JVM Performance Optimisation Training Summary (Sept 2024 by yCrash)

Performance KPI (Key Performance Index)

1 – Throughput

⇨ Percentage of work-related tasks done. E.g. in 24 hours, a system spends a total of 5 minutes for GC. ∴throughput = (100 - (5 / (24 x 60)) x 100)% = 99.652%.

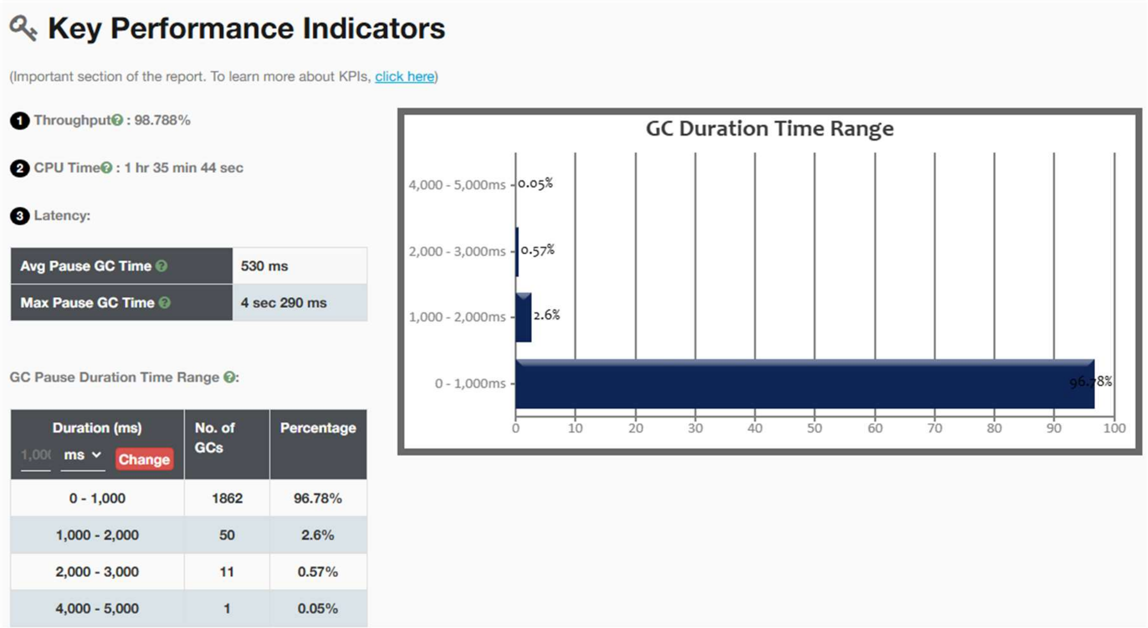
2 – Latency

⇨ Amount of time taken for GC: maximum, average, distribution.

3 – Footprint

⇨ Amount of CPU time taken for GC.

GCEasy can be used to display the information based on GC log as shown below. See GC log section for more details on GC log).



Key Performance Indicators

(Important section of the report. To learn more about KPIs, click here)

1 Throughput : 98.788%

2 CPU Time : 1 hr 35 min 44 sec

3 Latency:

Avg Pause GC Time	530 ms
Max Pause GC Time	4 sec 290 ms

GC Pause Duration Time Range :

Duration (ms) 1,00(ms Change	No. of GCs	Percentage
0 - 1,000	1862	96.78%
1,000 - 2,000	50	2.6%
2,000 - 3,000	11	0.57%
4,000 - 5,000	1	0.05%

Performance Problems

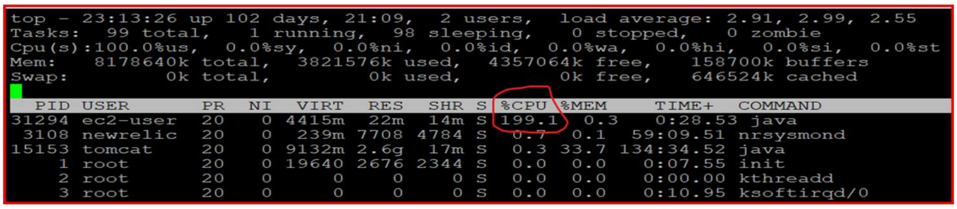
CPU Spike

Why

- Blocked thread(s)

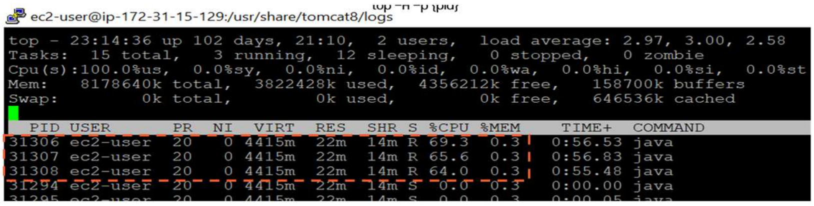
How to Solve

1) Using *top* tool, confirm that there is CPU spike. Command: *# top -H -p <pid>*



```
top - 23:13:26 up 102 days, 21:09,  2 users,  load average: 2.91, 2.99, 2.55
Tasks:  99 total,   1 running,  98 sleeping,   0 stopped,   0 zombie
Cpu(s):100.0%us,  0.0%sy,  0.0%ni,  0.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Mem:   8178640k total,  3821576k used,  4357064k free,   158700k buffers
Swap:        0k total,        0k used,        0k free,   646524k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
31294 ec2-user  20   0 4415m  22m  14m S 199.1  0.3   0:28.53 java
 3108 newrelic  20   0  239m 7708 4784 S  0.7  0.1  59:09.51 nrsysmond
15153 tomcat    20   0 9132m 2.6g  17m S  0.3 33.7 134:34.52 java
    1 root      20   0 19640 2676 2344 S  0.0  0.0   0:07.55 init
    2 root      20   0     0    0    0 S  0.0  0.0   0:00.00 kthreadd
    3 root      20   0     0    0    0 S  0.0  0.0   0:10.95 ksoftirqd/0
```

2) Identify which threads caused the spike.



top -H -p {pid}

ec2-user@ip-172-31-15-129:/usr/share/tomcat8/logs

```
top - 23:14:36 up 102 days, 21:10,  2 users,  load average: 2.97, 3.00, 2.58
Tasks:  15 total,   3 running,  12 sleeping,   0 stopped,   0 zombie
Cpu(s):100.0%us,  0.0%sy,  0.0%ni,  0.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Mem:   8178640k total,  3822428k used,  4356212k free,   158700k buffers
Swap:        0k total,        0k used,        0k free,   646536k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
31306 ec2-user  20   0 4415m  22m  14m R 69.3  0.3   0:56.53 java
31307 ec2-user  20   0 4415m  22m  14m R 65.6  0.3   0:56.83 java
31308 ec2-user  20   0 4415m  22m  14m R 64.0  0.3   0:55.48 java
31294 ec2-user  20   0 4415m  22m  14m S  0.0  0.3   0:00.00 java
31295 ec2-user  20   0 4415m  22m  14m S  0.0  0.3   0:00.05 java
```

3) Collect thread dump and lookup these threads: JMC, jcmd, JVisualVM
4) Identify and resolve lines of code that causes the blocking from stack trace of the identified threads. This can be also done in thread analysis tools: JVisualVM, FastThread.

OutOfMemoryError

Why

Not enough certain type of memory specified in the type of the OOME - specified in *java.lang.OutOfMemoryError: <type>*, where *type* can be:

- Java heap space
- GC overhead limit exceeded
- Requested array size exceed VM limit
- Permgen space
- Metaspace
- Unable to create new native thread
- Kill process or sacrifice child
- Reason stack_trace_with_native method

Memory leak can be the main cause.

How to Solve

1) Capture heap dumps: jmap, jcmd
2) Analyse to look for root cause: HeapHero, JVisualVM

See Heap Dump section for more details.

StackOverflowError

Why

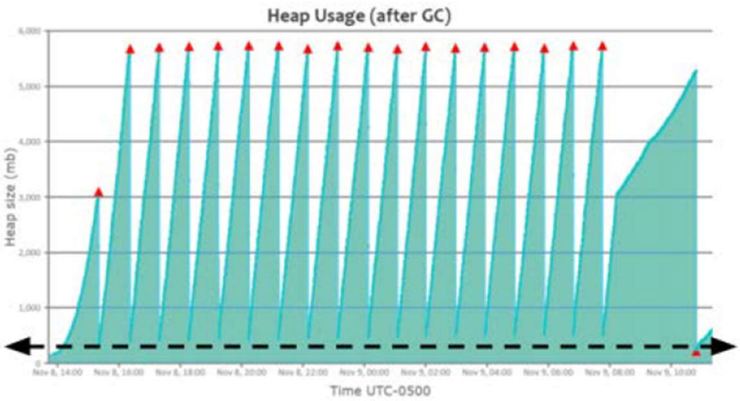
Not enough stack size configured.

How to Solve

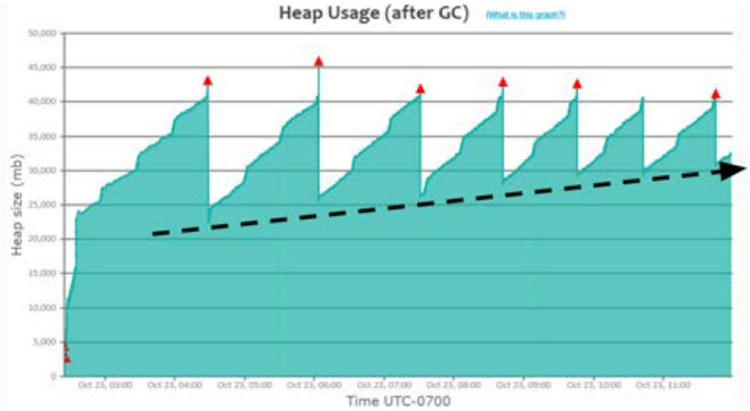
Adjust stack size using JVM argument -Xss.

GC Patterns

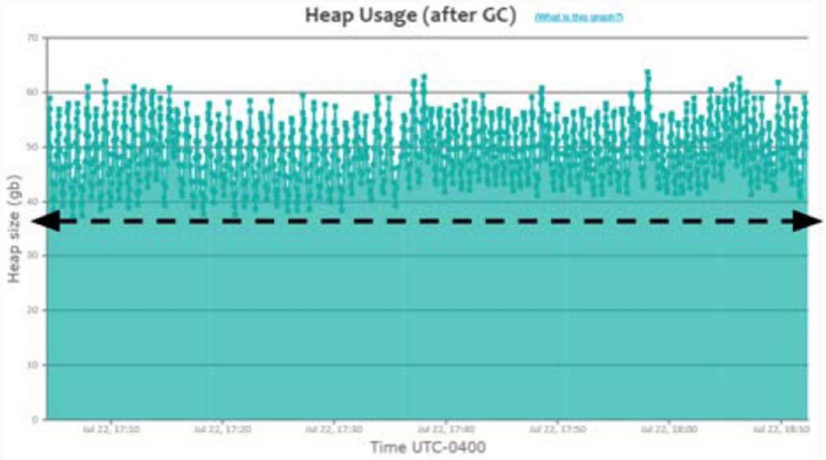
Healthy GC (sawtooth)



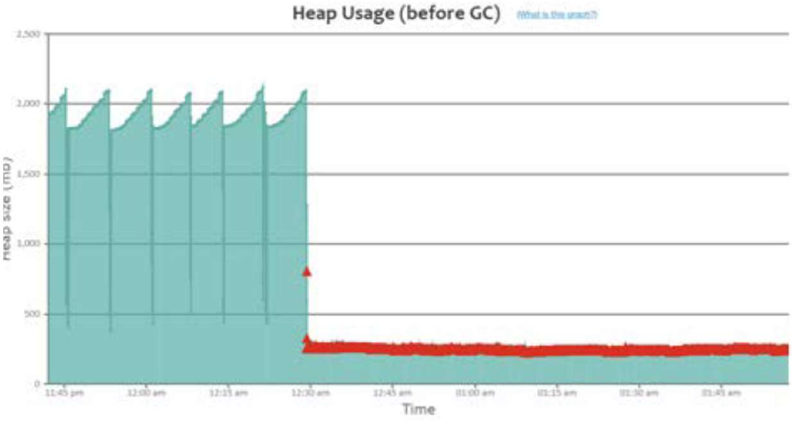
Acute Memory Leak (uptrend)



Heavy Caching



Metaspace memory problem



Useful JVM Arguments for Optimisation

Heap

Use one of the followings to specify your application heap size:

JVM Arguments	Remarks
-Xmx	Supported in all versions of JDK. Example: # java -Xmx512m -XshowSettings:vm -version
-XX:MaxRAMFraction -XX:MinRAMFraction	Only available in JDK8u131-190. Recommended for containers. Should be used in conjunction with: -XX:+UnlockExperimentalVMOptions -XX:+UseCGroupMemoryLimitForHeap Example: # docker run -m 1GB openjdk:8u131 java -XX:+UnlockExperimentalVMOptions -XX:+UseCGroupMemoryLimitForHeap -XX:MaxRAMFraction=2 -XshowSettings:vm -version
-XX:MaxRAMPercentage -XX:MinRAMPercentage	Available in JDK8u191 and above. Recommended for app deployed in containers. Example: # docker run -m 1GB openjdk:10 java -XX:MaxRAMPercentage=50 -XshowSettings:vm -version

Metaspace

- ⇨ Region where class definitions, method definitions, and other JVM metadata are stored.

JVM Arguments	Remarks
-Xx:MaxMetaspaceSize	Example: -XX:MaxMetaspaceSize=256m

Stack:

JVM Arguments	Remarks
-Xss	Specifies stack size. Adjust with knowledge of total number of thread as value is per thread. Example: -Xss256k

GC selection

Use one of the followings:

JVM Arguments	Remarks
-XX:+UseSerialGC	To use Serial GC algorithm.
-XX:+UseParallelGC	To use Parallel GC algorithm.
-XX:+UseConcMarkSweepGC	To use CMS GC algorithm.
-XX:+ UseG1GC	To use G1GC GC algorithm.
-XX:+ UseShenandoahGC	To use Shenandoah GC algorithm.
-XX:+ UseZGC	To use Z GC algorithm. Starting JDK23: To use Z GC Generational, add: -XX:+ZGenerational Or, to use non-generational, add: -XX:-ZGenerational

Timeouts

Use any of the followings if needed:

JVM Arguments	Remarks
-Dsun.net.client.defaultConnectTimeout	Timeout to connect to host. Example: -Dsun.net.client.defaultConnectTimeout=2000
-Dsun.net.client.defaultReadTimeout	Timeout when reading from input stream. Example: -Dsun.net.client.defaultReadTimeout=2000

Useful JVM Arguments for Troubleshooting

GC log:

- Analyse GC log for period of 24 hours during weekdays for high and low traffic monitoring.
- Can be used to troubleshoot GC-related problems: long GC pauses, irresponsive application, low throughput, memory leak indication GC pattern

JVM Arguments	**Remarks**
-verbose:gc -Xloggc:<log_file_path> -XX:+PrintGCDetails -XX:+PrintGCDateStamps	For Java 7 and below Example: *java -verbose:gc -Xloggc:/var/log/myapp/gc.log -XX:+PrintGCDetails -XX:+PrintGCDateStamps -jar myapp.jar*
-XX:+PrintGC -XX:+PrintGCDetails -XX:+PrintGCDateStamps -Xloggc:<log_file_path> -XX:+UseGCLogFileRotation -XX:NumberOfGCLogFiles=<number_of_files> -XX:GCLogFileSize=<size>[k|m|g]	For Java 8: it has additional log rotation option. Example: *java -XX:+PrintGC –XX:+PrintGCDetails -XX:+PrintGCDateStamps -Xloggc:/var/log/myapp/gc.log -XX:+UseGCLogFileRotation -XX:NumberOfGCLogFiles=5 -XX:GCLogFileSize=10m -jar myapp.jar*

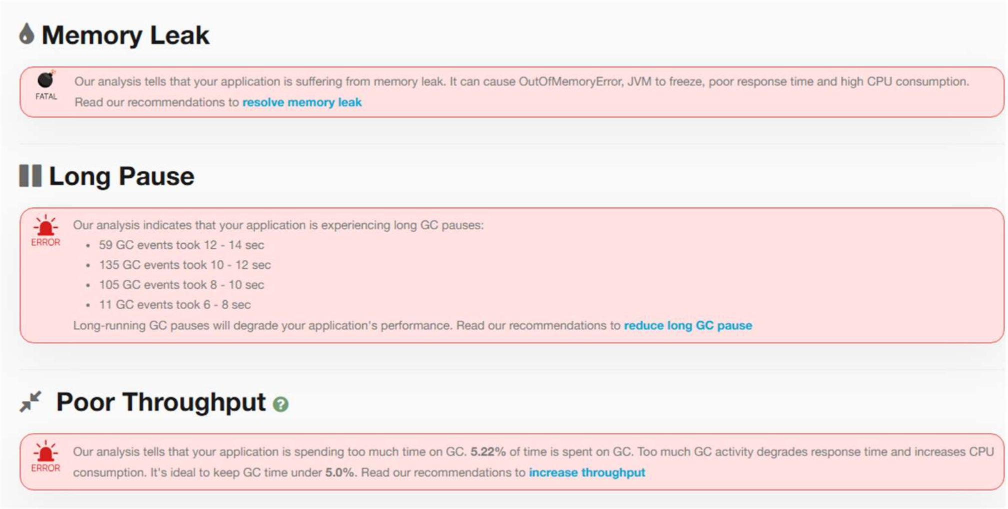
-Xlog:gc*:
file=<log_file_path>:
time,uptime,level,tags:
filecount=<number_of_files>,
filesize=<size>[k|m|g]
	For Java 9 and above: it is using unified logging -Xlog
	
	Example:
	java -Xlog:gc: file=/var/log/myapp/gc.log: time,uptime,level,tags: filecount=5, filesize=10m -jar myapp.jar*

Tools

To analyse GC log, we can use:

- GCEasy by yCrash
- JDK Mission Control
- JVisualVM
- IBM Health Centre and/or IBM GC and Memory Visualizer
- Garbage Cat

With GCEasy, summary and recommendations are provided like shown below:



Memory Leak

FATAL Our analysis tells that your application is suffering from memory leak. It can cause OutOfMemoryError, JVM to freeze, poor response time and high CPU consumption. Read our recommendations to **resolve memory leak**

Long Pause

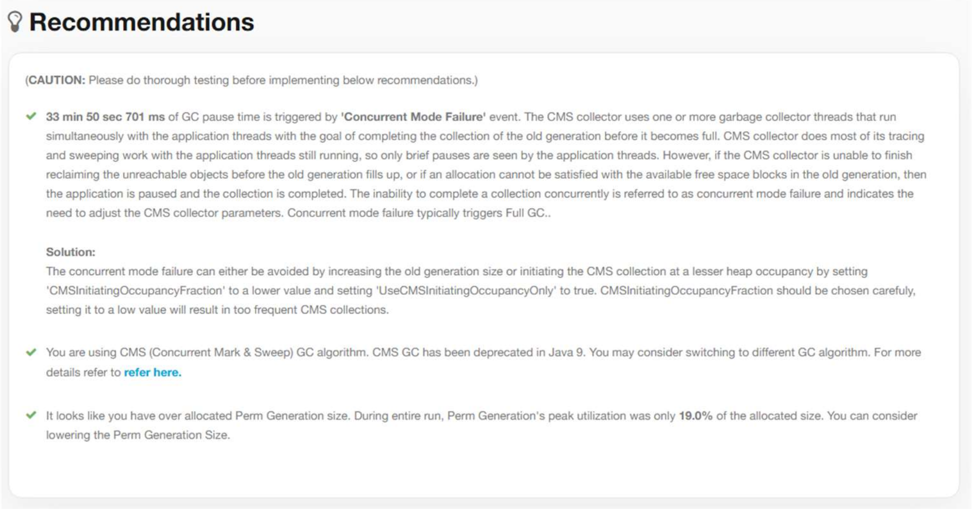
ERROR Our analysis indicates that your application is experiencing long GC pauses:

- 59 GC events took 12 - 14 sec
- 135 GC events took 10 - 12 sec
- 105 GC events took 8 - 10 sec
- 11 GC events took 6 - 8 sec

Long-running GC pauses will degrade your application's performance. Read our recommendations to **reduce long GC pause**

Poor Throughput

ERROR Our analysis tells that your application is spending too much time on GC. **5.22%** of time is spent on GC. Too much GC activity degrades response time and increases CPU consumption. It's ideal to keep GC time under **5.0%**. Read our recommendations to **increase throughput**



Recommendations

(**CAUTION:** Please do thorough testing before implementing below recommendations.)

✓ **33 min 50 sec 701 ms** of GC pause time is triggered by **'Concurrent Mode Failure'** event. The CMS collector uses one or more garbage collector threads that run simultaneously with the application threads with the goal of completing the collection of the old generation before it becomes full. CMS collector does most of its tracing and sweeping work with the application threads still running, so only brief pauses are seen by the application threads. However, if the CMS collector is unable to finish reclaiming the unreachable objects before the old generation fills up, or if an allocation cannot be satisfied with the available free space blocks in the old generation, then the application is paused and the collection is completed. The inability to complete a collection concurrently is referred to as concurrent mode failure and indicates the need to adjust the CMS collector parameters. Concurrent mode failure typically triggers Full GC..

Solution:
The concurrent mode failure can either be avoided by increasing the old generation size or initiating the CMS collection at a lesser heap occupancy by setting 'CMSInitiatingOccupancyFraction' to a lower value and setting 'UseCMSInitiatingOccupancyOnly' to true. CMSInitiatingOccupancyFraction should be chosen carefuly, setting it to a low value will result in too frequent CMS collections.

✓ You are using CMS (Concurrent Mark & Sweep) GC algorithm. CMS GC has been deprecated in Java 9. You may consider switching to different GC algorithm. For more details refer to **refer here.**

✓ It looks like you have over allocated Perm Generation size. During entire run, Perm Generation's peak utilization was only **19.0%** of the allocated size. You can consider lowering the Perm Generation Size.

Heap Dump

- Can be used to troubleshoot memory-related problems: slow memory leaks, GC problems, OutOfMemoryError

JVM Arguments	**Remarks**
-XX:+HeapDumpOnOutOfMemoryError -XX:HeapDumpPath=<file_path>	Example: -XX:+HeapDumpOnOutOfMemoryError -XX:HeapDumpPath=/opt/tmp/heapdump.hprof

Tools

1 - HeapHero

2 – jcmd.exe: available with JDK

- ⇨ $ jcmd <pid> GC.heap_dump <file_path>. E.g.: $ jcmd 37320 GC.heap_dump /opt/tmp/heapdump.bin

3 – JVisualVM

Thread Dump

- It's a snapshot of all the threads running in a Java process.
- Can be used to troubleshoot: CPU spikes, unresponsiveness, poor response time, hung threads, high memory consumption.
- No noticeable overhead in capturing thread dumps on every 5 minutes or 2 minutes interval.

JVM Arguments	**Remarks**
-	

Tools

1 – jstack.exe: available with JDK

⇨ $ jstack -l <pid> > <file_path>. E.g.: jstack -l 37320 > /opt/tmp/threadDump.txt

2 – kill-3 <pid>: available with JRE

⇨ $ kill -3 <pid>. E.g.: $ kill -3 37320

3 – jcmd.exe: available with JDK

⇨ $ jcmd <pid> Thread.print > <file_path>. E.g.: $ jcmd 37320 Thread.print > /opt/tmp/threadDump.txt

4 - JVisualVM

Useful JVM Arguments as Reactive Actions

When OutOfMemoryError

Use any of the followings:

JVM Arguments	**Remarks**
-XX:+HeapDumpOnOutOfMemoryError -XX:HeapDumpPath={heap-dump-file-path}	See Heap Dump section for more details.
-XX:OnOutOfMemoryError=<script_path>	Execute a script when OOME occurs. Example: -XX:OnOutOfMemoryError=/scripts/restart-myapp.sh
-XX:+CrashOnOutOfMemoryError	JVM exits when OOME occurs. Text and binary files are produced before exit. Not recommended.
-XX:+ExitOnOutOfMemoryError	Like CrashOnOutOfMemoryError but without text and binary files. Not recommended.

Recommended Practice for Optimum Performance

Set Max Heap Size and Metaspace Size accordingly

Heap size and Metaspace size plays a role in determining the frequency of GC events for your application.

Setup

JVM Arguments	Remarks
-Xmx	Example: Setting heap size to 2GB: -Xmx2g
-XX:MaxMetaspaceSize	Example: Setting Metaspace size to 256MB: -XX:MaxMetaspaceSize=256m

Use ZGC for Java 11+ Application

ZGC is known for its sub-millisecond pauses, allowing latency-sensitive systems to thrive.

Setup

JVM Arguments	Remarks
-XX:+UseZGC	

Always Enable GC Logging

GC Logging has (close to) no-impact to application performance but very useful for troubleshooting purposes. See GC log section for more details.

Setup

JVM Arguments	Remarks
-XX:+PrintGCDetails -XX:+PrintGCDateStamps -Xloggc: <file_path>	Up to JDK 8 Example: -XX:+PrintGCDetails -XX:+PrintGCDateStamps -Xloggc:/opt/workspace/myAppgc.log
-Xlog:gc*:file=<file_path>	JDK 8 + Example: -Xlog:gc*:file=/opt/workspace/myAppgc.log

Create Heap Dump on Out of Memory Error

Heap dump is very useful in troubleshooting OutOfMemoryError in application when it happens.

Setup

JVM Arguments	Remarks
-XX:+HeapDumpOnOutOfMemoryError -XX:HeapDumpPath=<file_path>	Example: -XX:+HeapDumpOnOutOfMemoryError -XX:HeapDumpPath=/dmp/my-heap-dump.hprof

Increase Stack Memory Only When Needed

Each thread will have its own stack. When not enough memory in stack, StackOverflowError is thrown.

Setup

JVM Arguments	Remarks
-Xss	Example: Setting stack size to 256KB: -Xss256k

Set timeout for connection

This is to avoid unresponsiveness in your application caused by remote applications and safeguard your applications high availability.

Setup

JVM Arguments	Remarks
-Dsun.net.client.defaultConnectTimeout -Dsun.net.client.defaultReadTimeout	Example: -Dsun.net.client.defaultConnectTimeout=2000 -Dsun.net.client.defaultReadTimeout=2000

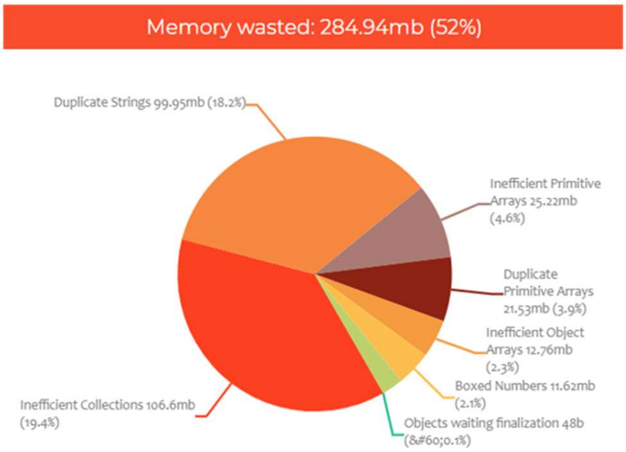
Set Time Zone for Your Application

This is particularly useful for sensitive business requirements in an application running in a distributed environment.

Setup

JVM Arguments	Remarks
-Duser.timezone	Example: -Duser.timezone="Asia/Kolkata"

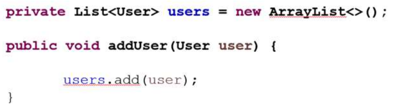
Stop Wasting Memory in Your Code



Several ways to optimise object creation and management:

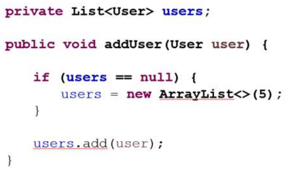
Collections:

1) Lazy initialisation



```
private List<User> users = new ArrayList<>();

public void addUser(User user) {

        users.add(user);
}
```

✗



```
private List<User> users;

public void addUser(User user) {

    if (users == null) {
        users = new ArrayList<>(5);
    }

    users.add(user);
}
```

✓

2) Specify capacity:

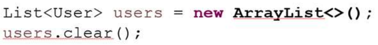
```
new ArrayList<>();
```

✗

```
new ArrayList<>(3);
```

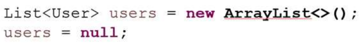
✓

3) Null instead of clear():



```
List<User> users = new ArrayList<>();
users.clear();
```

✗

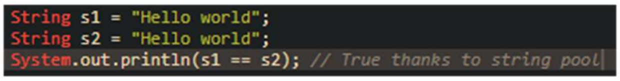


```
List<User> users = new ArrayList<>();
users = null;
```

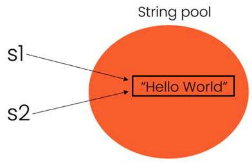
✓

Strings:

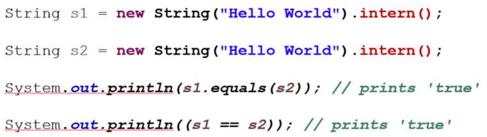
1) Use string literals



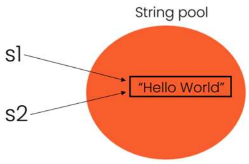
```
String s1 = "Hello world";
String s2 = "Hello world";
System.out.println(s1 == s2); // True thanks to string pool
```



2) Use intern()

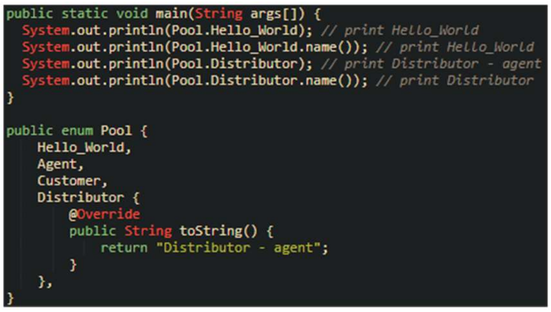


```
String s1 = new String("Hello World").intern();

String s2 = new String("Hello World").intern();

System.out.println(s1.equals(s2)); // prints 'true'

System.out.println((s1 == s2)); // prints 'true'
```



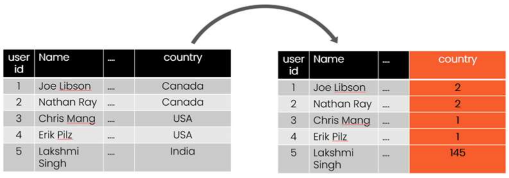
3) Alternative:

   a. Use Enum



```
public static void main(String args[]) {
  System.out.println(Pool.Hello_World); // print Hello_World
  System.out.println(Pool.Hello_World.name()); // print Hello_World
  System.out.println(Pool.Distributor); // print Distributor - agent
  System.out.println(Pool.Distributor.name()); // print Distributor
}

public enum Pool {
    Hello_World,
    Agent,
    Customer,
    Distributor {
        @Override
        public String toString() {
            return "Distributor - agent";
        }
    },
}
```

   b. In db, consider storing data as primitive types



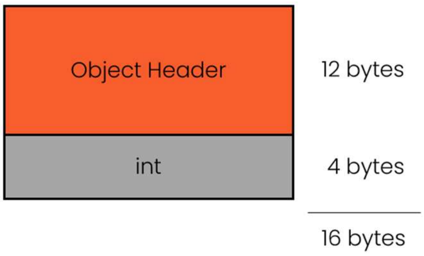
user id	Name	....	country
1	Joe Libson	....	Canada
2	Nathan Ray	....	Canada
3	Chris Mang	....	USA
4	Erik Pilz	....	USA
5	Lakshmi Singh	....	India

user id	Name	....	country
1	Joe Libson	....	2
2	Nathan Ray	....	2
3	Chris Mang	....	1
4	Erik Pilz	....	1
5	Lakshmi Singh	....	145

4) G1GC only: -XX:+UseDeduplication

Objects: use primitive types as much as possible and avoid boxed object as each object incurs overhead of 12bytes for header:

java.lang.Integer



Heap Monitoring Tools

See heap dump tools section for details.