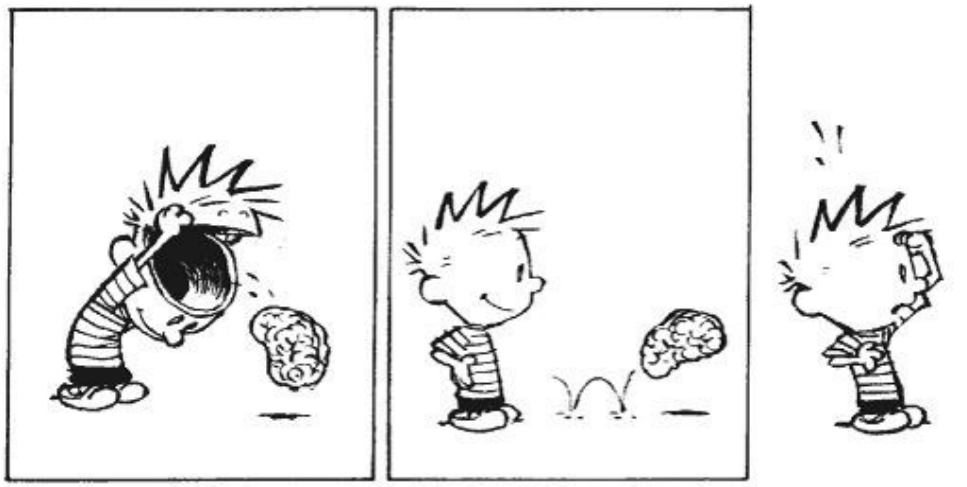
Java Garbage Collectors

Rémi Forax – Jan 2026



Don’t believe what I’m saying !

Vocabulary

Parallel

Uses several threads to do the GC work

Concurrent

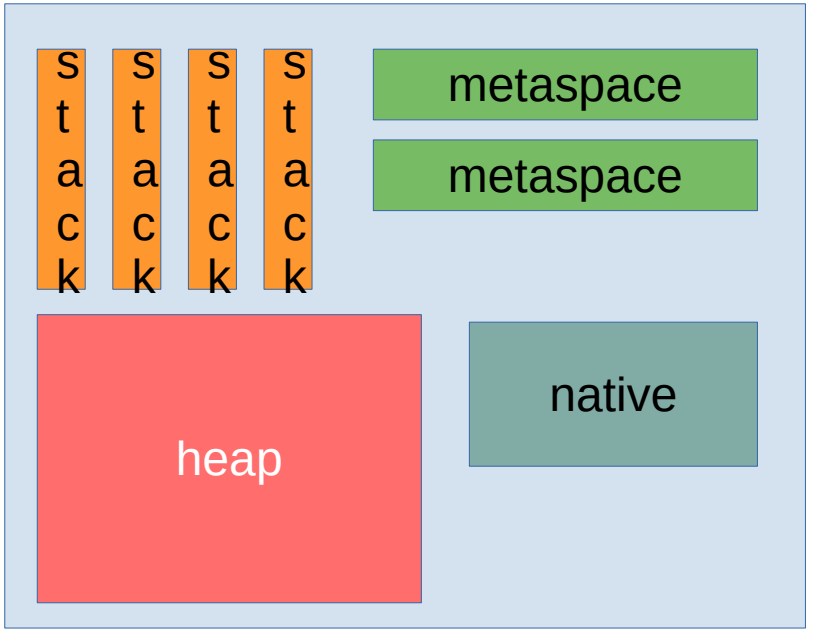
GC work is done at the same time as the application run

Incremental

Current GC cycle reclaim only parts of the heap,
but keep information for later cycles

Kind of Memory inside the JVM

- Each thread has its own stack
- GC Heap (-Xmx, -Xms)
- Native C memory (system calls)
- Metaspace
  - metadata per ClassLoader
    - Klass, constant pool, bytecode, oopMap, profiles for IC, …
  - code cache



Java GCs

5 Garbage Collectors :

- Serial (Sun then Oracle)
- Parallel (Sun then Oracle)
- G1 (Sun then Oracle)
- Shenandoah (RedHat then Amazon)
- ZGC (Oracle)

GC Performance

Throughput

The % of the time spend in the application instead of in the GC

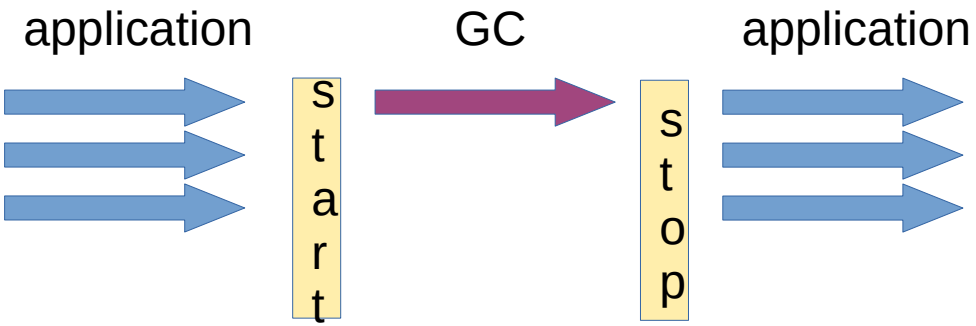
Latency

Duration of the GC pause + GC work done by the application

Footprint

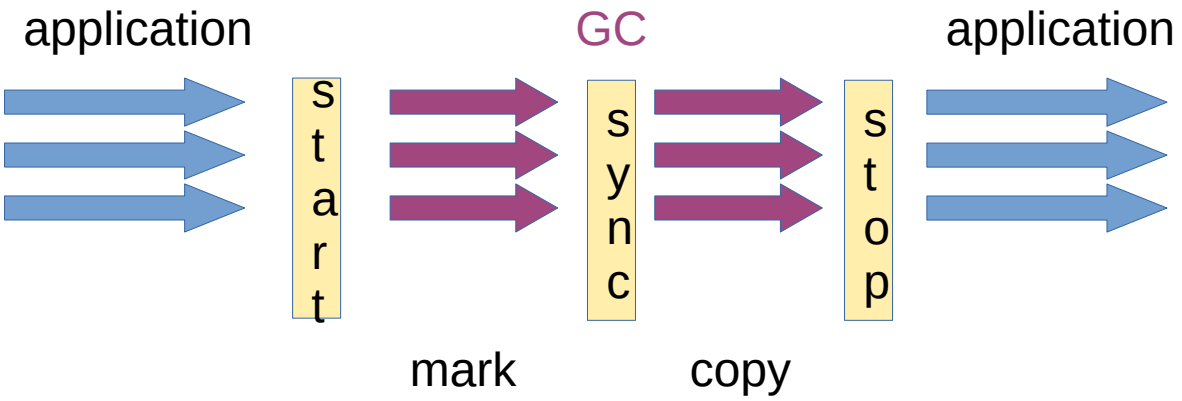
Amount of native memory used by the GC itself

Serial GC



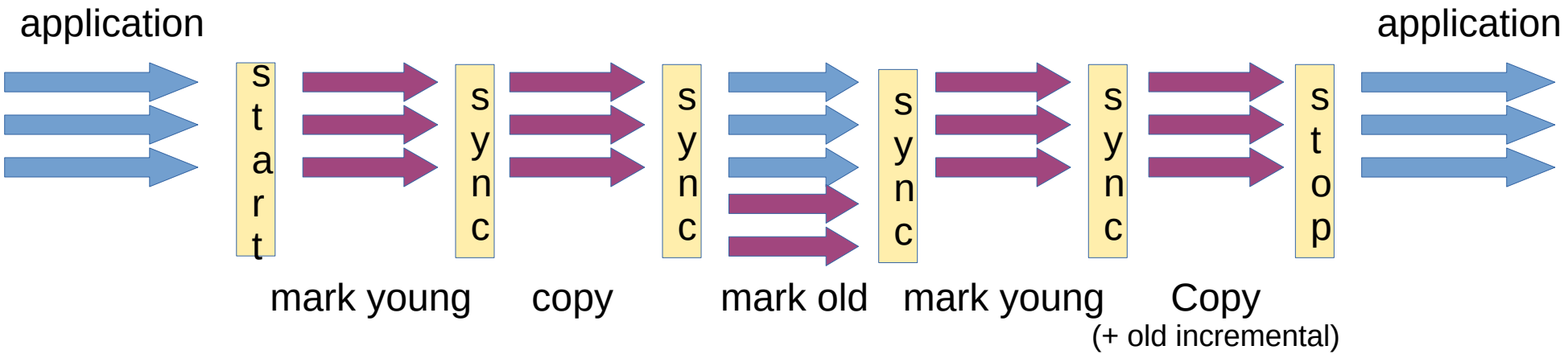
GC for CLIs and small cloud containers

Parallel GC



GC for batch programs (good throughput)

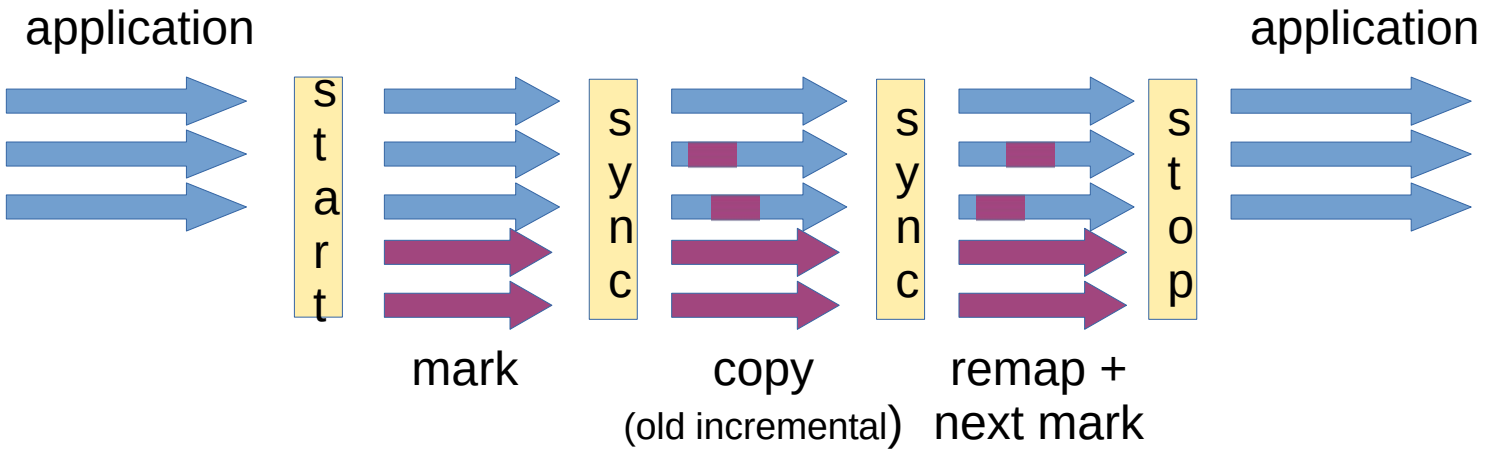
G1 GC (default)



latency goal <= 200 ms (-XX:MaxGCPauseMillis)

Garbage-first garbage collection – ISMM 2004
Printezis, Flood, and al

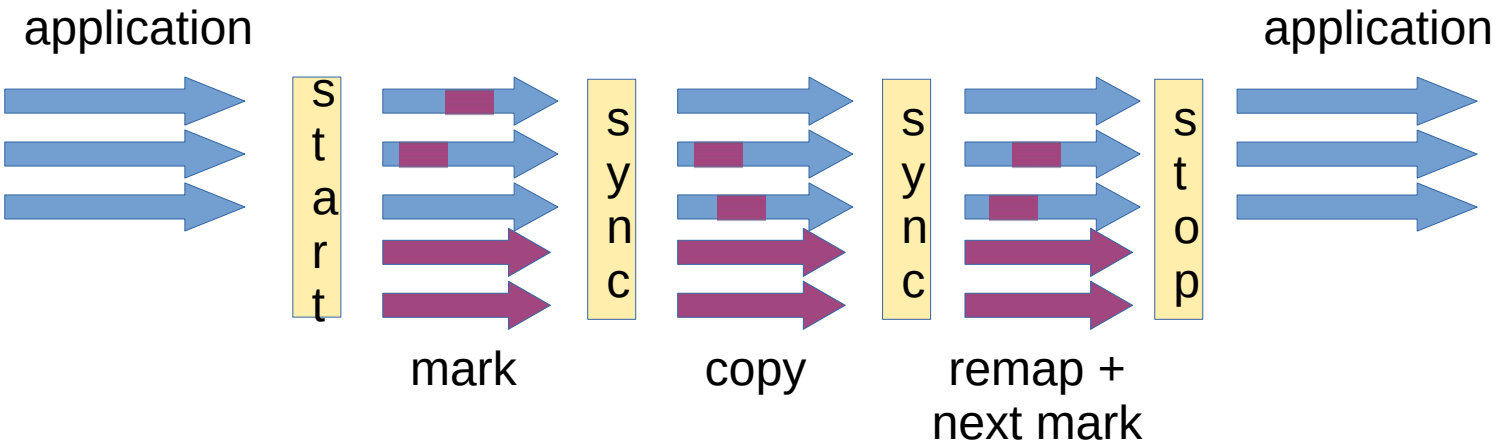
Shenandoah v2



latency goal <= 1 ms)

Shenandoah - Clark et al. – 2021, Flood al. – 2016

ZGC v2



latency goal <= 1 ms

The Z Garbage Collector - Österlund – 2026 - ISBN 9781003595366

Garbage Collectors

Kind of GCs

Determine liveness

- Reference counting (find dead objects, problem with cycles)
- Tracing (find live objects)
  - Precise GC ? (stack is parseable)
    - vs conservative GC : BoehmGC (interop with C)

Moving GCs ?

- Non moving (interop with C), sweep → free
- Moving GC (pointers are opaque)

Allocation (Moving GC)

Thread Local Allocation Buffer (TLAB)

- Bump pointer
- Concurrency (CAS) only at the end of the pages

In Java, allocators are NUMA aware

Object Header from GCs POV

Each Java object has a header

- Tag bits (2 bits)
- Age bits (4 bits)

```
Mark Word (normal):
 64                      39                                      8    3  0
 [.....................HHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHH.AAAA.TT]

Class Word (compressed):
 32                                  0
 [CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC]



Header (compact):
 64                     42                                  11   7   3  0
 [CCCCCCCCCCCCCCCCCCCCCCHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHVVVVAAAASTT]
 (Compressed Class Pointer)       (Hash Code)          /(GC Age)^(Tag)
```

Mark Phase

Tracing

From the roots, mark recursively all objects by following pointers

- Roots in Java
  - references on stack
  - static fields
  - + JNI roots, VM roots

Real implementations are not really recursive

=> stack overflow !

Precise GC => Safepoints

The stack frame layout has to be known for marking

- GC operations can only run at safepoints

In Java

- Abstract interpretation of the bytecode
  - Stored in oopMap (metadata)
- Safepoints
  - At allocation site (**new** ...)
  - At the end of a method
  - At the end of a body of a non counted loops

Tri-colors Marking

Breadth First Search:

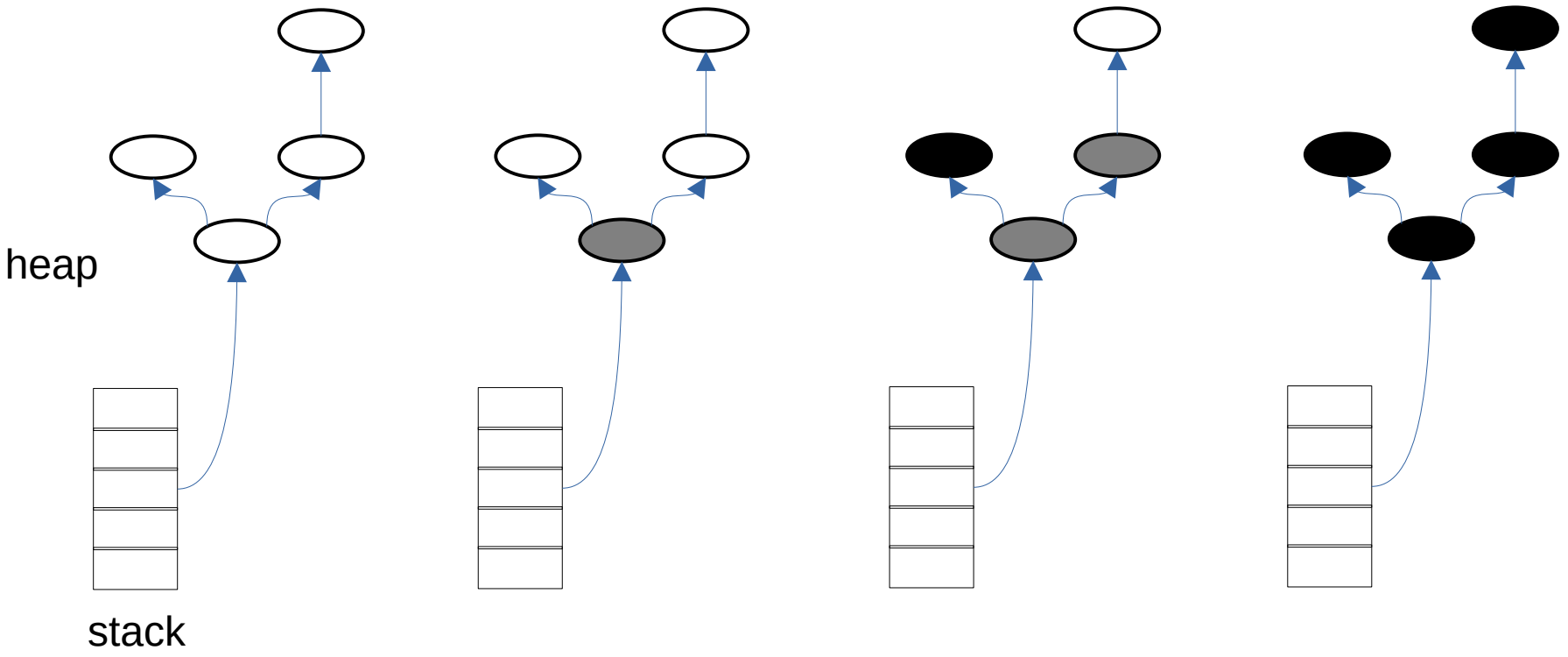
- White : not yet marked
- Grey : working set
- Black : marked and children are marked

    At the end, black and white are swapped

In Java

- grey set : 1 queue per GC threads + Work stealing
- white/black are bits in the object header

Tri-colors Marking



Concurrent marking

Mark when application run (not STW)

- Snapshot At The Beginning (SATB)

During marking phase

- New objects are marked black
- If an object field mutated by the application, the object is added to grey set (if ref not null)

Generational GCs

Generational Hypothesis

young objects are more likely to die than old objects

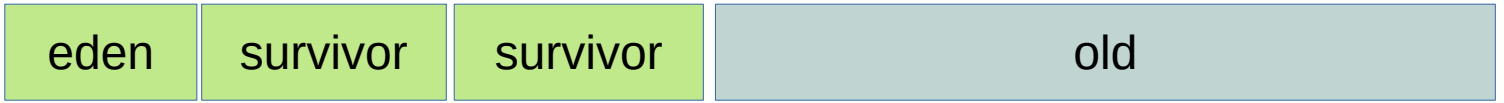
"most objects die young"

A Real-Time Garbage Collector Based on the Lifetimes of Objects
Henry Lieberman and Carl Hewitt – MIT - 1981

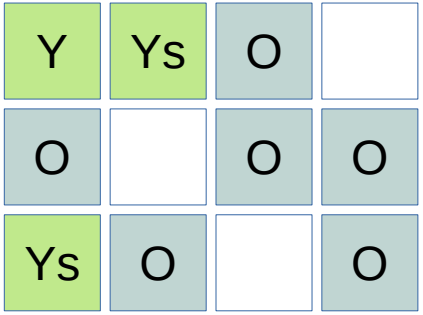
Generation Scavenging: A non-disruptive high performance storage reclamation algorithm
David Ungar – ACM - 1984

Regions

Contiguous (Serial, Parallel)



Partitions (G1, Shenandoah, ZGC)



Y = young
O = old

Regions have different sizes
2M < size < as big as an array

Generational GCs

Different kind of collection

- Minor
  - mark young (+ RememberSets) and collect young
- Major (10 to 100 x less)
  - mark all and collect old
- Full (`System.gc()` or allocation failure)
  - mark all and collect all

Remember Set

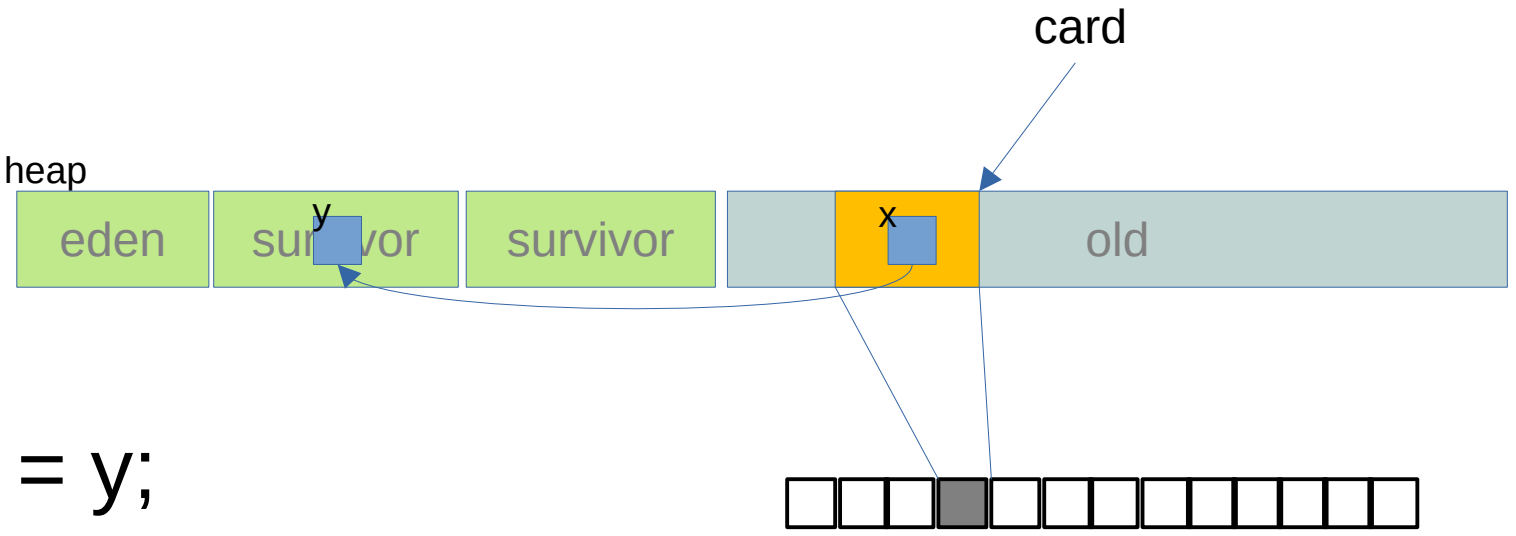
Data structure that register a card (zone) in a region that contains a reference to another region

- G1: 1 card 512 bytes
- ZGC: 1 card == 1 field

RememberSet Old regions -> Young regions

- A card is a root for minor collections

Remember Set using a card table



x.a = y;

each bit is 1 card of the old region

GC write barrier

Snippet of code added after each field write

- Update the corresponding RememberSet (if necessary)

- Write Barrier of G1

```
if (region(x.a) == region(y)) goto done;  // Ignore refs same region
if (y == nullptr) goto done;              // Ignore null
if (card(x.a) != CLEAN) goto done;        // Ignore non-clean card
*card(x.a) = DIRTY;                       // Mark the card
done:
```

G1GC: Collection of old regions

Old generation is usually big

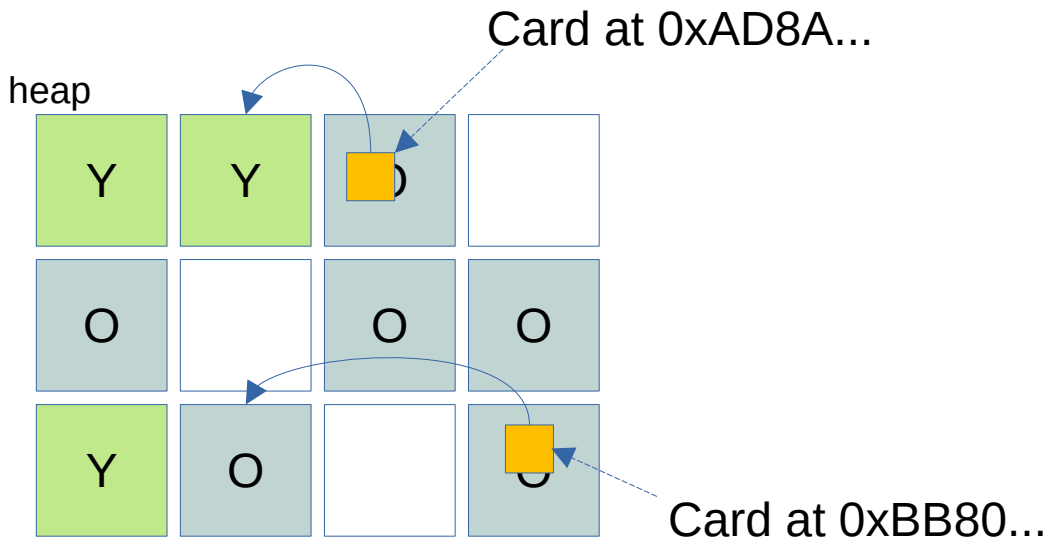
- Slow if collected in one go => collect incrementally

- Marking old gen
  - starts with marking young
    + young to old pointers are additional roots
  - concurrent marking
    + compute the RememberSet in between each old regions

RememberSet

Young regions share a RememberSet

- When doing a major collection
  - Each old region has its own RememberSet



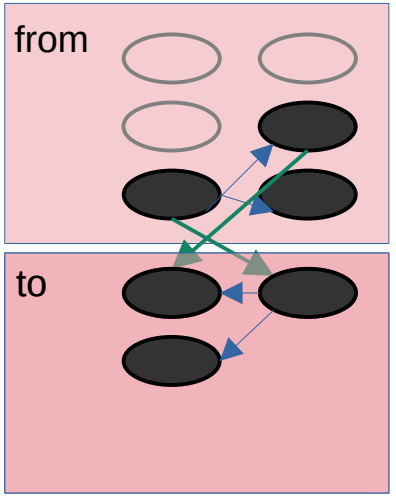
Young RememberSet
0xAD8A...

Zone 9 RememberSet
0xBB80...

Evacuation Phase

Copying

Semispace Cheney 1970

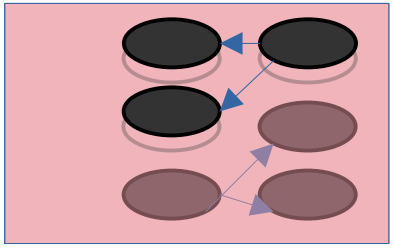


All objects are allocated in the from-space

If STW

- Copy all the marked objects to the to-space
  - Object headers in the from-space point to the new address
  - Fields are updated during the copy
- Update all the root pointers
  - roots (stacks + static)
  - cards
- Then, swap the two spaces

Compacting
LISP-2 1970



Forward:
4 → 1, 5 → 2, 6 → 3

Reuse the same space

=> Move all objects at the beginning of the space

If STW

- Create a relocation forward buffer (hash, sorted list) with old address → new address for all the marked objects
- Update all the root pointers
  - roots (stacks + static) + cards
- Move all the objects at the beginning of the space, update fields
- The allocation pointer is set after the last object

G1 (garbage first) GC

Incremental evacuation

- Sort young regions by % garbage
  - Sort old regions by % garbage and freed space
- Evacuate incrementally the first regions in latency budget
  - Non evacuated zones will be in subsequent GC cycles

GC must to be triggered before allocation pool exhausted

Shenandoah GC

G1GC algo + concurrent evacuation/remap

Load Barrier

- application threads can also do object evacuation

GenShen (generational):

- Major marking = minor marking but ref to old gen are kept as root for old gen marking
- Old gen regions are evacuated incrementally by young collections

Load Barrier

Snippet of code inserted in front of each field read

- Copy object if necessary, fix source field pointer

- Load Barrier of Shenandoah

```
if ((thread.gc_state & HAS_FORWARDED) != 0)
  if ($in_cset_fast_test[region(x.a)])    // region in collection set ?
    slow_path();                          // copy? and fix x.a
```

Floating garbage ?

Even if a region if fully evacuated, it can not be re-used

- We have to wait until all pointers have been fixed

  => floating garbage

The Heap has to be dimensioned accordingly

ZGC

Fully concurrent (mark/fix refs + evacuation)

- Application also does mark/fix refs + evacuation

Use pointer coloring + Load/Write Barrier

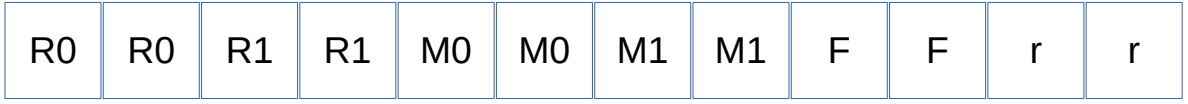
- Mark bits (if invalid => add to the grey set)
- Remap bits (if invalid => object must be copied)
- Free region even with dangling pointers

Colors of pointers

Store GC states in low bits of pointers of the heap

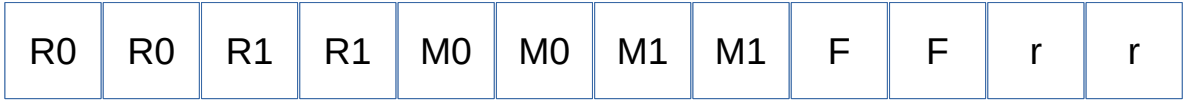
12 color bits:

- 4 bits **R**emap: 1000, 0100, 0010, 0001 (where it goes?)
- 4 bits **M**ark: Mark0, Mark1 * 2 (old or young)
- 2 bits **F**inalizer: 10 or 01 (reachable only using finalizer)
- 2 bits RememberSet (**r**): 01, 10, 11

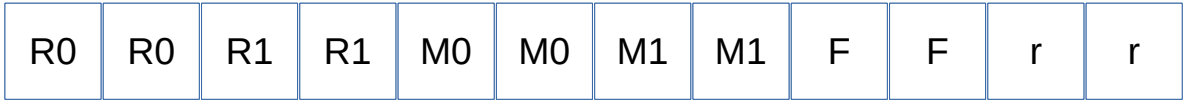


Good colors

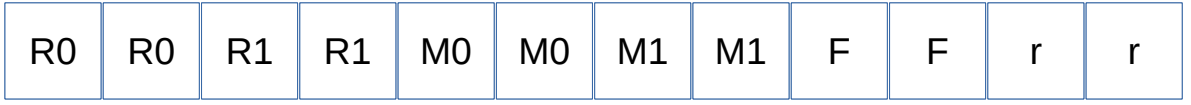
Load Good



Mark Good

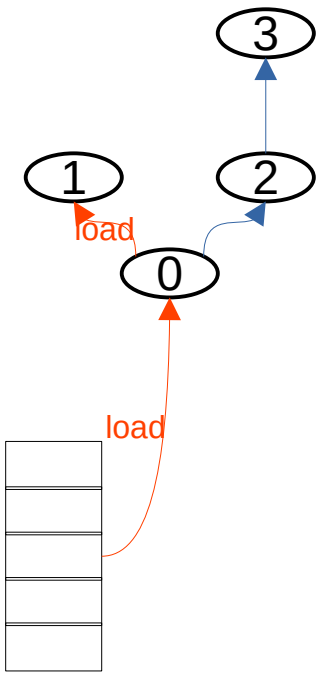


Store Good

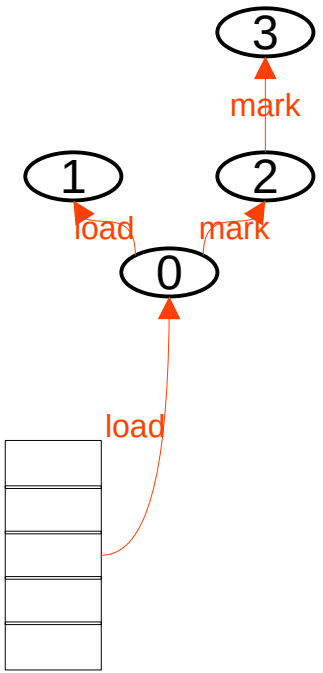


Color during Marking/Remapping

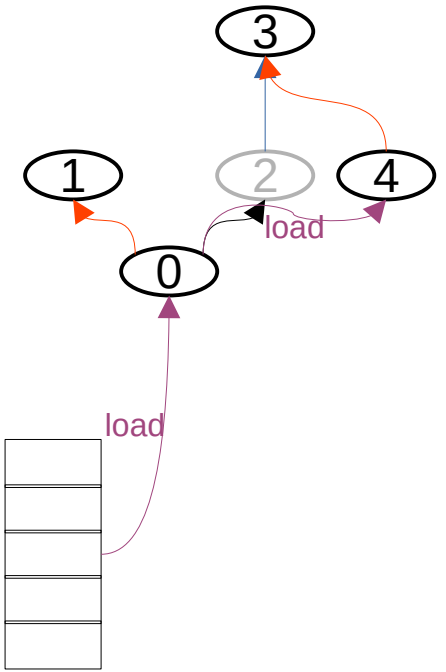
Good : Mark0



Good : Mark0

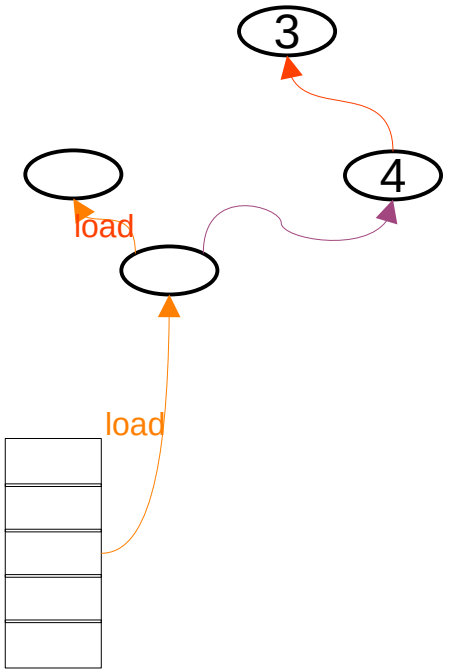


Good : Remapped



Forward: 2 → 4, 3 → 5

Good : Mark1



Forward: 2 → 4, 3 → 5

ZGC Load/Write Barrier intel x64

Load Barrier

```
movq 0x10(%rax), %rcx
shrq $good_remap_bit_index, %rcx   // load good or null + uncolor
ja slow_path
```

Store Barrier

```
testq $store_bad_mask, 0x10(%rax)  // store good or null
jnz slow_path
shlq $good_remap_bit_index, %rcx
orq $store_good_color, %rcx              // recolor
movq %rcx, 0x10(%rax)
```

Summary

Java GCs

5 generational GCs :

- Serial STW, 1 GC thread, WB
- Parallel STW, n GC threads, WB
- G1 regional, concurrent mark, STW incremental evacuation, n GC threads, latency < 200ms, WB
- Shenandoah regional, concurrent, latency < 1ms, n GC threads, WB + LB (not generational by default)
- ZGC regional, concurrent, latency < 1ms, n GC threads, WB + LB (restricted to 64bits pointers)