GC Tuning Confessions Of A Performance Engineer

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

Philly Emerging Tech Conference 2015
April 8th, 2015

About Me

- JVM/GC Performance Engineer/Consultant
- Worked at AMD, Sun, Oracle
- Worked with HotSpot JVM
- Worked with Parallel(Old) GC, G1 GC and CMS GC

©2015 CodeKaram

About Today's Talk

- A little bit about Performance Engineering
- Insight into Garbage Collectors
- Introduction to a few main GC Algorithms in OpenJDK HotSpot (ParallelOld GC, CMS GC, and G1 GC)
- Summary
- GC Tunables

©2015 CodeKaram

Performance Engineering

- A performance engineer helps ensure that the system is designed + implemented to meet the performance requirements.

- The performance requirements could include the service level agreements (SLAs) for throughput, latency and other response time related metrics - also known as non-functional requirements.

  - E.g. Response time (RT) metrics - Average (RT), max or worst-case RT, 99th percentile RT…

    - Let's talk more about these RTs…

©2015 CodeKaram

Performance Engineering - Response Time Metrics

	Average (ms)	Minimum (ms)
System1	307.741	7.622
System2	320.778	7.258
System3	321.483	6.432
System4	323.143	7.353

©2015 CodeKaram

Performance Engineering - Response Time Metrics

	Average (ms)	Number of GCs
System1	307.741	37353
System2	320.778	34920
System3	321.483	36270
System4	323.143	40636

©2015 CodeKaram

Performance Engineering - Response Time Metrics

	Average (ms)	Maximum (ms)
System1	307.741	3131.331
System2	320.778	2744.588
System3	321.483	1681.308
System4	323.143	**20699.505**

©2015 CodeKaram

Performance Engineering - Response Time Metrics

	Average (ms)	Maximum (ms)
System1	307.741	3131.331
System2	320.778	2744.588
System3	321.483	1681.308
System4	323.143	**20699.505**

5 full GCs and 10 evacuation failures

©2015 CodeKaram

Performance Engineering

- Monitoring, analysis and tuning are a big part of performance engineering.
    - Monitoring utilization - CPU, IO, Memory bandwidth, Java heap, …
    - Analyzing utilization and time spent - GC logs, CPU, memory and application logs
    - Profiling - Application, System, Memory - Java Heap.
- 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).

©2015 CodeKaram

Insight Into Garbage Collectors (GCs)

- A GC is an automatic memory management unit.
- An ideal GC is the one that requires minimum footprint (concurrent CPU or native memory), and provides maximum throughput while minimizing predictable latency.
- Fun Fact - In reality you will have to tradeoff one (footprint or latency or throughput) in lieu of the others. A healthy performance engineering exercise can help you meet or exceed your goals.
- Fun Fact - GC can NOT eliminate your memory leaks!
- Fun Fact - GC (and heap dump) can provide an insight into your application.

©2015 CodeKaram

GC Algorithms in OpenJDK HotSpot - The Tradeoff

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

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

©2015 CodeKaram

GC Algorithms in OpenJDK HotSpot - Throughput Maximizer

- Throughput has driven us to parallelization of GC worker threads:
  - Parallel Collection Threads
  - Parallel Concurrent Marking Threads
- Throughput has driven us to generational GCs
  - Most objects die young.
  - Fast path allocation into “young” generation.
  - Age and then promote (fast path again) to “old” generation
  - Fun Fact: All GCs in OpenJDK HotSpot are generational.

©2015 CodeKaram

GC Algorithms in OpenJDK HotSpot - Latency Sensitive

- Latency has driven algorithms to no compaction or partial compaction
    - Time is of essence, no need to fully compact if free space is still available!
- Latency has driven algorithms towards concurrency - i.e. running with the application threads.
    - Mostly concurrent mark and sweep (CMS) and concurrent marking in G1.
- 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.

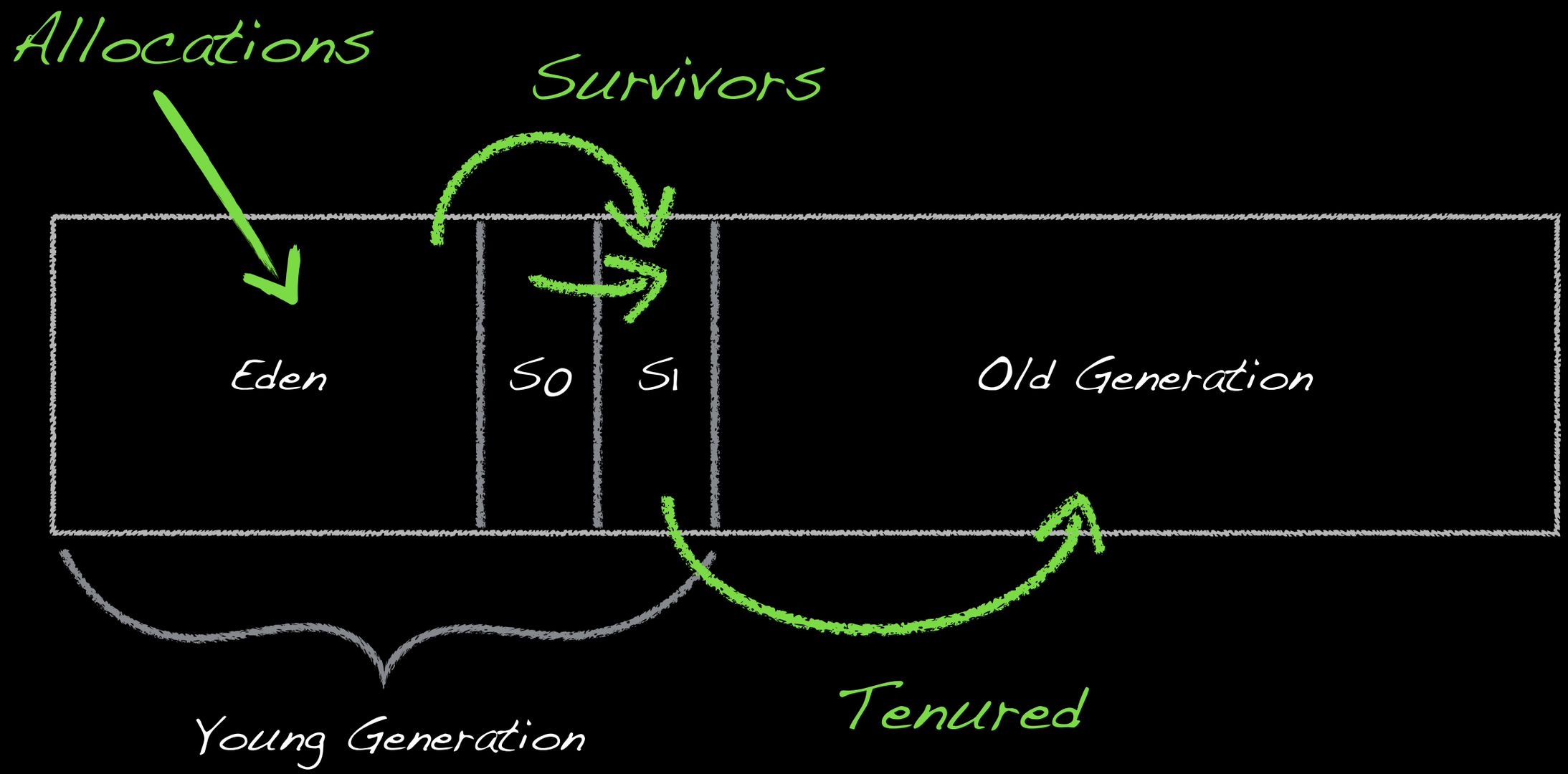
©2015 CodeKaram

The Throughput Collector

- ParallelOld is the throughput collector in OpenJDK HotSpot.
- But, first, what is throughput?
  - Throughput is the percentage of time NOT spent in GC :)
- The throughput goal for ParallelOld Collector is 99%.
  - That is, all the GC pauses that happen during the life of the application should account to 1% of the run time.
- How does ParallelOld try to achieve its throughput goal?

©2015 CodeKaram

The Throughput Collector - Java Heap



©2015 CodeKaram

The Throughput Collector - Young Collection

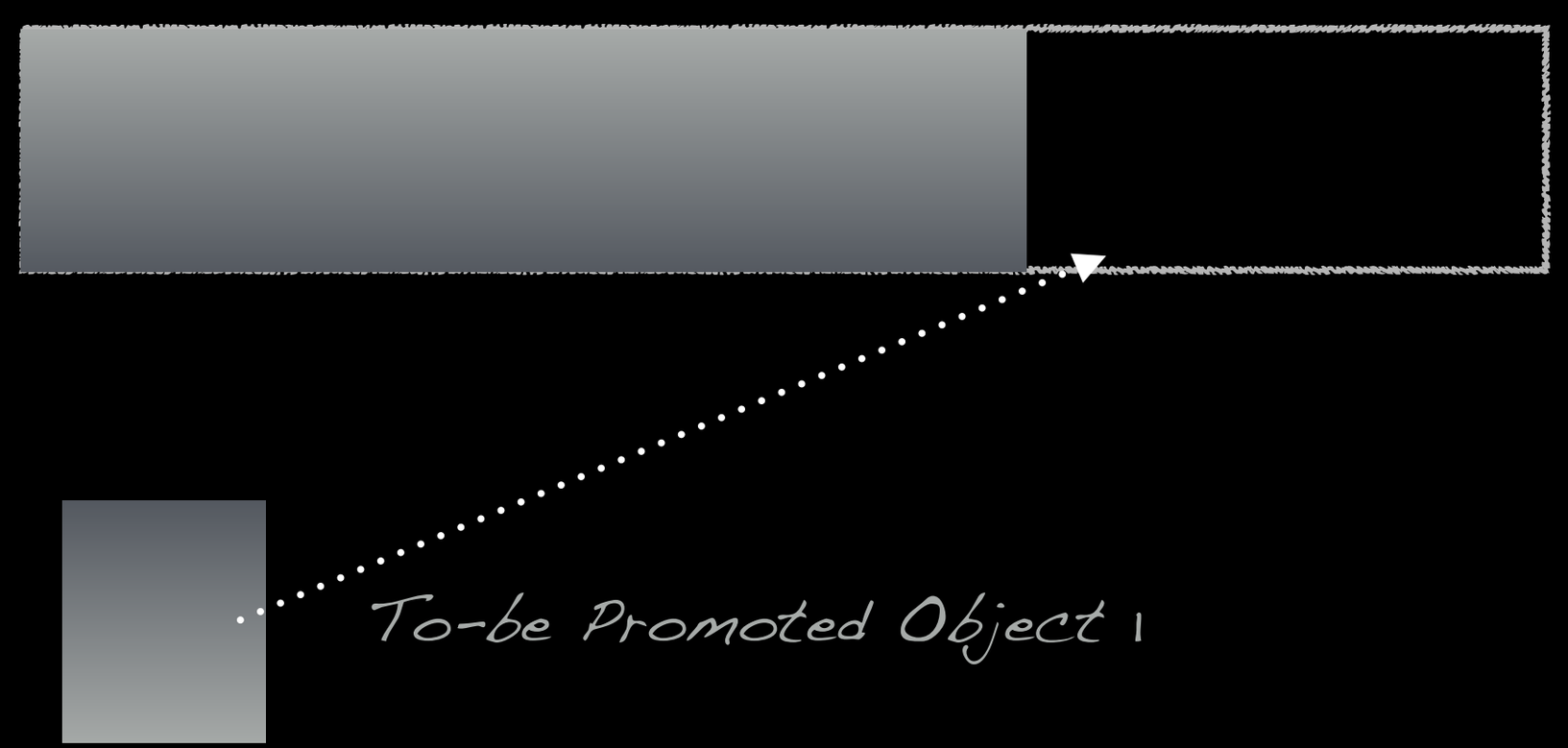
- An allocation failure results in a stop-the-world young collection.
- The young generation is collected in its entirety i.e. all objects (dead or alive) are emptied from the eden and S0 spaces.
- After the young collection is complete, the surviving objects (objects that are live) are moved into S1.
- Objects are aged in the survivor space until ready for promotion
- When the age threshold is met, objects are promoted into the old generation.
- Allocations and promotions are both fast tracked (lock-free) by using Thread/Promotion Local Allocation Buffers (TLAB/PLAB)

©2015 CodeKaram

The Throughput Collector - Contiguous Old Generation

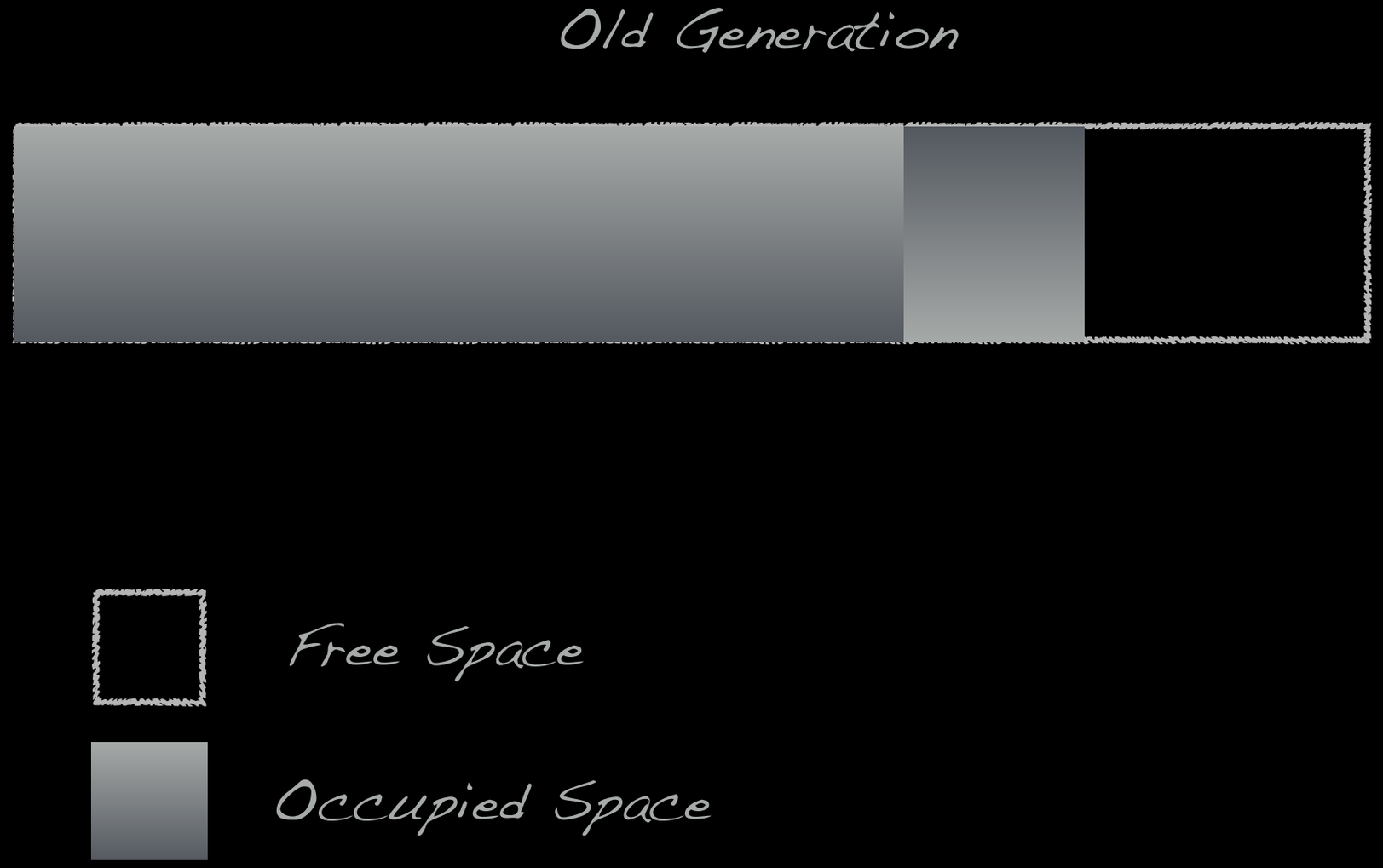
©2015 CodeKaram

The Throughput Collector - Contiguous Old Generation



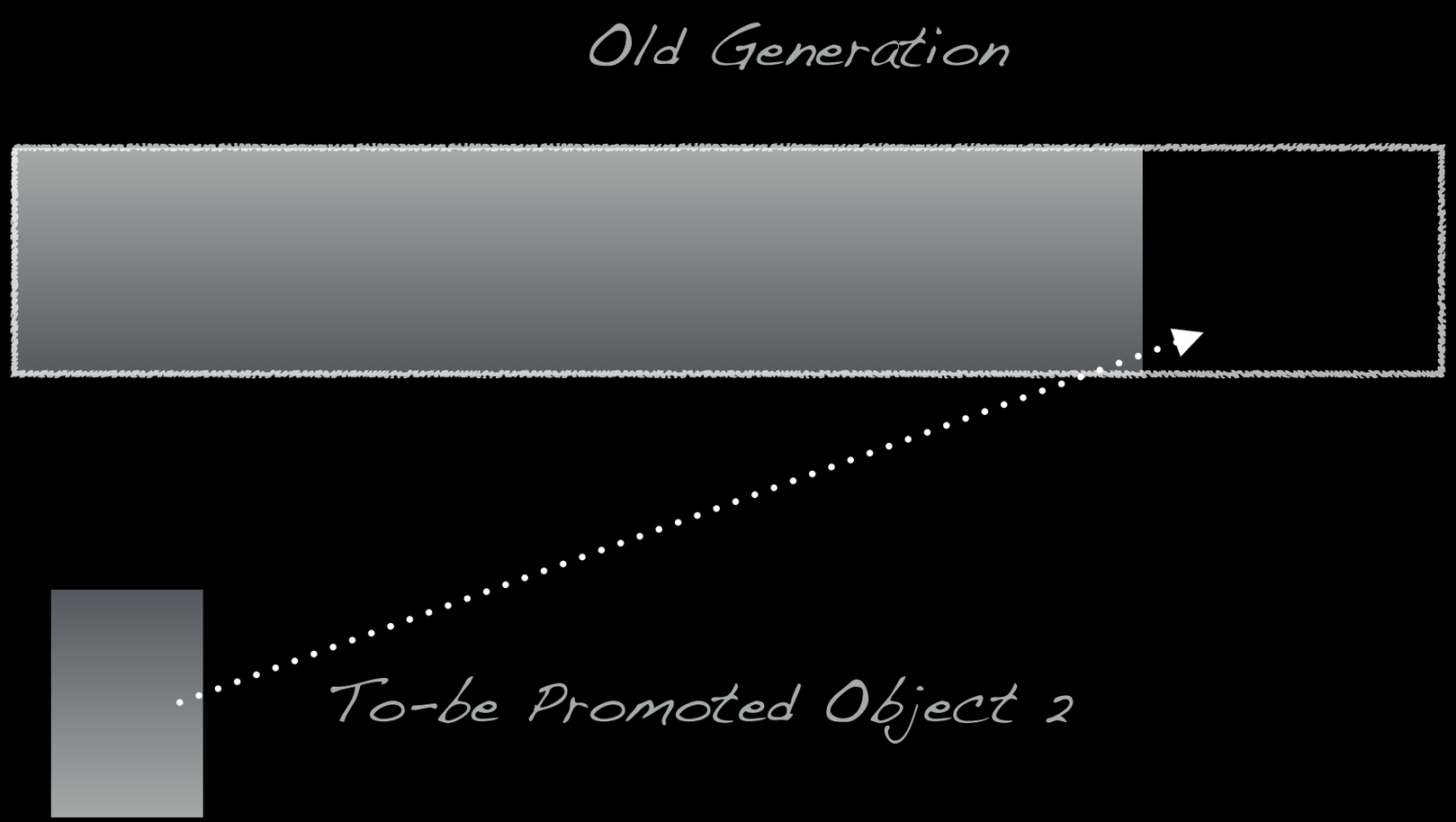
©2015 CodeKaram

The Throughput Collector - Contiguous Old Generation



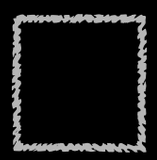
©2015 CodeKaram

The Throughput Collector - Contiguous Old Generation



©2015 CodeKaram

The Throughput Collector - Contiguous Old Generation

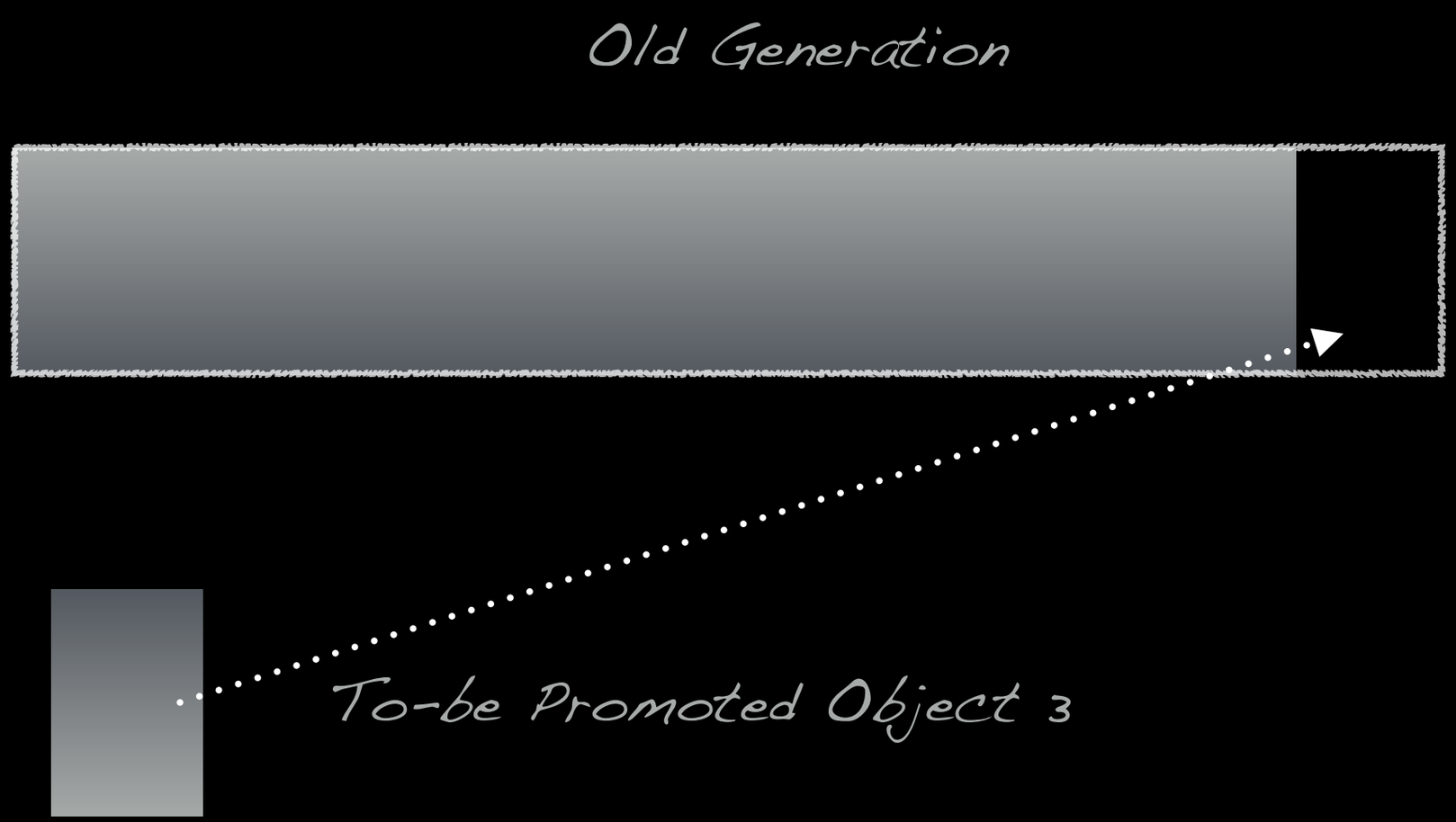


Free Space

Occupied Space

©2015 CodeKaram

The Throughput Collector - Contiguous Old Generation



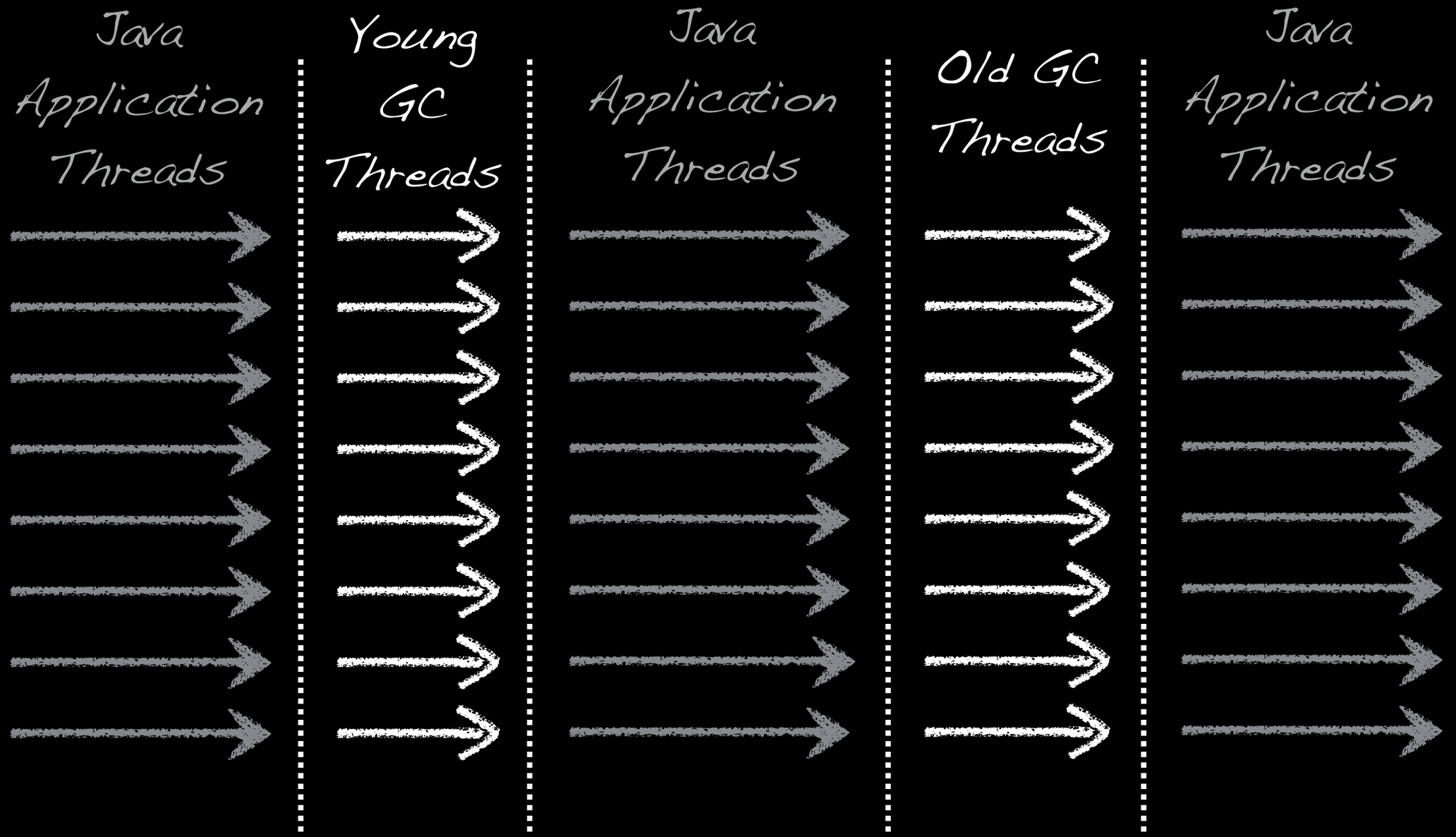
©2015 CodeKaram

The Throughput Collector - Contiguous Old Generation

- When the old generation is full, i.e. when the old generation can't accept any promotions, it is time to start a full compaction collection.
- The ParallelOld collector utilized parallel stop-the-world GC threads to help compact the entire heap.
- The ParallelOld collector achieves compaction via copying and moving live data.
- After the end of the full collection, only the compacted live data set of the java application occupies the old generation. The rest of the space in the old generation is free for future promotions.

©2015 CodeKaram

The Throughput Collector



©2015 CodeKaram

The CMS Collector

- Mostly Concurrent Mark & Sweep collector is geared towards latency sensitive applications.
- The young collections are pretty similar to what you would observe with ParallelOld GC's young collection
- The main difference is in the old generation collection.
  - An occupancy threshold determines when the mostly concurrent mark and sweep cycle can start.
  - The sweep cycle just reclaims the dead objects and updates a list of free spaces. Thus, the CMS collector is not a compacting collector.
    - This can lead to fragmentation which can lead to promotion failures which eventually lead to a fallback single-threaded fully compacting collection.

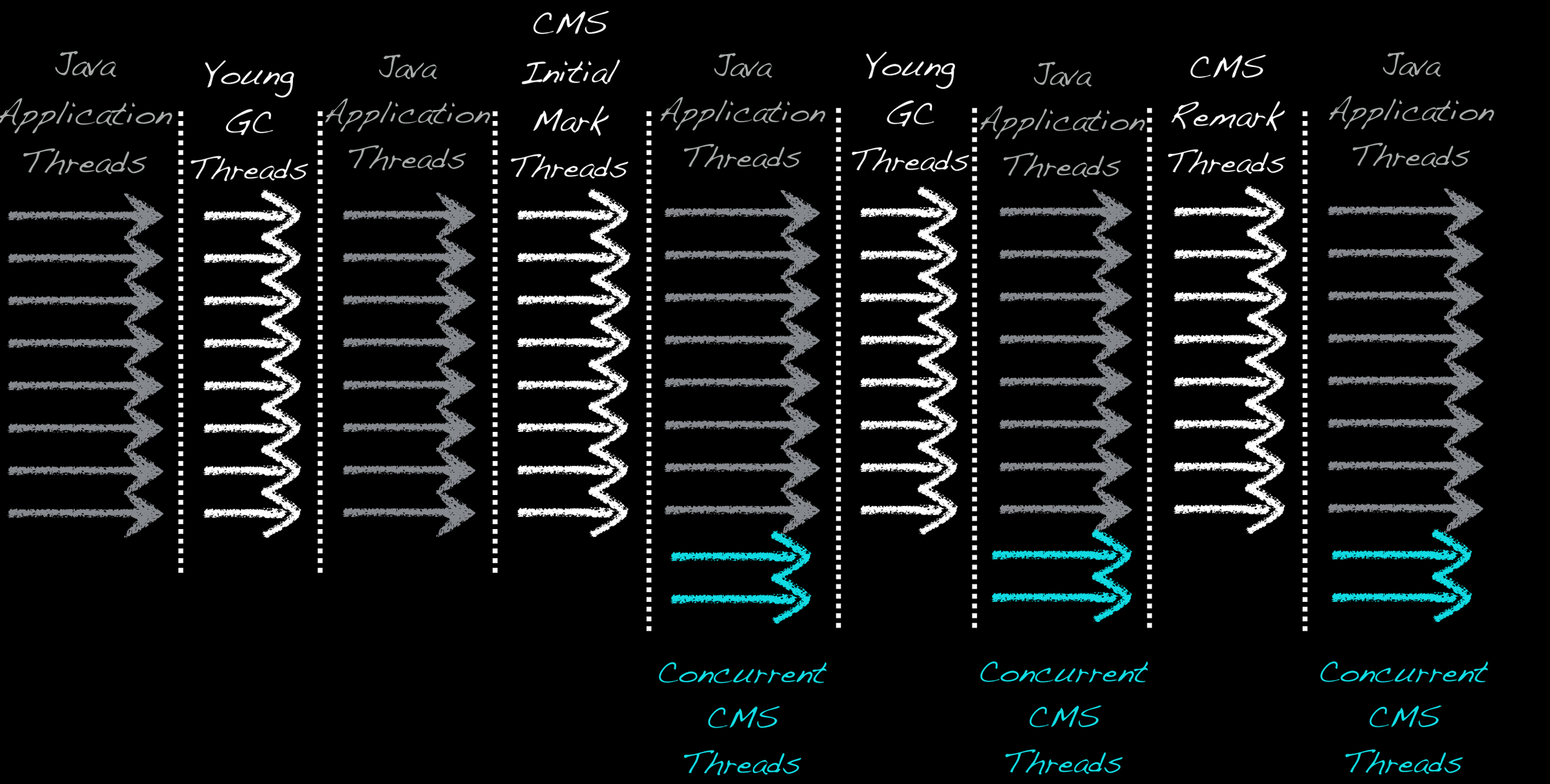
©2015 CodeKaram

The CMS Collector

- The CMS collector has multiple stop-the-world and concurrent phases -
  - After the old generation occupied space reaches or crosses the occupancy threshold, a CMS cycle is initiated.
  - Initially, parallel stop-the-world CMS threads help find the GC roots to start a live object graph.
  - Concurrent CMS threads help in marking further live objects (objects that are reachable by the roots).
  - Parallel stop-the-world CMS threads help in making sure that the final changes to the graph are duly noted.
  - Concurrent sweep phase helps in collecting all dead objects and updating the free list to track the reclaimed spaces.

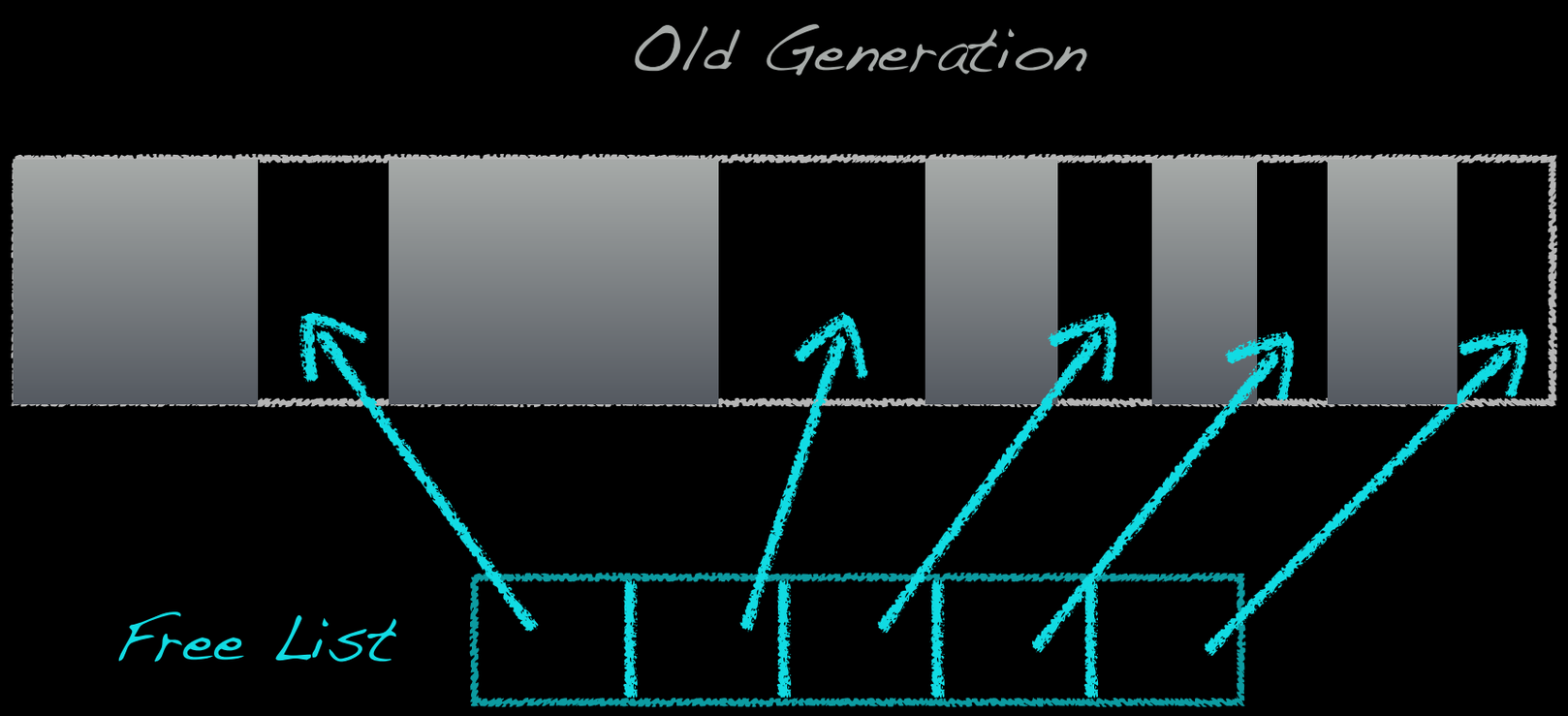
©2015 CodeKaram

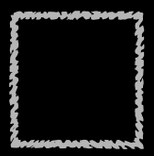
The CMS Collector



©2015 CodeKaram

The CMS Collector - Contiguous Old Generation



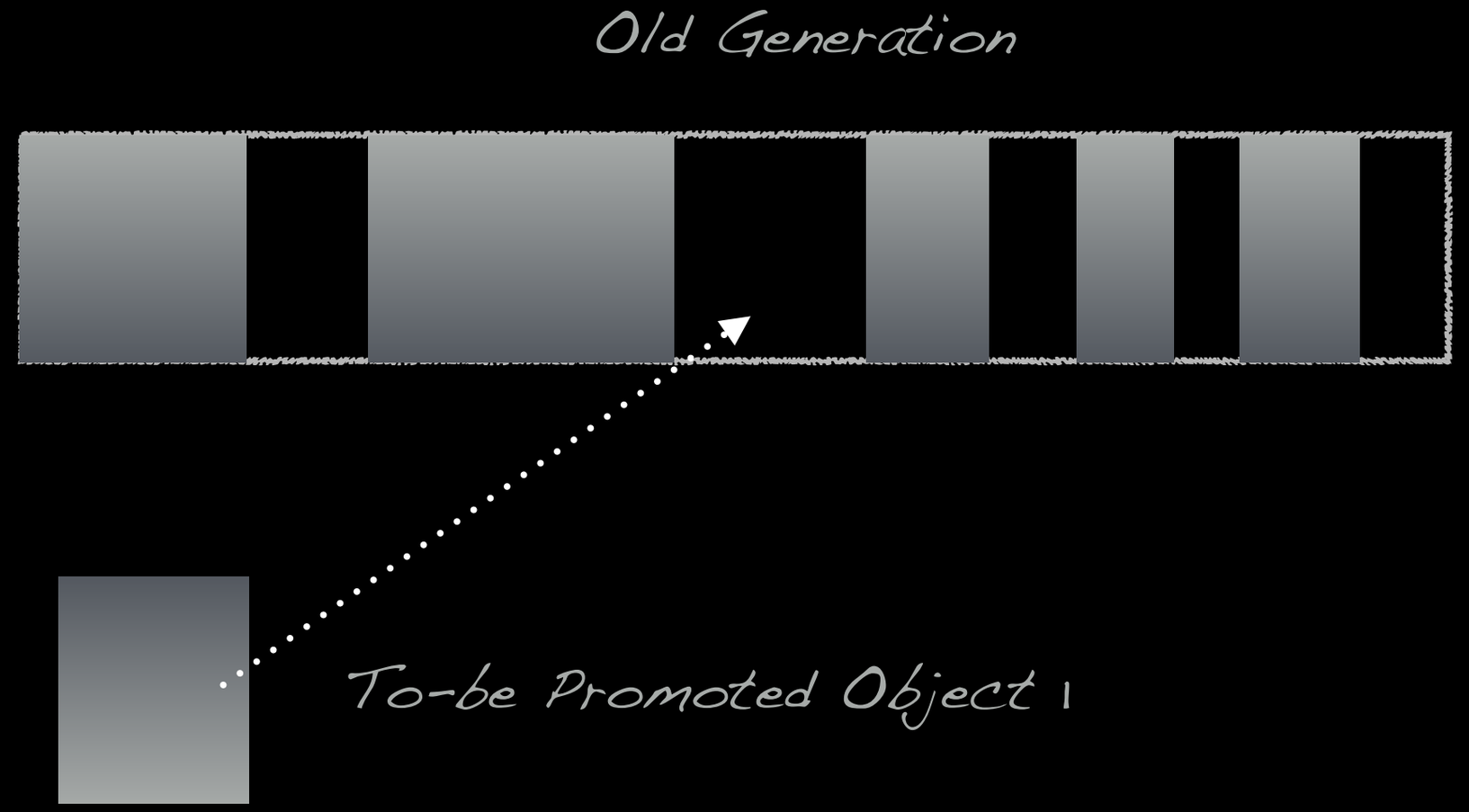


Free Space

Occupied Space

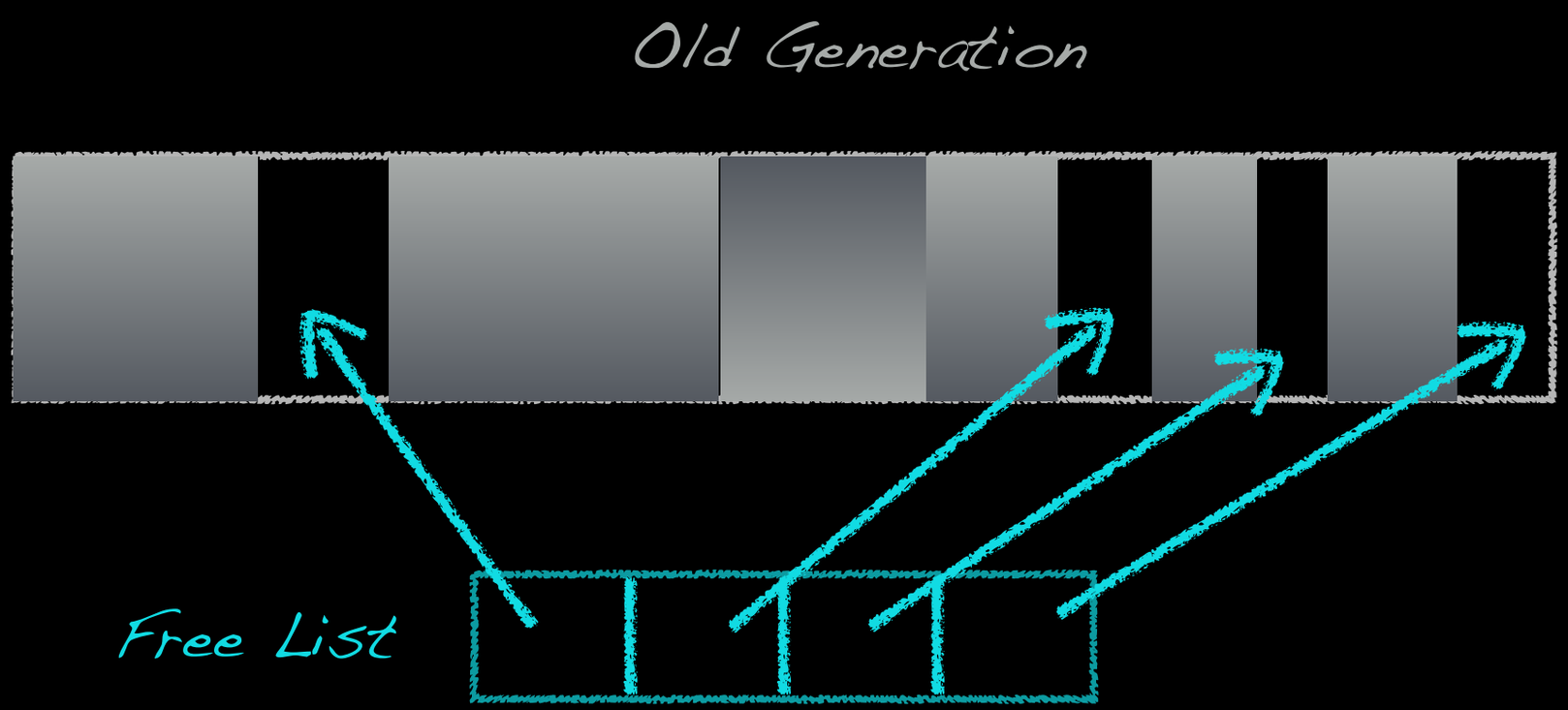
©2015 CodeKaram

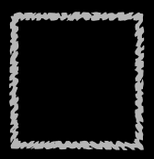
The CMS Collector - Contiguous Old Generation



©2015 CodeKaram

The CMS Collector - Contiguous Old Generation



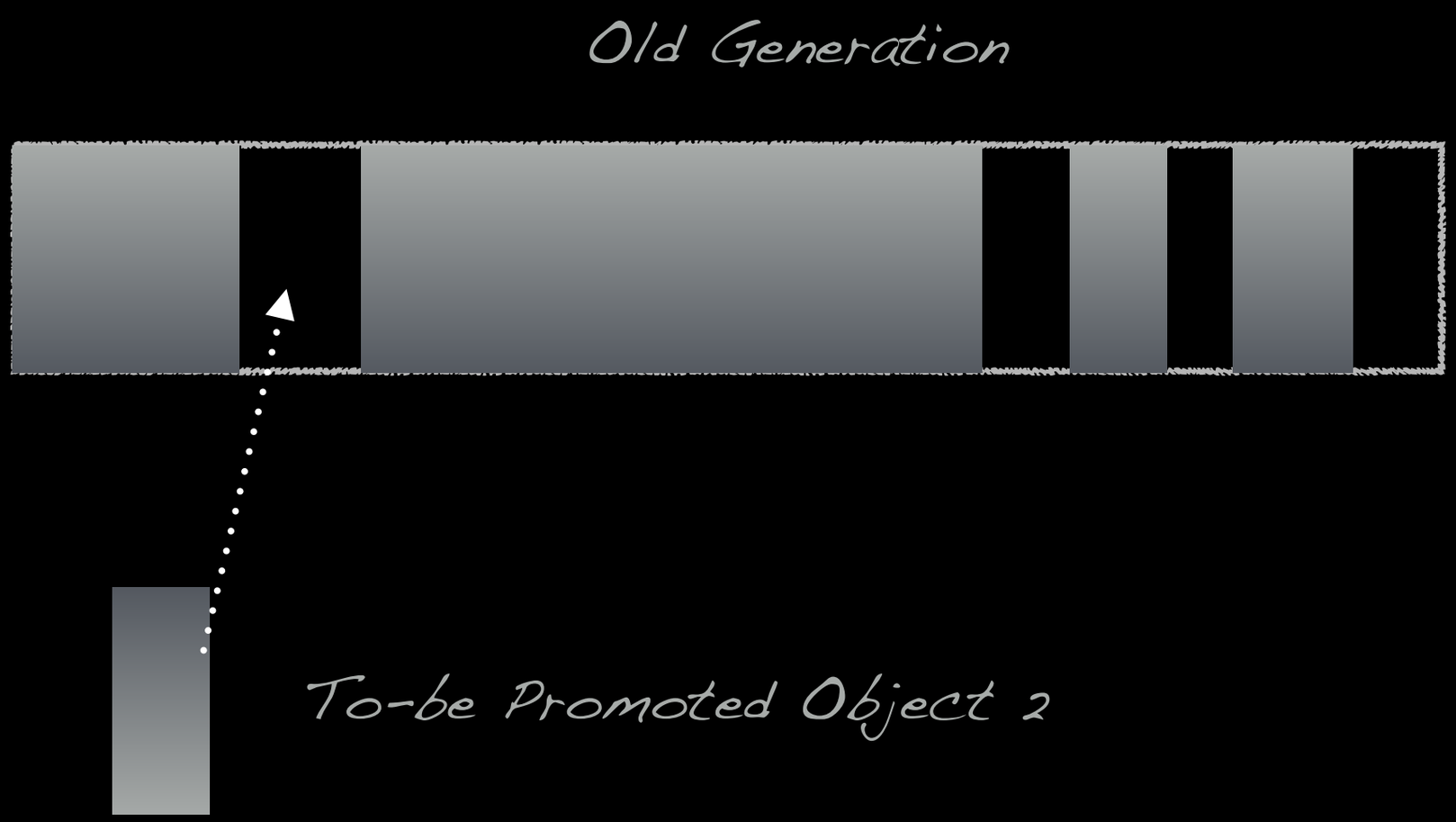


Free Space

Occupied Space

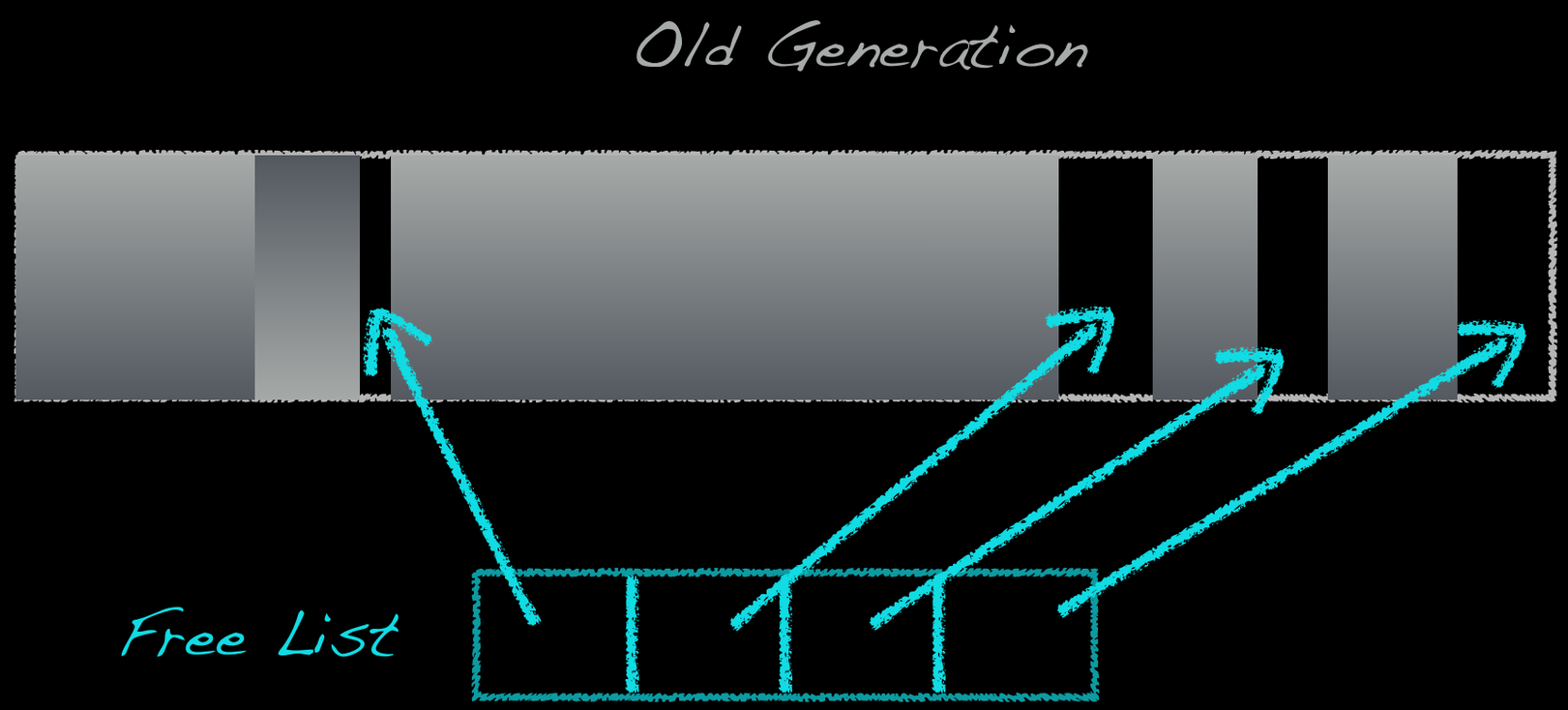
©2015 CodeKaram

The CMS Collector - Contiguous Old Generation



©2015 CodeKaram

The CMS Collector - Contiguous Old Generation

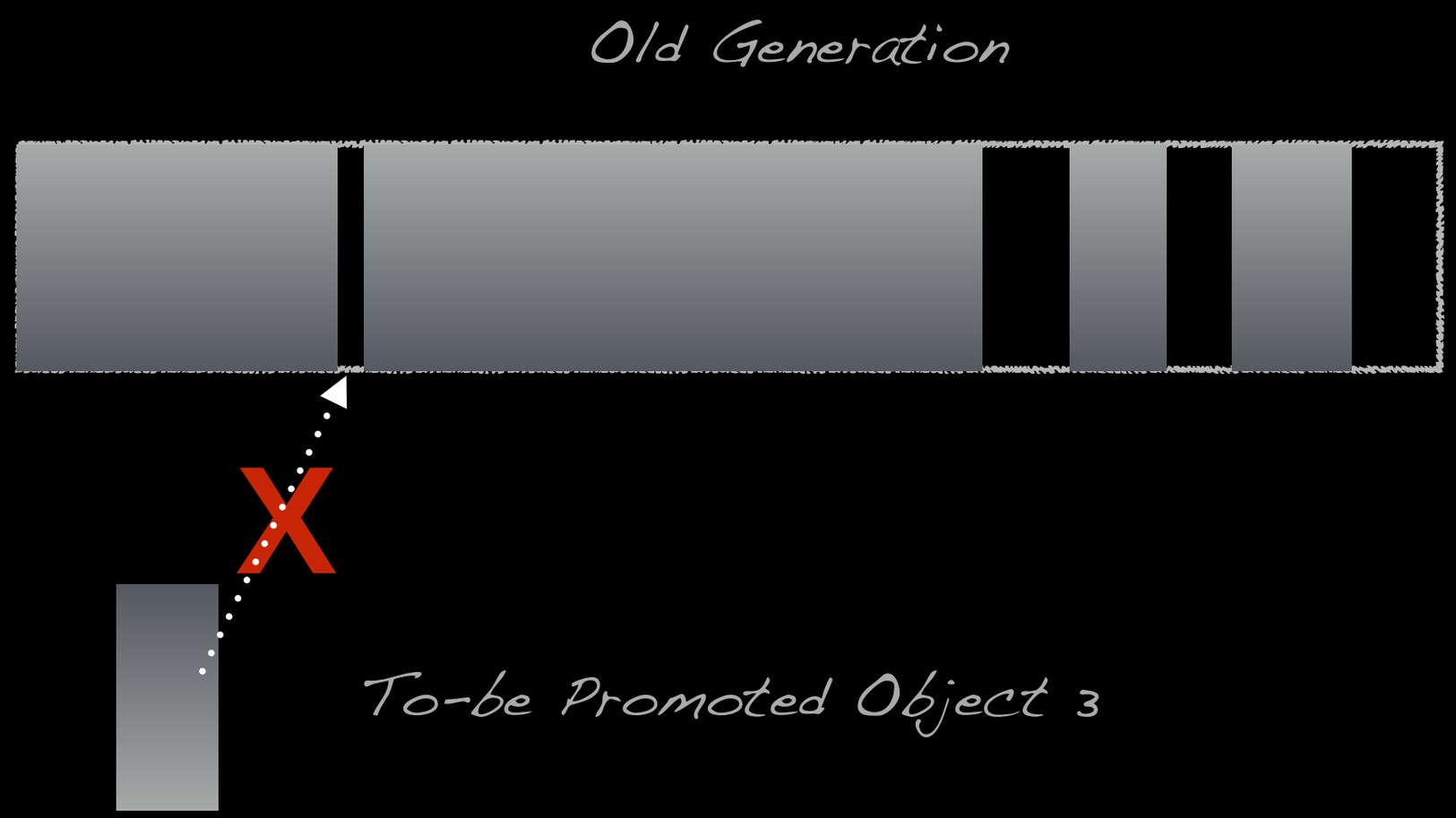


Free Space

Occupied Space

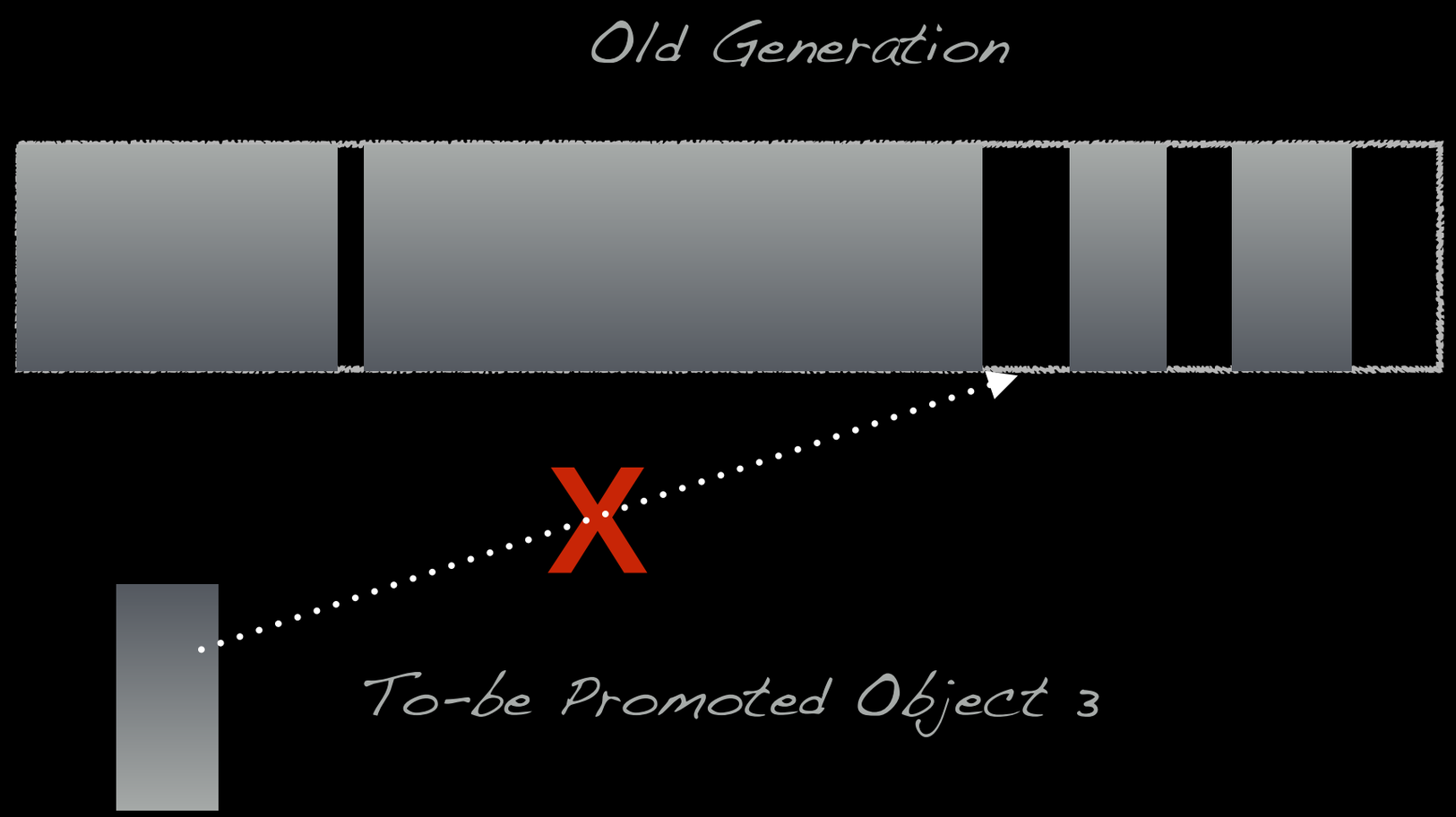
©2015 CodeKaram

The CMS Collector - Contiguous Old Generation



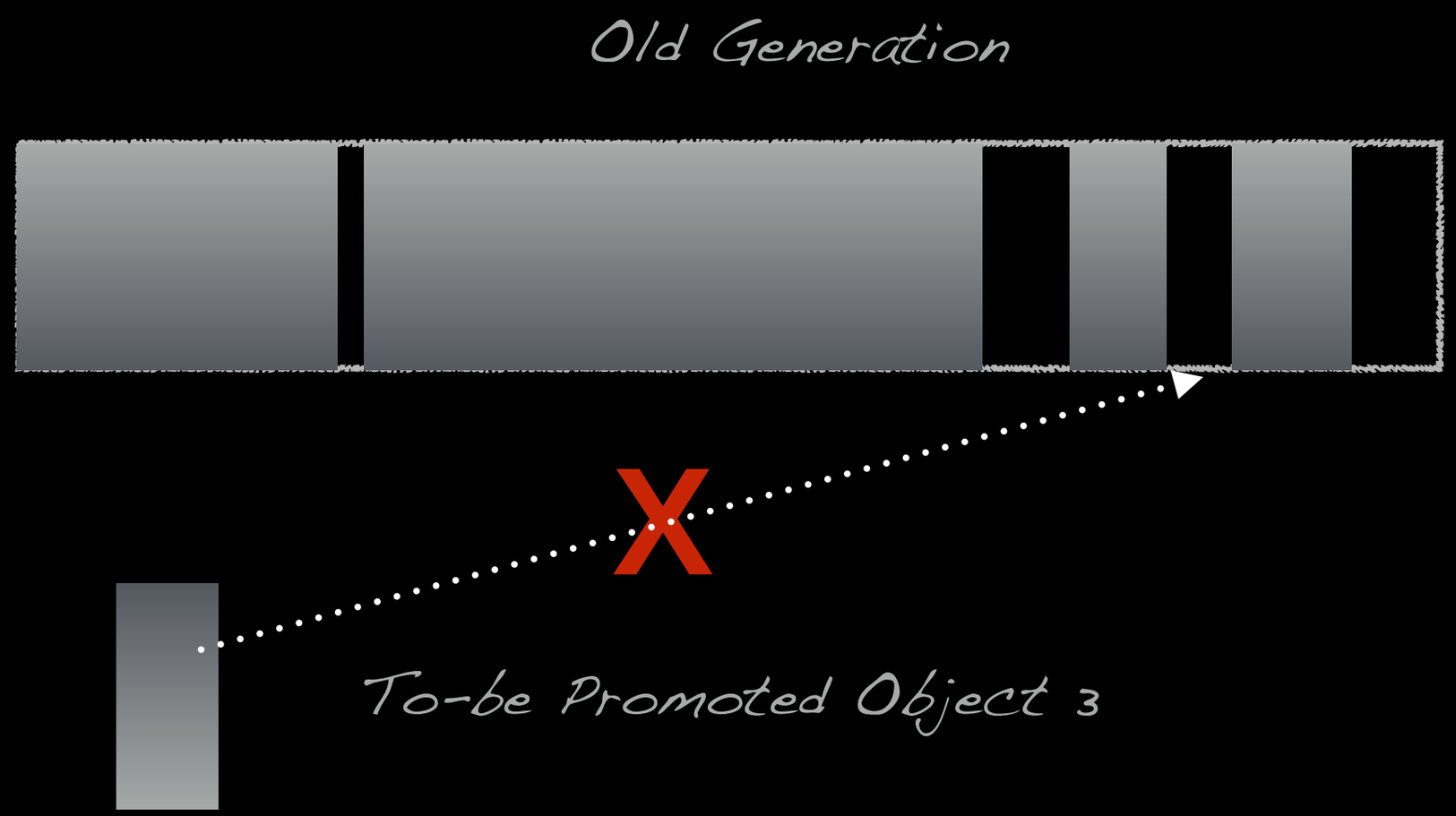
©2015 CodeKaram

The CMS Collector - Contiguous Old Generation



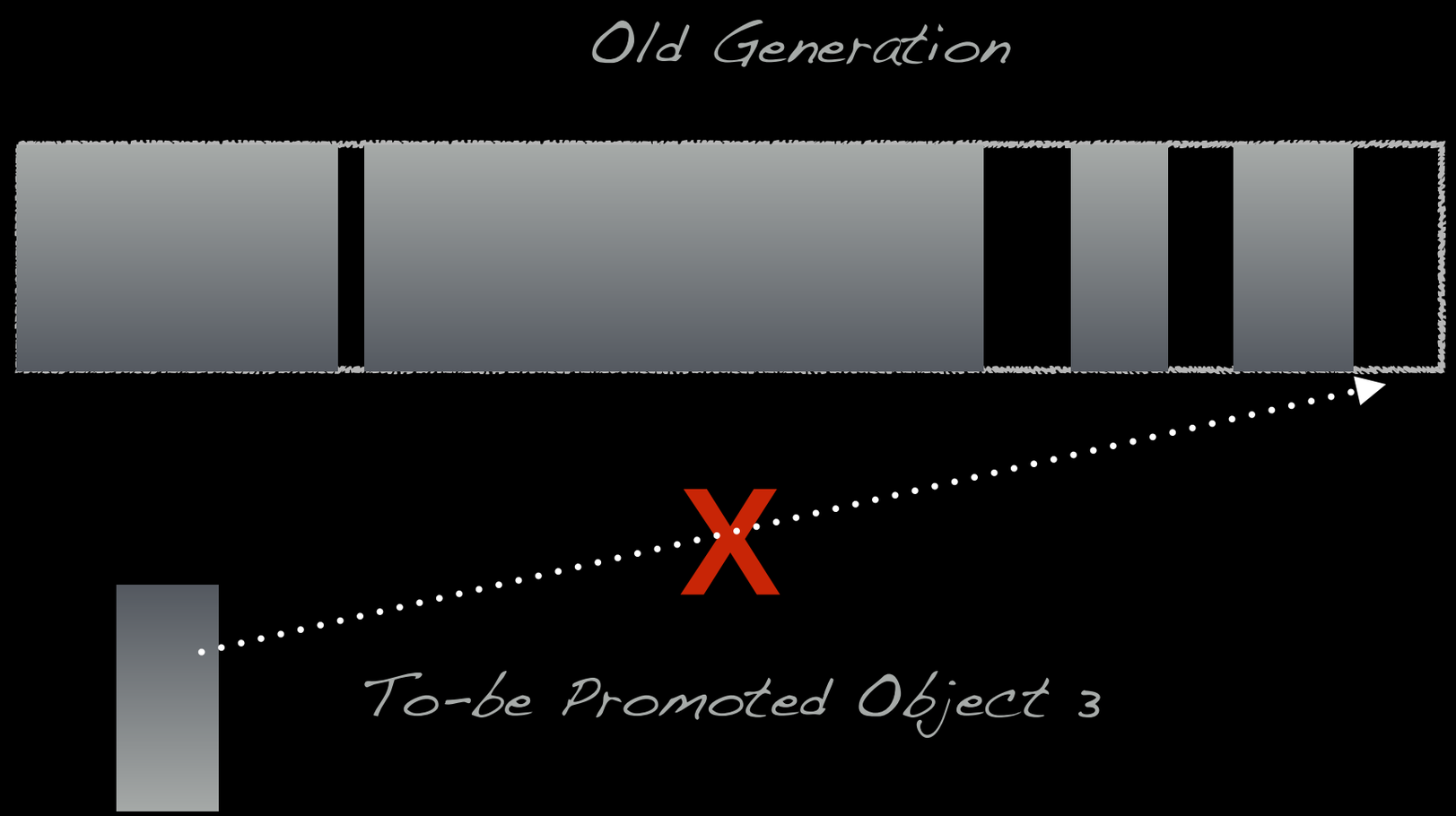
©2015 CodeKaram

The CMS Collector - Contiguous Old Generation



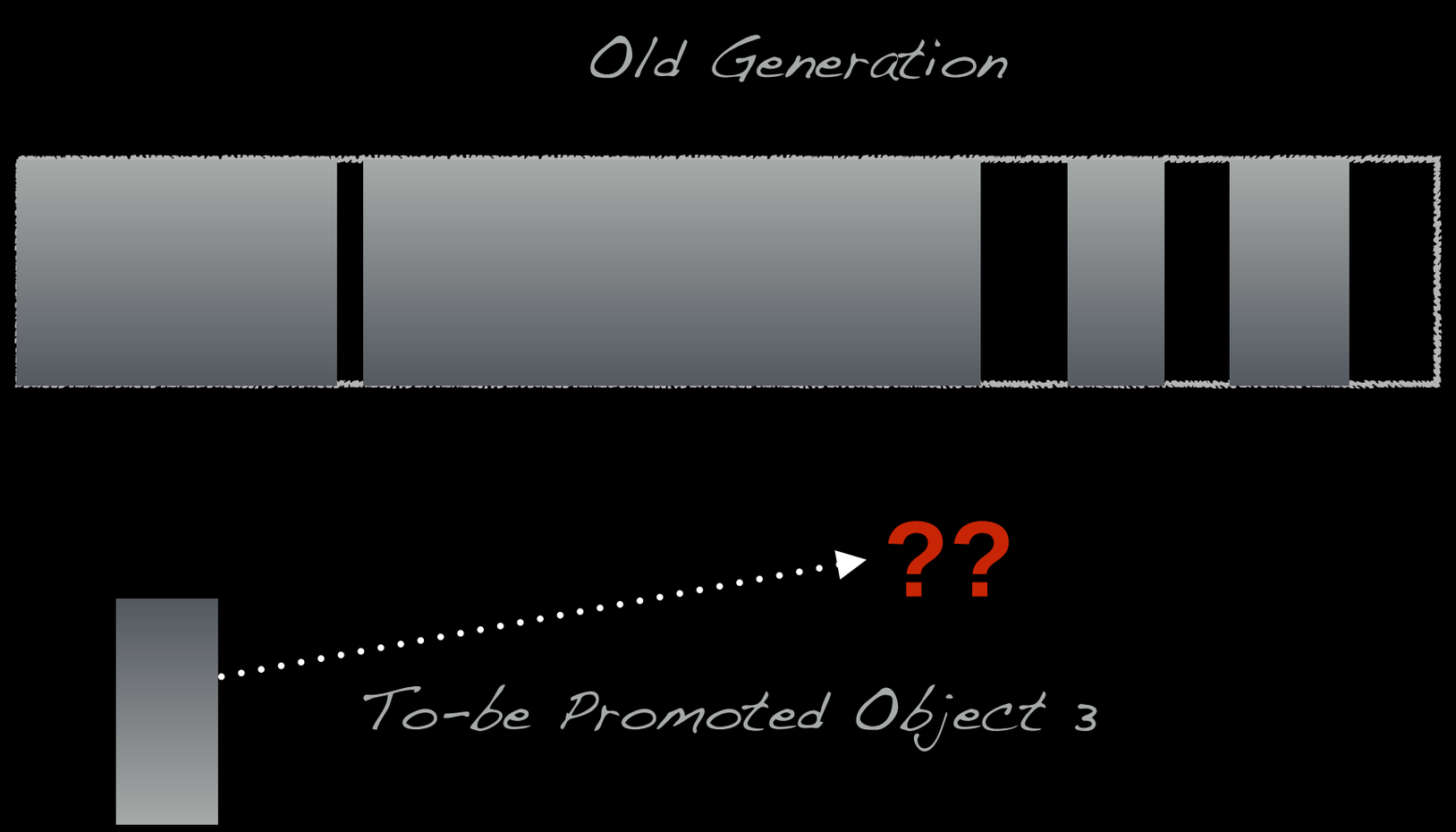
©2015 CodeKaram

The CMS Collector - Contiguous Old Generation



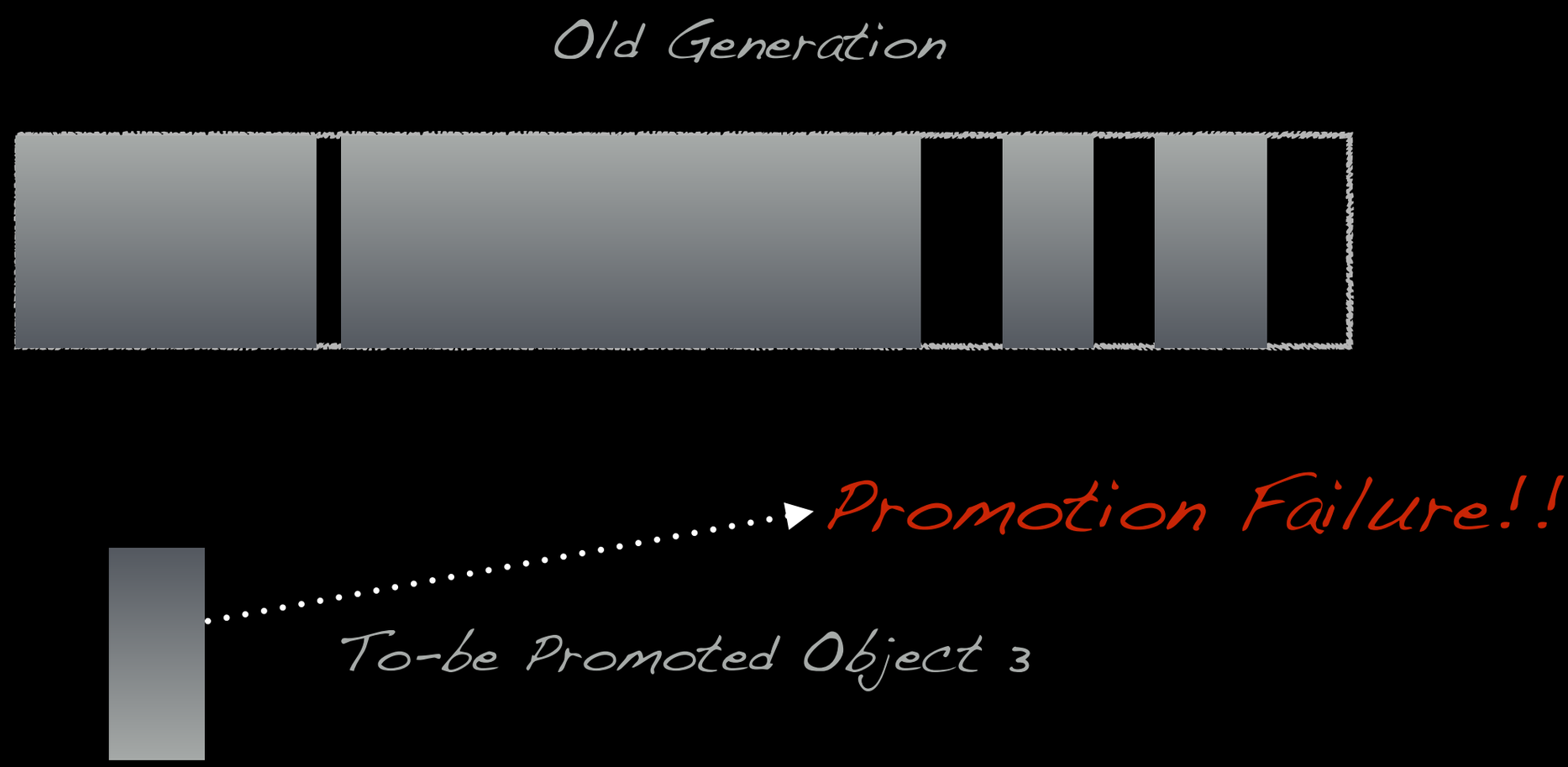
©2015 CodeKaram

The CMS Collector - Contiguous Old Generation



©2015 CodeKaram

The CMS Collector - Contiguous Old Generation



©2015 CodeKaram

The CMS Collector - Concurrent Mode Failures

- When CMS just can't keep up with the promotion rate
- Your old generation is getting filled before a concurrent cycle can free up space and complete.
- Causes - marking threshold is too high, heap too small, or high application mutation rate

©2015 CodeKaram

The Garbage First Collector

- Garbage First (G1) is the newest collector in OpenJDK HotSpot.
- G1 GC is supposed to be a long term replacement of CMS GC.
- G1 GC introduces the concept of regionalized heap.
- The idea is to set a pause time goal which will be used as a hint by the ergonomics and prediction logic and G1 GC will try to adjust its generations and number of regions per collections so as to best meet its pause time goal.
- G1 has multi-staged concurrent marking phase as well as stop the world collections for young and old generations

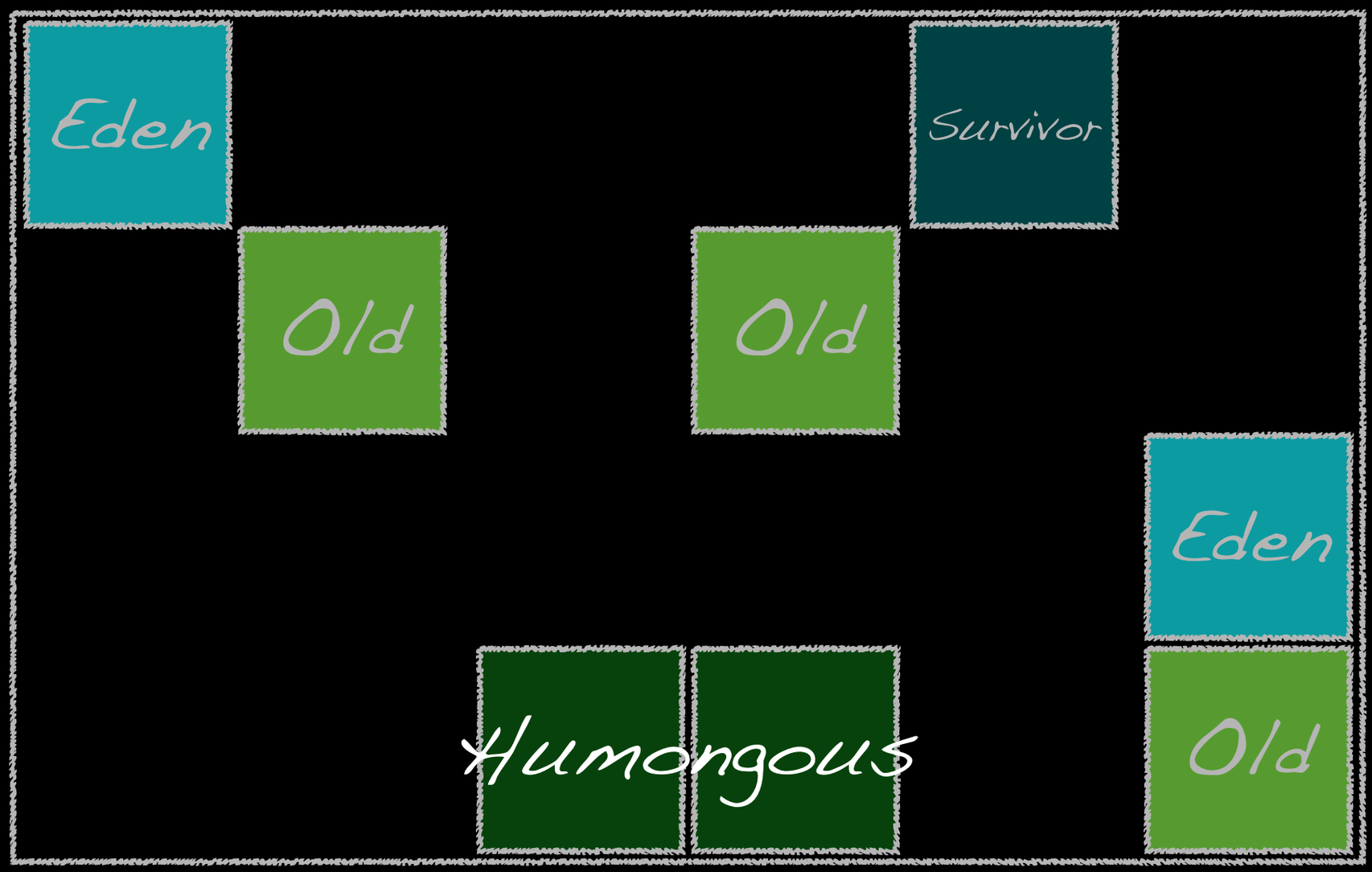
©2015 CodeKaram

The Garbage First Collector

- Throughput is a secondary consideration for G1. Hence the throughput goal is to have about 10% of the total time be spent in GC activities.
  - G1 GC aims to provide more predictable pause times.
- The G1 GC regionalized heap ensures that the generations don't have to be contiguous (as they were in ParallelOld and CMS GCs).
- With regions, G1 just needs to know the limits for each generation and the pause time goal.
  - G1 will add regions as needed from the list of free regions.
  - Occupied regions can be eden, survivor, old and humongous *(more on this later)*.

©2015 CodeKaram

The Garbage First Collector - Regionalized Heap



©2015 CodeKaram

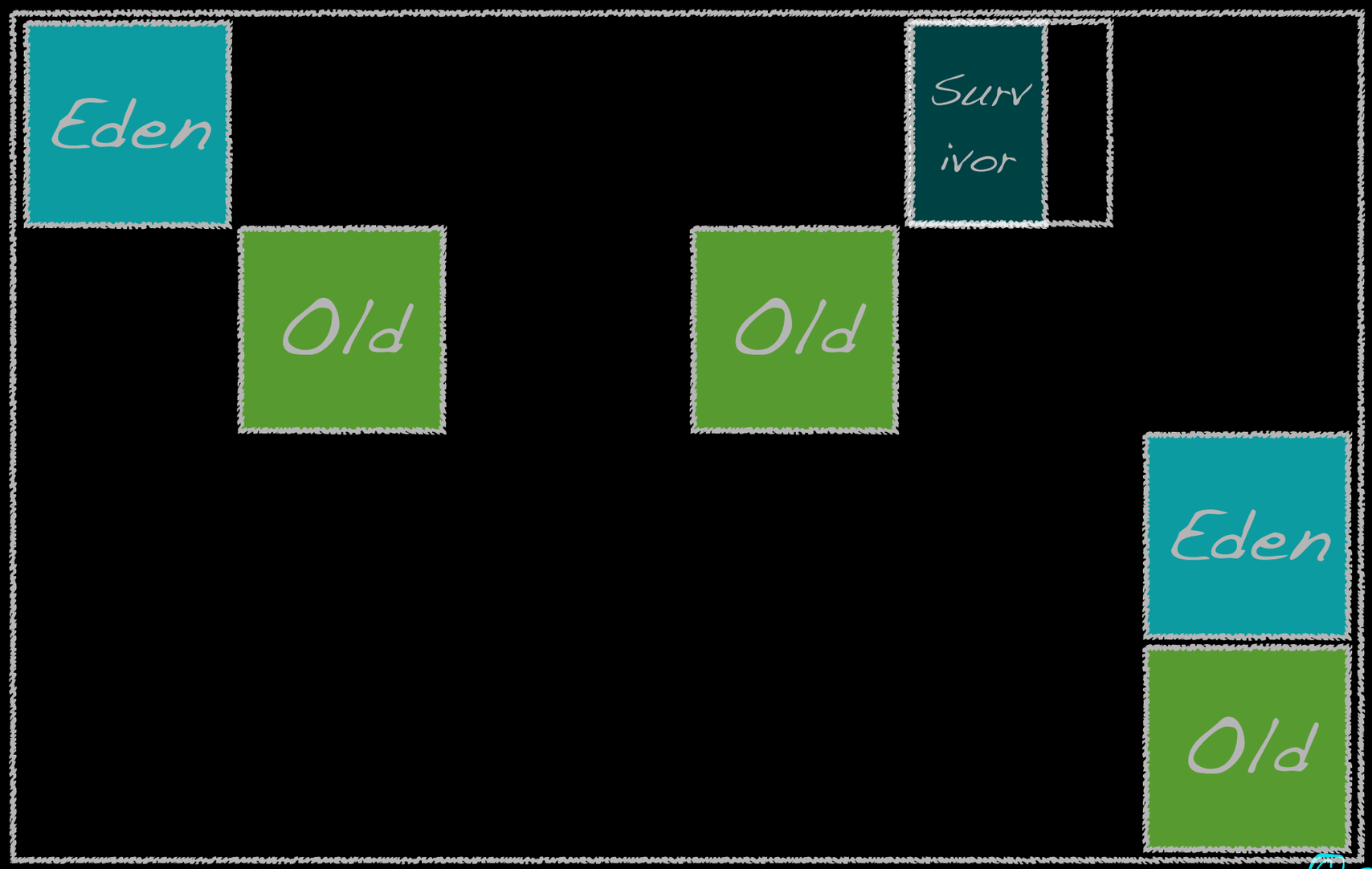
The Garbage First Collector - The Young Generation

- For G1 GC, the young generation consists of eden regions and survivor regions.
  - A region is a unit of collection
  - The region size is determined at JVM startup (and can be changed if needed)
- Most allocations are fast path allocations into the eden regions.
- The survivor regions help in aging the objects which are eventually promoted into the old generation after the age threshold is crossed
- The young generation is resized after every collection based on the time it took the current collection, the pause time goal and other factors considered by the prediction logic.

©2015 CodeKaram

The Garbage First Collector

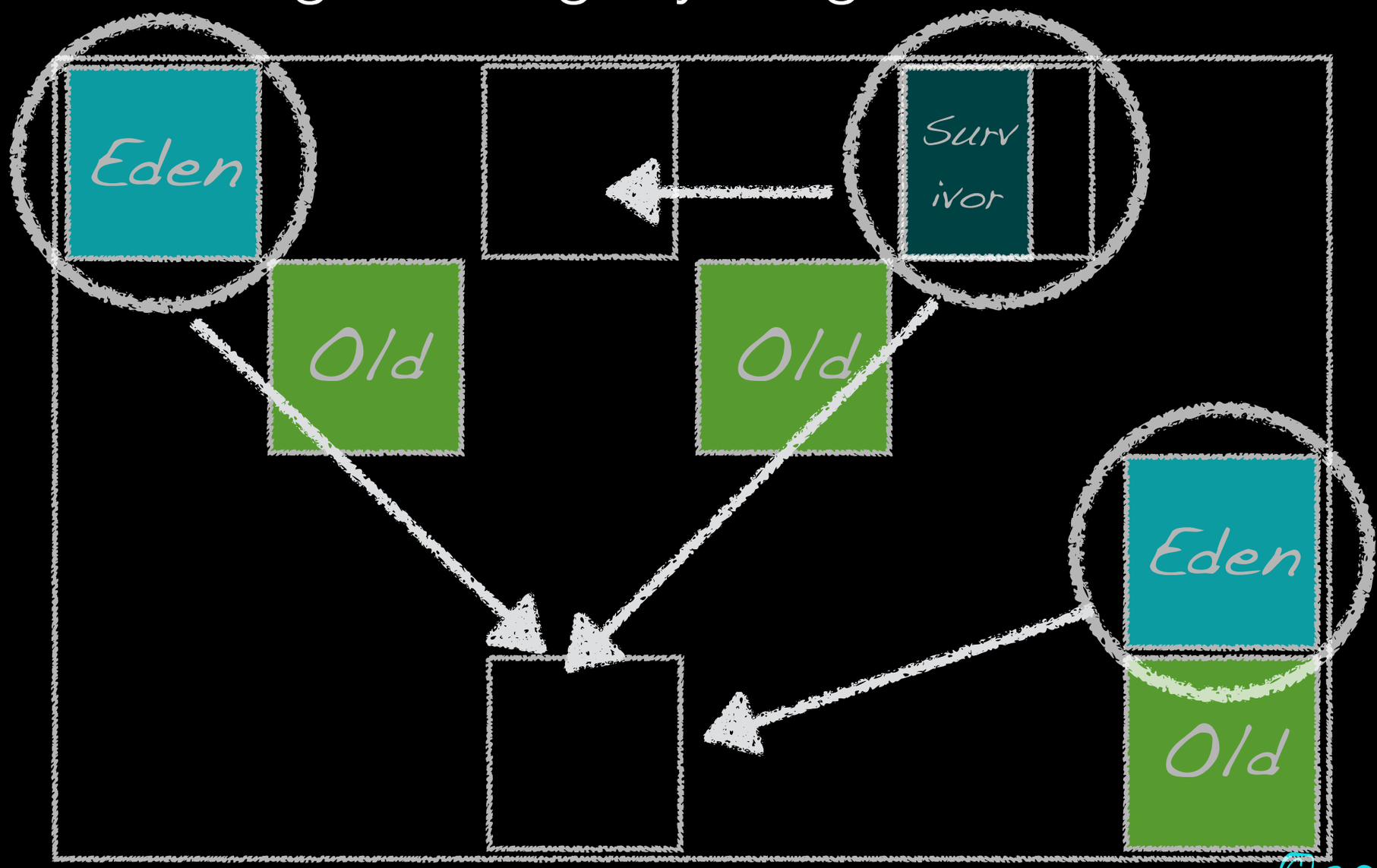
E.g.: Current heap configuration -



©2015 CodeKaram

The Garbage First Collector

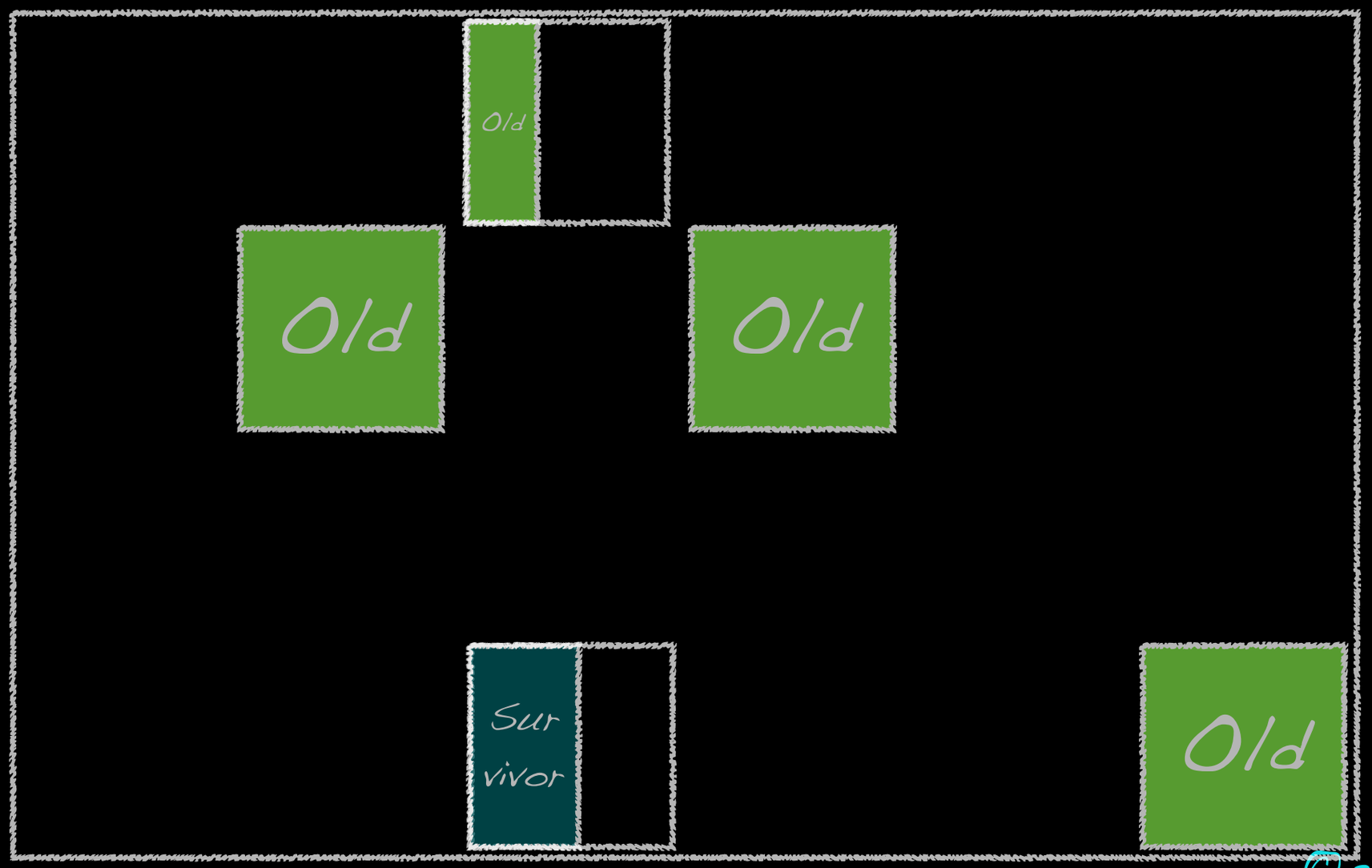
E.g.: During a young collection -



©2015 CodeKaram

The Garbage First Collector

E.g.: After a young collection -



©2015 CodeKaram

The Garbage First Collector - The Old Generation

- For G1 GC, the old generation collection consists of all of the young regions and a few candidate old regions. Such a collection is called a mixed collection.
    - The GC ergonomics selects the candidate old regions based on certain thresholds and calculations that help determine the regions with the most reclaimable space and at the same time giving consideration to the pause time goal.
- Based on the total number of candidate old regions, the pause time goal and other collection related thresholds, you could see more that one mixed collection pause. Thus the old generation is collected incrementally.

©2015 CodeKaram

The Garbage First Collector - The Marking Threshold

- In-order to start a mixed collection, the heap occupancy threshold (also known as the marking threshold) must be crossed.

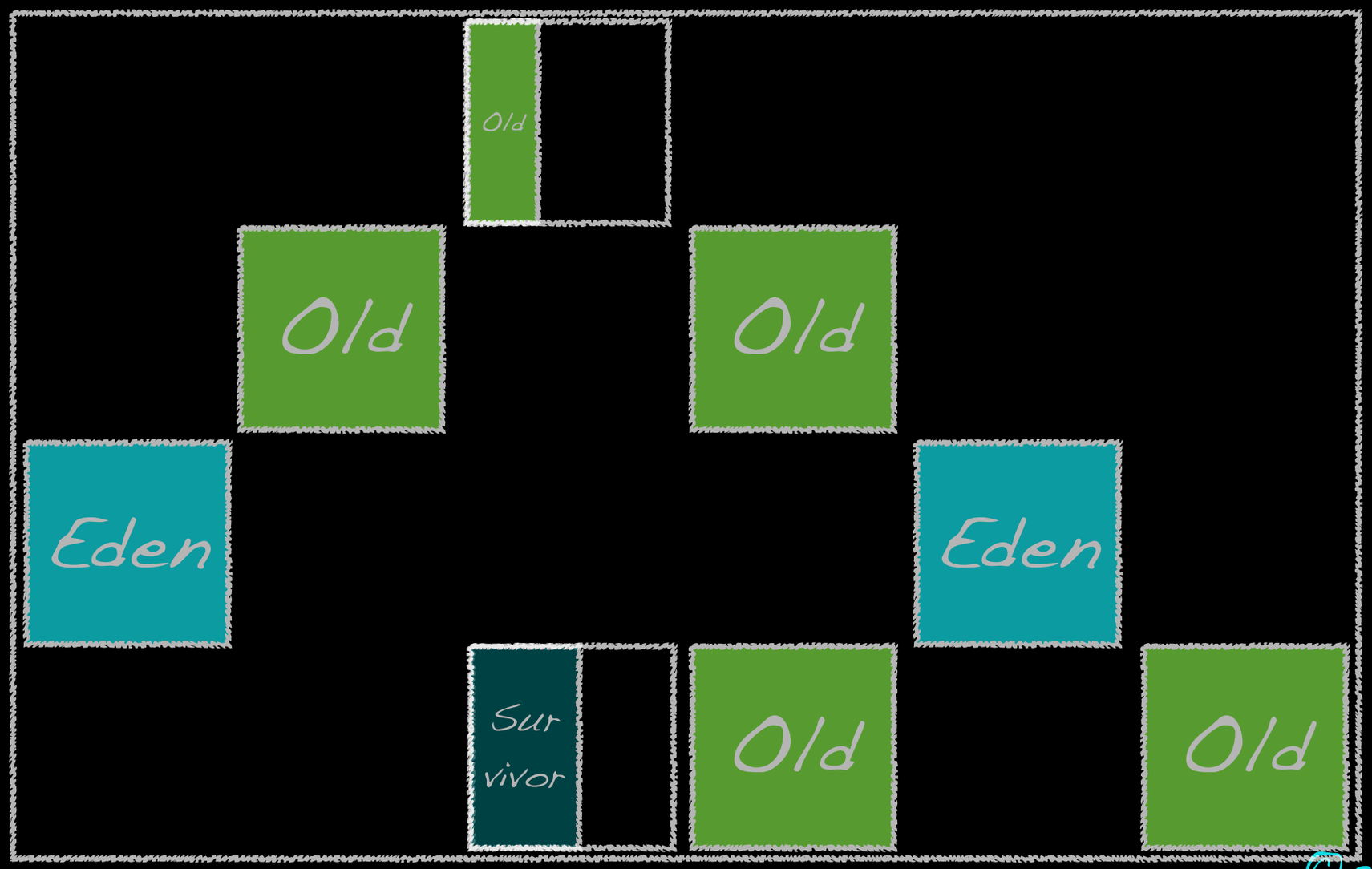
- When the marking threshold is crossed, G1 GC initiates a concurrent marking cycle. The concurrent cycle is mostly concurrent, except for when marking the roots and when remarking to make sure all the mutations are captured appropriately in the object graph.

- During the concurrent marking phase, any garbage-filled regions are reclaimed as a part of the cleanup phase.

©2015 CodeKaram

The Garbage First Collector

E.g.: Current heap configuration -



©2015 CodeKaram

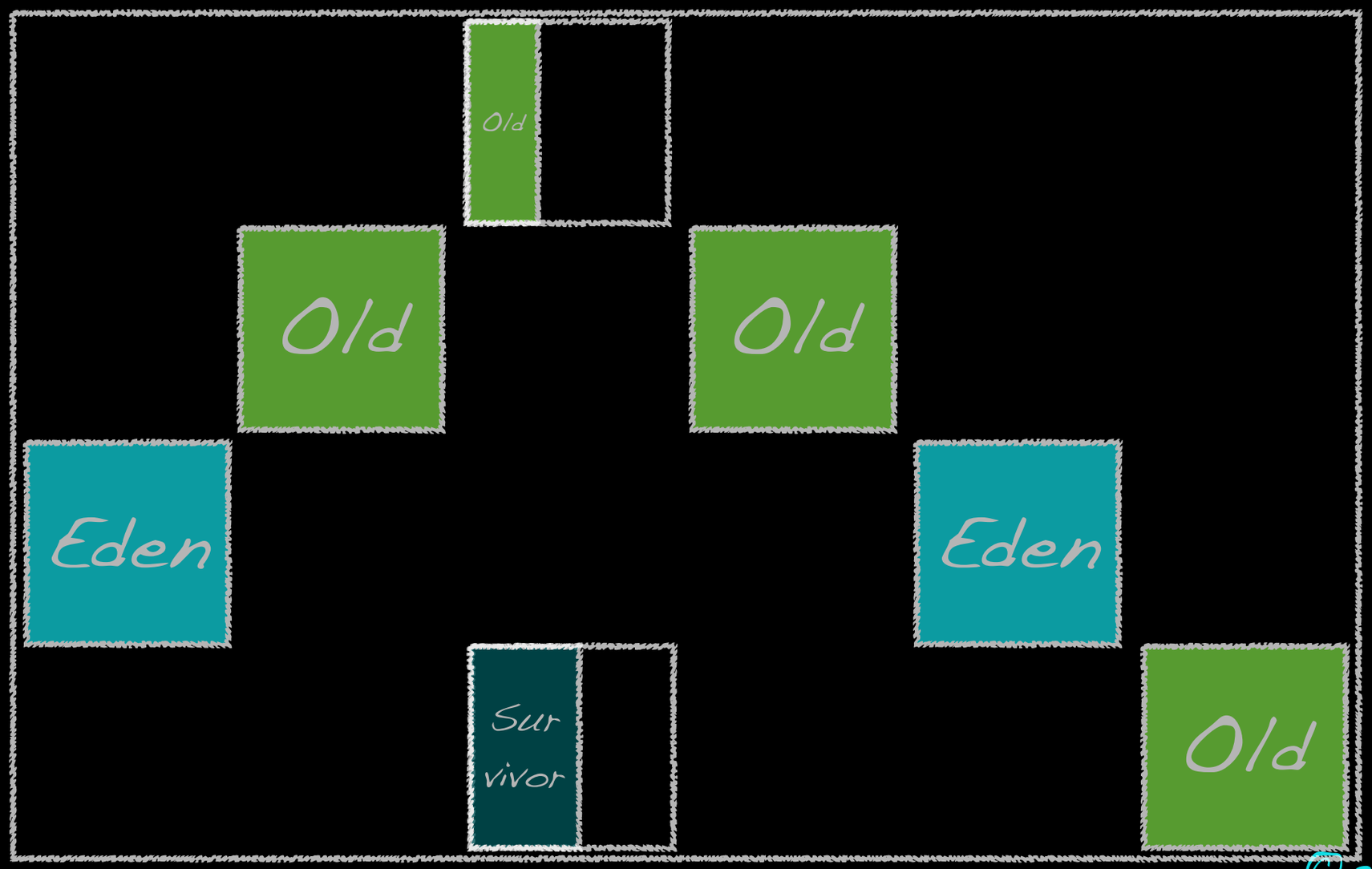
The Garbage First Collector

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

©2015 CodeKaram

The Garbage First Collector

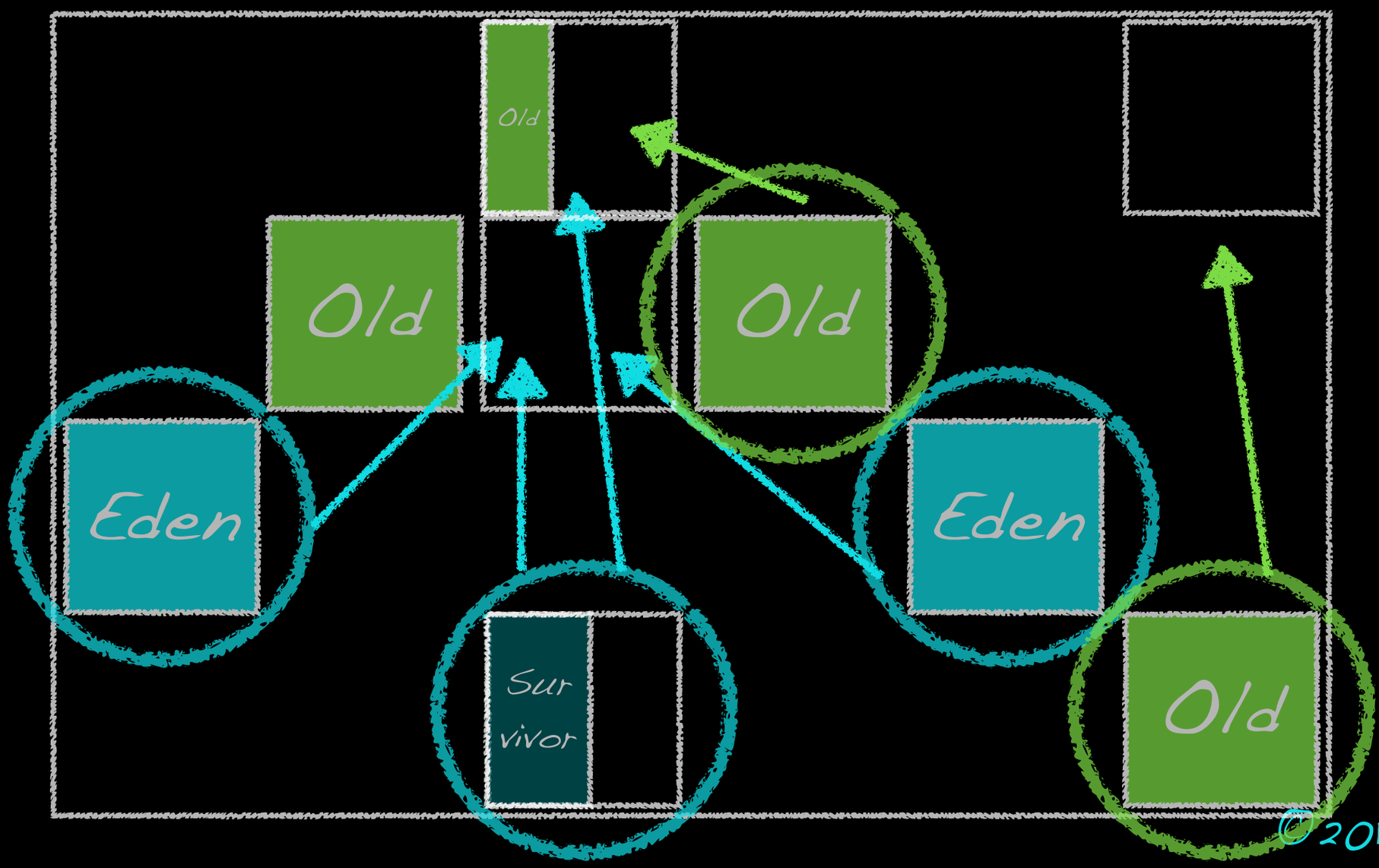
E.g.: Current heap configuration -



©2015 CodeKaram

The Garbage First Collector

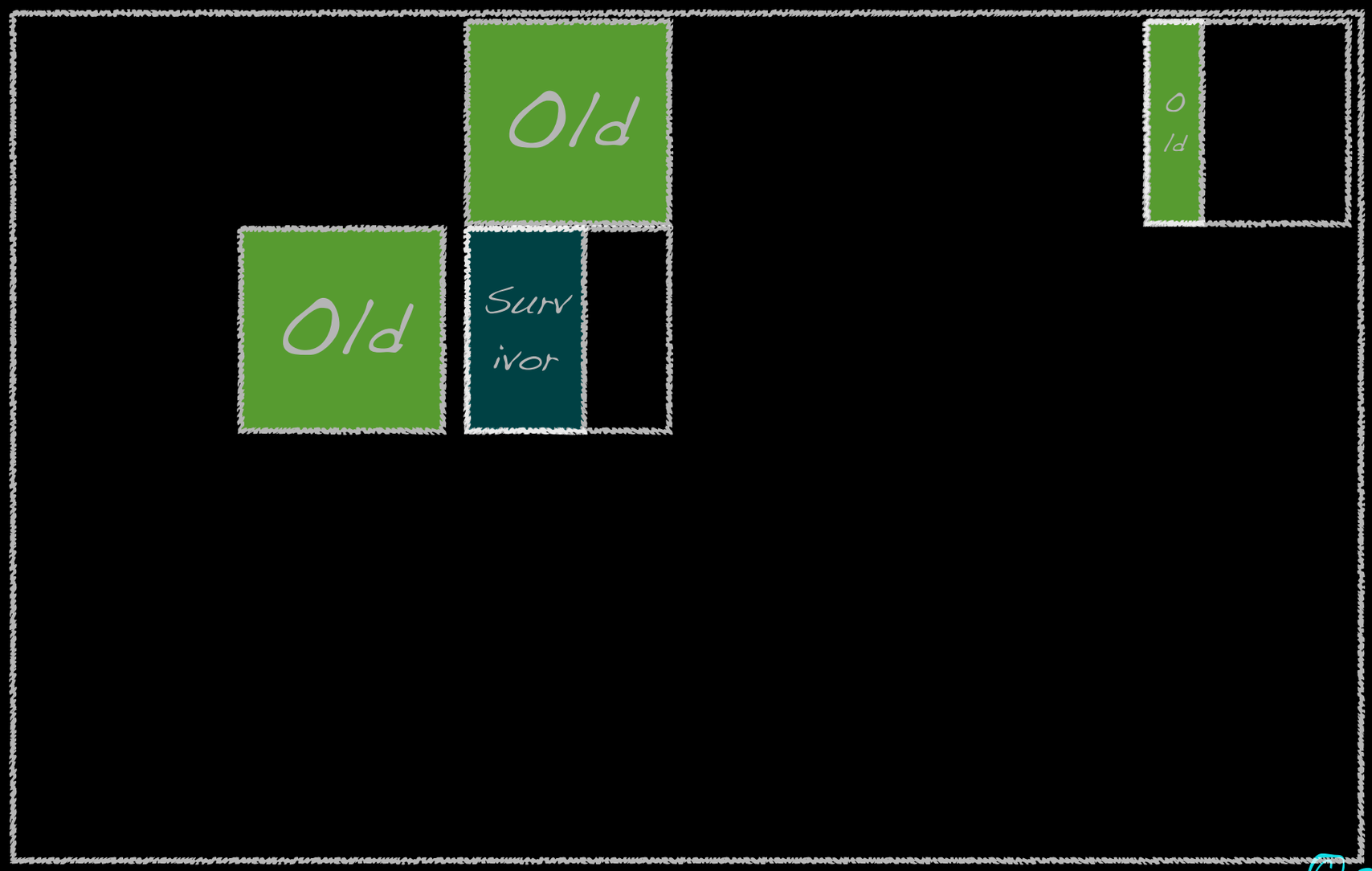
E.g.: During a mixed collection -



©2015 CodeKaram

The Garbage First Collector

E.g.: After a mixed collection -



©2015 CodeKaram

The Garbage First Collector - Humongous Objects

- Objects >= 50% of G1 region size == Humongous Objects
- These objects are allocated directly into the old generation into Humongous Regions
- If a humongous object is bigger than a G1 region, then the humongous regions that contain it have to be contiguous.
- Ideally, humongous objects are few in number and are short lived.
    - Can cause evacuation failures if application allocates a lot of long-lived humongous objects, since humongous regions add to the old generation occupancy.

©2015 CodeKaram

The Garbage First Collector - Evacuation Failures

- Evacuation failures or promotion failures or to-space overflow or exhausted all refer to the same notion.

- 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 will soon follow.

©2015 CodeKaram

The Garbage First Collector - Avoiding Evacuation Failures

- Make sure your command line is not overloaded - Avoid over-tuning!
- Try increasing your heap size.
- Check if humongous allocations are the cause of your problem.
- Adjust the marking threshold
- Try increasing your concurrent threads to help reduce the time for concurrent marking phase.
- If survivor objects are the issue, try increase the G1ReservePercent

©2015 CodeKaram

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

- Young generation is always collected in its entirety.
- All 3 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 ergonomics for young generation collections.

©2015 CodeKaram

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

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.

©2015 CodeKaram

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

All 3 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 size 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.

©2015 CodeKaram

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.

©2015 CodeKaram

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.

©2015 CodeKaram

GC Tunables - 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

©2015 CodeKaram

GC Tunables - 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 long-lived 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%.

©2015 CodeKaram

GC Tunables - 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

©2015 CodeKaram

GC Tunables - 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

©2015 CodeKaram

GC Tunables - The G1 Collector

Goal:

Get the 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

©2015 CodeKaram

GC Tunables - 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.
- Don’t set really aggressive pause time goals - this will increase the GC overhead.
- Spend time taming your mixed GCs - mixed GCs are incremental old generation collections

©2015 CodeKaram

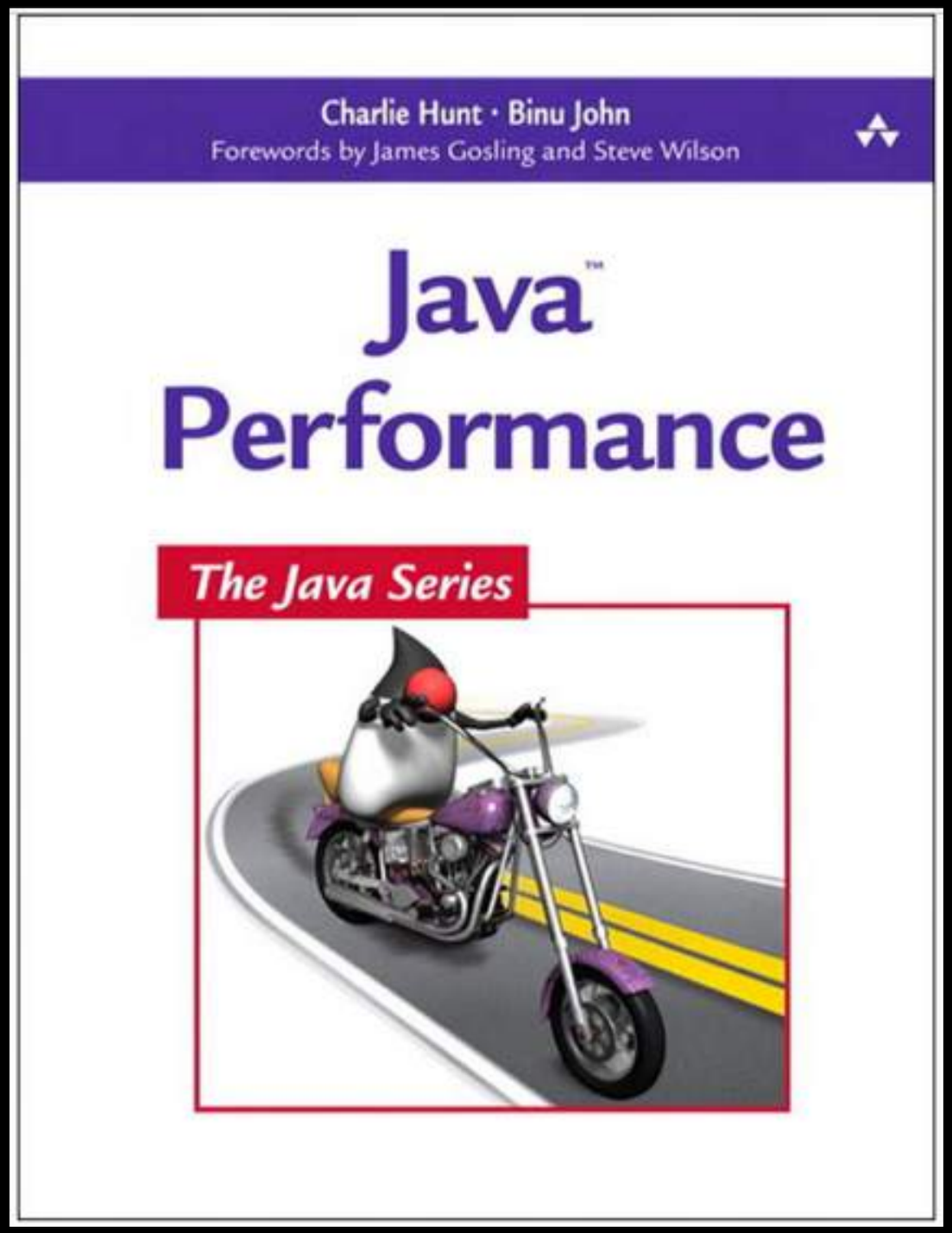
GC Tunables - The G1 Collector

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

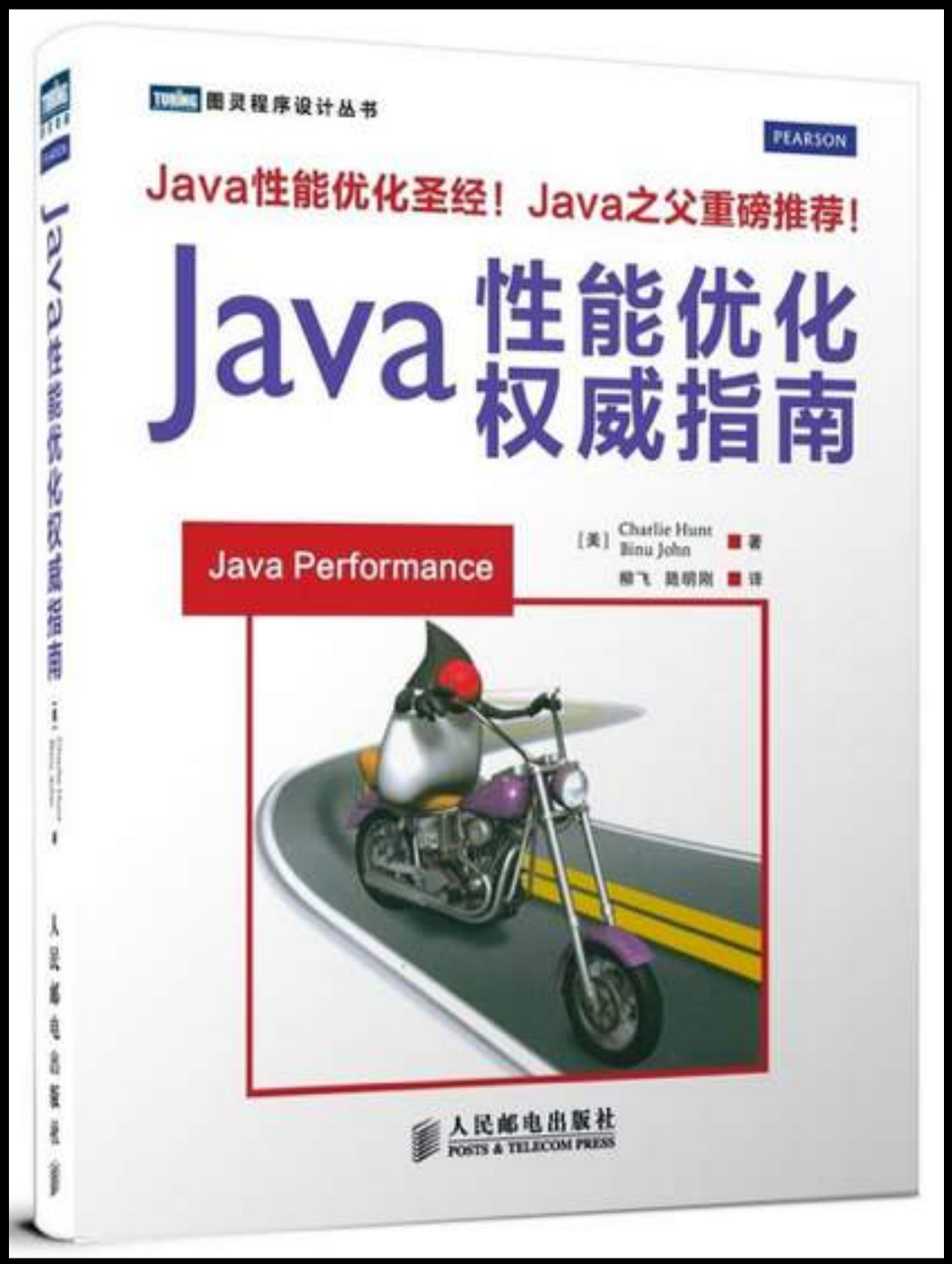
©2015 CodeKaram

Further Reading



©2015 CodeKaram

Further Reading



©2015 CodeKaram

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

©2015 CodeKaram