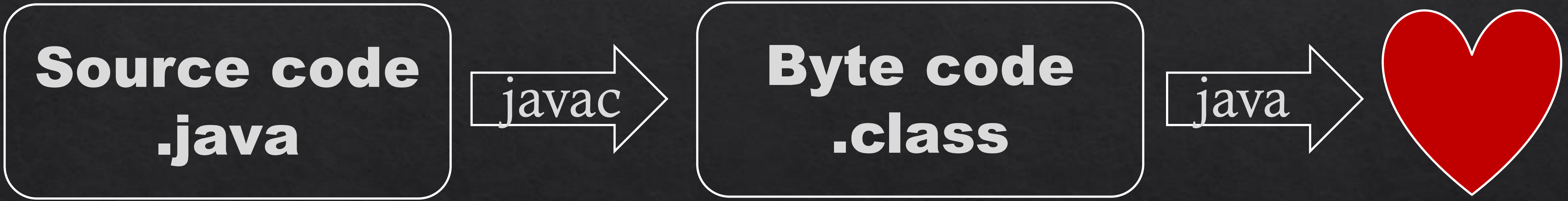
JVM

- Monitoring
- Garbage Collection
- Potential Issues

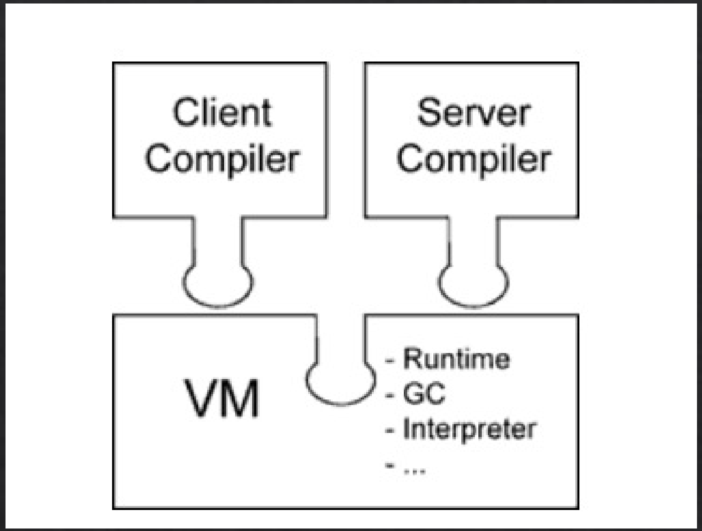
Maxym Borachok

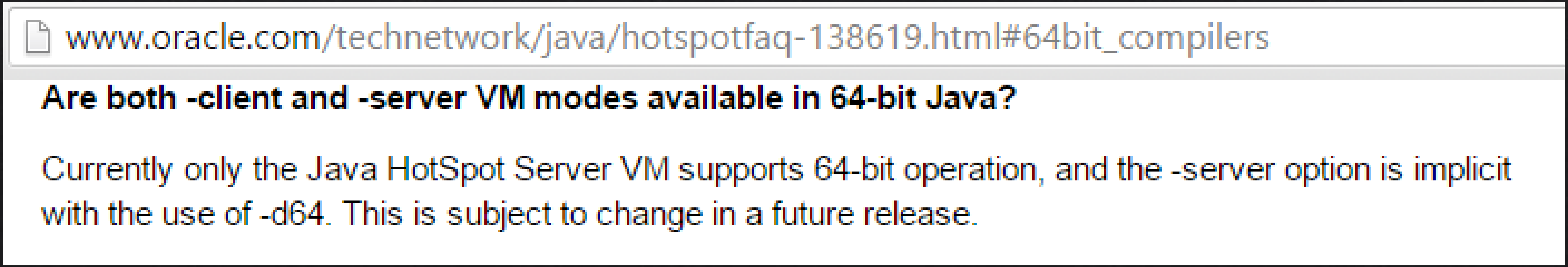
Compile & Execute



```
javac Main.java
java Main
```

-client vs -server



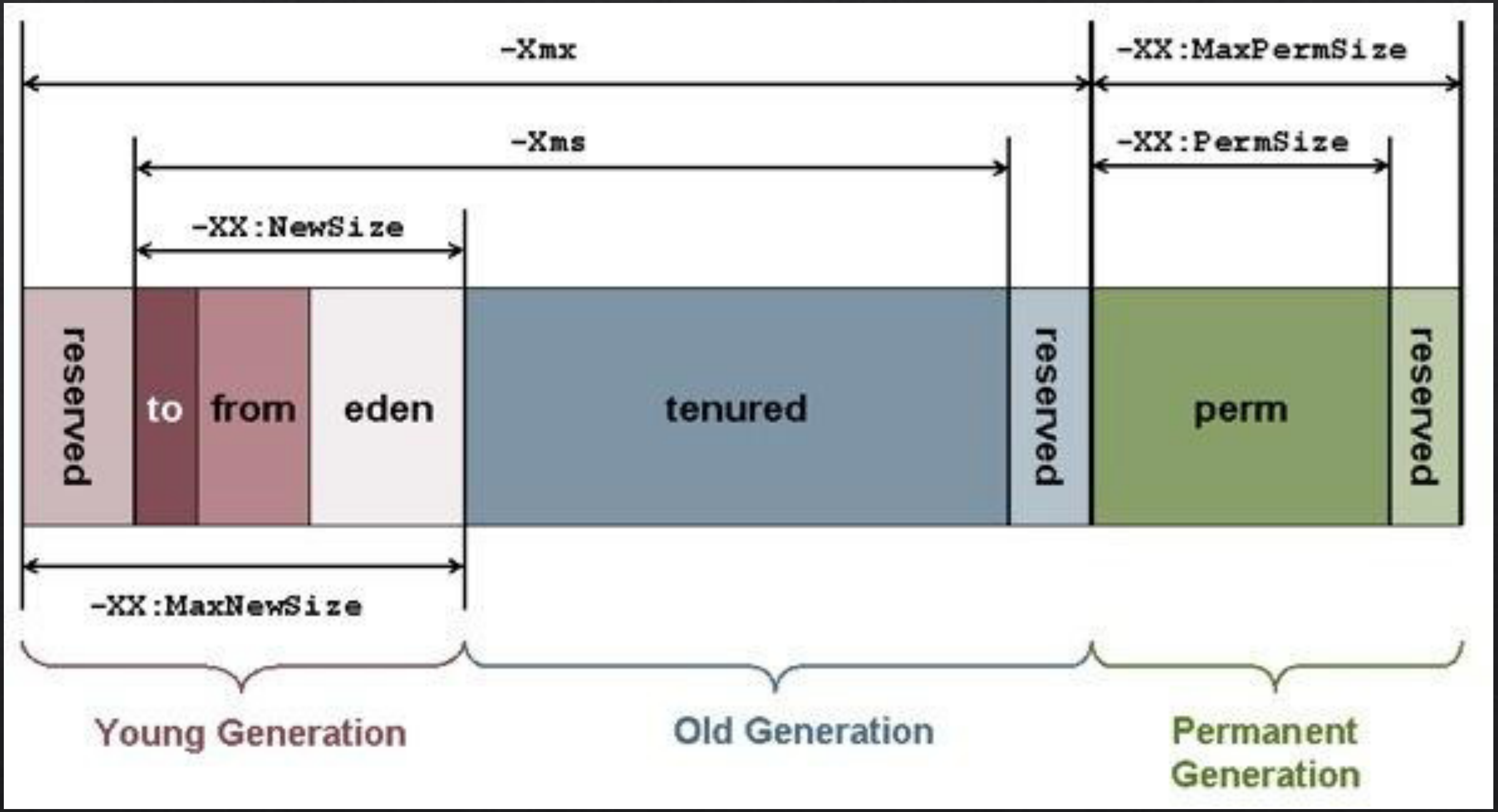


www.oracle.com/technetwork/java/hotspotfaq-138619.html#64bit_compilers

Are both -client and -server VM modes available in 64-bit Java?

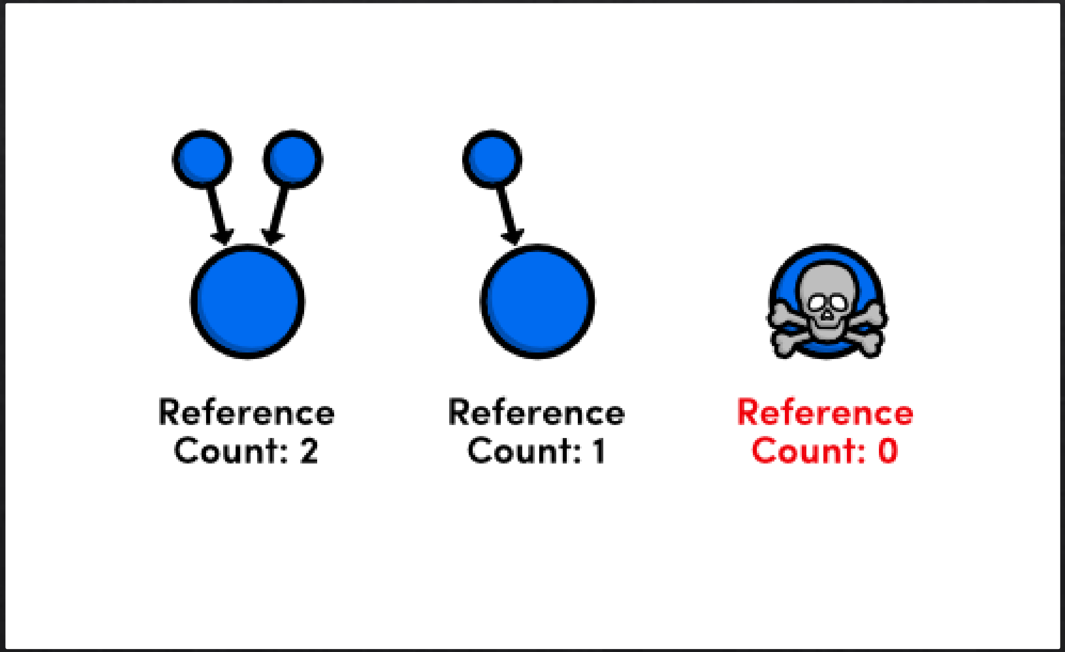
Currently only the Java HotSpot Server VM supports 64-bit operation, and the -server option is implicit with the use of -d64. This is subject to change in a future release.

Heap

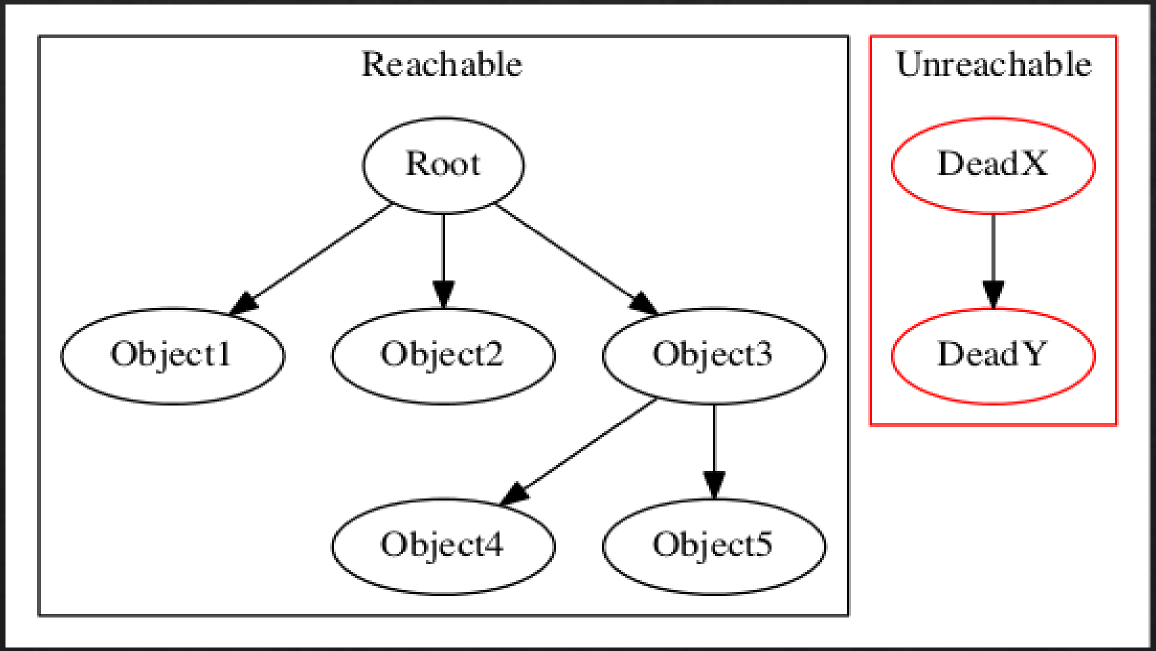

-Xmx
-XX:MaxPermSize
-Xms
-XX:PermSize
-XX:NewSize
reserved
to
from
eden
tenured
reserved
perm
reserved
-XX:MaxNewSize
Young Generation
Old Generation
Permanent Generation

How To Find A Garbage

Reference counting



Tracing



How To Remove A Garbage

Mark-Sweep

- **Phase 1:** Mark reachable objects;
- **Phase 2:** All non-marked memory areas are “free list”. It requires compacting.

Copy collector

- **Phase 1:** Use 2 memory areas, copy from one area to another and swap areas.

Generational Garbage Collection

Weak generational hypothesis:

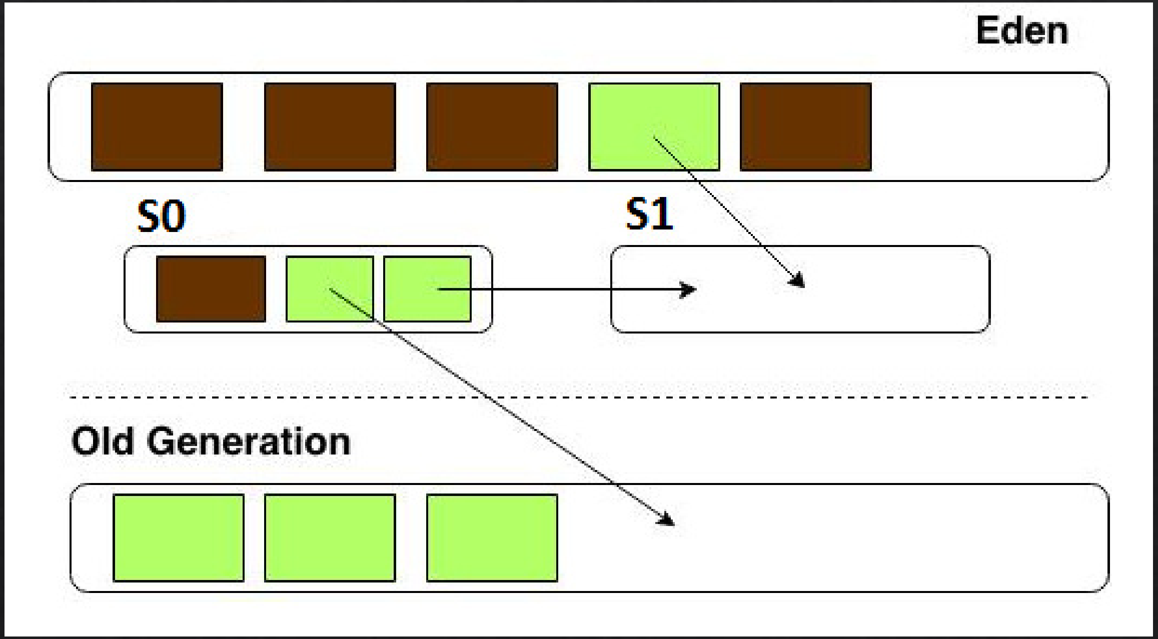
- Most objects soon become unreachable.
- **<u>References</u>** from old objects to young objects only **<u>exist</u>** in small numbers.

HotSpot JVM Garbage Collectors

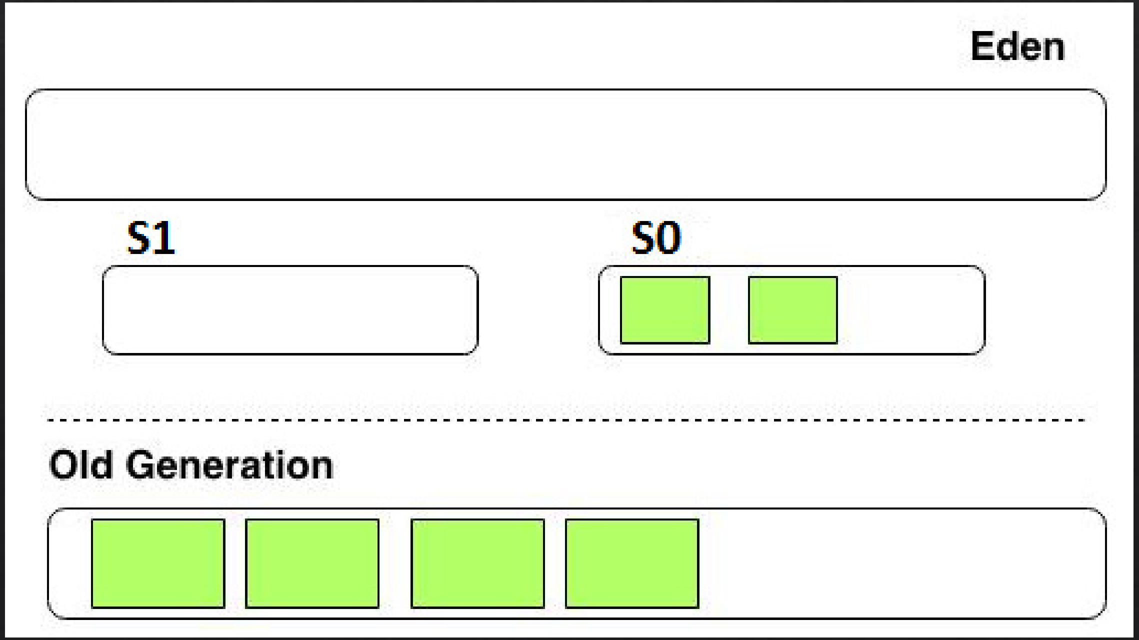
- Serial GC
- Parallel GC
- CMS GC (Concurrent Mark-Sweep)
- G1 GC

Serial GC: Minor GC

Before minor GC

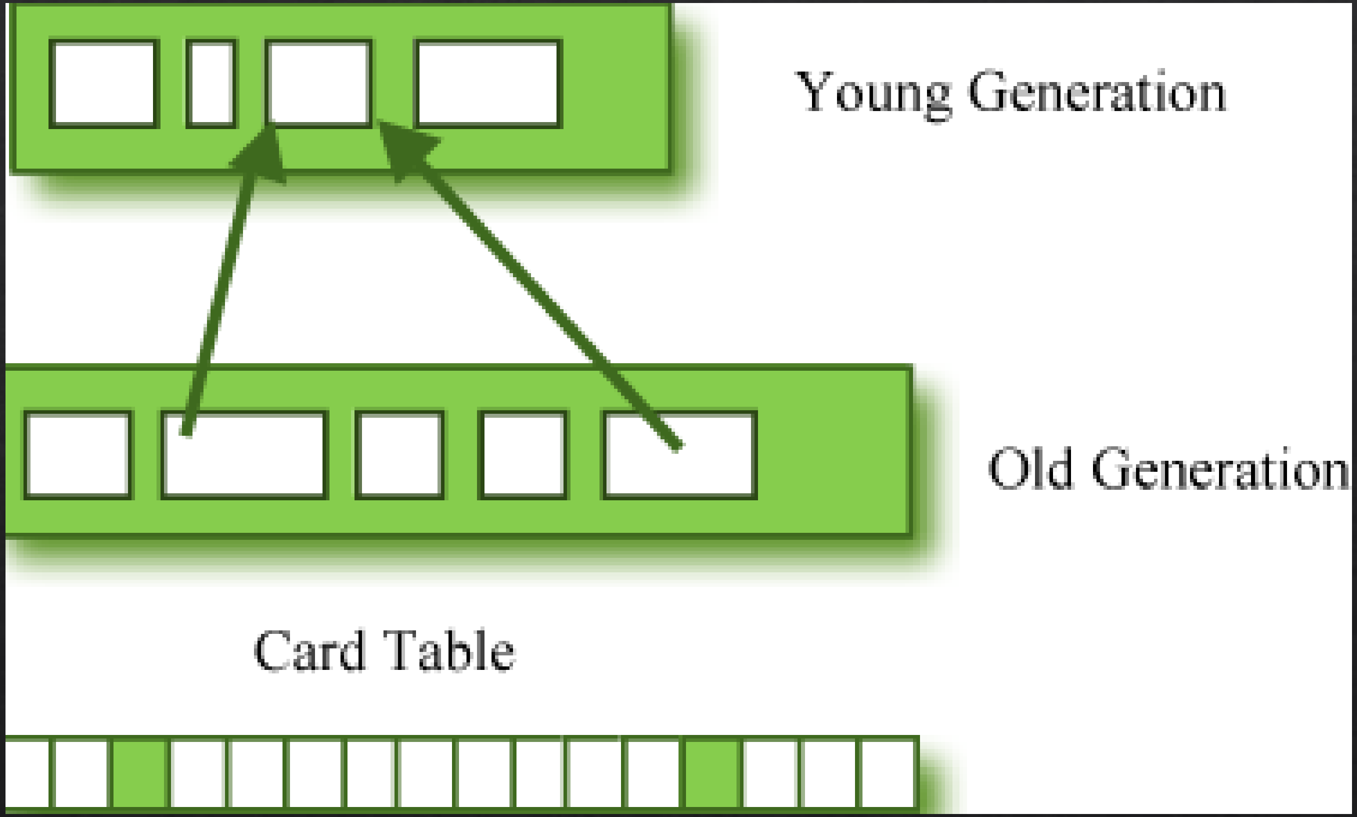


After minor GC

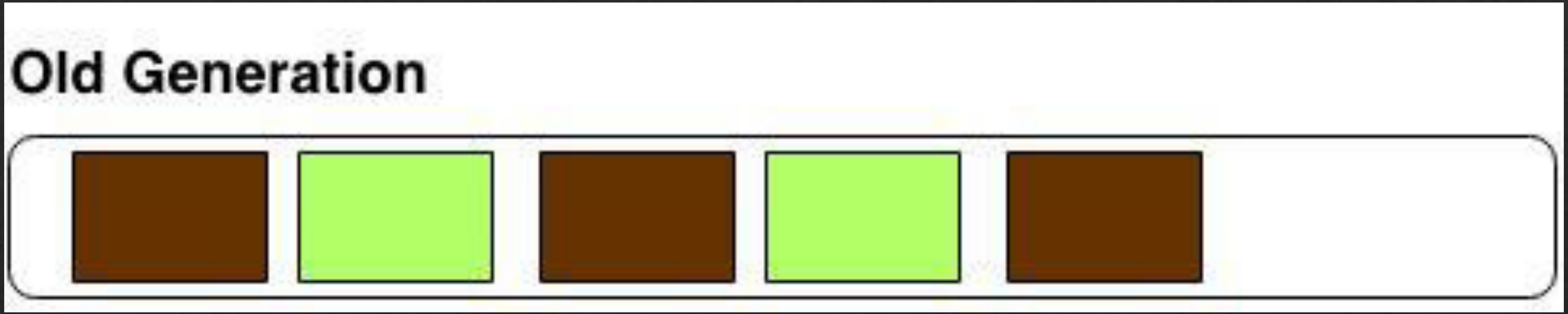


Card Table

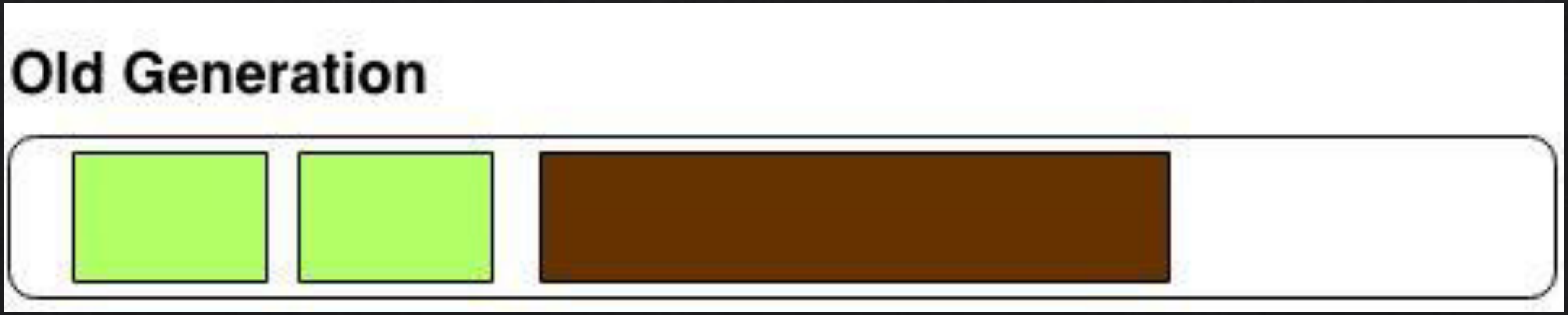
- 512 byte array in old gen
- 1 byte per cart
- old generation’s object references to young generation’s object = record in cart



Serial GC: Major GC

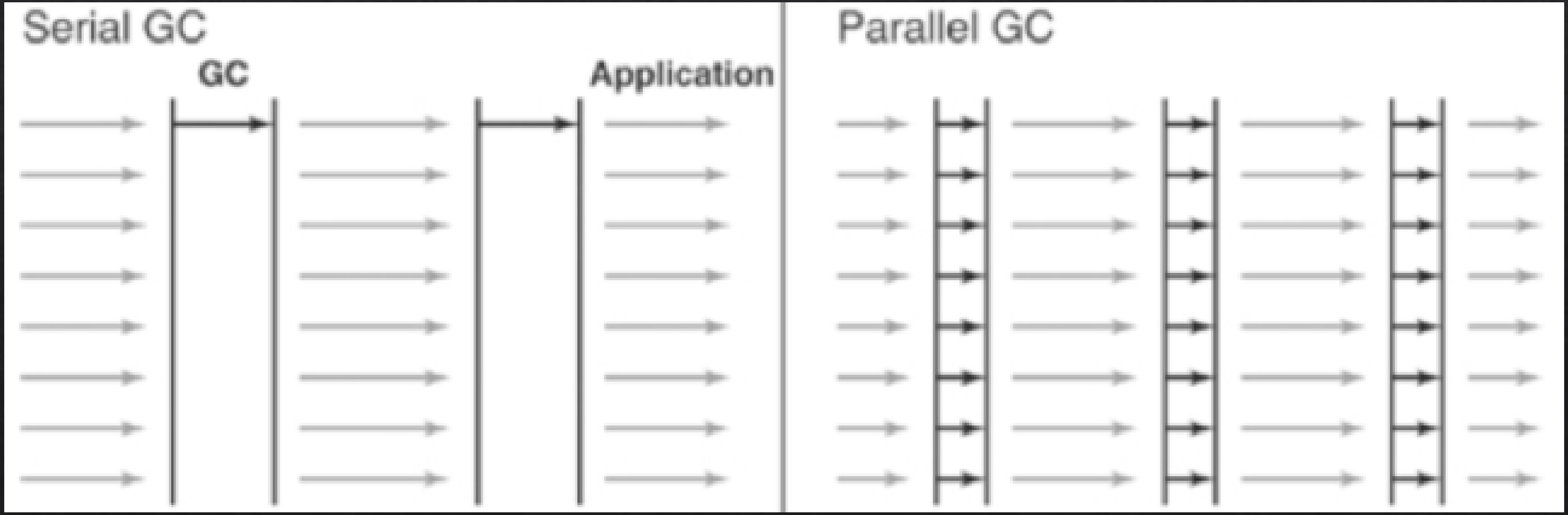


After compacting



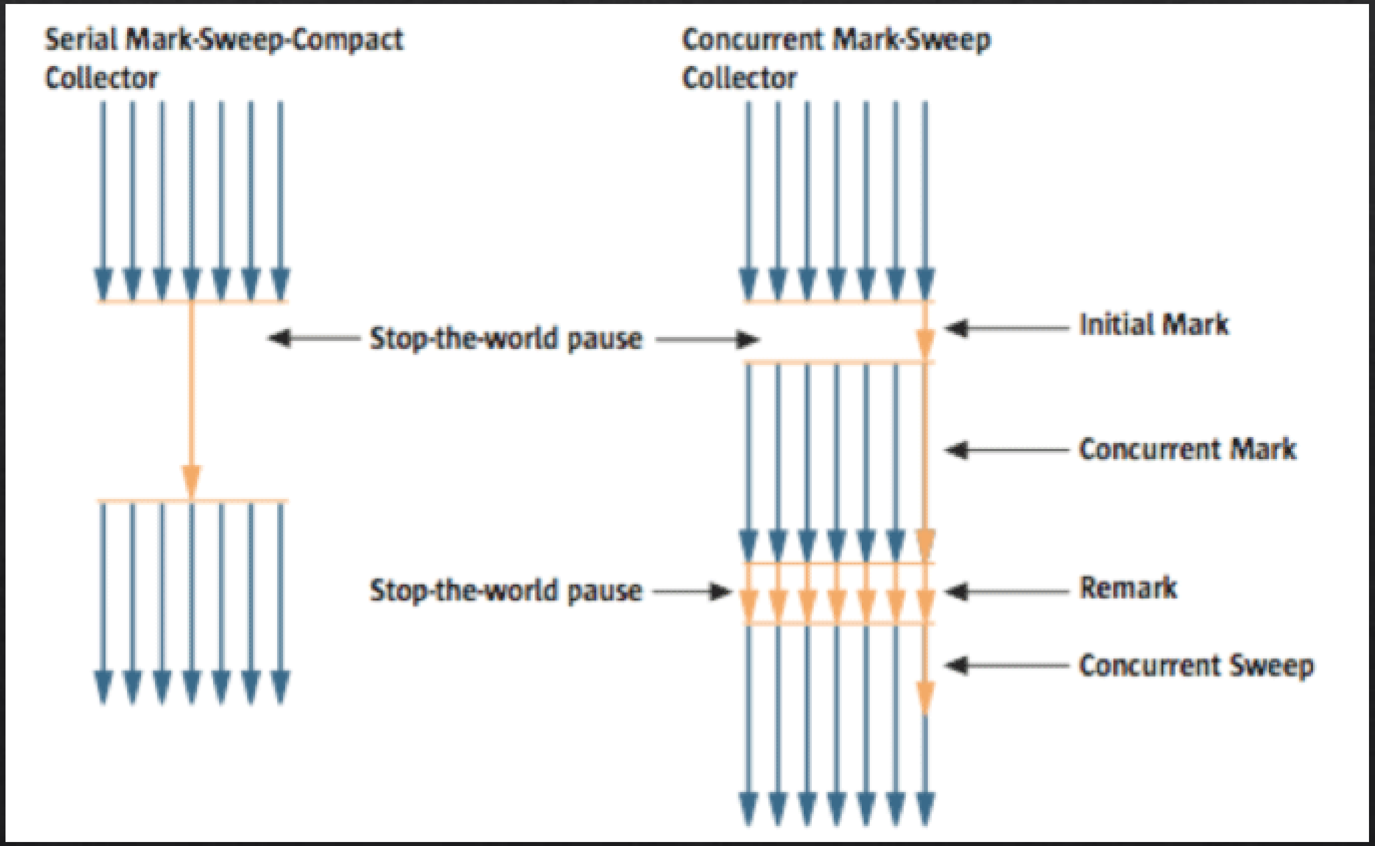
Parallel GC (-XX:+UseParallelGC)

The parallel GC uses several threads to process a GC and therefore faster

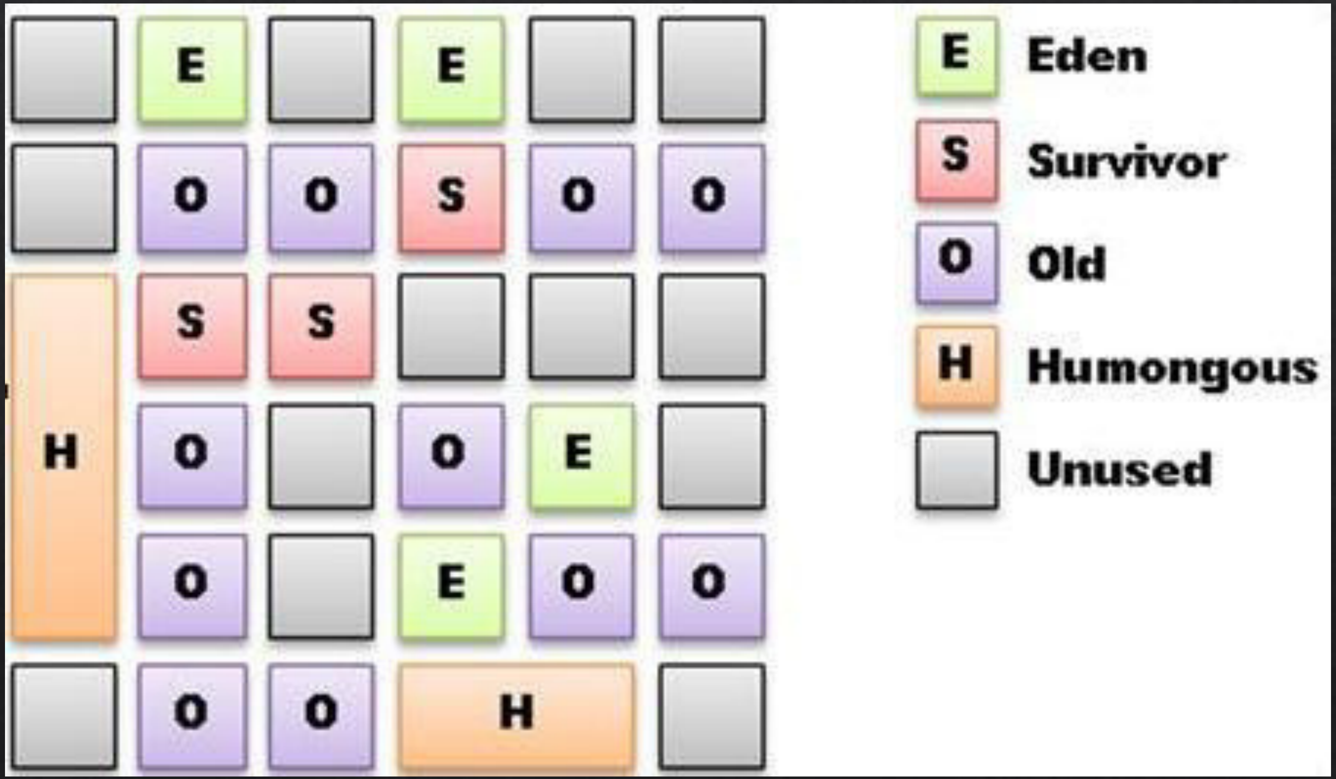


CMS GC (-XX:+UseConcMarkSweepGC)

- It uses more memory and CPU than other GC types.
- The compaction step is not provided by default.

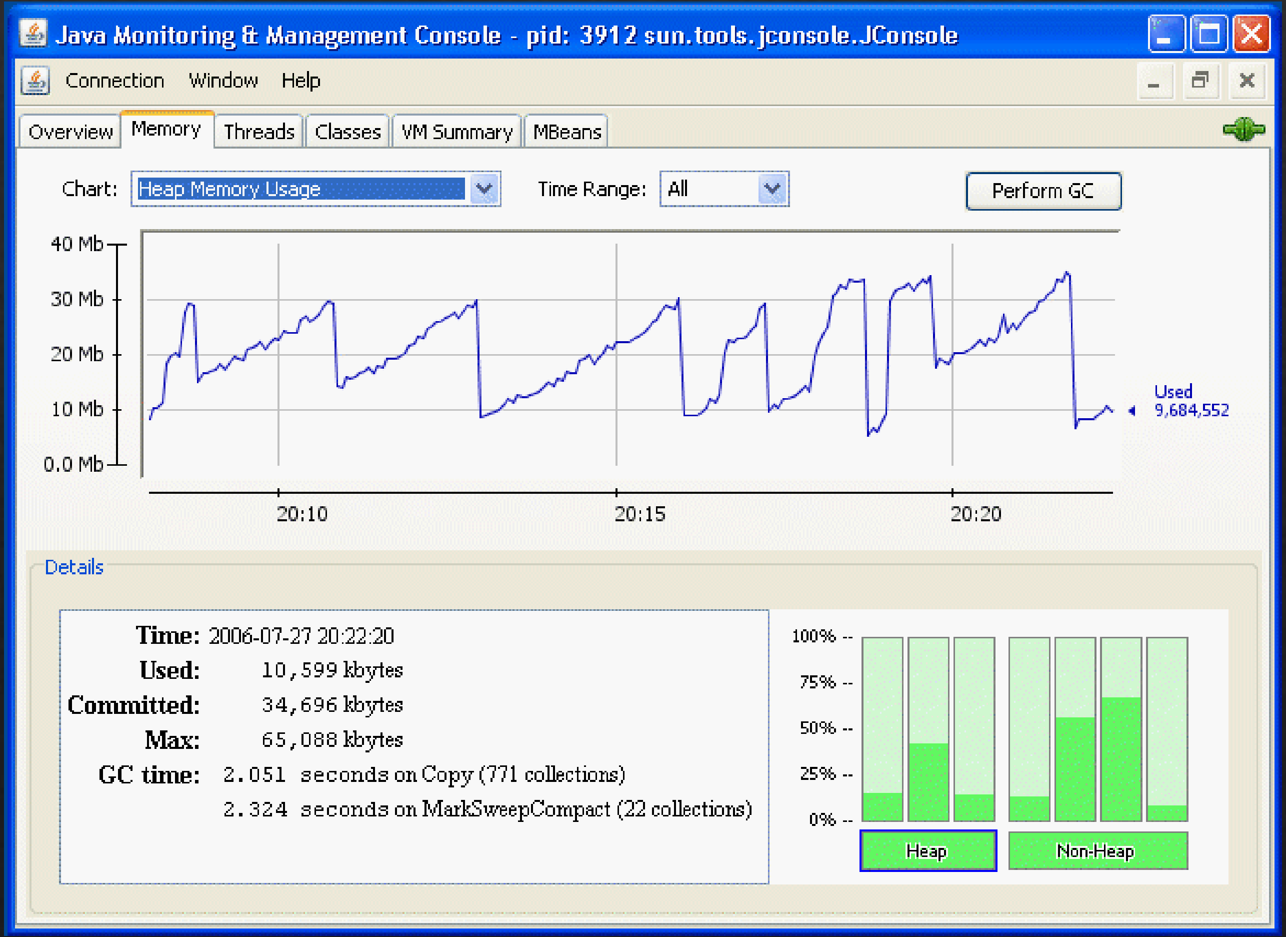


G1 (-XX:+UseG1GC)



How To Monitor JVM and GC

- jconsole
- jvisualvm



Time: 2006-07-27 20:22:20
Used: 10,599 kbytes
Committed: 34,696 kbytes
Max: 65,088 kbytes
GC time: 2.051 seconds on Copy (771 collections)
2.324 seconds on MarkSweepCompact (22 collections)

Performance issues

- Over-serialization and deserialization
- Overuse of finalizers
- Too much synchronization
- Not discarding unused variables
- Rampant use of System.out.printIn() and Logging
- Sessions that are not released when they are no longer needed
- Failing to close resources (for example, database and network connections)

GC Friendly Programming

- Avoid large objects
  - Expensive to allocate (Time & CPU instructions);
  - Large objects of different sizes can cause Java heap fragmentation
- Avoid data structure re-sizing
- Object pooling potential issues
  - GC duration is a function of live objects

Is “finalize()” an evil?

- When to use finalize() ?
- When this shit can be called by JVM?
- How much memory it consumes?
- Resurrection?

That's all Folks!