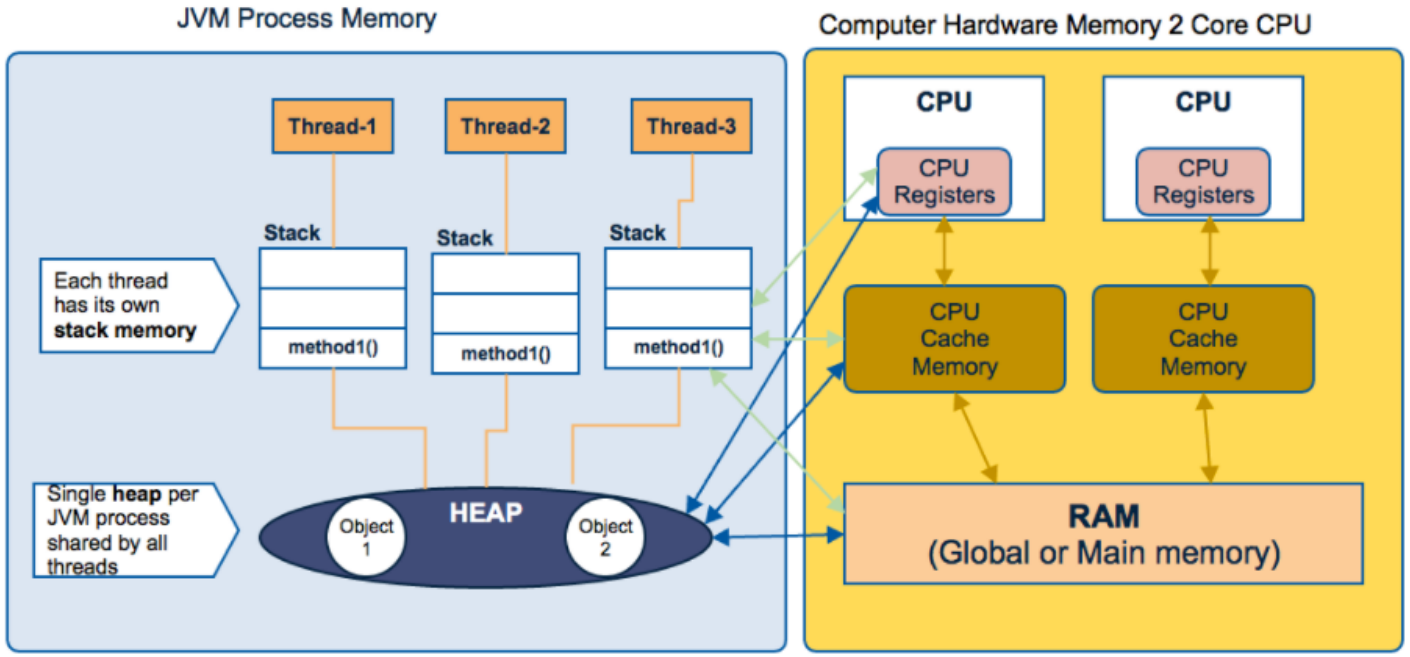
JVM Memory Management - Garbage Collection, GC Tools

Azat Satklichov
azats@seznam.cz,
https://github.com/azatsatklichov/Java-Features/tree/master/src/main/java/features/jvm/memory/management

Agenda

- ❑ Java Garbage Collector
- ❑ Where Storage Lives
- ❑ Types of Garbage Collections
- ❑ Java Memory Model
- ❑ How Garbage Collection Works – Minor, Major and Full GC
- ❑ Garbage Collectors Implementations
- ❑ Garbage Collection Tools - MXBeans, jstat, visualvm, visualgc

Where Storage Lives

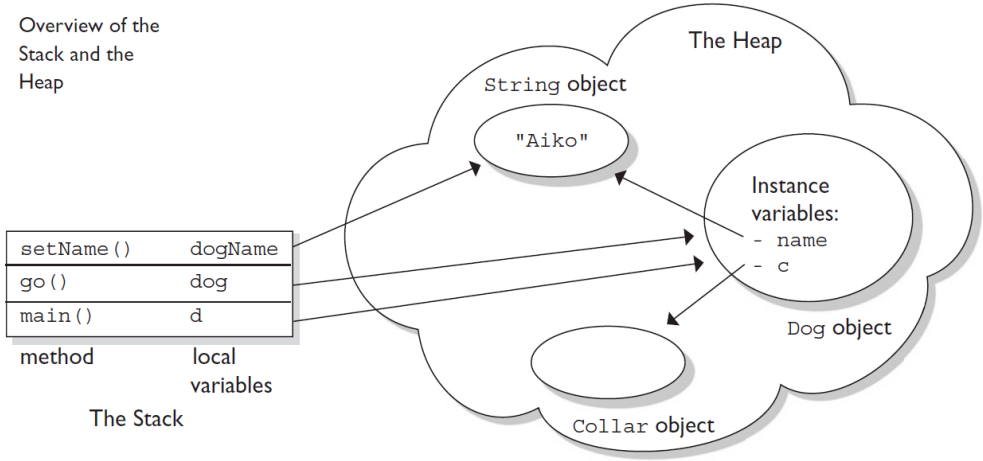


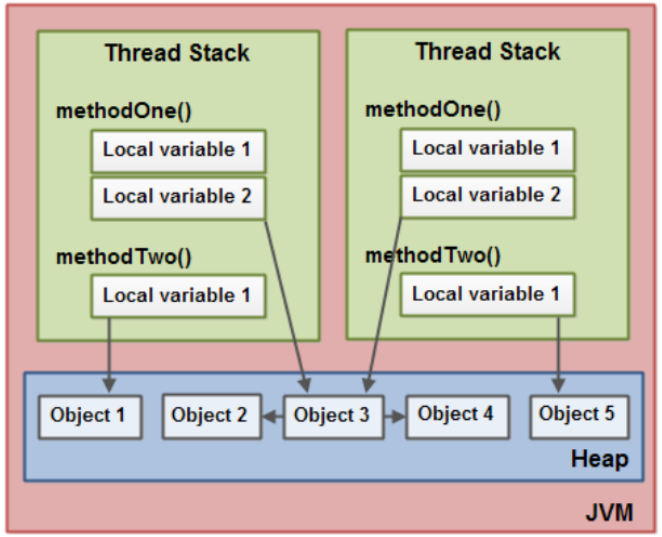
Hardware memory architecture:

- Registers – fastest storage
- The Stack – in RAM
- Static Storage - (fixed) in RAM
- Constant Storage - ROM
- Non-RAM Storage (persistence)

The **hardware memory architecture does not distinguish between thread stacks and heap.** On the hardware, both the thread stack and the heap are located in main memory.

CODE: WhereStorageLives.java





- Instance variables and objects live on HEAP
- Methods and Local (automatic/stack/method) vars. live on STACK, but its referencing object or created object live on STACK

Note: local variables created on stack whereas Instance variables created on HEAP

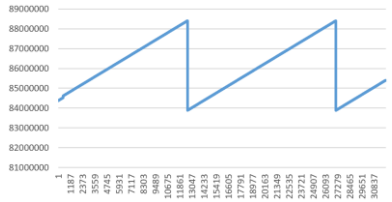
When objects and variables can be stored in various different memory areas in the computer, certain problems may occur. The two main problems are:

• Visibility of thread updates (writes) to shared variables.

• Race conditions when reading, checking and writing shared variables. See multithreading blog

Java Garbage Collector

Can we force to run GC?



Java's garbage collector provides an automatic solution to memory management.

- Garbage collection cannot ensure that there is enough memory, only it manages available memory as efficiently as possible.
- The garbage collector is under the control of the JVM. The JVM decides when to run the garbage collector.
- JVM will typically run the garbage collector when it senses that memory is running low.
- It tracks each and every object available in the JVM heap space and removes unused ones.
- You can only ask [System.gc()] for GC (can't force) but no guarantee when it happens.
- **Pros**: Automatic memory management (compare by C/C++..) . **Cons:** Managed by JVM. Stop the World issue. Not like man-mem-efficiency
- An object is eligible for garbage collection when can't be reached by any live thread

CODE: Memory Alloc. – **GC illustration** via addr of Obj using **Unsafe** by Sun.

When Objects are Eligible for Collection?

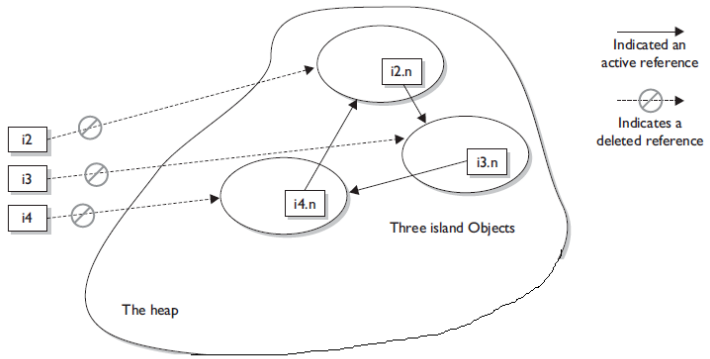
Once Garbage collector runs, it can discover those objects and delete them.

- Nulling a Reference
- Reassigning a Reference Variable
- Once method call is over BUT …
- Islands of Isolation //self
- .. ? //pools (interning), caches, …

```
public static void main(String[] args) {
    StringBuffer sb = new StringBuffer("Hey GC");
    System.out.println(sb); //not eligible for collection
    sb = null;  //Nulling a Reference makes it Eligible for GC
    // Now StringBuffer object is eligible for collection
}
```

```
public static void main(String[] args) {
    StringBuffer sb1 = new StringBuffer("Hello GC");
    StringBuffer sb2 = new StringBuffer("Hi GC");
    System.out.println(sb1); // "Hello GC" not eligible for collection
    sb1=sb2; // sb2 re-referenced to "Hi GC" object
    // Now "Hello GC" object is eligible for collection
}
```

```
public static void main(String[] args) {
    IsolatingAReference i2 = new IsolatingAReference();
    IsolatingAReference i3 = new IsolatingAReference();
    IsolatingAReference i4 = new IsolatingAReference();
    i2.i = i3; // i2 refers to i3
    i3.i = i4; // i3 refers to i4
    i4.i = i2; // i4 refers to i2
    i2 = null;
    i3 = null;
    i4 = null;
    // do calc
```

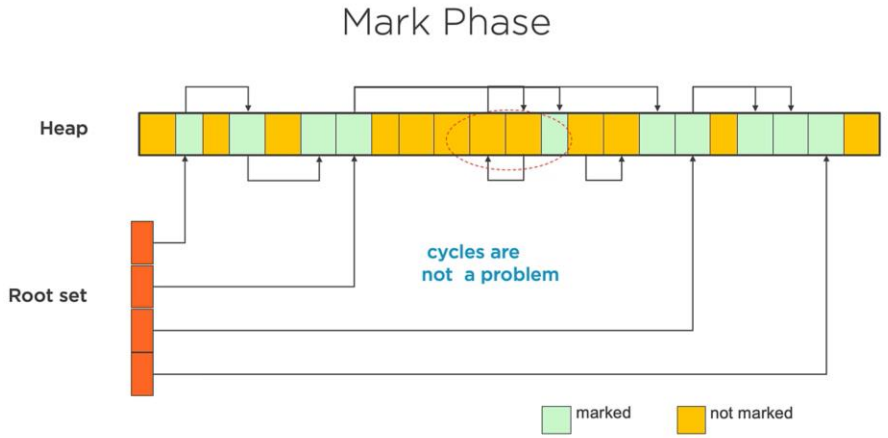


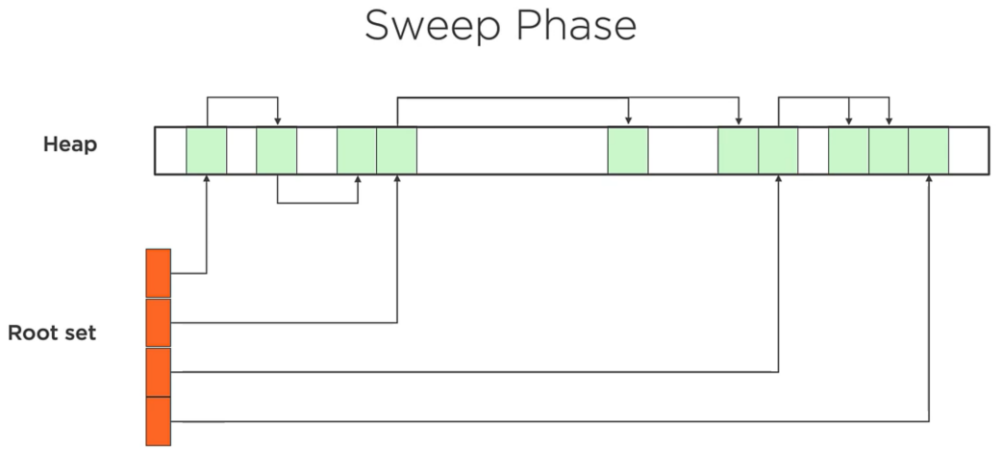

```
public static void main(String[] args) {
    Date d = getDate();
    System.out.println("d = " + d);
}

public static Date getDate() {
    Date d2 = new Date();
    StringBuffer now = new StringBuffer(d2.toString());
    System.out.println(now);
    return d2;
    //StringBuffer object will be eligible for Collection
    //Date object will not be
}
```

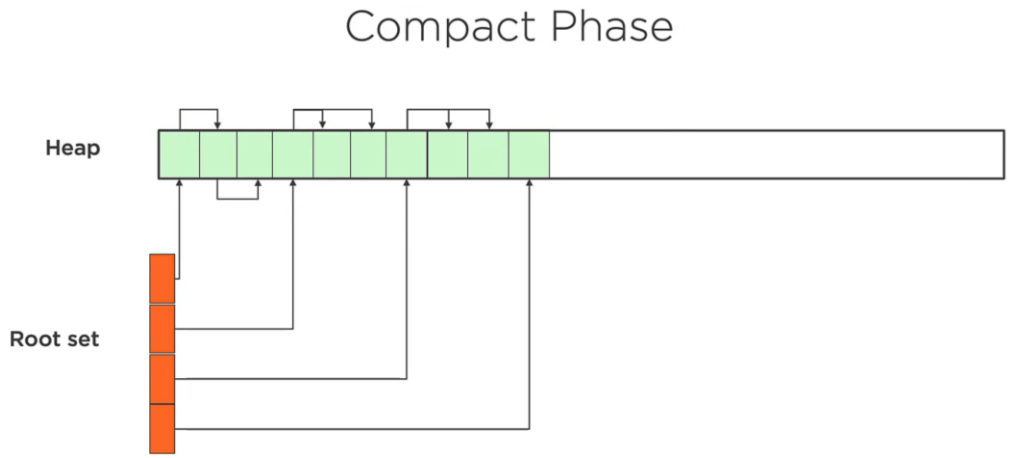
Types of Garbage Collection

- Reference Counting – Count inc. once object started to be referenced, decreased until zero and become eligible for GC. Issue with circular ref. It happens when one object refers to another, and that other one refers to the first object.

- Mark and Sweep – "**mark**" phase identifies the objects are still in being used, "**sweep**" phase removes un-used objs. "**compact**" phase defragments the memory.



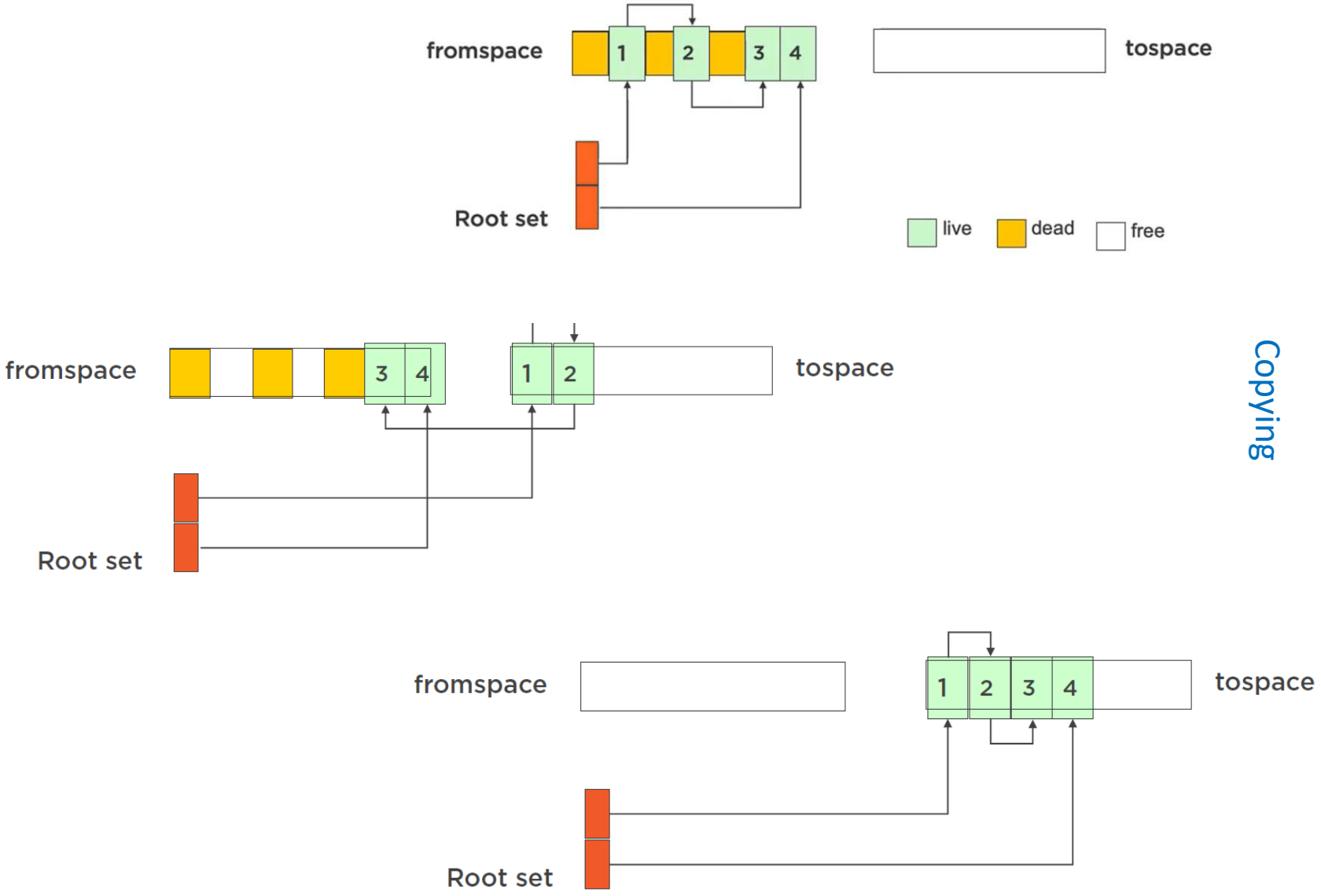




Mark and Sweep

Types of Garbage Collection

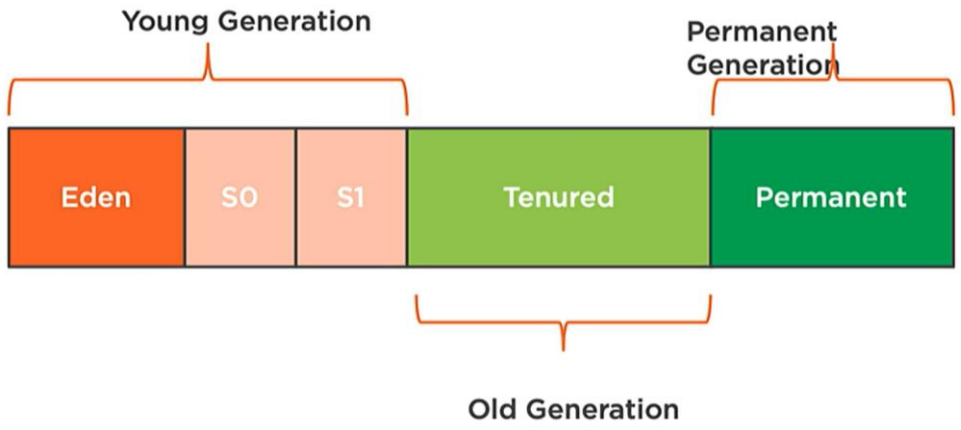
- Copying – mark & sweep happens but uses different spaces to manage the memory. During copying memory is compacted at the same time, and from space is cleared after all live objects are moved.
- Generational – Manages different generations for memory, long living objects (survived) are promoted to diff-gen. Sweeps happen more often in young generations than old one.
- Incremental – does not scan all the memory all the time.

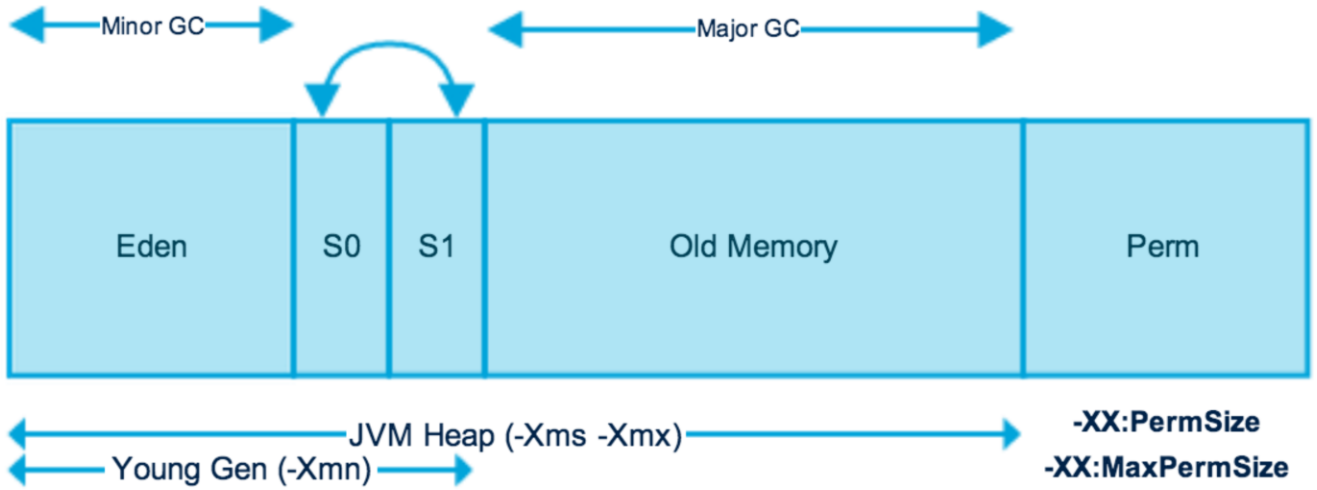


Java Memory Model

- Memory space is broken into two spaces: Young and Old (Tenured Space) Generations.
- The JVM internally separates the Young Generation into Eden and Survivor Spaces (S0, S1).
- Newly allocated objects are going to Eden [Eden Gardens] space.
- Objects **survived** in young generation promoted to one of Survivor spaces.
- One survivor space is used at a time, and objects are copied between survivor spaces
- Old generation where long lived objects (survived many times) live (most stable objects, ..)
- GC runs frequently on Young Gen. So motto is: die young or live forever.

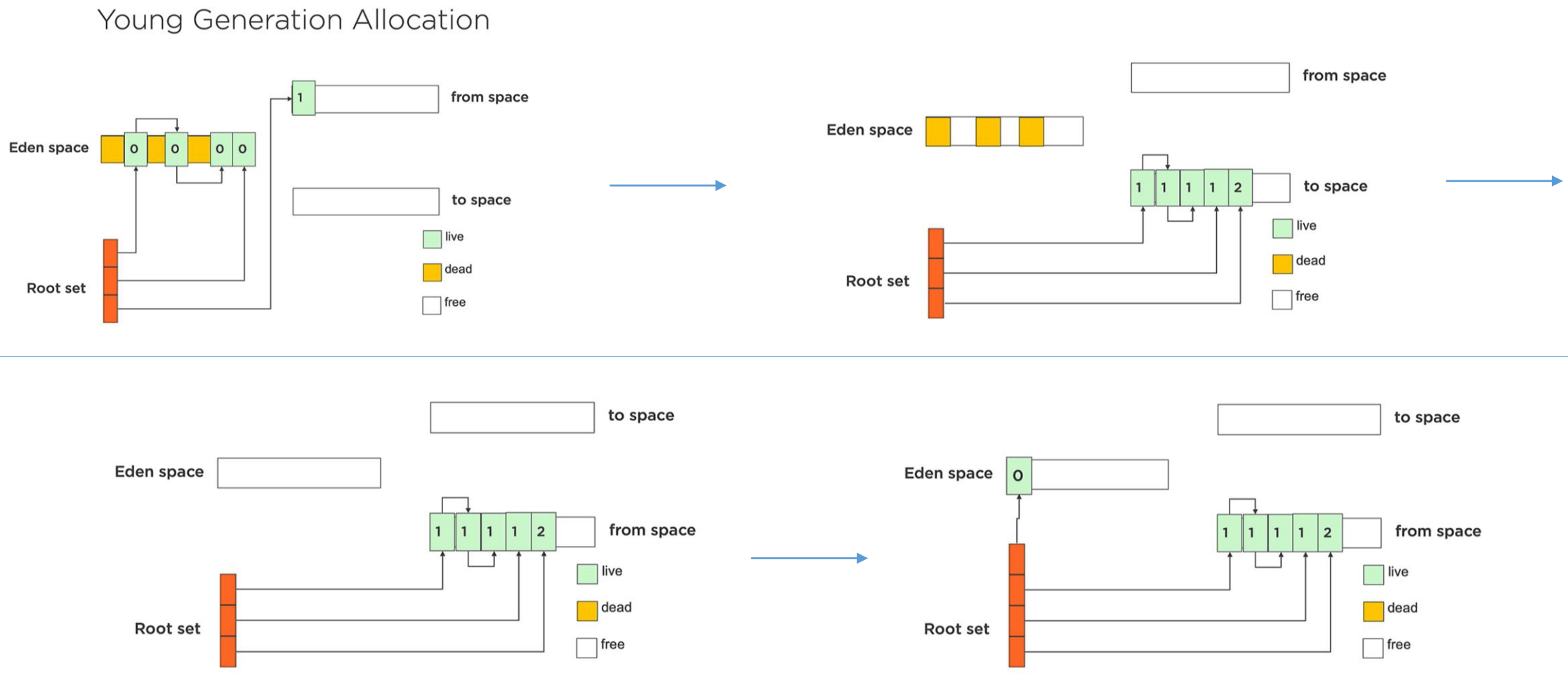
- Permanent space (never GC run here) – used by Java Runtime (class info stored,)





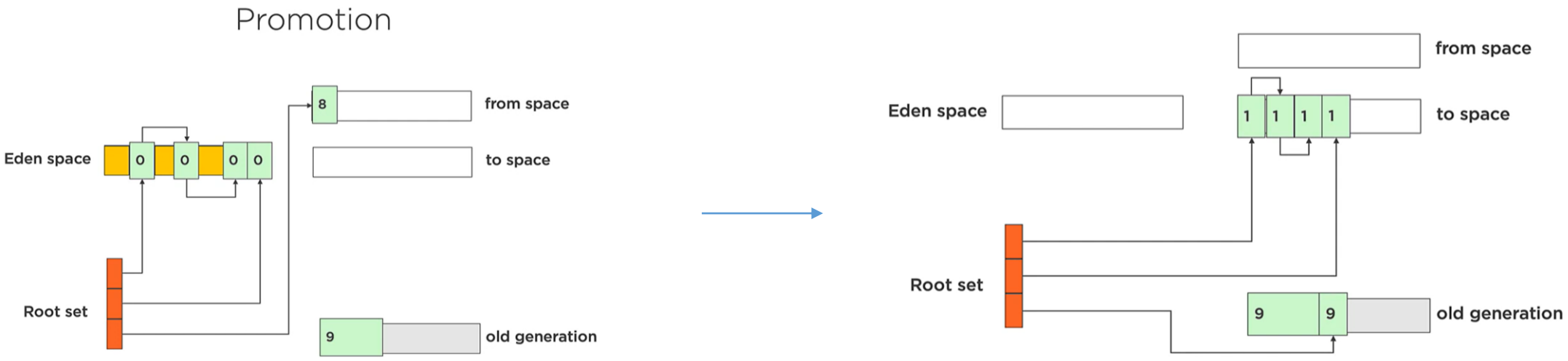
How Garbage Collection Works - Minor GC

- Minor GC happens when the Young Generation is FULL, happens more frequently.
- New objects are allocated to Eden space. Initial survivor space is called ‘from space’
- When GC runs (Minor GC happens), objects are copied to newer survivor space, eden is cleared, and surviver spaces will be swapped



How Garbage Collection Works – Major GC, Full GC

- **Major GC** triggered when tenured (Old Generation) is FULL. Happens less frequently. It is slower than Minor GC
- **Full GC -** Collects OLD and YOUNG gen. at the same time (on Oracle Java VM)
- In Major CG memory is copied from Young to Old generation – called promotion
  - after survived several times
  - once survivor space is full.
  - Or JVM has been told, create object in Old space (via JVM flag set: -XX:+AlwaysTenure)



Allocating Objects to Old Space

- Objects with certain size can be directly allocated to Old space. There is no JVM flag for this to allocate always.
- But can be done via: -XX:PretenureSizeThreshold=<n> if objSize>n then objects allocated to Old gen. If object size fits to TLAB, then JVM allocates it into TLAB (Thread Local Allocation Buffer).

Garbage Collectors Implementations

Which GC to choose? Consider: Stop the world events (1), memory fragmentation(2), Throughput(3). And profile (via mxbean, jstat, visualgc) the app. As close to production load.

- Serial Garbage Collector – simplest GC impl., works with a single thread (1) :

>java -XX:+UseSerialGC -jar App.java

Good for apps. that do not have small pause time req. and to run on client-style machines.

- Parallel [**Parallel GC, Parallel Old GC**] Garbage Collector – also called Throughput Collector (3). Uses multiple threads for managing heap space. Mostly used in production servers.

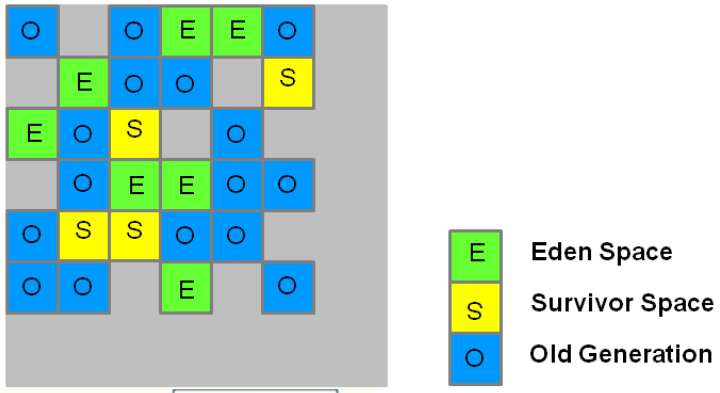
Parallel for MinorGC and serial (1) for MajorGC -XX:+UseParallelGC, and parallel for both -XX:+UseParallelOldGC.
Tune max. gc. *threads and pause time, throughput, and footprint* (heap size). Max-pause-goal: -XX:MaxGCPauseMillis=<N>

- CMS Garbage Collector (low latency) – Deprecated in Java 9, removed in 14. Concurrent Mark Sweep uses multiple garbage collector threads for gc. designed for apps. in need shorter gc. pauses, throughput is higher …

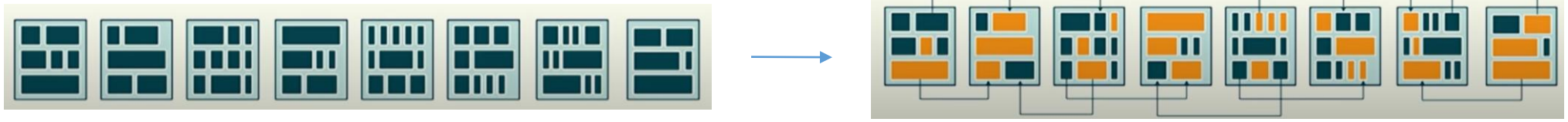
–XX:+UseConcMarkSweepGC , –XX:-UseParNewGC. With serial (1) young space collector
–XX:+UseConcMarkSweepGC , –XX:+UseParNewGC. With parallel young space collector
We can limit the number of threads in CMS collector using –XX::ParallelCMSThreads=<n> JVM option.

Java 8 JVM has new param – Optimizing heap-memory for reducing the unnecessary use of memory by creating too many instances of the same *String.* > -XX:+UseStringDeduplication

Garbage Collectors Implementations

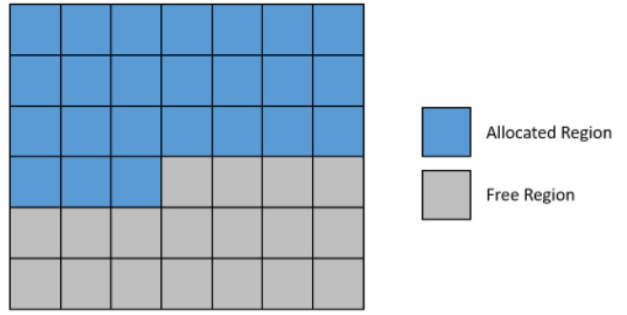


- G1 (Garbage First, or **compacting-collector**) Garbage Collector – incremental GC. Paper 2004, experimental in Java 6, official in Java 7, and default GC since Java 9. Designed for apps running on multi-cores with large memory space. Goal is: **Througput/Latency balance.** Tuneable pause goals. A bit more CPU intensive. Regions (#2000, size 2MB for 4GB heap)
- Evacuation (moved/copied between regions). Minor & Major collection,..
- >java -XX:+UseG1GC -jar App.java



- Epsilon Collector – A **No-Op GC**. Apps with predictable, bounded memory usage, performance[stress] testing, short-lived obj. Allocates memory but not collect any garbage (memory allocation), once the Java heap is exhausted, the JVM will shut down. Params: -XX:+UnlockExperimentalVMOptions, -XX:+UseEpsilonGC
- Z Garbage Collector (ZGC is **scalable low** latency GC) [pause time **under 10 ms**, maybe in future 1 ms] – is a scalable low-latency garbage collector, experimental in Java 11 for Linux, in Java 14 for Win. & macOS. Since Java 15 ZGC is onwards. Single generation(all opt. done here via colored pointers), **No pause time increase with heap size increase**, Scale to **multi-terabyte heaps**. -XX:+UnlockExperimentalVMOptions, -XX:+UseZGC

- Shenandoah Garbage Collector (derived from G1, scalable low-latency garbage collector) – experimental in Java 12, [Brooks pointers] support large Heaps, Low pause times.
Contributed by **RedHat** to OpenJDK. Became product feature in Java 15. Oracle JDK and Open JDK has not this feature. Single generation. Can be used in Java 8 so no need colored pointers like ZGC.
-XX:+UnlockExperimentalVMOptions -XX:+UseShenandoahGC



The Z Garbage Collector (ZGC): Low Latency in JDK 11with Per Liden

Garbage Collection Tools

- MX Beans – management bean [GarbageCollectorMXBean] to monitor GC, to get name of garbage collections, numbers of collectors, times of collections and info about memories, … Each bean for per memory manager (old, young)

CODE: MainMXBeanDemo.java, E_BigObjMemoryAddrs
Run with: a) //default GC
b) -XX:+UseParallelGC
c) -XX:+UseConcMarkSweepGC

- Jstat – command line tool to monitor memory and GC activities, also can be used to profile remote JVM

Run with: jstat –option <pid> <inteval> <count>
E.g. >jstat -gc 10632 100 10

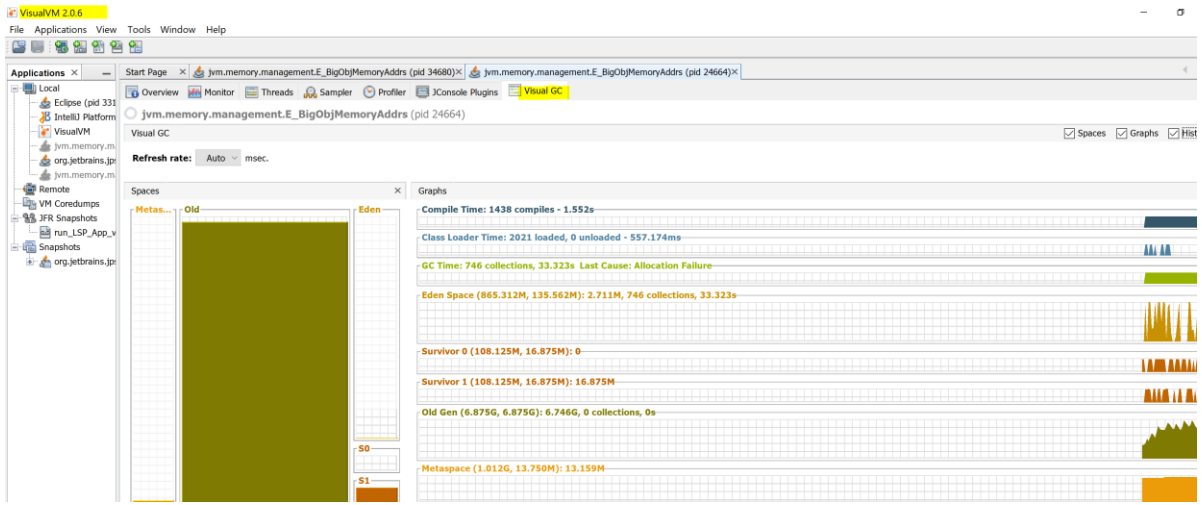
Try with: -XX:+UseParallelGC, and G1

See jstat descriptions here

```
C:\Users\as892333>jps
25824
36576 Jps
30324 E_BigObjMemoryAddrs
27996 Launcher
33196 Eclipse

C:\Users\as892333>jstat -gcutil 30324
  S0     S1     E      O      M     CCS    YGC     YGCT    FGC    FGCT    CGC    CGCT     GCT
  0.00  99.99  35.99  50.99  79.89  66.89    386   16.563     0    0.000    22    0.115   16.678

C:\Users\as892333>jstat -gccause 30324
  S0     S1     E      O      M     CCS    YGC     YGCT    FGC    FGCT    CGC    CGCT     GCT    LGCC                 GCC
  0.00  99.99  37.99  52.47  80.33  66.89    512   22.065     0    0.000    28    0.148   22.213 Allocation Failure   Allocation Failure

C:\Users\as892333>jstat -gccapacity 30324
 NGCMN       NGCMX        NGC         S0C         S1C           EC       OGCMN       OGCMX        OGC         OC       MCMN      MCMX       MC      CCSMN      CCSMX     CCSC     YGC    FGC    CGC
   192.0   1107520.0   173376.0    17280.0    17280.0   138816.0        64.0   7209408.0   5808688.0   5808688.0        0.0  1056768.0     4864.0        0.0  1048576.0    512.0     612      0     34

C:\Users\as892333>jstat -gc 30324
 S0C    S1C    S0U    S1U      EC       EU        OC         OU       MC     MU    CCSC   CCSU   YGC     YGCT    FGC    FGCT     CGC    CGCT     GCT
17280.0 17280.0  0.0   17279.1 138816.0   0.0    5808688.0  2404206.3  4864.0 3910.2 512.0  342.5     798   35.420   0      0.000  44     0.269   35.689

C:\Users\as892333>jstat -gc 30324 1000 10
 S0C    S1C    S0U    S1U      EC       EU        OC         OU       MC     MU    CCSC   CCSU   YGC     YGCT    FGC    FGCT     CGC    CGCT     GCT
17280.0 17280.0 17280.0  0.0   138816.0 97152.2  5808688.0  2666411.2  4864.0 3910.2 512.0  342.5     873   39.133   0      0.000  48     0.289   39.422
17280.0 17280.0 17280.0  0.0   138816.0 69394.5  5808688.0  3097911.0  4864.0 3910.2 512.0  342.5     887   39.715   0      0.000  50     0.305   40.020
17280.0 17280.0 17280.0  0.0   138816.0 66618.7  5808688.0  2740607.6  4864.0 3910.2 512.0  342.5     901   40.334   0      0.000  50     0.305   40.639
17280.0 17280.0 17280.0  0.0   138816.0 127685.7 5808688.0  2967835.0  4864.0 3910.2 512.0  342.5     915   40.976   0      0.000  51     0.305   41.281
17280.0 17280.0 17280.0  0.0   138816.0   0.0    5808688.0  2706199.8  4864.0 3910.2 512.0  342.5     928   41.619   0      0.000  52     0.318   41.937
17280.0 17280.0  0.0   17279.1 138816.0   0.0    5808688.0  2474330.0  4864.0 3910.2 512.0  342.5     939   42.244   0      0.000  52     0.318   42.562
17280.0 17280.0  0.0   17279.1 138816.0 33309.4  5808688.0  3128731.5  4864.0 3910.2 512.0  342.5     952   42.831   0      0.000  53     0.319   43.150
17280.0 17280.0  0.0   17279.1 138816.0 38861.0  5808688.0  3108791.6  4864.0 3910.2 512.0  342.5     965   43.461   0      0.000  54     0.335   43.796
17280.0 17280.0 17280.0  0.0   138816.0 63842.8  5808688.0  2837968.4  4864.0 3910.2 512.0  342.5     979   44.070   0      0.000  54     0.335   44.404
17280.0 17280.0  0.0   17279.1 138816.0   0.0    5808688.0  3146669.0  4864.0 3910.2 512.0  342.5     991   44.702   0      0.000  56     0.345   45.047
```

CODE: E_BigObjMemoryAddrs

- Visualvm - VisualVM monitors and troubleshoots applications running on Java



Install Pluin for VisualGC

- visualgc – get more info in VM what is happening on GC

THANK YOU

References

https://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/g1_gc_tuning.html
https://www.baeldung.com/jvm-experimental-garbage-collectors
https://www.journaldev.com/2856/java-jvm-memory-model-memory-management-in-java
https://docs.oracle.com/javase/7/docs/technotes/guides/vm/G1.html
http://tutorials.jenkov.com/java-concurrency/java-memory-model.html
https://www.youtube.com/watch?v=WU_mqNBEacw
https://shipilev.net/
https://www.youtube.com/watch?v=Gee7QfoY8ys https://www.youtube.com/watch?v=WU_mqNBEacw