JAVA GARBAGE COLLECTION - THE BASICS

CHANDRA GUNTUR

@CGuntur

http://cguntur.me

About

- Java Champion
- JCP Executive Committee Rep. for BNY Mellon
- Programming in Java since 1998
- JUG Leader @ NYJavaSIG and NJJavaSIG
- Ardent blogger and tweeter
- Saganist (with a ‘g’ not a ‘t’) 😈

What we cover

- Overview and history of garbage collection
- Types of garbage collection
- General patterns
- Deeper dive into Generational GCs
- The newer Garbage First GC
- Recently added garbage collectors

- Overview and history of garbage collection
- Types of garbage collection
- General patterns
- Deeper dive into Generational GCs
- The newer Garbage First GC
- Recently added garbage collectors

What is Garbage Collection?

What is Garbage Collection?

As a broad definition, garbage collection is:

What is Garbage Collection?

As a broad definition, garbage collection is:

- looking up a managed memory area
- identify objects in-use as live objects
- mark objects no longer used as garbage

What is Garbage Collection?

As a broad definition, garbage collection is:

- looking up a managed memory area
- identify objects in-use as live objects
- mark objects no longer used as garbage
- [occasionally] reclaim memory by deleting garbage

What is Garbage Collection?

As a broad definition, garbage collection is:

- looking up a managed memory area
- identify objects in-use as live objects
- mark objects no longer used as garbage
- [occasionally] reclaim memory by deleting garbage
- [occasionally] compact memory by defragmenting

History

First garbage collection process: Lisp

When: 1959

Author: John MacCarthy

His other contributions:

- Author of Lisp
- Major contributor to ALGOL
- A founding father in the AI space

- Overview and history of garbage collection
- Types of garbage collection
- General patterns
- Deeper dive into Generational GCs
- The newer Garbage First GC
- Recently added garbage collectors

Classifications

Focus on the four classifications:

1. based on collection type
2. based on object marking
3. based on execution volume (run interval)
4. based on space compaction

1. Based on collection type

Classification

- Serial collection
- Parallel collection
- Concurrent collection

based on collection type

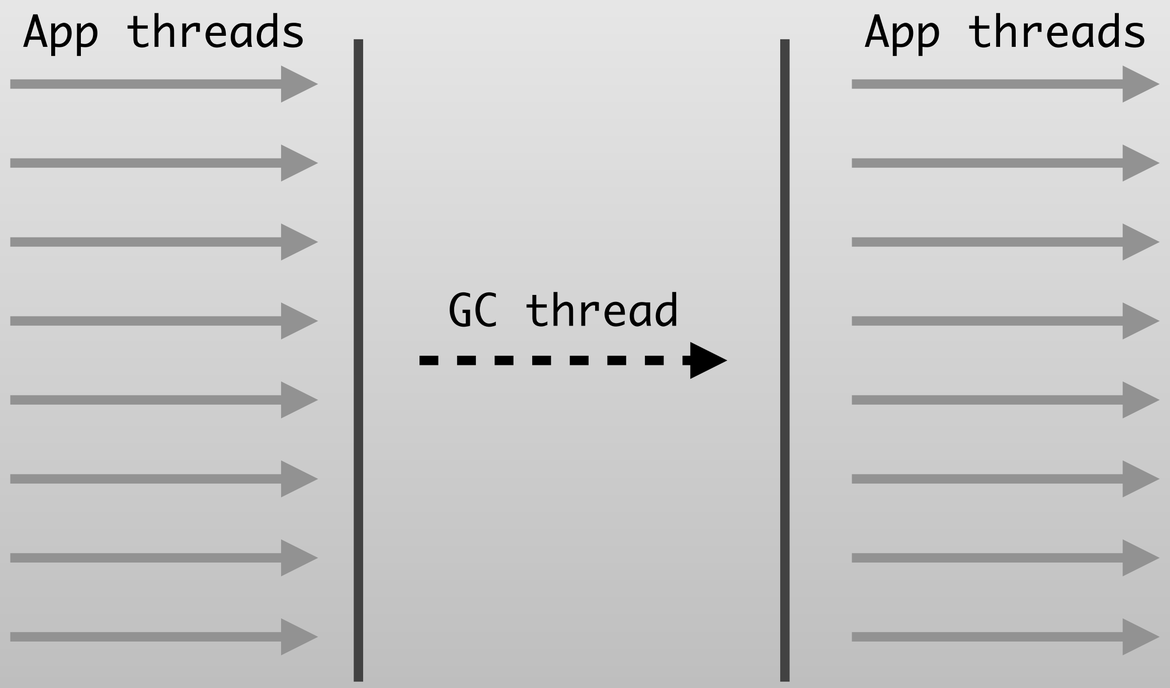
based on object marking

based on execution volume (run interval)

based on space compaction

Serial Collection

Classification - based on collection type



based on collection type

based on object marking

based on execution volume (run interval)

based on space compaction

Serial Collection

Classification - based on collection type

- Single thread collects
- Halts all application threads
- STOP-THE-WORLD collector 

- Real-life analogy (Mail box):
  - Stop incoming mail
  - Single person checks current mails

based on collection type

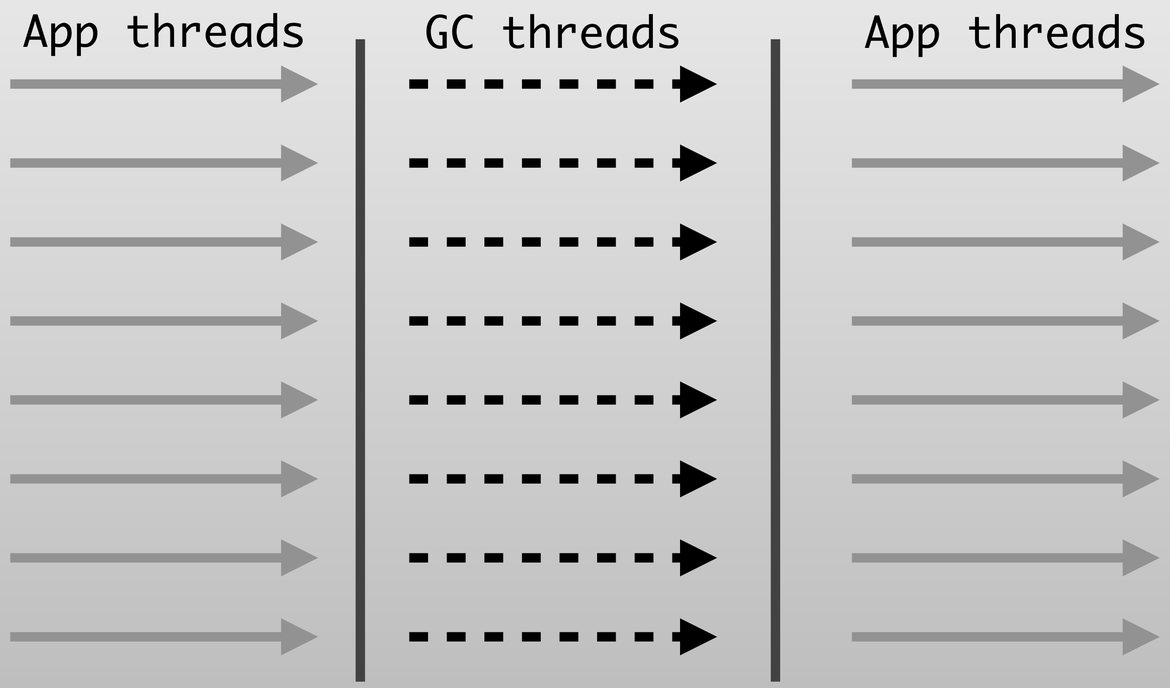
based on object marking

based on execution volume (run interval)

based on space compaction

Parallel Collection

Classification - based on collection type



based on collection type

based on object marking

based on execution volume (run interval)

based on space compaction

Parallel Collection

Classification - based on collection type

- Many threads collects
- Halts all application threads
- STOP-THE-WORLD collector

- Real-life analogy (Mail box):
  - Stop incoming mail
  - Many people check current mails

based on collection type

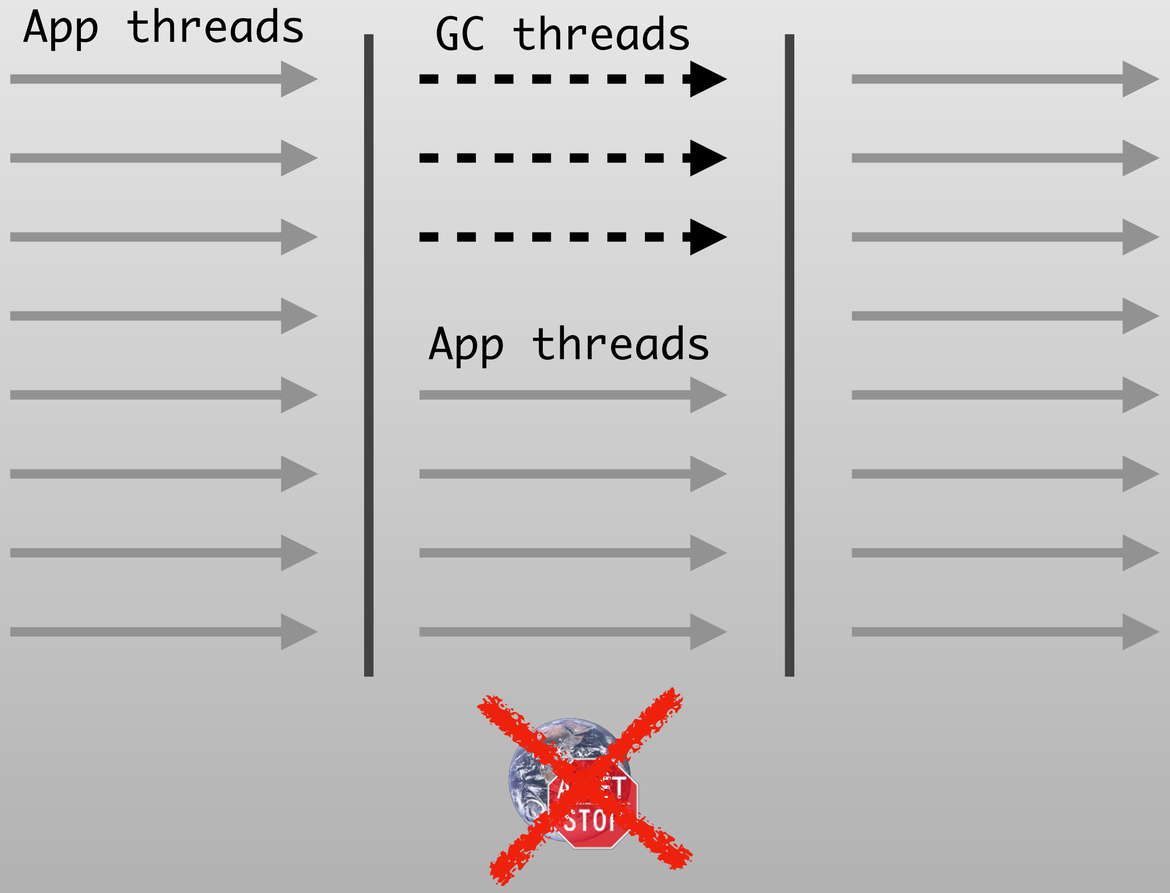
based on object marking

based on execution volume (run interval)

based on space compaction

Concurrent Collection

Classification - based on collection type



based on collection type

based on object marking

based on execution volume (run interval)

based on space compaction

Concurrent Collection

Classification - based on collection type

- Many threads collects
- Application threads can continue
- CONCURRENT collector

- Real-life analogy (Mail box):
  - Do not stop incoming mail
  - One or more people check current mails

based on collection type

based on object marking

based on execution volume (run interval)

based on space compaction

2. Based on object marking

Classification

- Precise collection
- Conservative collection

based on collection type

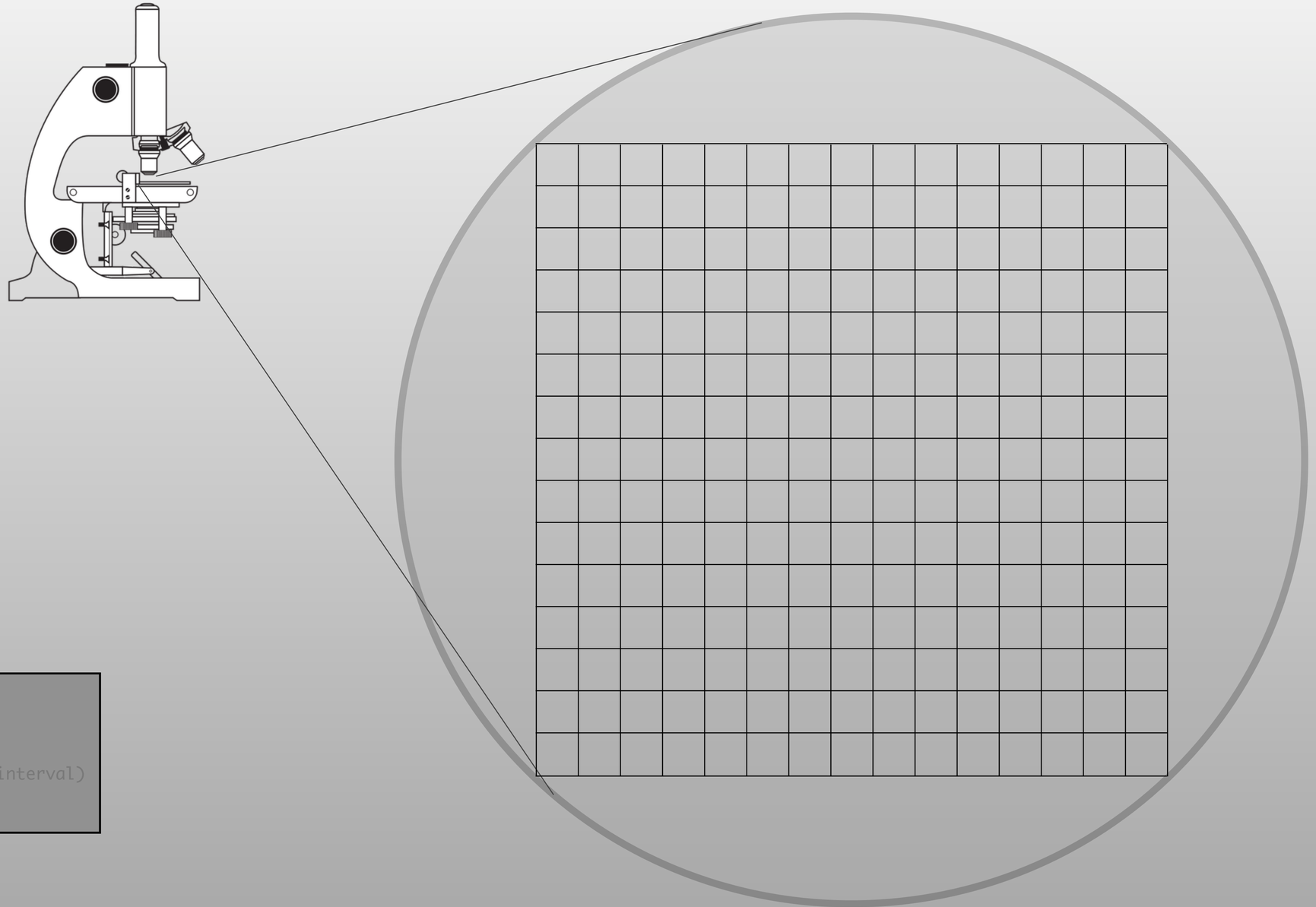
based on object marking

based on execution volume (run interval)

based on space compaction

Precise collection

Classification - based on object marking



based on collection type

based on object marking

based on execution volume (run interval)

based on space compaction

Precise collection

Classification - based on object marking

- All objects identified
- On finish, objects are live or garbage.
- Typically slow
- Aims for thoroughness not speed
- Real-life analogy (Mail box):
  - check all mail
  - discard fliers and junk mail

based on collection type

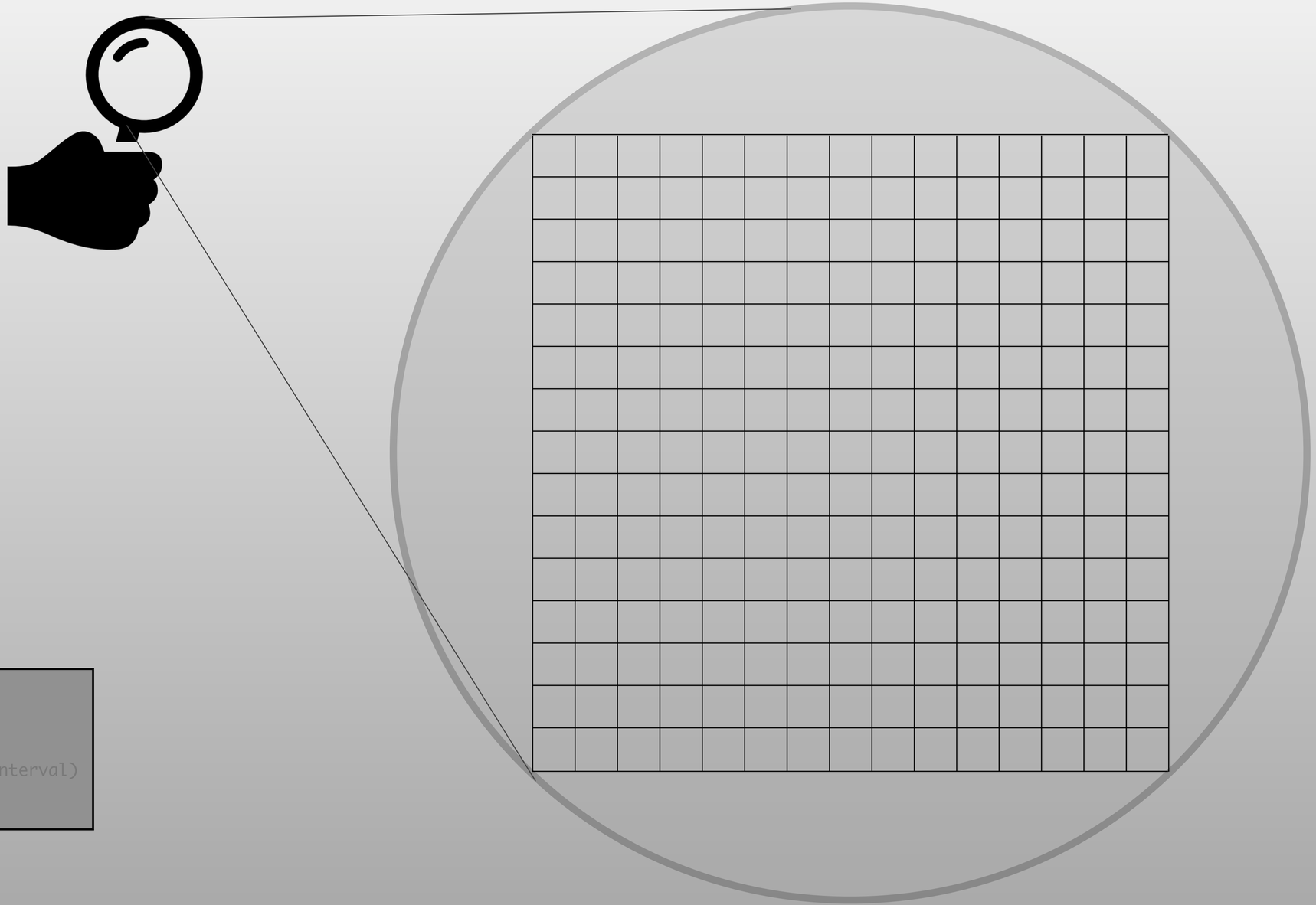
based on object marking

based on execution volume (run interval)

based on space compaction

Conservative collection

Classification - based on object marking



based on collection type

based on object marking

based on execution volume (run interval)

based on space compaction

Conservative collection

Classification - based on object marking

- Not all objects identified
- If not identified, object considered live.
- Typically fast
- Aims for speed not thoroughness
- Real-life analogy (Mail box):
  - check mail, find fliers and discard
  - assume all addressed mail is not junk

based on collection type

based on object marking

based on execution volume (run interval)

based on space compaction

3. Based on execution volume

Classification

- All-at-once collection
- Incremental collection

based on collection type

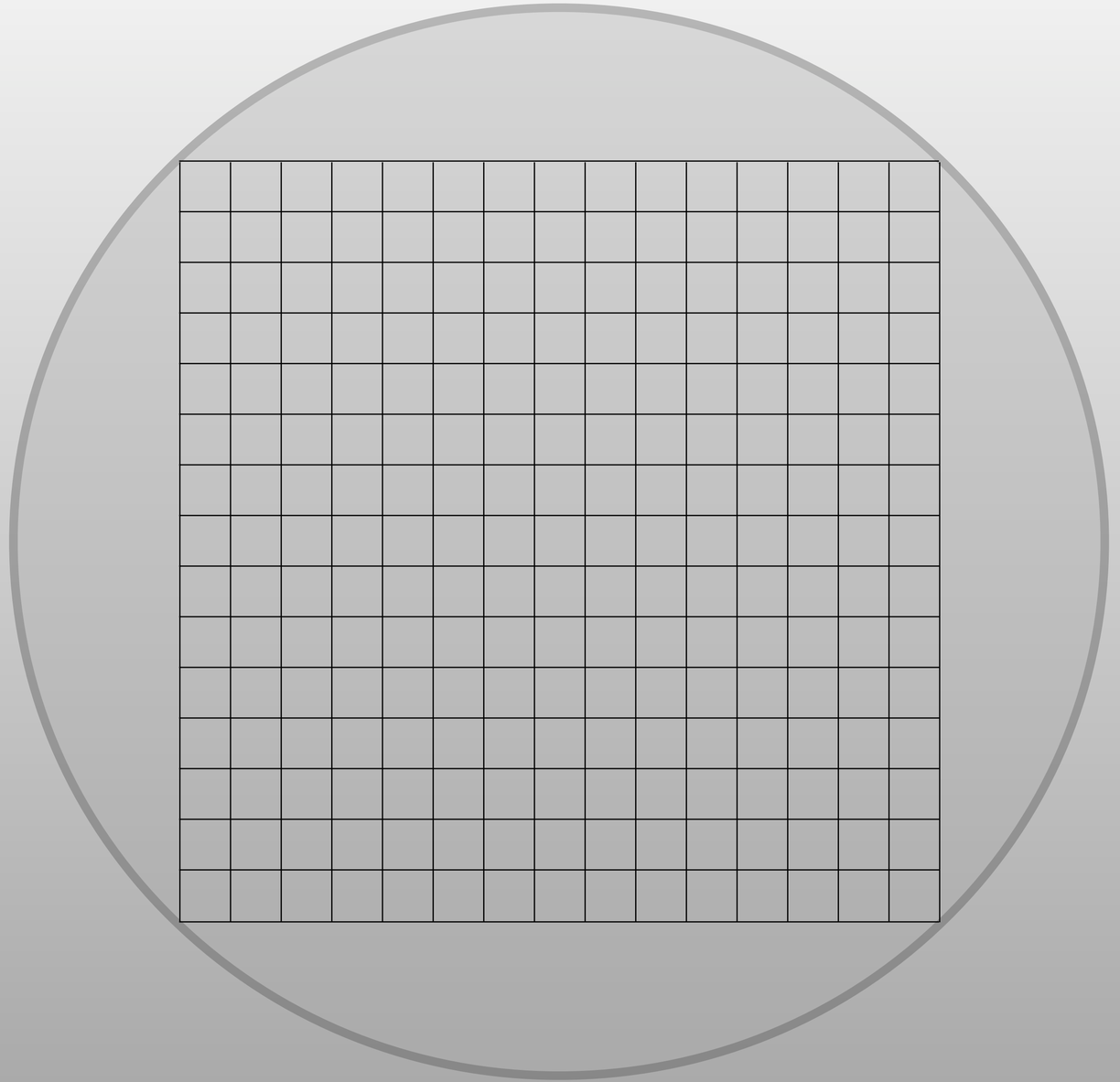
based on object marking

based on execution volume (run interval)

based on space compaction

All-at-once collection

Classification - based on execution volume



based on collection type

based on object marking

based on execution volume (run interval)

based on space compaction

All-at-once collection

Classification - based on execution volume

- Entire collection at one go
- Real-life analogy (Mail box):
  - check all current mail at once

based on collection type

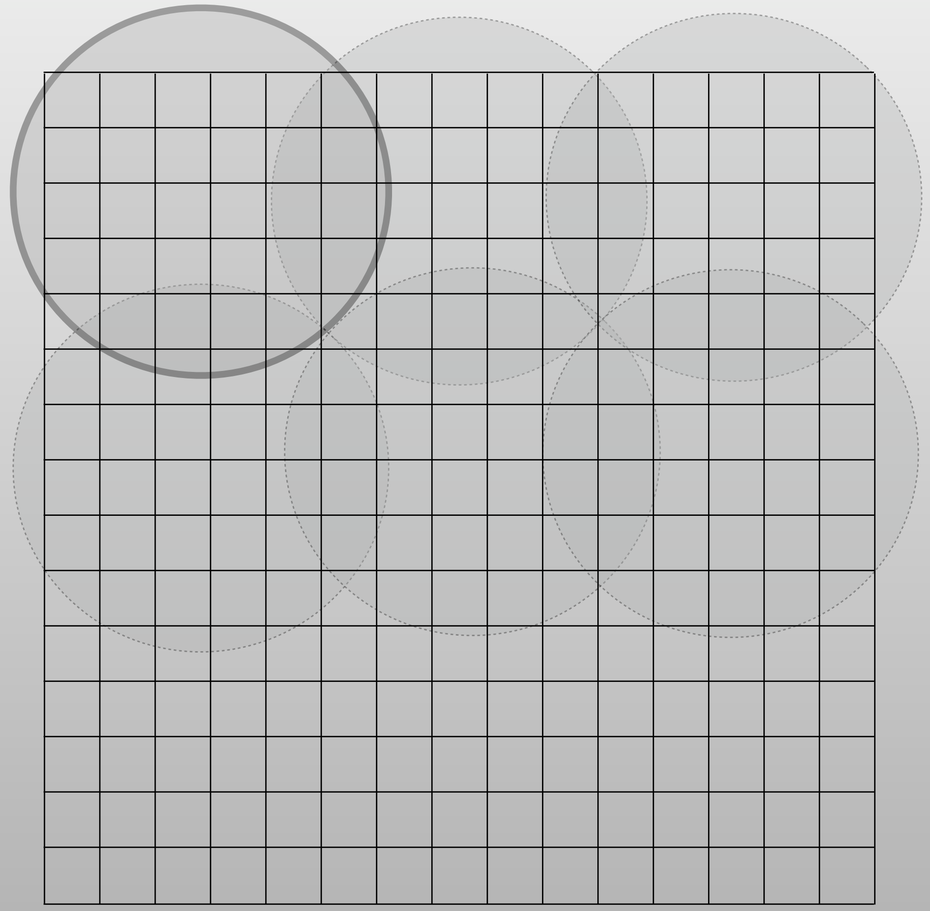
based on object marking

based on execution volume (run interval)

based on space compaction

Incremental collection

Classification - based on execution volume



based on collection type

based on object marking

based on execution volume (run interval)

based on space compaction

Incremental collection

Classification - based on execution volume

- Slices of memory area collected
- Real-life analogy (Mail box):
  - pick a few mails at a time and check

based on collection type

based on object marking

based on execution volume (run interval)

based on space compaction

4. Based on space compaction

Classification

- Non-moving collection
- Moving collection

based on collection type

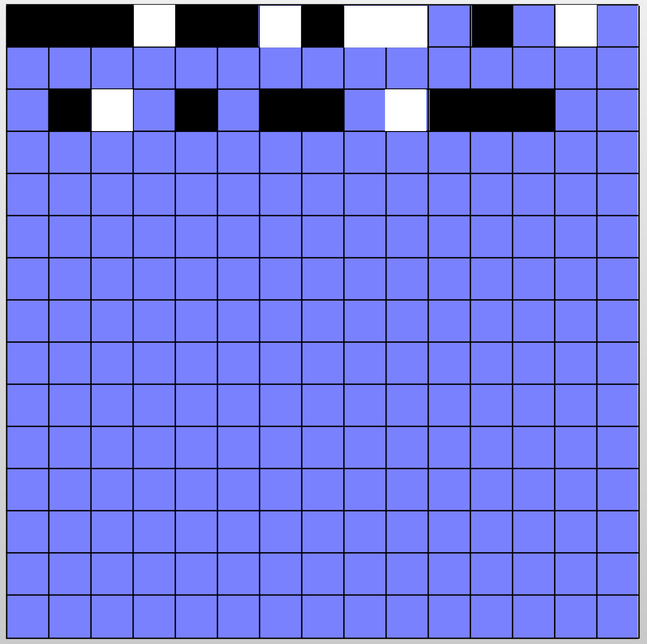
based on object marking

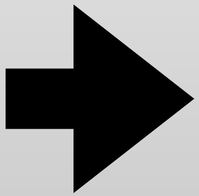
based on execution volume (run interval)

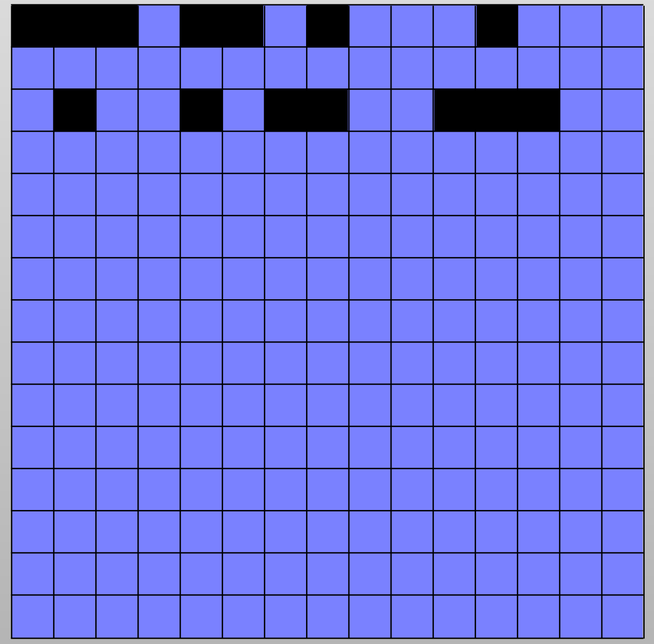
based on space compaction

Non-moving collection

Classification - based on space compaction







White = Dead objects

Black = Live objects

based on collection type

based on object marking

based on execution volume (run interval)

based on space compaction

Non-moving collection

Classification - based on space compaction

- Live objects not moved
- Live objects stay fragmented
- Aims for speed not contiguous space
- Real-life analogy (Mail box):
  - remove fliers and junk mail
  - leave other mail as is, still cluttered

based on collection type

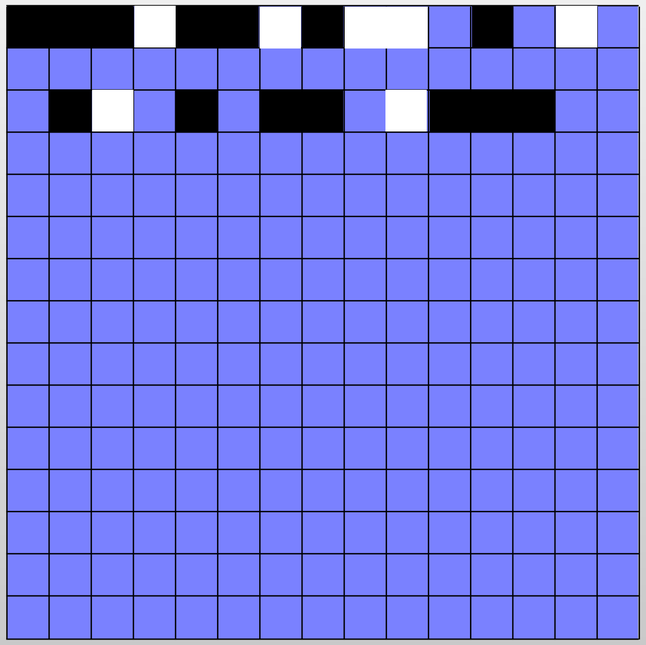
based on object marking

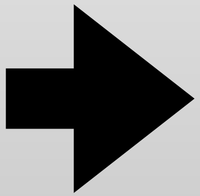
based on execution volume (run interval)

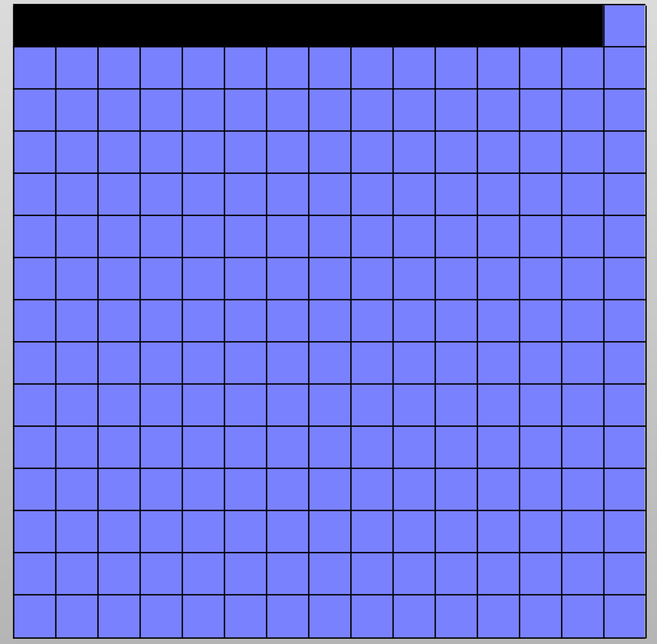
based on space compaction

Moving collection

Classification - based on space compaction







White = Dead objects

Black = Live objects

based on collection type

based on object marking

based on execution volume (run interval)

based on space compaction

Moving collection

Classification - based on space compaction

- Live objects moved to new location
- Live objects contiguously placed
- Aims for contiguous space not speed
- Real-life analogy (Mail box):
  - remove junk mail
  - stack remaining mail to make space for new

based on collection type

based on object marking

based on execution volume (run interval)

based on space compaction

- Overview and history of garbage collection
- Types of garbage collection
- General patterns
- Deeper dive into Generational GCs
- The newer Garbage First GC
- Recently added garbage collectors

Definitions

- What is a Root Object?
- What is tri-color marking?

What is Garbage Collection?

Garbage collection - Recap

As a broad definition, garbage collection is:

- looking up a managed memory area
- identify objects in-use as live objects
- mark objects no longer used as garbage
- [occasionally] reclaim memory by deleting garbage
- [occasionally] compact memory by defragmenting

What is Garbage Collection?

Garbage collection - Recap

As a broad definition, garbage collection is:

- looking up a managed memory area
- identify objects in-use as live objects
- mark objects no longer used as garbage
- [occasionally] reclaim memory by deleting garbage
- [occasionally] compact memory by defragmenting

Definition

Root object:

Object accessible outside the heap

- System class
- Local variable
- Thread
- Many more …

https://www.yourkit.com/docs/java/help/gc_roots.jsp

Definition

Tri-color marking:

Painting objects with colors

- Gray = Object needs evaluation
- Black = Object evaluated as live
- White = Object evaluated as garbage

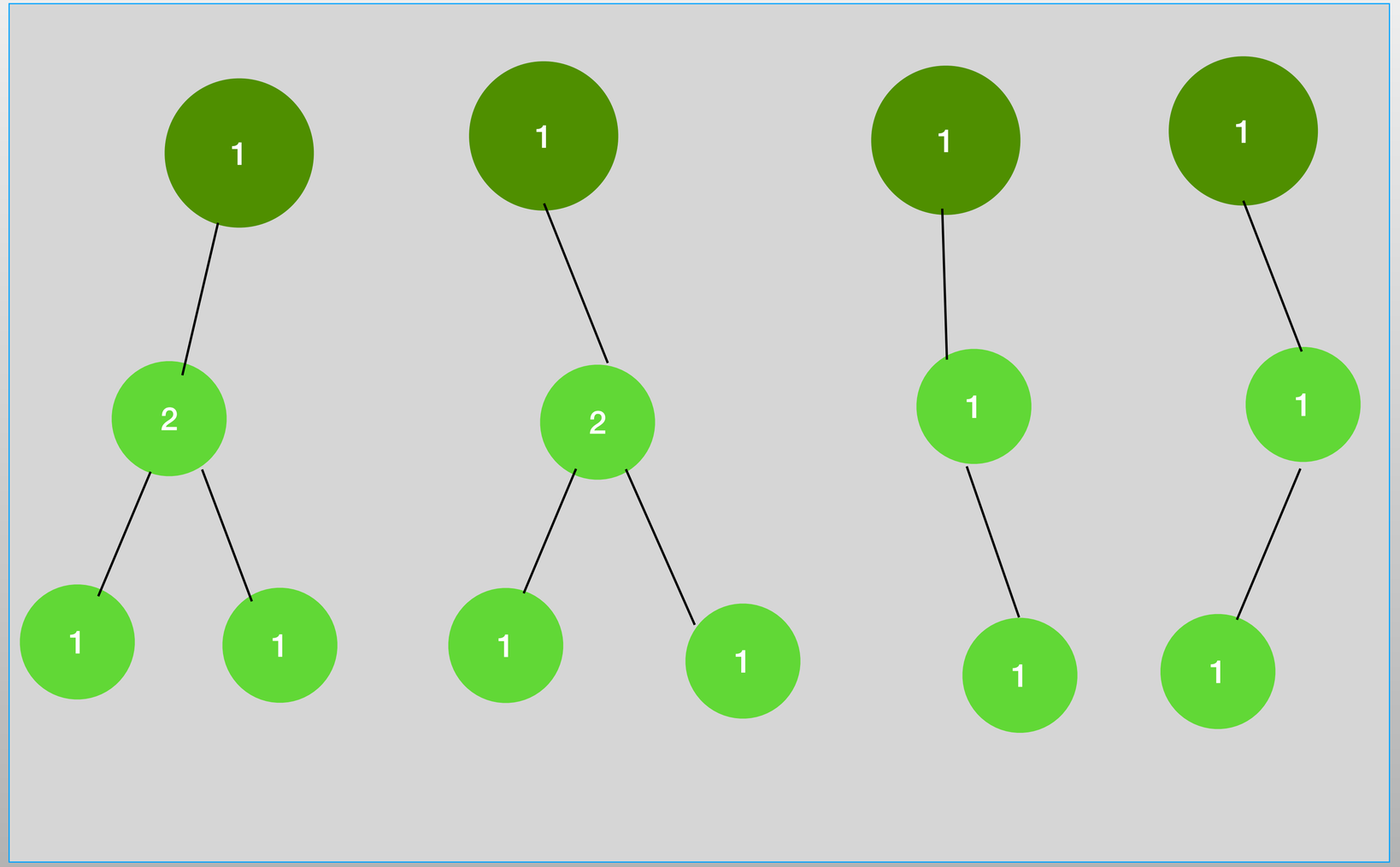
Patterns of GC

Three common patterns of garbage collection:

1. ~~Reference Counting~~ Pattern
2. Mark/Sweep[/Compacting] Pattern
3. Copying Pattern

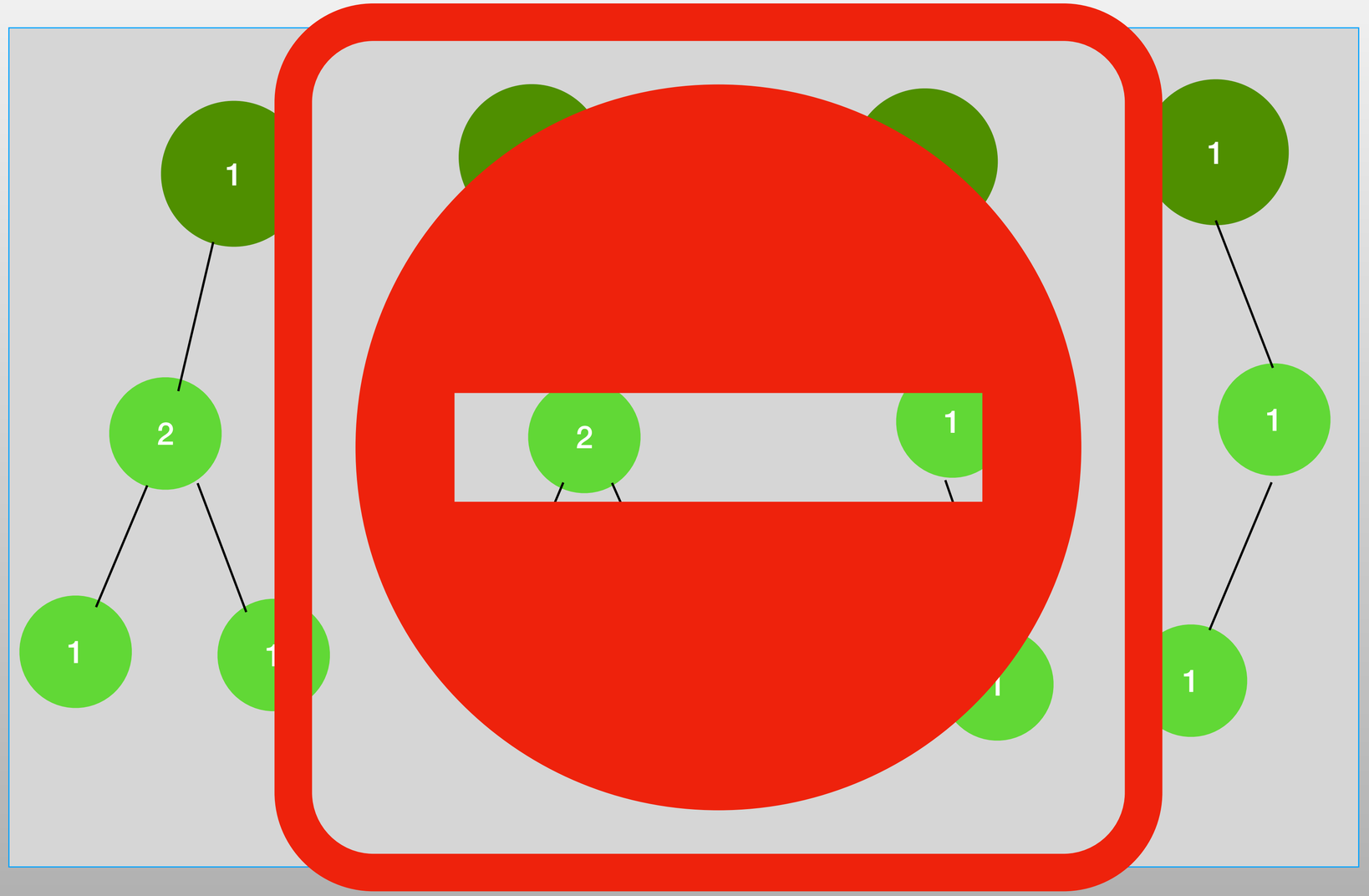
Reference Counting

Patterns in GC



Reference Counting

Patterns in GC



Mark/Sweep[/Compacting]

Patterns in GC

Objects are marked in Gray to be evaluated

Mark/Sweep[/Compacting]

Patterns in GC

Objects are marked in Gray to be evaluated

Live objects are marked Black, Dead objects are marked White

Mark/Sweep[/Compacting]

Patterns in GC

Objects are marked in Gray to be evaluated

Live objects are marked Black, Dead objects are marked White

Dead objects are swept off

Mark/Sweep[/Compacting]

Patterns in GC

Objects are marked in Gray to be evaluated

Live objects are marked Black, Dead objects are marked White

Dead objects are swept off

[Optional step] Compact objects to create contiguous free space

Copying

Patterns in GC

Space divided into 2 equal size areas (From and To regions)

Copying

Patterns in GC

Space divided into 2 equal size areas (From and To regions)

Objects are marked in Gray to be evaluated

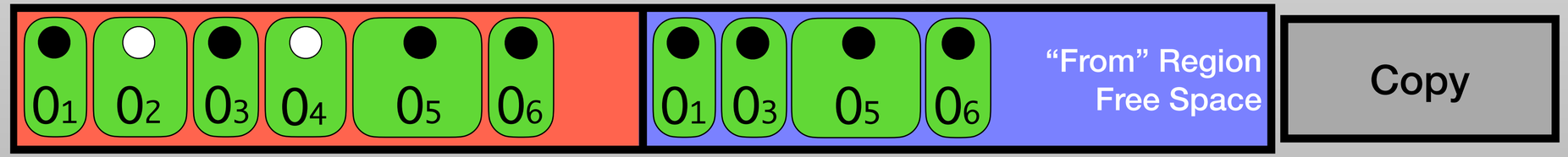
Copying

Patterns in GC

Space divided into 2 equal size areas (From and To regions)

Objects are marked in Gray to be evaluated

Live objects marked Black copied to To region



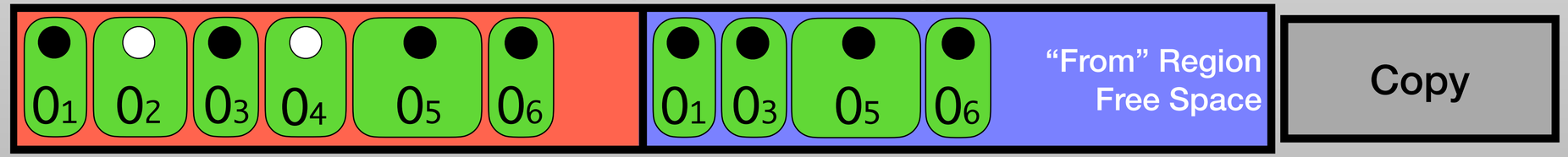
Copying

Patterns in GC

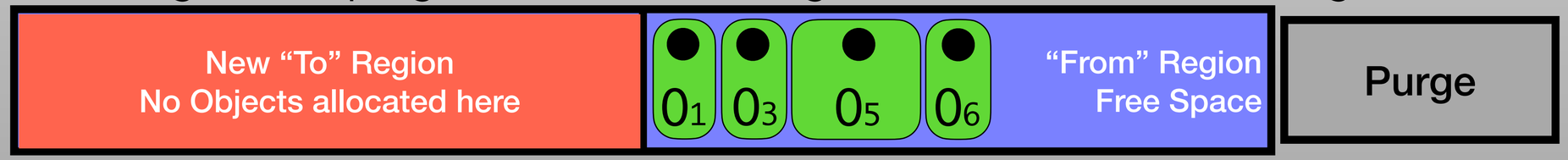
Space divided into 2 equal size areas (From and To regions)

Objects are marked in Gray to be evaluated

Live objects marked Black copied to To region



From region is purged. Current To region becomes new From region



Comparing M/S[/C] and Copying

Patterns in GC

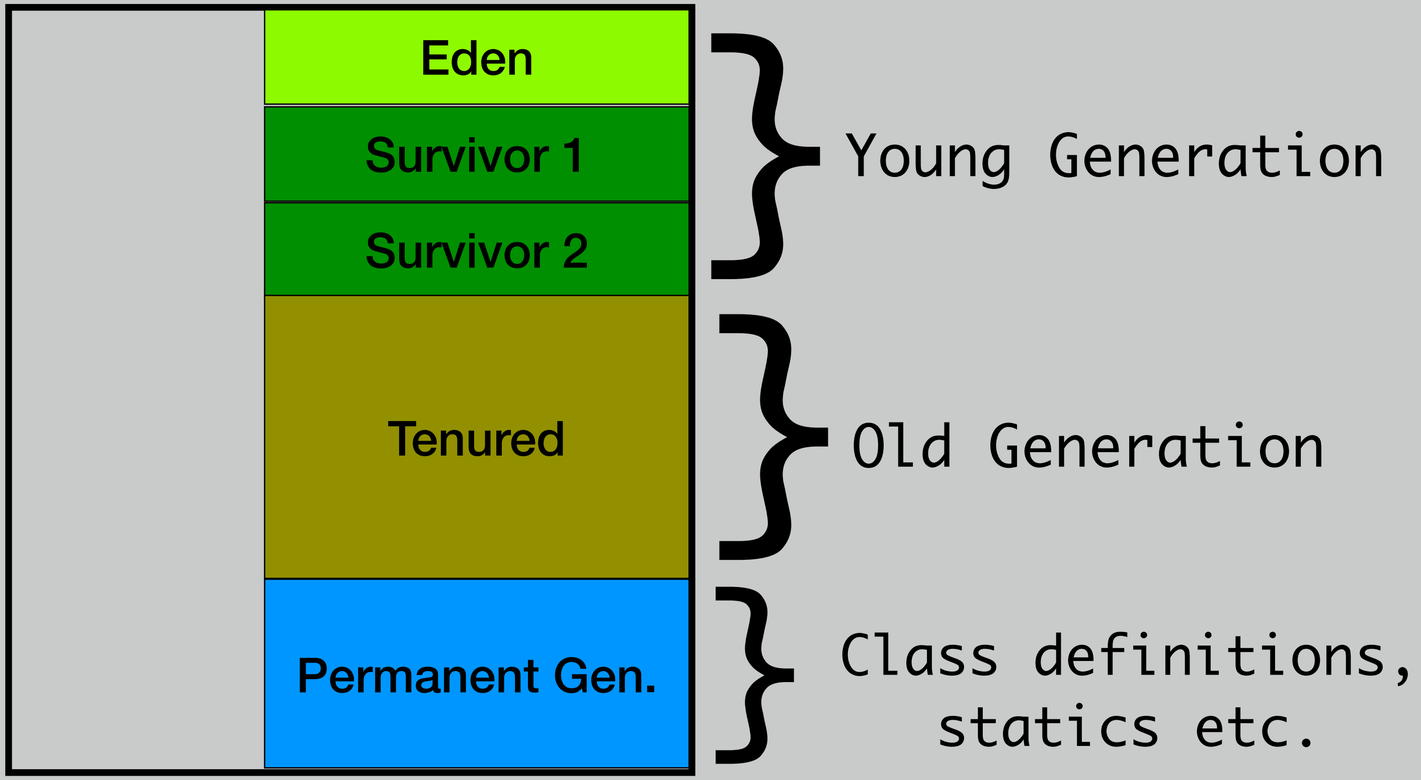
Mark/Sweep[/Compact]	Copying
efficient for larger memory areas	efficient for small memory areas
works best on longer-lived objects	works best on short-lived objects
incremental or all-at-once collector	all-at-once collector
no extra free space needed to run	needs 2x the memory space to run
concurrent e.g. Concurrent Mark Sweep stop-the-world e.g. Mark Sweep Compact	stop-the-world collector

- Overview and history of garbage collection
- Types of garbage collection
- General patterns
- Deeper dive into Generational GCs
- The newer Garbage First GC
- Recently added garbage collectors

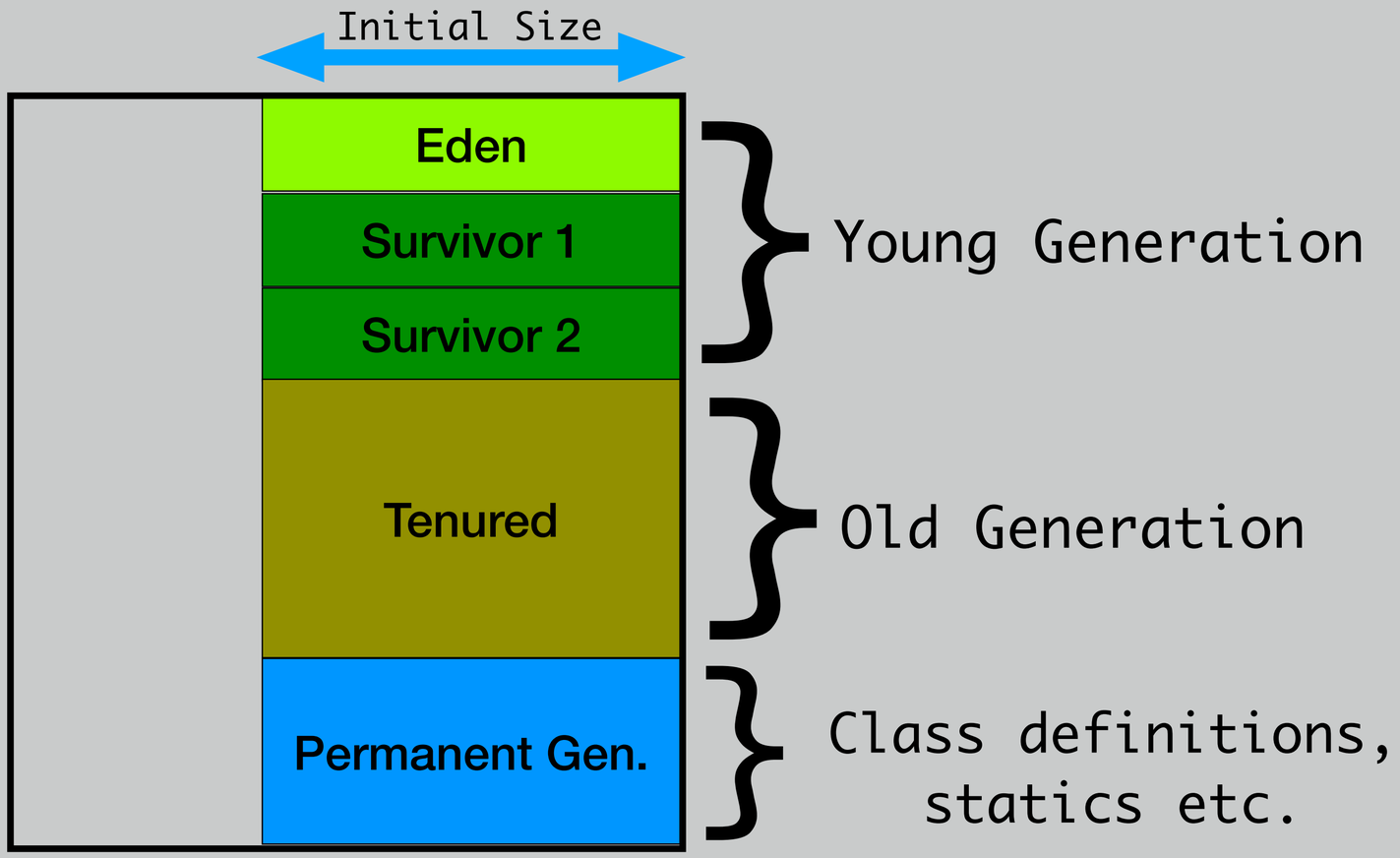
Generational GC

- Default GC in Java since Java 5
- Divides memory into smaller areas
- Collection patterns different per area
- Based on Weak Generational Hypothesis:
  - Most objects don’t have a long life
  - Older gen. objects rarely reference younger gen. objects

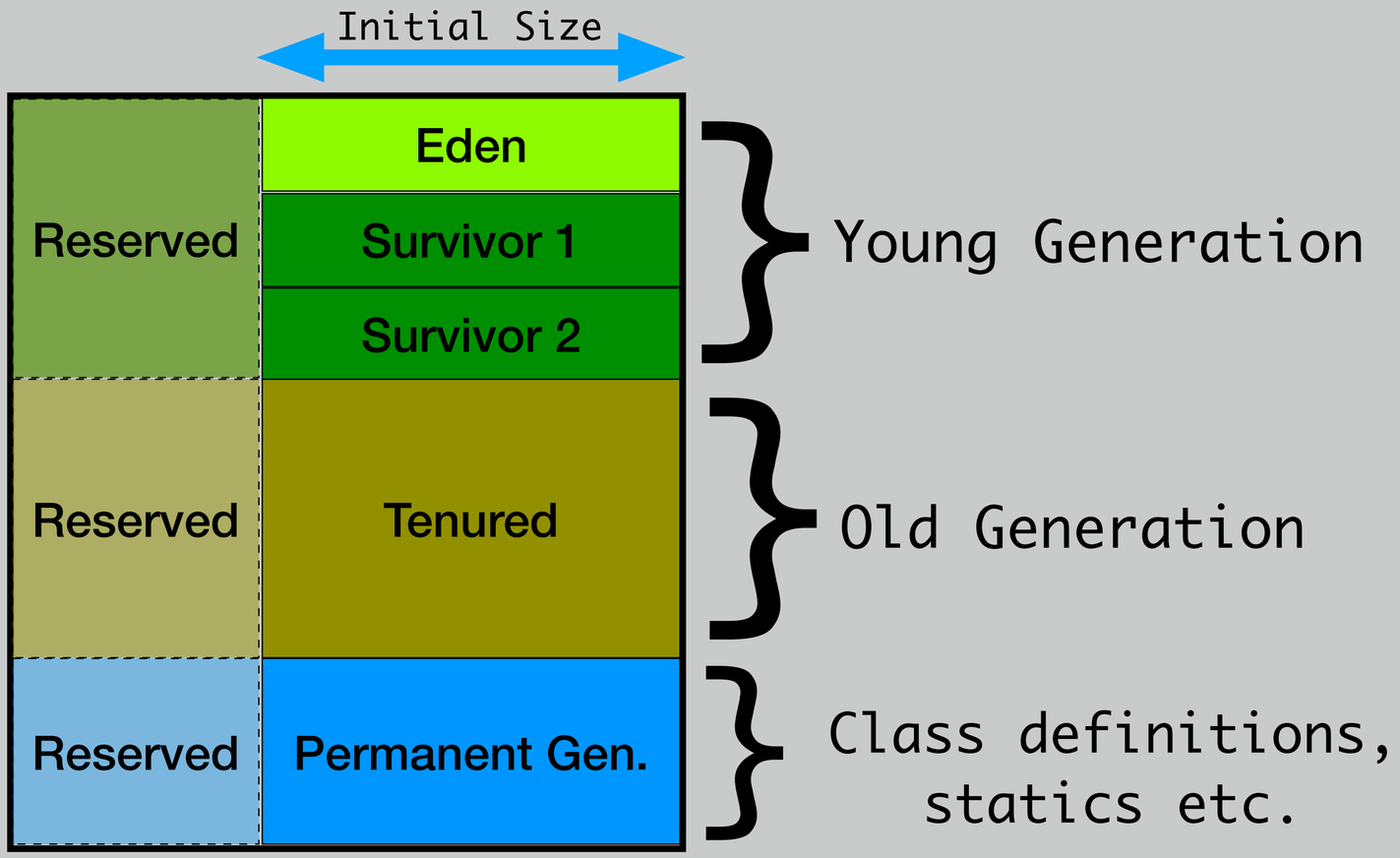
Generational GC



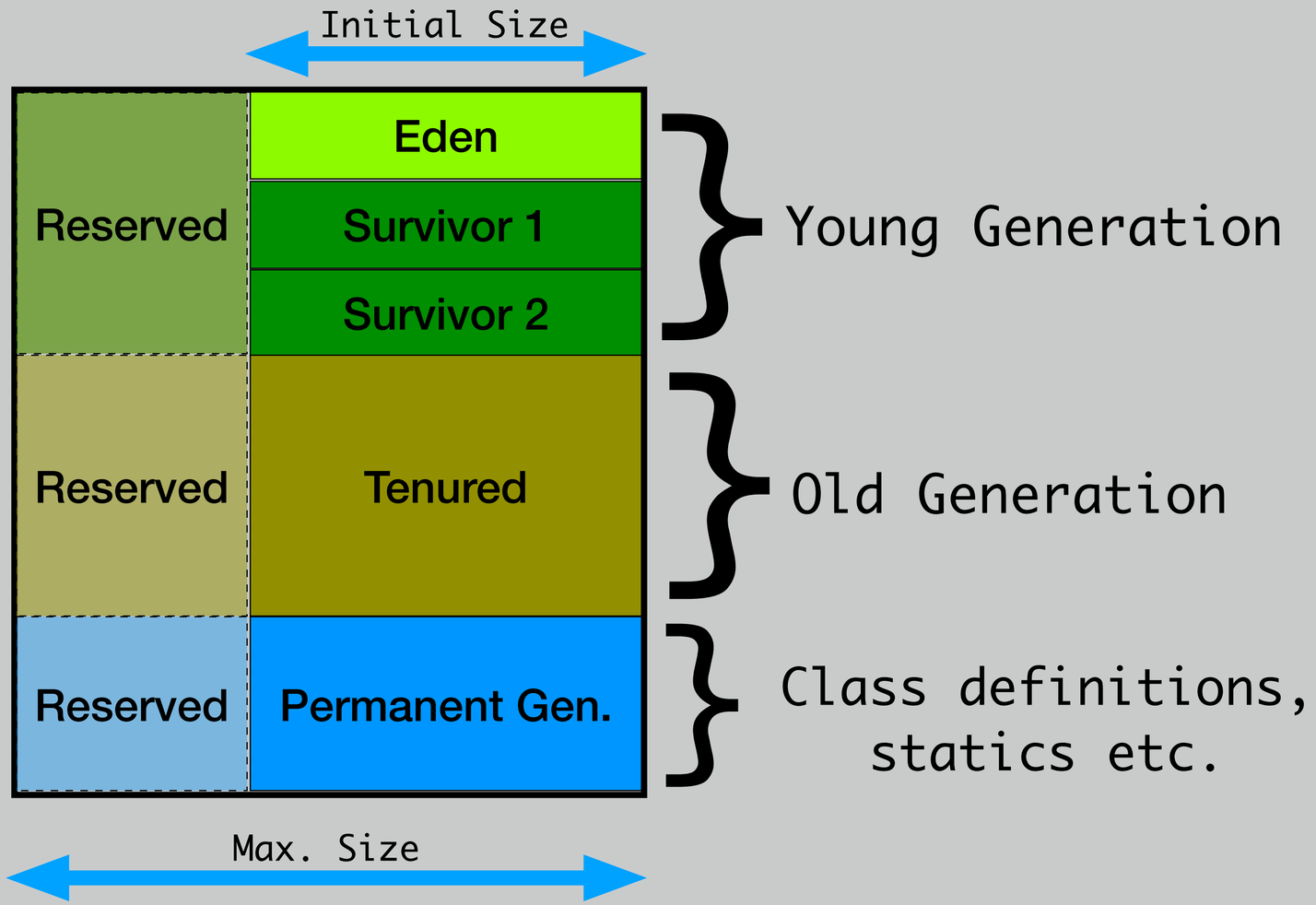
Generational GC



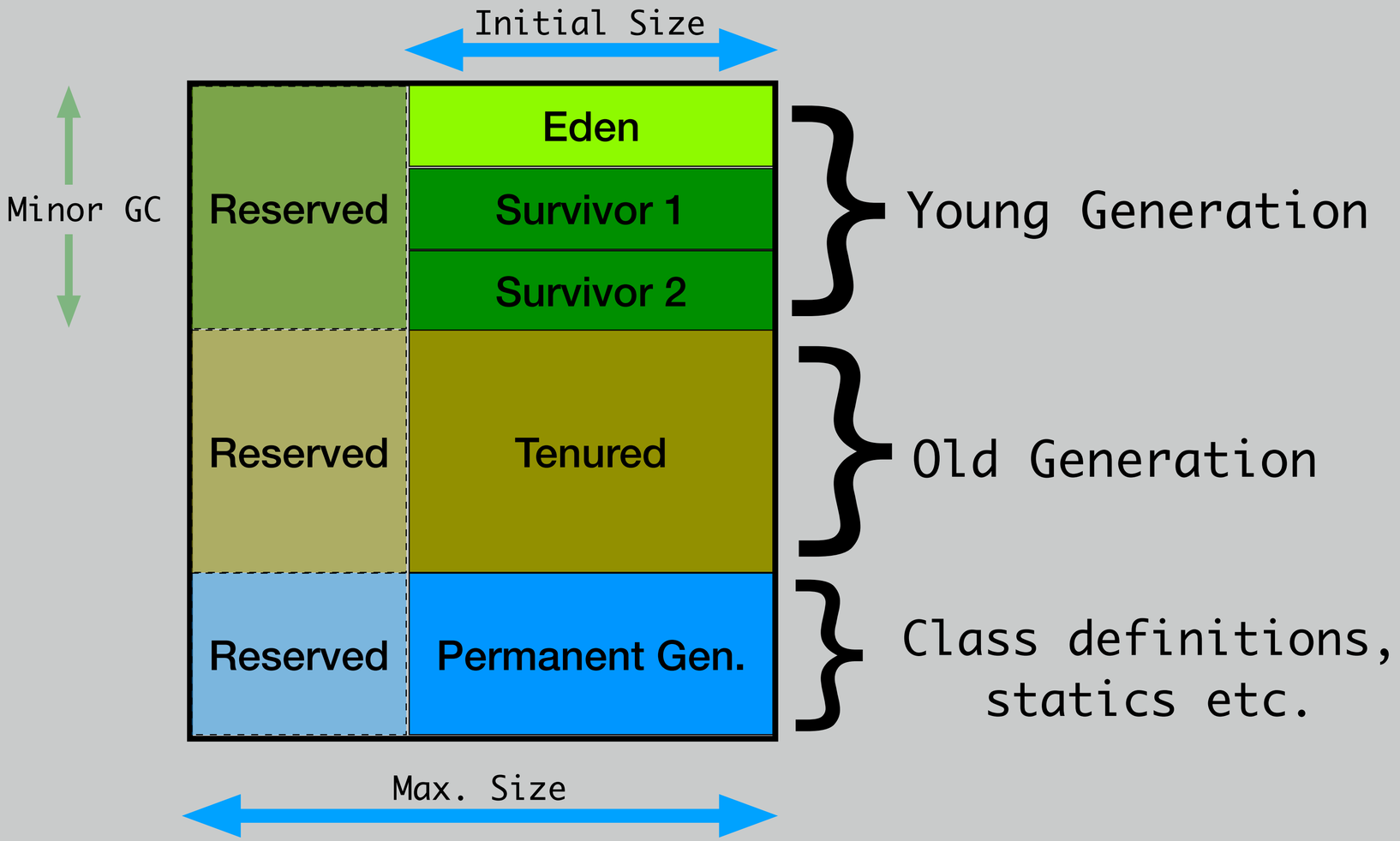
Generational GC



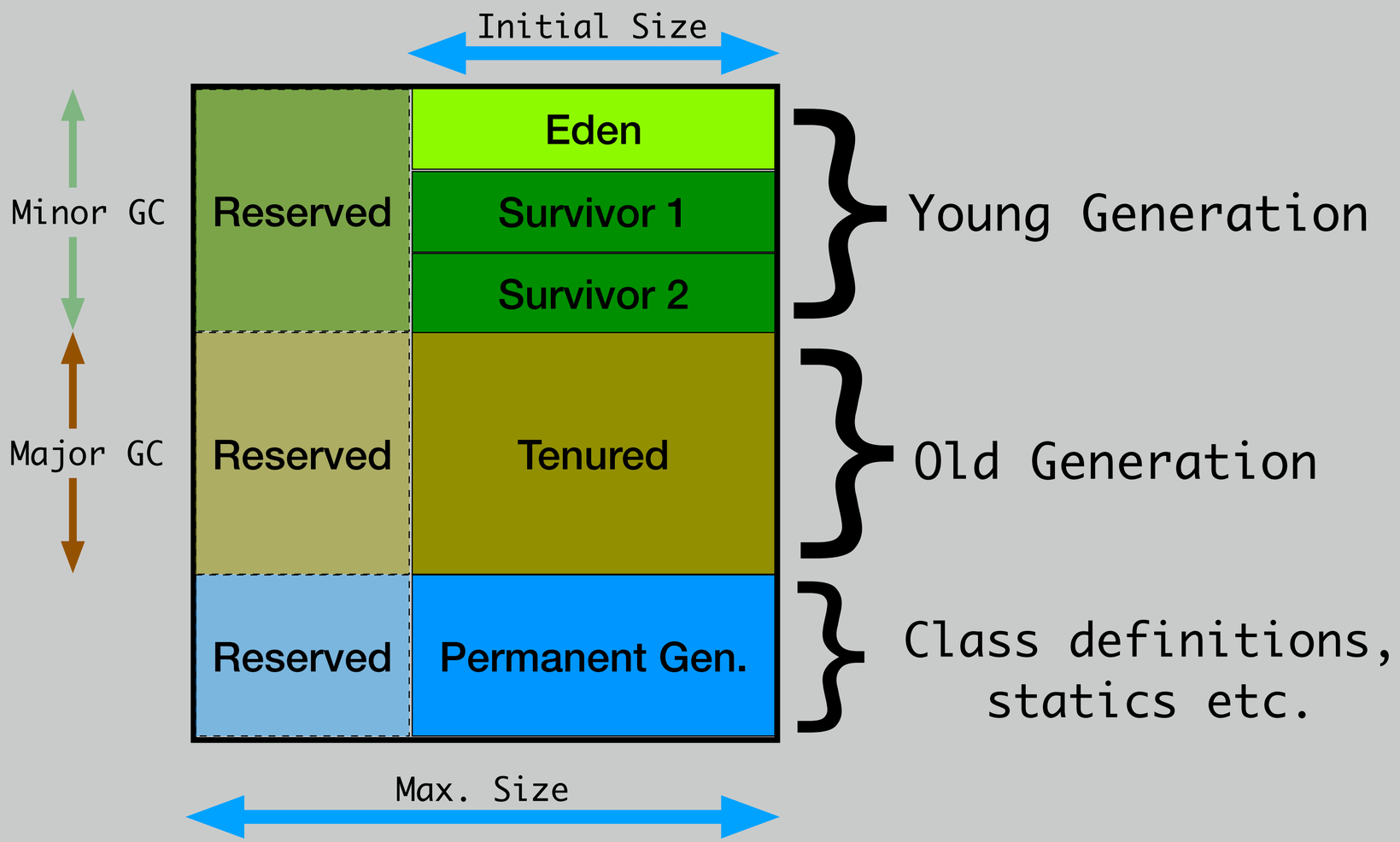
Generational GC



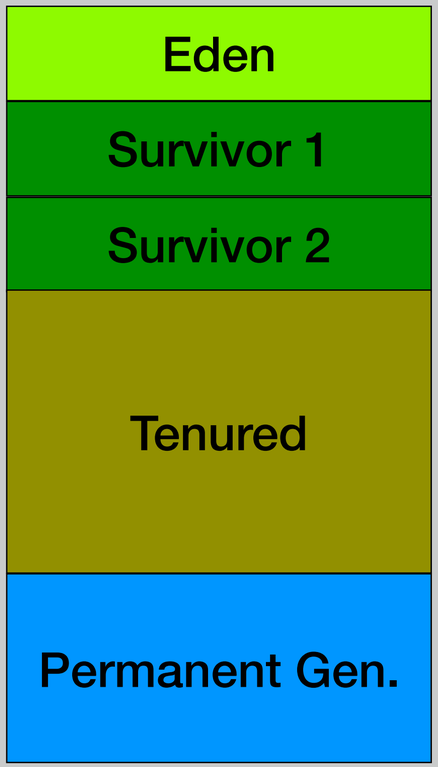
Generational GC



Generational GC

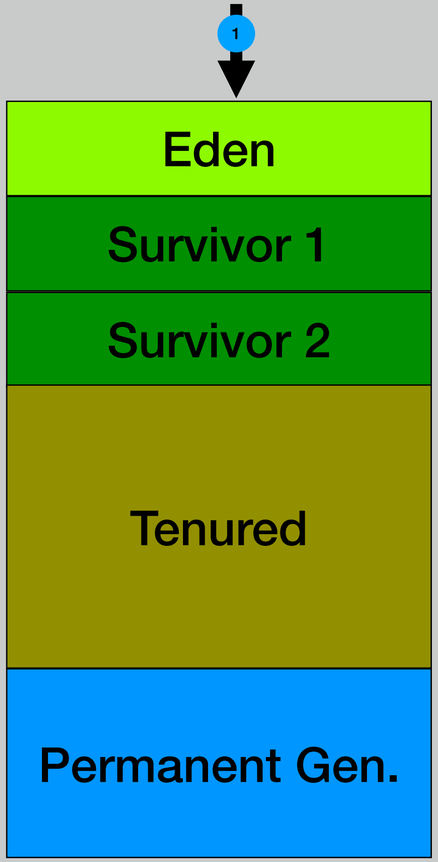


Generational GC



See Appendix 1 for details

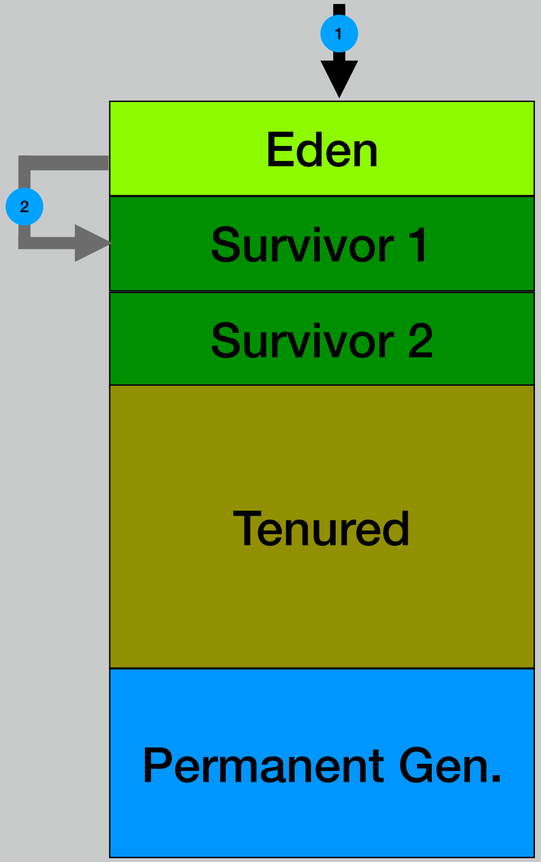
Generational GC



1 New object(s) added to Eden
Age = 0

See Appendix 1 for details

Generational GC

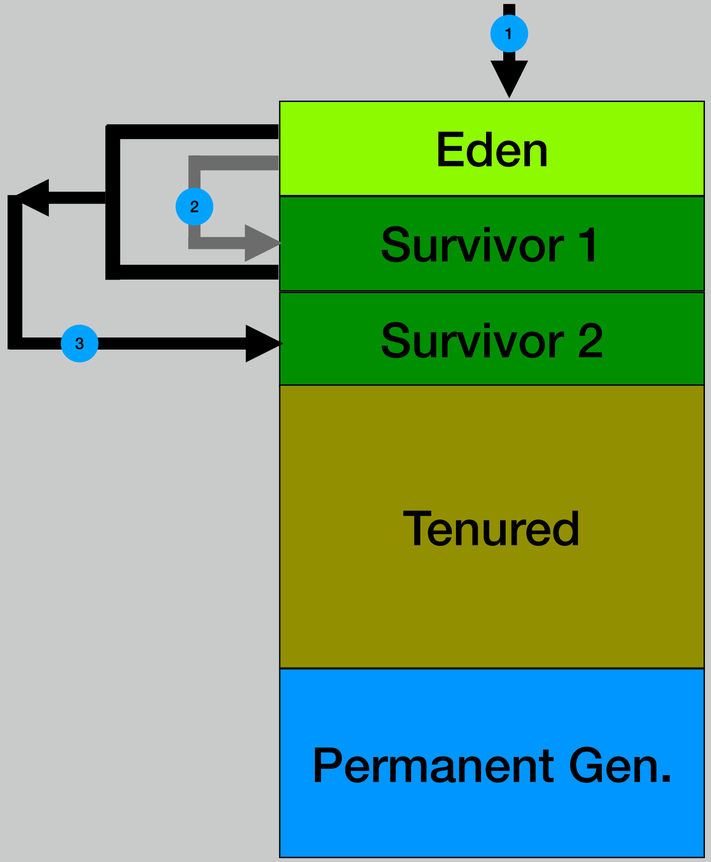


1. New object(s) added to Eden
   Age = 0

2. First GC: Live objects moved
   From Eden -> Survivor 1 (S1)
   Age = 1

See Appendix 1 for details

Generational GC



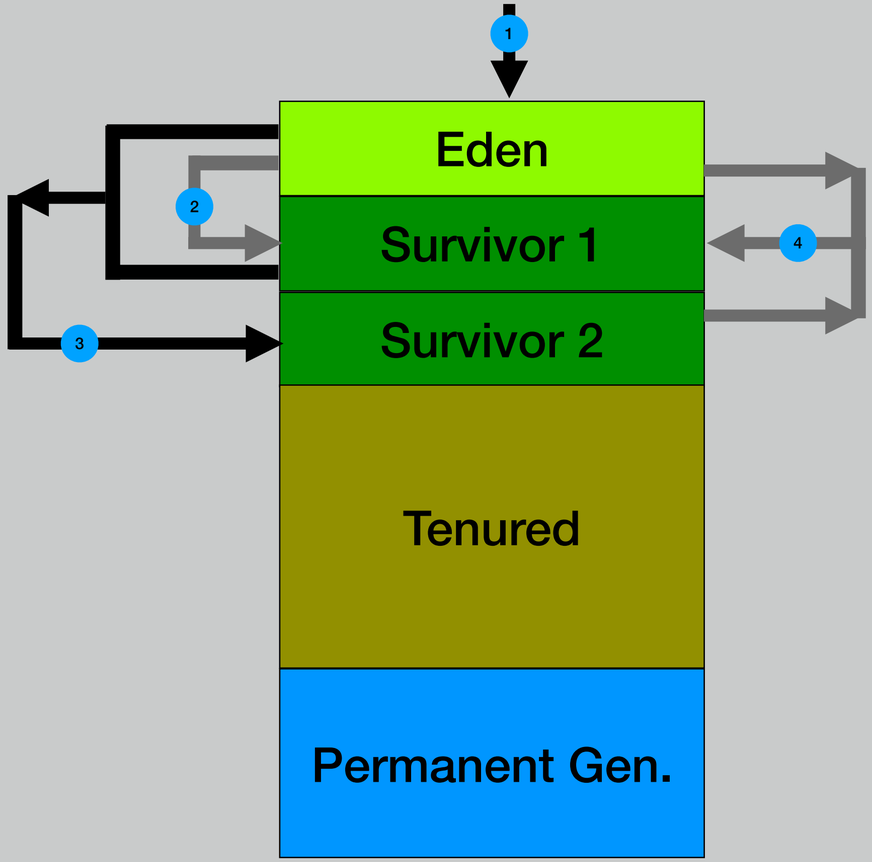
1. New object(s) added to Eden
   Age = 0

2. First GC: Live objects moved
   From Eden -> Survivor 1 (S1)
   Age = 1

3. Second GC: Live objects moved
   From Eden, S1 -> S2
   Age ++ (E -> S1 = 1, S1 -> S2 = 2)

See Appendix 1 for details

Generational GC



1. New object(s) added to Eden
   Age = 0

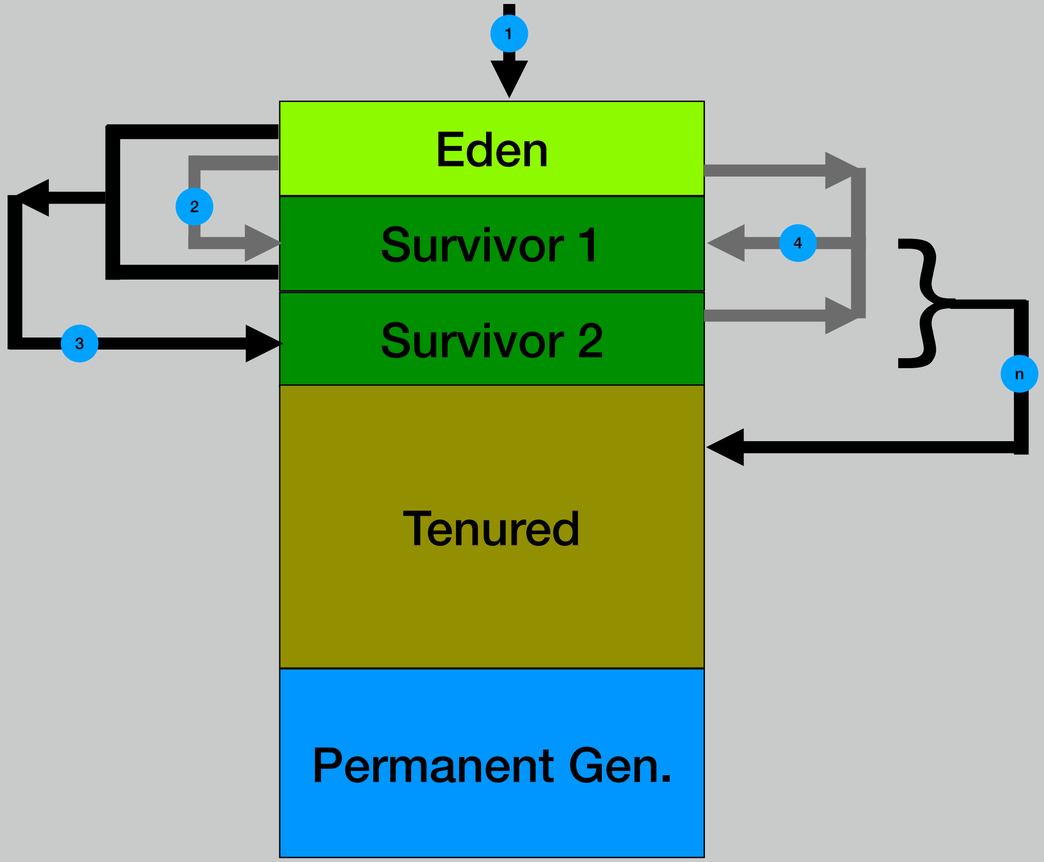
2. First GC: Live objects moved
   From Eden -> Survivor 1 (S1)
   Age = 1

3. Second GC: Live objects moved
   From Eden, S1 -> S2
   Age ++ (E -> S1 = 1, S1 -> S2 = 2)

4. Third GC: Live objects moved
   From Eden, S2 -> S1
   Age ++ (E -> S1 = 1, S2 -> S1 = 3)

See Appendix 1 for details

Generational GC



1. New object(s) added to Eden
   Age = 0

2. First GC: Live objects moved
   From Eden -> Survivor 1 (S1)
   Age = 1

3. Second GC: Live objects moved
   From Eden, S1 -> S2
   Age ++ (E -> S1 = 1, S1 -> S2 = 2)

4. Third GC: Live objects moved
   From Eden, S2 -> S1
   Age ++ (E -> S1 = 1, S2 -> S1 = 3)

. . .

n. Nth GC: Live objects moved
   From Current Survivor -> Tenured
   Age ~= 15

See Appendix 1 for details

Generational GC

- Heap is broken into smaller areas (sub-heaps)
- newly created objects (Age = 0)
  -> allocated to a young area: Eden
- new objects that live beyond than a GC cycle (Age = 1)
  -> move to a young area: Survivor (S1)
- objects that “survive” a few GC cycles (Age ++)
  -> get moved between young areas: S1 -> S2 or S2 -> S1
- objects that survive longer (Age > 15)
  -> move to a an older area: Tenured
- method definitions/statics etc
  -> allocated to PermGen, need not be collected

Collections for Young & Old Areas

GC Type \ Generation	Young Generation Eden, Survivors	Old Generation Tenured
Serial Collection	Serial 	Serial 
Parallel Collection	Parallel Scavenge 	Parallel 
Concurrent Mark-Sweep	Parallel New Same as Parallel New  (same algorithm diff impl)	Concurrent Mark Sweep until compaction time (Concurrent Mode Failure) then Parallel STW

See Appendix 2, 2a, 2b, 2c

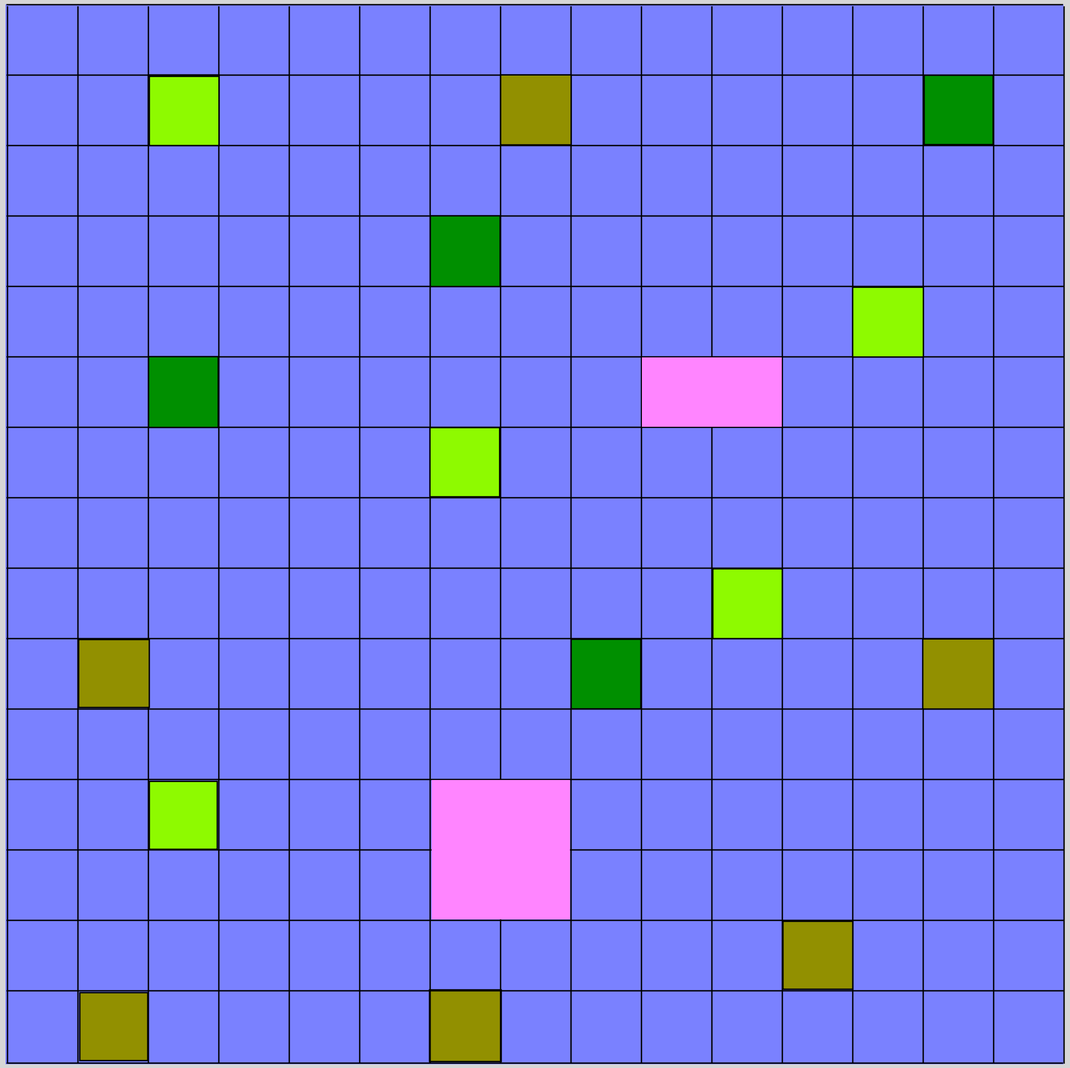
- Overview and history of garbage collection
- Types of garbage collection
- General patterns
- Deeper dive into Generational GCs
- The newer Garbage First GC
- Recently added garbage collectors

Why a new GC?

Issues with existing Generational GC

- Mostly focused on live objects not on garbage
- Contiguous memory area arrangement reduces flexibility
- Different collectors for different generations
- By default, unpredictable and inconsistent pause-times
- May need heavy tune-ups to stabilize pause times
- Not performant for large memory heaps
- Either highly prone to fragmentation
  or demand smaller heaps for predictable times.

Garbage First GC (G1GC)



Eden Region

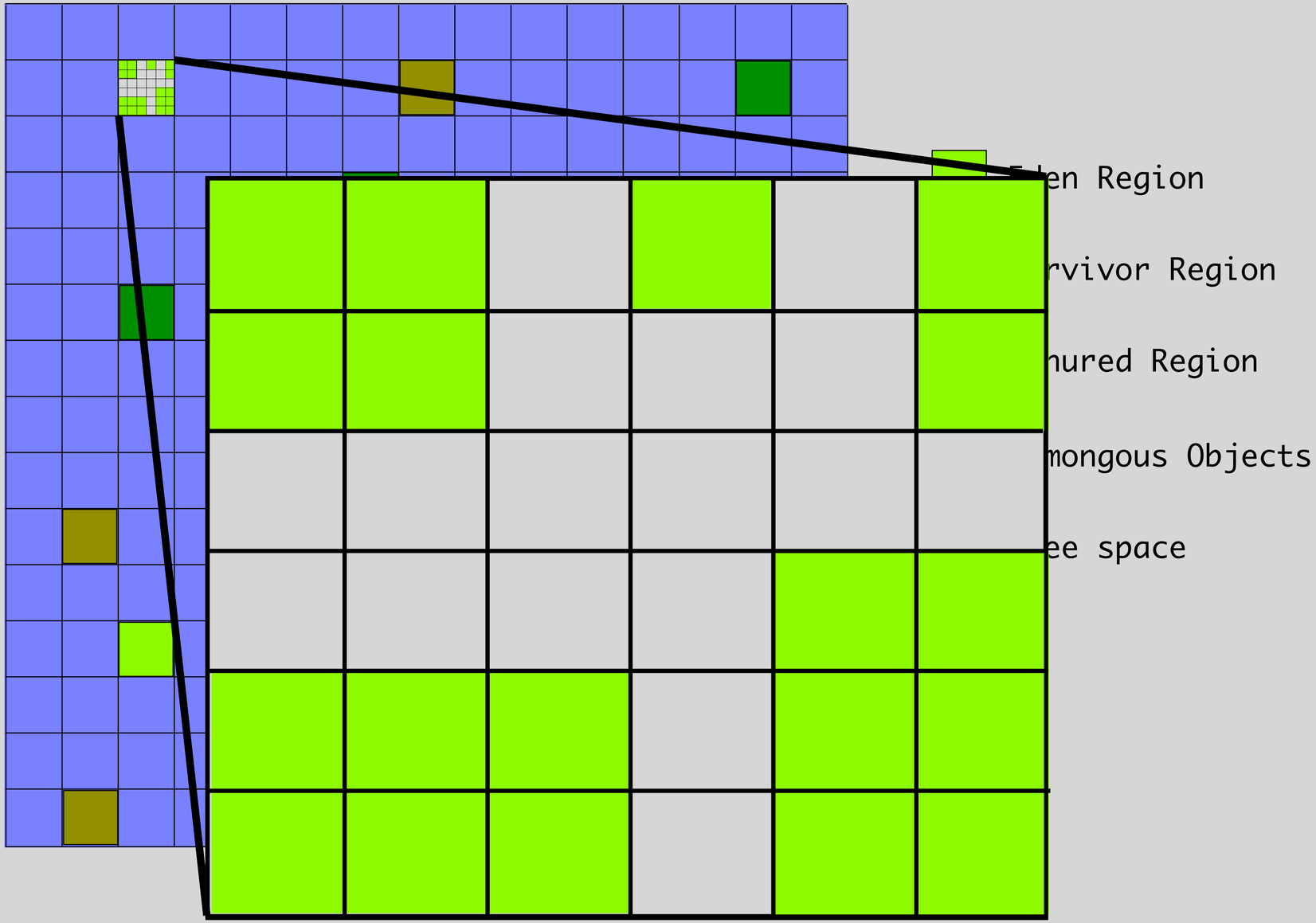
Survivor Region

Tenured Region

Humongous Objects

Free space

Garbage First GC (G1GC)



Garbage First GC (G1GC) Regions

See Appendix 3 for Region Math

Garbage First GC (G1GC) Regions

- A region is a contiguous unit of memory

See Appendix 3 for Region Math

Garbage First GC (G1GC) Regions

- A region is a contiguous unit of memory
- Both for allocation and space reclamation

See Appendix 3 for Region Math

Garbage First GC (G1GC) Regions

- A region is a contiguous unit of memory
- Both for allocation and space reclamation
- Regions are formed by dividing heap into ~2048 or more equal size blocks

See Appendix 3 for Region Math

Garbage First GC (G1GC) Regions

- A region is a contiguous unit of memory
- Both for allocation and space reclamation
- Regions are formed by dividing heap into ~2048 or more equal size blocks
- Region sizes are from 1 MB - 32 MB (power of 2) based on the heap size

See Appendix 3 for Region Math

Garbage First GC (G1GC) Regions

- A region is a contiguous unit of memory
- Both for allocation and space reclamation
- Regions are formed by dividing heap into ~2048 or more equal size blocks
- Region sizes are from 1 MB - 32 MB (power of 2) based on the heap size
- The memory manager assigns a region

See Appendix 3 for Region Math

Garbage First GC (G1GC) Regions

- A region is a contiguous unit of memory
- Both for allocation and space reclamation
- Regions are formed by dividing heap into ~2048 or more equal size blocks
- Region sizes are from 1 MB - 32 MB (power of 2) based on the heap size
- The memory manager assigns a region
- Regions are free, eden, survivor, tenured

See Appendix 3 for Region Math

Garbage First GC (G1GC) Regions

- A region is a contiguous unit of memory
- Both for allocation and space reclamation
- Regions are formed by dividing heap into ~2048 or more equal size blocks
- Region sizes are from 1 MB - 32 MB (power of 2) based on the heap size
- The memory manager assigns a region
- Regions are free, eden, survivor, tenured
- Large objects allocated as a humongous region

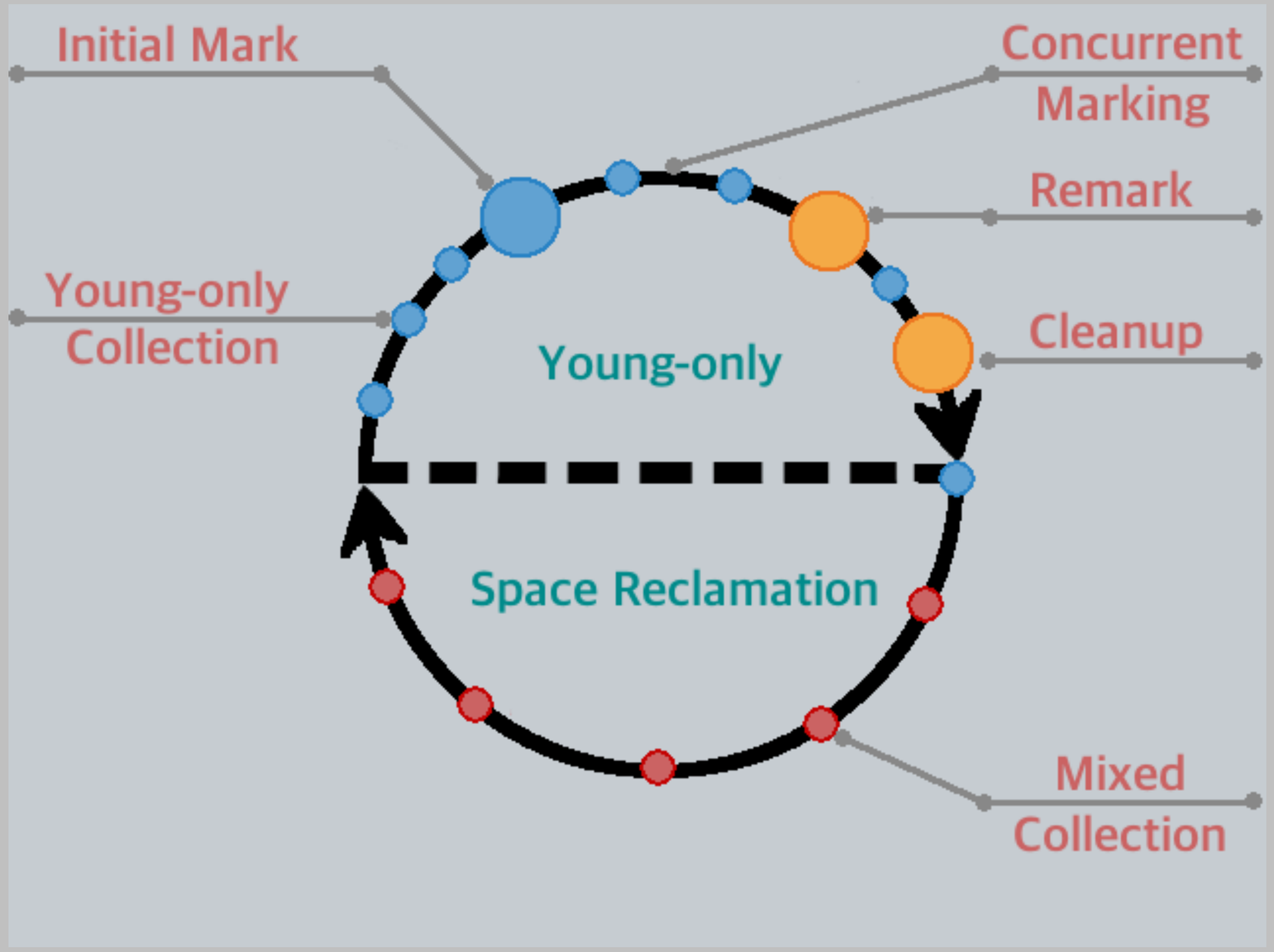
See Appendix 3 for Region Math

Garbage First GC (G1GC) Regions

- Humongous objects occupy complete regions
- For larger objects, contiguous regions used
- defined as objects ≥ 50% of region size.
- not allocated as young regions
- not collected in young gen. GC

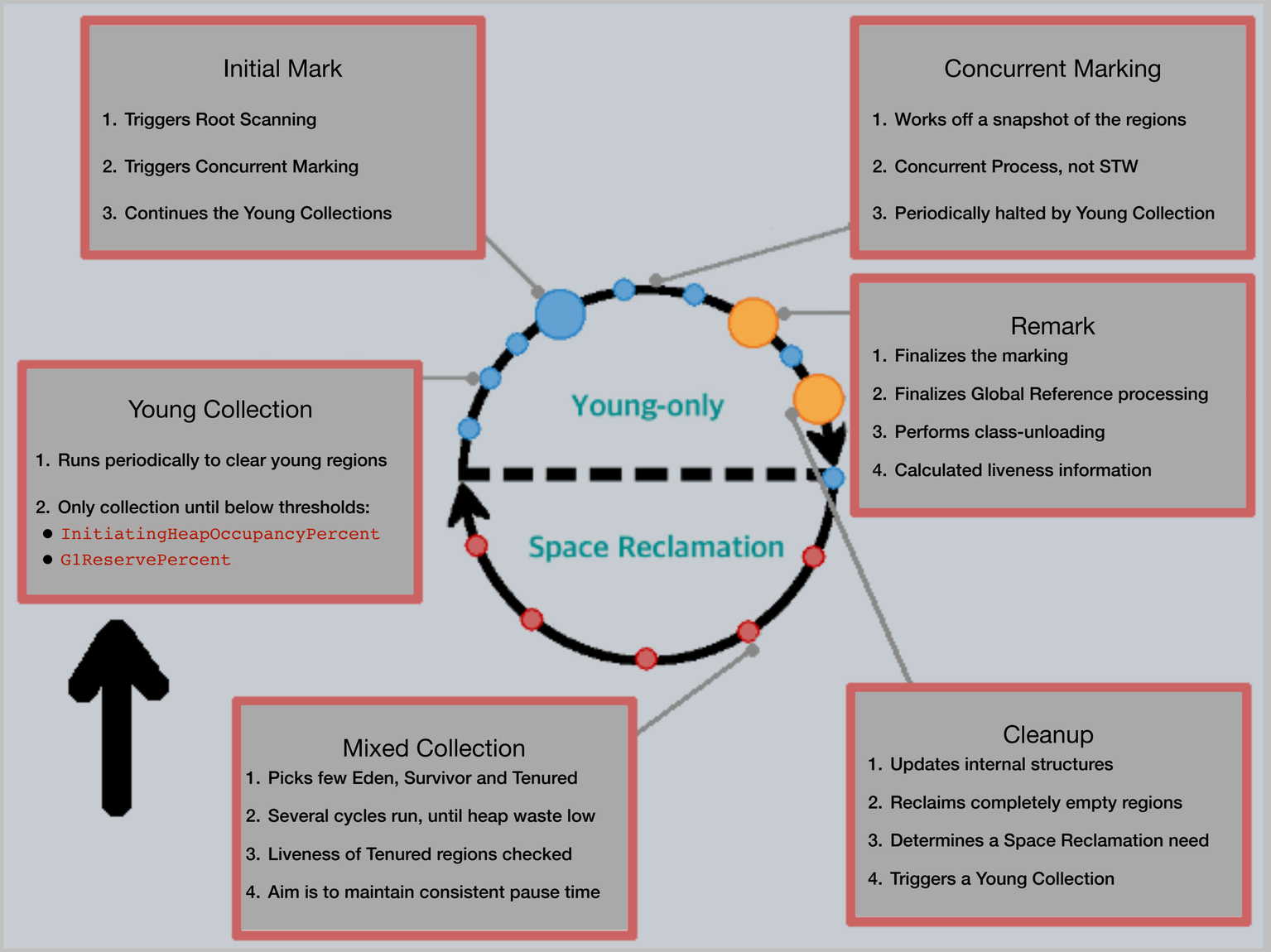
See Appendix 3 for Region Math

G1GC Cycle



See Appendix 4 for Initial Mark

G1GC Cycle



See Appendix 5 for details

See Appendix 6 for graphics

The old PermGen

- PermGen allocated as a part of JVM Heap
- PermGen is implicitly bounded
- PermGen is allocated at startup.
- PermGen could not use O/S memory swaps.
- Default PermGen size is 64M
  (85M for 64-bit scaled pointers).

The new Metaspace

- Metaspaces are not a panacea for OutOfMemoryErrors.
- Metaspace is explicitly bounded from the O/S memory, taking up unlimited amounts otherwise.
- Initial Metaspace Size is set by `-XX:MetaspaceSize` (replaces -XX:PermSize)
- Max Metaspace is set by `-XX:MaxMetaspaceSize` (replaces -XX:MaxPermSize)

- Overview and history of garbage collection
- Types of garbage collection
- General patterns
- Deeper dive into Generational GCs
- The newer Garbage First GC
- Recently added garbage collectors

Z GC

JDK 11

(Oracle ⇢ OpenJDK)

- Region-based, non-generational concurrent collector, based on the G1GC
- Designed for very large memory heaps and predictable throughput
- Low GC pause times, not exceeding 10ms
- No more than 15% application throughput reduction compared to using G1
- Aims at simplifying tuning of GCs as well
- More reading material:
  - JEP-333 (https://openjdk.java.net/jeps/333)
  - ZGC Wiki (https://wiki.openjdk.java.net/display/zgc/Main)

Epsilon GC

JDK 11

(Redhat → OpenJDK)

- Completely passive GC with just bounded memory allocation and lowest latency guarantees
- Linear allocation in a single chunk of memory but does not actually reclaim memory
- Uses trivial lock-free Thread Local Allocation Buffers (TLABs) that do not need managing.
- Best suited for:
  - performance testing the app (without GC latency).
  - extremely short lived jobs.
  - memory pressure testing.
- More reading material:
  - JEP-318 (https://openjdk.java.net/jeps/318)
  - Remove the Garbage Collector (https://www.infoq.com/news/2017/03/java-epsilon-gc)

Shenandoah GC

JDK 12

(Redhat → OpenJDK)

- Region-based, non-generational collector, based on the G1GC
- Young collection equivalent is run in a concurrent-partial mode
- Uses a concurrent mark and a concurrent compact for longer lived objects
- In slower collection, cycles are stolen from application, but is not STW
- Pause times said to be independent of heap size (be it 2GB or even 100GB)
- Best for responsiveness and predictable pauses than more cpu cycles and space
- More reading material:
  - JEP-189 (http://openjdk.java.net/jeps/189)
  - Shenandoah Wiki (https://wiki.openjdk.java.net/display/shenandoah/Main)


System.gc()
JAVA Garbage Collector
JDK GIGC

Appendix

Appendix

Appendix	Content
Appendix 0	definitions for garbage collection jargon
Appendix 1	details about how JDK v5 - v8 generational GC works
Appendix 2	details about the combinations of collectors in JDK v5 - v8 generational GCs
Appendix 3	how region sizes are calculated in G1GC
Appendix 4	various triggers for Initial Mark in the G1GC Young Phase
Appendix 5	detailed about the Young Phase of G1GC
Appendix 6	graphical representation of the collection steps in G1GC
Appendix 7	printing default values for the JVM options
Appendix 8	logging the garbage collection using Unified Logging

Appendix 0

Garbage Collection - Definitions

Throughput – the percentage of total time not spent in garbage collection, considered over long periods of time.

Pause time – the length of time the application execution is stopped for garbage collection to occur.

GC overhead – the inverse of throughput, that is, the percentage of total time spent in garbage collection.

Collection frequency – how often collection occurs, relative to application execution.

Footprint – a measure of size, such as heap size.

Promptness – the time between when an object becomes garbage and when the memory becomes available

Appendix 1

Generational GC - Details

- there is a per-thread region for reducing synchronization
  Thread Local Allocation Buffer (TLAB)
- Eden itself divided into TLAB sections for
  - individual threads
  - a common area.
- different GC processes for each generation
- young region gc collectors typically are:
  - all-at-once
  - stop-the-world
  - copying pattern
- old region gc typically concurrent

Appendix 2

Generational GC

Generational GC

Young & Old GC Combinations

Appendix 2a

Generational GC - Combinations

Serial Collector

- both young and old gen. are Serial
- single thread, stop-the-world
- designed for small heap sizes
- useful for mobile or small apps

Appendix 2b

Generational GC - Combinations

Parallel Collector

- young gen. uses Parallel Scavenge
- old gen. initially was a Serial
- old gen. later improved to Parallel Old
- default in JDK 1.5, 1.6, 1.7 and 1.8*

Appendix 2c

Generational GC - Combinations

Concurrent Mark/Sweep (CMS) Collector

- young gen. uses Serial or Parallel New
- old gen. uses to Concurrent Mark Sweep
- eventually old gen. compaction needed
- old gen. compaction is Parallel Old

Appendix 3

G1GC - Region Math

Assume a 9GB heap

1. 9GB = 9 x 1024MB = 9216 MB

2. Approximate number of regions (2048 or more)

3. 9216 MB / 2048 regions = 4.5 MB per region

4. Power of 2 less than 4.5 = 2^2 = 4MB
   Possible regions = 9216MB / 4MB = 2304 regions

5. Power of 2 more than 4.5 = 2^3 = 8MB
   Possible regions = 9216MB / 8MB = 1152 regions

6. Choosing: 2304 ≥ 2048, so region size 4MB is chosen

7. An object is thus considered humongous if size > 50% of region
   Humongous object size = 2MB or greater

Appendix 4

G1GC Young Collection

What triggers Initial Mark?

Appendix 4a

G1GC Young Collection - Initial Mark - Triggers (1)

Initiating Heap Occupancy Percent (IHOP)

`-XX:InitiatingHeapOccupancyPercent`

- Default value is 45, thus an Initial Mark is triggered when old gen heap size is 45% filled.
- This is just an initiating value. G1 determines via measurement what the optimal percentage should be.
- Such an adaptive HOP can be turned off by un-setting the flag (notice the -): `-XX:-G1UseAdaptiveIHOP`.
- Turning off the Adaptive IHOP will make the G1 collector rely on the IHOP value alone.
- This value is usually considered a soft threshold, reaching this limit may not immediately trigger Initial Mark.

Appendix 4b

G1GC Young Collection - Initial Mark - Triggers (2)

Guaranteed Survivor Space Availability Percent

`-XX:G1ReservePercent`

- Default value is 10, thus an Initial Mark is triggered when survivor space availability falls to 10% filled.
- This is a flat unchanging value. G1 honors the value set during startup.
- This value supersedes the Heap Occupancy Percentage triggers.
- This value is usually considered a hard threshold, reaching this limit will immediately trigger Initial Mark.

Appendix 4c

G1GC Young Collection - Initial Mark - Triggers (3)

A Humongous allocation is reached

- Humongous objects are objects with a size of 50% or greater than, a region size.
- Directly allocated to Old gen. regions to avoid the potentially costly collections and moves of young gen.
- G1GC tries to eagerly reclaim such objects if they are found to not have references after many collections.
- Can be disabled by `-XX:-G1EagerReclaimHumongousObjects`, may need to turn on Experimental options.

Appendix 5

G1GC Young Collection - Steps

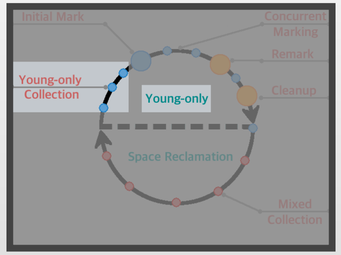
- Root Scanning
  - scan local and static objects for root objects
  - mark in a "dirty queue"
- Update Remembered Set (RSet)
  - all marked references in the dirty queue updated into a RSet
- Process RSet
  - detect references to objects in the collection set
  - including objects in Tenured regions
- Copy Live Objects
  - traverse the object graph and promote/age live objects
- Process references
  - update references to new location
  - process soft, weak, phantom and final references

Appendix 6

G1GC Young Collection - Details

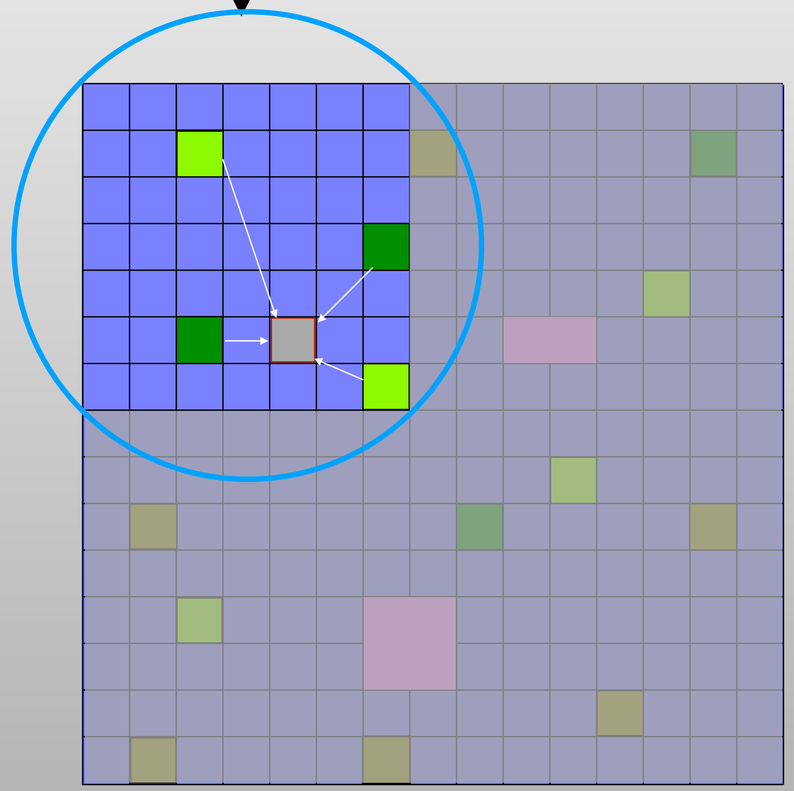
Appendix 6a

G1GC Young Collection - Eden + Survivor1 -> Survivor2

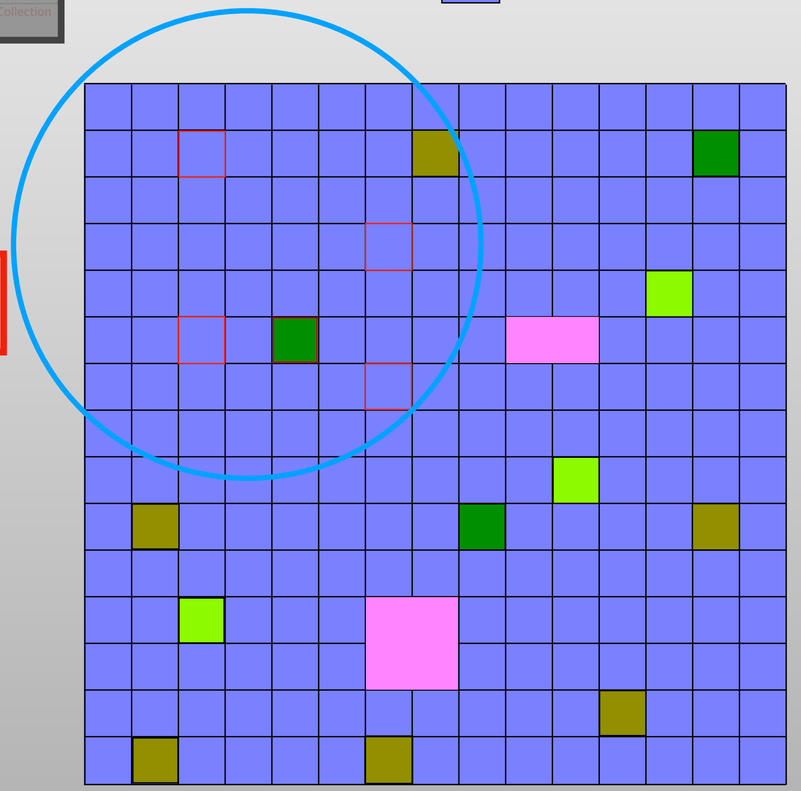


Eden Region
Survivor Region
Tenured Region
Humongous Objects
Free space

Collection Set



E, S_F —> S_T

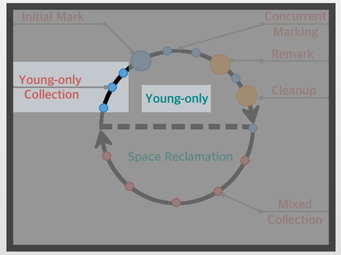


Before

After

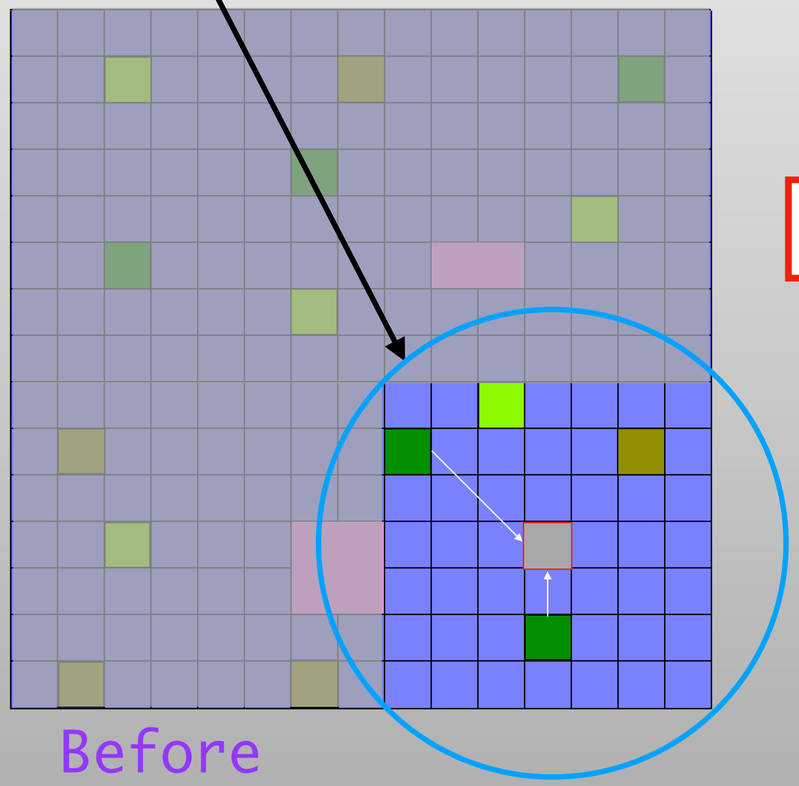
Appendix 6b

G1GC Young Collection - Survivor -> Tenured

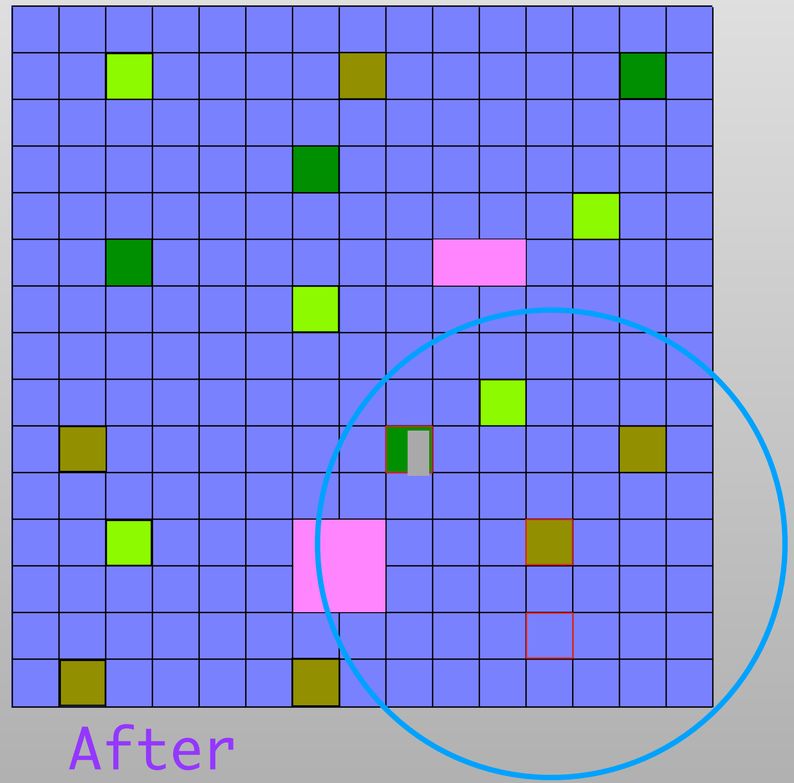


Eden Region
Survivor Region
Tenured Region
Humongous Objects
Free space

Collection Set



S_F —> T

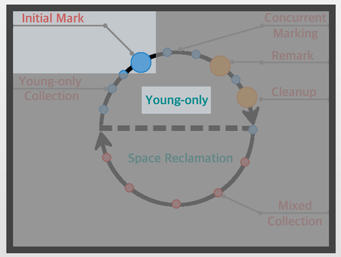


Before

After

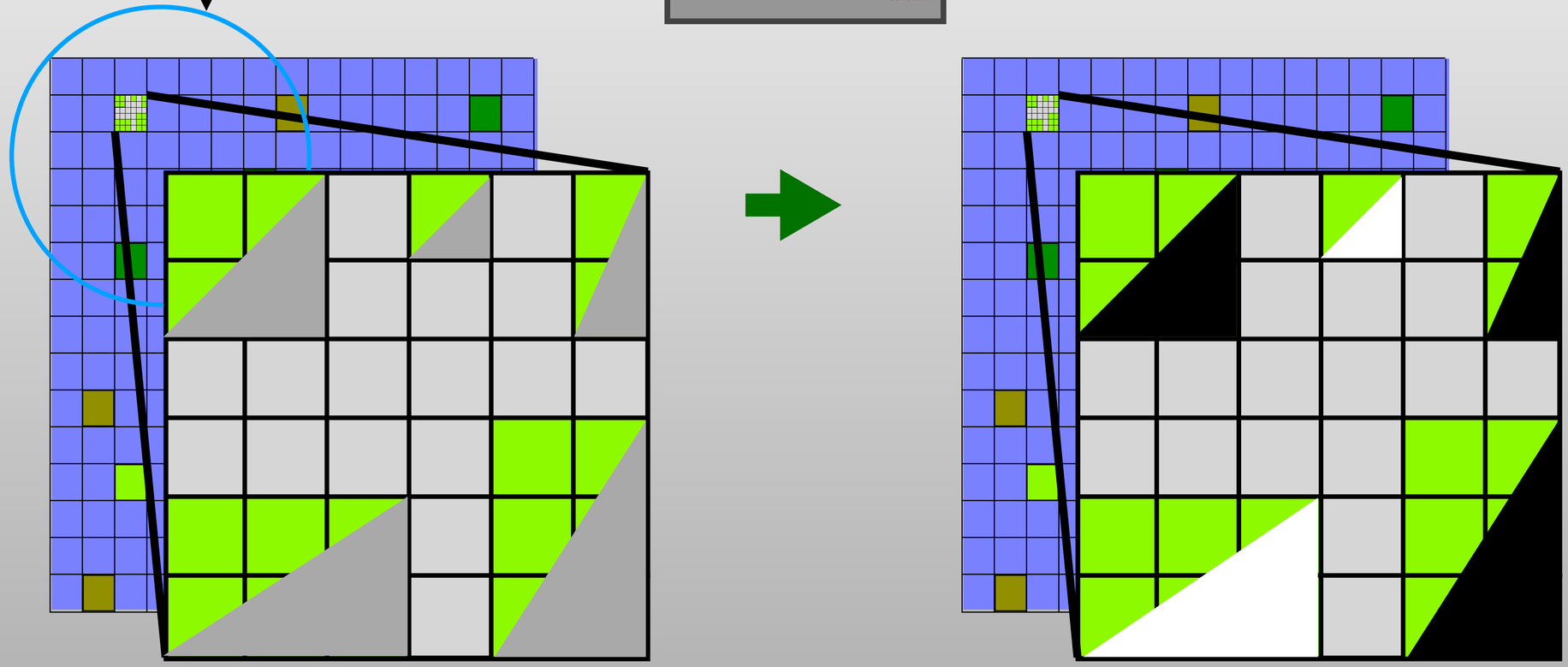
Appendix 6c

G1GC Young Collection - Initial Mark



Eden Region
Survivor Region
Tenured Region
Humongous Objects
Free space

Collection Set

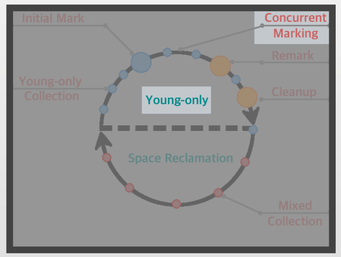


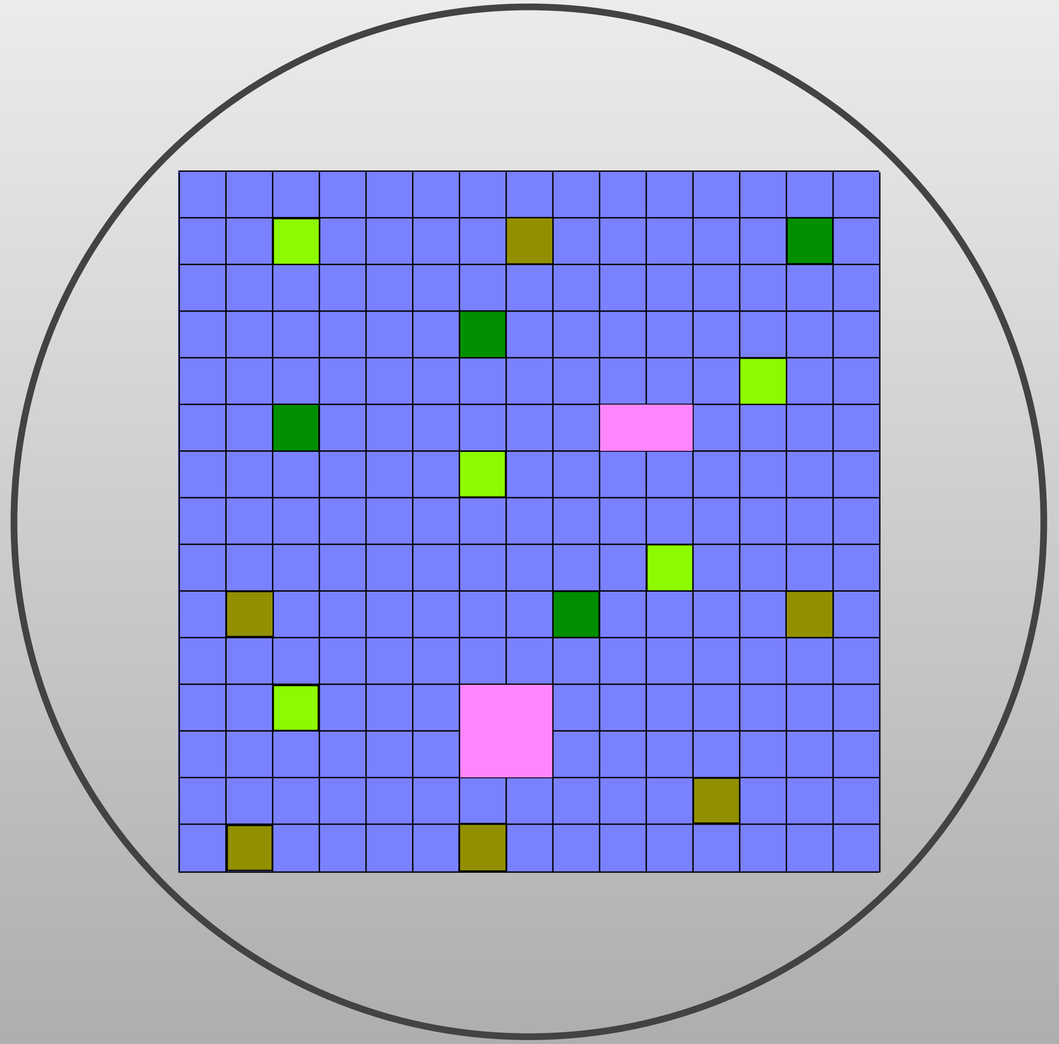
Before

After

Appendix 6d

G1GC Young Collection - Concurrent Marking - Overview



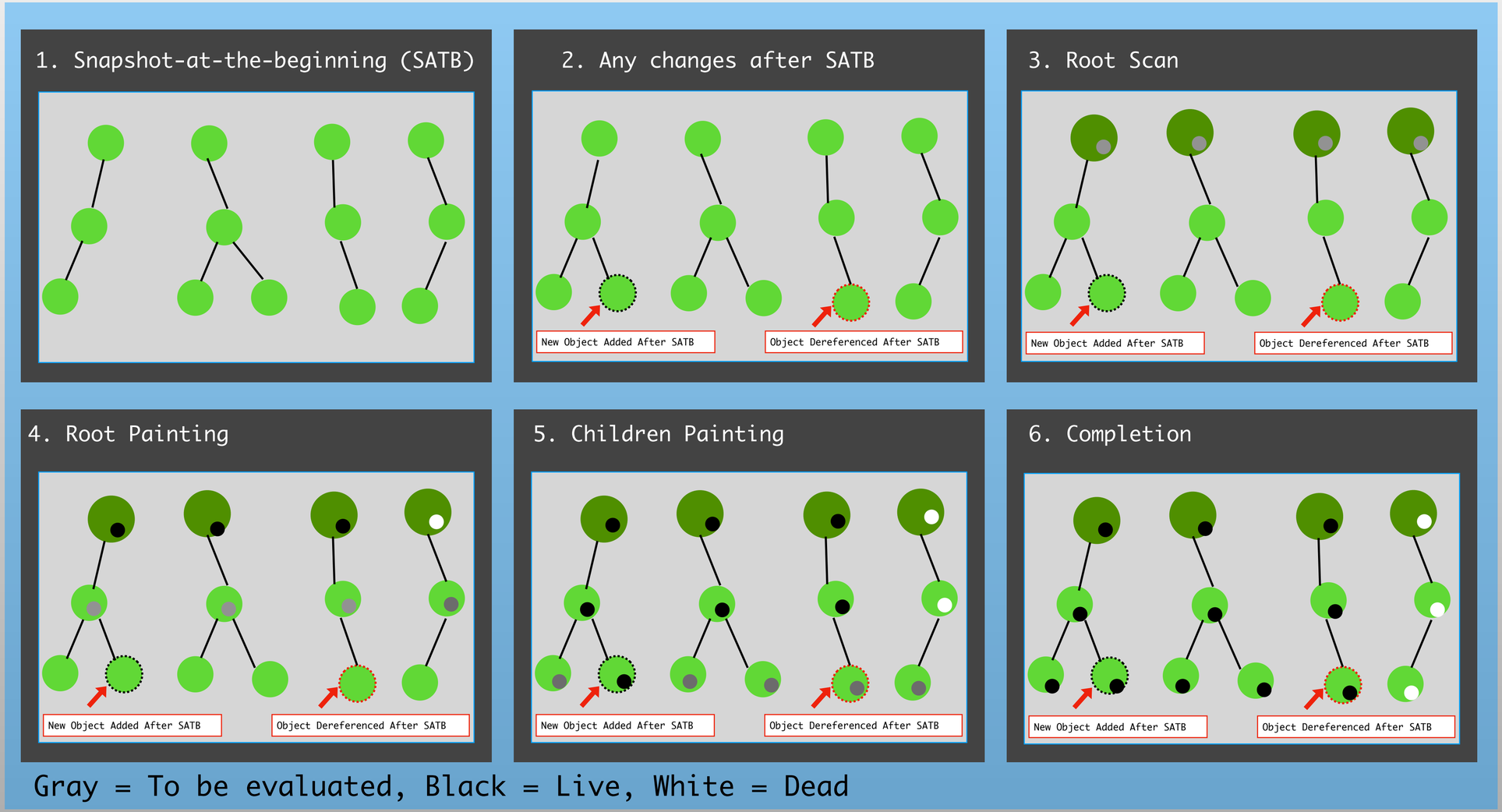


Eden Region
Survivor Region
Tenured Region
Humongous Objects
Free space

Appendix 6e

G1GC Young Collection - Concurrent Marking - SATB

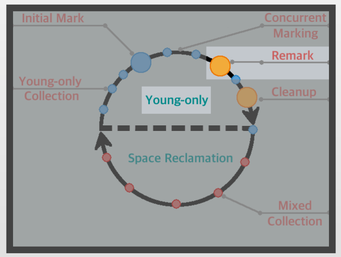
Concurrent Marking - Snapshot-At-The-Beginning



Gray = To be evaluated, Black = Live, White = Dead

Appendix 6f

G1GC Young Collection - Re-Mark



Eden Region
Survivor Region
Tenured Region
Humongous Objects
Free space

Collection Set

Reference Outside Collection Set

Add to RSet

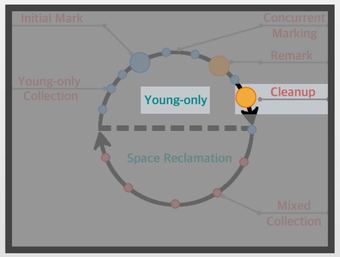
Remembered Set

Before

After

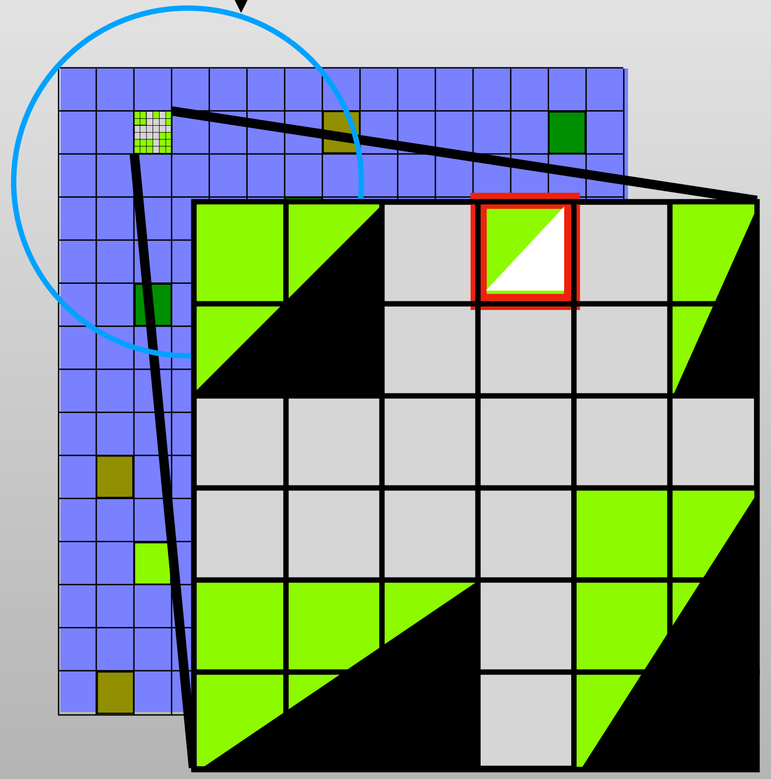
Appendix 6g

G1GC Young Collection - Cleanup

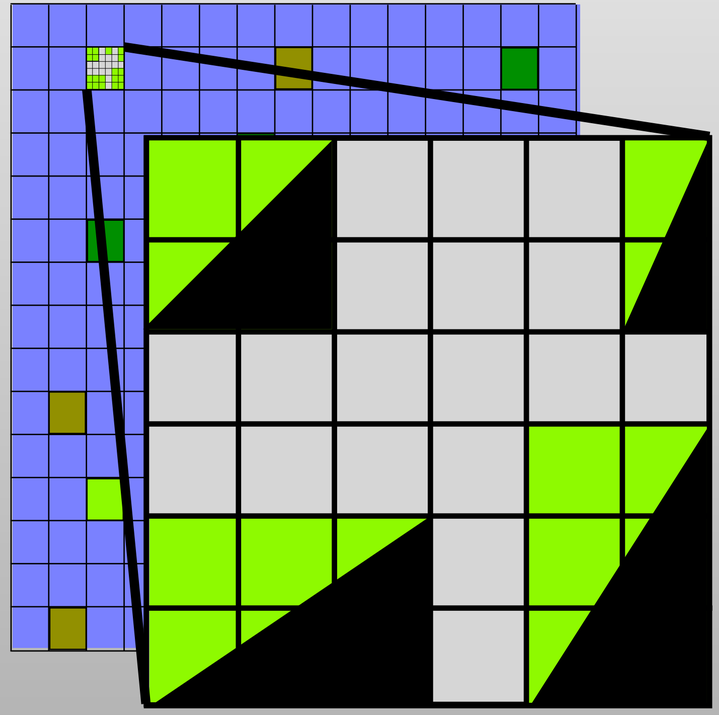


Eden Region
Survivor Region
Tenured Region
Humongous Objects
Free space

Collection Set



Remembered Set



Before

After

Appendix 7

G1GC - Default values - Printing

Print initial defaults for the operating system
(caution, this is a long list, best to redirect to a file)

```
java -XX:+PrintFlagsInitial -version
                    or
java -XX:+PrintFlagsInitial MyApplication
```

Print final defaults with overridden values
(caution, this is a long list, best to redirect to a file)

```
java -XX:+PrintFlagsFinal -version
                    or
java -XX:+PrintFlagsFinal MyApplication
```

Print current flags

```
java -XX:+PrintCommandLineFlags -version
                    or
java -XX:+PrintCommandLineFlags MyApplication
```

The `-version` is used as the executable above. As is shown, it can be replaced with any java class with a `main(...)`

Appendix 8

G1GC Logging

Appendix 8a

G1GC Logging - Unified logging in JDK 8+

Understanding the content in the table: `-Xlog : <tags to log>[=<log level>] [: <output> [: <decorations> ]]`

Option	Meaning
`-Xlog:gc`	Log messages with gc tag using info level to stdout, with default decorations
`-Xlog:gc,safepoint`	Log messages with either gc or safepoint tags (exclusive), both using info level, to stdout, with default decorations
`-Xlog:gc+ref=debug`	Log messages with both gc and ref tags, using debug level, to stdout, with default decorations
`-Xlog:gc=debug:file=gc.txt:none`	Log messages with gc tag using debug level to file gc.txt with no decorations
`-Xlog:gc=trace:file=gc.txt:uptimemillis,pids:filecount=5,filesize=1m`	Log messages with gc tag using trace level to a rotating logs of 5 files of size 1MB, using the base name gc.txt, with uptimemillis and pid decorations
`-Xlog:gc::uptime,tid`	Log messages with gc tag using info level to output stdout, using uptime and tid decorations
`-Xlog:gc*=info,safepoint*=off`	Log messages with at least gc using info level, but turn off logging of messages tagged with safepoint

Unified logging changes (for reference use): `java -Xlog:help`

Reference: https://www.slideshare.net/PoonamBajaj5/lets-learn-to-talk-to-gc-logs-in-java-9

Appendix 8b

G1GC Logging - Tags commonly used

Main tag is gc

Tag Type	Tag
Region	region
Liveness	liveness
Marking	marking
Remembered Set	remset
Ergonomics	ergo
Class Histogram	classhisto
Safepoint	safepoint
Task	task
Heap	heap
JNI	jni
Promotion Local Allocation Buffer	plab

Tag Type	Tag
Promotion	promotion
Reference	ref
String Deduplication	stringdedup
Statistics	stats
Tenuring	age
Thread Local Allocation Buffer	tlab
Metaspace	metaspace
Humongous Allocation	alloc
Refinement	refine
Humongous	humongous
String Symbol Table	stringtable