GAME OF PERFORMANCE

A Song of JIT and GC

QCon London 2016

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

About Me

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

Many Thanks

- Vladimir Kozlov (Hotspot JIT Team) - for clearing my understanding of tiered compilation, escape analysis and nuances of dynamic deoptimizations.

- Jon Masamitsu (Hotspot GC Team) - for keeping me on track with JDK8 changes that related to GC.

Agenda

- The Helpers within the JVM
  - JIT & Runtime
  - Adaptive Optimization
    - CompileThreshold
    - Inlining
    - Dynamic Deoptimization

Agenda

- The Helpers within the JVM
  - JIT & Runtime
    - Tiered Compilation
    - Intrinsics
    - Escape Analysis
    - Compressed Oops
      - Compressed Class Pointers

Agenda

- The Helpers within the JVM
  - Garbage Collection
  - Allocation
    - TLAB
    - NUMA Aware Allocator

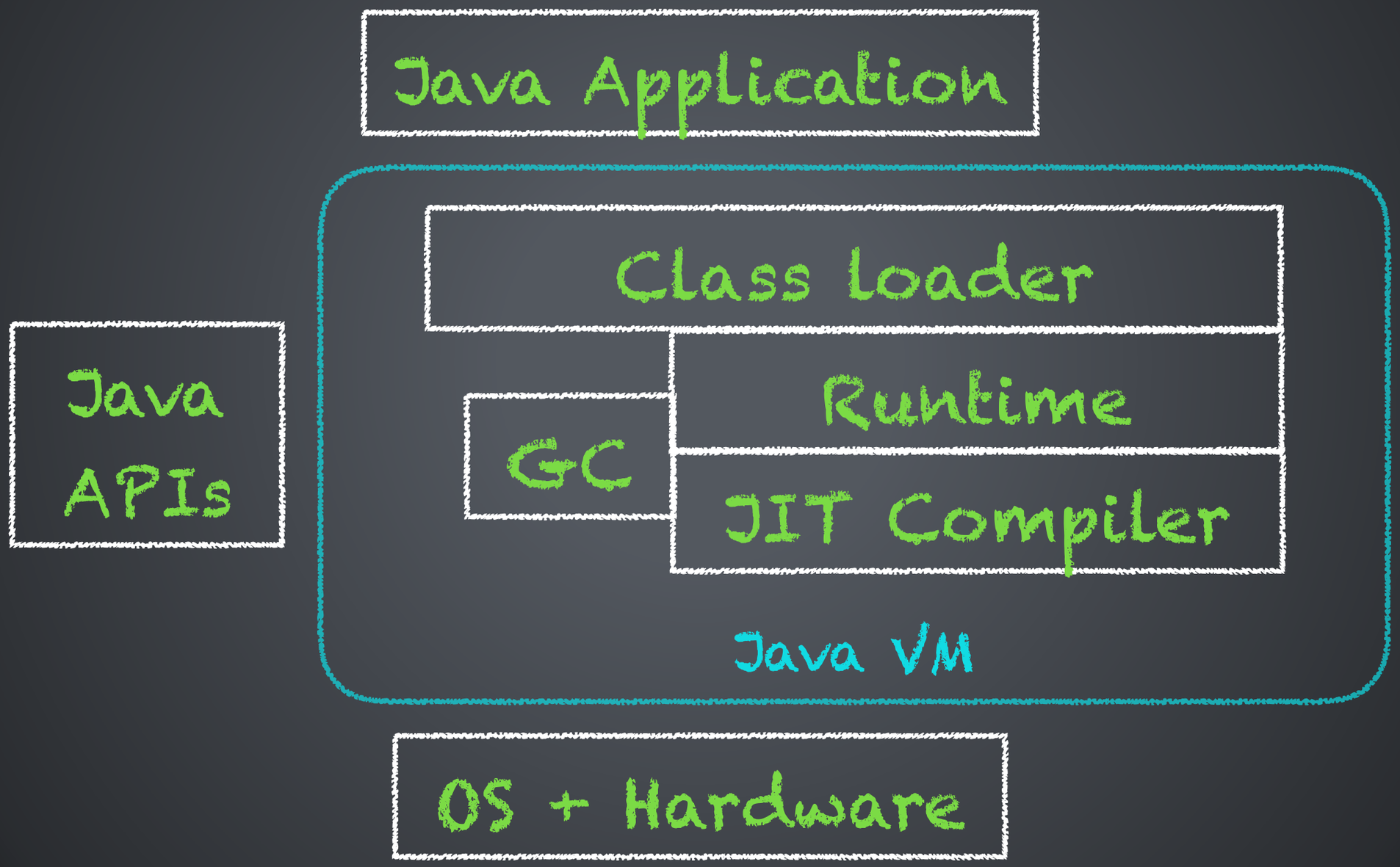
Agenda

- The Helpers within the JVM
  - Garbage Collection
  - Reclamation
    - Mark-Sweep-Compact (Serial + Parallel)
    - Mark-Sweep
    - Scavenging

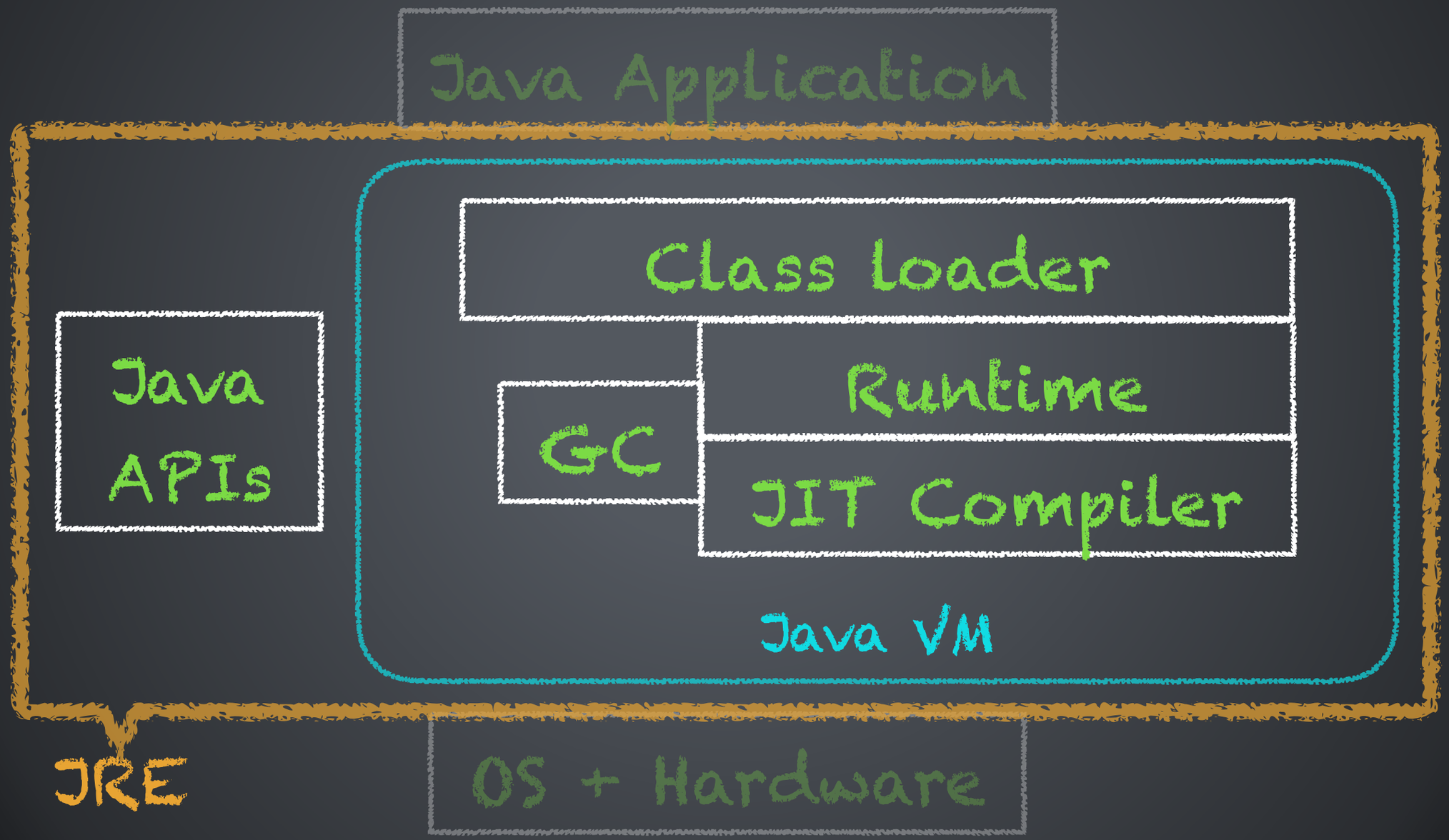
Agenda

- The Helpers within the JVM
- Garbage Collection
    - String Interning and Deduplication

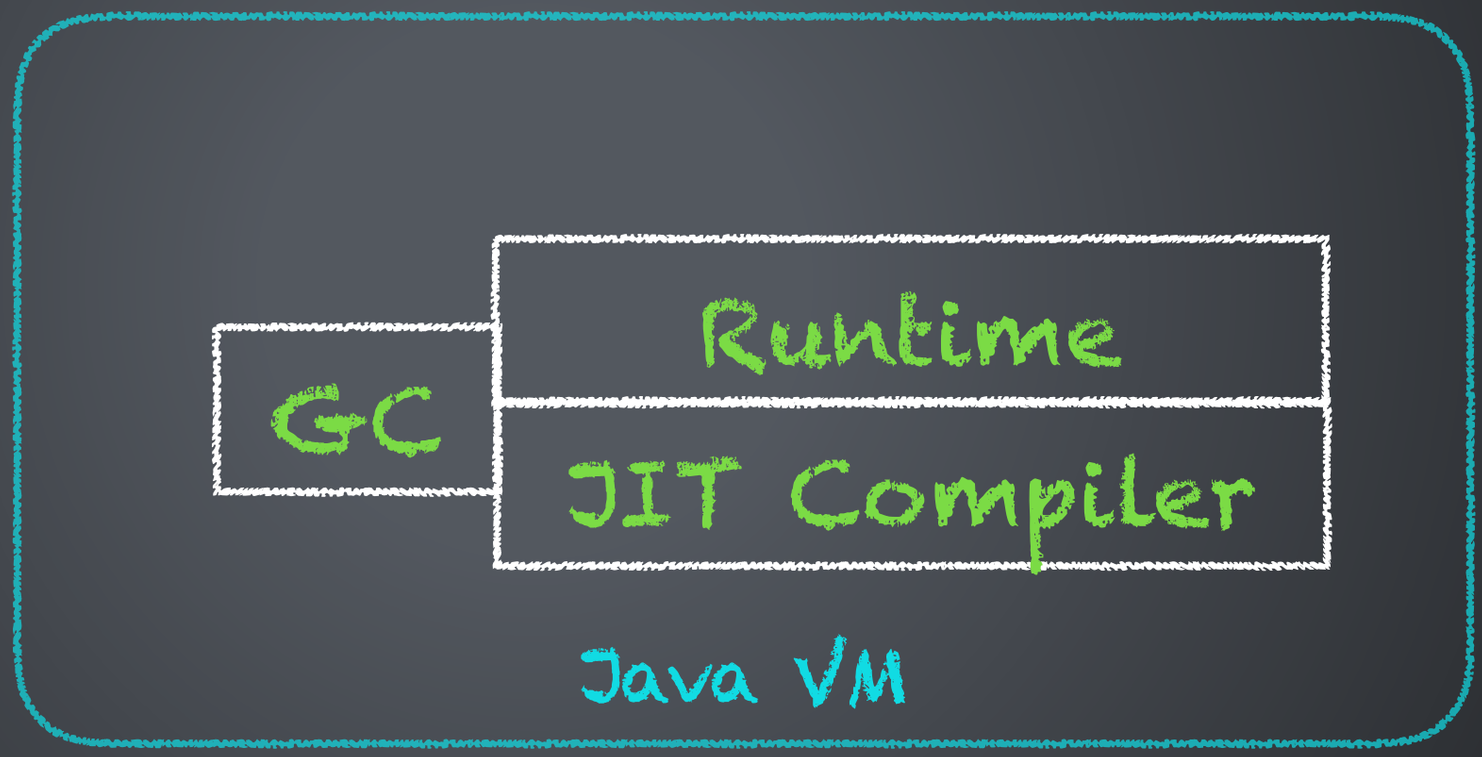
Interfacing The Metal



Interfacing The Metal



The Helpers



The Helpers - JIT And Runtime.

Advanced JIT & Runtime Optimizations - Adaptive Optimization

JIT & Runtime

- Startup - Interpreter
- Adaptive optimization - Performance critical methods
- Compilation:
  - CompileThreshold
  - Identify root of compilation
  - Method Compilation or On-stack replacement (Loop)?

PrintCompilation

```
timestamp compilation-id flags tiered-
compilation-level Method <@ osr_bci> code-
size <deoptimization>
```

PrintCompilation

```
Flags:
%: is_osr_method
s: is_synchronized
!: has_exception_handler
b: is_blocking
n: is_native
```

PrintCompilation

```
    567  693 % !   3
org.h2.command.dml.Insert::insertRows @ 76 (513 bytes)

    656  797     n 0
java.lang.Object::clone (native)

    779  835  s    4
java.lang.StringBuffer::append (13 bytes)
```

JIT & Runtime

- Adaptive optimization - Performance critical methods

- Inlining:

    - MinInliningThreshold, MaxFreqInlineSize, InlineSmallCode, MaxInlineSize, MaxInlineLevel, DesiredMethodLimit …

PrintInling*

```
@ 76   java.util.zip.Inflater::setInput (74 bytes)
too big
@ 80   java.io.BufferedInputStream::getBufIfOpen (21
bytes)   inline (hot)
@ 91   java.lang.System::arraycopy (0 bytes)
(intrinsic)
@ 2   java.lang.ClassLoader::checkName (43 bytes)
callee is too large
```

* needs -XX:+UnlockDiagnosticVMOptions

JIT & Runtime

- Adaptive optimization - Performance critical methods
- Dynamic de-optimization
  - dependencies invalidation
  - classes unloading and redefinition
  - uncommon path in compiled code

PrintCompilation

```
    573  704       2
org.h2.table.Table::fireAfterRow (17 bytes)
   7963 2223       4
org.h2.table.Table::fireAfterRow (17 bytes)
   7964  704       2
org.h2.table.Table::fireAfterRow (17 bytes)   made not
entrant
  33547  704       2
org.h2.table.Table::fireAfterRow (17 bytes)   made
zombie
```

Advanced JIT & Runtime Optimizations - Tiered Compilation

Tiered Compilation

- Start in interpreter
- Tiered optimization with client compiler
  - Code profiled information
- Enable server compiler

Tiered Compilation - Effect on Code Cache

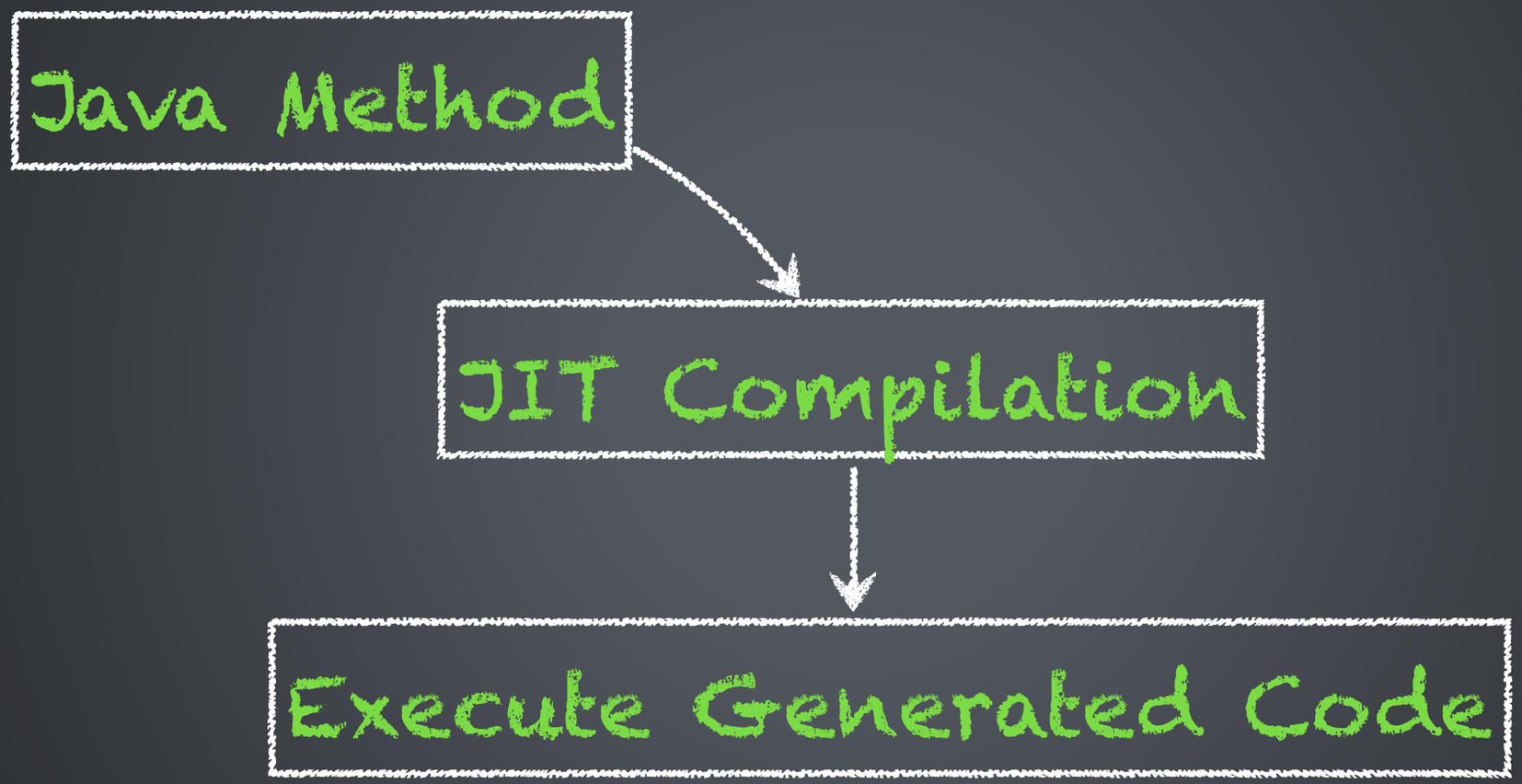
- C1 compilation threshold for tiered is about a 100 invocations
- Tiered Compilation has a lot more profiled information for C1 compiled methods
- CodeCache needs to be 5x larger than non-tiered
  - Default on JDK8 when tiered is enabled (48MB vs 240MB)
  - Need more? Use -XX:ReservedCodeCacheSize

Other JIT Compiler Optimizations

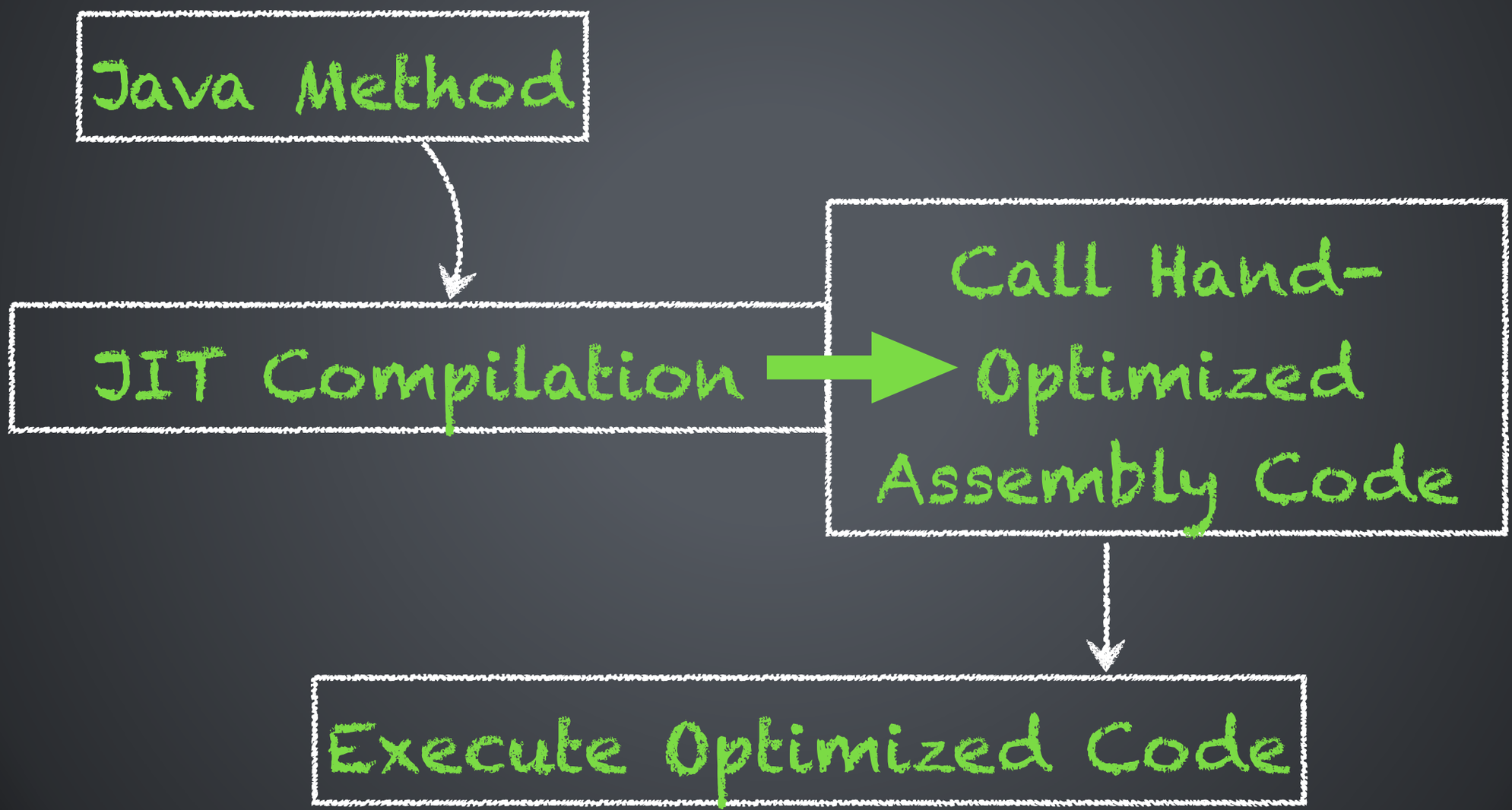
- Fast dynamic type tests for type safety
- Range check elimination
- Loop unrolling
- Profile data guided optimizations
- Escape Analysis
- Intrinsics
- Vectorization

Advanced JIT & Runtime Optimizations - Intrinsics

Without Intrinsics



Intrinsics



PrintInling*

```
@ 76   java.util.zip.Inflater::setInput (74 bytes)
too big
@ 80   java.io.BufferedInputStream::getBufIfOpen (21
bytes)   inline (hot)
@ 91   java.lang.System::arraycopy (0 bytes)
(intrinsic)
@ 2   java.lang.ClassLoader::checkName (43 bytes)
callee is too large
```

* needs -XX:+UnlockDiagnosticVMOptions

Advanced JIT & Runtime Optimizations - Escape Analysis

Escape Analysis

- Entire IR graph
- escaping allocations?
- not stored to a static field or non-static field of an external object,
- not returned from method,
- not passed as parameter to another method where it escapes.

Escape Analysis

allocated object doesnt
escape the compiled method

+

allocated object not
passed as a parameter

=

remove allocation and keep field
values in registers

Escape Analysis

allocated object doesn't
escape the compiled method

+

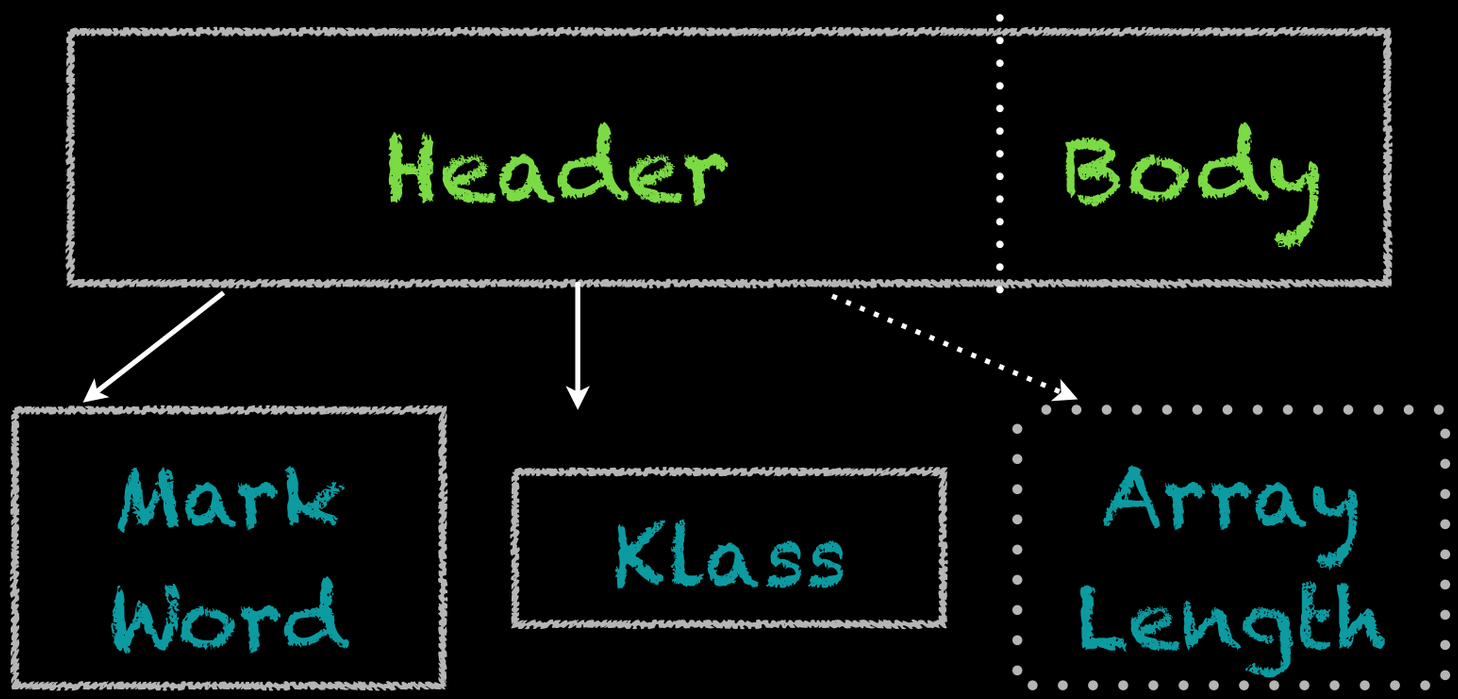
allocated object is
passed as a parameter

=

remove locks associated with object
and use optimized compare instructions

Advanced JIT & Runtime Optimizations - Compressed Oops

A Java Object



Objects, Fields & Alignment

- Objects are 8 byte aligned (default).
- Fields:
  - are aligned by their type.
  - can fill a gap that maybe required for alignment.
  - are accessed using offset from the start of the object

ILP32 vs. LP64 Field Sizes

Mark Word - 32 bit vs 64 bit

Klass - 32 bit vs 64 bit

Array Length - 32 bit on both

boolean, byte, char, float, int, short - 32 bit on both

double, long - 64 bit on both

Compressed OOPs

<wide-oop> = <narrow-oop-base> +
(<narrow-oop> << 3) + <field-offset>

Compressed OOPs

Heap Size?	<4 GB (no encoding/ decoding needed)	>4GB; <28GB (zero-based)
<wide-oop> =	<narrow-oop>	<narrow-oop> << 3

Compressed OOPs

Heap Size?	>28 GB; <32 GB (regular)	>32 GB; <64 GB * (change alignment)
<wide-oop> =	<narrow-oop-base> + (<narrow-oop> << 3) + <field-offset>	<narrow-oop-base> + (<narrow-oop> << 4) + <field-offset>

Compressed OOPs

Heap Size?	<4 GB	>4GB; <28GB	<32GB	<64GB
Object Alignment?	8 bytes	8 bytes	8 bytes	16 bytes
Offset Required?	No	No	Yes	Yes
Shift by?	No shift	3	3	4

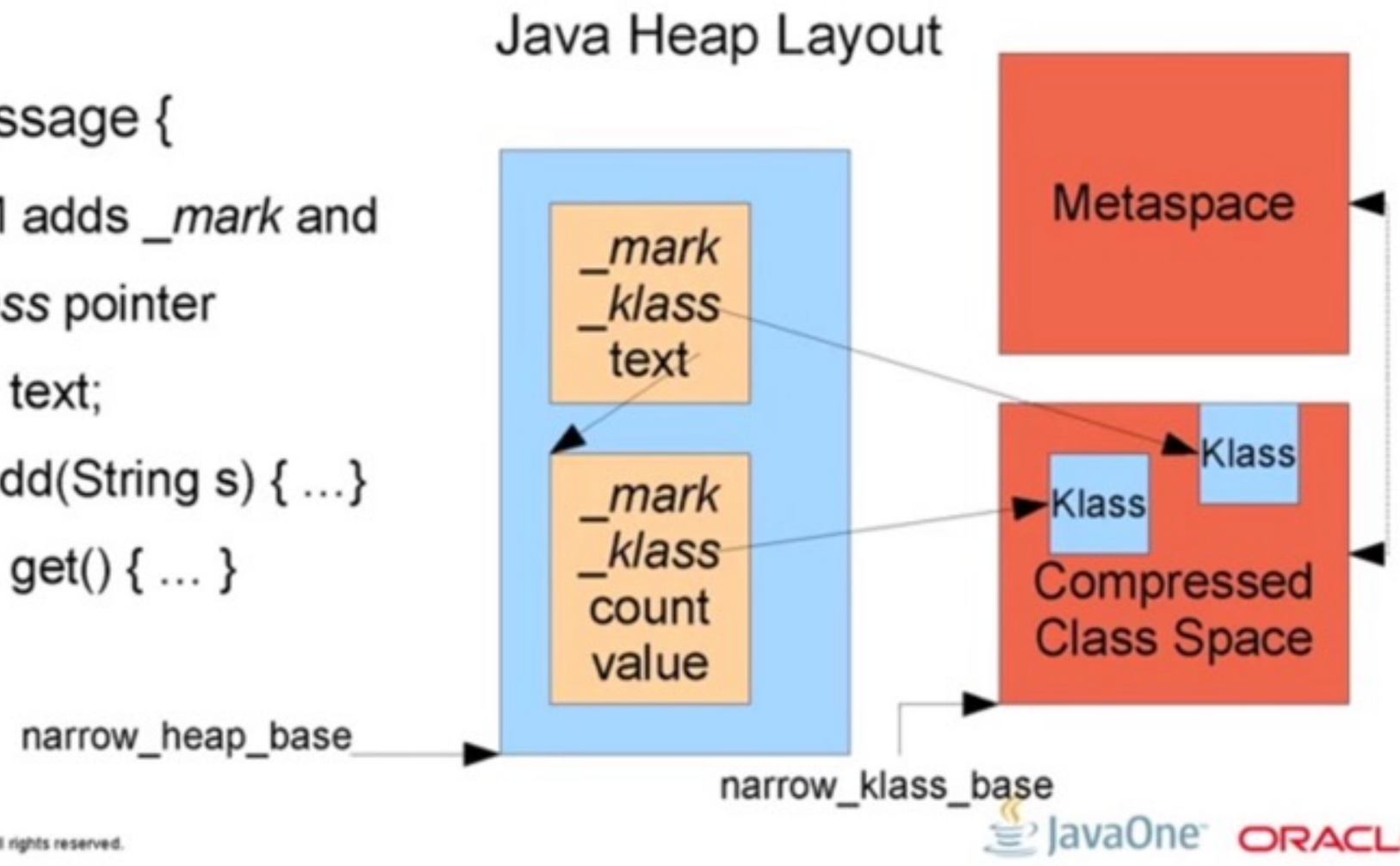
Compressed Class Pointers

- JDK 8 —> Perm Gen Removal —> Class Data outside of heap
- Compressed class pointer space
  - contains class metadata
  - is a part of Metaspace

Compressed Class Pointers

Java Object Memory Layout with Compressed Pointers

```
class Message {
    // JVM adds _mark and
    // _klass pointer
    String text;
    void add(String s) { …}
    String get() { … }
}
```



JavaOne ORACLE

PermGen Removal Overview by Coleen Phillimore + Jon Masamitsu @JavaOne 2013

The Helpers - Garbage Collection.

Garbage Collection

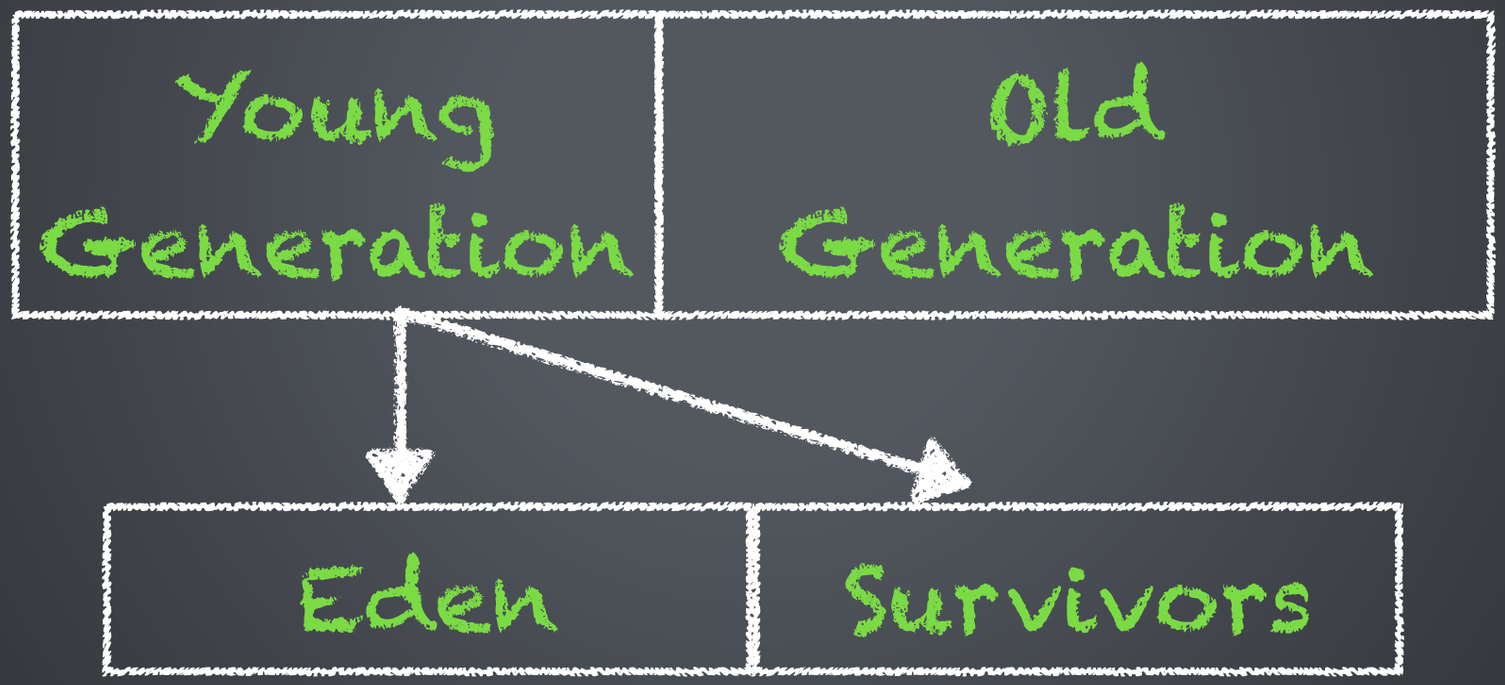
Allocation

+

Reclamation

Garbage Collection - Allocation.

Generational Heap



Fast Path Allocation - Thread Local Allocation Buffers (TLABs).

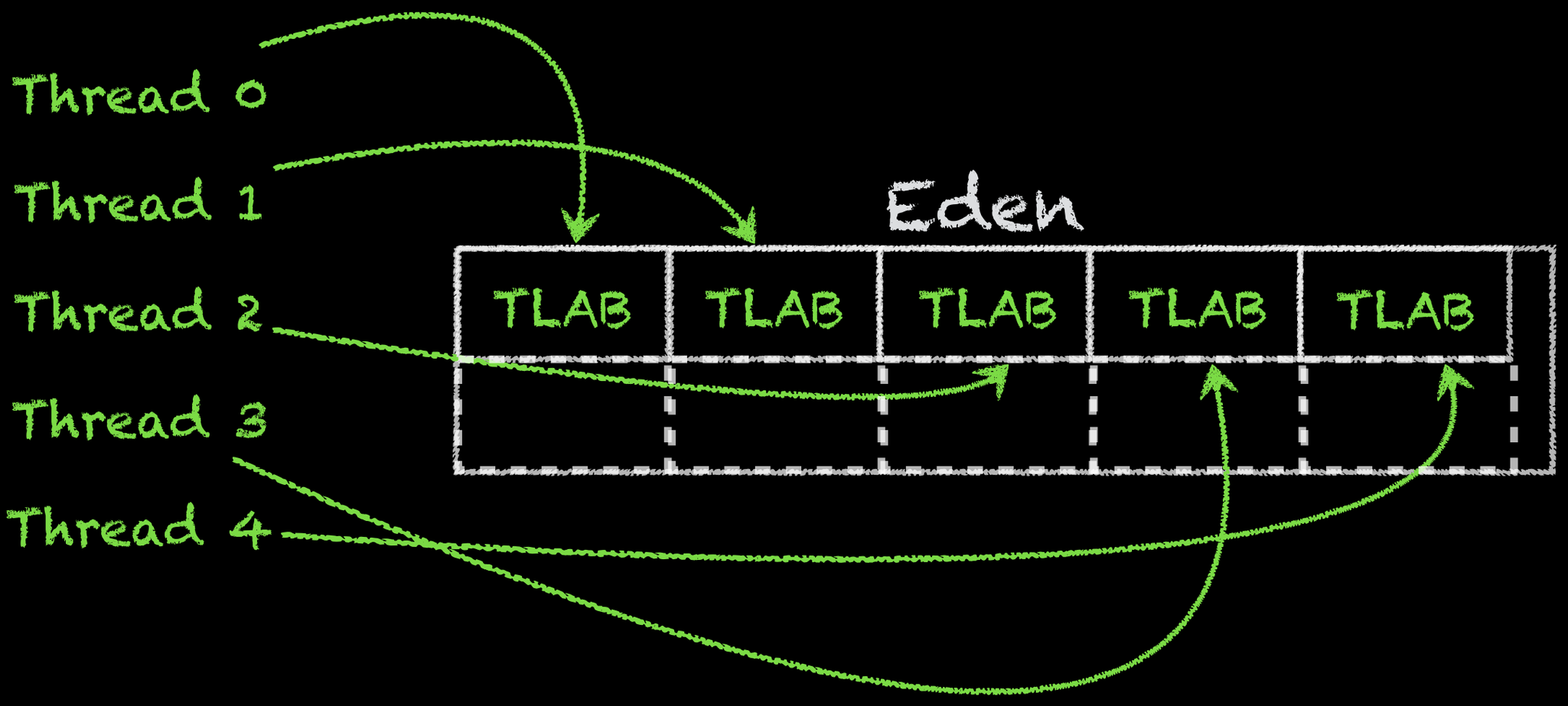
Garbage Collection - Allocation

Most Allocations → Fast Path → Eden Space

TLABs per thread:

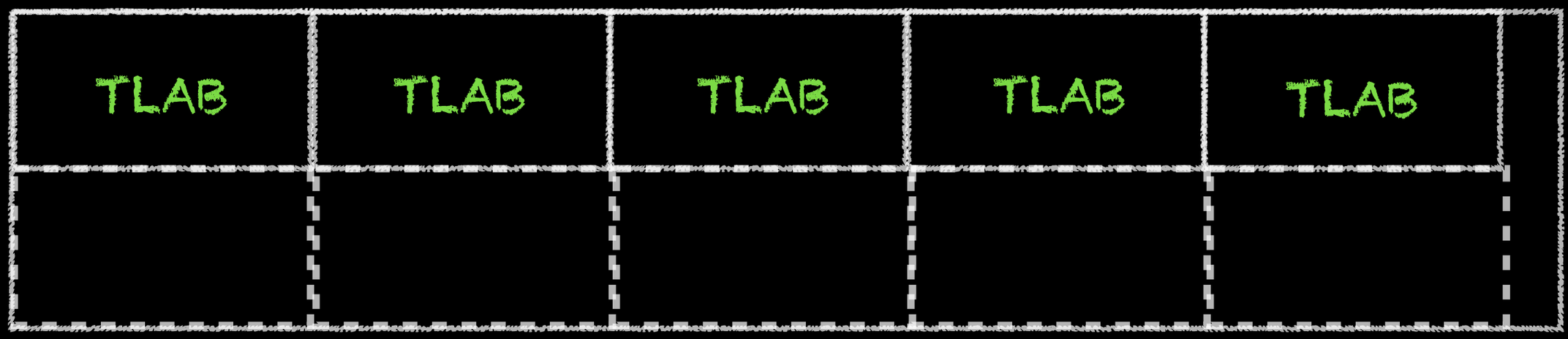
- allocate lock-free
- bump-(your-own)-pointer allocation
- only co-ordinate when needing a new TLAB

TLAB



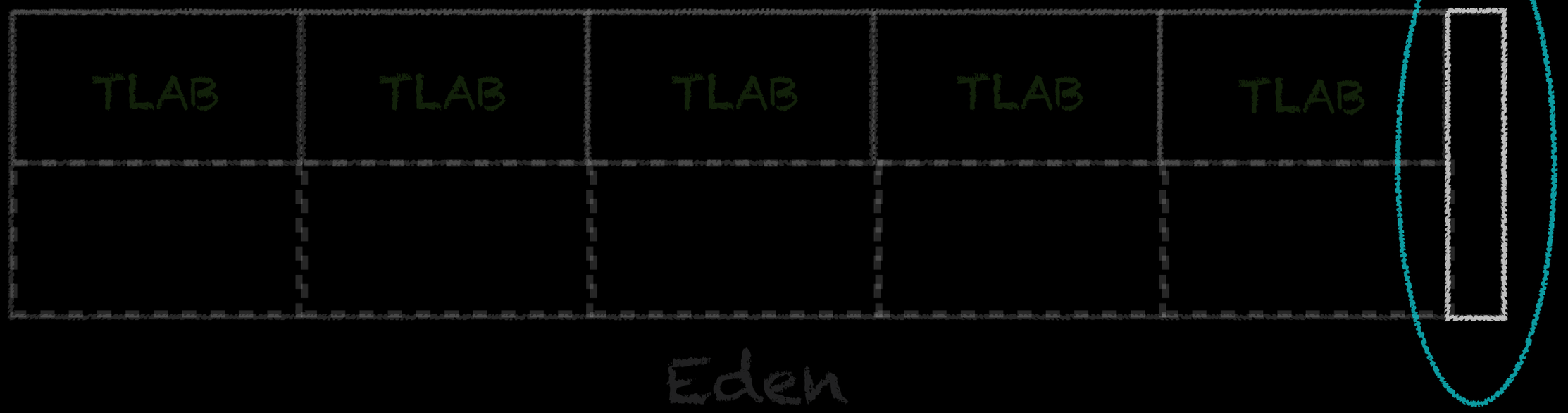
(Min)TLABSize
ResizeTLAB
TLABWasteTargetPercent
PrintTLAB

TLAB

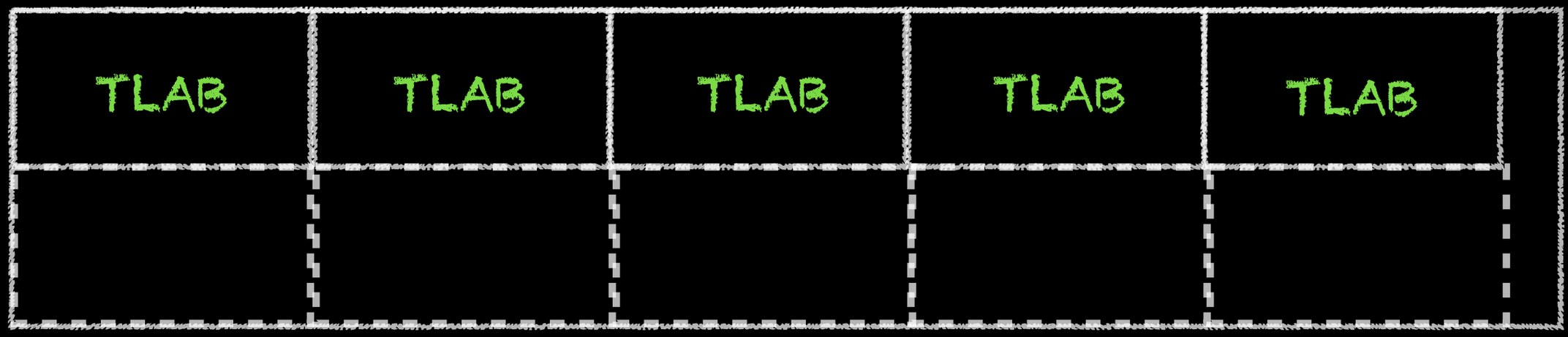


TLAB

TLABWasteTargetPercent (default = 1%)



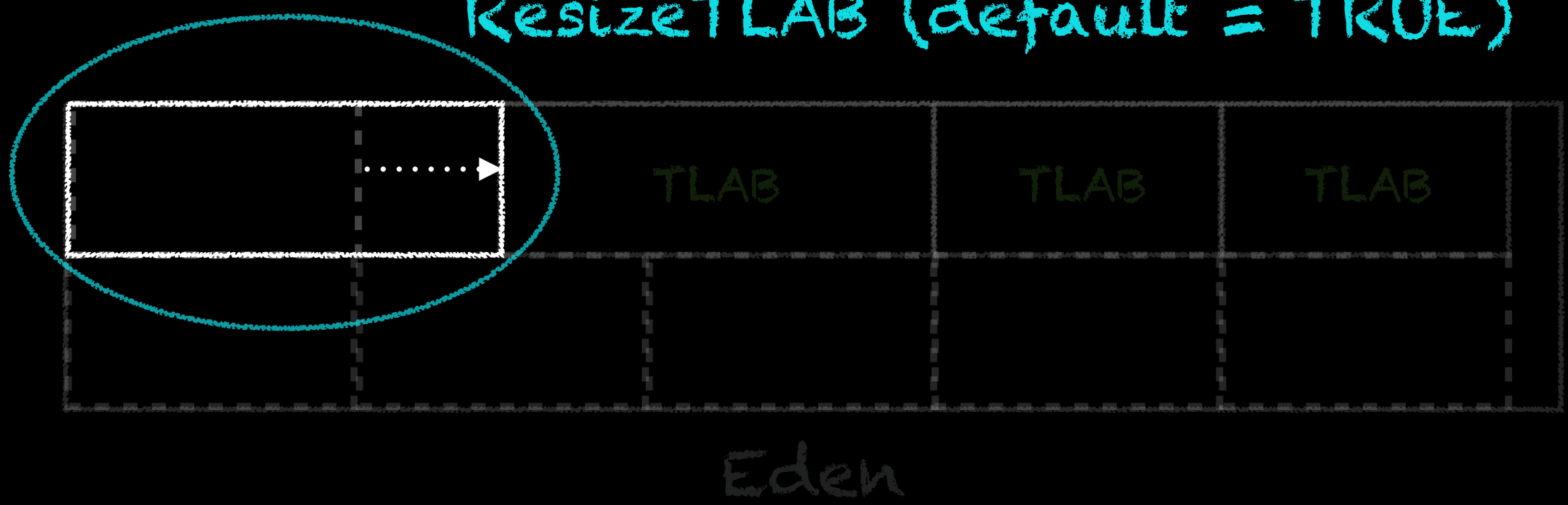
TLAB



TLAB

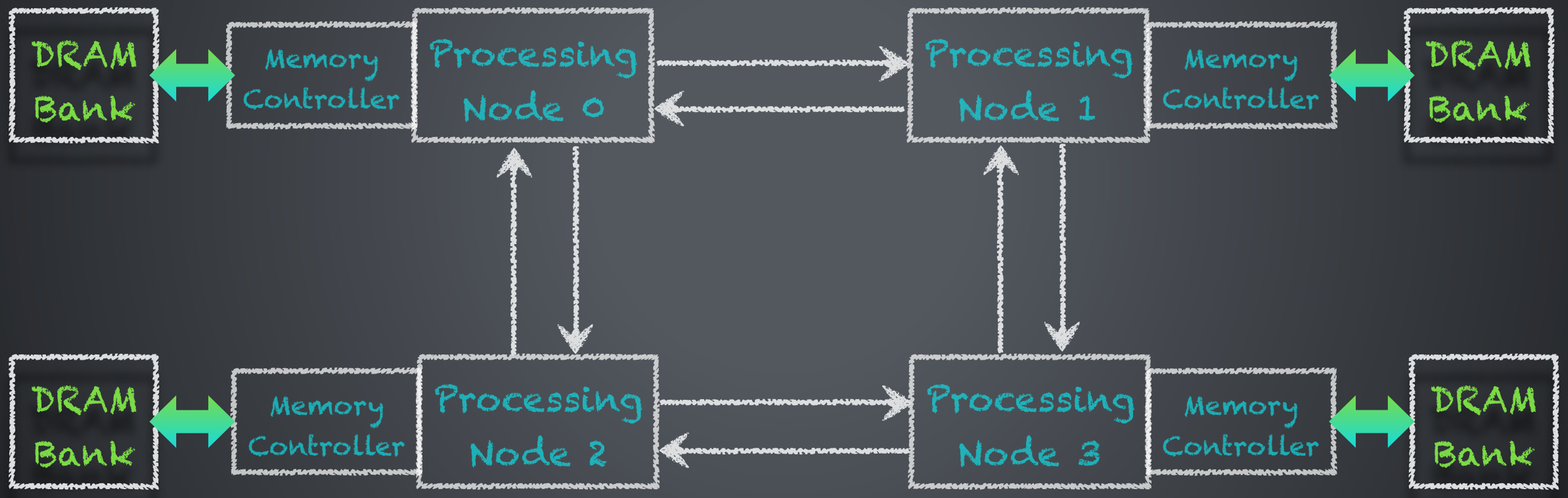
TLAB

ResizeTLAB (default = TRUE)

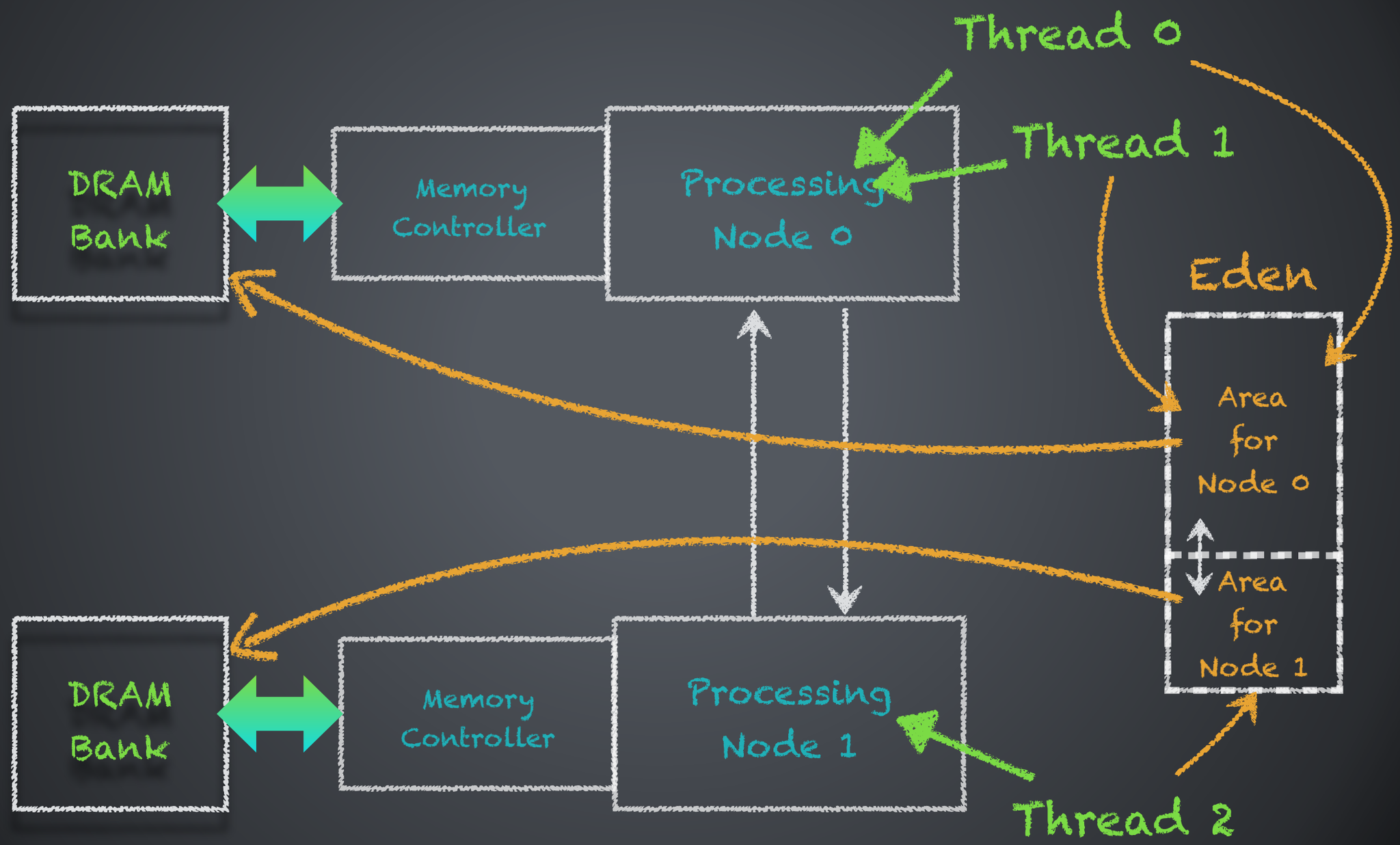


Allocation - Non Uniform Memory Access (NUMA) Aware Allocator.

NUMA



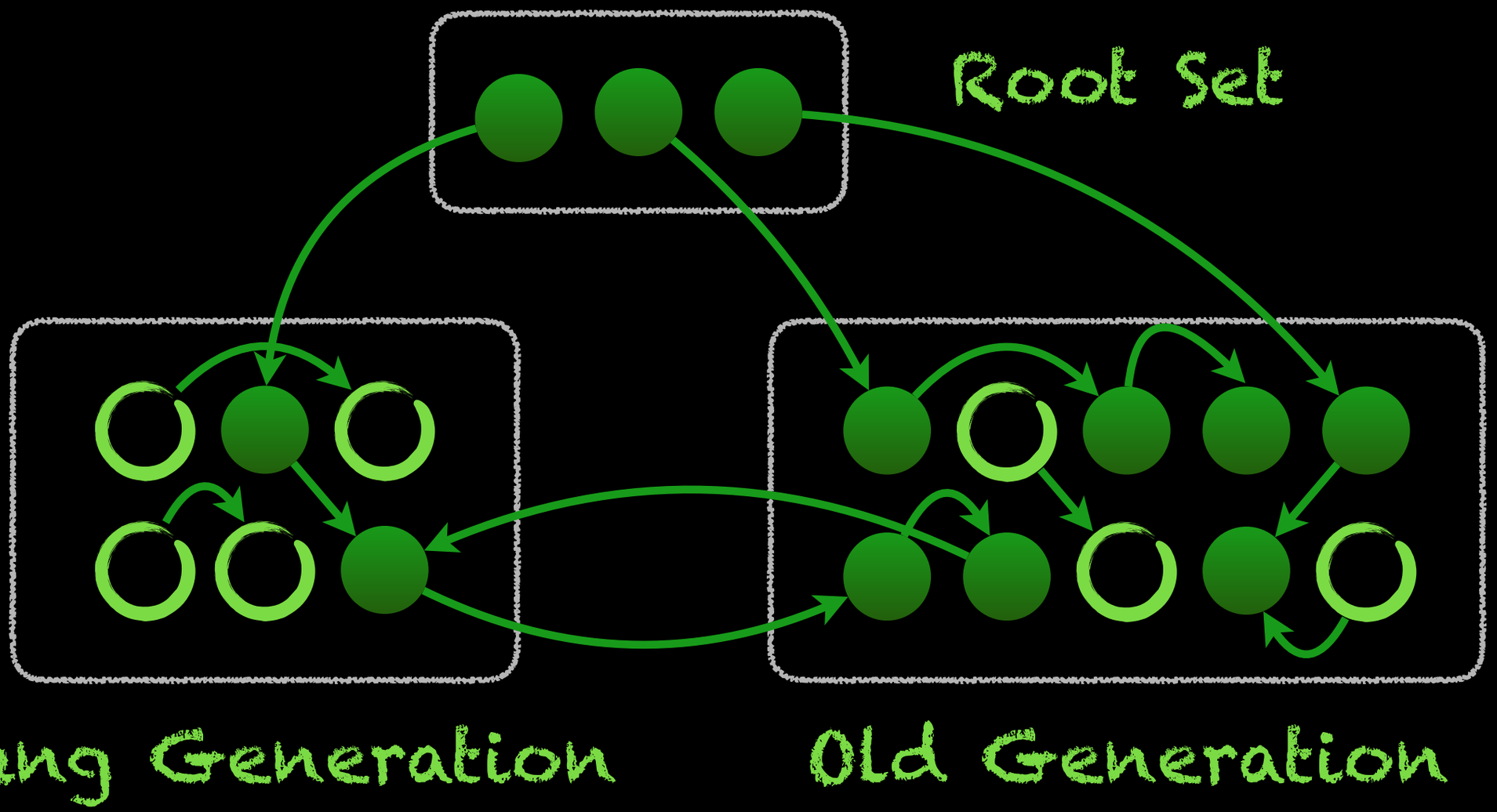
UseNUMA



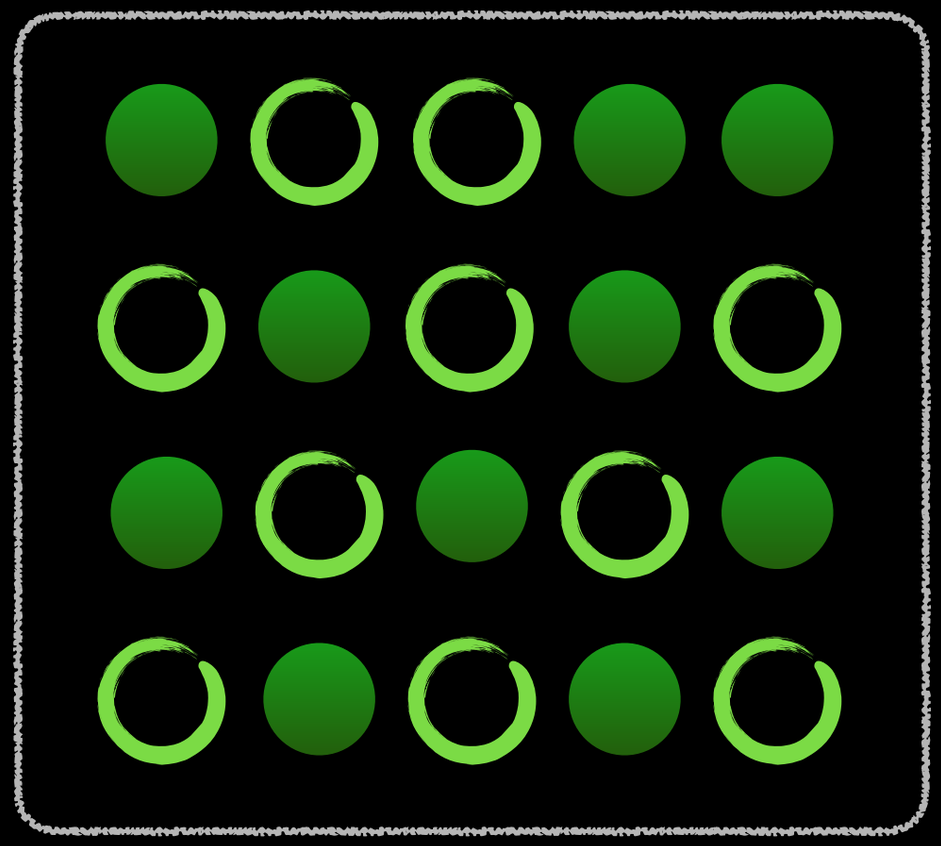
UseNUMA
UseNUMAInterleaving
UseAdaptiveNUMAChunkSizing
NUMAStats

Garbage Collection - Reclamation.

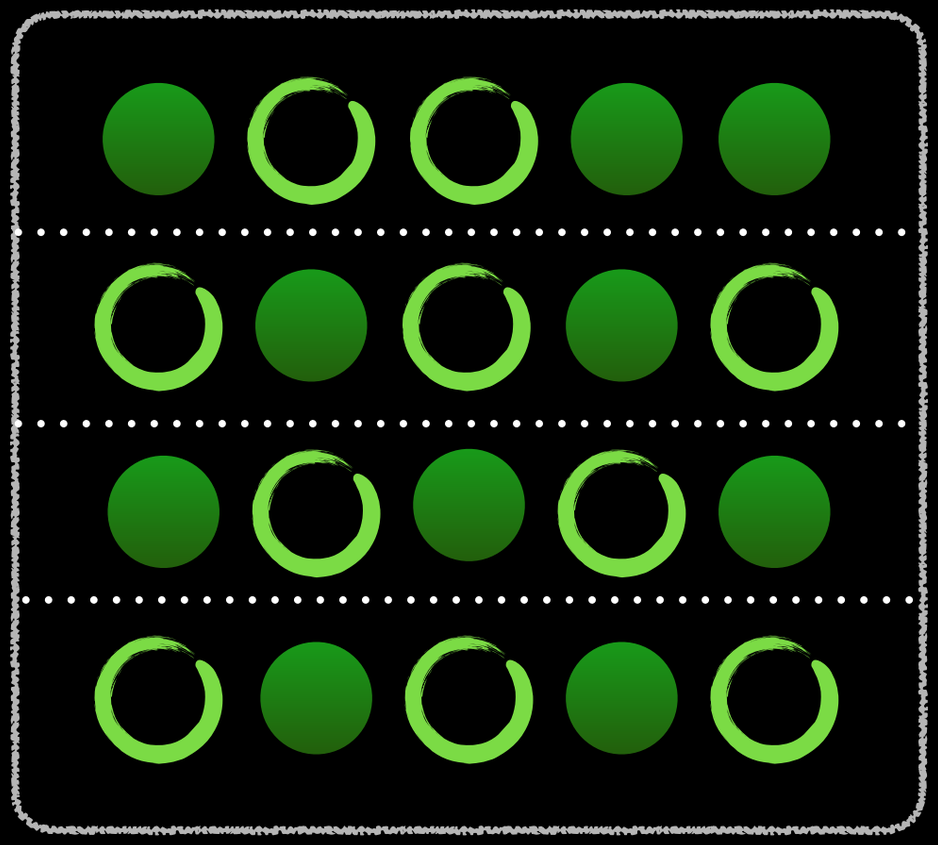
Garbage Collection -Reclamation via (Serial) Mark-Sweep-Compact



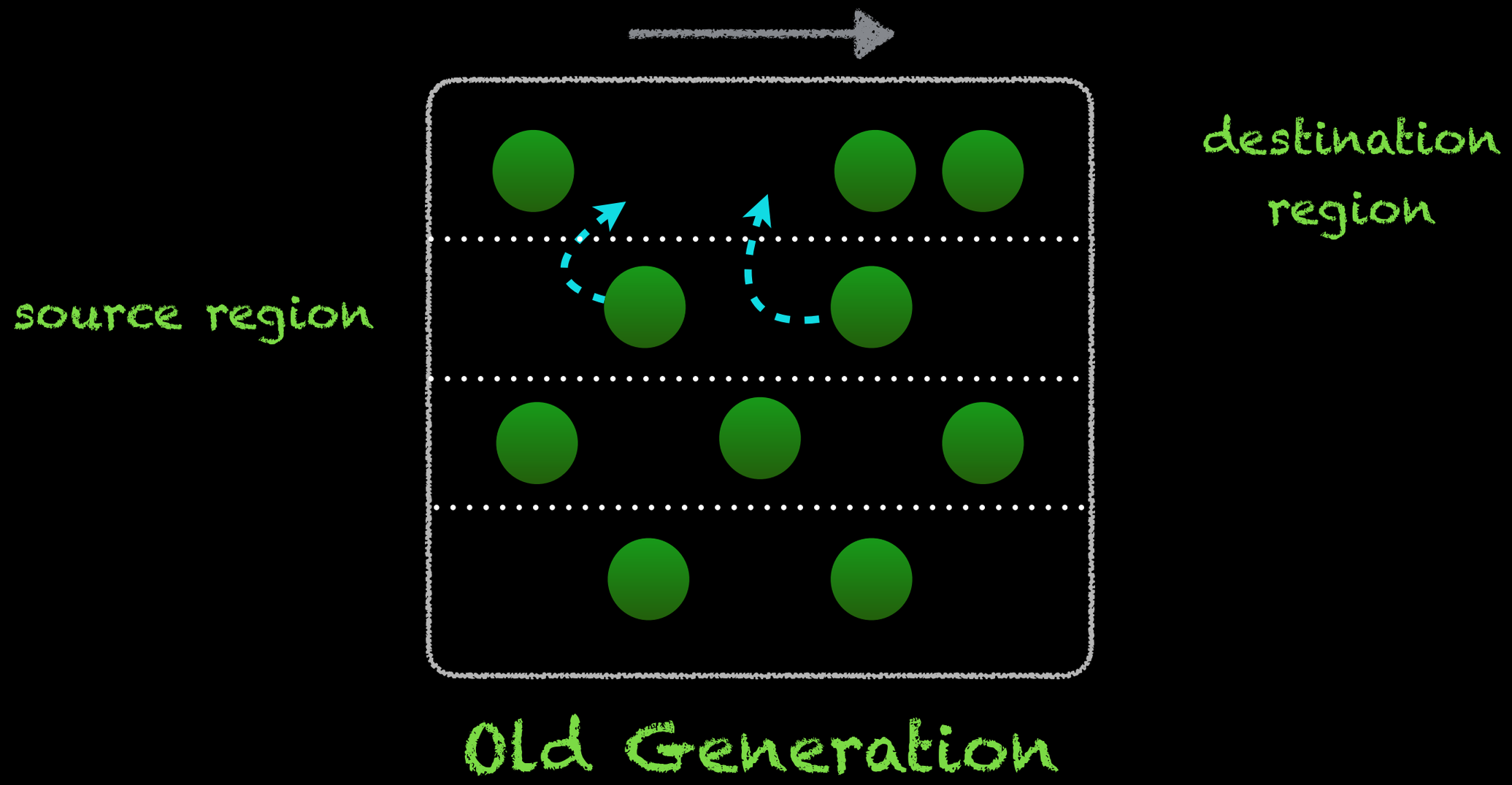
Garbage Collection -Reclamation via Parallel Mark-Compact



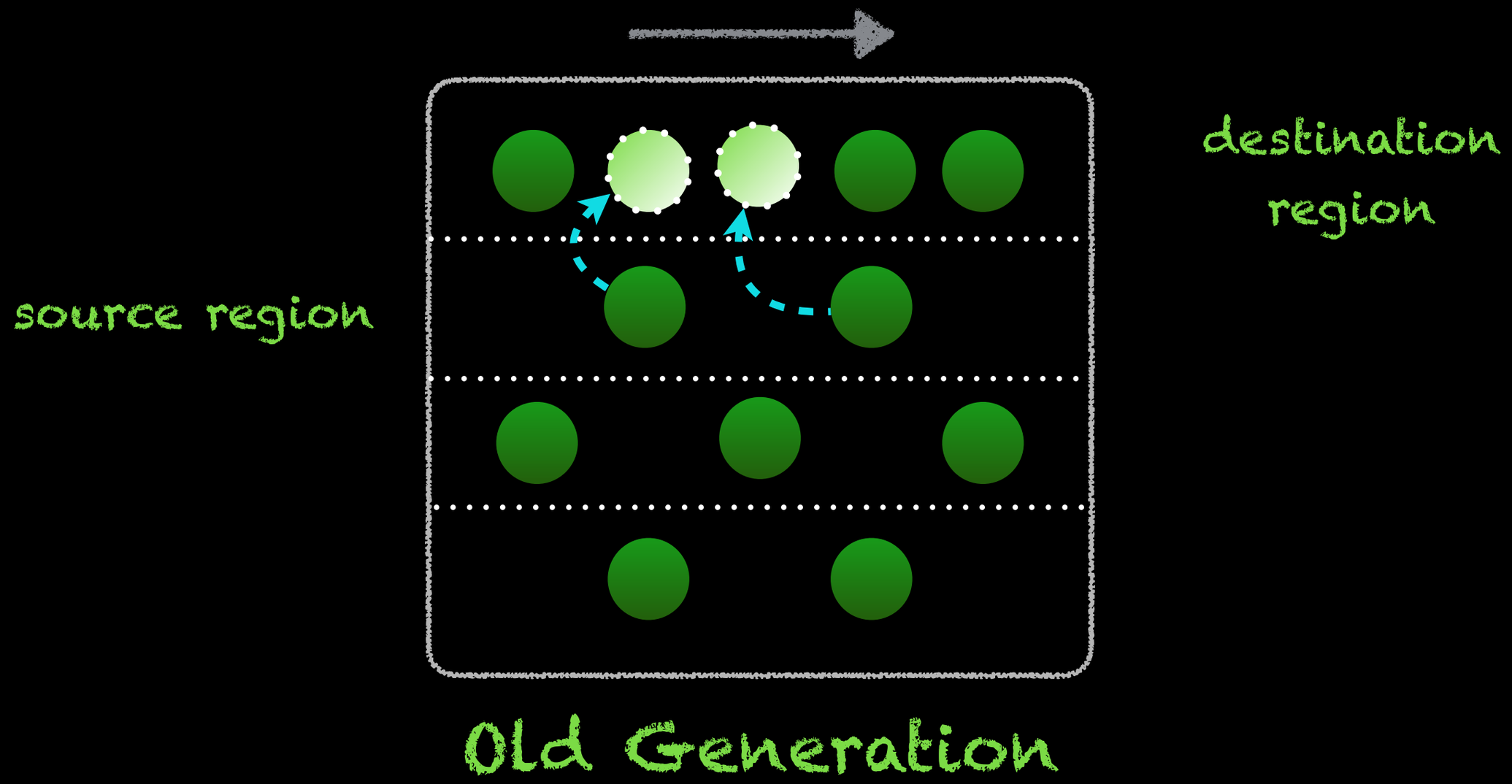
Garbage Collection -Reclamation via Parallel Mark-Compact



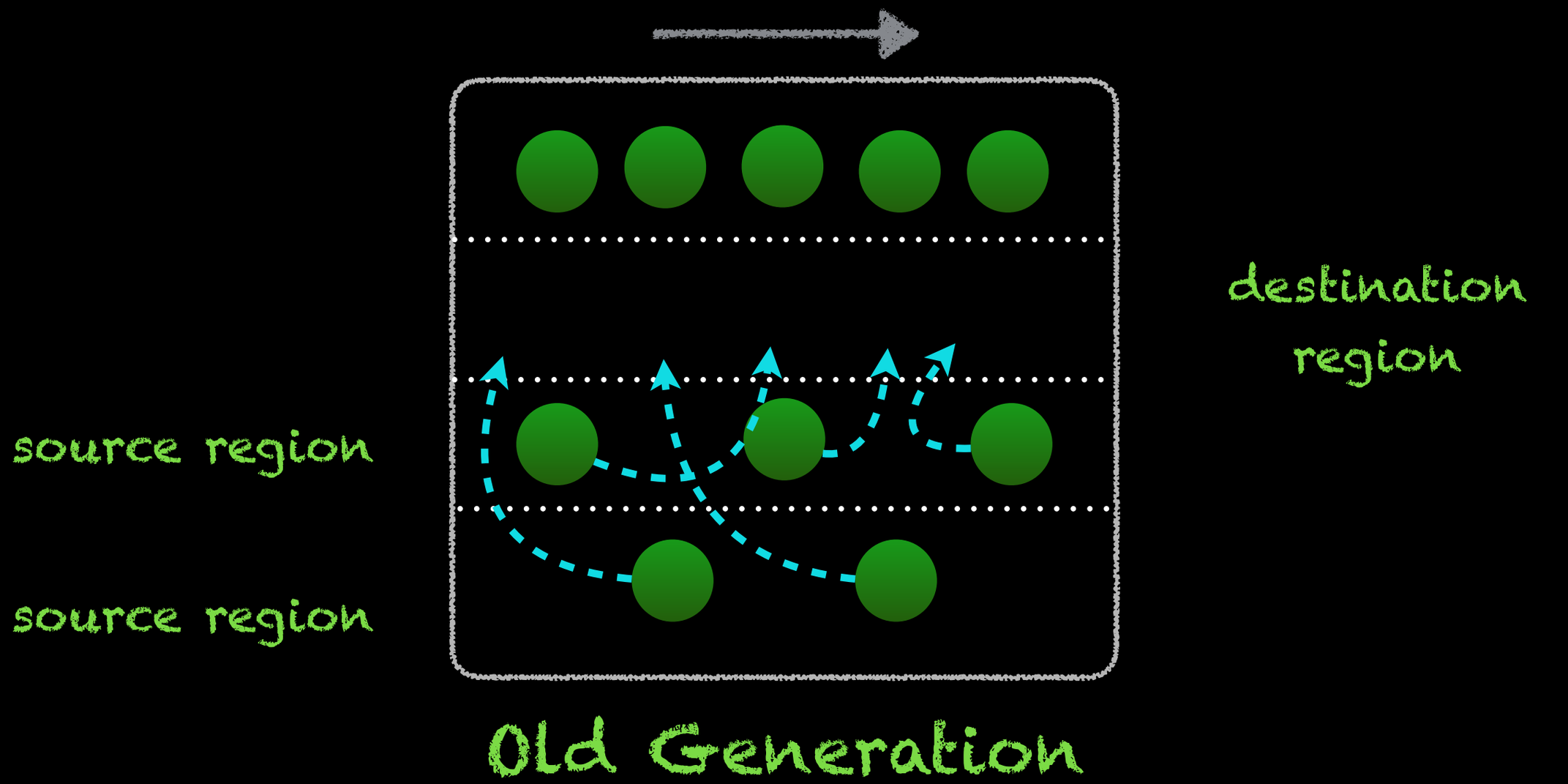
Garbage Collection -Reclamation via Parallel Mark-Compact



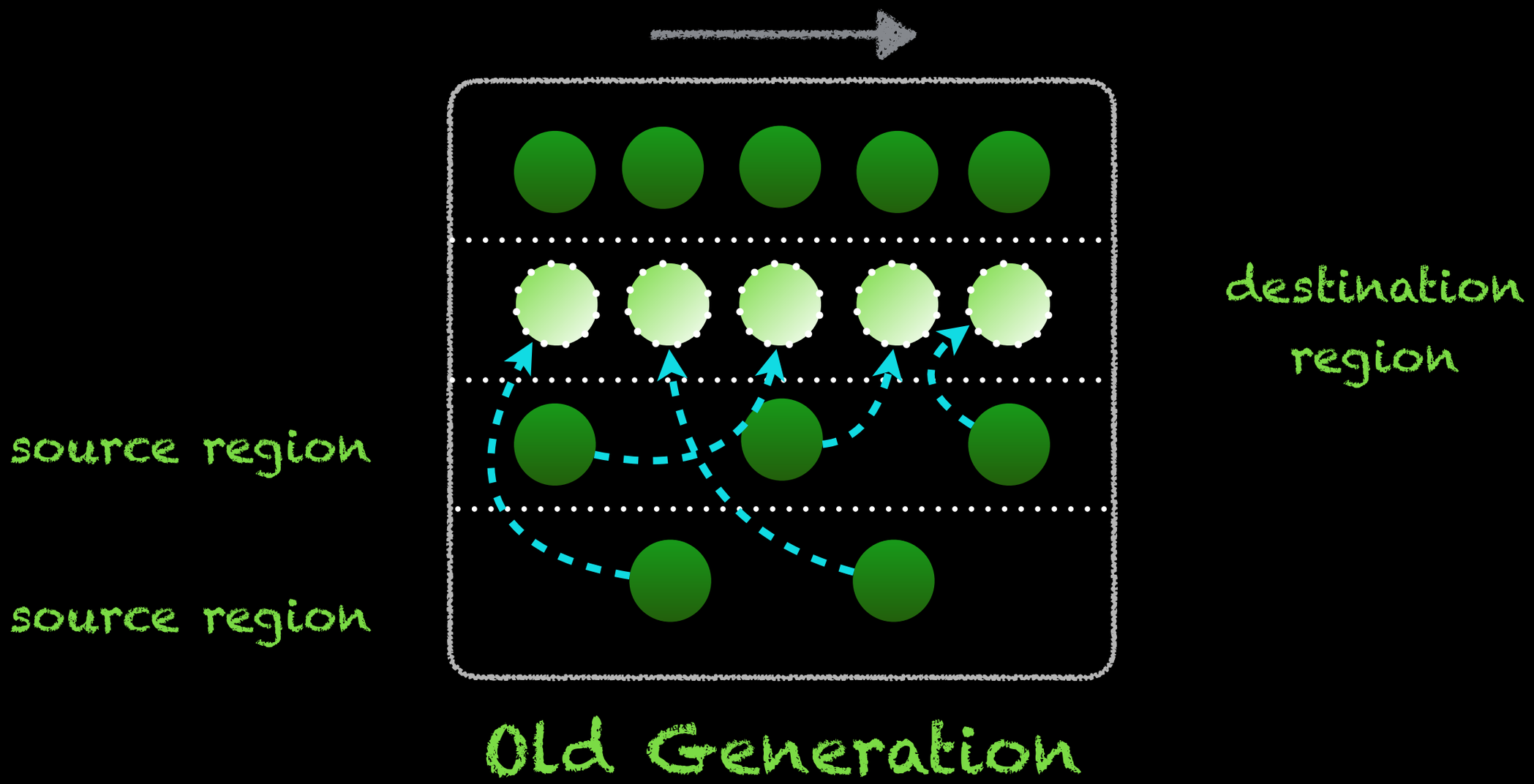
Garbage Collection -Reclamation via Parallel Mark-Compact



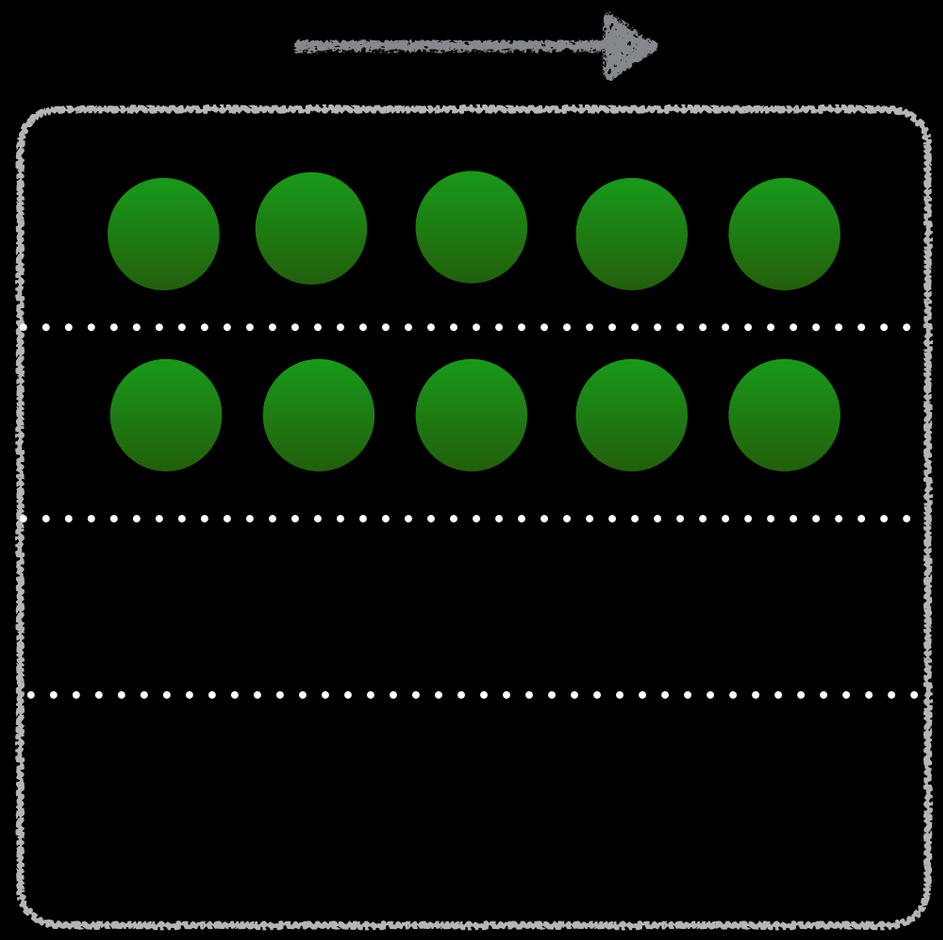
Garbage Collection -Reclamation via Parallel Mark-Compact



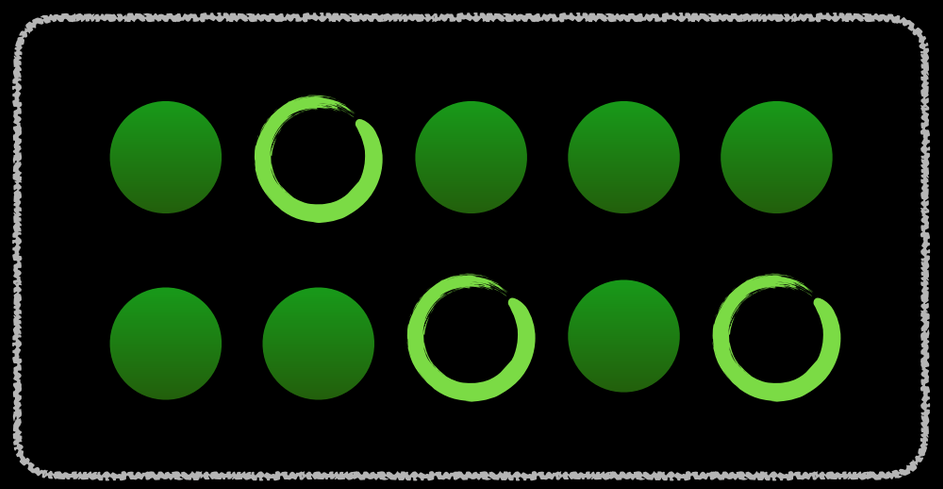
Garbage Collection -Reclamation via Parallel Mark-Compact



Garbage Collection -Reclamation via Parallel Mark-Compact

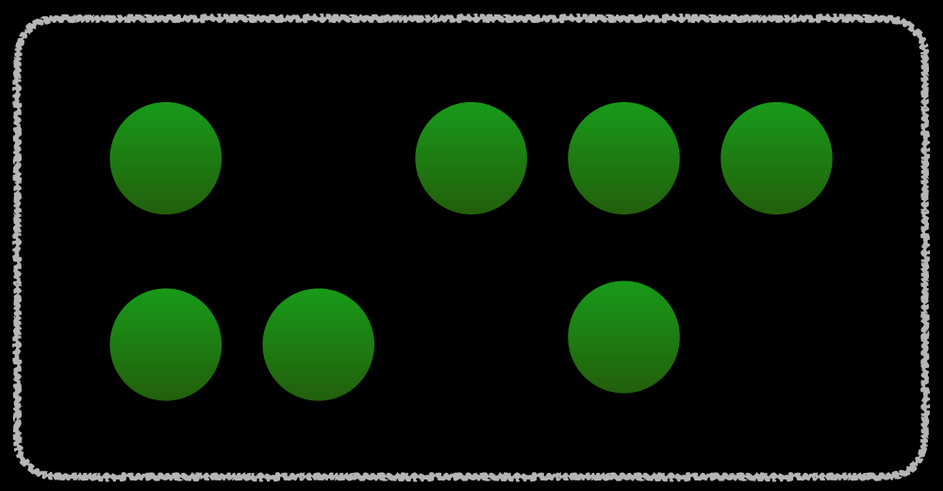


Garbage Collection - Reclamation via Mark-Sweep



Old Generation

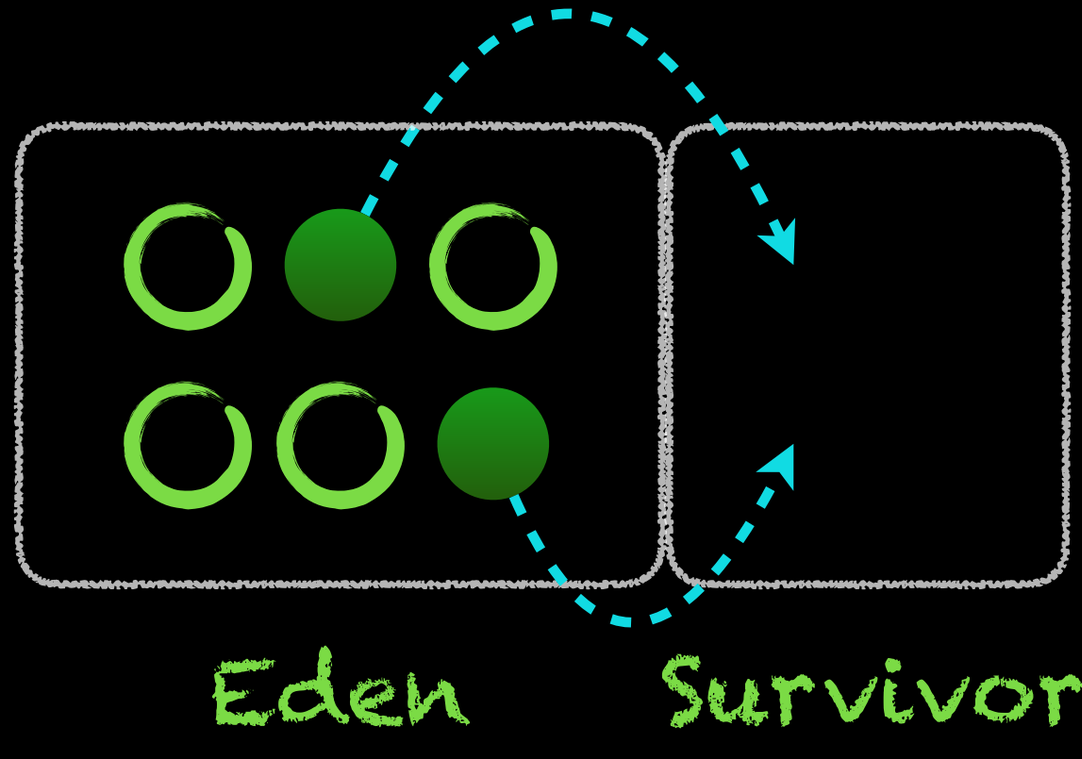
Garbage Collection - Reclamation via Mark-Sweep



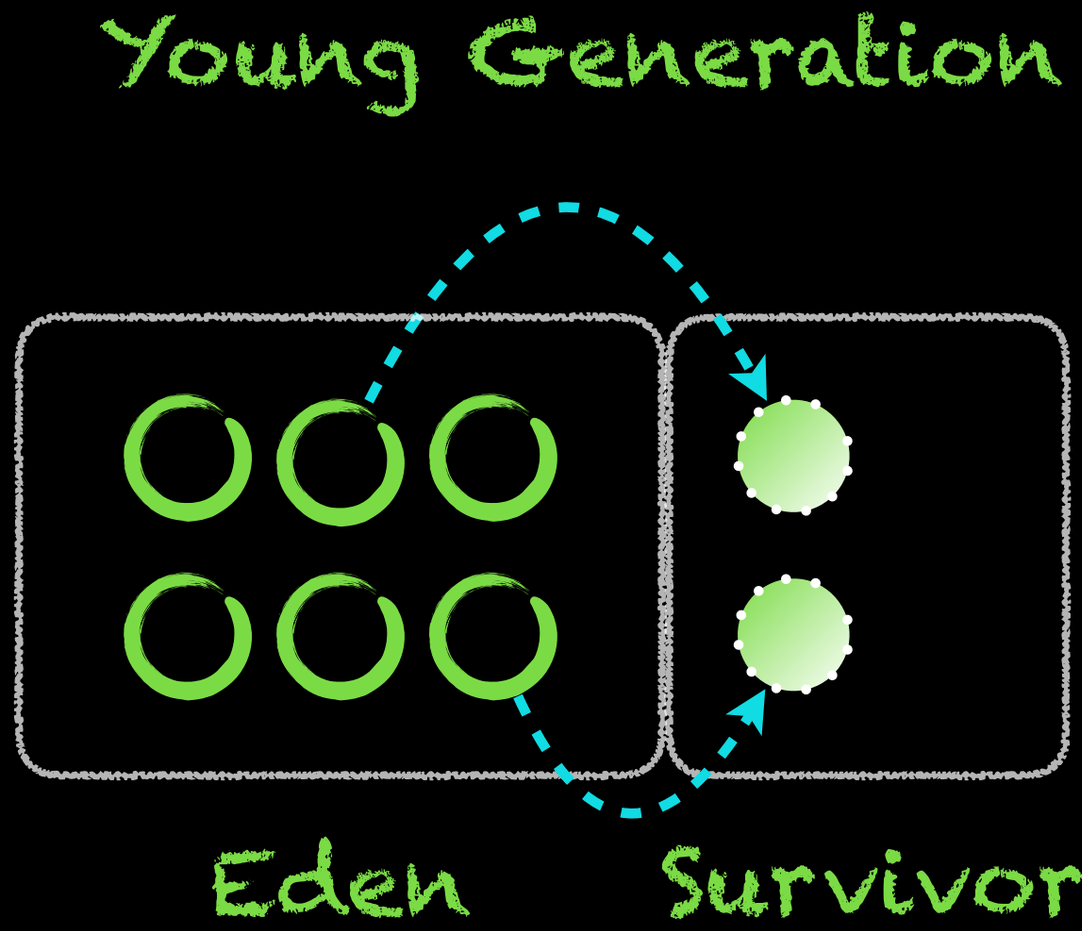
Old Generation

Garbage Collection - Reclamation via Scavenging

Young Generation

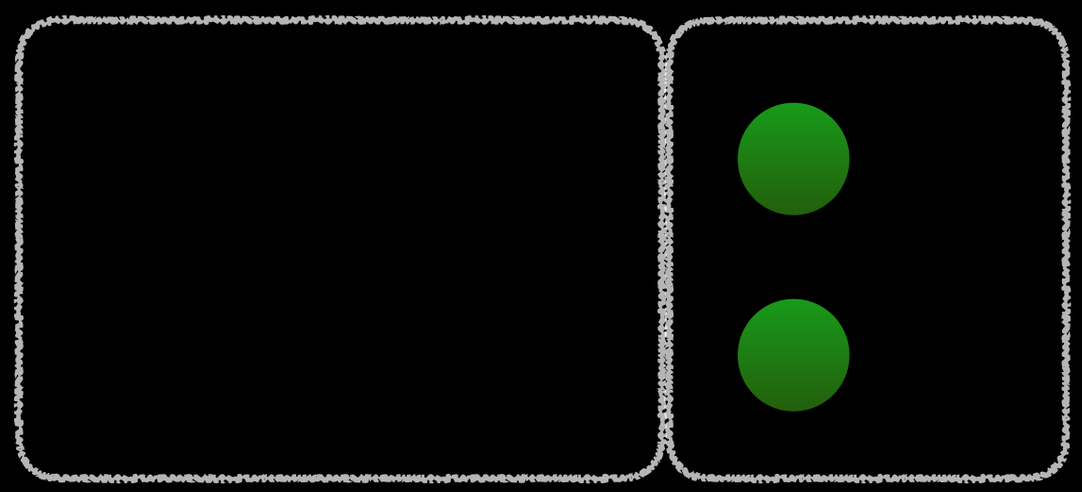


Garbage Collection - Reclamation via Scavenging



Garbage Collection - Reclamation via Scavenging

Young Generation



Garbage Collection In OpenJDK HotSpot

The Throughput Collector -

- Young Collections - Parallel Scavenge
- Old Collections - Parallel Mark-Compact

Garbage Collection In OpenJDK HotSpot

CMS Collector -

- Young Collections - Parallel New (similar to Parallel Scavenge)
- Old Collections - (Mostly Concurrent) Mark-Sweep
- Fallback Collections - Serial Mark-Sweep-Compact

Garbage Collection In OpenJDK HotSpot

G1 Collector -

- Young and Mixed Collections - Compaction via Copying (similar to Parallel Scavenge)
- Fallback Collections - Serial Mark-Sweep-Compact

What's The #1 Contributor To A GC Pause Duration?

Copying Costs!

Garbage Collection - Tuning Recommendations

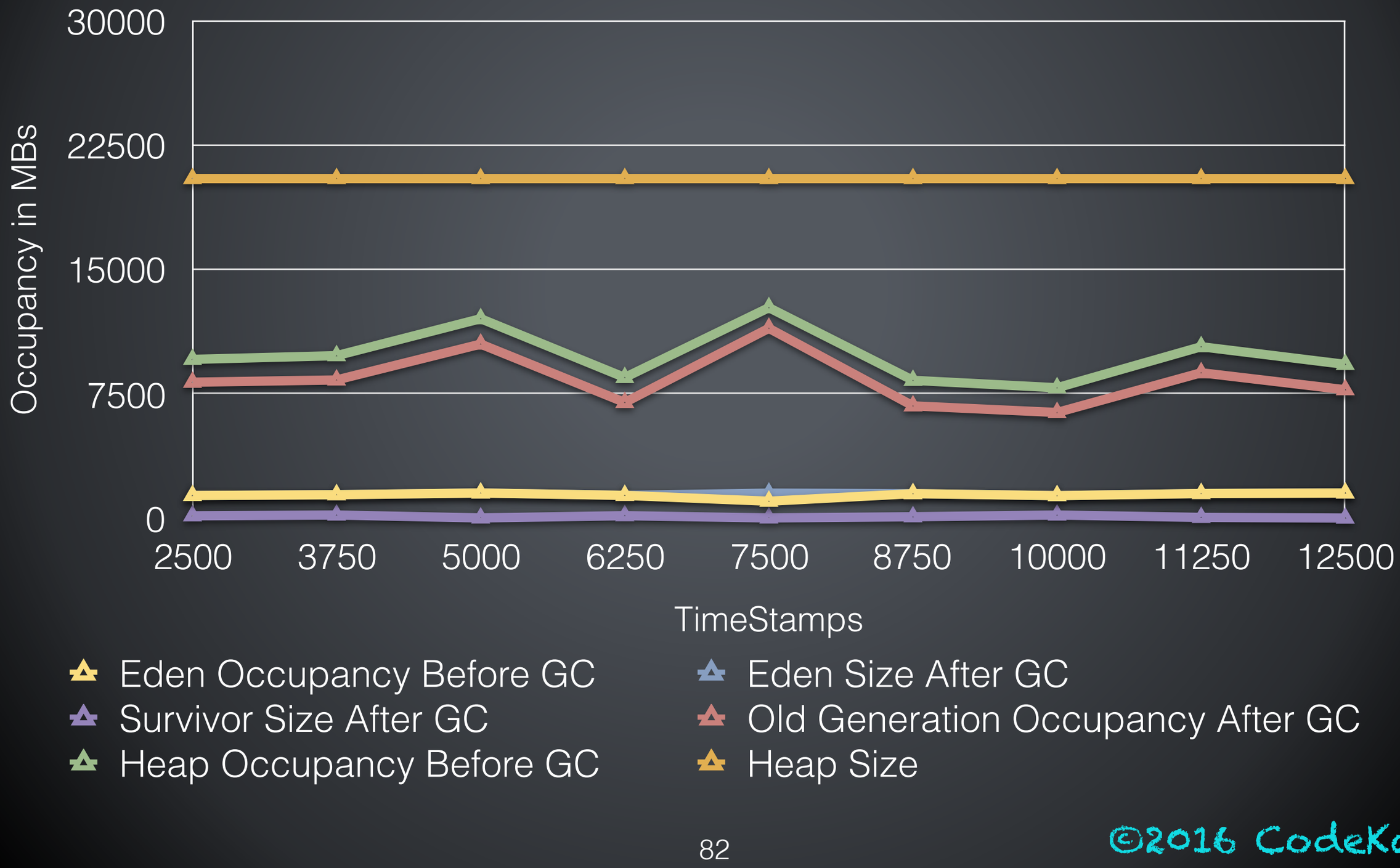
- Size generations keeping your application's object longevity and size in mind.
- short-lived; medium-lived; long-lived transient + permanent set.
- **premature promotions are a big problem!**

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



What Are The Contributors To GC Pause Frequency?

Allocation Rate and Promotion Rate

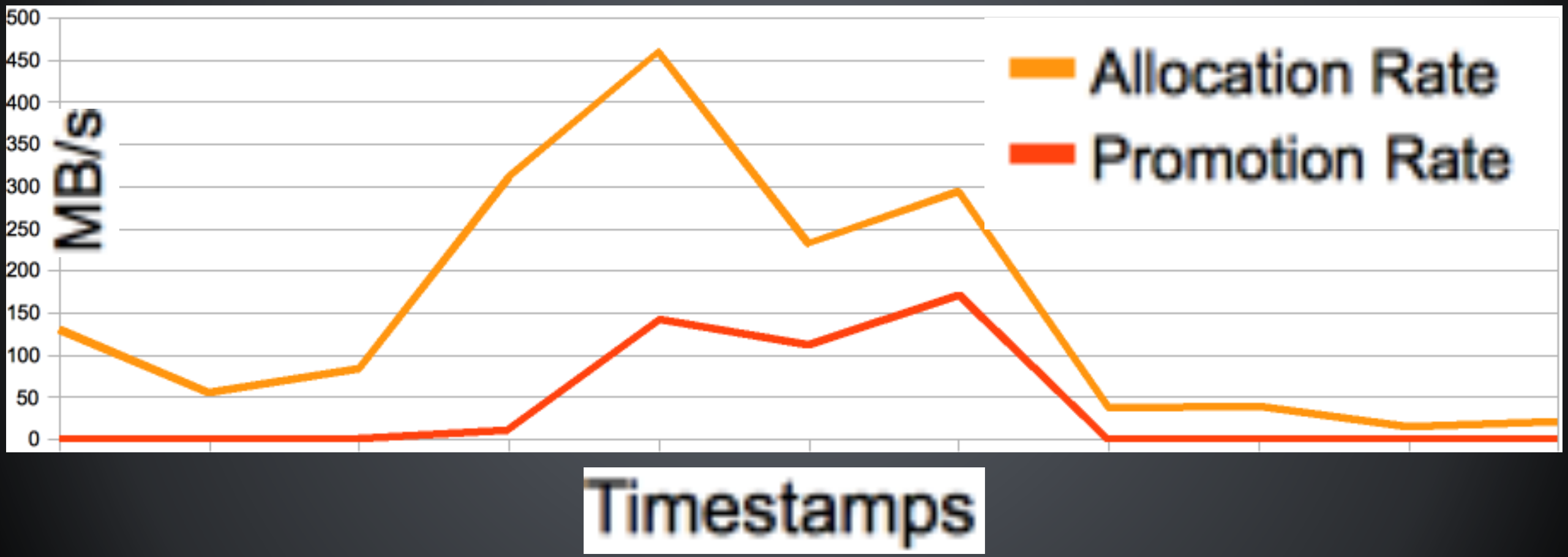
Garbage Collection - Tuning Recommendations

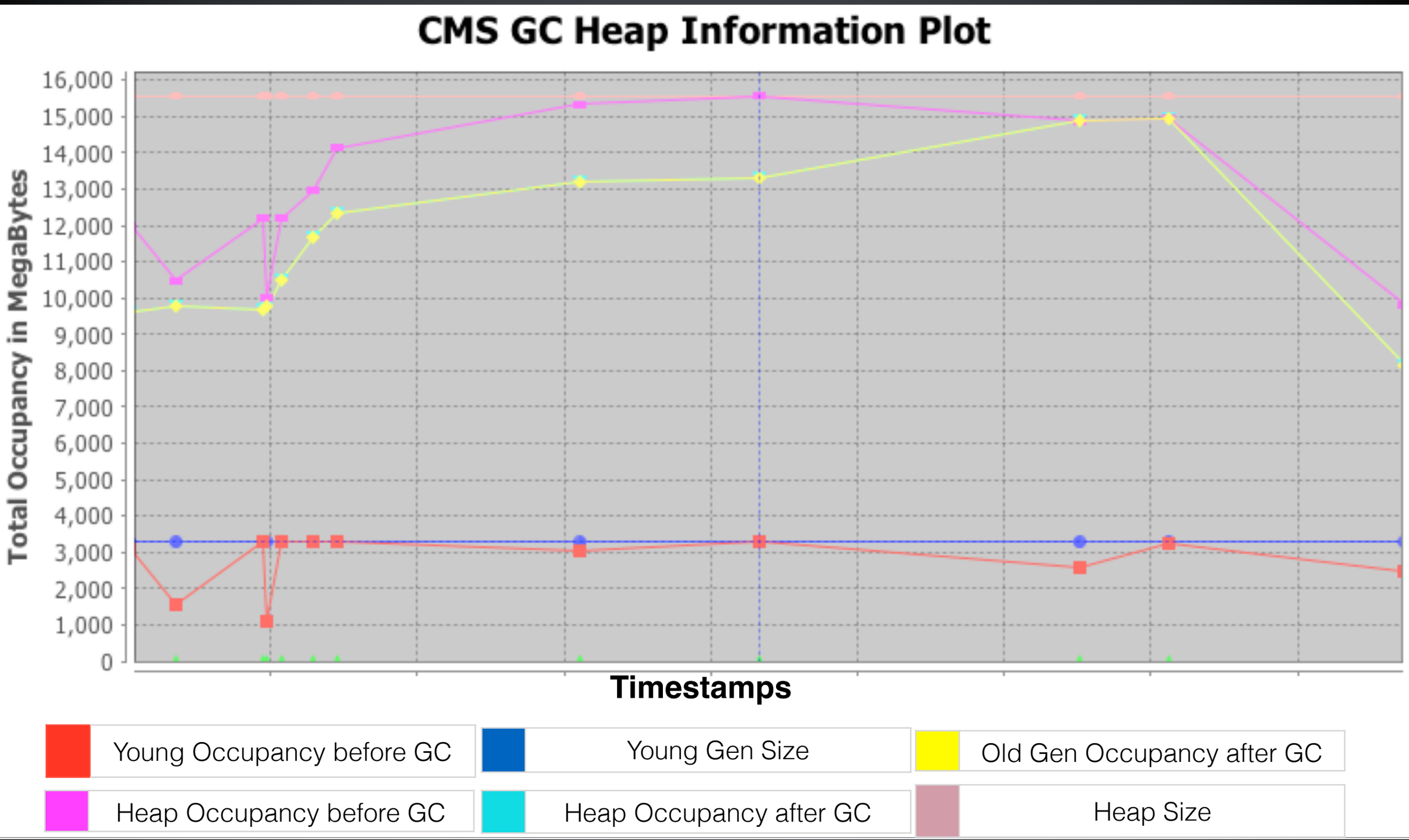
- The faster the generation gets “filled”; the sooner a GC is triggered.

- **Premature promotions are a big problem!**

  - Size your generations and age your objects appropriately.

Plot Allocation & Promotion Rates





Allocation Rate

38.692: [GC (Allocation Failure) [PSYoungGen: 917486K->131054K(858112K)] 1004844K->224424K(2955264K), 0.0827570 secs] [Times: user=0.61 sys=0.00, real=0.08 secs]

51.013: [GC (Allocation Failure) [PSYoungGen: 858094K->50272K(777728K)] 951464K->239014K(2874880K), 0.1414536 secs] [Times: user=0.70 sys=0.07, real=0.14 secs]

Allocation Rate

38.692: [GC (Allocation Failure) [PSYoungGen: 917486K->131054K(858112K)] 1004844K->224424K(2955264K), 0.0827570 secs] [Times: user=0.61 sys=0.00, real=0.08 secs]

51.013: [GC (Allocation Failure) [PSYoungGen: 858094K->50272K(777728K)] 951464K->239014K(2874880K), 0.1414536 secs] [Times: user=0.70 sys=0.07, real=0.14 secs]

Allocation rate = (858094K-131054K)/()

Allocation Rate

38.692: [GC (Allocation Failure) [PSYoungGen: 917486K->131054K(858112K)] 1004844K->224424K(2955264K), 0.0827570 secs] [Times: user=0.61 sys=0.00, real=0.08 secs]

51.013: [GC (Allocation Failure) [PSYoungGen: 858094K->50272K(777728K)] 951464K->239014K(2874880K), 0.1414536 secs] [Times: user=0.70 sys=0.07, real=0.14 secs]

Allocation rate = (858094K-131054K)/(51.013-38.692) = **57.6MB/s**

Promotion Rate

38.692: [GC (Allocation Failure) [PSYoungGen: 917486K->131054K(858112K)] 1004844K->224424K(2955264K), 0.0827570 secs] [Times: user=0.61 sys=0.00, real=0.08 secs]

51.013: [GC (Allocation Failure) [PSYoungGen: 858094K->50272K(777728K)] 951464K->239014K(2874880K), 0.1414536 secs] [Times: user=0.70 sys=0.07, real=0.14 secs]

Promotion rate = ((239014K - 50272K) – ()

Promotion Rate

38.692: [GC (Allocation Failure) [PSYoungGen: 917486K->131054K(858112K)] 1004844K->224424K(2955264K), 0.0827570 secs] [Times: user=0.61 sys=0.00, real=0.08 secs]

51.013: [GC (Allocation Failure) [PSYoungGen: 858094K->50272K(777728K)] 951464K->239014K(2874880K), 0.1414536 secs] [Times: user=0.70 sys=0.07, real=0.14 secs]

Promotion rate = ((239014K - 50272K) – (951464K - 858094K))/()

Promotion Rate

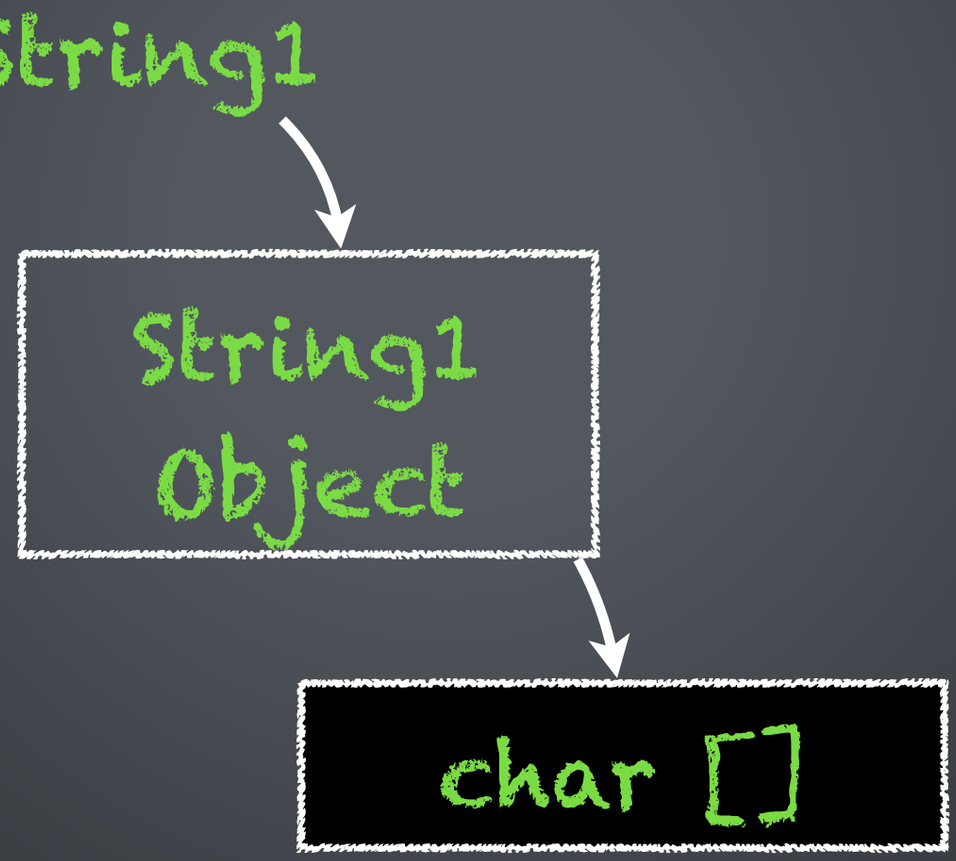
38.692: [GC (Allocation Failure) [PSYoungGen: 917486K->131054K(858112K)] 1004844K->224424K(2955264K), 0.0827570 secs] [Times: user=0.61 sys=0.00, real=0.08 secs]

51.013: [GC (Allocation Failure) [PSYoungGen: 858094K->50272K(777728K)] 951464K->239014K(2874880K), 0.1414536 secs] [Times: user=0.70 sys=0.07, real=0.14 secs]

Promotion rate = ((239014K - 50272K) – (951464K - 858094K))/(51.013 - 38.692) = **7.56MB/s**

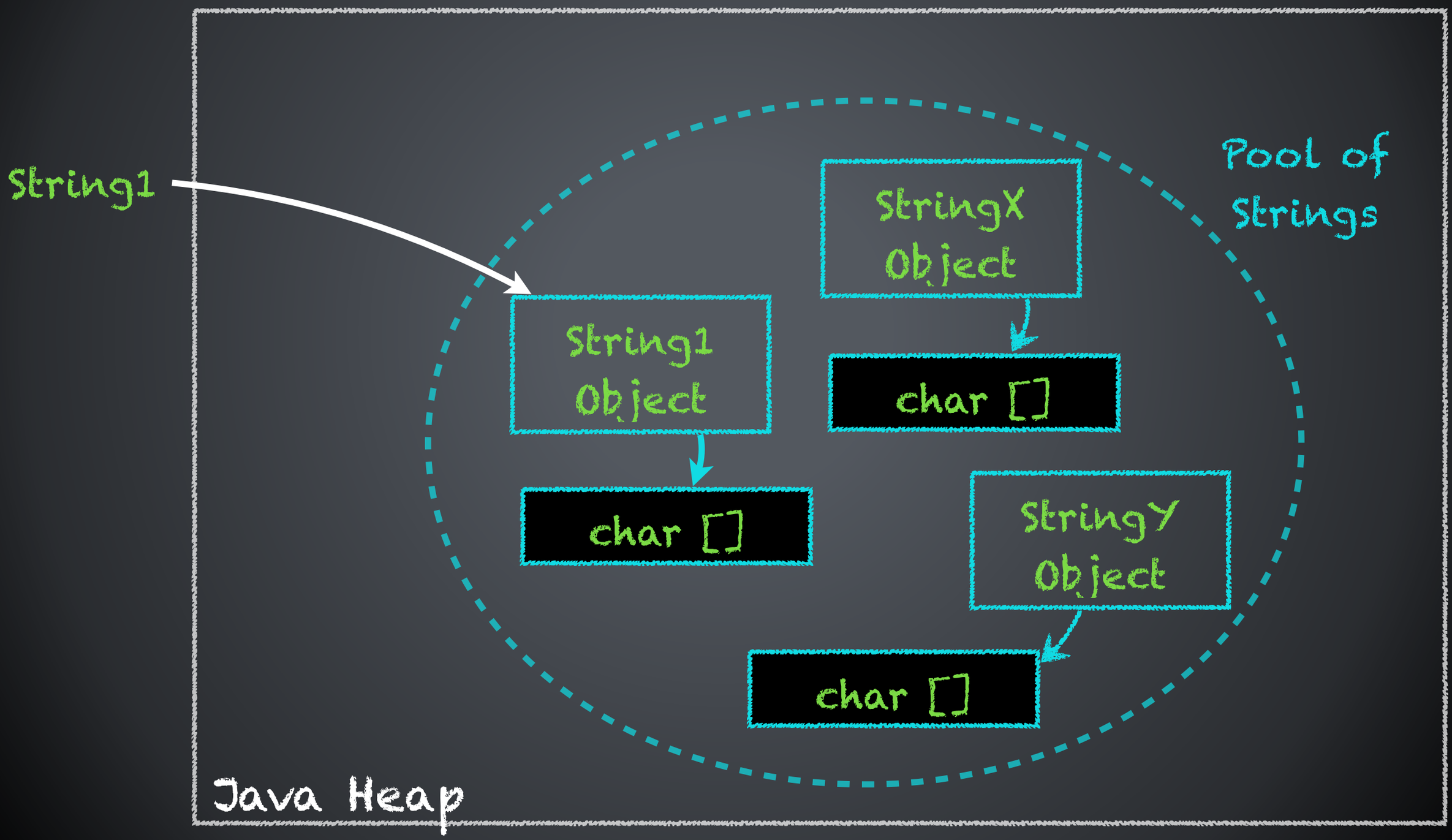
String Interning and Deduplication

Java String Object



What If String1 Is Interned?

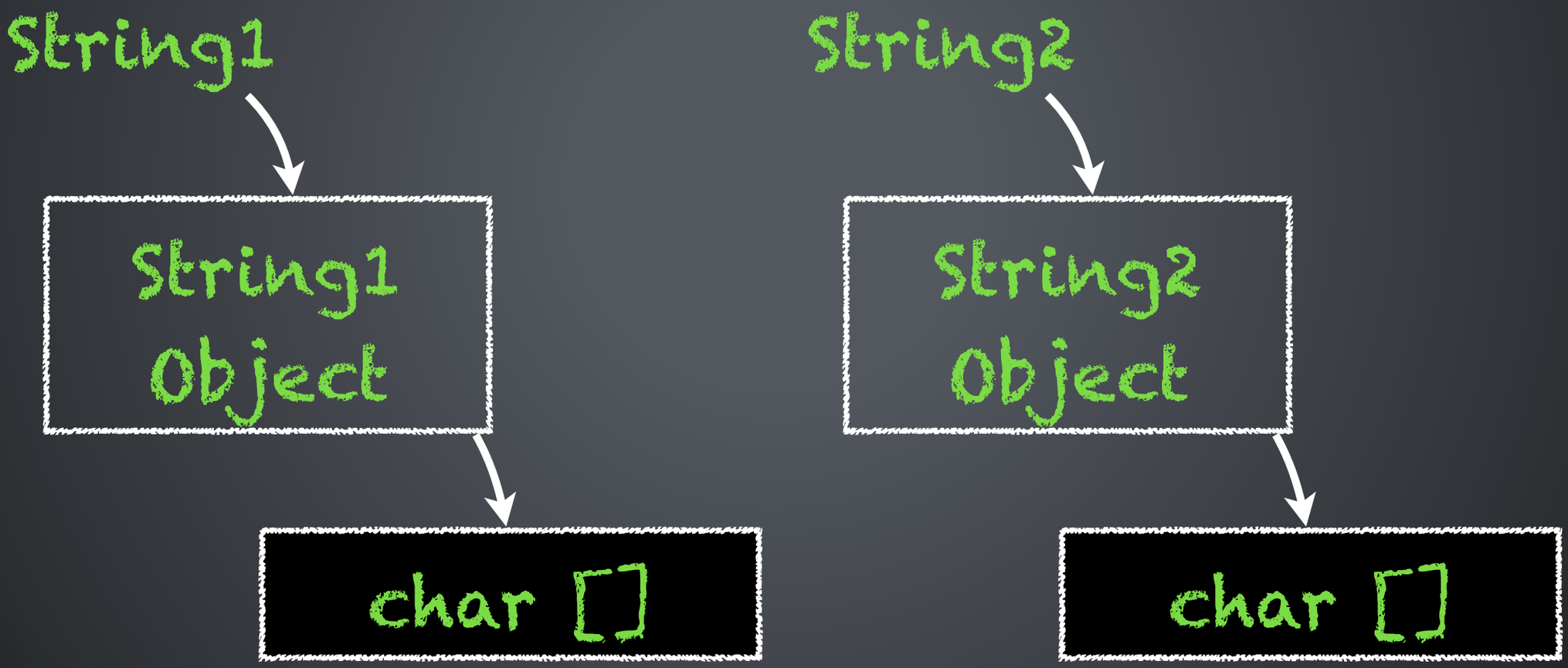
Java Interned String Object



Java String Intern

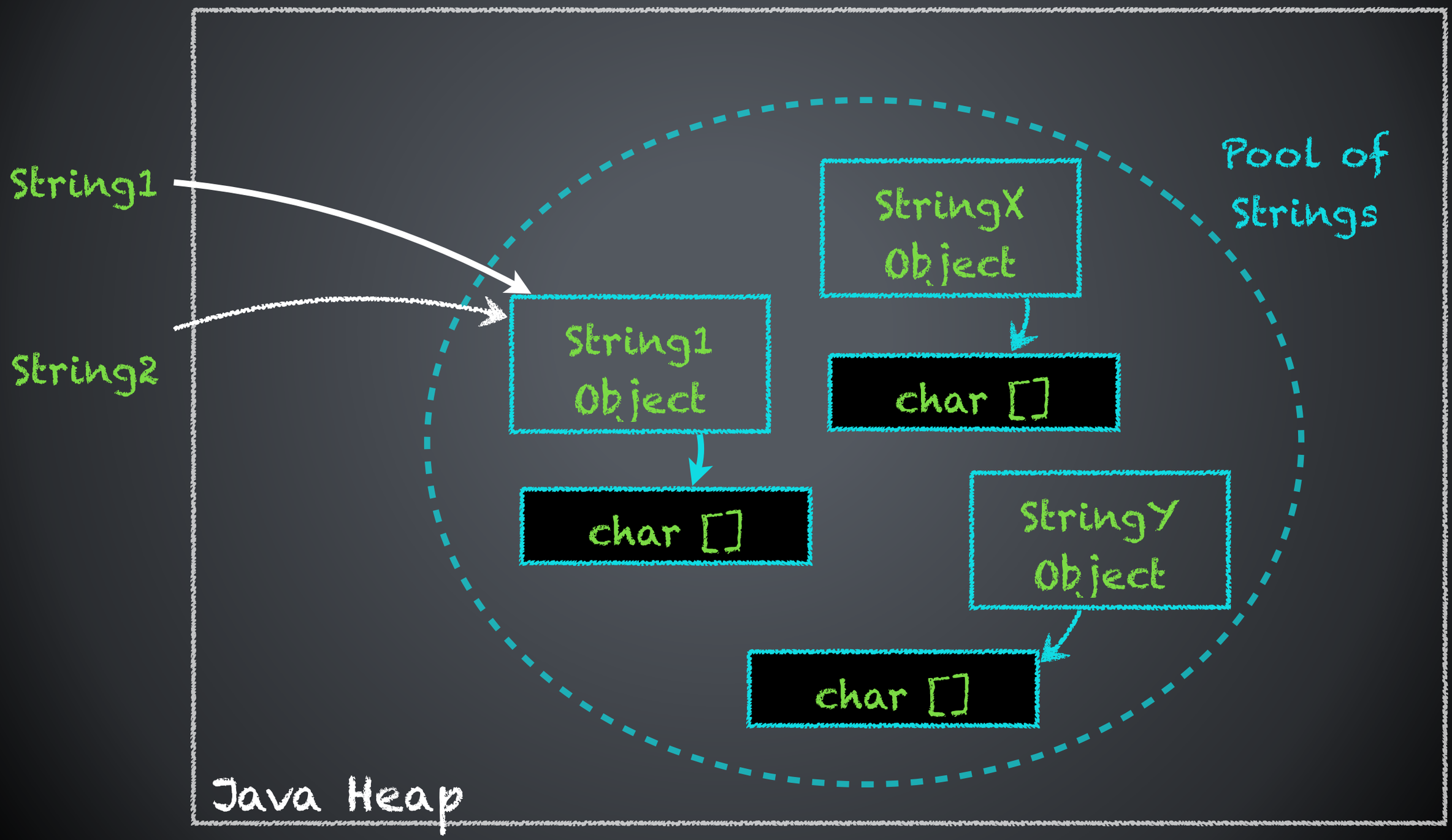
- unique + constant pool of strings
  - hashtable
    - default = 60013 (on LP64)
    - if pool already contain string.equals(String1)?
      - return string from the string pool
      - else, add String1 to the pool; return its reference

Java Interned String Objects



If String1.intern() ==
String2.Intern() Then
String1.equals(String2) Is True.

Java Interned String Objects

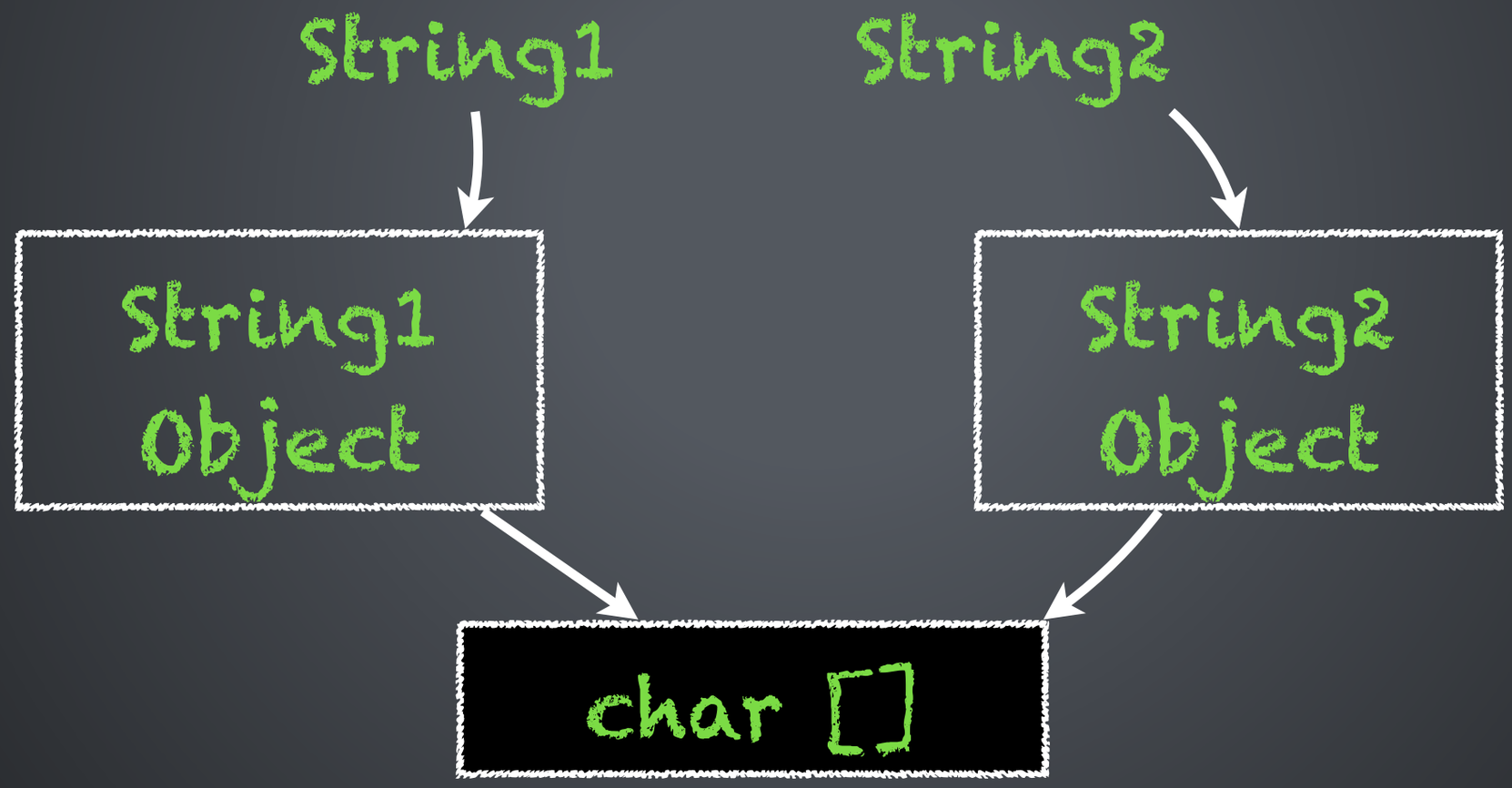


What If String1.equals(String2)

And Both Are Not Interned*?

*And You Are Using The G1 Collector

Java String Deduplication (G1)



Further Reading

- Mark Price's talk: https://qconlondon.com/presentation/hot-code-faster-code-addressing-jvm-warm
- https://wiki.openjdk.java.net/display/HotSpot/Server+Compiler+Inlining+Messages
- https://wiki.openjdk.java.net/display/HotSpot/EscapeAnalysis
- Compressed Class Pointers: https://youtu.be/AHtfza2Tkt0?t=754
- String Deduplication: http://openjdk.java.net/jeps/192
- Perm Gen Removal: http://www.infoq.com/articles/Java-PERMGEN-Removed

Appendix

Real-World Issues & Workarounds - JDK 8 update 45+

https://gist.github.com/mileema/6fb667a215e95919242f

https://github.com/facebook/presto/commit/91e1b3bb6bbfffc62401025a24231cd388992d7c