiguous Old Generation



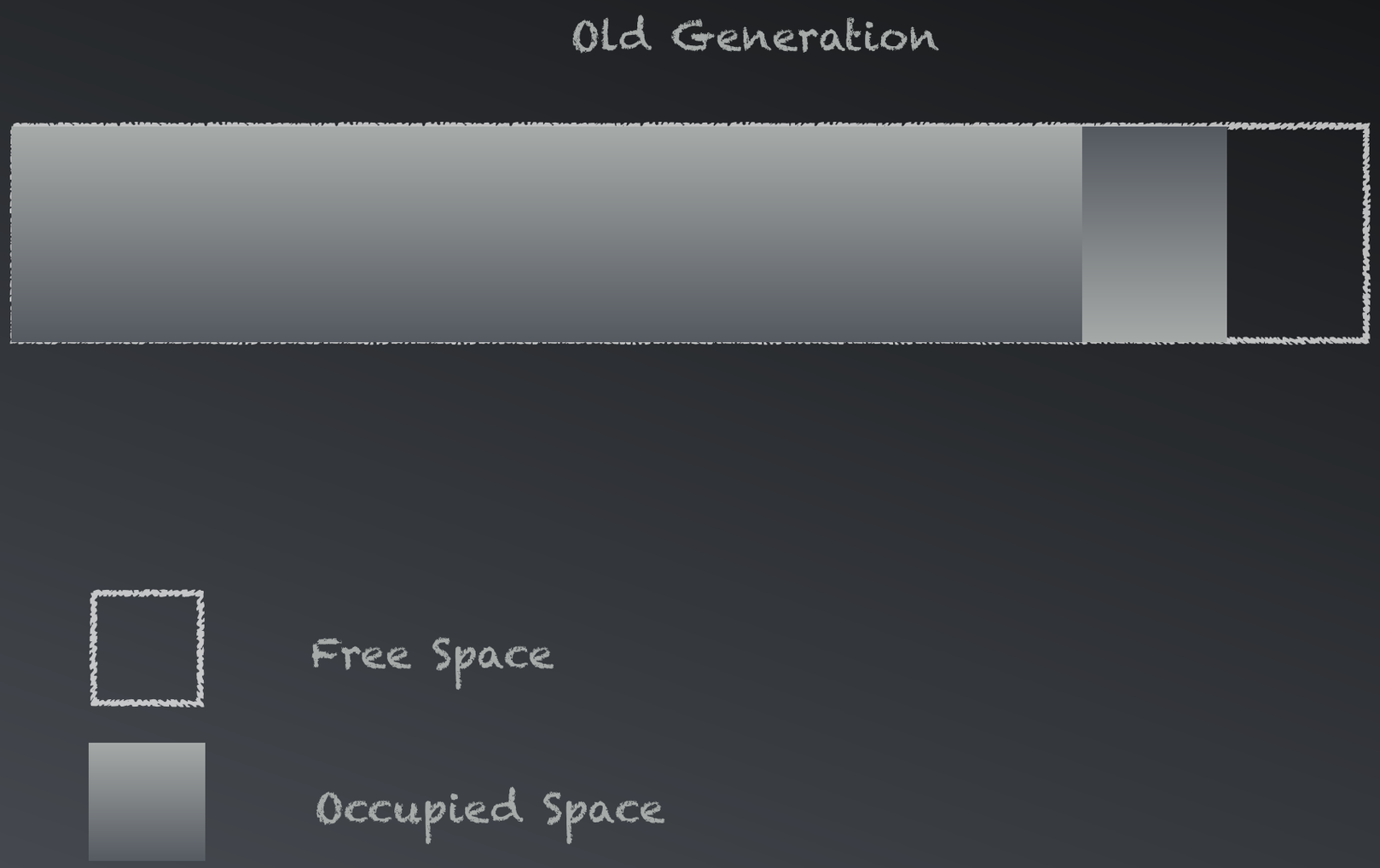
The Throughput Collector - Contiguous Old Generation



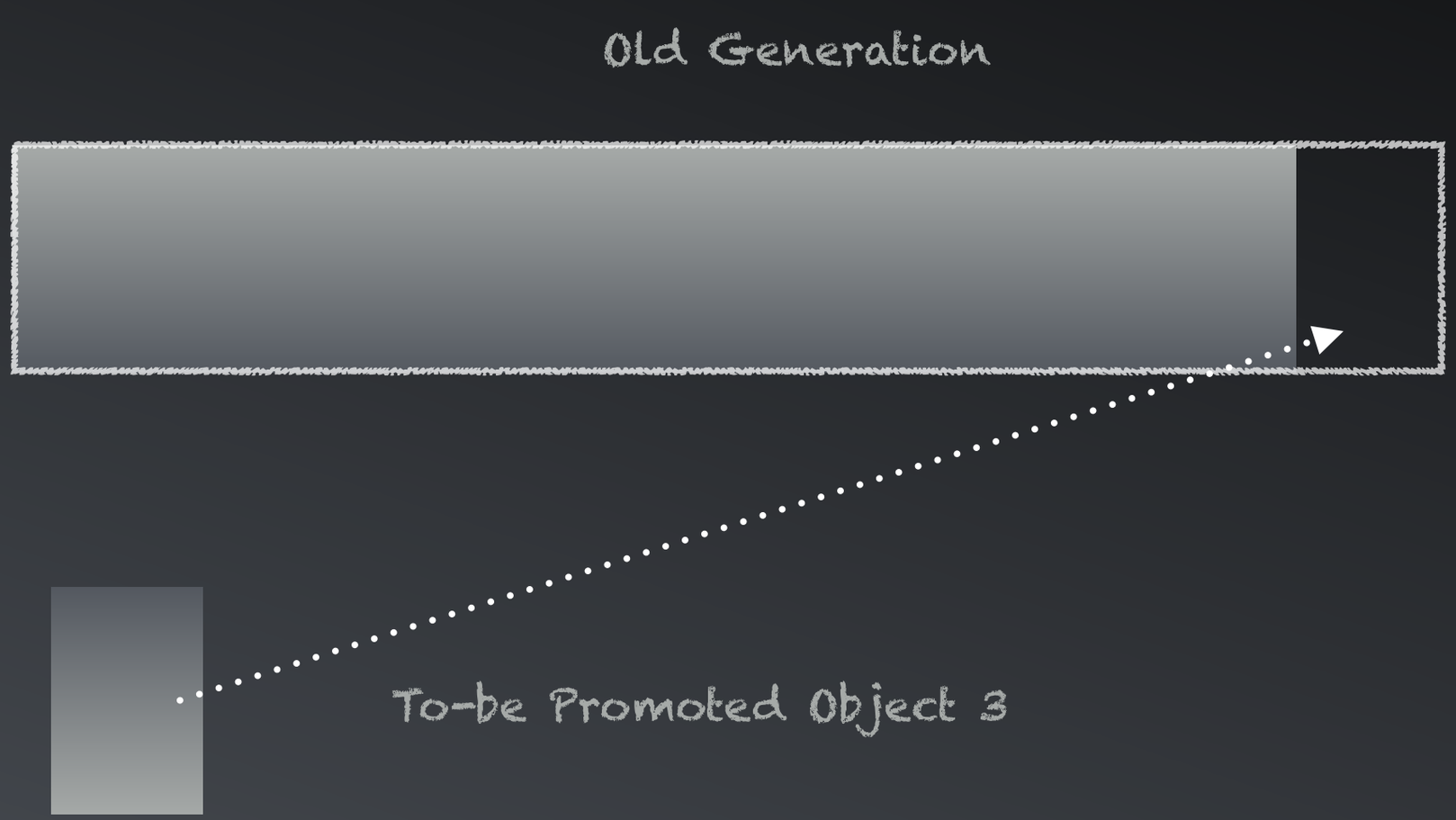
The Throughput Collector - Contiguous Old Generation



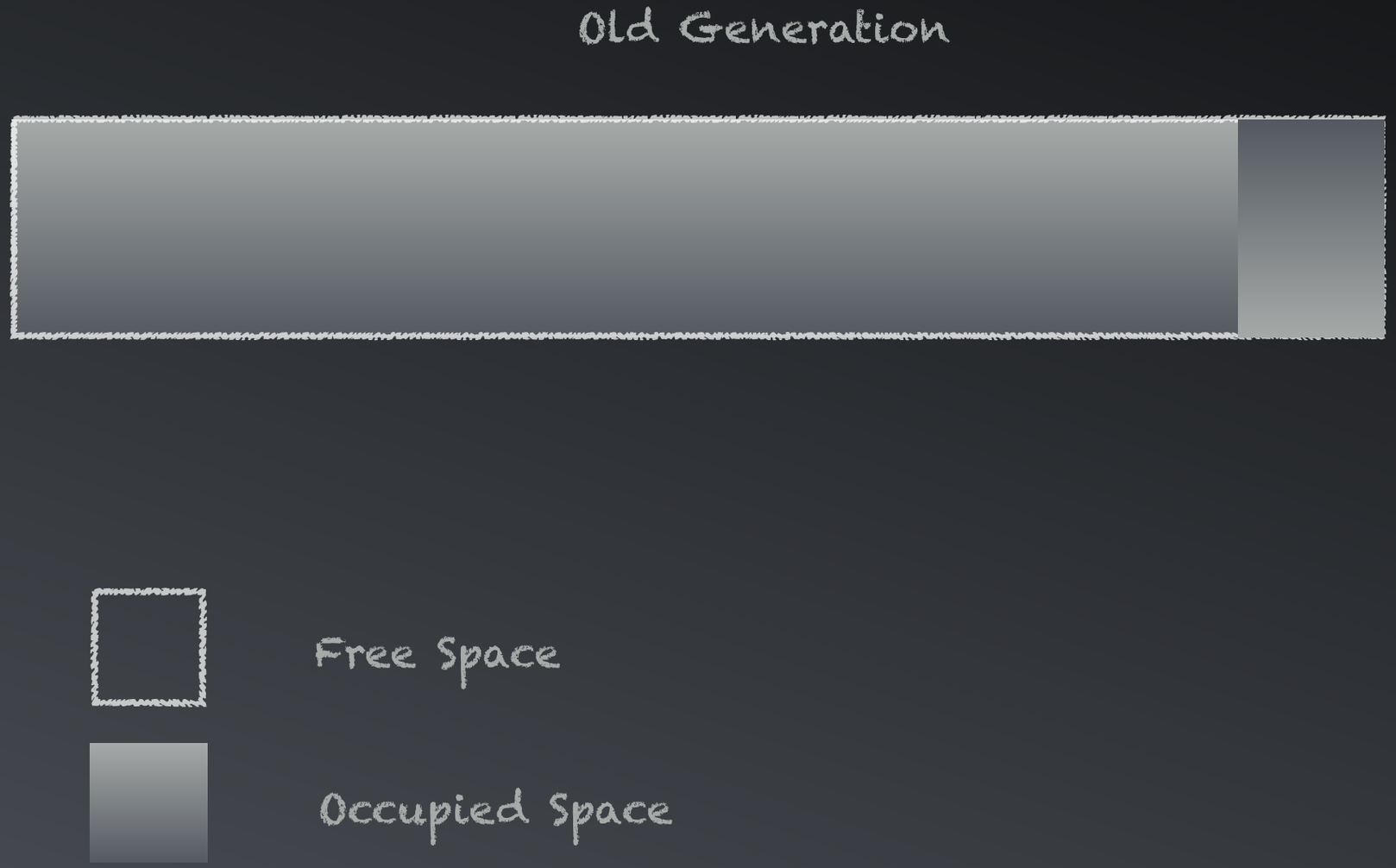
The Throughput Collector - Contiguous Old Generation



The Throughput Collector - Contiguous Old Generation

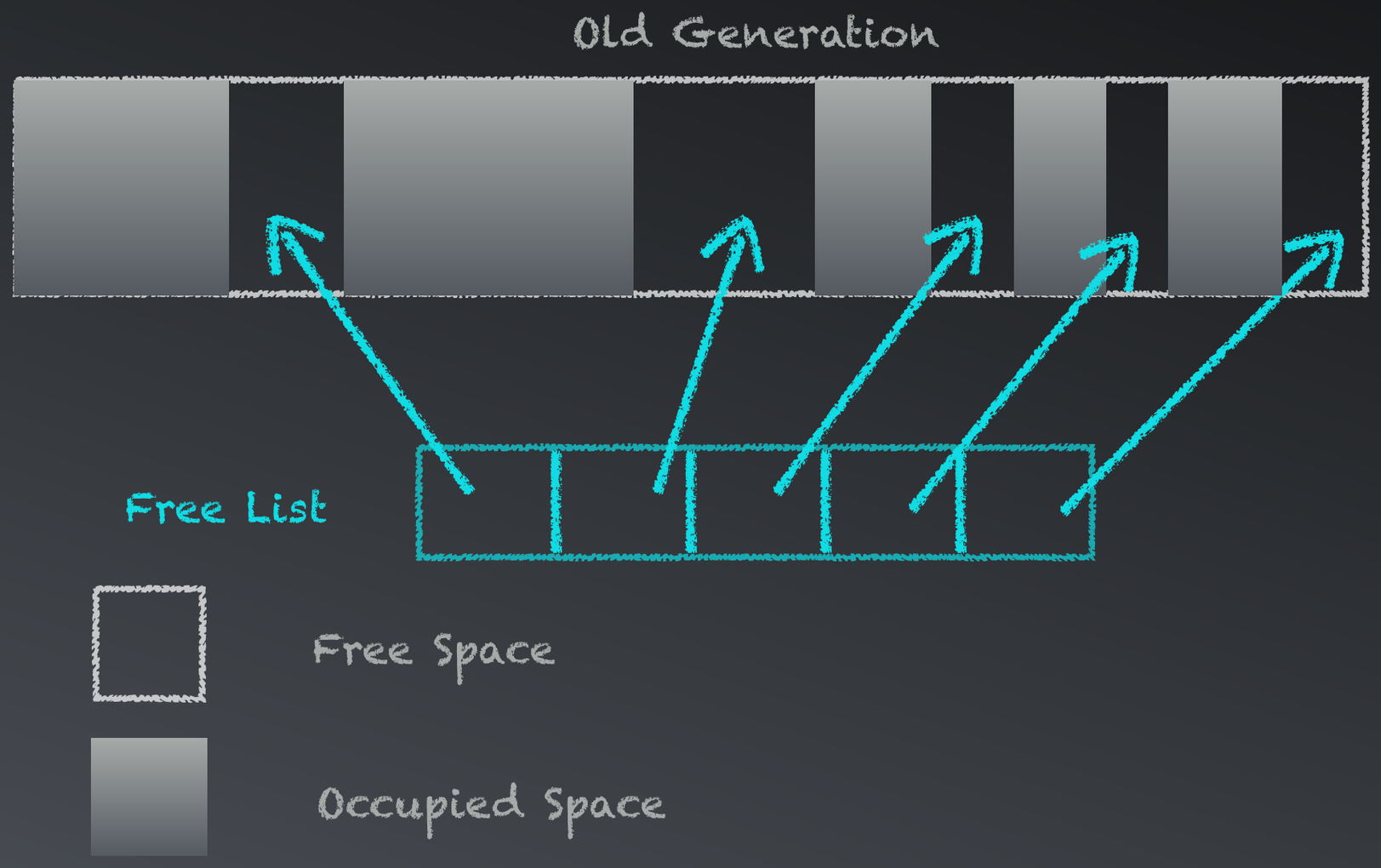


The Throughput Collector - Contiguous Old Generation

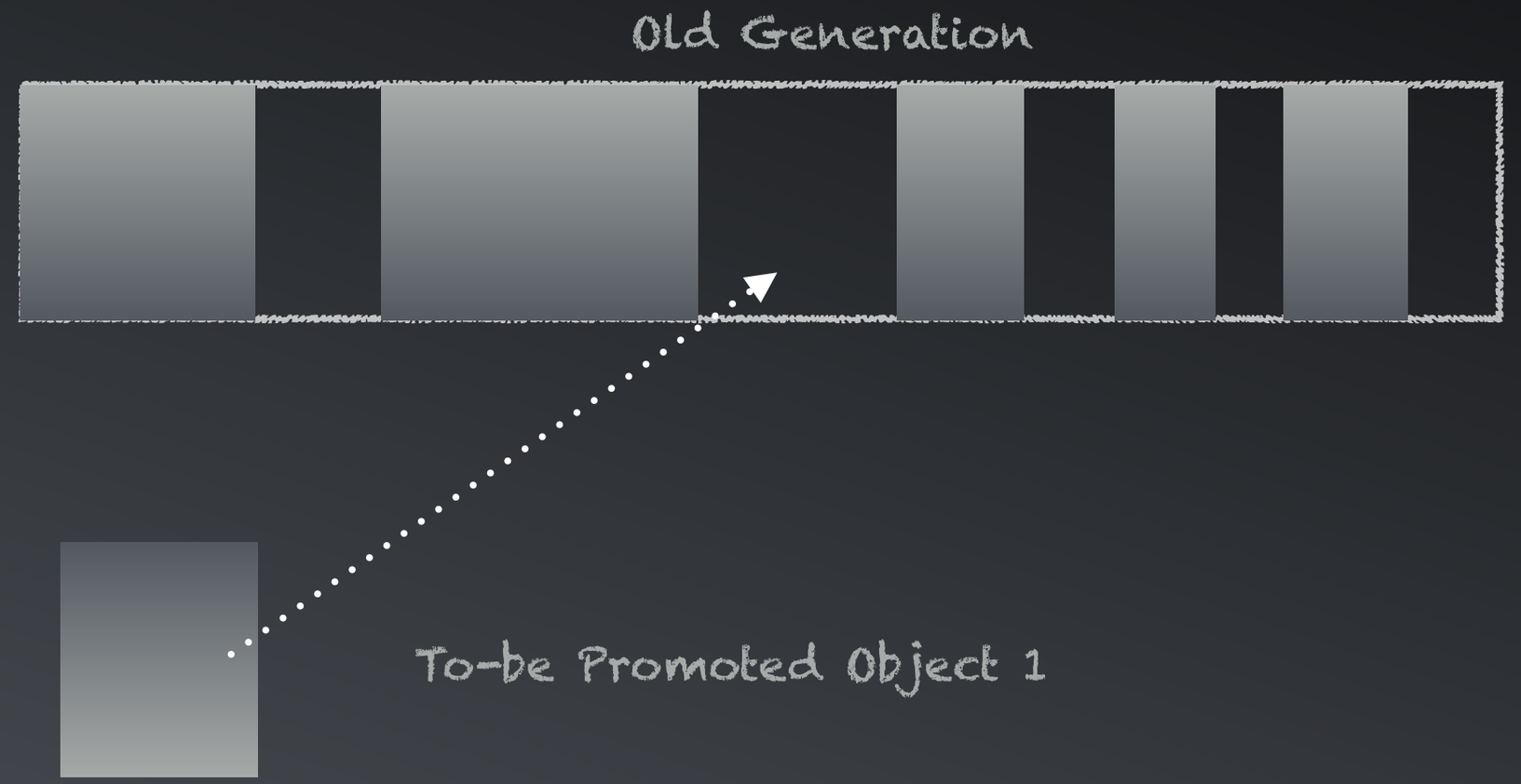


Promotion Failures & Concurrent Mode Failures In The CMS Collector

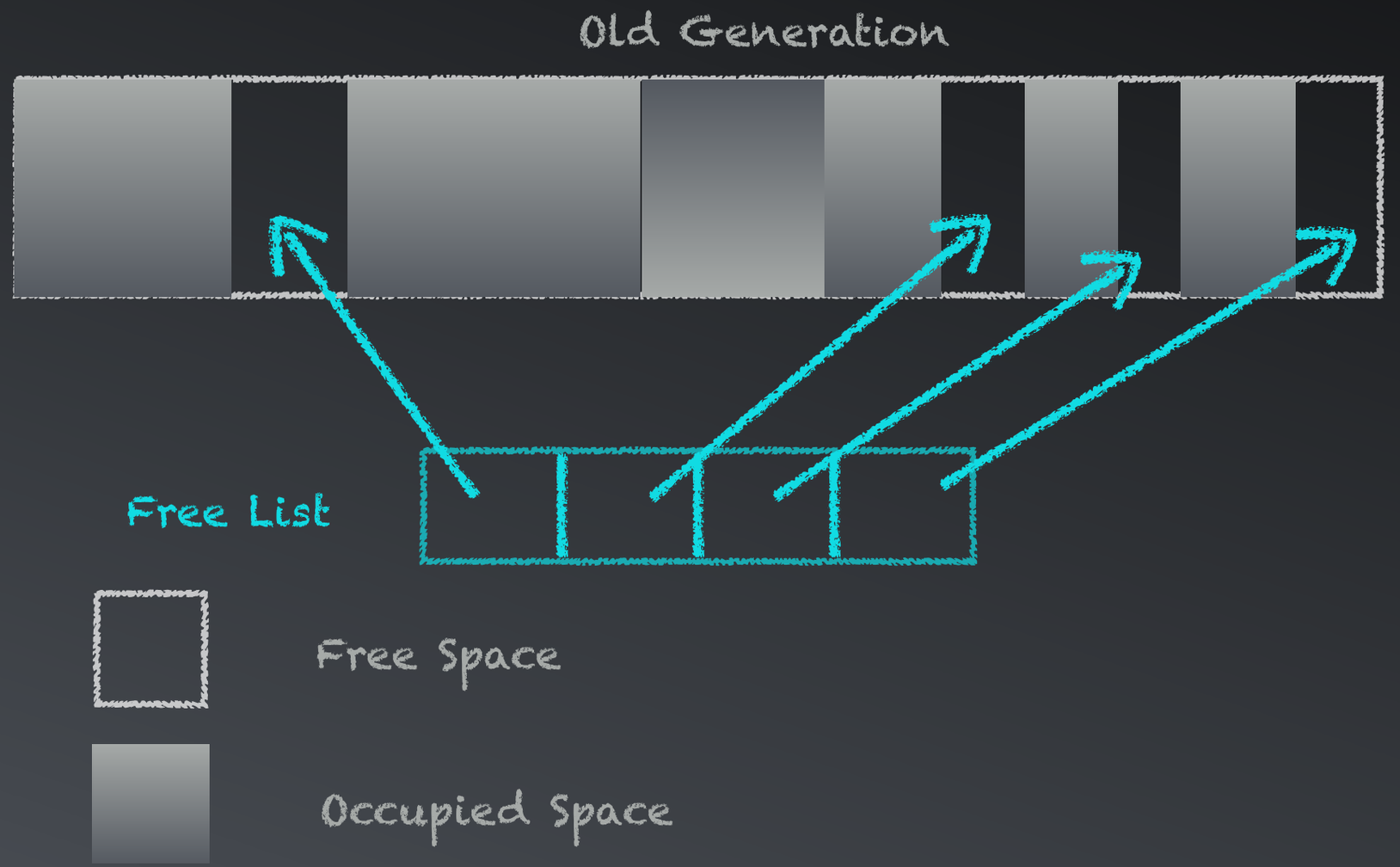
The CMS Collector - Old Generation Maintained By Free Lists



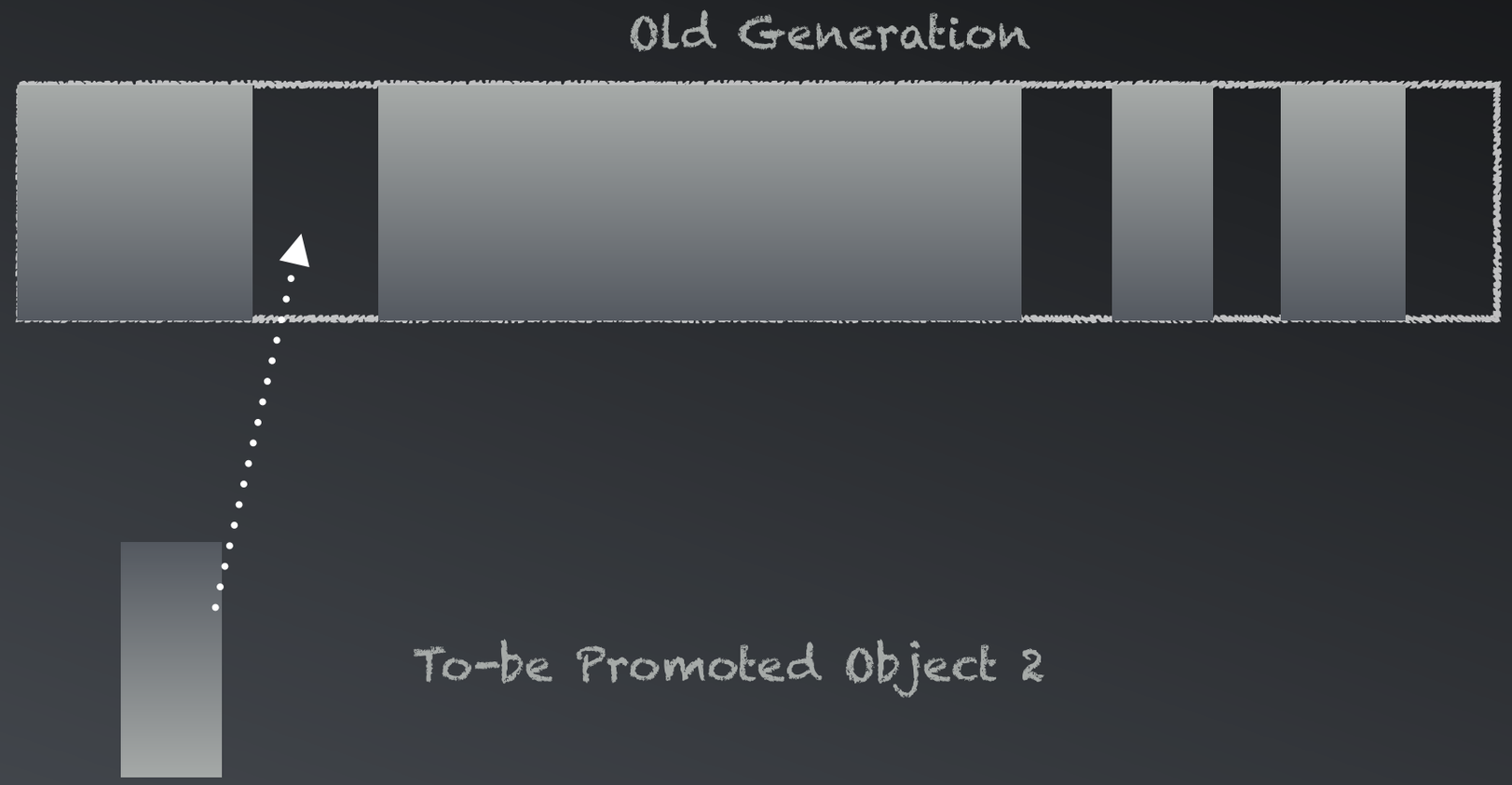
The CMS Collector - Old Generation Maintained By Free Lists



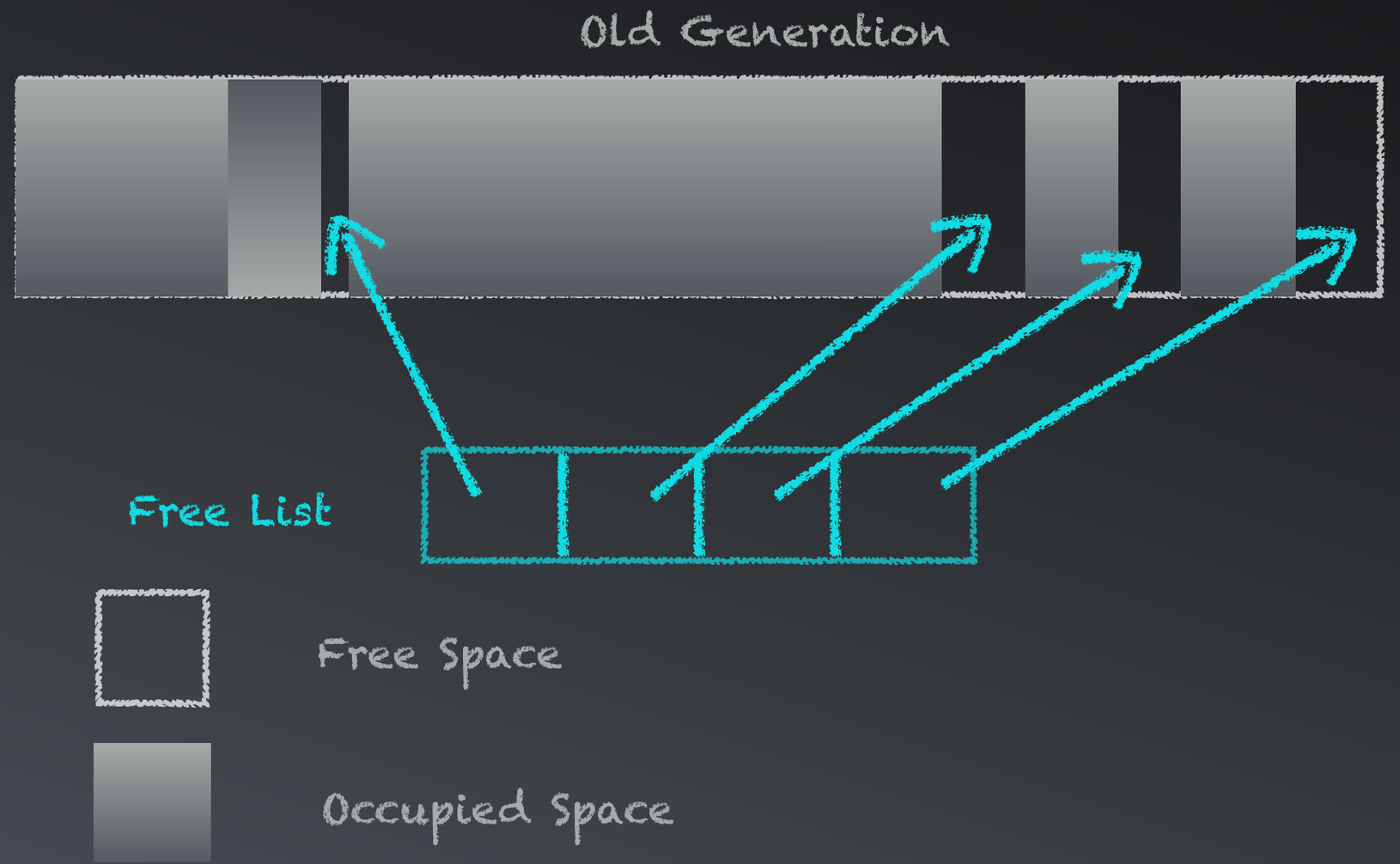
The CMS Collector - Old Generation Maintained By Free Lists



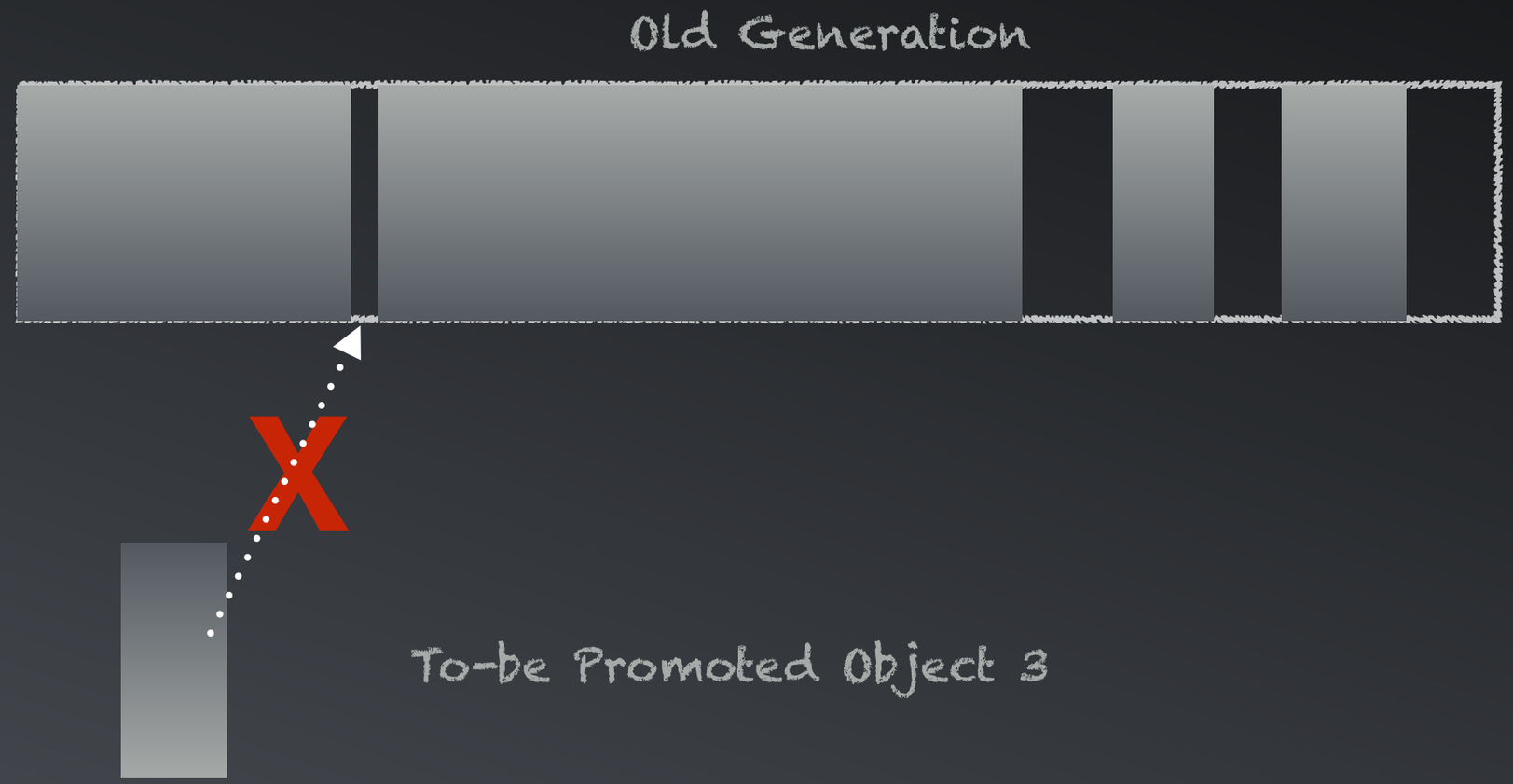
The CMS Collector - Old Generation Maintained By Free Lists



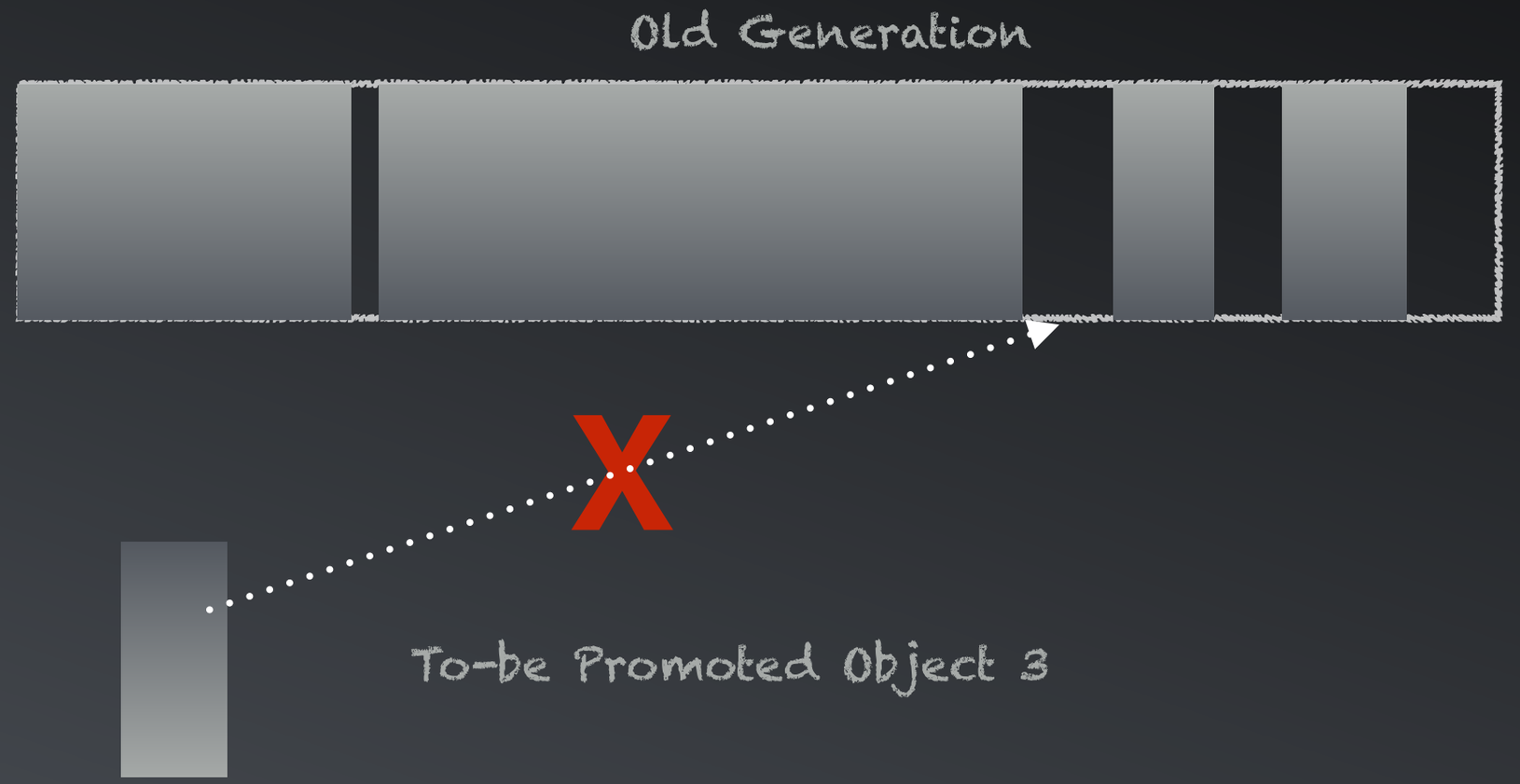
The CMS Collector - Old Generation Maintained By Free Lists



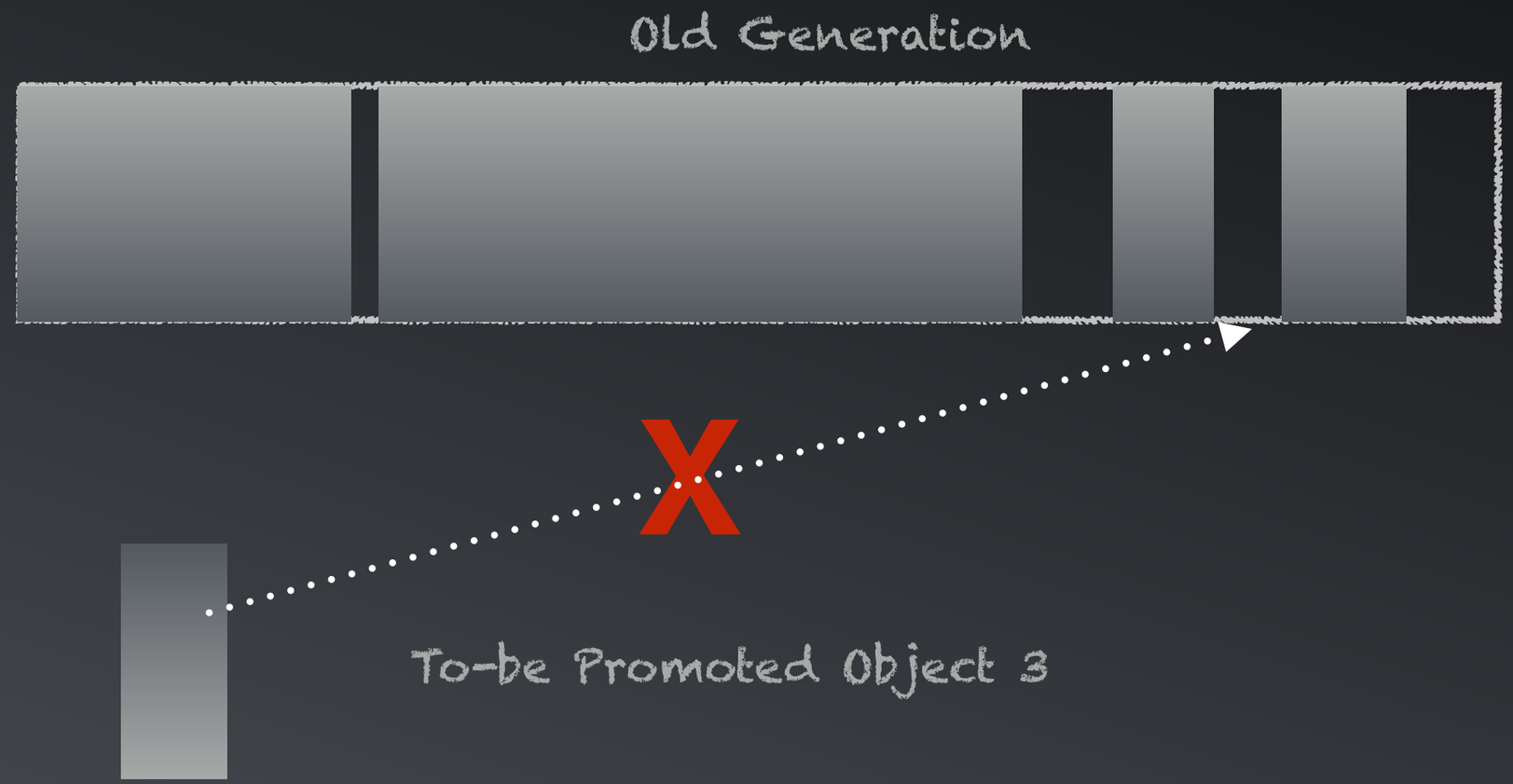
The CMS Collector - Old Generation Maintained By Free Lists



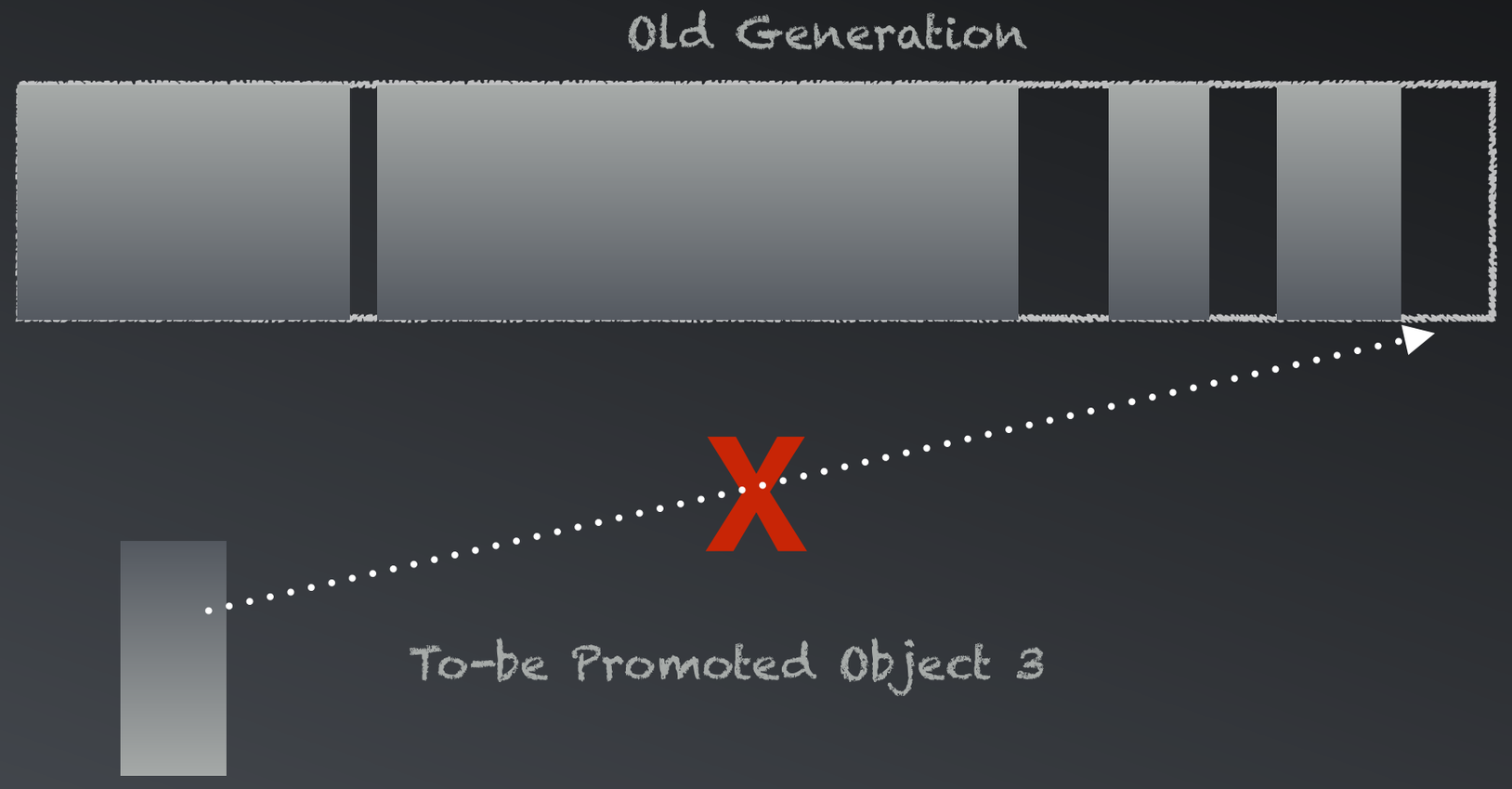
The CMS Collector - Old Generation Maintained By Free Lists



The CMS Collector - Old Generation Maintained By Free Lists

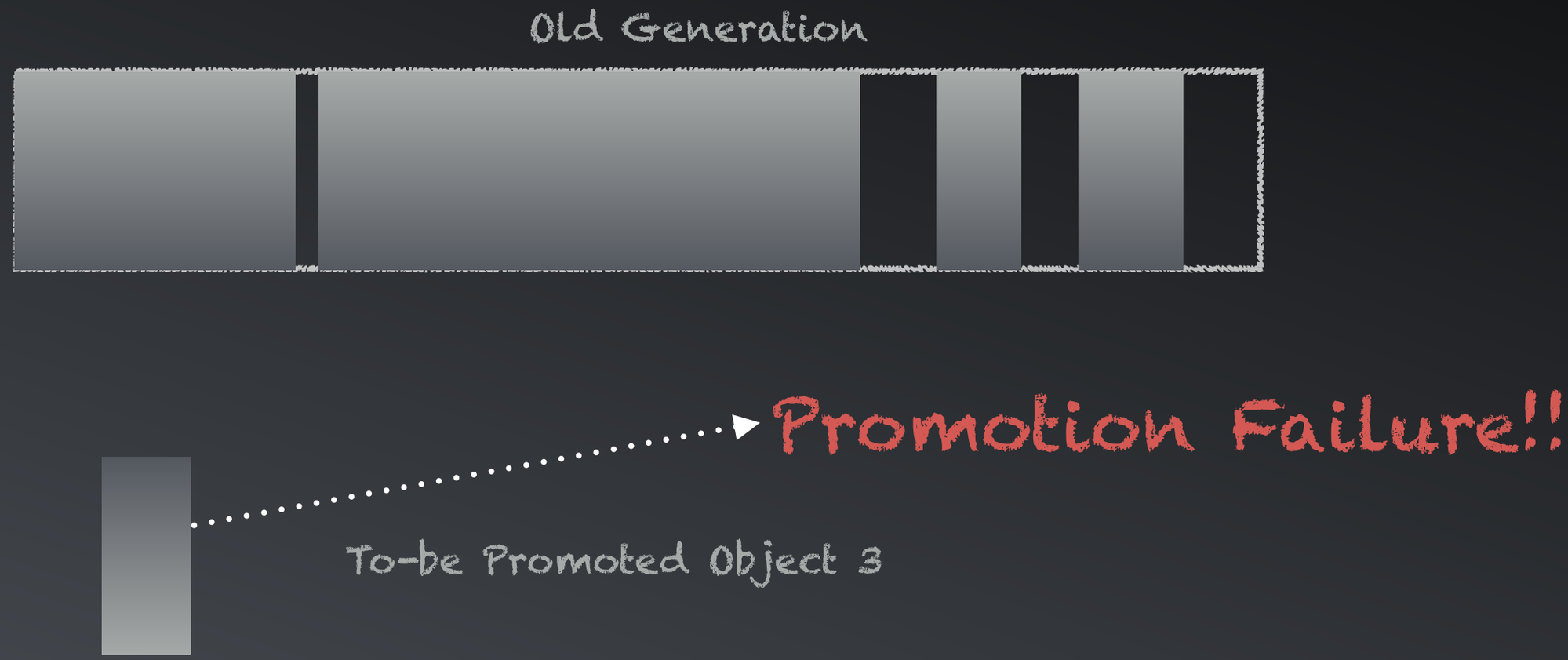


The CMS Collector - Old Generation Maintained By Free Lists



The CMS Collector - Old Generation Maintained By Free Lists

The CMS Collector - Fragmented Old Generation

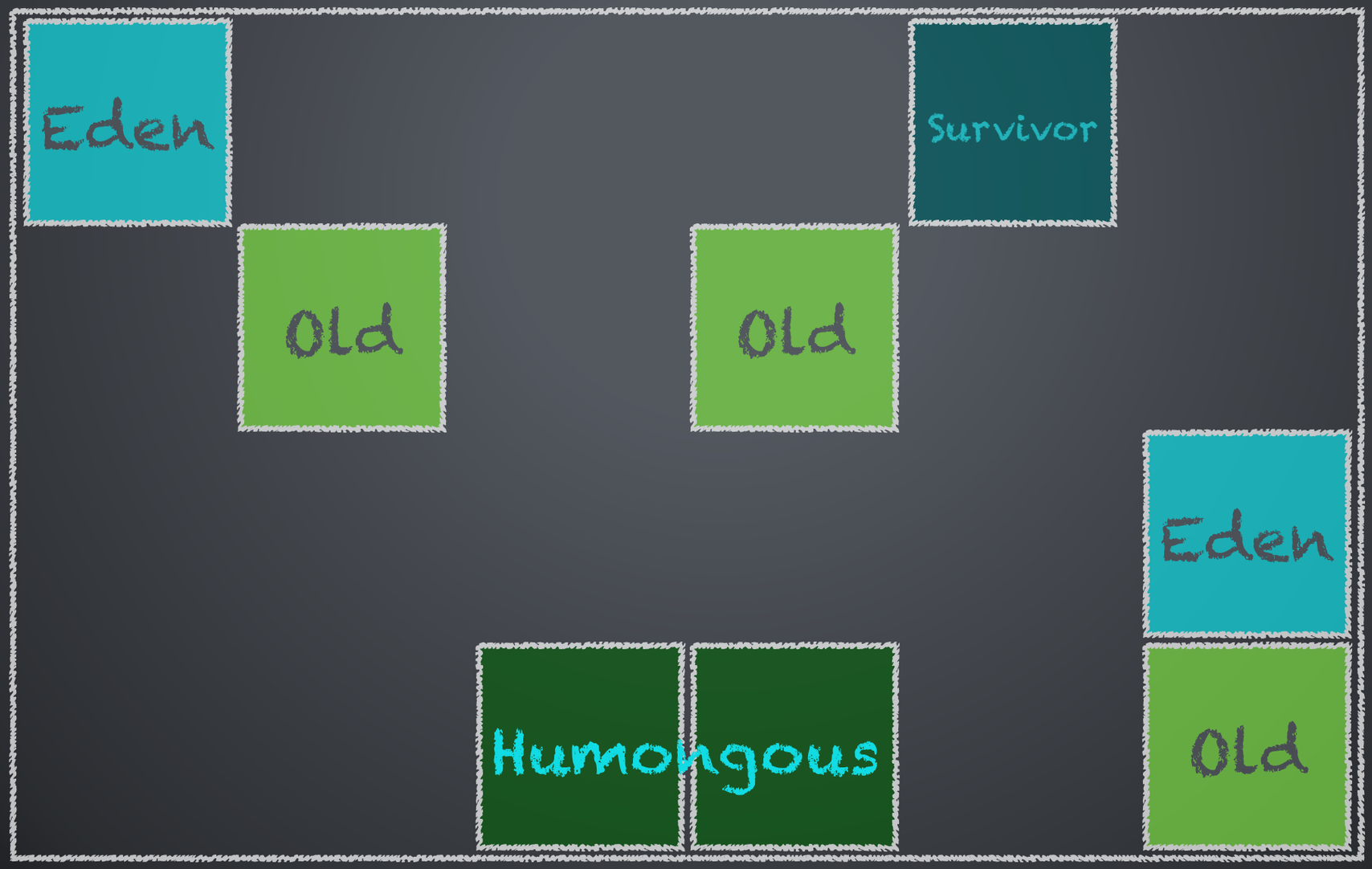


The CMS Collector - Concurrent Mode Failures

- Your old generation is getting filled before a concurrent cycle can complete and free up space.
- Fragmentation has crept in.
- Causes - marking threshold is too high, heap too small, or high application mutation rate

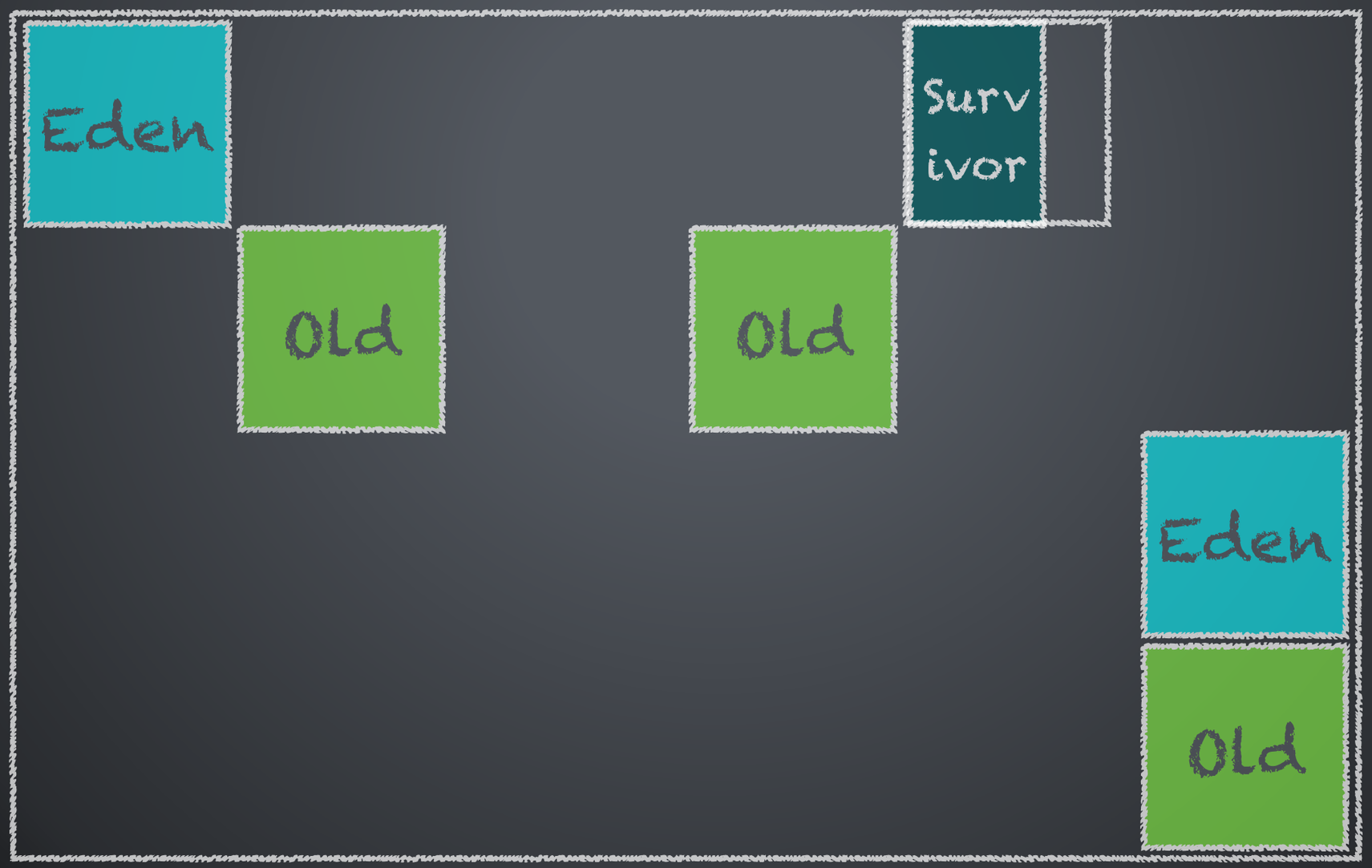
Incremental Compaction In The G1 Collector

The Garbage First Collector - Regionalized Heap

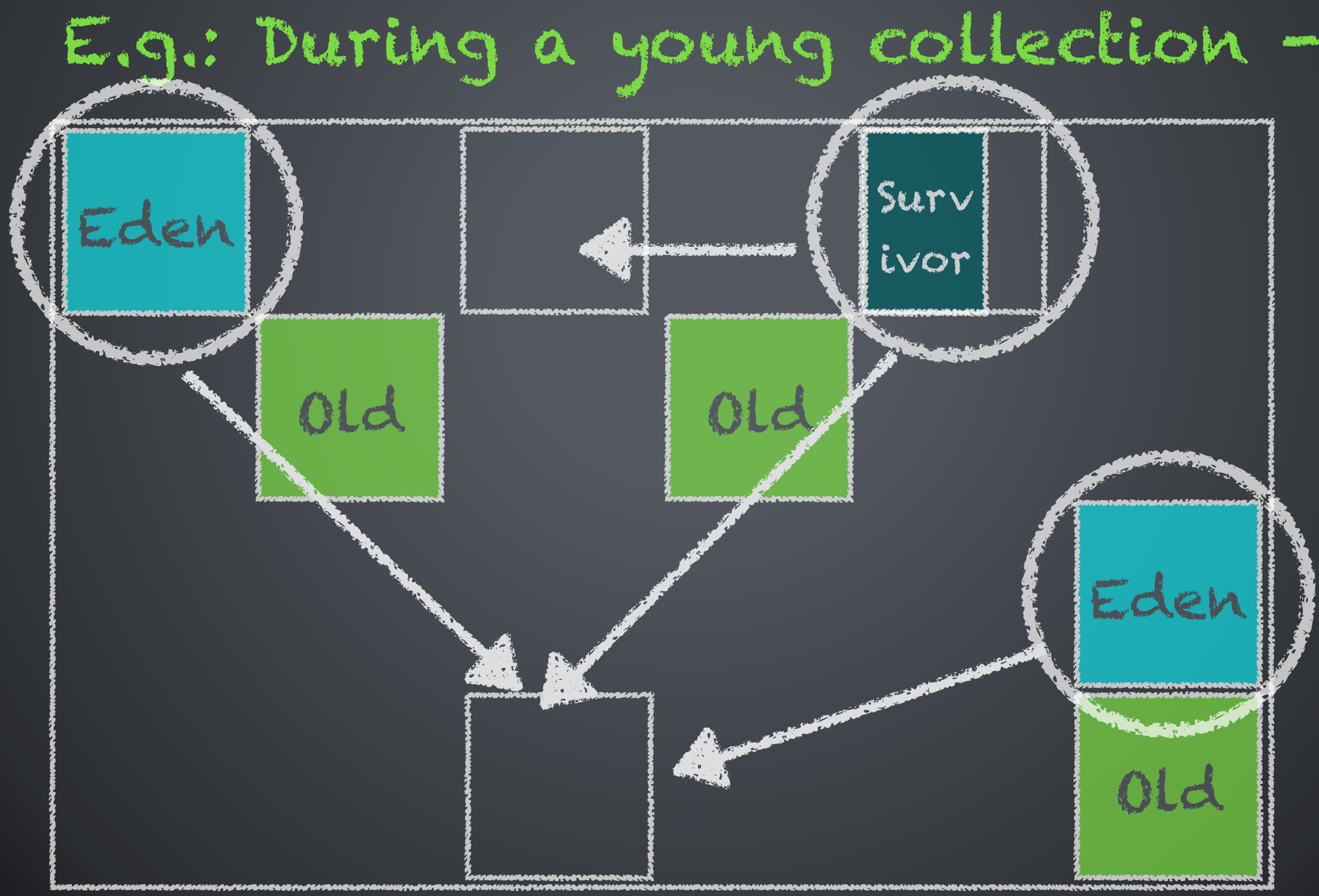


The Garbage First Collector

E.g.: Current heap configuration -

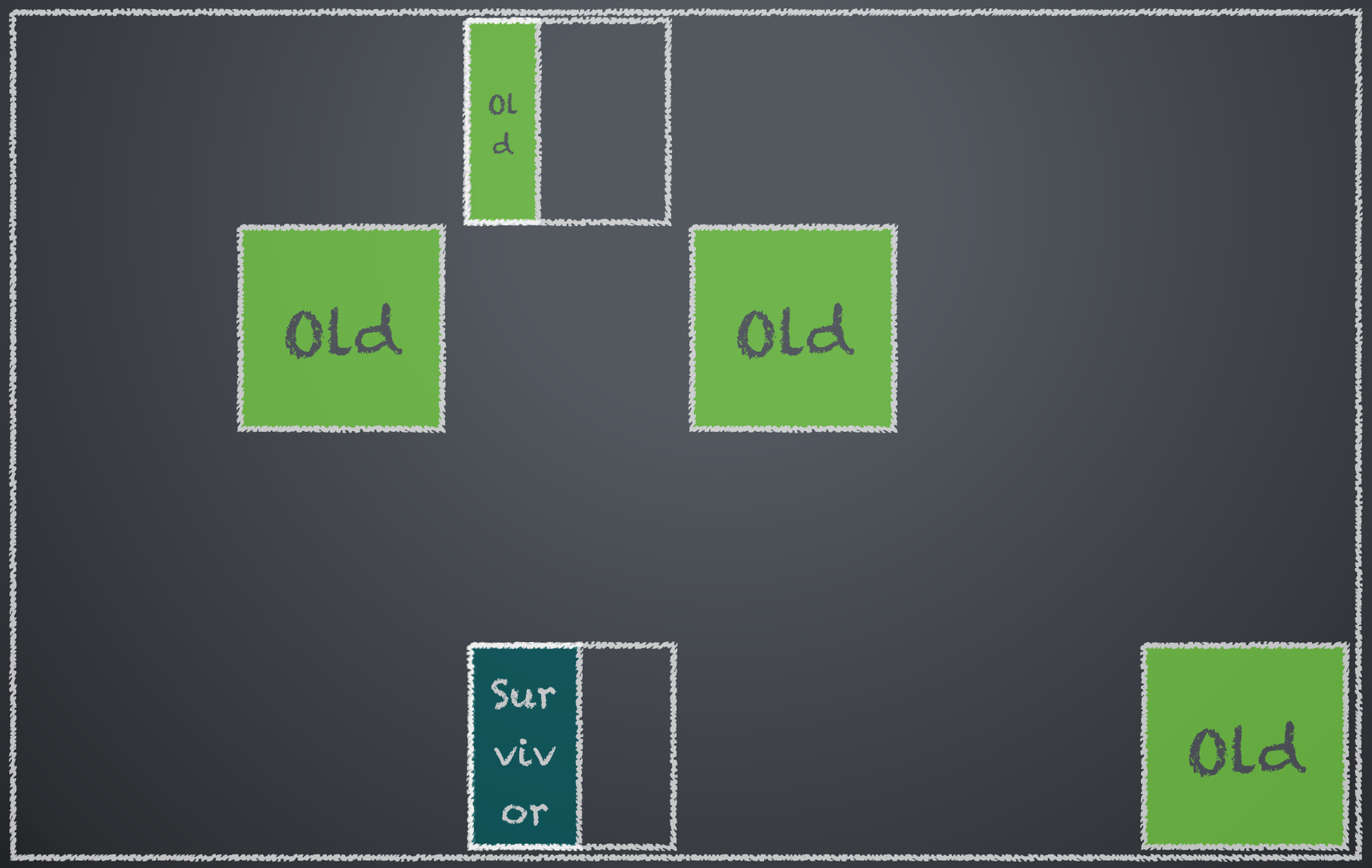


The Garbage First Collector



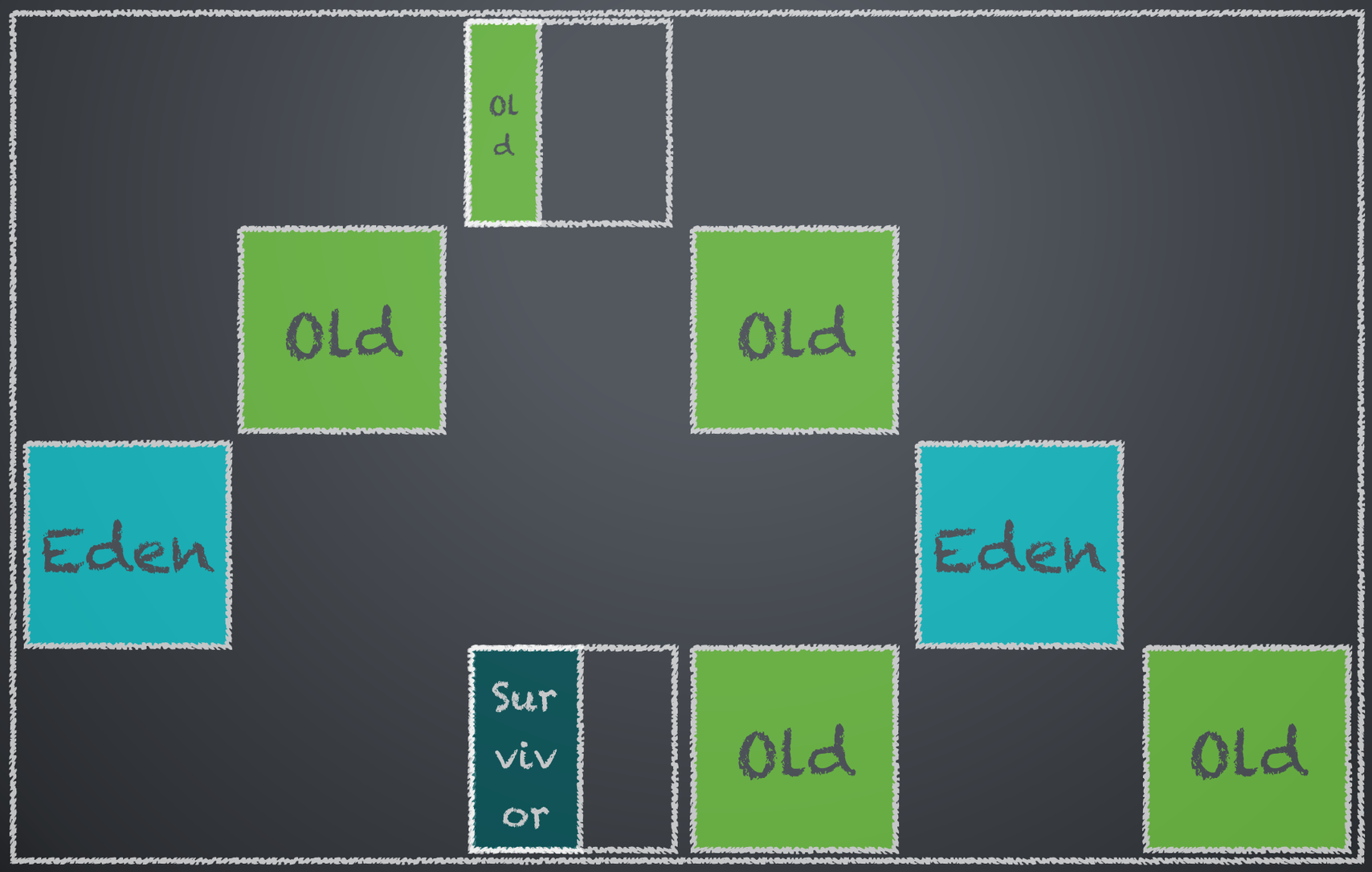
The Garbage First Collector

E.g.: After a young collection -



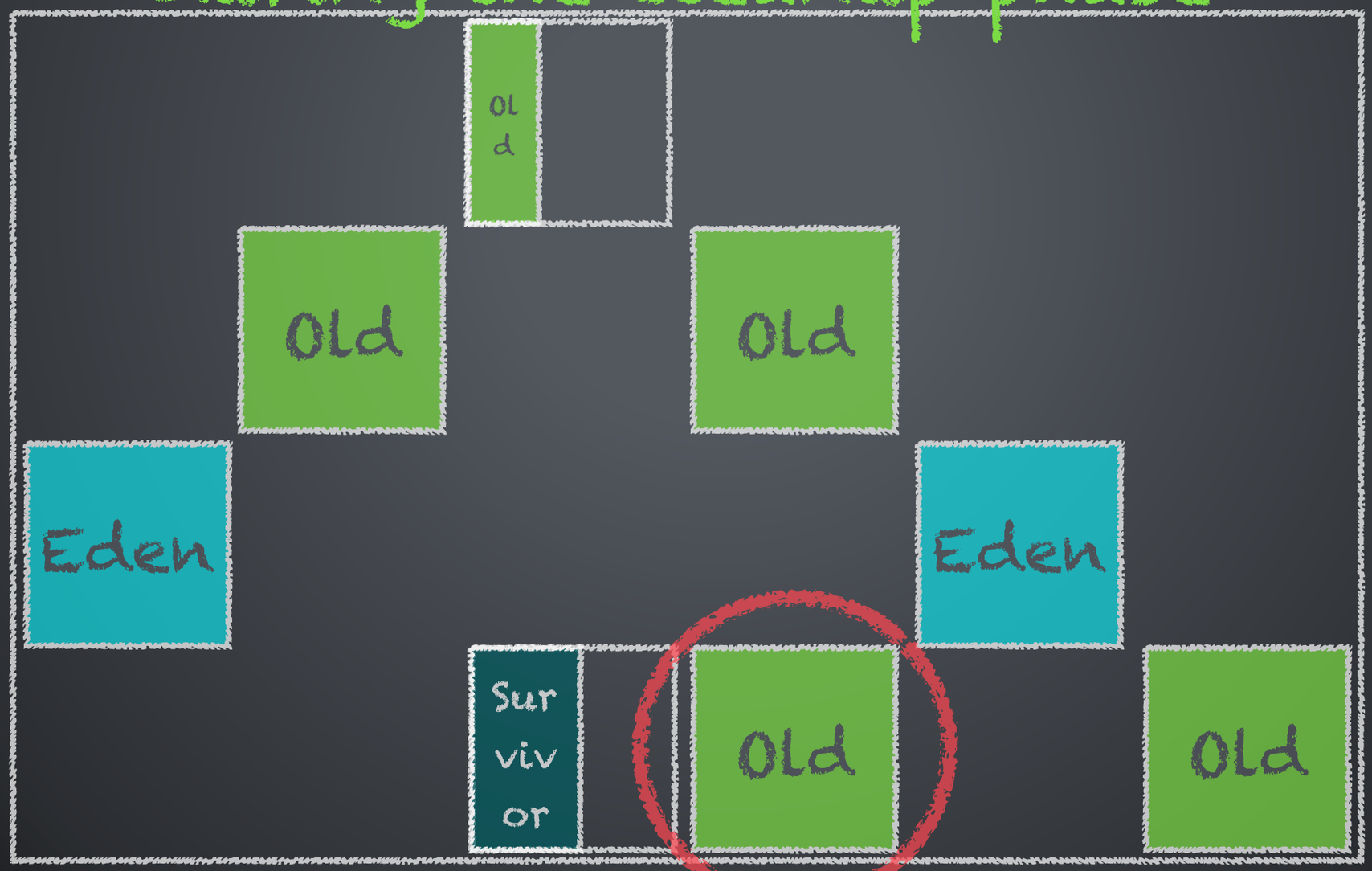
The Garbage First Collector

E.g.: Current heap configuration -



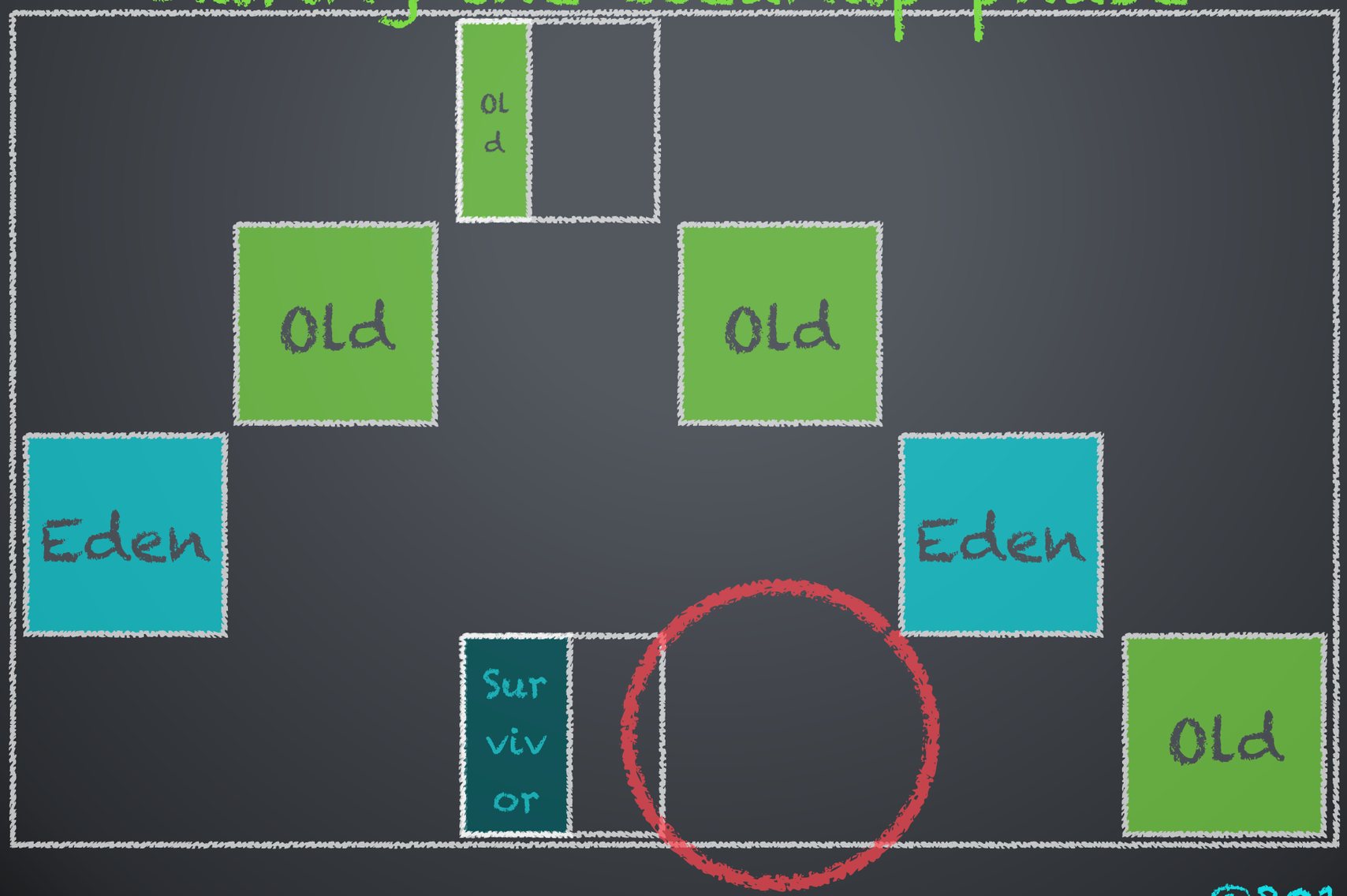
The Garbage First Collector

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



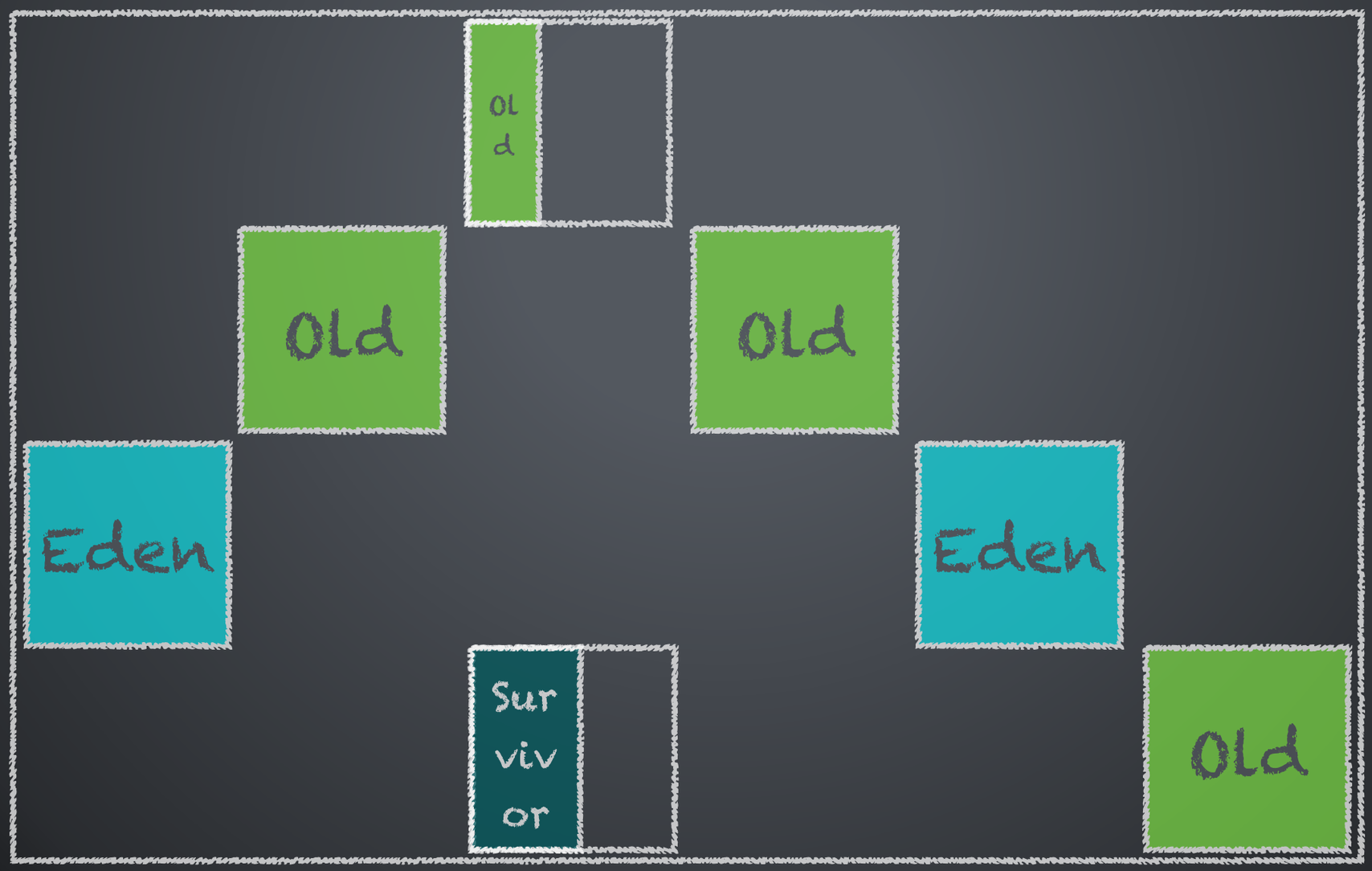
The Garbage First Collector

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



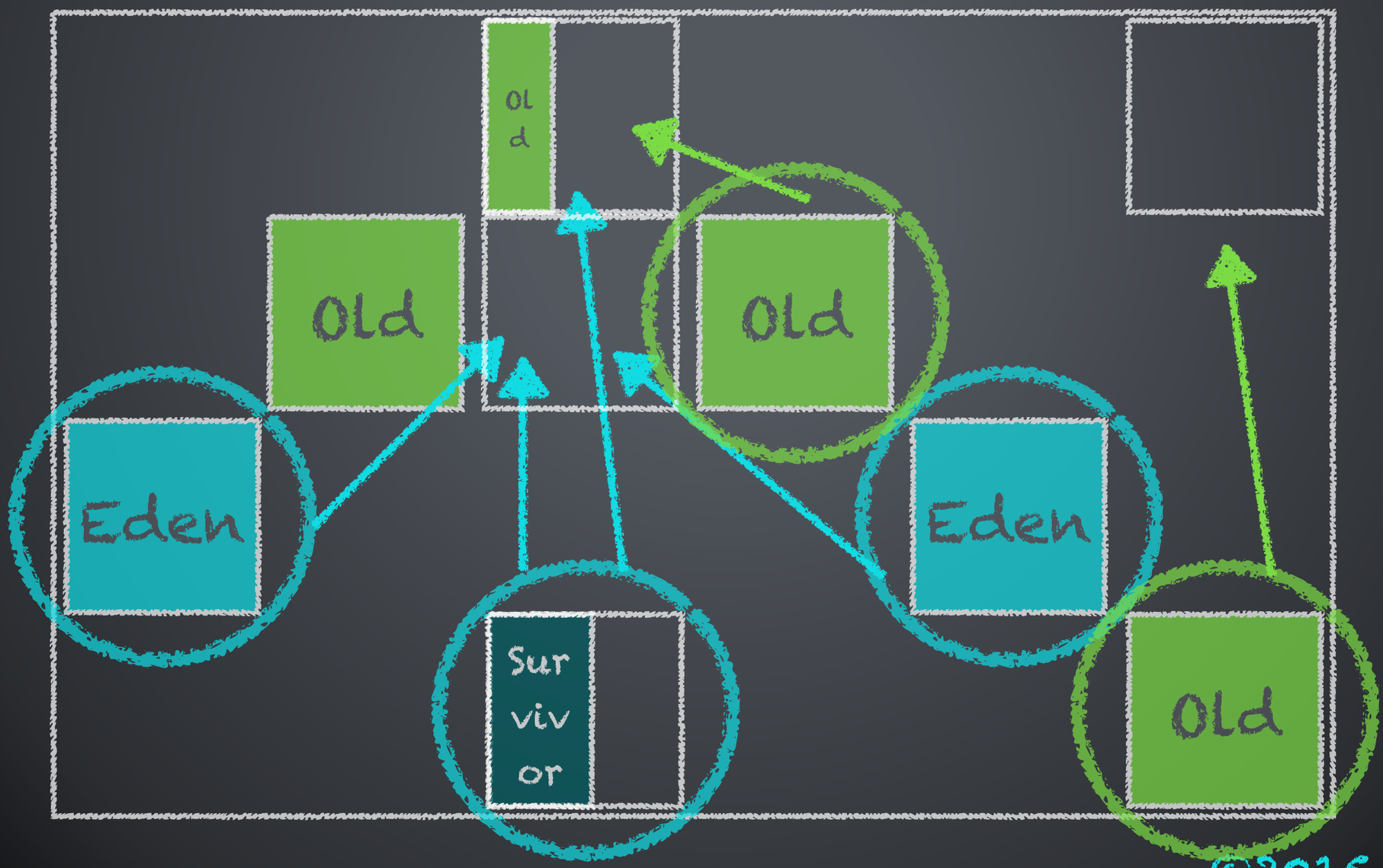
The Garbage First Collector

E.g.: Current heap configuration -



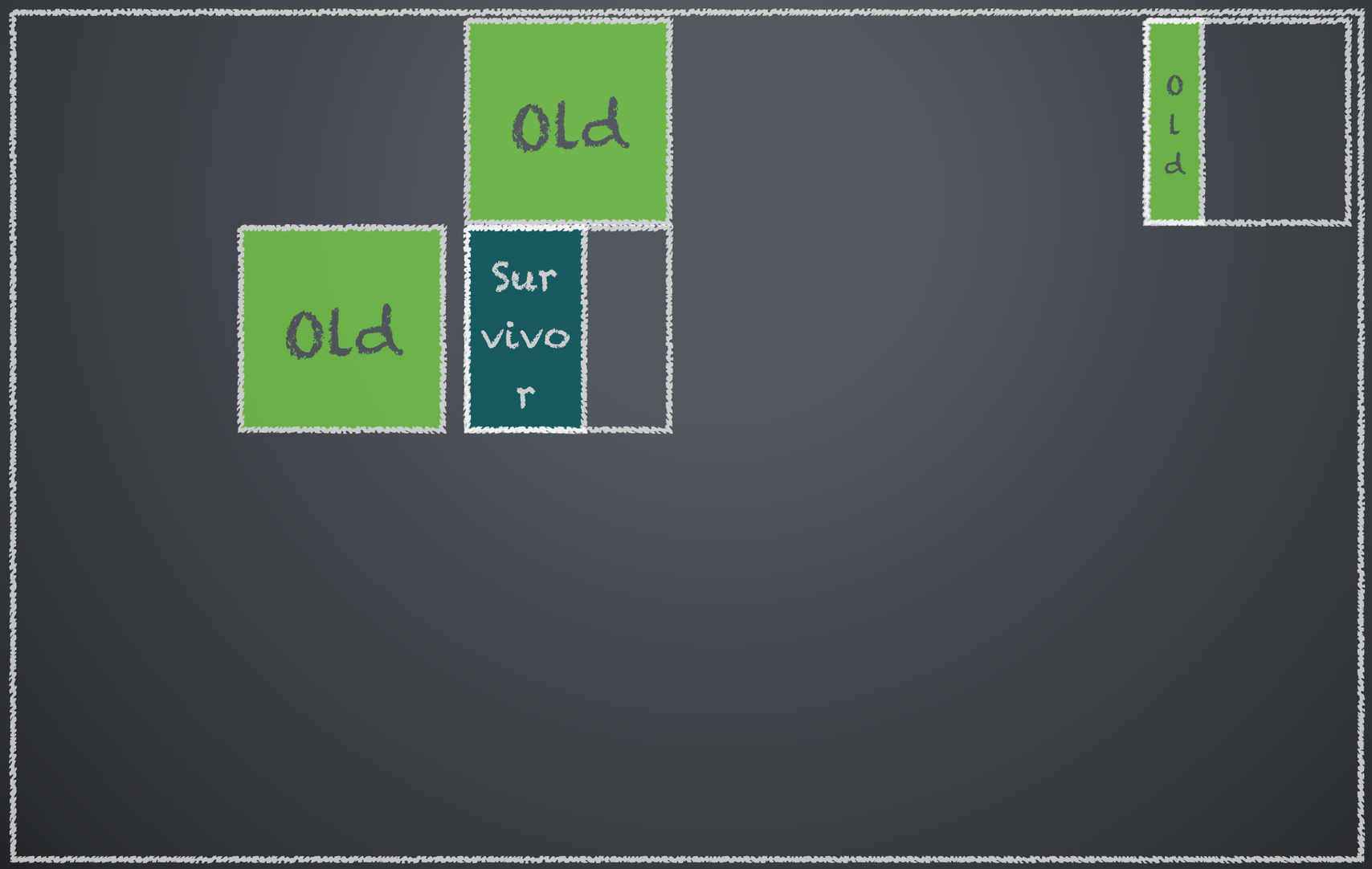
The Garbage First Collector

E.g.: During a mixed collection -



The Garbage First Collector

E.g.: After a mixed collection -



Promotion/Evacuation Failures In The G1 Collector

The Garbage First Collector - Evacuation Failures

- When there are no more regions available for survivors or tenured objects, G1 GC encounters an evacuation failure.

- An evacuation failure is expensive and the usual pattern is that if you see a couple of evacuation failures; full GC could* soon follow.

The Garbage First Collector - Avoiding Evacuation Failures

A heavily tuned JVM command line may be restricting the G1 GC ergonomics and adaptability.

- Start with just your heap sizes and a reasonable pause time goal

The Garbage First Collector - Avoiding Evacuation Failures

Your live data set + long live transient data may be too large for the old generation

- Check LDS+ and increase heap to accommodate everything in the old generation.

The Garbage First Collector - Avoiding Evacuation Failures

Initiating Heap Occupancy Threshold could be the issue.

- Check IHOP and make sure it accommodates the LDS+.
- IHOP threshold too high -> Delayed marking -> Delayed incremental compaction -> Evacuation Failures!

The Garbage First Collector - Avoiding Evacuation Failures

Marking Cycle could be taking too long to complete?

- Increase concurrent marking threads
- Reduce IHOP

The Garbage First Collector - Avoiding Evacuation Failures

to-space survivors are the problem?

- Increase the G1ReservePercent, if to-space survivors are triggering the evacuation failures!

The Garbage First Collector - Avoiding Evacuation Failures

fragmentation an issue?

Fragmentation In The G1 Collector

G1 Heap Waste Percentage

- G1 GC is designed to “absorb” some fragmentation.
- Default is 5% of the total Java heap
- Tradeoff so that expensive regions are left out.

G1 Mixed GC (Region) Liveness Threshold

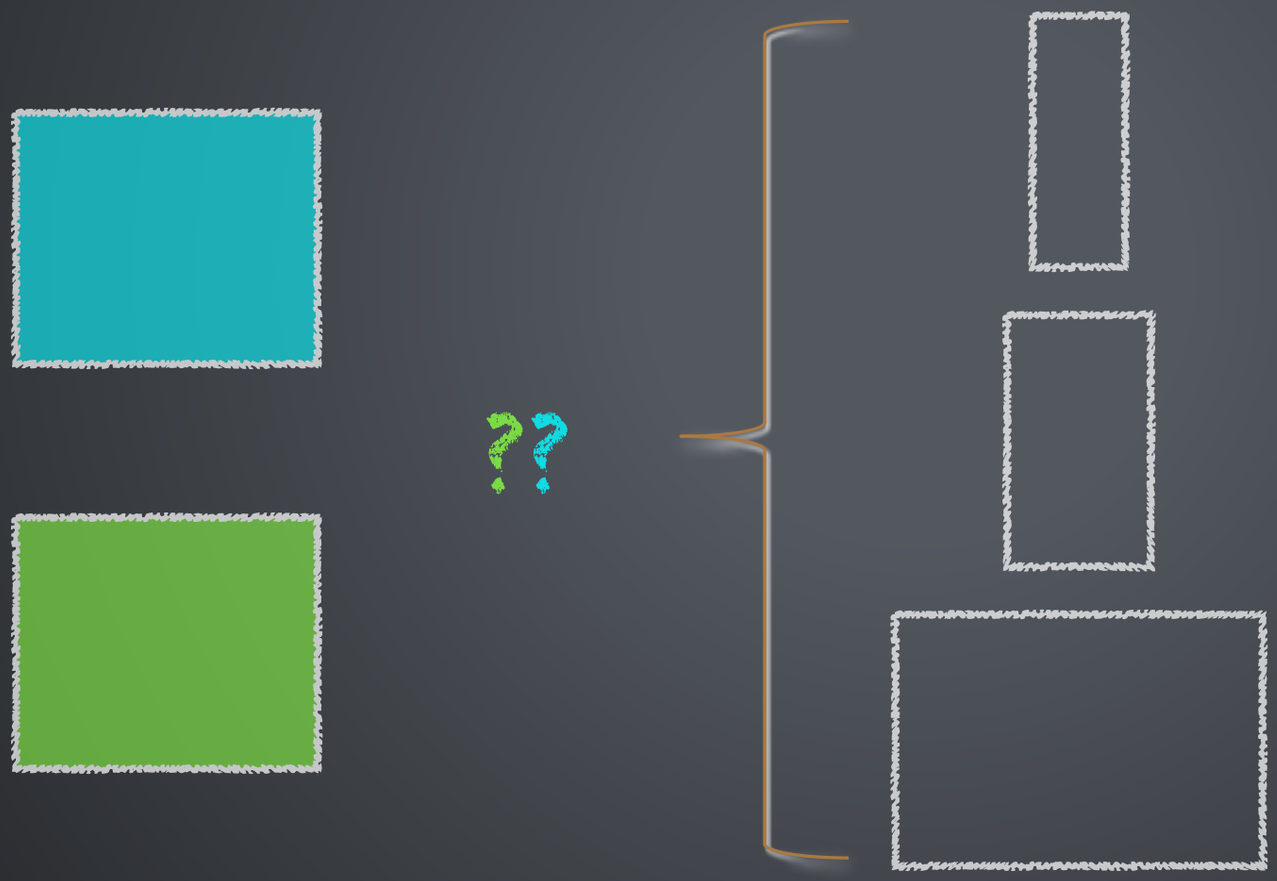
- G1 GC's old regions are also designed to "absorb" some fragmentation.
- Default is 85% liveness in a G1 region.
- Tradeoff so that expensive regions are left out.

The Garbage First Collector - Humongous Objects

A young generation region

An old generation region

The Garbage First Collector - Humongous Objects



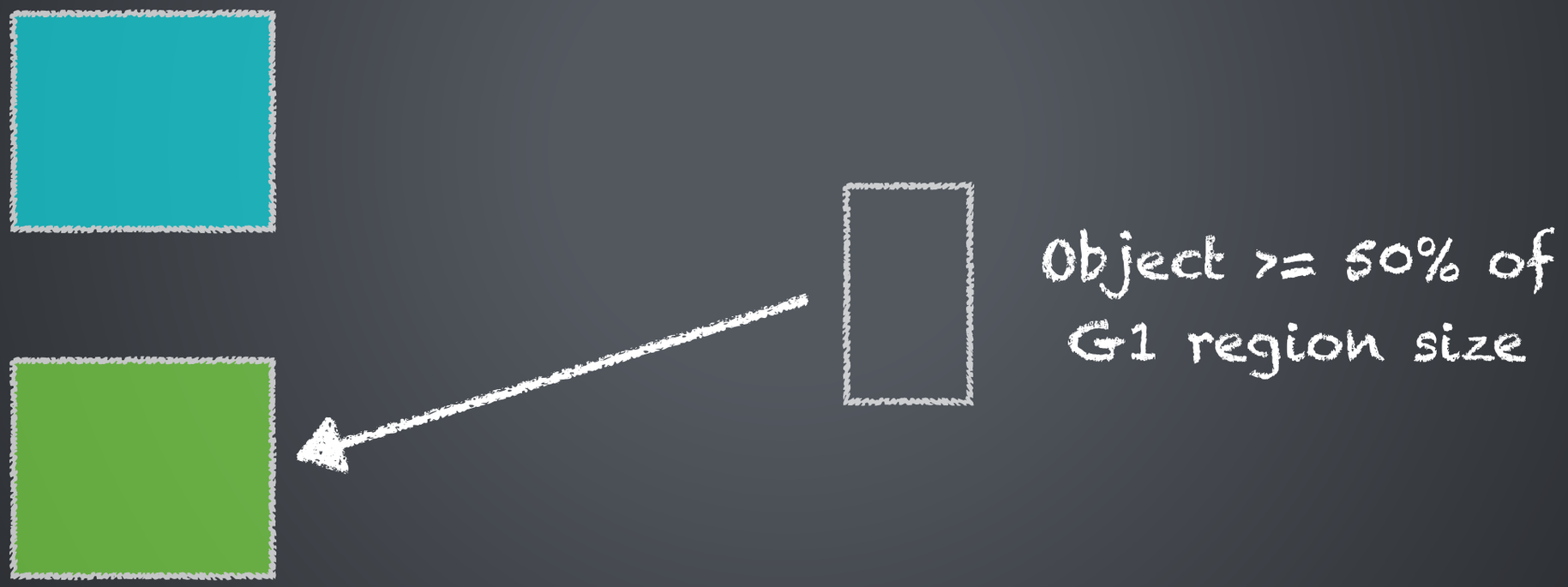
Object < 50% of G1 region size

Object >= 50% of G1 region size

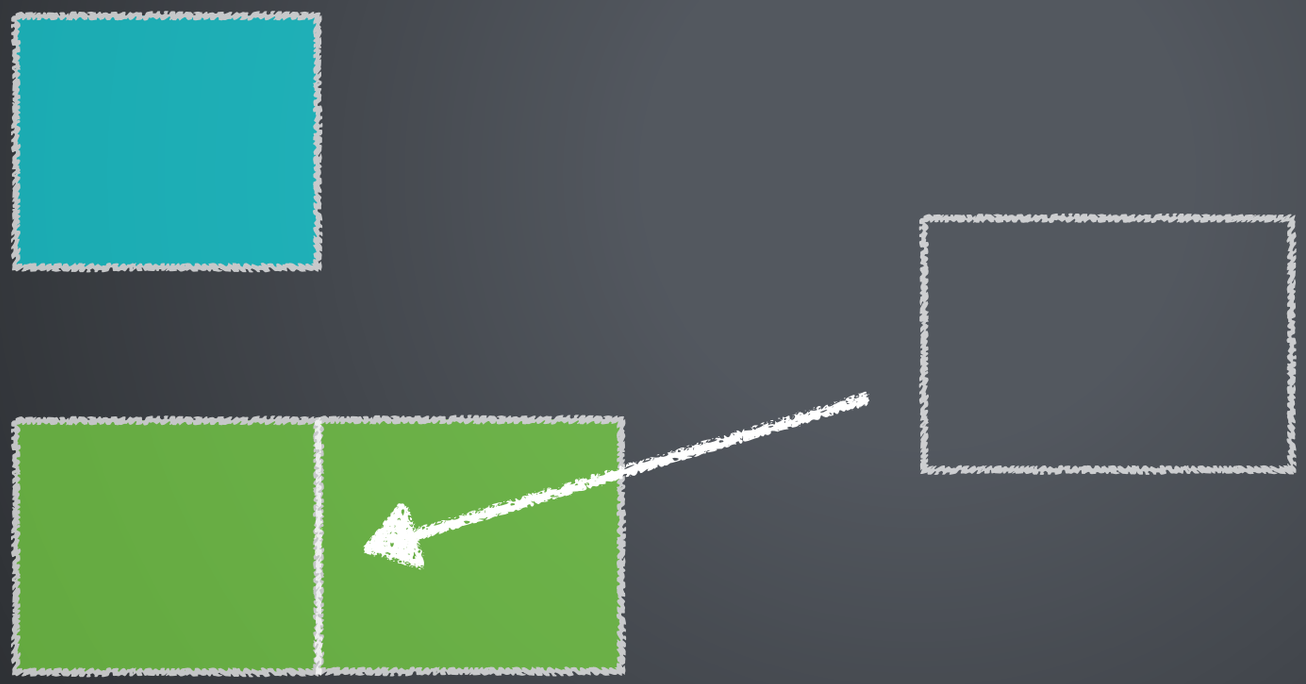
Object > G1 region size

The Garbage First Collector - Humongous Objects

The Garbage First Collector - Humongous Objects

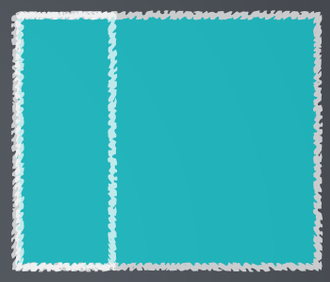


The Garbage First Collector - Humongous Objects



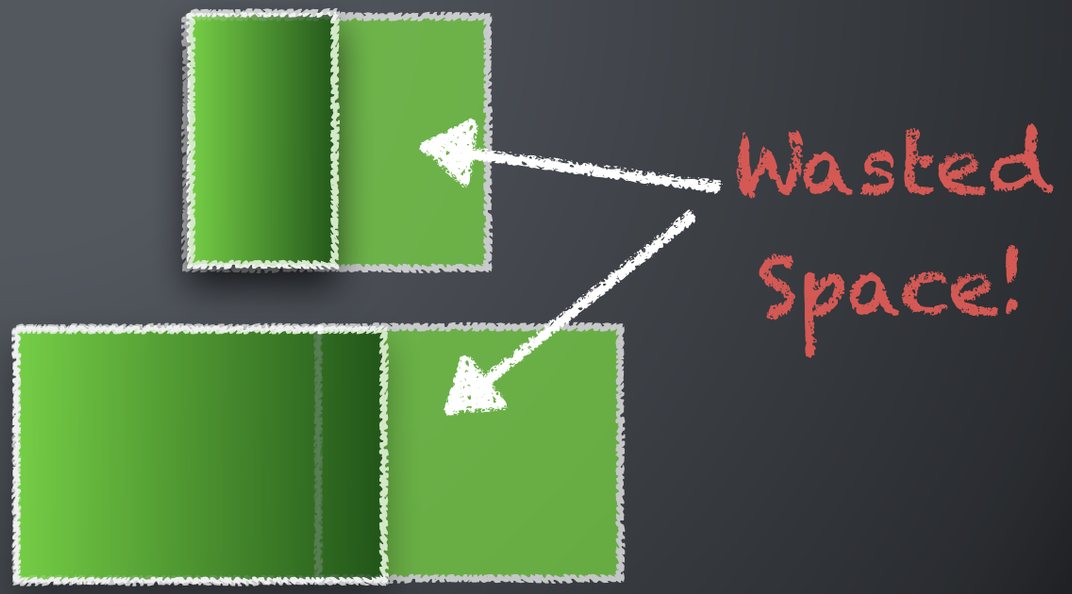
The Garbage First Collector - Humongous Objects

Object NOT Humongous



Object Humongous

Object Humongous ->
Needs Contiguous Regions



The Garbage First Collector - Humongous Objects

Ideally, humongous objects are few in number and are short lived.

- A lot of long-lived humongous objects can cause evacuation failures since humongous regions add to the old generation occupancy.

G1 GC Logs - Key Topics

G1 GC Log

```
154.431: [GC pause (G1 Evacuation Pause) (young), 0.2584864 secs]
   [Parallel Time: 253.2 ms, GC Workers: 8]
      [GC Worker Start (ms): Min: 154431.3, Avg: 154431.4, Max: 154431.5, Diff: 0.1]
      [Ext Root Scanning (ms): Min: 0.1, Avg: 0.2, Max: 0.3, Diff: 0.1, Sum: 1.4]
      [Update RS (ms): Min: 3.3, Avg: 3.5, Max: 3.8, Diff: 0.6, Sum: 28.2]
         [Processed Buffers: Min: 3, Avg: 3.5, Max: 5, Diff: 2, Sum: 28]
      [Scan RS (ms): Min: 46.1, Avg: 46.4, Max: 46.7, Diff: 0.6, Sum: 371.2]
      [Code Root Scanning (ms): Min: 0.0, Avg: 0.1, Max: 0.1, Diff: 0.1, Sum: 0.5]
      [Object Copy (ms): Min: 202.7, Avg: 202.8, Max: 202.9, Diff: 0.3, Sum: 1622.4]
      [Termination (ms): Min: 0.0, Avg: 0.1, Max: 0.1, Diff: 0.1, Sum: 0.5]
      [GC Worker Other (ms): Min: 0.0, Avg: 0.1, Max: 0.1, Diff: 0.1, Sum: 0.6]
      [GC Worker Total (ms): Min: 253.0, Avg: 253.1, Max: 253.1, Diff: 0.1, Sum: 2024.7]
      [GC Worker End (ms): Min: 154684.5, Avg: 154684.5, Max: 154684.5, Diff: 0.1]
   [Code Root Fixup: 0.1 ms]
   [Code Root Purge: 0.0 ms]
   [Clear CT: 0.7 ms]
   [Other: 4.4 ms]
      [Choose CSet: 0.0 ms]
      [Ref Proc: 0.3 ms]
      [Ref Enq: 0.0 ms]
      [Redirty Cards: 0.3 ms]
      [Humongous Reclaim: 0.0 ms]
      [Free CSet: 3.2 ms]
   [Eden: 4972.0M(4972.0M)->0.0B(4916.0M) Survivors: 148.0M->204.0M Heap: 5295.8M(10.0G)->379.4M(10.0G)]
[Times: user=1.72 sys=0.14, real=0.26 secs]
```

G1 GC Log

```
154.431: [GC pause (G1 Evacuation Pause) (young), 0.2584864 secs]
   [Parallel Time: 253.2 ms, GC Workers: 8]
      [GC Worker Start (ms): Min: 154431.3, Avg: 154431.4, Max: 154431.5, Diff: 0.1]
      [Ext Root Scanning (ms): Min: 0.1, Avg: 0.2, Max: 0.3, Diff: 0.1, Sum: 1.4]
      [Update RS (ms): Min: 3.3, Avg: 3.5, Max: 3.8, Diff: 0.6, Sum: 28.2]
         [Processed Buffers: Min: 3, Avg: 3.5, Max: 5, Diff: 2, Sum: 28]
      [Scan RS (ms): Min: 46.1, Avg: 46.4, Max: 46.7, Diff: 0.6, Sum: 371.2]
      [Code Root Scanning (ms): Min: 0.0, Avg: 0.1, Max: 0.1, Diff: 0.1, Sum: 0.5]
      [Object Copy (ms): Min: 202.7, Avg: 202.8, Max: 202.9, Diff: 0.3, Sum: 1622.4]
      [Termination (ms): Min: 0.0, Avg: 0.1, Max: 0.1, Diff: 0.1, Sum: 0.5]
      [GC Worker Other (ms): Min: 0.0, Avg: 0.1, Max: 0.1, Diff: 0.1, Sum: 0.6]
      [GC Worker Total (ms): Min: 253.0, Avg: 253.1, Max: 253.1, Diff: 0.1, Sum: 2024.7]
      [GC Worker End (ms): Min: 154684.5, Avg: 154684.5, Max: 154684.5, Diff: 0.1]
   [Code Root Fixup: 0.1 ms]
   [Code Root Purge: 0.0 ms]
   [Clear CT: 0.7 ms]
   [Other: 4.4 ms]
      [Choose CSet: 0.0 ms]
      [Ref Proc: 0.3 ms]
      [Ref Enq: 0.0 ms]
      [Redirty Cards: 0.3 ms]
      [Humongous Reclaim: 0.0 ms]
      [Free CSet: 3.2 ms]
   [Eden: 4972.0M(4972.0M)->0.0B(4916.0M) Survivors: 148.0M->204.0M Heap: 5295.8M(10.0G)->379.4M(10.0G)]
[Times: user=1.72 sys=0.14, real=0.26 secs]
```

G1 GC Log

```
154.431: [GC pause (G1 Evacuation Pause) (young), 0.2584864 secs]
   [Parallel Time: 253.2 ms, GC Workers: 8]
      [GC Worker Start (ms): Min: 154431.3, Avg: 154431.4, Max: 154431.5, Diff: 0.1]
      [Ext Root Scanning (ms): Min: 0.1, Avg: 0.2, Max: 0.3, Diff: 0.1, Sum: 1.4]
      [Update RS (ms): Min: 3.3, Avg: 3.5, Max: 3.8, Diff: 0.6, Sum: 28.2]
         [Processed Buffers: Min: 3, Avg: 3.5, Max: 5, Diff: 2, Sum: 28]
      [Scan RS (ms): Min: 46.1, Avg: 46.4, Max: 46.7, Diff: 0.6, Sum: 371.2]
      [Code Root Scanning (ms): Min: 0.0, Avg: 0.1, Max: 0.1, Diff: 0.1, Sum: 0.5]
      [Object Copy (ms): Min: 202.7, Avg: 202.8, Max: 202.9, Diff: 0.3, Sum: 1622.4]
      [Termination (ms): Min: 0.0, Avg: 0.1, Max: 0.1, Diff: 0.1, Sum: 0.5]
      [GC Worker Other (ms): Min: 0.0, Avg: 0.1, Max: 0.1, Diff: 0.1, Sum: 0.6]
      [GC Worker Total (ms): Min: 253.0, Avg: 253.1, Max: 253.1, Diff: 0.1, Sum: 2024.7]
      [GC Worker End (ms): Min: 154684.5, Avg: 154684.5, Max: 154684.5, Diff: 0.1]
   [Code Root Fixup: 0.1 ms]
   [Code Root Purge: 0.0 ms]
   [Clear CT: 0.7 ms]
   [Other: 4.4 ms]
      [Choose CSet: 0.0 ms]
      [Ref Proc: 0.3 ms]
      [Ref Enq: 0.0 ms]
      [Redirty Cards: 0.3 ms]
      [Humongous Reclaim: 0.0 ms]
      [Free CSet: 3.2 ms]
   [Eden: 4972.0M(4972.0M)->0.0B(4916.0M) Survivors: 148.0M->204.0M Heap: 5295.8M(10.0G)->379.4M(10.0G)]
[Times: user=1.72 sys=0.14, real=0.26 secs]
```

G1 GC Log

```
154.431: [GC pause (G1 Evacuation Pause) (young), 0.2584864 secs]
   [Parallel Time: 253.2 ms, GC Workers: 8]
      [GC Worker Start (ms): Min: 154431.3, Avg: 154431.4, Max: 154431.5, Diff: 0.1]
      [Ext Root Scanning (ms): Min: 0.1, Avg: 0.2, Max: 0.3, Diff: 0.1, Sum: 1.4]
      [Update RS (ms): Min: 3.3, Avg: 3.5, Max: 3.8, Diff: 0.6, Sum: 28.2]
         [Processed Buffers: Min: 3, Avg: 3.5, Max: 5, Diff: 2, Sum: 28]
      [Scan RS (ms): Min: 46.1, Avg: 46.4, Max: 46.7, Diff: 0.6, Sum: 371.2]
      [Code Root Scanning (ms): Min: 0.0, Avg: 0.1, Max: 0.1, Diff: 0.1, Sum: 0.5]
      [Object Copy (ms): Min: 202.7, Avg: 202.8, Max: 202.9, Diff: 0.3, Sum: 1622.4]
      [Termination (ms): Min: 0.0, Avg: 0.1, Max: 0.1, Diff: 0.1, Sum: 0.5]
      [GC Worker Other (ms): Min: 0.0, Avg: 0.1, Max: 0.1, Diff: 0.1, Sum: 0.6]
      [GC Worker Total (ms): Min: 253.0, Avg: 253.1, Max: 253.1, Diff: 0.1, Sum: 2024.7]
      [GC Worker End (ms): Min: 154684.5, Avg: 154684.5, Max: 154684.5, Diff: 0.1]
   [Code Root Fixup: 0.1 ms]
   [Code Root Purge: 0.0 ms]
   [Clear CT: 0.7 ms]
   [Other: 4.4 ms]
      [Choose CSet: 0.0 ms]
      [Ref Proc: 0.3 ms]
      [Ref Enq: 0.0 ms]
      [Redirty Cards: 0.3 ms]
      [Humongous Reclaim: 0.0 ms]
      [Free CSet: 3.2 ms]
   [Eden: 4972.0M(4972.0M)->0.0B(4916.0M) Survivors: 148.0M->204.0M Heap: 5295.8M(10.0G)->379.4M(10.0G)]
 [Times: user=1.72 sys=0.14, real=0.26 secs]
```

G1 GC Log

```
154.431: [GC pause (G1 Evacuation Pause) (young), 0.2584864 secs]
   [Parallel Time: 253.2 ms, GC Workers: 8]
      [GC Worker Start (ms): Min: 154431.3, Avg: 154431.4, Max: 154431.5, Diff: 0.1]
      [Ext Root Scanning (ms): Min: 0.1, Avg: 0.2, Max: 0.3, Diff: 0.1, Sum: 1.4]
      [Update RS (ms): Min: 3.3, Avg: 3.5, Max: 3.8, Diff: 0.6, Sum: 28.2]
         [Processed Buffers: Min: 3, Avg: 3.5, Max: 5, Diff: 2, Sum: 28]
      [Scan RS (ms): Min: 46.1, Avg: 46.4, Max: 46.7, Diff: 0.6, Sum: 371.2]
      [Code Root Scanning (ms): Min: 0.0, Avg: 0.1, Max: 0.1, Diff: 0.1, Sum: 0.5]
      [Object Copy (ms): Min: 202.7, Avg: 202.8, Max: 202.9, Diff: 0.3, Sum: 1622.4]
      [Termination (ms): Min: 0.0, Avg: 0.1, Max: 0.1, Diff: 0.1, Sum: 0.5]
      [GC Worker Other (ms): Min: 0.0, Avg: 0.1, Max: 0.1, Diff: 0.1, Sum: 0.6]
      [GC Worker Total (ms): Min: 253.0, Avg: 253.1, Max: 253.1, Diff: 0.1, Sum: 2024.7]
      [GC Worker End (ms): Min: 154684.5, Avg: 154684.5, Max: 154684.5, Diff: 0.1]
   [Code Root Fixup: 0.1 ms]
   [Code Root Purge: 0.0 ms]
   [Clear CT: 0.7 ms]
   [Other: 4.4 ms]
      [Choose CSet: 0.0 ms]
      [Ref Proc: 0.3 ms]
      [Ref Enq: 0.0 ms]
      [Redirty Cards: 0.3 ms]
      [Humongous Reclaim: 0.0 ms]
      [Free CSet: 3.2 ms]
   [Eden: 4972.0M(4972.0M)->0.0B(4916.0M) Survivors: 148.0M->204.0M Heap: 5295.8M(10.0G)->379.4M(10.0G)]
[Times: user=1.72 sys=0.14, real=0.26 secs]
```

G1 GC Log

```
154.431: [GC pause (G1 Evacuation Pause) (young), 0.2584864 secs]
   [Parallel Time: 253.2 ms, GC Workers: 8]
      [GC Worker Start (ms): Min: 154431.3, Avg: 154431.4, Max: 154431.5, Diff: 0.1]
      [Ext Root Scanning (ms): Min: 0.1, Avg: 0.2, Max: 0.3, Diff: 0.1, Sum: 1.4]
      [Update RS (ms): Min: 3.3, Avg: 3.5, Max: 3.8, Diff: 0.6, Sum: 28.2]
         [Processed Buffers: Min: 3, Avg: 3.5, Max: 5, Diff: 2, Sum: 28]
      [Scan RS (ms): Min: 46.1, Avg: 46.4, Max: 46.7, Diff: 0.6, Sum: 371.2]
      [Code Root Scanning (ms): Min: 0.0, Avg: 0.1, Max: 0.1, Diff: 0.1, Sum: 0.5]
      [Object Copy (ms): Min: 202.7, Avg: 202.8, Max: 202.9, Diff: 0.3, Sum: 1622.4]
      [Termination (ms): Min: 0.0, Avg: 0.1, Max: 0.1, Diff: 0.1, Sum: 0.5]
      [GC Worker Other (ms): Min: 0.0, Avg: 0.1, Max: 0.1, Diff: 0.1, Sum: 0.6]
      [GC Worker Total (ms): Min: 253.0, Avg: 253.1, Max: 253.1, Diff: 0.1, Sum: 2024.7]
      [GC Worker End (ms): Min: 154684.5, Avg: 154684.5, Max: 154684.5, Diff: 0.1]
   [Code Root Fixup: 0.1 ms]
   [Code Root Purge: 0.0 ms]
   [Clear CT: 0.7 ms]
   [Other: 4.4 ms]
      [Choose CSet: 0.0 ms]
      [Ref Proc: 0.3 ms]
      [Ref Enq: 0.0 ms]
      [Redirty Cards: 0.3 ms]
      [Humongous Reclaim: 0.0 ms]
      [Free CSet: 3.2 ms]
   [Eden: 4972.0M(4972.0M)->0.0B(4916.0M) Survivors: 148.0M->204.0M Heap: 5295.8M(10.0G)->379.4M(10.0G)]
[Times: user=1.72 sys=0.14, real=0.26 secs]
```

G1 GC Log

```
154.431: [GC pause (G1 Evacuation Pause) (young), 0.2584864 secs]
   [Parallel Time: 253.2 ms, GC Workers: 8]
      [GC Worker Start (ms): Min: 154431.3, Avg: 154431.4, Max: 154431.5, Diff: 0.1]
      [Ext Root Scanning (ms): Min: 0.1, Avg: 0.2, Max: 0.3, Diff: 0.1, Sum: 1.4]
      [Update RS (ms): Min: 3.3, Avg: 3.5, Max: 3.8, Diff: 0.6, Sum: 28.2]
         [Processed Buffers: Min: 3, Avg: 3.5, Max: 5, Diff: 2, Sum: 28]
      [Scan RS (ms): Min: 46.1, Avg: 46.4, Max: 46.7, Diff: 0.6, Sum: 371.2]
      [Code Root Scanning (ms): Min: 0.0, Avg: 0.1, Max: 0.1, Diff: 0.1, Sum: 0.5]
      [Object Copy (ms): Min: 202.7, Avg: 202.8, Max: 202.9, Diff: 0.3, Sum: 1622.4]
      [Termination (ms): Min: 0.0, Avg: 0.1, Max: 0.1, Diff: 0.1, Sum: 0.5]
      [GC Worker Other (ms): Min: 0.0, Avg: 0.1, Max: 0.1, Diff: 0.1, Sum: 0.6]
      [GC Worker Total (ms): Min: 253.0, Avg: 253.1, Max: 253.1, Diff: 0.1, Sum: 2024.7]
      [GC Worker End (ms): Min: 154684.5, Avg: 154684.5, Max: 154684.5, Diff: 0.1]
   [Code Root Fixup: 0.1 ms]
   [Code Root Purge: 0.0 ms]
   [Clear CT: 0.7 ms]
   [Other: 4.4 ms]
      [Choose CSet: 0.0 ms]
      [Ref Proc: 0.3 ms]
      [Ref Enq: 0.0 ms]
      [Redirty Cards: 0.3 ms]
      [Humongous Reclaim: 0.0 ms]
      [Free CSet: 3.2 ms]
   [Eden: 4972.0M(4972.0M)->0.0B(4916.0M) Survivors: 148.0M->204.0M Heap: 5295.8M(10.0G)->379.4M(10.0G)]
[Times: user=1.72 sys=0.14, real=0.26 secs]
```

G1 GC Log

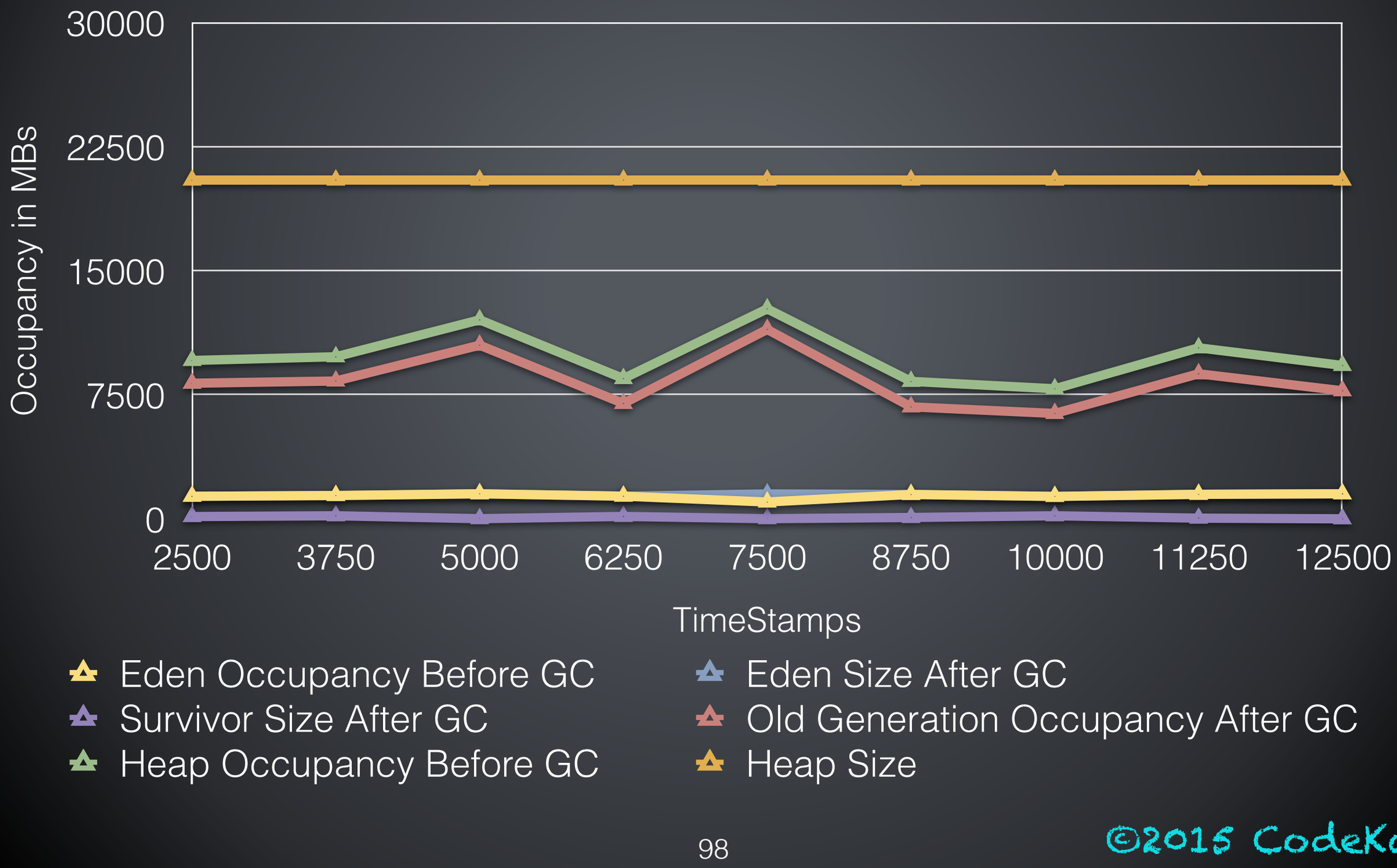
```
154.431: [GC pause (G1 Evacuation Pause) (young), 0.2584864 secs]
   [Parallel Time: 253.2 ms, GC Workers: 8]
      [GC Worker Start (ms): Min: 154431.3, Avg: 154431.4, Max: 154431.5, Diff: 0.1]
      [Ext Root Scanning (ms): Min: 0.1, Avg: 0.2, Max: 0.3, Diff: 0.1, Sum: 1.4]
      [Update RS (ms): Min: 3.3, Avg: 3.5, Max: 3.8, Diff: 0.6, Sum: 28.2]
         [Processed Buffers: Min: 3, Avg: 3.5, Max: 5, Diff: 2, Sum: 28]
      [Scan RS (ms): Min: 46.1, Avg: 46.4, Max: 46.7, Diff: 0.6, Sum: 371.2]
      [Code Root Scanning (ms): Min: 0.0, Avg: 0.1, Max: 0.1, Diff: 0.1, Sum: 0.5]
      [Object Copy (ms): Min: 202.7, Avg: 202.8, Max: 202.9, Diff: 0.3, Sum: 1622.4]
      [Termination (ms): Min: 0.0, Avg: 0.1, Max: 0.1, Diff: 0.1, Sum: 0.5]
      [GC Worker Other (ms): Min: 0.0, Avg: 0.1, Max: 0.1, Diff: 0.1, Sum: 0.6]
      [GC Worker Total (ms): Min: 253.0, Avg: 253.1, Max: 253.1, Diff: 0.1, Sum: 2024.7]
      [GC Worker End (ms): Min: 154684.5, Avg: 154684.5, Max: 154684.5, Diff: 0.1]
   [Code Root Fixup: 0.1 ms]
   [Code Root Purge: 0.0 ms]
   [Clear CT: 0.7 ms]
   [Other: 4.4 ms]
      [Choose CSet: 0.0 ms]
      [Ref Proc: 0.3 ms]
      [Ref Enq: 0.0 ms]
      [Redirty Cards: 0.3 ms]
      [Humongous Reclaim: 0.0 ms]
      [Free CSet: 3.2 ms]
   [Eden: 4972.0M(4972.0M)->0.0B(4916.0M) Survivors: 148.0M->204.0M Heap: 5295.8M(10.0G)->379.4M(10.0G)]
[Times: user=1.72 sys=0.14, real=0.26 secs]
```

Generation Sizing

```
[Eden: 4972.0M(4972.0M)->0.0B(4916.0M) Survivors:
148.0M->204.0M Heap: 5295.8M(10.0G)->379.4M(10.0G)]
```

[Eden: Occupancy before GC(Eden size before GC)->Occupancy after GC(Eden size after GC) Survivors: Size before GC->Size after GC Heap: Occupancy before GC(Heap size before GC)->Occupancy after GC(Heap size after GC)]

Heap Information Plot

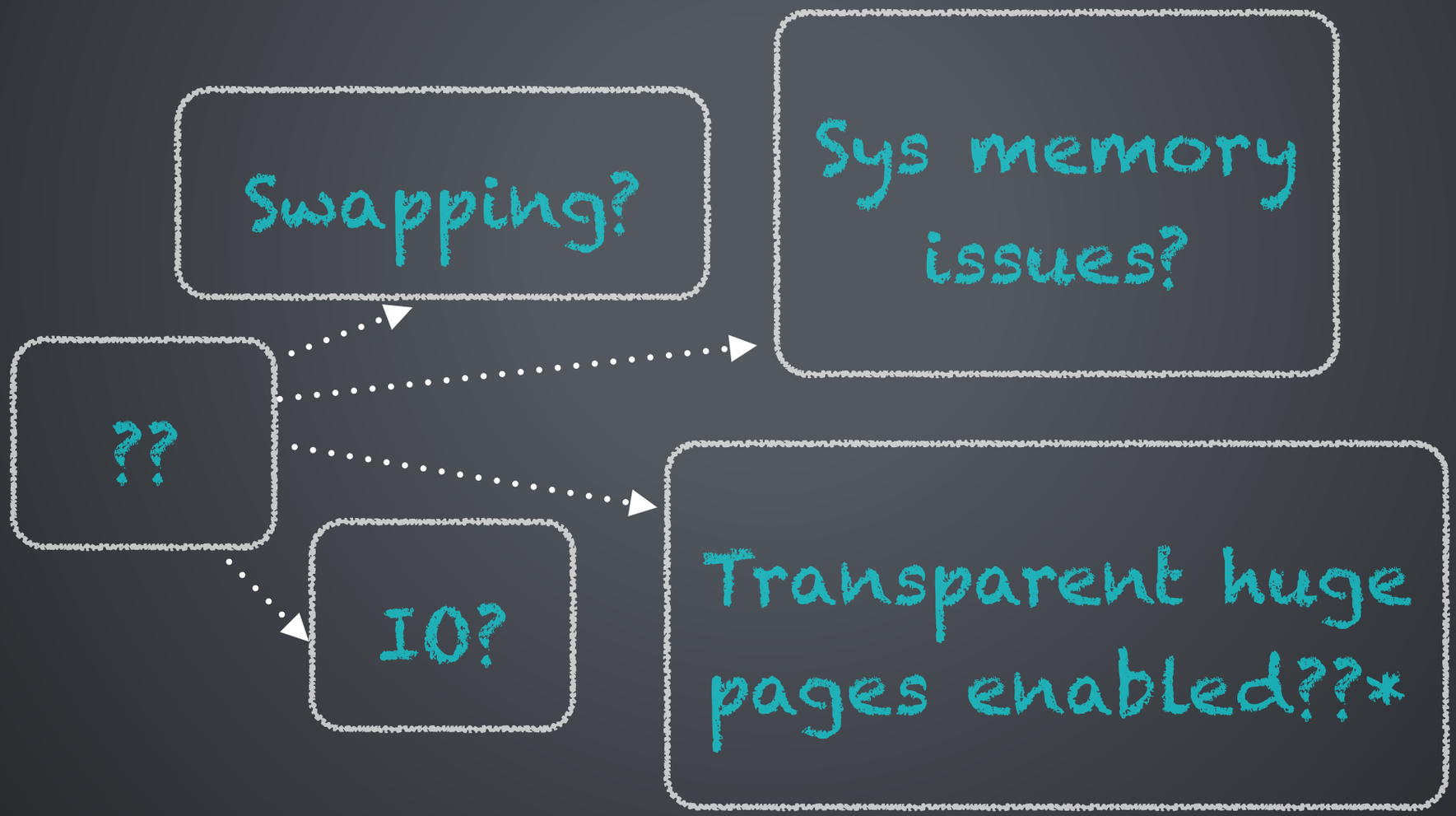


Scaling: GC Times vs Sys Times

```
[Times: user=1.72 sys=0.14, real=0.26 secs]
```

Scaling: user time/real time = 6.6

Scaling: High Sys Times



Reference Processing Times

```
[Ref Proc: 0.3 ms]
```

Check for high time spent in ‘Ref Proc’.

Also, check your Remark pause times:

```
9.972: [GC remark 9.972: [Finalize Marking, 0.0007865 secs]
9.973: [GC ref-proc, 0.0027669 secs] 9.976: [Unloading,
0.0075832 secs], 0.0116215 secs]
```

If Remark pauses are high or increasing and if ‘ref-proc’ is the major contributor - use: -XX: +ParallelRefProcEnabled

GC Overhead vs Elapsed Time

- 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

- 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

Summary

What have we learned so far?

Most Allocations —Fast Path→ Eden Space

Eden Full? Start Young Collection: Keep Allocating :)

Objects Aged in Survivor Space? Promote: Keep Aging :)

Promotions —Fast Path★→ Old Generation

★ CMS promotes to fitting free space out of a free list.

GC Tuneables - The Throughput Collector

Goal:

Only promote objects after you have ~~hazed~~ aged them appropriately

Tunables:

Everything related to aging objects and generation sizing -

NewRatio, (Max)NewSize, SurvivorRatio, (Max)TenuringThreshold

GC Tuneables - The Throughput Collector

Things to remember -

- Applications with steady behavior rarely need AdaptiveSizePolicy to be enabled.
- Overflow gets promoted into the old generation
- Provide larger survivor spaces for transient data.
- In most cases, young generation sizing has the most effect on throughput
  - Size the young generation to maintain the GC overhead to less than 5%.

GC Tuneables - The CMS Collector

Goal:

Only promote objects after you have ~~hazed~~ aged them appropriately

Tunables:

Everything related to aging objects and young generation sizing still applies here.

The concurrent thread counts and marking threshold are addition tunables for CMS

GC Tuneables - The CMS Collector

Things to remember -

- Premature promotions are very expensive in CMS and could lead to fragmentation
- You can reduce the CMS cycle duration by adding more concurrent threads: ConcGCThreads.
  - Remember that this will increase the concurrent overhead.
- You can manually tune the marking threshold (adaptive by default)
  - CMSInitiatingOccupancyFraction & UseCMSInitiatingOccupancyOnly will help fix the marking threshold.
  - Note: The threshold is expressed as a percentage of the old generation occupancy

GC Tuneables - The G1 Collector

Goal:

Get the GC ergonomics to work for you and know the defaults

Tunables:

- Pause time goal, heap size, max and min nursery, concurrent and parallel threads

- The marking threshold, number of mixed GCs after marking, liveness threshold for the old regions, garbage toleration threshold, max old regions to be collected per mixed collection

GC Tuneables - The G1 Collector

Things to remember -

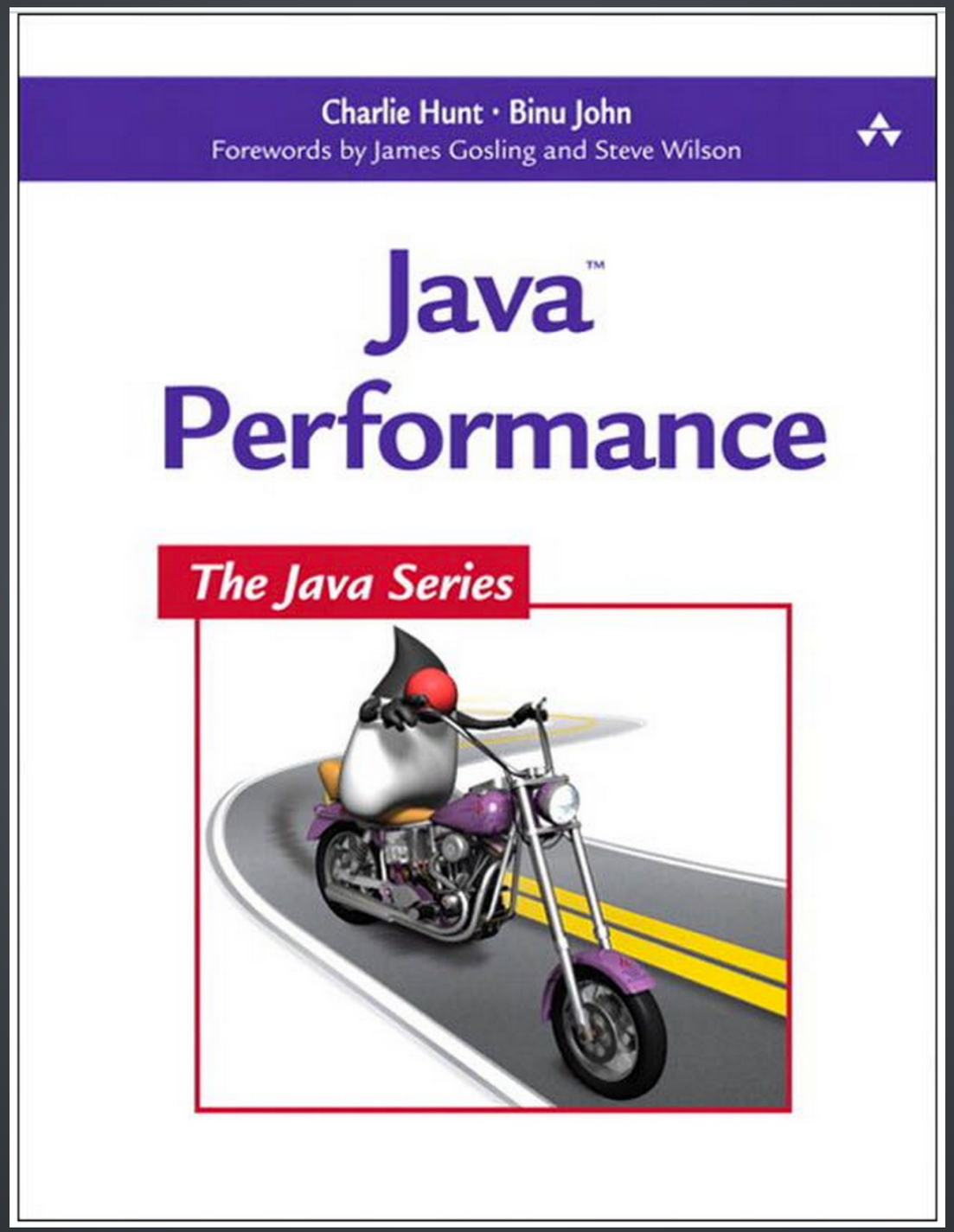
- Know your defaults!
    - Understand your G1HeapRegionSize - It could be any factor of two from 1MB to 32MB. G1 strives for 2048 regions.
- Fixing the nursery size (using Xmn) will meddle with the GC ergonomics/adaptiveness.
- Don’t set really aggressive pause time goals - this will increase the GC overhead.
- Spend time taming your mixed GCs - mixed GCs are incremental collections

GC Tuneables - The G1 Collector

Things to remember -

- Taming mixed GCs:
    - Adjust the marking cycle according to you live data set.
    - Adjust you liveness threshold - this is the live occupancy threshold per region. Any region with liveness beyond this threshold will not be included in a mixed collection.
    - Adjust your garbage toleration threshold - helps G1 not get too aggressive with mixed collections
    - Distribute mixed GC pauses over a number of mixed collections - adjust your mixed GC count target and change your max old region threshold percent so that you can limit the old regions per collection

Further Reading



Further Reading

- Jon Masa's blog: https://blogs.oracle.com/jonthecollector/entry/our_collectors
- A few of my articles on InfoQ: http://www.infoq.com/author/Monica-Beckwith
- Presentations: http://www.slideshare.net/MonicaBeckwith
- Mail archives on hotspot-gc-use@openjdk.java.net & hotspot-gc-dev@openjdk.java.net

Appendix

Performance Analysis

- Monitoring: System Under Load
  - Utilization - CPU, IO, Sys/ Kernel, Memory bandwidth, Java heap, …
  - Lock statistics
- Analyzing:
  - Utilization and time spent - GC logs, CPU, memory and application logs
- Profiling:
  - Application, System, Memory - Java Heap.

JVM Performance Engineering

Java/JVM performance engineering includes the study, analysis and tuning of the Just-in-time (JIT) compiler, the Garbage Collector (GC) and many a times tuning related to the Java Development Kit (JDK).

GC Performance Engineering

- Monitor the JVM - Visual VM
- Monitor and collect GC information - Visual GC (online), GC logs (offline)
- Develop scripts to process GC logs; use GC Histo and JFreeCharts to plot your GC logs or use specialized tools/ log analyzers that serves your purpose.

Summary

What have we learned so far? - Young Generation & Collections

- Young generation is always collected in its entirety.
- All 3 server GCs discussed earlier follow similar mechanism for young collection. ★
- The young collections achieve reclamation via compaction and copying of live objects.
- There are a lot of options for sizing the Eden and Survivor space optimally and many GCs also have adaptive sizing and GC ergonomics for young generation collections.

What have we learned so far? - Old Generation & Collections

All 3 server GCs vary in the way they collect the old generation:

- For ParallelOld GC, the old generation is reclaimed and compacted in its entirety

- Luckily, the compaction cost is distributed amongst parallel garbage collector worker threads.

- Unluckily, the compaction cost depends a lot on the make of the live data set since at every compaction, the GC is moving live data around.

- No tuning options other than the generation size adjustment and age threshold for promotion.

What have we learned so far? - Old Generation & Collections

- For CMS GC, the old generation is (mostly) concurrently marked and swept. Thus the reclamation of dead objects happen in place and the space is added to a free list of spaces.
    - The marking threshold can be tuned adaptively and manually as well.
- Luckily, CMS GC doesn't do compaction, hence reclamation is fast.
- Unluckily, a long running Java application with CMS GC is prone to fragmentation which will eventually result in promotion failures which can eventually lead to full compacting garbage collection and sometimes even concurrent mode failures.
    - Full compacting GCs are singled threaded in CMS GC.

What have we learned so far? - Old Generation & Collections

- For G1 GC, the old generation regions are (mostly) concurrently marked and an incremental compacting collection helps with optimizing the old generation collection.

- Luckily, fragmentation is not “untunable” in G1 GC as it is in CMS GC.

- Unluckily, sometimes, you may still encounter promotion/evacuation failures when G1 GC runs out of regions to copy live objects. Such an evacuation failure is expensive and can eventually lead to a full compacting GC.

    - Full compacting GCs are singled threaded in G1 GC.

    - Appropriate tuning of the old generation space and collection can help avoid evacuation failures and hence keep full GCs at bay.

- G1 GC has multiple tuning options so that the GC can be adapted to your application needs.