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

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

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…

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

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

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**

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

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).

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.

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.

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.

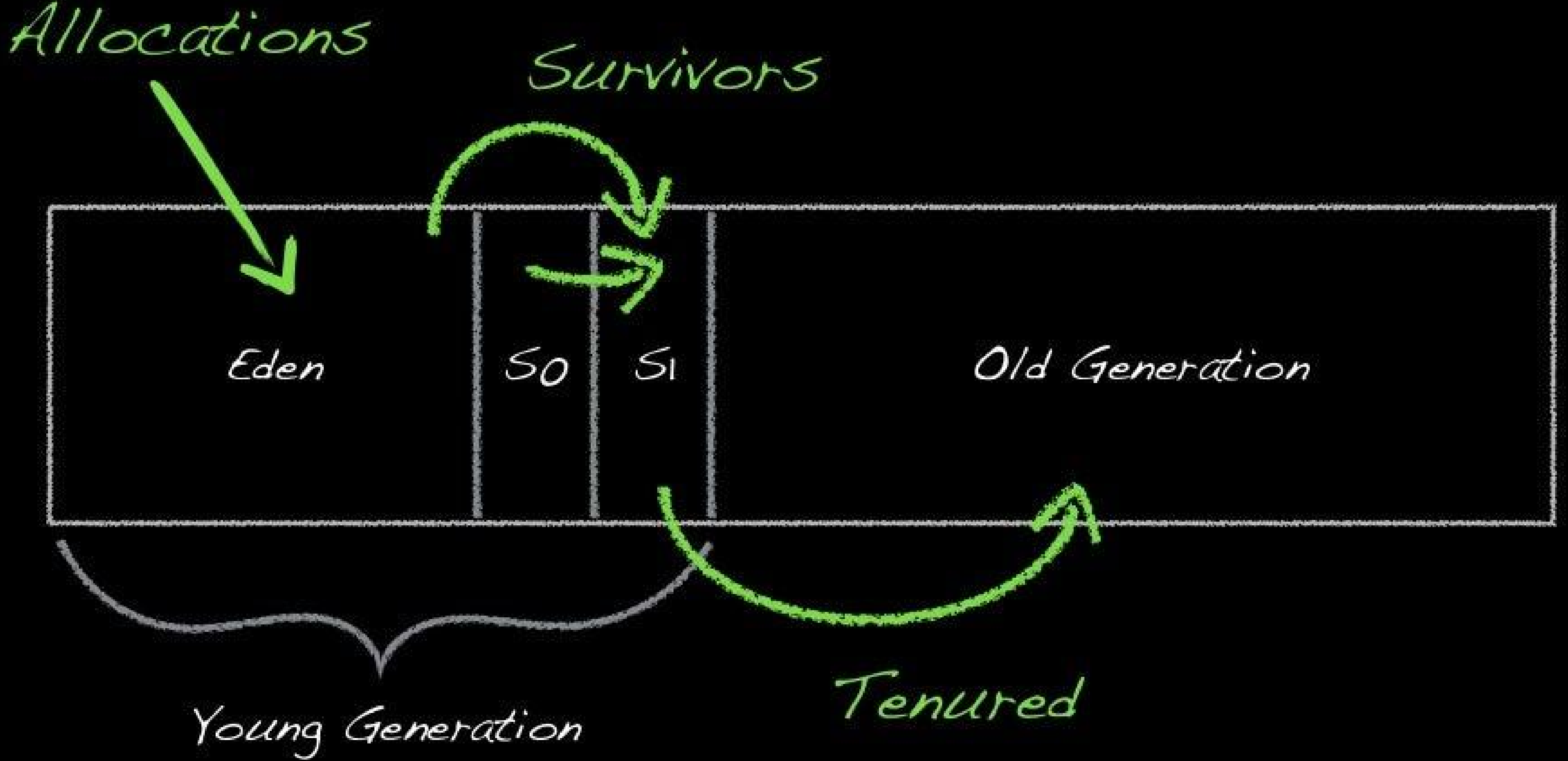
GC Algorithms in OpenJDK HotSpot - Latency Sensitive

- 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.

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?

The Throughput Collector - Java Heap

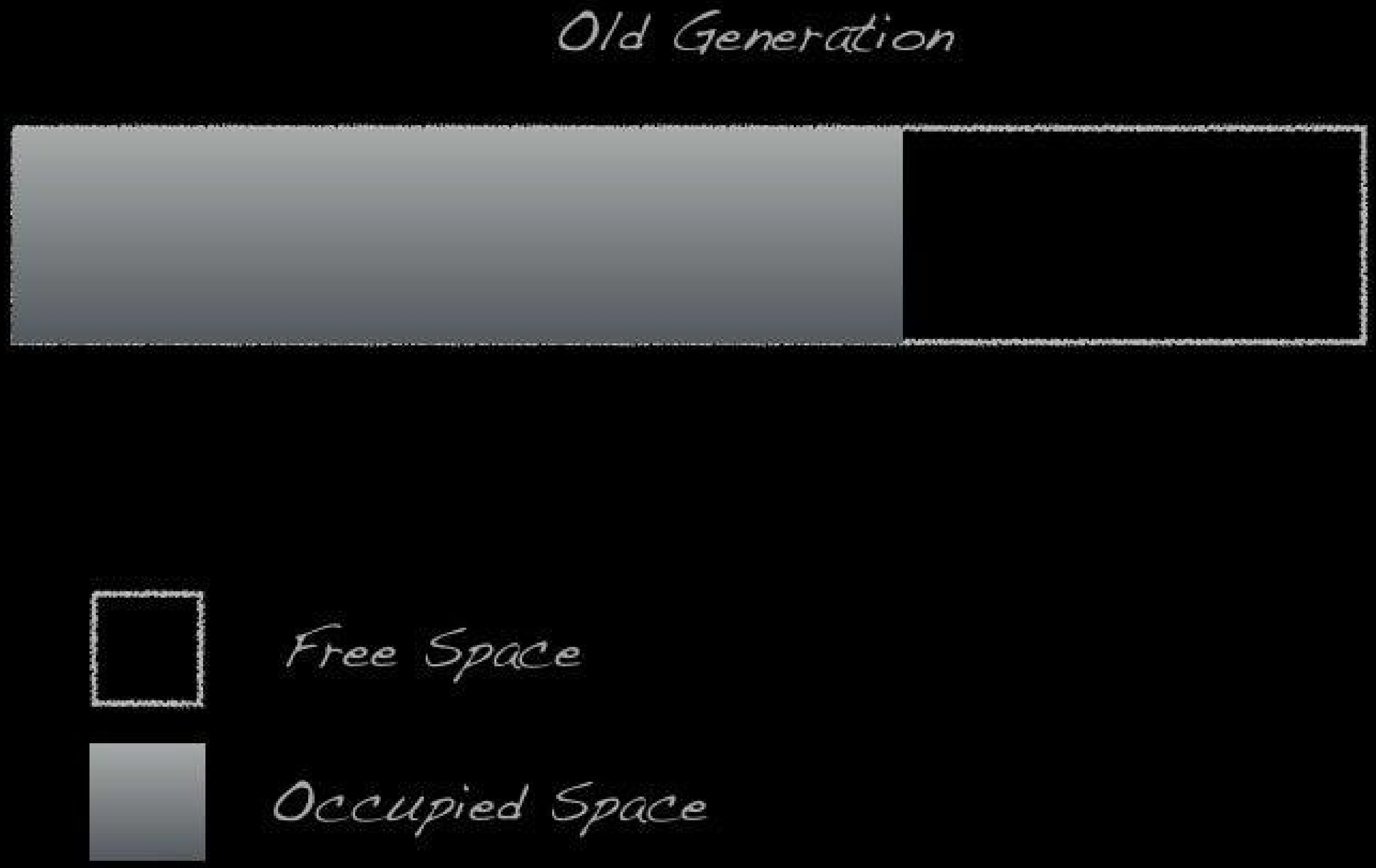


©2015 CodeKaram

The Throughput Collector - Young Collection

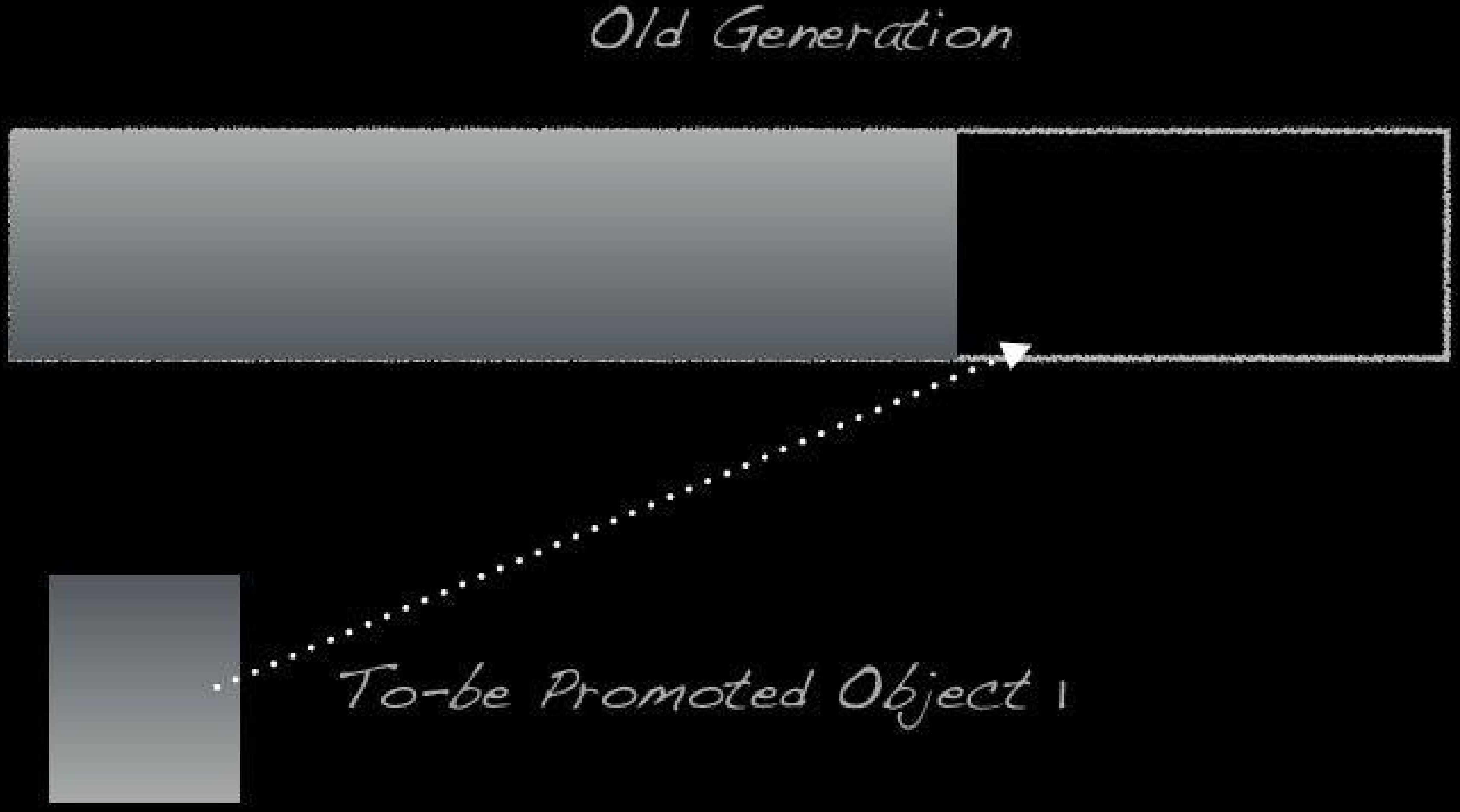
- An allocation failure results in a stop-the-world young collection.
- The young generation is collected in its entirety i.e. all objects (dead or alive) are emptied from the eden and S0 spaces.
- After the young collection is complete, the surviving objects (objects that are live) are moved into S1.
- 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)

The Throughput Collector - Contiguous Old Generation

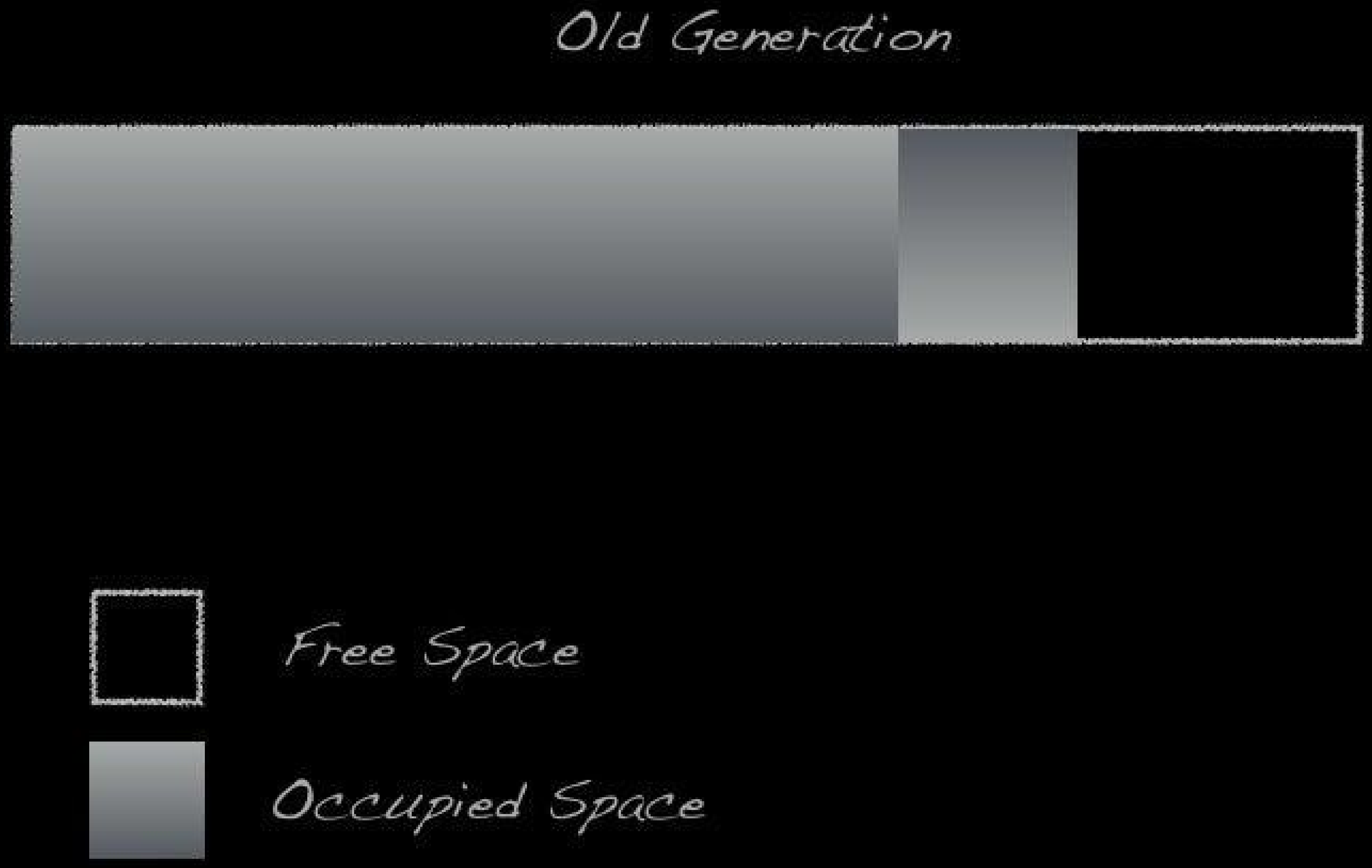


©2015 CodeKaram

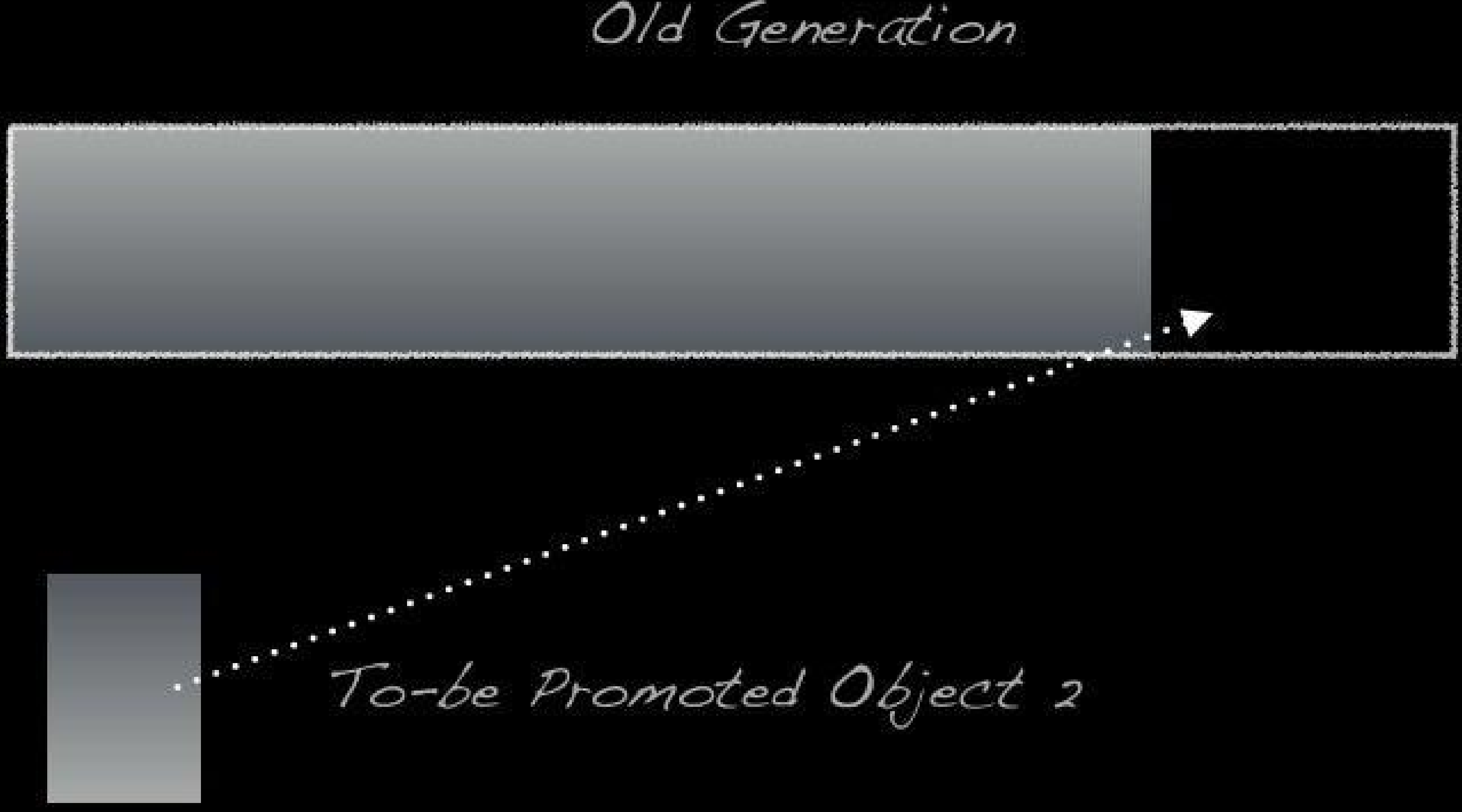
The Throughput Collector - Contiguous Old Generation



The Throughput Collector - Contiguous Old Generation



The Throughput Collector - Contiguous Old Generation



©2015 CodeKaram