Java & JVM Performance Optimisation

# Topics

## Your KPI in Performance

Latency, throughput,

## Recommended JVM Parameters

Always turns on GC log:

|  |  |
| --- | --- |
| < JDK 9 | > JDK 9 |
| -XX:+PrintGCDetails  -Xloggc:<file\_name>  Optional:  -XX:+PrintGCTimeStamps  -XX:+PrintGCDateStamps  -XX:+PrintReferenceGC  -XX:+PrintTenuringDistribution  -XX:+PrintGCApplicationStoppedTime  -XX:+UseGCLogFileRotation  -XX:NumberOfGCLogFiles=10  -XX:GCLogFileSize=10M | -Xlog:gc:gc\*,gc+ref=debug,gc+age=trace,  gc+heap=debug:file=gc%p%t.log:tags,  uptime,time:filecount=10,filesize=10m  e.g.  -Xlog:gc:/tmp/idea.vgc 🡺 log GC activities into /tmp/idea.vgc file.  Using vi or vim editor, use ‘search and replace’ to look for certain keyword: ***:%s/old\_text/new\_text/g***  ***:q!*** 🡺 quit discarding any changes  ***:%s/.\*)*** 🡺 look for lines with .\*), e.g. in (3.24ms)  .,$!sort -n -r 🡺 to sort the search keyword in reversed order |

-XX:HeapDumpOnOutOfMemoryError

NOTE: to check what GC options are available and enabled, type ***$ java -XX:+PrintFlagsFinal -version***



### Use JFR as light-weight logging

-XX:StartFlightRecording:filename=/jfr/app1.jfr,dumponexit=true

To enable JFR in an already running VM:

*$ jcmd <pid> JFR.start*

e.g. *$ jcmd 9979 JFR.start*

To dump a recording from running VM with max size of data:

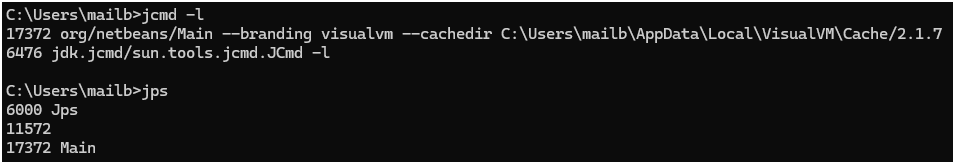
*$ jcmd <pid> JFR.dump filename=<fileName.jfr> maxsize=<size>*

e.g. *$ jcmd 9979 JFR.dump filename=app1.jfr maxsize=75MB*

## Tools

### jdk/bin

jps 🡺see process id of running application

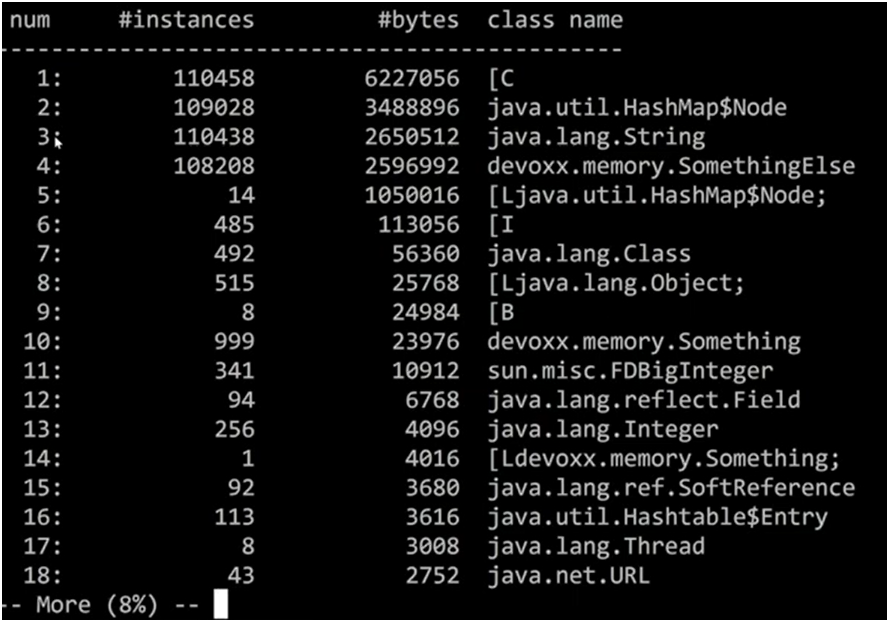


jcmd 🡺

[jmap](https://docs.oracle.com/javase/8/docs/technotes/guides/troubleshoot/tooldescr014.html) 🡺 print memory-related statistics for a running VM or core file

$ jmap -histo:live <pid> 🡺print live objects in heap after gc (by ‘live’ option)

$ jmap -heap <pid> 🡺print Java heap information



Info:

* [B ==> array of primitive byte
* [C ==> array of primitive char
* [L<name\_of\_class>$<name\_of\_inner\_class>; ==> array of objects of certain type. Note: semicolon at end

### JFR (Java Flight Recorder) & JMC (Java Mission Control)

For system performance metrics

### JMH (Java Microbenchmark Harness)

For code performance metrics

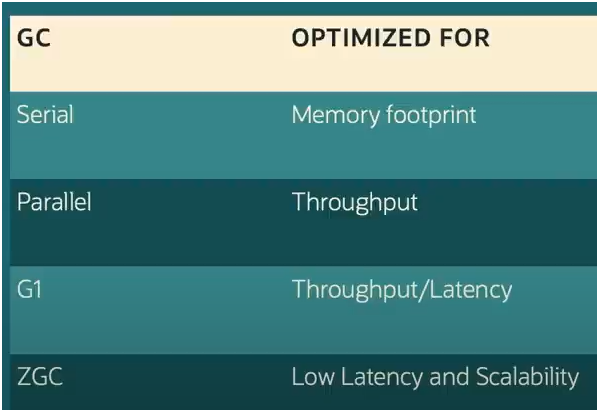
## GC

Important JVM parameters

# Performance Optimisation Steps

1. Remove all existing JVM parameters
2. Measure performance among possible GC algorithms – start with G1GC:
   1. G1: -XX:+UseG1GC (default-no need to specify)
   2. If throughput is insufficient, try enabling parallel: -XX:+UseParallelGC
   3. If pause times are too long, try ZGC: -XX:+UseZGC
   4. If most of the performance characteristics are pretty close, try tuning. GC Tuning Guide [here](https://docs.oracle.com/en/java/javase/23/gctuning/).

NOTE: focus of GC algorithms as shown below.



# Analysis Methodology

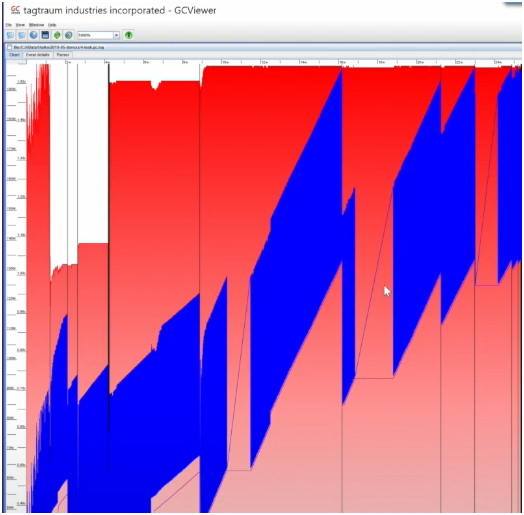
## Memory Leak (OOME)

1. Do I have a leak?
2. What is leaking? Which class is leaking?
3. What is keeping objects alive?
4. Where is it leaking from

### 1 – Do I have a leak?

Tool:

* GC log, and
* GCViewer – analyse GC log and see GC and heap pattern



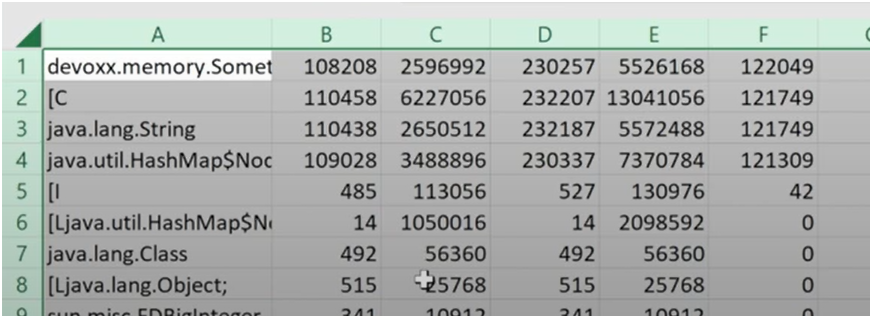
Heap is getting gradually fuller, i.e. GC cannot cope with allocation rate

### 2 – What is leaking?

$ jmap -histo:live <pid> > histo1.txt

$ jmap -histo:live <pid> > histo2.txt

As these 2 records are done after full GC, the top objects with growing totals are the leaking objects.



(4 objects have grown in number after 2 GCs - see column F = D - B.)

### 3 – What is keeping objects alive?

Tools:

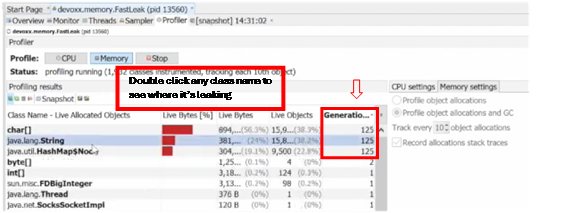
* JVM param: -XX:+HeapDumpOnOutOfMemoryError, or
* $ jmap -dump:live,file=<file\_path> <pid>, or
* $ jcmd <pid> GC.heap\_dump <file\_path>, or
* JMX from VisualVm, JMC

### 4 - Where is it leaking from?

Tools:

* VisualVm

Track the objects for ‘generations’:



# ZGC (Production ready since 2015)

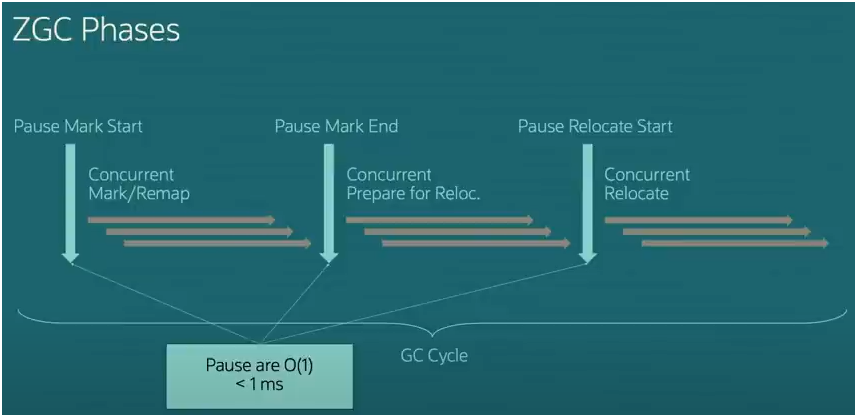
## Source

[YouTube presentation on ZGC](https://youtu.be/OnodHoNYE1Y?si=XJdNz5Pa8GdW51sy).

## Notes

Benefits of ZGC:

* Scalable: able to handle heap size of few MB all to way to TB
* Low latency: O(1) pause time of < 1ms regardless of heap size
* Auto tuning: simpler configuration
* Max application throughput reduction of 15% compared to G1GC
* 99.9 percentile of <150 us pause times
* Generational region-based heap which benefits from the weak generational hypothesis, i.e. most objects are short lived.



## Tuning Parameters Needed in Comparison

|  |  |
| --- | --- |
| **GC** | **Tuning parameter possibly needed** |
| Before ZGC (mostly with generational GC, i.e. GC that has heap split into young and old generations) | Generation sizing (young generation size):  -Xmn  Tenuring threshold (# of young collections before objects are promoted to old gen):  -XX:TenuringThreshold  IHOP-Initiating Heap Occupancy Percent (when to start collecting the old gen)  -XX:InitiatingHeapOccupancyPercent  Number of threads:  -XX:ConcGCThreads  Max heap size:  -Xmx |
| ZGC | Max heap size:  -Xmx  All other parameters are dynamically and automatically tuned by heuristics. |