DEVOXX™
the java™ community conference

G1 Garbage Collector

Jaromir Hamala <jhamala@c2b2.co.uk>

Senior Consultant

C2B2

@c2b2consulting

Jaromir Hamala

Who Am I?

- A curious developer
- Who likes to know more
  - About computers
  - World
  - And beer!

Agenda

- Memory Management
- Garbage Collectors in JVM
- CMS vs. G1
- G1 Tuning
- Recommendations

Manual Memory Management I

- Not all the memory is yours!
- You need to ask for it

  malloc() -> sbrk()

- And return when you no longer need it

  free() -> sbrk()

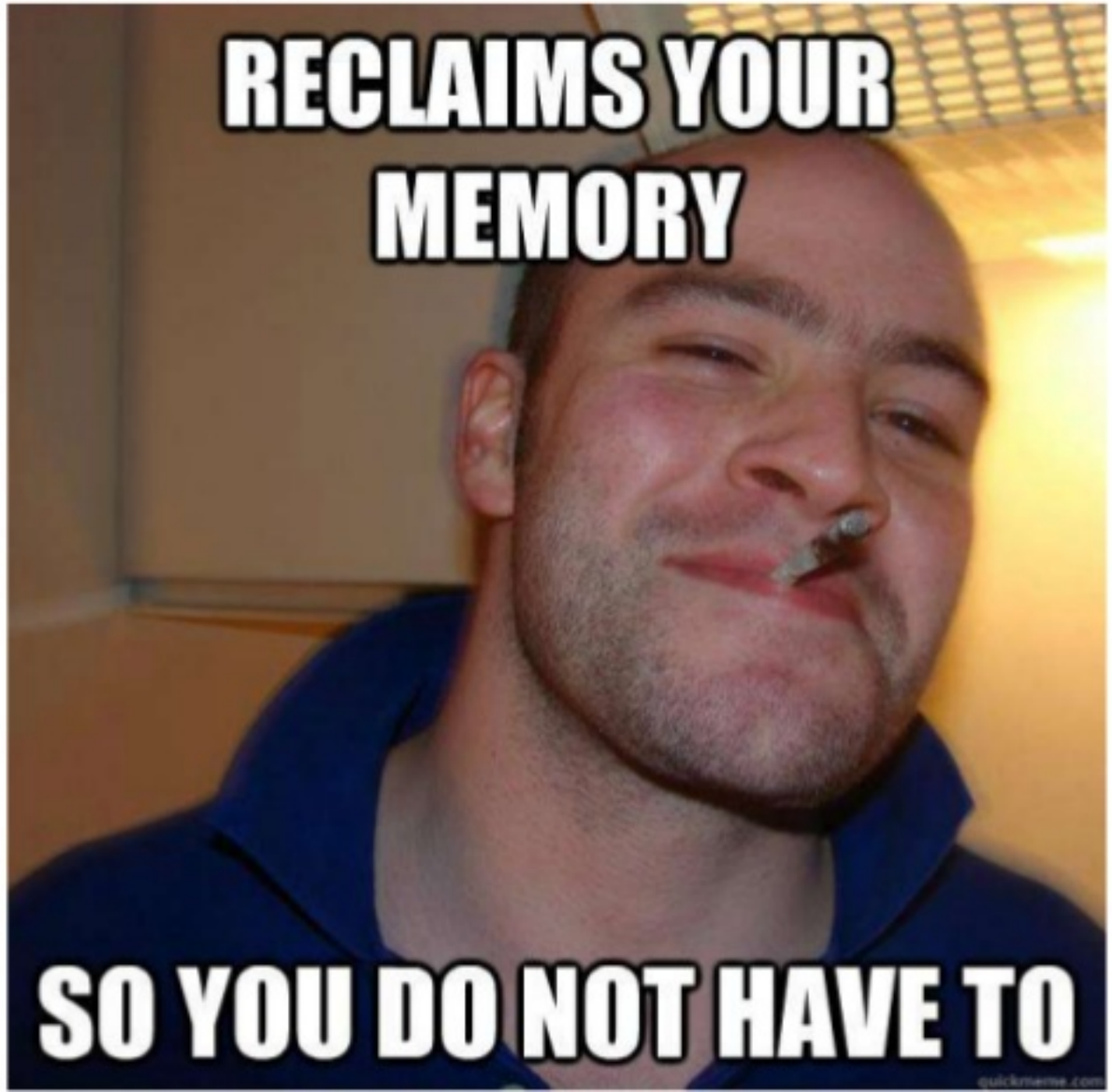
Manual Memory Management II

- Asking for something is easy

- Returning is way more complicated!

(we all know it!)

Garbage Collector I



A little bit of history

- **3000 B.C.** – "*The first landfill is developed when Knossos, Crete digs large holes for refuse. Garbage is dumped and filled with dirt at various levels.*"
- In LISP from end of '50s!
- In Java from the 1st version

Requirements on GC

- **High Throughput**
- **Low Latency**
- **Minimal Footprint**
- Reliable
- Easy to use/tune

Generational Hypothesis

- Most objects die young
- Old objects rarely reference young objects

Generational Garbage Collector

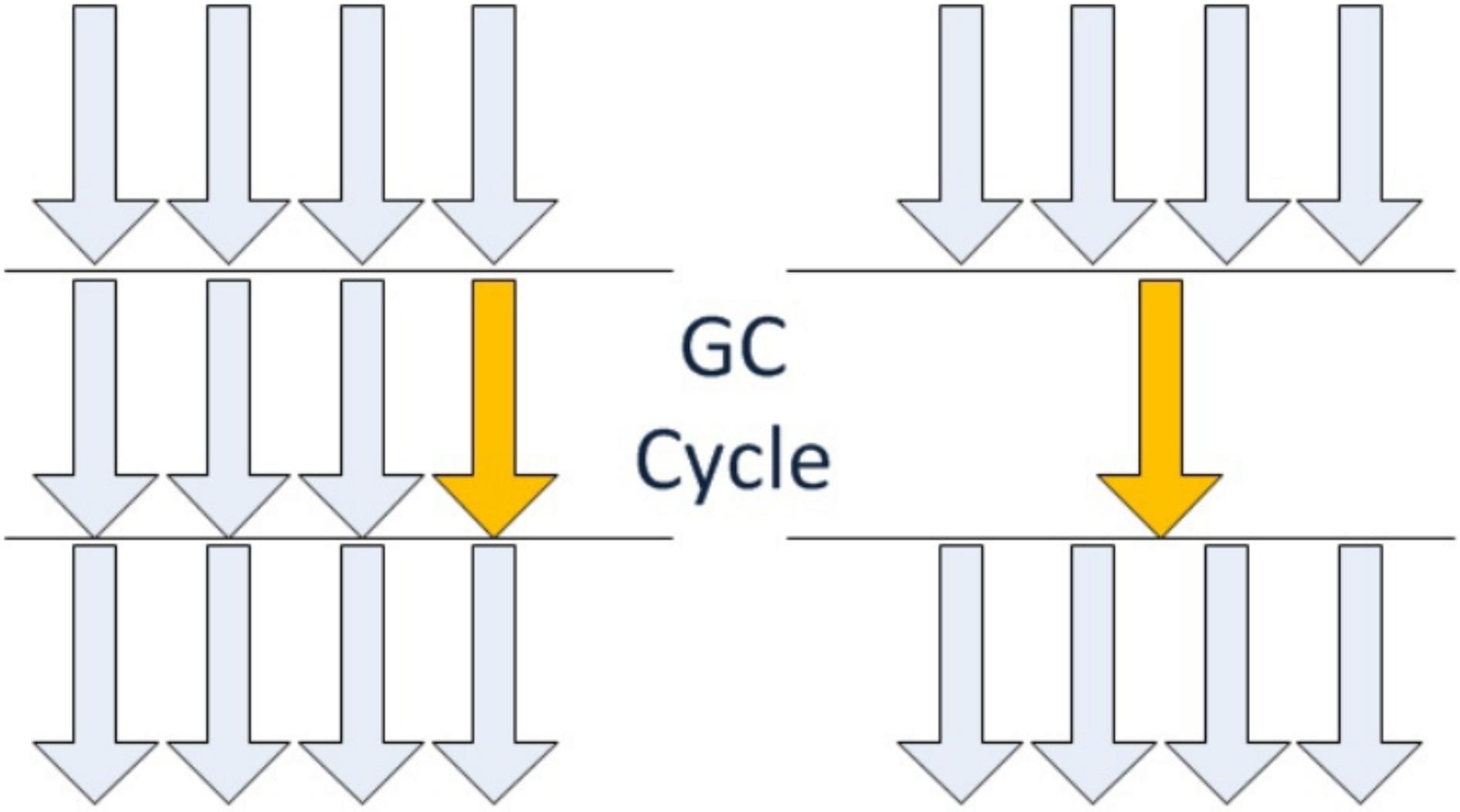
- Split the heap into regions
- Create new objects in Young
- Move mature objects to Old

- Different strategies for different regions

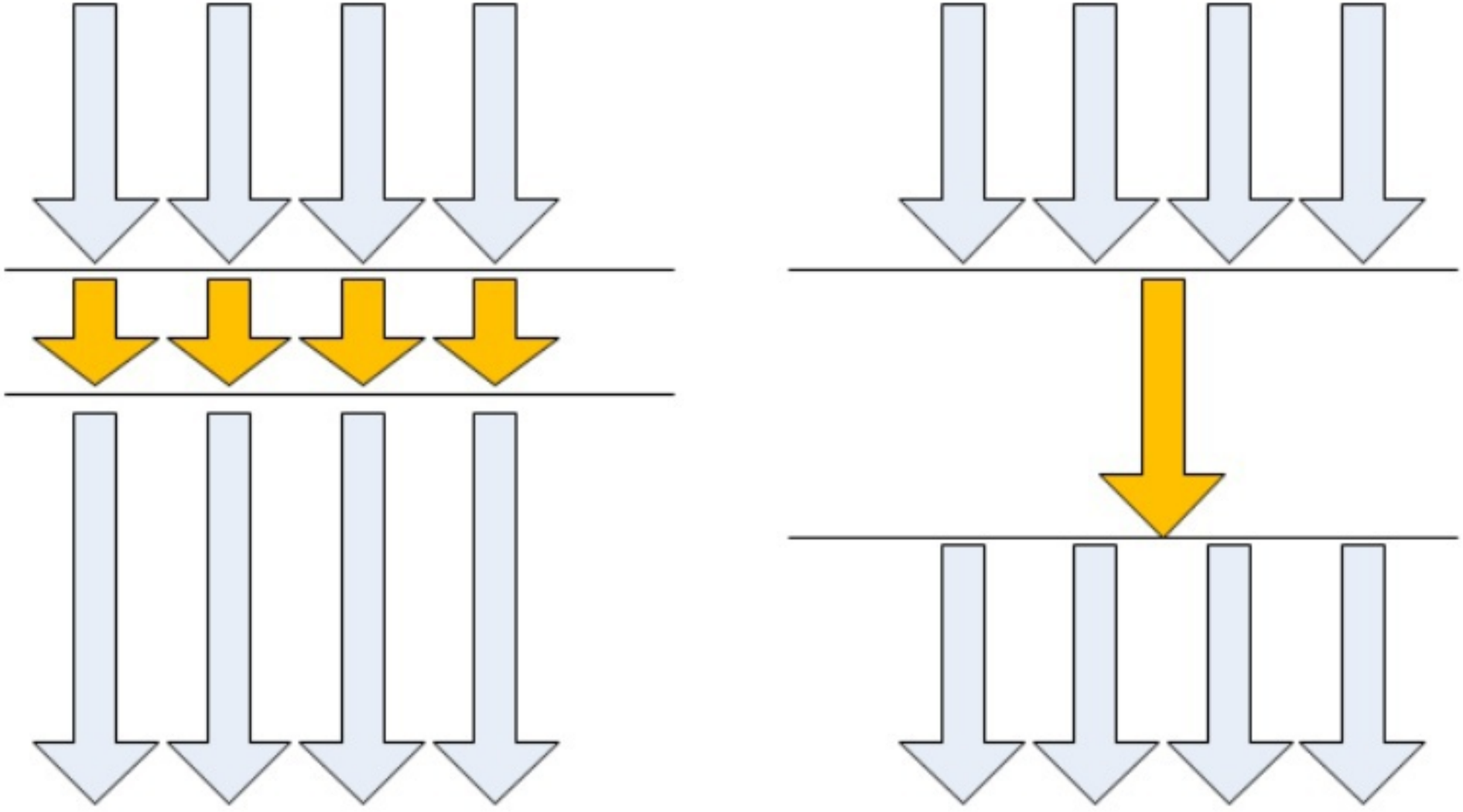
Characteristics of GCs

- Concurrent
- Parallel
- Compacting

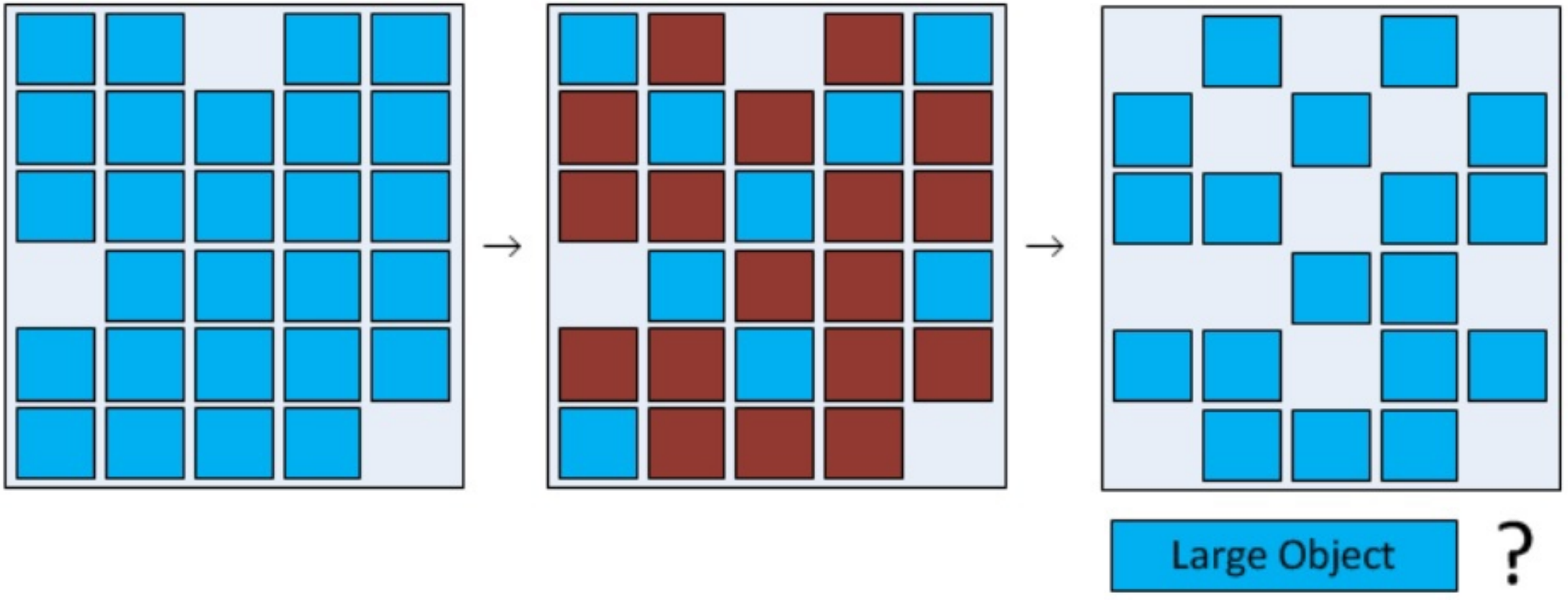
Concurrent GC



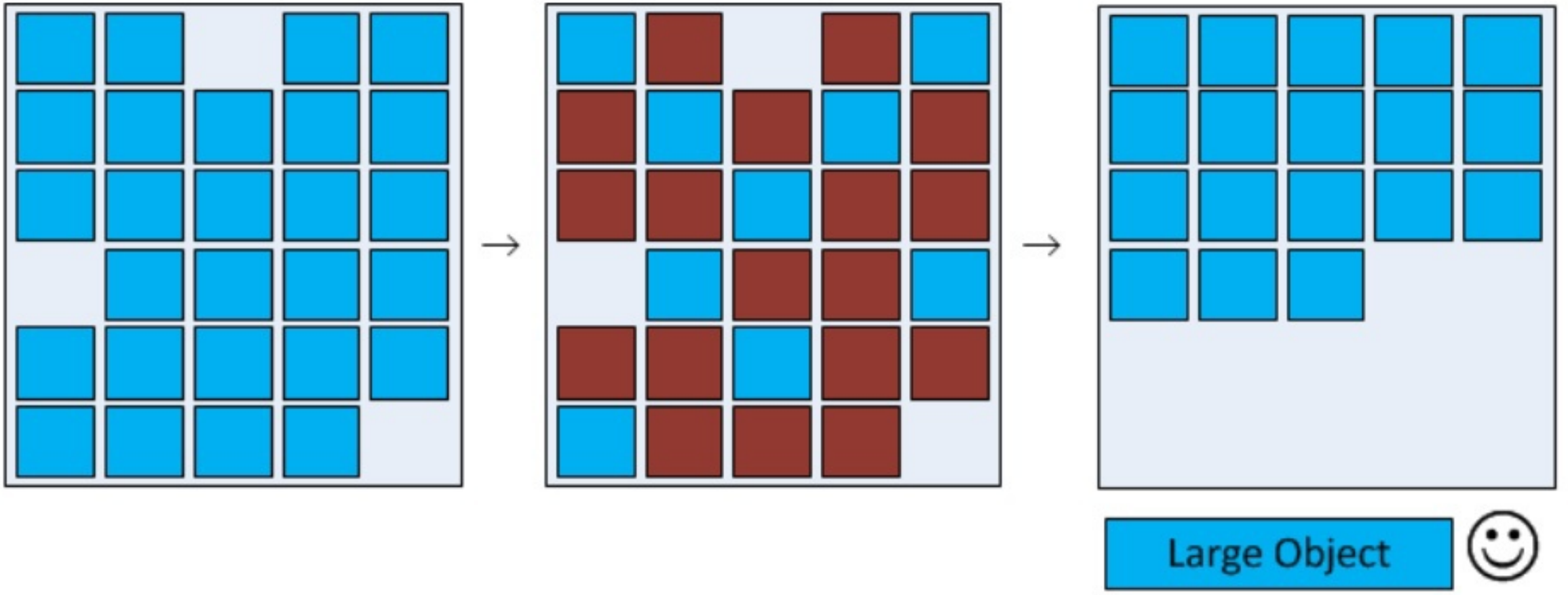
Parallel GC



Heap Fragmentation

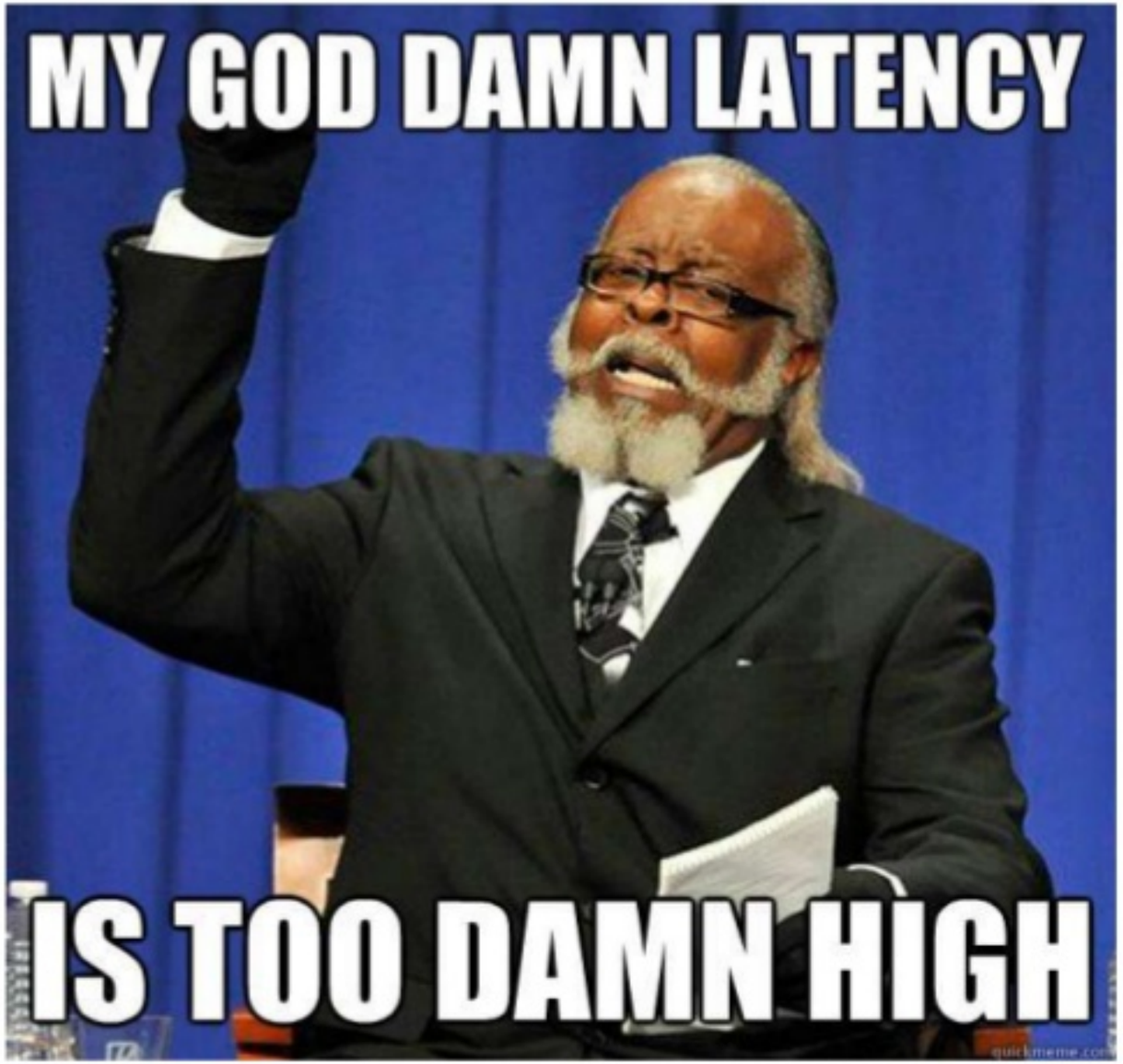


Heap Compacted



What we have in JVM? I

- Serial GC
- Parallel GC



What we have in JVM? II

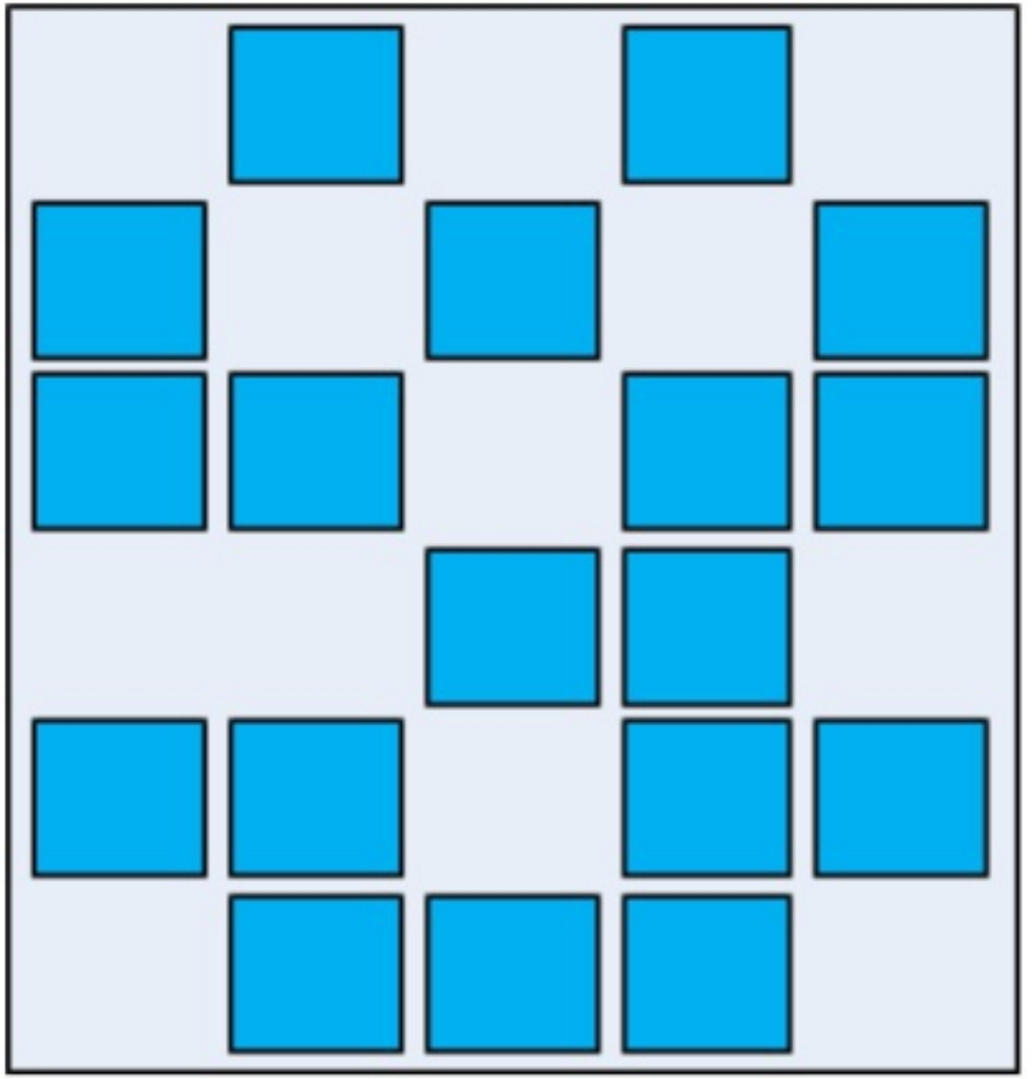
- CMS
- G1
  - since 6u14
  - supported since 7u4

How CMS works in young generation II

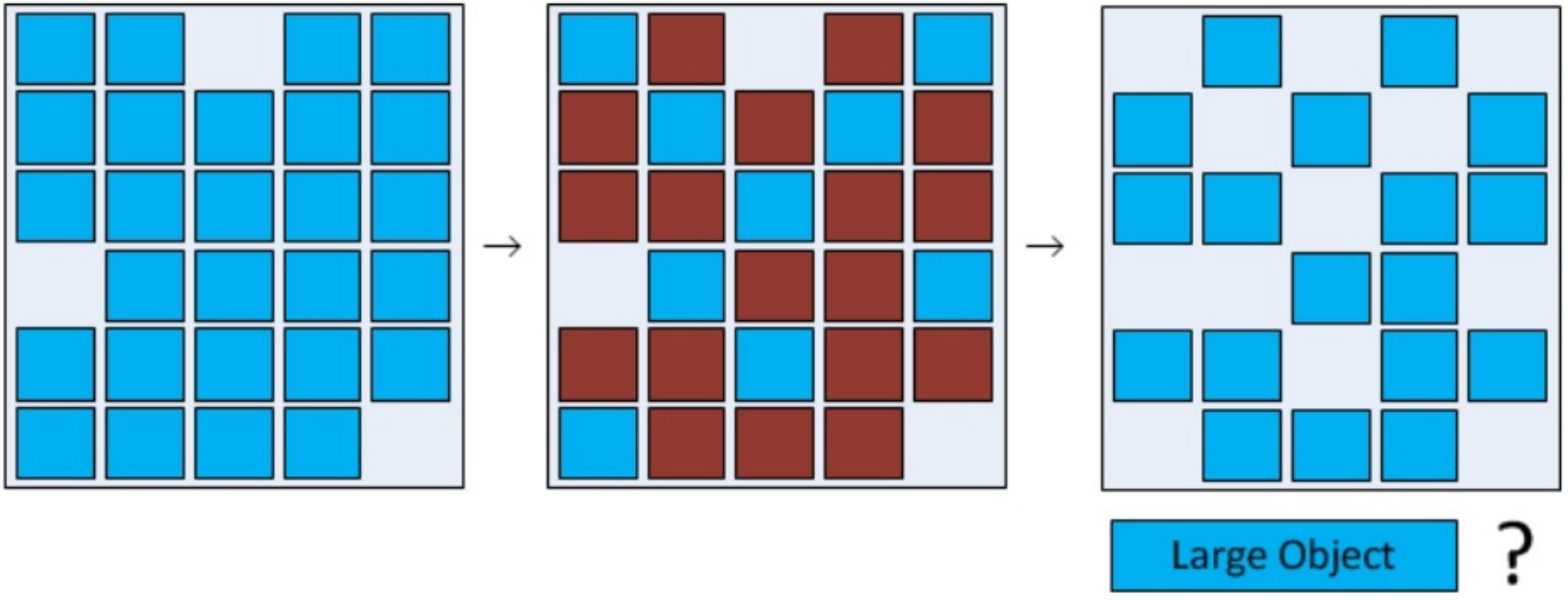
How CMS works in young generation II

How CMS works in old generation

- Main phases of CMS:
  - Initial Mark (STW)
  - Concurrent Mark
  - Remark (STW)
  - Concurrent Sweep
- Features:
  - Continuous block of memory
  - No compacting!



Heap Fragmentation



What's the catch?

- Fragmentation
- Remark phase (STW)
- Very Large Heaps

"It's working very well except when it's not". –*M. Brasier*

Promotion Failure

Concurrent Mode Failure

Stop The World

G1

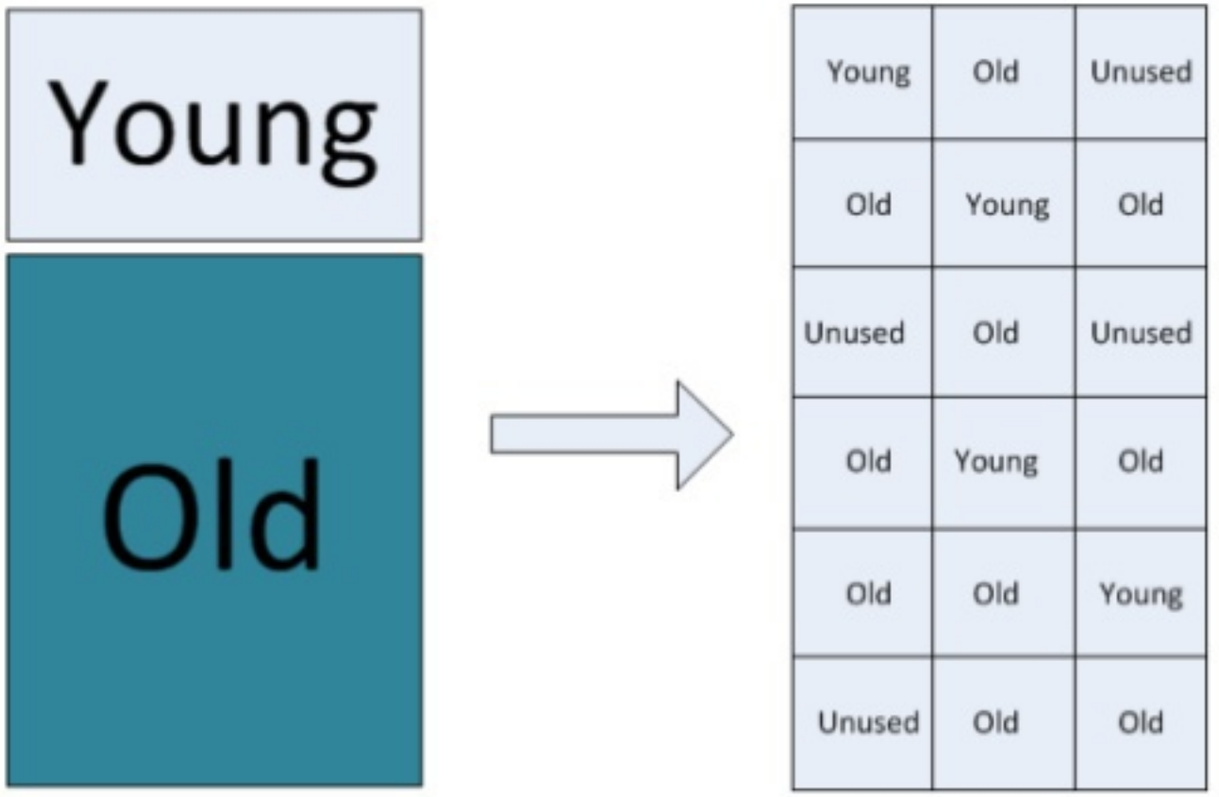
- Concurrent
- Parallel
- Compacting

G1 Goals

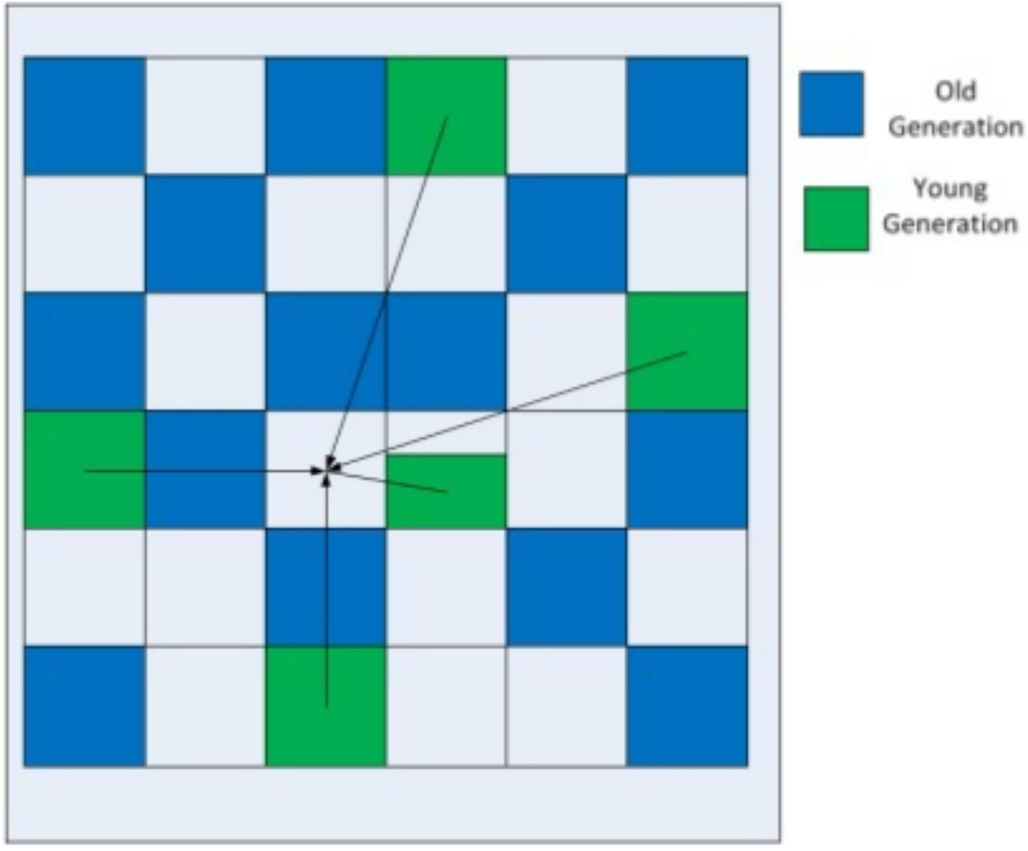
- Low Latency
- Better Predictability
- Easy to use & tune

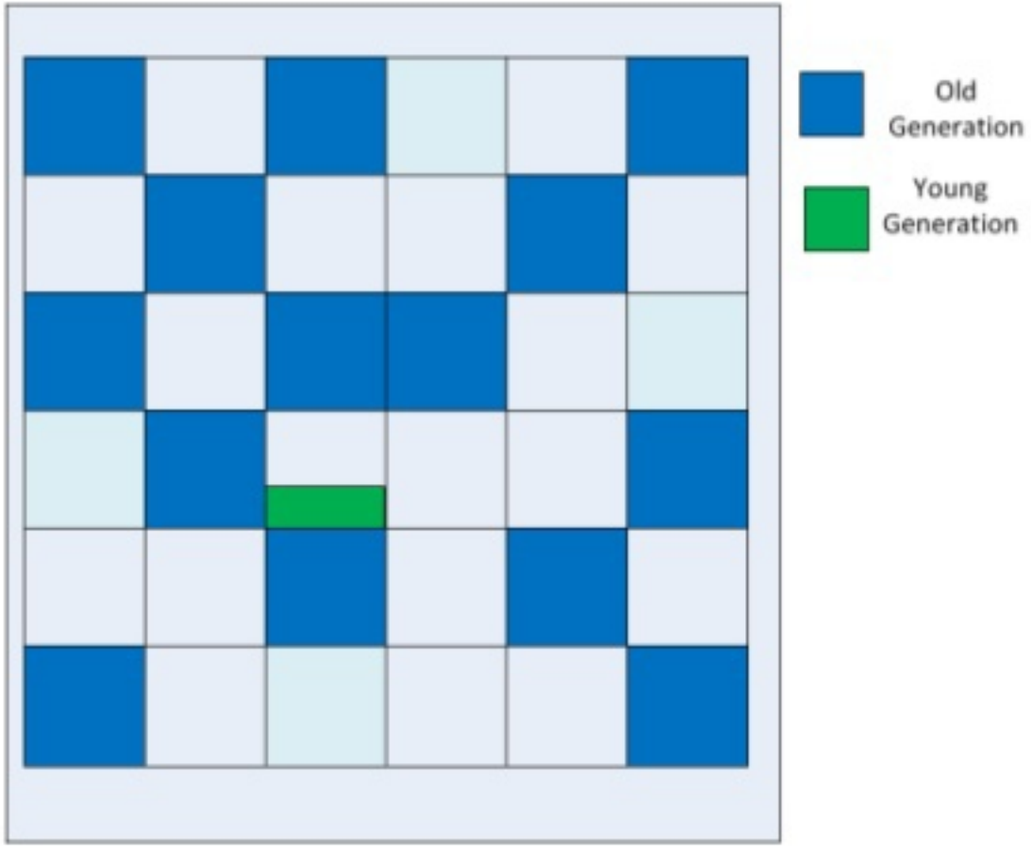
G1 Heap Layout

- Heap divided into +- 2000 regions of equal size
- No physical separation between young / old generation
- Each region can be either young or old
- Object moved between regions during collections
- Humongous Region
  - For large objects
  - Expensive collecting as of now!



G1 Young Generation GC





G1 Old Generation GC I.

- *Combination of CMS and Parallel Compacting GC*
- Concurrent Marking Phase
- Remark still STW

```
1929.299: [GC remark 1929.299: [GC ref-proc, 0.0048180 secs], 0.0258670 secs]
 [Times: user=0.03 sys=0.00, real=0.02 secs]
```

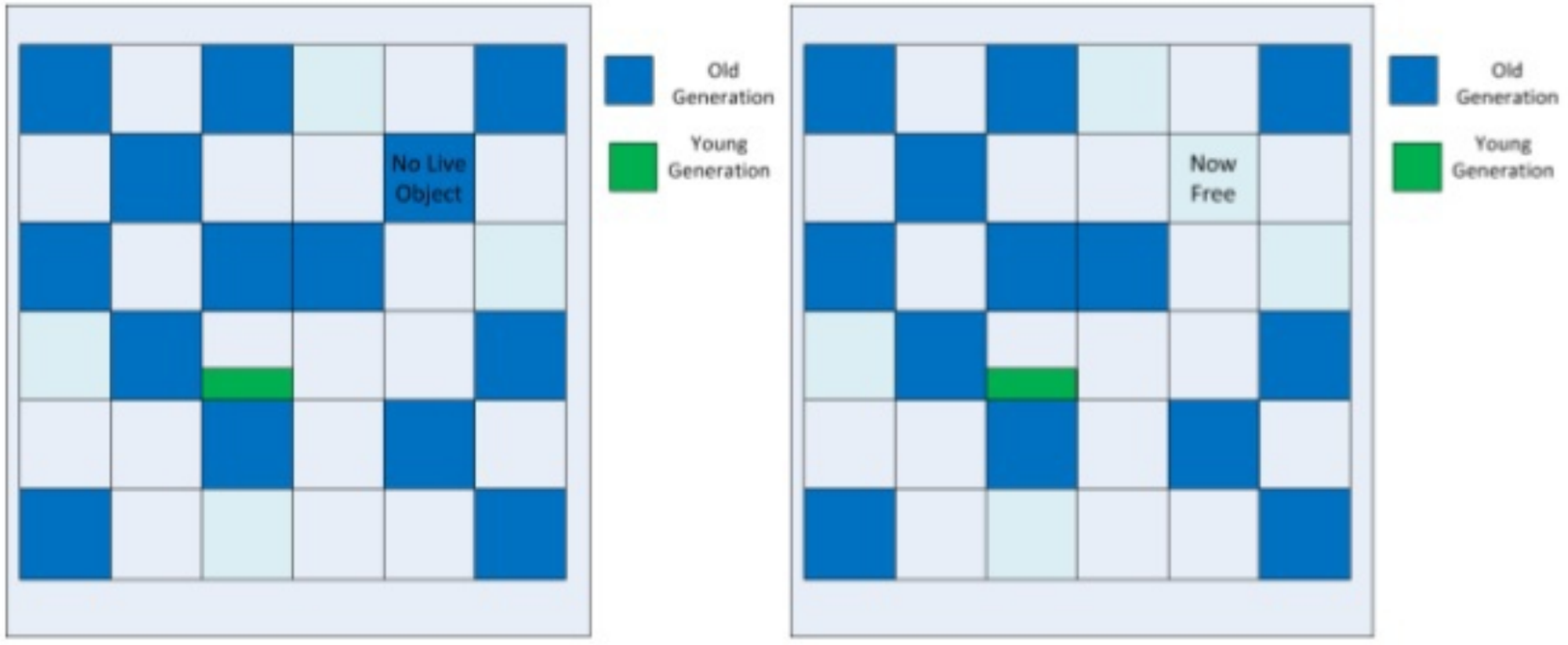
- Initial Marking part of evacuation pause

```
62.583: [GC pause (young) (initial-mark), 0.09812400 secs]
```

- Calculating statistics data
  - Ratio of living objects inside region
  - Who references our region

G1 Old Generation GC II.

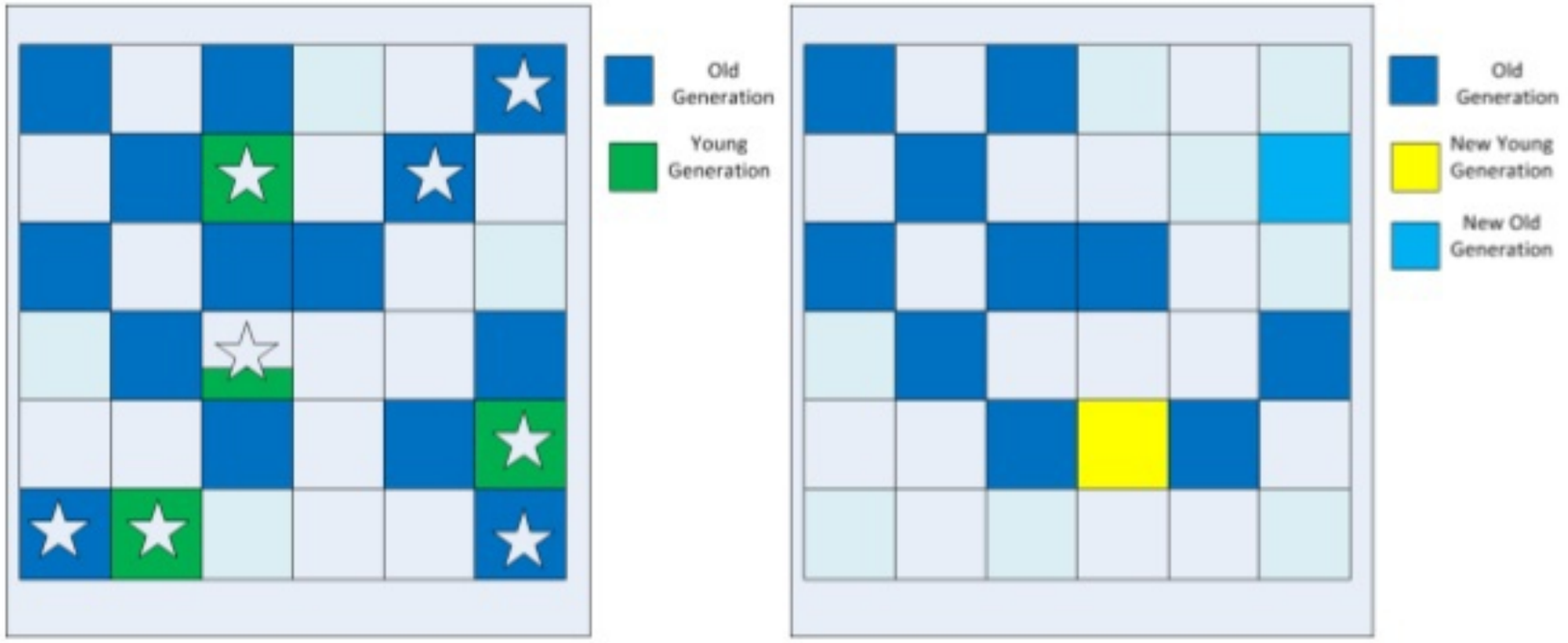
- Regions with no living objects can be reclaimed immediately (after remark)



G1 Old Generation GC III.

- Regions with smallest ratio of living objects chosen to evacuate during next GC

```
53.695: [GC pause
(mixed), 0.05568800 secs]
```



G1 Old Generation GC IV.

- Concurrent Marking Phase
  - Calculate liveness ratio
  - Find *best* regions to evacuate
- Reclaming Regions
  - After Remark
  - During evacuation pause

Remember Set

- Structure to track references from other regions
- One per region
- Allows to collects objects without tracking whole heap to find references
- Maintenance based on write barrier

There is no free lunch!

Gotcha! (Part I.)

- More inter-region references -> Bigger Remember Set
- Large Remember Set -> Slow G1

- Remember: Large Object -> Humongous Region -> Collecting not optimized as yet

G1 Tuning Fundamentals

- Main goal of G1 is a low latency
- If you can tolerate latency use Parallel GC
- Another goal is simplicity (of tuning)

- Most important tuning option:

  `-XX:MaxGCPauseMillis=300`

- Influence maximum amount of work per collection
- No hard promise
- Best effort only

```
-XX:+UseCMSInitiatingOccupancyOnly
-XX:CMSInitiatingOccupancyFraction=60
-XX:CMSTriggerRatio=60
-XX:CMSWaitDuration=1500
-XX:ConcGCThreads=2
-XX:SurvivorRatio=8
-Xmn1g
….
```

Other Import Tuning Options I

`-XX:InitiatingHeapOccupancyPercent=n`

- When to start a concurrent GC cycle
- Percent of an entire heap, not just old genereration!

Too Low -> Unnecessary GC overhead

Too High -> “Space Overflow” -> Full GC

Other Import Tuning Options II

`-XX:G1OldCSetRegionLiveThresholdPercent=n`

- Threshold for region to be included in a Collection Set

- To High -> More aggressive collecting
  - More live objects to copy
- To Low -> Wasting some heap

Other Import Tuning Options III

`-XX:G1MixedGCCountTarget=n`

- How many Mixed GC / Concurrent Cycle

- To High -> Unnecessary overhead
- To Low -> Longer pauses

Considerations Before Tuning

- Some options cause your PauseTimeTarget to be ignored!

  `-Xmn`

- Fixed young generation size

Logging the GC Activity

- Verbose GC is your friend!

```
-XX:+PrintGC -XX:+PrintGCDetails -
XX:+PrintGCTimeStamps
```

- Always use it! Even in production.
- Very low overhead. There is no excuse!

- Will help you analyze help your GC is performing

G1 GC Log

```
1.297: [GC pause (young), 0.02828800 secs]
   [Parallel Time:  24.2 ms]
      [GC Worker Start (ms):  1296.7  1302.7
       Avg: 1299.7, Min: 1296.7, Max: 1302.7, Diff:   6.0]
      [Ext Root Scanning (ms):  16.5  2.9
       Avg:   9.7, Min:   2.9, Max:  16.5, Diff:  13.6]
      [Update RS (ms):  0.0  0.0
       Avg:   0.0, Min:   0.0, Max:   0.0, Diff:   0.0]
         [Processed Buffers : 0 9
          Sum: 9, Avg: 4, Min: 0, Max: 9, Diff: 9]
[…]
```

G1 GC Log

```
1.297: [GC pause (young), 0.02828800 secs]
```

- Young GC
- Started 1.297s from the start of JVM
- Total elapsed time was 0.02828800s

```
[Parallel Time:  24.2 ms]
```

- Total elapsed time spent by parallel GC thread

G1 GC Log

```
[GC Worker Start (ms):  1296.7  1302.7
       Avg: 1299.7, Min: 1296.7, Max: 1302.7, Diff:   6.0]
```

- Two threads and starting times

```
 [Ext Root Scanning (ms):  16.5  2.9
       Avg:   9.7, Min:   2.9, Max:  16.5, Diff:  13.6]
```

- Time spent by each thread by scanning the roots

G1 GC Log

```
[Update RS (ms):  0.0  0.0
       Avg:   0.0, Min:   0.0, Max:   0.0, Diff:   0.0]
         [Processed Buffers : 0 9
          Sum: 9, Avg: 4, Min: 0, Max: 9, Diff: 9]
```

- Time spent by updating Remember Sets regions
- Update Buffers is a structure tracking changes in references

```
[Scan RS (ms):  0.0  0.1
       Avg:   0.0, Min:   0.0, Max:   0.1, Diff:   0.1].
```

- Time spent by scanning areas from Remember Set

G1 GC Log

```
[Object Copy (ms):  7.6  15.1
       Avg:  11.3, Min:   7.6, Max:  15.1, Diff:   7.5]
```

- Time spent by copying objects to other regions
- Usually significant portion of a total time

```
[Termination (ms):  0.0  0.0
       Avg:   0.0, Min:   0.0, Max:   0.0, Diff:   0.0]
         [Termination Attempts : 1 1
          Sum: 2, Avg: 1, Min: 1, Max: 1, Diff: 0]
```

- Time spent by offering to terminate
- Threads are stealing work from the others

G1 Logging Options I

- `-XX:+PrintAdaptiveSizePolicy`

```
53.628: [GC pause (mixed) 53.628: [G1Ergonomics (CSet Construction) start choosing CSet, predicted base time: 54.79 ms, remaining time: 145.21 ms, target pause time: 200.00 ms]

 53.628: [G1Ergonomics (CSet Construction) add young regions to CSet, eden: 51 regions, survivors: 2 regions, predicted young region time: 9.75 ms]

 53.628: [G1Ergonomics (CSet Construction) finish adding old regions to CSet, reason: old CSet region num reached max, old: 27 regions, max: 27 regions]

 53.628: [G1Ergonomics (CSet Construction) finish choosing CSet, eden: 51 regions, survivors: 2 regions, old: 27 regions, predicted pause time: 131.56 ms, target pause time: 200.00 ms]

 53.700: [G1Ergonomics (Concurrent Cycles) do not request concurrent cycle initiation, reason: still doing mixed collections, occupancy: 127926272 bytes, allocation request: 0 bytes, threshold: 125986365 bytes (45.00 %), source: end of GC]
```

G1 Evacuation Failure

- Evacuation Failure Happens when we do not have empty regions
- Heap is already at maximum size
- Very expensive operation!
- Search for "`to-space overflow`"

- Remedy:
  - Increase to heap size
  - `-XX:InitiatingHeapOccupancyPercent=`
  - More marking threads

Is G1 Production Ready?

- It depends!
  - Well, some people use it in production
- The days of G1 crashing JVM are hopefully over
- Still quite significant changes
- Bugs are still being found
  - *Bug 7181612 : G1: Premature Evacuation (7143858) strikes again*
  - *Bug 7143858 : G1: Back to back young GCs with the second GC having a minimally sized eden*

- You should probably evaluate/use G1 when everything else is failing
- And you have no alternative!

GC Decisioning: Rules of Thumb I

- Do I really need a Low Latency
  - No? Use Parallel GC
- Do I really need a Big Heap?
  - Yes? Think again!

- Do I really need a Big Heap?
  - No? Use small heap & Parallel GC
  - Yes? Try CMS 1st

GC Decisioning: Rules of Thumb II

- Is CMS performing well?
  - Yes? Done!
  - No? Tune it!
- Is tuned CMS performing well?
  - Yes? Done!
  - No? Tune it more!
- Is CMS performing well now?
  - Yes? Done!
  - No? Do you **really** need such big heap?

GC Decisioning: Rules of Thumb III

- Yes, I really need a Big Heap a Low Pauses!
  - This is the moment where you should start considering to use G1!

- In Nutshell:
  - For now treat G1 as a last resort
  - When other GCs are failing
  - Be aware of gotchas:
    - Very Large Objects
    - Large Remember Sets
    - Footprint?
  - Test carefully before using in a production!

G1 – Recent Commits

date: Tue Oct 30 11:45:51 2012 -0700
summary: 7200261: G1: Liveness counting inconsistencies during marking verification

date: Thu Sep 20 09:52:56 2012 -0700
summary: 7190666: G1: assert(_unused == 0) failed: Inconsistency in PLAB stats

date: Tue Aug 21 14:10:39 2012 -0700
summary: 7185699: G1: Prediction model discrepancies

G1 – Recent Commits – Conclusion?

- Do not use anything before 7u4
  - Use 7u9 or later
  - The latest version in the best case

- Do not use G1 from Java 6

What Are The Alternatives?

- Workaround / Hacks:
- Off Heap Allocations
  - DirectBuffers
  - JNI
- Object Pooling

Further Resources

- JavaOne 2012 G1 Talk, Charlie Hunt, Monica Beckwith
- hotstop-gc-use mailinglist
- http://www.quora.com/What-is-the-status-of-the-implementation-of-the-G1-garbage-collector-for-the-JVM#
- ....

Q&A

DEVOXX™
the java™ community conference