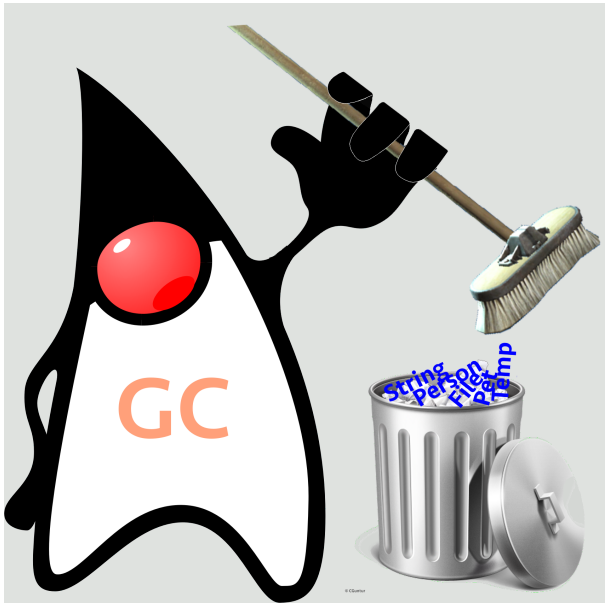
GARBAGE COLLECTION IN JAVA 9



Below icons are from SuperTinyIcons: https://github.com/edent/SuperTinyIcons

@CGuntur Chandra Guntur http://cguntur.me

about: presenter

Chandra Guntur

- coding since mid-nineties, mostly Smalltalk and Java.
- an evangelist and heavy user of Spring and Spring Boot.
- an organizer/presenter at NY Java Special Interest Group (javasig.com).
- a co-chair of NYJavaSIG Hands On Workshop (NYJavaSIG HOW).
- committed to *docendo discimus* (by teaching, you learn).
- a **Saganist** (the **g** not to be confused with a **t**). 😈

Twitter:
http://www.twitter.com/cguntur

LinkedIn:
http://www.linkedin.com/in/chandraguntur

My blog:
http://www.cguntur.me

LinkedIn and my blog QR codes generated at QRStuff: http://www.qrstuff.com.

AGENDA

Topics covered:

- Brief walk-through:
  - Garbage Collection - Basics
  - Garbage Collection - Patterns
  - Garbage Collection - Generational
- G1GC - How it works
- G1GC - Logging Options
- G1GC - Common Tuning Situations
- A discussion (Time permitting):
  - Garbage Collection - What's Next

GARBAGE COLLECTION - BASICS

As a broad definition, **garbage collection** is the process of:

- looking up managed memory
- identifying all objects that are in use (*live objects*)
- marking non-live objects as *garbage* (*unused* or *no-reference* are other common terms)
- [*occasionally*] reclaiming memory by deleting garbage
- [*occasionally*] compacting memory defragmenting live objects, to create more contiguous space.

An unused object is one that is no longer referenced by any part of the program.

Trivia:

The first garbage collection process: Lisp, in 1959 by **John MacCarthy** (*author of Lisp and major contributor to ALGOL*) fame.

GARBAGE COLLECTORS - CLASSIFICATION

Garbage collectors can be classified with several classifiers.

Based on how collection runs	Based on how objects are marked
• **Serial collector**: Single GC thread, halting all application threads → **STOP THE WORLD** • **Parallel collector**: Multiple GC threads in parallel, halting all application threads → **STOP THE WORLD** • **Concurrent collector**: GC thread(s) concurrently run with the application threads → **CONCURRENT** **Note**: Parallel and concurrent are two different things.	• **Precise**: When the collector can fully identify and process all references at the time of collection. • **Conservative**: When collector cannot/has not fully identified references to a certain object, it assumes a reference to it exists.
Based on run interval of collections	**Based on what collection does to objects**
• **All-at-once**: Entire GC operation performed in a single run. • **Incremental**: GC operations are sliced into smaller operations, to be more performant.	• **Moving**: When reachable objects are moved to a new area of memory (a.k.a compacting). • **Non-moving**: When unreachable objects are simply released (a.k.a non-compacting).

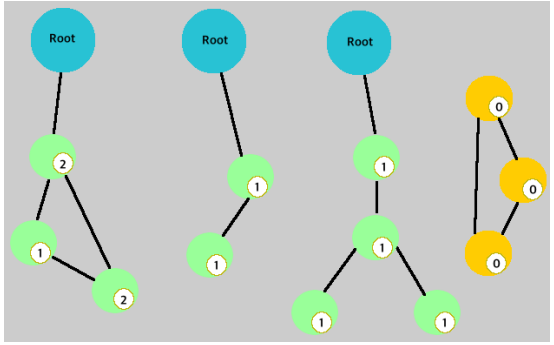
GARBAGE COLLECTION - PATTERNS

This section will list a few garbage collection patterns.

GARBAGE COLLECTION - REFERENCE COUNTING

The most rudimentary garbage collection is based on *references*:

- associates a reference counter to each object
- increments the counter for each reference to the object.
- decrements the counter for each de-reference of the object.
- marks object as garbage if reference count reaches zero.
- is very under-performant and maintenance-heavy for JVMs.
- is considered obsolete in commercial JVMs.

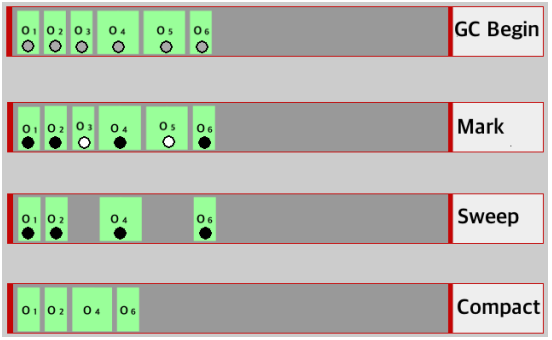


Memory allocator may place objects anywhere in the free space.

GARBAGE COLLECTION - MARK/SWEEP/COMPACT 5.3

The mark/sweep collection pattern is based on *tracing*:

- includes a *tri-color marker* on each object.
- all markers **painted gray** before the collection cycle.
- sets marker as **used (black)** or **unused (white)**.
- each object checked only once due to the marker being set.
- deletes the **unused (white)** marked objects.
- collection ends when the **gray set** is empty.
- (optionally/periodically) compact or defragment space for new objects.
- more efficient for larger memory areas.
- works well on longer-lived non-volatile objects.
- process efficiency is linearly proportional to the live objects.
- uses **garbage collection roots** to initiate marks:
  - Active threads including the green/main thread.
  - Local and static variables.
  - JNI references.
- incremental collector, doesn't need a lot of free space to run.
- can be either concurrent or stop-the-world collecting.



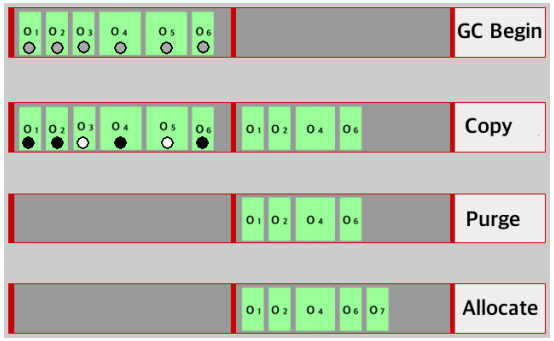
Memory allocator moves pointer to the beginning of free space.

Objects are added sequentially to where the free memory pointer is.

GARBAGE COLLECTION - COPYING

The copying garbage collection is based on *tracing*:

- marks objects as used or unused.
- copies live objects to another memory area.
- purging current area, one of the two areas is always empty.
- very specialized case of **M/S/C**, the sweep and compact are one step.
- needs a **from** and a **to** memory area.
- has the **from** and **to** swap roles each run of the collection.
- is efficient for small memory areas.
- works best on short-lived objects.
- all-at-once collector that needs 2x the memory space to run.
- stop-the-world collector.



Memory allocator moves pointer to the beginning of free space in the new memory area.

Objects are added sequentially to where the free memory pointer is.

GARBAGE COLLECTION - GENERATIONAL

Java, since JDK 5 has had a generational garbage collector as its default.

The generational collector divides the memory into smaller memory areas.

Management of objects in different memory areas is done differently.

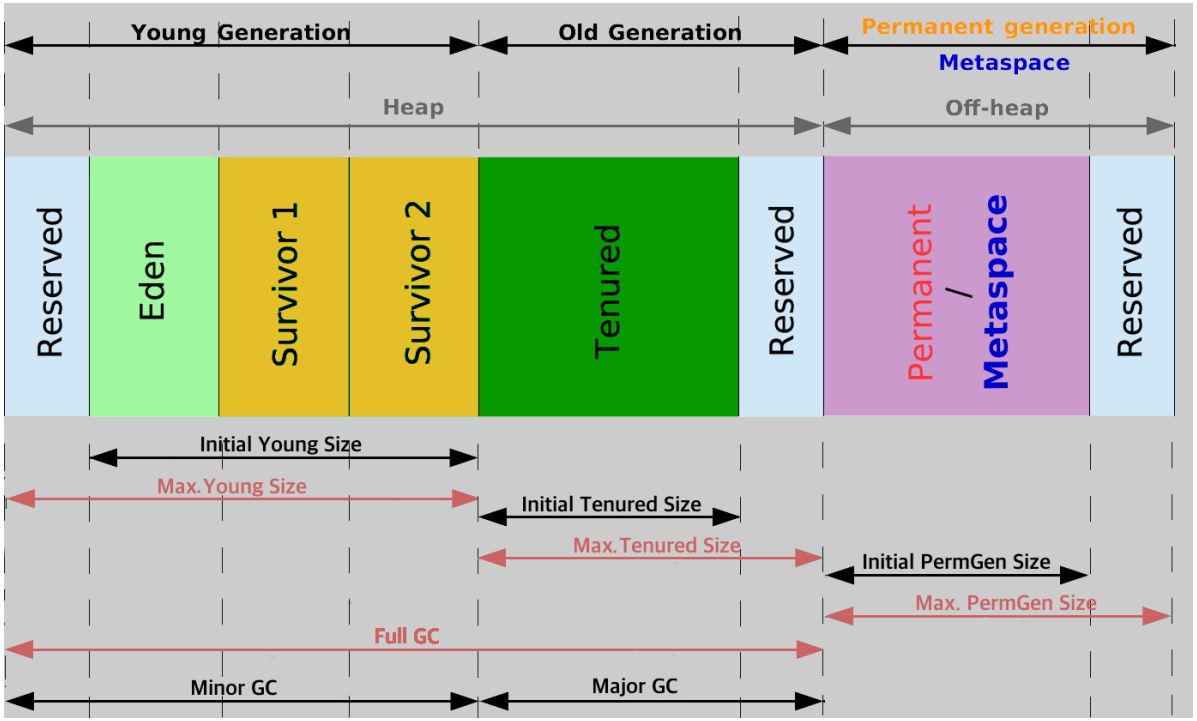
Weak Generational Hypothesis :

- Most objects do not have a long life.
- It is usually rare for an older generation object to reference a young generation object.

GENERATIONAL GC - PICTORIAL

Overview :

- Objects initially allocated to the young **Eden** region.
- Objects that live past a collection are *promoted* to **Survivor** and subsequently to **Tenured**.
- Collection patterns can be different for each generation to deliver an optimal performance.



Picture influenced by Jörg Prante's writeup: http://jprante.github.io/2012/11/28/Elasticsearch-Java-Virtual-Machine-settings-explained.html

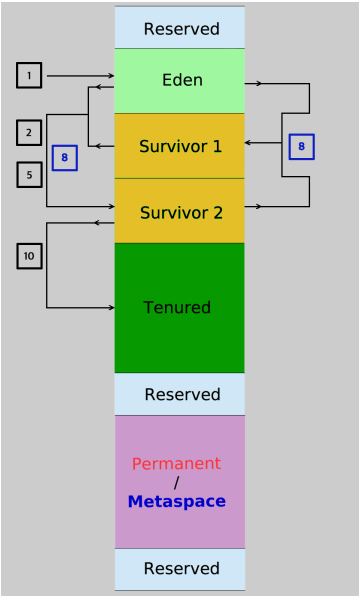
GENERATIONAL GC - COLLECTION

Explanation:

- heap is broken into smaller areas (sub-heaps).
- newly created objects are allocated to a young area called **Eden**.
- Eden itself divided into TLAB sections for individual threads and a common area.
- different GC processes for each generation, young gc typically is all-at-once, stop-the-world, copying collector.
- objects that stay "alive" longer get moved to older generation.
- **Permanent Generation** is for storing class/method definitions and static content.

Generational Collection Summary

1. Objects first allocated into **Eden**.
2. On first GC, live objects from Eden moved to **Survivor 1**.
3. All moved live objects from Eden marked with an **Age (tenure)** of 1.
4. Eden cleared and ready to allocate new objects.
5. Next GC, collects live objects from Eden and Survivor *From* into **Survivor *To***.
6. Objects from Survivor *From* get Age incremented by 1, while objects from Eden get an Age of 1.
7. Eden cleared and ready to allocate new objects.
8. Steps 5, 6, 7 recur flipping Survivor 1 and Survivor 2 as *From* and *To*.
9. Done until Age threshold (default = 15) reached for any live object.
10. Next GC, live objects from the current Survivor with Age above threshold, move to **Tenured**.
11. Steps 5 - 10 recur until a need for a full GC.



GENERATIONAL GC - TYPES - OLDER

There are a few different types of collectors:

- **Serial Collector**
  - Freezes all application threads to collect garbage.
  - Designed for single-thread and tiny heap sized (~100 MB) apps.
  - Low memory footprint makes it good for for mobile or small apps.

- **Parallel Collector a.k.a Throughput collector - Java 1.5 onwards, default collector Java 1.5, 1.6, 1.7 and 1.8***
  - **Young** gen., only has the parallel (scavenge) collection.
  - **Tenured** gen. earlier had a default serial collector (later versions changed default to parallel old collector).
  - * *G1GC was originally planned as default for Java 1.8 but was deferred until Java 9*.

- **Concurrent Mark-Sweep (CMS) Collector (mostly*) - available Java 1.5 onwards until Java 1.8**
  - By default, **Young** gen. uses a serial collection and **Tenured** gen. use a CMS collector.
  - Does not compact by default, to make GC a low-pause, low-latency, but causes heap fragmentation.
  - * *An eventual fill up of old memory causes a fall-back to a STW M/S/C that results in performance degradation*.
  - **Focuses on live objects and defers garbage handling, until crisis time.**

GENERATIONAL GC - TYPES - WHY NEW

Some notes on the existing garbage collectors:

- Mostly focused on live objects, not on garbage (an exact antithesis of the name).
- The contiguous memory area arrangement reduces flexibility of "moving the walls".
- Required different collectors on **young** and **tenured generations** to achieve performance benefits.
- By default, unpredictable and inconsistent pause-times and lesser number of STW GCs.
- Need heavy tune-ups to stabilize pause times due to non-compacting nature of recent collectors.
- Not performant for large memory heaps (large heaps are common these days).
- *Either* highly prone to fragmentation *or* demand the need for smaller heaps for predictable times.
- Not much effort put into re-use of duplicated content.

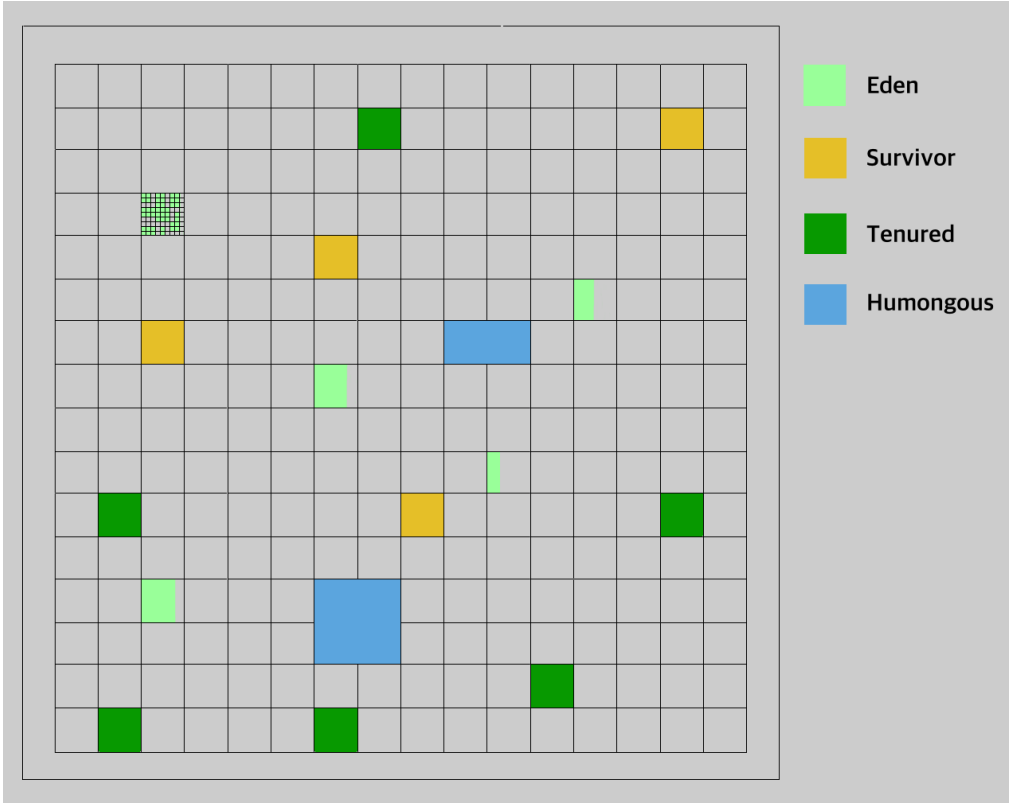
The G1GC (**Garbage First Garbage Collector**), was designed and developed to resolve the above.

G1GC is the new default collector since JDK 9.

GARBAGE FIRST GARBAGE COLLECTOR

Java 9 introduced a new default Generational Garbage Collector.

This new collector is called Garbage First Garbage Collector (G1GC).

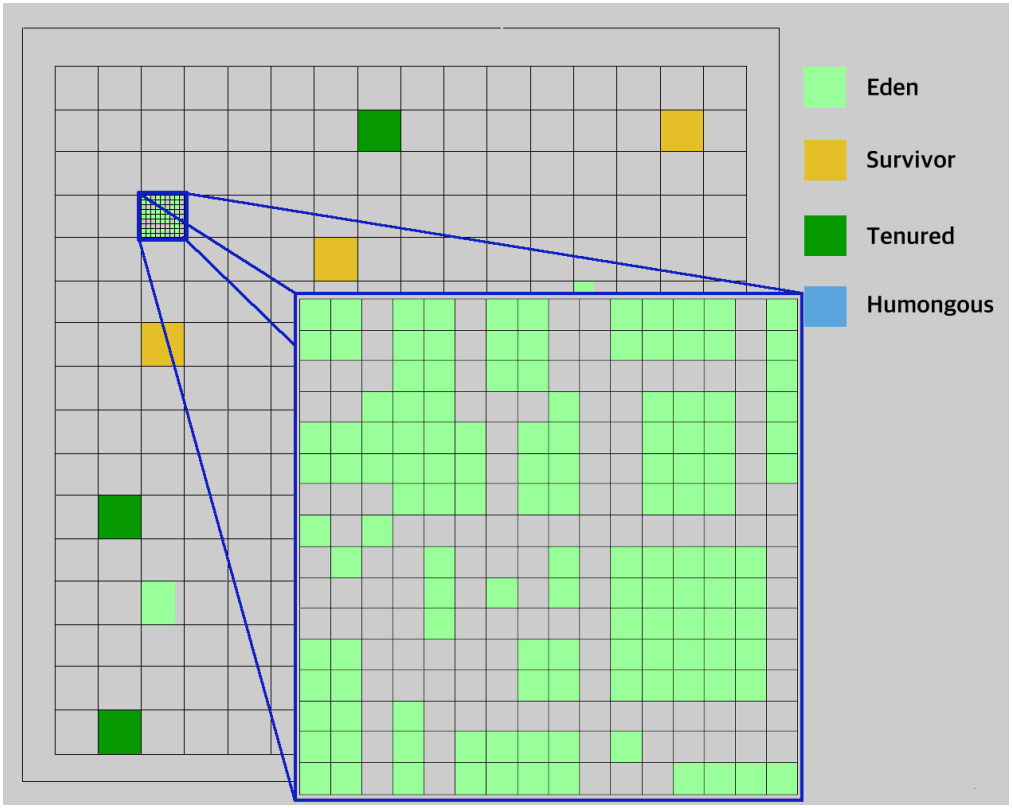


Recommended slide deck: https://www.slideshare.net/MonicaBeckwith/java-9-the-g1-gc-awakens.

GARBAGE FIRST GARBAGE COLLECTOR - REGIONS 7.2

Each block of memory is called a **Region**.

Heap is compartmentalized into virtual non-contiguous memory regions.



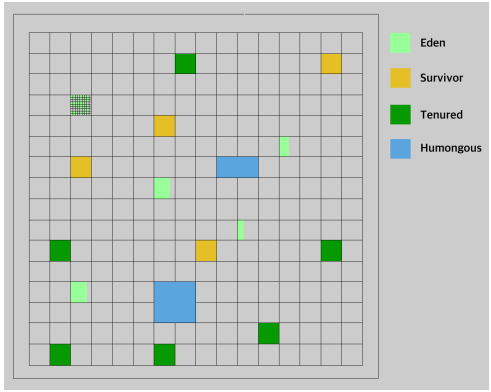
G1GC - REGIONS

Some observations **:**

- A **region** is a contiguous unit of memory allocation and space reclamation.
- Regions are formed by **dividing the heap into ~ 2048 or more blocks** of **equal size**.
- The region **sizes can range from 1 MB to 32 MB** depending on the heap size.
- The **memory manager** assigns a region as free, eden, survivor, tenured or part of a humongous allocation.
- **Humongous** objects occupy complete (*and if needed, contiguous*) regions; defined as objects with **size > 50% of region size**.
- **Humongous** objects are **not allocated in** young regions.
- G1GC has a **consistent pause time-target that it tries to meet** (soft, real time), hence region sizes are controlled.

Math Stuff: Calculating regions on a 9GB heap

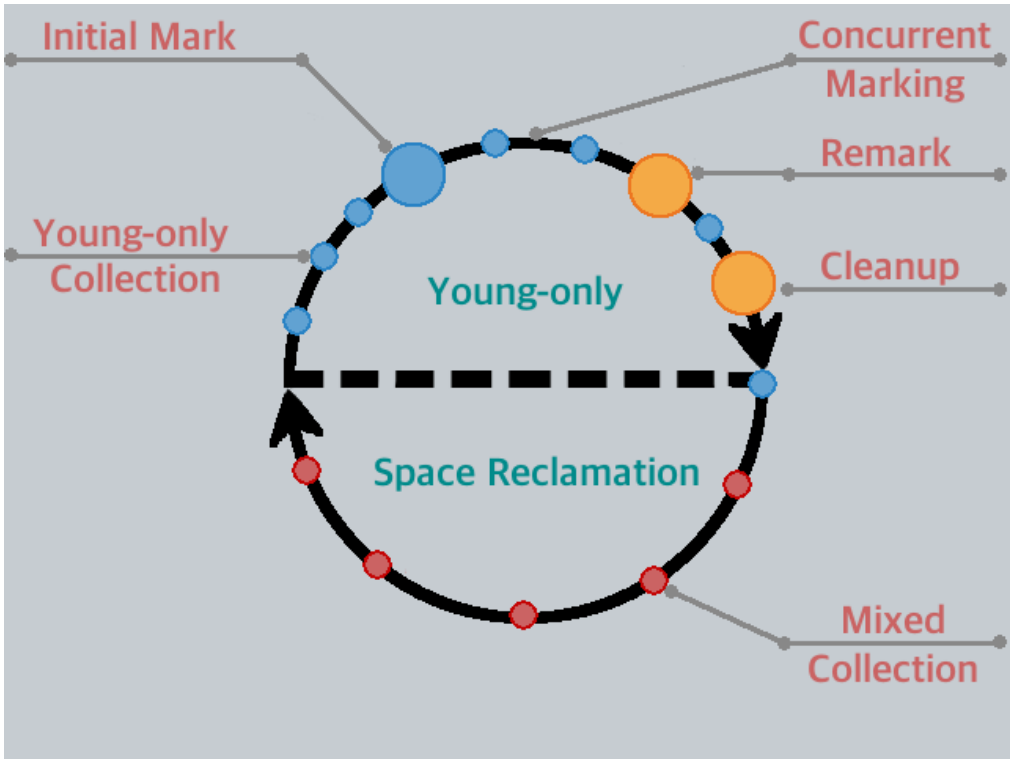
1. Convert heap size to MB → 9GB = 9 x 1024 MB = **9216 MB**.
2. Approximate region size for 2048 regions → 9216 MB ÷ 2048 regions = **4.5 MB per region**
3. Regions for factor of 2 *below* the approximate region size (= 4) → 9216 ÷ 4MB = **2304 regions**.
4. Regions for factor of 2 *above* the approximate region size (= 8) → 9216 ÷ 8MB = **1152 regions**.
5. Since **2304** is closest to and above 2048, it is chosen as the number of regions to use.
6. Each region is thus of **4**MB size.
7. Possible to disable this auto-calculation by setting a JVM option that overrides calculating.
8. Set the flag `-XX:G1HeapRegionSize` with a numeric value (that is a power of 2).
9. Watch out for **Humongous** objects. In this case any object **2MB** or above.



G1GC - EXPLANATION (PICTURE)

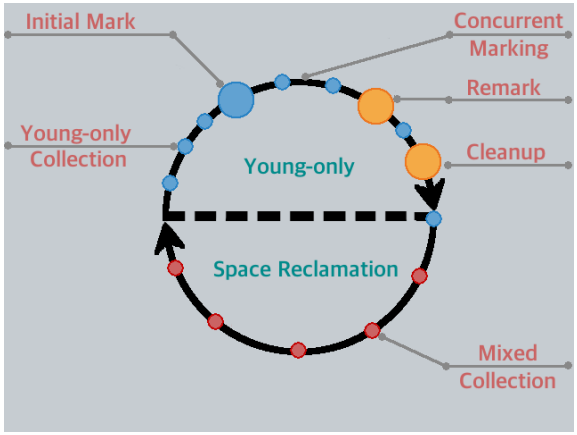
Below is a pictorial of how G1GC works under *normal* circumstances. ***Some notes*** **:**

- The top semicircle is the most common GC process.
- When space is too fragmented or tenured occupancy is high, the bottom semi-circle is triggered.
- Heavy heap occupancy will eventually lead to a Serially collecting **Full GC** (single-threaded).



G1GC - EXPLANATION

Explanation:



- *each circle* represents a STW process.
- two distinct phases of collection.
- starts with a **Young-only** phase.
- **Young Collection** runs periodically on a **Collection set (CSet)**.
- continues until a *threshold* is reached, which triggers an **Initial Mark**.
- Initial Mark triggers concurrent marking in addition to the young collection.
- Root scanning will halt the young collection process during this phase.
- **Concurrent Mark** takes a while, Young Collection could continue alongside.
- at end of Concurrent Mark, a **Remark** is initiated.
- Remark finalizes marking, summarizes liveness.
- Remark completion triggers **Cleanup**, reclaiming empty regions.
- Cleanup is used to determine if Tenured space reclamation required.
- Crossing Heap waste percentage threshold triggers **Space reclamation** phase.
- A **Collection set (CSet)** of Young & Tenured regions picked for collection.
- This **Mixed Collection** is made up to meet a *pause-time goal*.
- References to objects (in same generation or inter-generational) are tracked.
- References are tracked via **Remembered sets (RSets)** in a Card table.

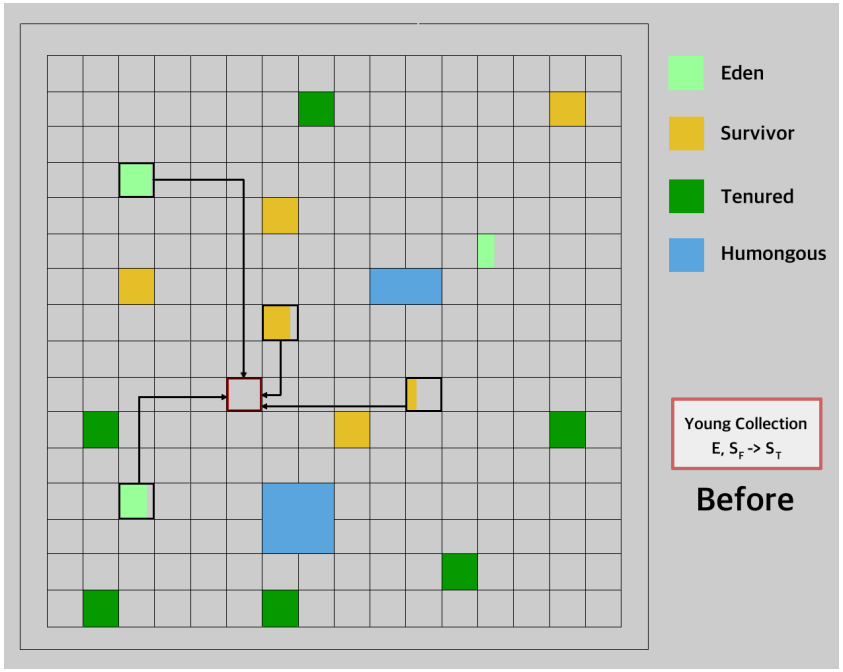
G1GC - YOUNG COLLECTION - STEPS

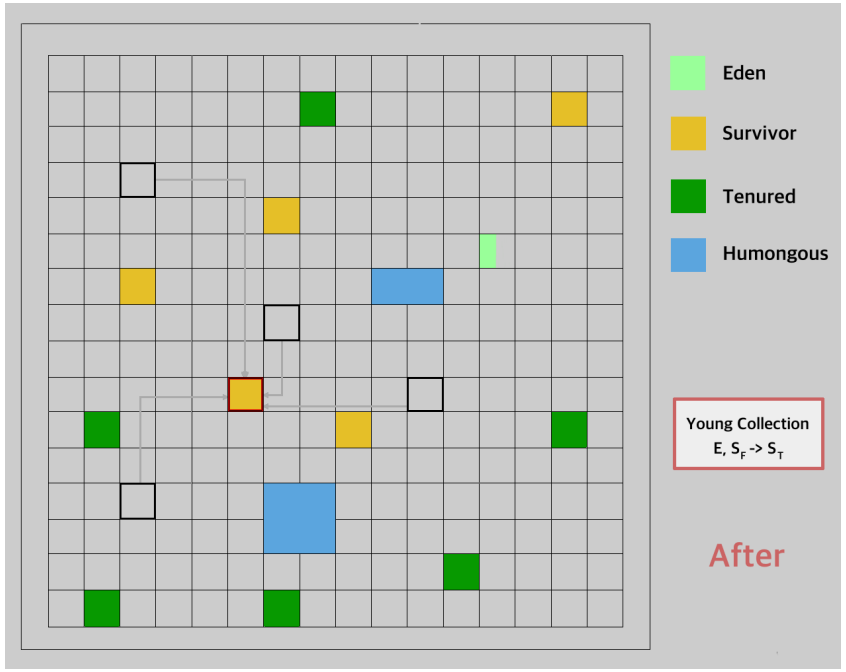
1. **Root Scanning**: Scan local and static objects for root objects and mark in a *"dirty queue"*.
2. **Update Remembered Set (RSet)**: All marked references in the dirty queue updated into a **RSet**.
3. **Process RSet**: Detect references to objects in the *collection set* **Young** regions, from objects in **Tenured** regions.
4. **Copy Live Objects**: Traverse the object graph and *promote/age* live objects.
5. **Process references**: Update references to new location, process soft, weak, phantom and final references.
   - **StrongReference**: Typical pointer allocation to an object, re-pointing to null will cause object to be collected.
   - **SoftReference**: The reference is kept unless there is no other space for any new allocation.
   - **WeakReference**: The reference is collected as soon as GC reaches it.
   - **PhantomReference**: The reference enqueued for collection, retained until all references to it are either dead/weak references.

G1GC - YOUNG-ONLY #1 (SURVIVOR PROMOTION) 7.7

1. Young collection (Eden & Survivor$_F$ → Survivor$_T$):

- Young Collection builds a CollectionSet with some Eden and Survivor regions.
- Young Collection eventually targets all young regions.
- Steps: Root scanning ↠ Update the Remembered Set (RSet) ↠ Process RSet ↠ Copy live objects ↠ Process references.
- Live objects from scavenged Eden and Survivor *From* regions are compacted and copied into a Survivor *To* region.
- The Age of live objects being moved from Eden is set to 1 with each execution of this GC.
- The Age of live objects being moved from Survivor *From* is incremented by a 1 with each execution of this GC.

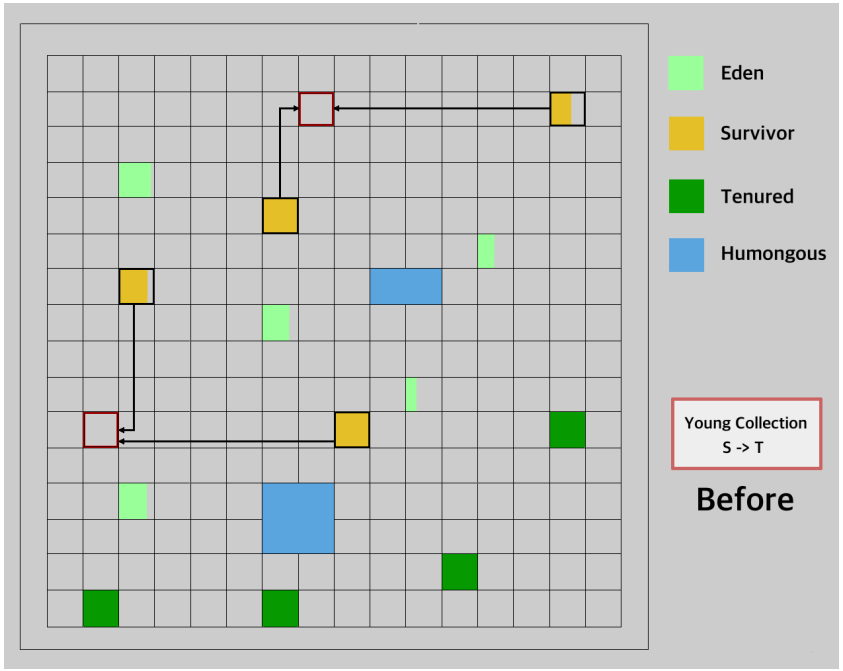


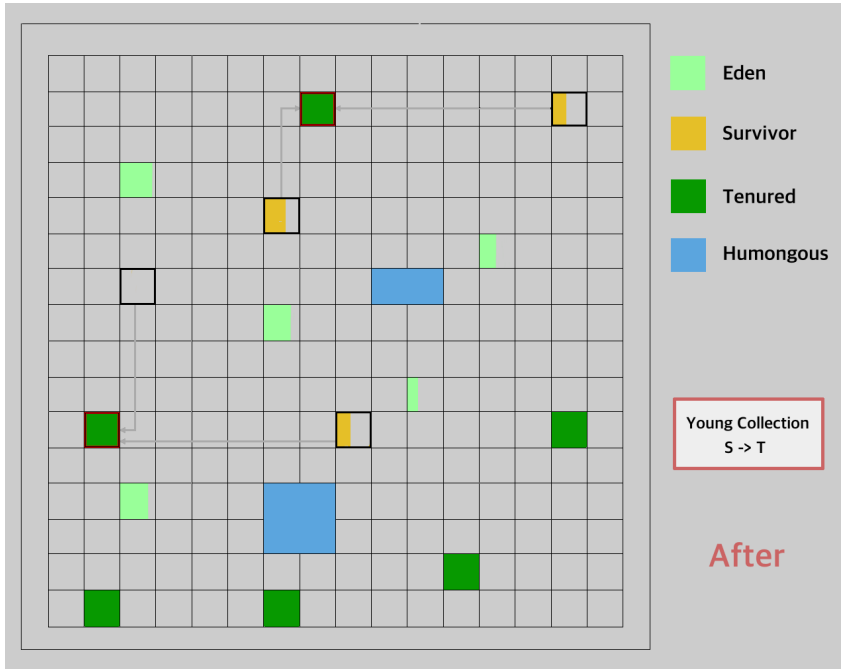


G1GC - YOUNG-ONLY #2 (TENURED PROMOTION) 7.8

2. Young collection (Survivor → Tenured):

- Young Collection builds a CollectionSet with some Eden and Survivor regions.
- Young Collection includes compact copying from Survivor to Tenured after objects survive a few iterations.
- Steps: Root scanning ↠ Update the Remembered Set (RSet) ↠ Process RSet ↠ Copy live objects ↠ Process references.
- All objects in the collecting Survivor regions with Age *below* threshold (default = 15) are left as-is.
- All objects in the collecting Survivor regions with Age ***above*** threshold (default = 15) are collected and moved to **Tenured** region.

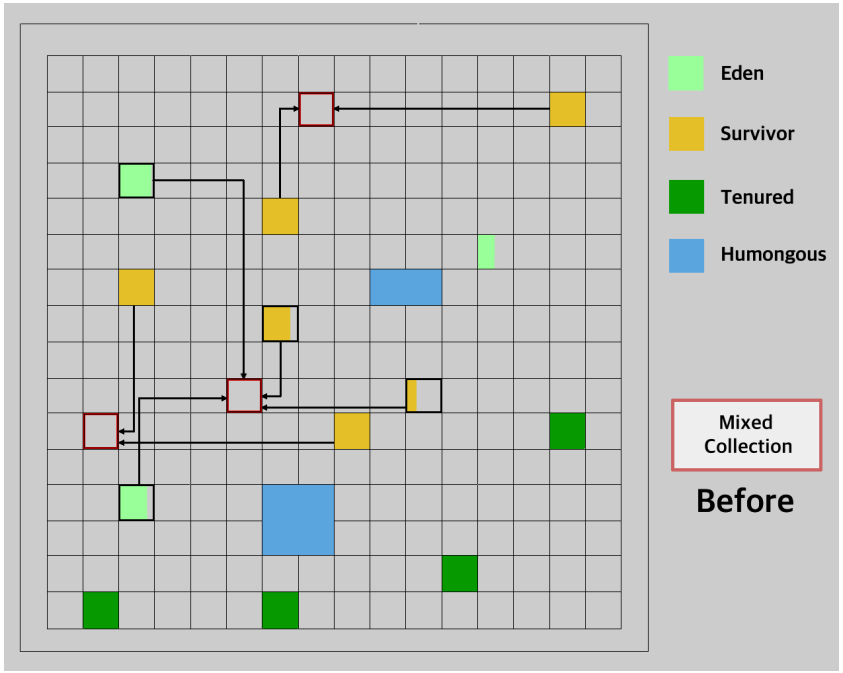


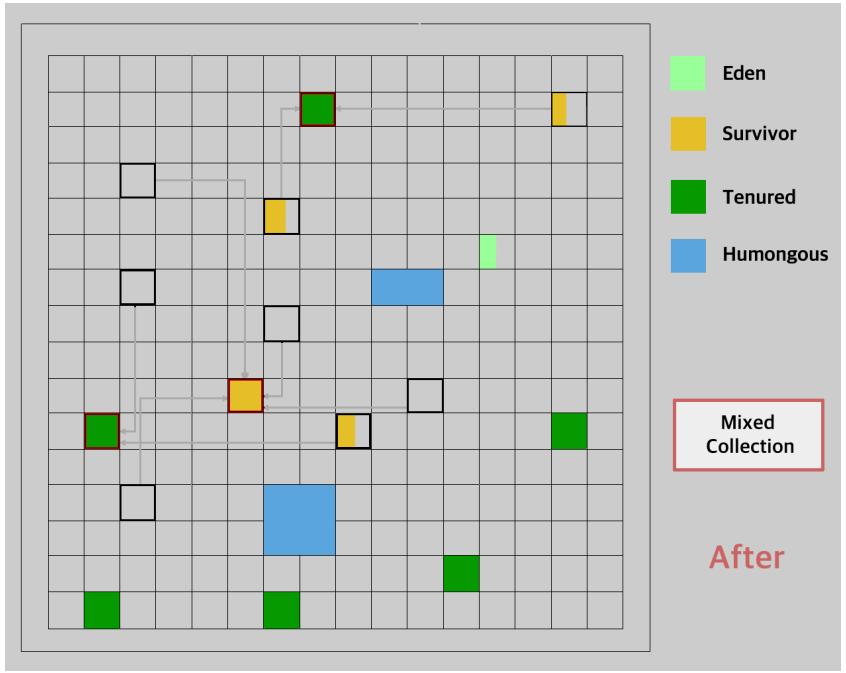


G1GC - MIXED COLLECTION

3. Mixed Collection:

- A Mixed Collection is triggered when the percentage of waste (garbage) in the heap reaches a certain threshold.
- Mixed Collection involves picking up a few regions (Eden, Survivor and a few Tenured) for collection.
- The combination is made up based on estimated sum of pause-times of the **Collection Set**.
- Several cycles of Mixed Collection may run, until the heap waste (garbage) percent is below the set threshold.
- After each collection, the liveness of the Tenured region objects are re-evaluated.

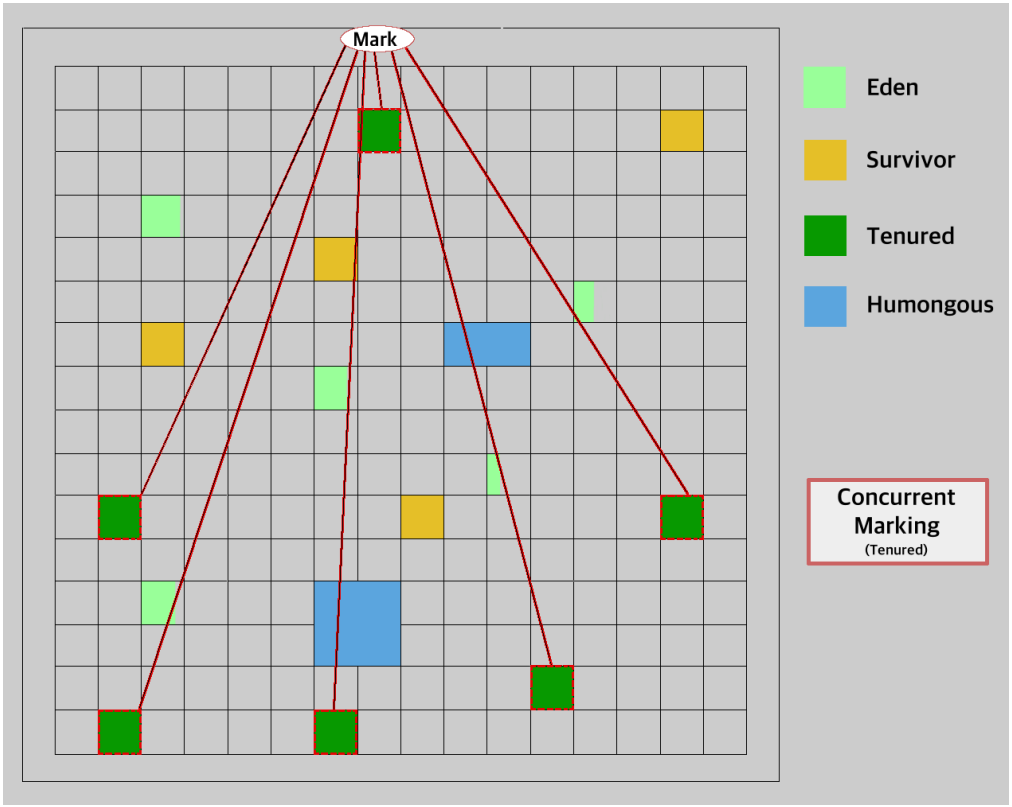




G1GC - CONCURRENT MARKING

4. Concurrent Marking:

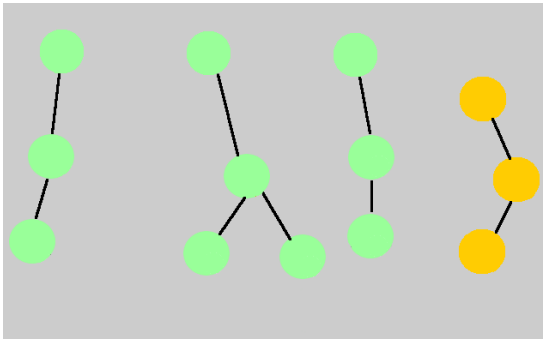
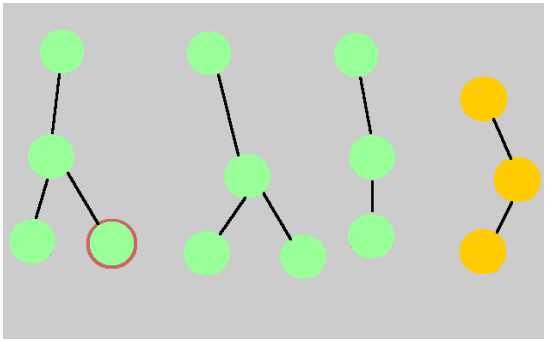
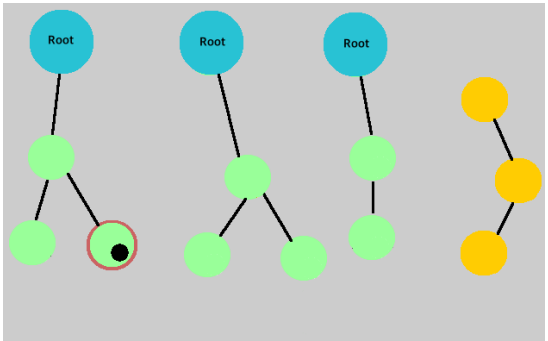
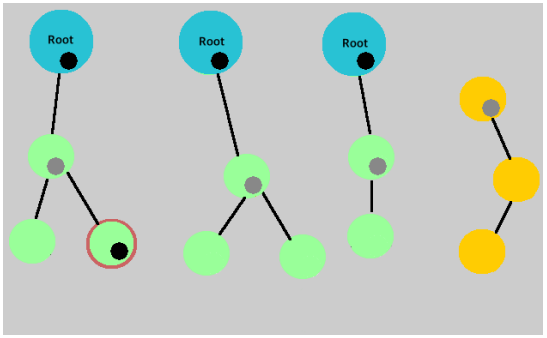
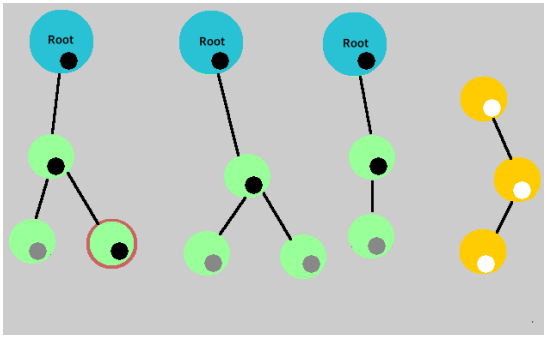
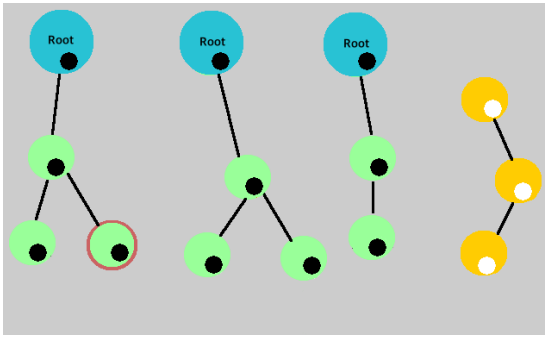
- Triggered when certain percentage of heap is occupied - or - when the Survivor *To* minimum creation space threshold is reached.
- Concurrent Mark begins with a **Snapshot-At-The-Beginning (SATB)** (conservative garbage collection).
- If more objects move to Tenured after the snapshot, all such objects are *implicitly considered live*, and not checked for references.
- The completion of the Concurrent Mark triggers a **Remark** and a subsequent **Cleanup** activity.
- Young Collection continues during Concurrent Mark, Remark and Cleanup.



GARBAGE COLLECTORS - CONCURRENT MARKING PICTORIAL

The below pictures show a concurrent mark liveness check (tri-color painting).

Gray = To check, **Black** = Live, **White** = Dead.

1. Snapshot-at-the-beginning	2. New Object added	3. Root Scan
		
4. Root Painting	**5. Children Painting**	**6. Completion**
		

G1GC - METASPACE

Some notes on MetaSpace versus PermGen:

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

- Metaspace (or rather metaspaces) are not a panacea for `OutOfMemoryError`s.
- Metaspace is explicitly bounded from the O/S memory, taking up unlimiited amounts otherwise.
- Initial Metaspace Size is set by `-XX:MetaspaceSize` (replaces -XX:PermSize), default = 21.8M.
- Max Metaspace is set by `-XX:MaxMetaspaceSize` (replaces -XX:MaxPermSize), default = unlimited.
- When porting from PermGen, simply replace `-XX:PermSize` and `-XX:MaxPermSize` with the new options.

G1GC - GORY DETAILS

Explanation:

- **Pause time** is used to calculate the mix and is controlled by `-XX:MaxGCPauseMillis` (default = 200).
- **Pause time intervals** controlled by an *ergonomic* goal that is not initially set.
  - controlled by `-XX:GCPauseTimeInterval` (no default).
- G1 continues with the **young collection** **until either of the below is reached**:
  - reaches a configurable *soft* limit known as the `-XX:InitiatingHeapOccupancyPercent` (default = 45).
  - reaches the configurable *strict* limit of `-XX:G1ReservePercent` (default = 10).
- If either constraint is met, it triggers the start of a Concurrent GC.
- Concurrent Mark includes both STW and concurrent activities.
- Concurrent Mark determines liveness of objects on a per-region basis.
- G1 goes after regions that have the most garbage, it is called **Garbage-First**.
- Concurrent Mark is followed by a Remark and a Cleanup.
- Cleanup activity is used to determine if a Space Reclamation is needed.
- Checks the percentage of garbage (waste) space to be below `-XX:G1HeapWastePercent` (default = 5).
- Collector picks up a minimum number of regions based on `-XX:G1MixedGCCountTarget` (default = 8).
  - The total number of tenured regions are divided by the above number and are picked up for collection.

G1GC - WHAT TRIGGERS INITIAL MARK

What triggers the Initial Mark :

- **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.

- **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.

- **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 a `-XX:-G1EagerReclaimHumongousObjects`, may need to turn on Experimental options.

GENERATIONAL GARBAGE COLLECTION - SUMMARY[8]

Common JVM options to use a specific collector:

Type	Young GC	Tenured GC	JVM Option
Serial GC	Serial	Serial	`-XX:+UseSerialGC`
Parallel GC	Parallel Scavenge	Parallel	`-XX:+UseParallelGC -XX:+UseParallelOldGC`
CMS GC	~~Parallel New~~	~~CMS~~	`-XX:+UseParNewGC -XX:+UseConcMarkSweepGC`
G1 GC	G1GC		`-XX:+UseG1GC`

NOTE: CMS is deprecated as of Java 9.

Differences in collectors:

Type	Parallel	Concurrent	Young GC	Tenured GC	Feature
Serial GC	-	-	Serial	Serial	Batch processing
Parallel GC	Yes	-	Parallel	Parallel	High throughput
CMS GC	~~Yes~~	~~Yes~~	~~Parallel~~	~~Parallel & Conc.~~	~~Low Pause~~
G1 GC	Yes	Yes	Parallel	Parallel & Conc.	Low pause & High throughput

Recommended reading material: https://blogs.oracle.com/jonthecollector/our-collectors.

G1GC - LOGGING OPTIONS

In this section salient logging setup options are discussed.

G1GC - CHECK DEFAULT VALUES

G1GC has several options that can be tuned for performance.

Finding out what flags exist and what their default values are, is important.

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

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

Print final defaults for the jvm, with overrides on the defaults (caution, this is a long list, best to redirect to a file):

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

Print current flags:

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

The `-version` is used as the executable above. This can be replaced with any java class with a `main(...)` as well.

G1GC - LOGGING G1GC PROCESSES

Common JVM options to print GC logs:

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

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

GC Type	Option	Meaning
Pre-G1GC	-Xloggc:/path/to/gc.log	Destination path for the logs.
Pre-G1GC	-XX:+PrintGCDetails	Increases the verbosity of logged content.
Pre-G1GC	-XX:+PrintGCDateStamps	Log date and timestamp of the collection.
G1GC	`-Xlog:gc`	Log messages with **gc** tag using **info** level to stdout, with default decorations.
G1GC	`-Xlog:gc,safepoint`	Log messages with *either* **gc** *or* **safepoint** tags (exclusive), both using 'info' level, to stdout, with default decorations.
G1GC	`-Xlog:gc+ref=debug`	Log messages with *both* **gc** *and* **ref** tags, using **debug** level, to stdout, with default decorations.
G1GC	`-Xlog:gc=debug:file=gc.txt:none`	Log messages with **gc** tag using **debug** level to file **gc.txt** with **no** decorations.
G1GC	`-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.
G1GC	`-Xlog:gc::uptime,tid`	Log messages with **gc** tag using **info** level to output **stdout**, using **uptime** and **tid** decorations.

G1GC - COMMON TAGS

Some common tags used in logging:

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

G1GC - COMMON TUNING SITUATIONS

In this section some tuning options for common issues are discussed.

Before we get there …

JVM optimizes for classes of devices. A **server-class machine** is defined as one with:

- two or more physical processors
- two or more GB of physical memory

Some defaults for a server-class machine:

1. Default garbage collector = **G1GC**.
2. Initial heap size defaults to 1/64th of physical memory.
3. Maximum heap size defaults to 1/4th of physical memory.
4. Default tiered compiler with **C1** and **C2** code caches.
    - C1 pre-compiles and optimizes non-profiled (non-dynamic) code.
    - C2 profiles code and defers optimizations.

JVM uses *ergonomics* to determine behaviour and environment-based heuristics to improve performance.

G1GC - PERFORMANCE DEFINITIONS

G1GC tuning is done to meet one of the below performance metrics:

- **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

G1GC - HOW TO TUNE

There are a few steps to performance tuning happiness.

Follow the EMPATHY model.

1. Execute - execute the application to determine issues visually.
2. Monitor - use appropriate alerts/tools/logs to monitor.
3. Profile - identify the areas that need special attention.
4. Analyze - determine what needs to be done to fix issues.
5. Tune - set the right parameters for sizes, ages, threads etc.
6. Hammer - test out each set of parameters thoroughly.
7. Yippee - beer time. 🍻

Amdahl's Law:

If you decide to increase the number of parallel threads to gain performance, remember that:

The nature of this overhead (in parallelism) appears to be sequential so that it is unlikely to be amenable to parallel processing techniques.

Chandra's Law:

Providing three copies of a book does not get a person read it three times faster.

G1GC - FREQUENT FULL GC

Frequent Full GCs are observed

- Usually caused by heavy heap occupancy.
- Logs contain the phrase *Pause Full (Allocation Failure)*.
- This is typically preceded by a to-space exhausted message.
- Steps to mitigate:
  - Try to reduce the humongous objects.
  - Increase the java heap region size (by `-XX:G1HeapRegionSize`).
  - Increase number of concurrent threads (by `-XX:ConcGCThreads`).
  - Force earlier marking by either:
    - Lower the `-XX:G1ReservePercent`
    - Disable the `-XX:G1UseAdaptiveIHOP` and manually set `-XX:InitiatingHeapOccupancyPercent`.
- Full GCs can also be caused by `System.gc()` calls in some library.
  Effects of such can be mitigated by:
  - Full GC frequency can be mitigated by `-XX:ExplicitGCInvokesConcurrent`.
  - Last resort, completely ignore `gc()` calls with `-XX:DisableExplicitGC`.

G1GC - LONG YOUNG COLLECTION

Young Collections seem to take too long

- Young collection time is proportional to the size of the young generation.
- Reducing the `-XX:G1NewSizePercent` reduces the young generation size.
- Sudden spikes in the application may cause influx of live objects.
- Limiting the maximum size of the young generation can help.
- Maximum young generation size can be controlled by `-XX:G1MaxNewSizePercent`.

G1GC - LONG MIXED COLLECTION

Mixed Collections seem to take too long

- Determine which generation is taking the time (Set `gc+ergo+cset=trace` in logging).
- The logs will then show *predicted young region* and *predicted old region* times.
- Spread reclamations to more mixed collections via `-XX:G1MixedGCCountTarget`.
- Limit threshold of collecting regions with high live occupancy: `-XX:G1MixedGCLiveThresholdPercent`.
- Limit space reclamation in high occupancy regions via `-XX:G1HeapWastePercent`.

WHAT'S NEXT IN GARBAGE COLLECTION?

Shenandoah (Redhat ⇢ OpenJDK)

- Evacuation and reference updates to run concurrently with application threads.
- Pause times are meant to be independent of heap size.
- More reading material:
  - JEP-189 (http://openjdk.java.net/jeps/189)
  - Shenandoah Wiki (https://wiki.openjdk.java.net/display/shenandoah/Main)

Z Garbage Collector (Oracle ⇢ OpenJDK)

ZGC is a concurrent, currently single-generation, region-based, incrementally compacting collector.

- Handle multi-terabyte heaps.
- 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:
  - OpenJDK Project (http://openjdk.java.net/projects/zgc/)
  - ZGC Wiki (https://wiki.openjdk.java.net/display/zgc/Main)
  - Per Liden (Lead) Interview (https://jaxenter.com/zgc-interview-per-liden-139985.html)

Time's up :: ¯_(ツ)_/¯

My contact information

Twitter:
http://www.twitter.com/cguntur

LinkedIn:
http://www.linkedin.com/in/chandraguntur

My blog:
http://www.cguntur.me

LinkedIn and my blog QR codes generated at QRStuff: http://www.qrstuff.com.