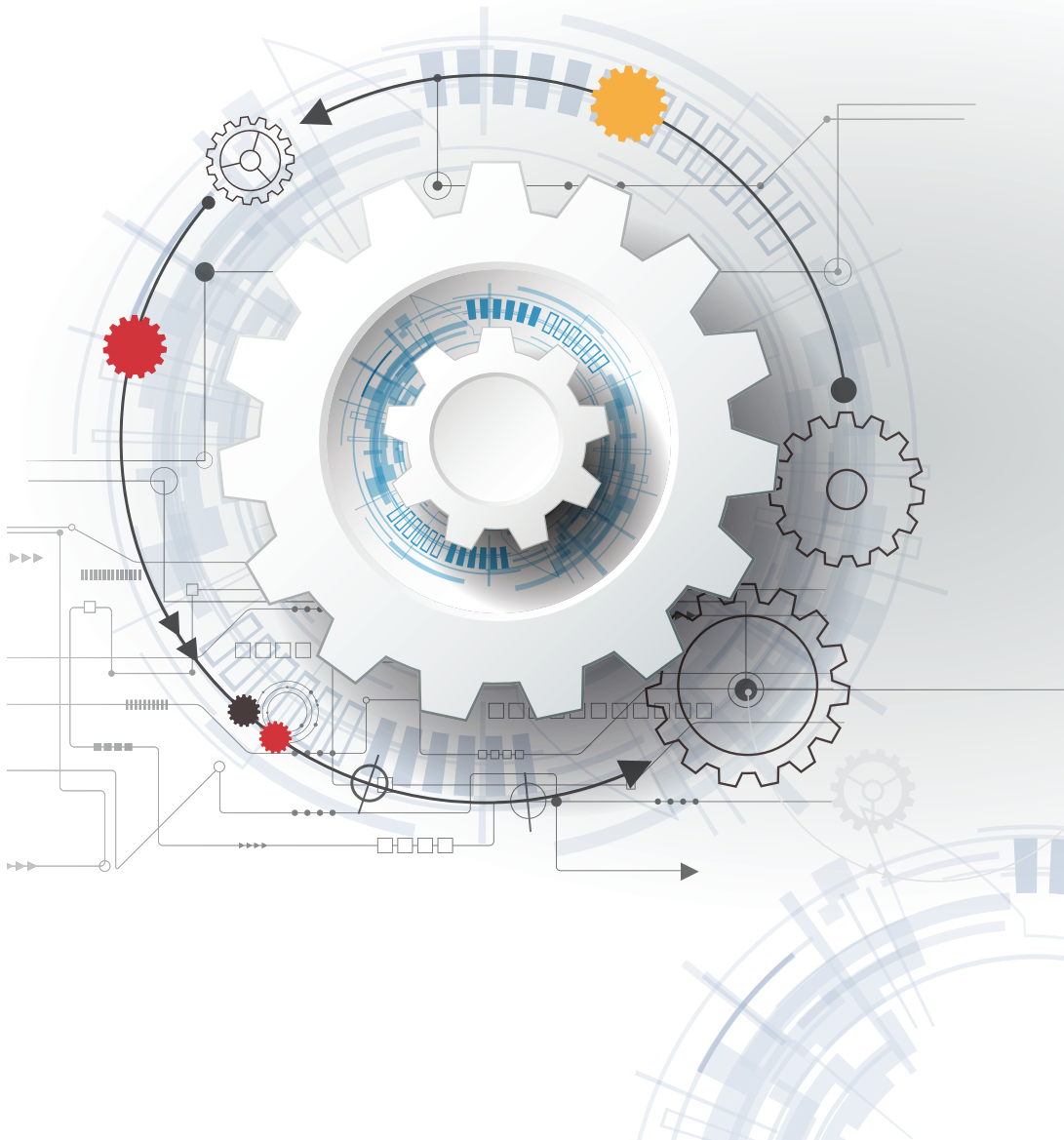


TIER1APP

# JVM Performance Engineering and Troubleshooting



COMPILATION OF RAM LAKSHMANAN'S BLOGS & ARTICLES

# TABLE OF CONTENTS

---

## Garbage Collection 01-26

---

1 - Why garbage collection might be more important than you think ?	01
2 - What is garbage collection log? How to analyze and enable?	03
3 - garbage collection log analysis API	05
4 - Memory tuning: Key performance Indicators	09
5 - 3 popular myths about garbage collection	11
6 - Sys time greater than user time	13
7 - Real time is greater than user and sys time	14
8 - Eliminate consecutive Full GCs	16
9 - Rotating GC Log Files	19
10 - Reduce long GC Pauses	22
11 - Which GC to use?	26

## Threads 27-64

---

12 - How to take thread dumps? 8 options	27
13 - Thread dump analysis API	33
14 - How to identify critical code path ?	36
15 - What's the difference between Blocked, Waiting and Timed_Waiting?	39
16 - Thread dump pattern – athlete	42
17 - Deadlock	44
18 - Thread dump analysis pattern – treadmill	46
19 - Thread dump analysis pattern - ATHEROSCLEROSIS	48
20 - Thread dump analysis pattern – traffic jam	50
21 - Thread dump analysis pattern - repetitive strain injury (RSI)	52
22 - Thread dump analysis pattern - Additives	54
23 - Thread dump analysis pattern - leprechaun trap	55

24 - StackOverFlowError: Causes & Solutions	57
25 - How to troubleshoot CPU problems?	61

## Memory 65-101

---

26 - Heap dump analysis API	65
27 - How to capture Heap Dump from android App ? - 3 options	68
28 - Disappointing story on Memory Optimization	72
29 - How to diagnose OutofMemory in android?	75
30 - How to diagnose memory leaks?	78
31 - What happened behind the scene – finalize() method?	80
32 - Troubleshoot OutOfMemory Error: Unable to create new native thread	84
33 - Eclipse Mat - Tidbits	89
34 - Shallow Heap, Retained Heap	93
35 - OutofMemoryError	97
36 - Virtual Machine Error	99

## General 102-104

---

37 - Remote Debugging Java Application	102
--	-----

# WHY GARBAGE COLLECTION MIGHT BE MORE IMPORTANT THAN YOU THINK?



I have heard from few of my developer friends saying: “Garbage Collection is Automatic. So, I don’t have to worry about it.”

First part is true, i.e. “Garbage Collection is Automatic” on all modern platforms – JVM (Java Virtual Machine), ART (Android Run Time)...

But the second part may not be as true as you think, i.e. “I don’t have to worry about it.” Garbage Collection is automatic, but it’s not free. It comes with a price.

**In fact, the price can be *\*very expensive\**.**

Poor Garbage Collection can lead to:

- Unpleasant user experience (SLA Breaches)
- Increase the bill from cloud hosting providers
- Puts entire Application Availability under risk



## 1. Unpleasant user experience (SLA Breaches)

To garbage collect objects automatically, entire application must be paused intermittently to mark the objects that are in use and sweep away the objects that are not used. During this pause period, all customer transactions which are in motion will be stalled (i.e. frozen). Depending on the type of GC algorithm and memory settings that you configure, pause times can run from few milliseconds to few seconds to few minutes. Thus, Garbage Collection can affect your application SLA (Service Level Agreement) significantly. Frequent pauses in the mobile application can jank the app (i.e. stuttering, juddering, or halting). It can leave a very unpleasant experience for your user.



## 2. Increase the bill from cloud hosting providers

Garbage collection consumes a lot of CPU cycles. Each application will have thousands/millions of objects sitting in memory. Each object in memory should be investigated periodically to see whether they are in use? If it's in use, who is referencing it? Whether those references are still active? If they are not in use, they should be evicted from memory. All these investigations and computation requires a considerable amount of CPU power.

Most applications saturate memory first before saturating other resources (CPU, network bandwidth, storage). Most applications upgrade their EC2 instance size to get additional memory rather get additional CPU or network bandwidth. More object creation translates to more frequent Garbage Collection. Thus, you will end up buying more compute power. It will increase the bill from your cloud hosting providers.

## 3. Puts entire Application Availability under risk

Sometimes garbage collection events pause the application for several seconds to several minutes. Sometimes garbage collection events might run repeatedly. When Garbage Collection runs repeatedly, no customers transactions will be processed. When Garbage Collection runs repeatedly, to recover from the situation, the application has to be recycled. Such events can put the availability of your applications under risk.

Thus, to achieve 'Wow' user experience, reduce bills from hosting providers and increase your application's availability, one would have to study and optimize the Memory/Garbage Collection settings. Tools such as GCeasy.io, HP Jmeter can help you to study and optimize the Memory/Garbage collection settings.

# WHAT IS GARBAGE COLLECTION LOG? HOW TO ENABLE & ANALYZE?



Garbage collection has more profound impact on the application in contrary to what most engineers think. In order to optimize memory and garbage collection settings and to troubleshoot memory-related problems, one has to analyze Garbage Collection logs.

## Enabling GC logs

GC Logging can be enabled by passing below-mentioned system properties during application startup

### Until Java 8:

Below is the system property that is supported by all version of Java until JDK 8.

```
-XX:+PrintGCDetails -Xloggc:<gc-log-file-path>  
Example:  
-XX:+PrintGCDetails -Xloggc:/opt/tmp/myapp-gc.log
```

### From Java 9:

Below is the system property that is supported by all version of Java starting from JDK 9.

```
-Xlog:gc*:file=<gc-log-file-path>  
Example:  
-Xlog:gc*:file=/opt/tmp/myapp-gc.log
```

## How to analyze GC logs?

GC log has rich information, however, understanding GC log is not easy. There isn't sufficient documentation to explain GC log format. On top of it, GC log format is not standardized. It varies by JVM vendor (Oracle, IBM, HP, Azul, ...), Java version (1.4, 5, 6, 7, 8, 9), GC algorithm (Serial, Parallel, CMS, G1, Shenandoah), GC system properties that you pass (-XX:+PrintGC, -XX:+PrintGCDetails, -XX:+PrintGCDateStamps, -XX:+PrintHeapAtGC ...). Based on this permutation and combination, there are easily 60+ different GC log formats.

```

bash-3.2$ java -Xlog:gc*,gc*phases=debug GCTest [0.115s][info][gc,heap] Heap region size: 1M
[0.227s][info][gc] Using G1
[0.227s][info][gc,heap,coops] Heap address: 0xffffffff50400000, size: 4064 MB, Compressed Oops mode: Non-zero based: 0xffffffff50000000, Oop shift amount: 3
[2.448s][info][gc,start] GC(0) Pause Young (G1 Evacuation Pause)
[2.448s][info][gc,task] GC(0) Using 23 workers of 23 for evacuation
[2.796s][info][gc,phases] GC(0) Pre Evacuate Collection Set: 0.2ms
[2.797s][debug][gc,phases] GC(0) Choose Collection Set: 0.0ms
[2.797s][debug][gc,phases] GC(0) Humongous Register: 0.1ms
[2.797s][info][gc,phases] GC(0) Evacuate Collection Set: 341.6ms
[2.797s][debug][gc,phases] GC(0) Ext Root Scanning (ms): Min: 0.0, Avg: 9.2, Max: 205.8, Diff: 205.8, Sum: 212.0, Workers: 23
[2.797s][debug][gc,phases] GC(0) Update RS (ms): Min: 0.0, Avg: 0.0, Max: 0.0, Diff: 0.0, Sum: 0.1, Workers: 23
[2.797s][debug][gc,phases] GC(0) Processed Buffers: Min: 0, Avg: 0.0, Max: 0, Diff: 0, Sum: 0, Workers: 23
[2.797s][debug][gc,phases] GC(0) Scan RS (ms): Min: 0.0, Avg: 0.0, Max: 0.0, Diff: 0.0, Sum: 0.2, Workers: 23
[2.797s][debug][gc,phases] GC(0) Code Root Scanning (ms): Min: 0.0, Avg: 0.8, Max: 4.6, Diff: 4.6, Sum: 17.7, Workers: 23
[2.797s][debug][gc,phases] GC(0) Object Copy (ms): Min: 135.2, Avg: 330.2, Max: 340.7, Diff: 205.6, Sum: 7594.1, Workers: 23
[2.797s][debug][gc,phases] GC(0) Termination (ms): Min: 0.0, Avg: 0.8, Max: 1.6, Diff: 1.5, Sum: 18.7, Workers: 23
[2.798s][debug][gc,phases] GC(0) Termination Attempts: Min: 1, Avg: 4.0, Max: 8, Diff: 7, Sum: 91, Workers: 23
[2.798s][debug][gc,phases] GC(0) GC Worker Other (ms): Min: 0.0, Avg: 0.1, Max: 0.2, Diff: 0.1, Sum: 1.2, Workers: 23
[2.798s][debug][gc,phases] GC(0) GC Worker Total (ms): Min: 340.6, Avg: 341.0, Max: 341.5, Diff: 0.8, Sum: 7844.1, Workers: 23
[2.798s][info][gc,phases] GC(0) Post Evacuate Collection Set: 4.7ms
[2.798s][debug][gc,phases] GC(0) Code Roots Fixup: 0.0ms
[2.798s][debug][gc,phases] GC(0) Preserve CR Refs: 0.0ms
[2.798s][debug][gc,phases] GC(0) Reference Processing: 3.1ms
[2.798s][debug][gc,phases] GC(0) Clear Card Table: 0.1ms
[2.798s][debug][gc,phases] GC(0) Reference Enqueuing: 0.1ms
[2.798s][debug][gc,phases] GC(0) Merge Per-Thread State: 0.2ms
[2.798s][debug][gc,phases] GC(0) Code Roots Purge: 0.1ms
[2.798s][debug][gc,phases] GC(0) Redirty Cards: 0.3ms
[2.798s][debug][gc,phases] GC(0) Free Collection Set: 0.6ms
[2.798s][debug][gc,phases] GC(0) Humongous Reclaim: 0.1ms
[2.799s][debug][gc,phases] GC(0) Expand Heap After Collection: 0.0ms
[2.799s][info][gc,phases] GC(0) Other: 1.9ms
[2.799s][info][gc,heap] GC(0) Eden regions: 24->0(9)
[2.799s][info][gc,heap] GC(0) Survivor regions: 0->3(3)
[2.799s][info][gc,heap] GC(0) Old regions: 0->2
[2.799s][info][gc,heap] GC(0) Humongous regions: 2->1
[2.799s][info][gc,metaspace] GC(0) Metaspace: 4719K->4719K(1056768K)
[2.799s][info][gc] GC(0) Pause Young (G1 Evacuation Pause) 26M->5M(256M) 350.839ms
[2.799s][info][gc,cpu] GC(0) User=0.61s Sys=5.00s Real=0.35s
[~]

```

Thus, to analyze GC logs, it's highly recommended to use GC log analysis tools such as GCeasy, HPJmeter. These tools parse GC logs and generate great graphical visualizations of data, reports Key Performance Indicators and several other useful metrics.

## 3

# GARBAGE COLLECTION LOG ANALYSIS API

In this modern world, Garbage collection logs are still analyzed in a tedious & manual mode. i.e. you have to get hold of your DevOps engineer who has access to production servers, then he will mail you the application's GC logs, then you will upload the logs to GC analysis tool, then you have to apply your intelligence to analyze it. There is no programmatic way to analyze Garbage Collection logs in a proactive manner. Thus to eliminate this hassle, gceasy.io is introducing a RESTful API to analyze garbage collection logs. With one line of code you can get your GC logs analyzed instantly.

**Here are few use cases where this API can be extremely useful.**

## Use case 1: Automatic Root cause Analysis

Most of the DevOps invokes a simple Http ping or APM tools to monitor the applications health. This ping is good to detect whether application is alive or not. APM tools are great at informing that application's CPU spiked up by 'x%', memory utilization increased by 'y%', response time dropped by 'z' milliseconds. It won't inform what caused the CPU to spike up, what caused memory utilization to increase, what caused the response time to degrade. If you can configure Cron job to capture thread dumps/GC logs on a periodic interval and invoke our REST API, we apply our intelligent patterns & machine learning algorithms to instantly identify the root cause of the problem.

### Advantage 1

Whenever these sort of production problem happens, because of the heat of the moment, DevOps team recycles the servers without capturing the thread dumps and GC logs. You need to capture thread dumps and GC logs at the moment when problem is happening, in order to diagnose the problem. In this new strategy you don't have to worry about it, because your cron job is capturing thread dumps/GC logs on a periodic intervals and invoking the REST API, all your thread dumps/GC Logs are archived in our servers.

### Advantage 2

Unlike APM tools which claims to add less than 3% of overhead, where as in reality it adds multiple folds, beauty of this strategy is: It doesn't add any overhead (or negligible overhead). Because entire analysis of the thread dumps/GC logs are done on our servers and not on your production servers..

## Use case 2: Performance Tests

When you conduct performance tests, you might want to take thread dumps/GC logs on a periodic basis and get it analyzed through the API. In case if thread count goes beyond a threshold or if too many threads are WAITING or if any threads are BLOCKED for a prolonged period or lock isn't getting released or frequent full GC activities happening or GC pause time exceeds thresholds, it needs to get the visibility right then and there. It should be analyzed before code hits the production. In such circumstance this API will become very handy.

## Use case 3: Continuous Integration

When you conduct performance tests, you might want to take thread dumps/GC logs on a periodic basis and get it analyzed through the API. In case if thread count goes beyond a threshold or if too many threads are WAITING or if any threads are BLOCKED for a prolonged period or lock isn't getting released or frequent full GC activities happening or GC pause time exceeds thresholds, it needs to get the visibility right then and there. It should be analyzed before code hits the production. In such circumstance this API will become very handy.

### HOW TO INVOKE GARBAGE COLLECTION LOG ANALYSIS API?

Register with us. We will email you the API key. This is a one-time setup process.

**Note:** If you have purchased enterprise version with API, you don't have to register. API key will be provided to you as part of installation instruction.

1

Post HTTP request to  
`http://api.gceasy.io/analyzeGC?apiKey={API_KEY_SENT_IN_EMAIL}`

2

The body of the HTTP request should contain the Garbage collection log that needs to be analyzed.

3

HTTP Response will be sent back in JSON format. JSON has several important stats about the GC log. Primary element to look in the JSON response is: "isProblem". This element will have value to be "true" if any memory/performance problems has been discovered. "problem" element will contain the detailed description of the memory problem.

4



## CURL command

Assuming your GC log file is located in “./my-app-gc.log,” then CURL command to invoke the API is:

```
curl -X POST --data-binary @./my-app-gc.log http://api.gceasy.io/analyzeGC?apiKey={API_KEY_SENT_IN_EMAIL} --header "Content-Type:text"
```

## It can't get any more simpler than that? Isn't it?

**Note:** use the option “--data-binary” in the CURL instead of using “-data” option. In “-data” new line breaks will be not preserved in the request. New Line breaks should be preserved for legitimate parsing.

## Other Tools

You can also invoke the API using any webservice client tools such as: SOAP UI, Postman Browser Plugin,.....



Fig: POSTing GC logs through PostMan plugin

```
{
  "isProblem": true,
  "problem": [
    "Our analysis tells that Full GCs are consecutively running in your application. It might cause intermittent OutOfMemoryErrors or degradation in response time or high CPU consumption or even make application unresponsive.",
  ],
  "jvmHeapSize": {
    "youngGen": {
      "allocatedSize": "7.5 gb",
      "peakSize": "6 gb"
    },
  },
}
```

```

"oldGen": {
  "allocatedSize": "22.5 gb",
  "peakSize": "22.5 gb"
},
"metaSpace": {
  "allocatedSize": "1.04 gb",
  "peakSize": "48.52 mb"
},
"total": {
  "allocatedSize": "30 gb",
  "peakSize": "28.5 gb"
}
},
"gcStatistics": {
  "totalCreatedBytes": "249.49 gb",
  "measurementDuration": "7 hrs 32 min 52 sec",
  "avgAllocationRate": "9.4 mb/sec",
  "avgPromotionRate": "1.35 mb/sec",
  "minorGCCount": "62",
  "minorGCTotalTime": "1 min 19 sec",
  "minorGCAvgTime": "1 sec 274 ms",
  "minorGCAvgTimeStdDeviation": "2 sec 374 ms",
  "minorGCMinTime": "0",
  "minorGCMaxTime": "13 sec 780 ms",
  "minorGCIntervalAvgTime": "7 min 25 sec 442 ms",
  "fullGCCount": "166",
  "fullGCTotalTime": "14 min 11 sec 620 ms",
  "fullGCAvgTime": "5 sec 130 ms",
  "fullGCAvgTimeStdDeviation": "5 sec 207 ms",
  "fullGCMinTime": "120 ms",
  "fullGCMaxTime": "57 sec 880 ms",
  "fullGCIntervalAvgTime": "2 min 19 sec 104 ms"
},
"gcKPI": {
  "throughputPercentage": 99.952,
  "averagePauseTime": 750.232,
  "maxPauseTime": 57880
},
"gcDurationSummary": {
  "groups": [
    {
      "start": "0",
      "end": "6",

```

# MEMORY TUNING: KEY PERFORMANCE INDICATORS

# 4

When you are tuning the application's memory & Garbage Collection settings, you should take well-informed decisions based on the key performance indicators. But there are overwhelming amount of metrics reported; which one to choose and which one to leave? This article intends to explain the right KPIs and right tools to source them.

## WHAT ARE THE RIGHT KPIS?

THROUGHPUT

LATENCY

FOOTPRINT

### Throughput

Throughput is the amount of productive work done by your application in a given time period. This brings the question what is productive work? what is non-productive work?

**Productive Work:** This is basically the amount of time your application spends in processing your customer's transactions.

**Non-Productive Work:** This is basically the amount of time your application spend in house-keeping work, primarily Garbage collection.

Let's say your application runs for 60 minutes. In this 60 minutes let's say 2 minutes is spent on GC activities.

It means application has spent 3.33% on GC activities (i.e.  $2 / 60 * 100$ )

It means application throughput is 96.67% (i.e.  $100 - 3.33$ ).

**Now the question is: What is the acceptable throughput %?**

**It depends on the application and business demands.**

**Typically one should target for more than 95% throughput.**

## Latency

This is the amount of time taken by one single Garbage collection event to run. This indicator should be studied from 3 fronts.

**1. Average GC time:** What is the average amount of time spent on GC?

**2. Maximum GC time:** What is the maximum amount of time spent on a single GC event? Your application may have service level agreements such as “no transaction can run beyond 10 seconds”. In such cases, your maximum GC pause time can't be running for 10 seconds. Because during GC pauses, entire JVM freezes – no customer transactions will be processed. So it's important to understand the maximum GC pause time.

**3. GC Time Distribution:** You should also understand how many GC events are completing within what time range (i.e. within 0 – 1 second, 200 GC events are completed, between 1 – 2 second 10 GC events are completed ...)

## Footprint

Footprint is basically the amount CPU consumed. Based on your GC algorithm, based on your memory settings, CPU consumption will vary. Some GC algorithms will consume more CPU (like Parallel, CMS), whereas other algorithms such as Serial will consume less CPU.

According to memory tuning Gurus, you can pick only 2 of them at a time.

If you want good throughput and latency, then footprint will degrade.

If you want good throughput and footprint, then latency will degrade.

If you want good latency and footprint, then throughput will degrade.

## Right Tools

Throughput and Latency can be obtained from analyzing Garbage collection Logs. Upload your application's Garbage Collection log file in <http://gceasy.io/> tool. This tool can parse Garbage Collection logs and generates Throughput and Latency indicators for you. Below is the screen shot from the <http://gceasy.io/> tool showing the throughput and latency:

### Key Performance Indicators

To learn more about KPIs, [click here](#)

**1 Throughput** : 99.85%

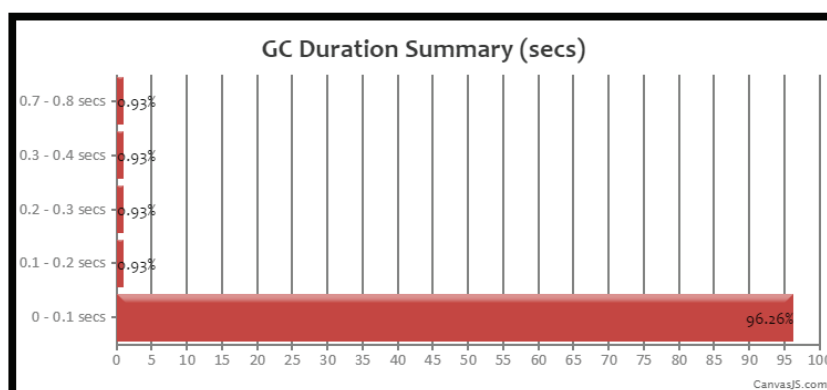
**2 Latency:**

Avg GC Time : 26 ms

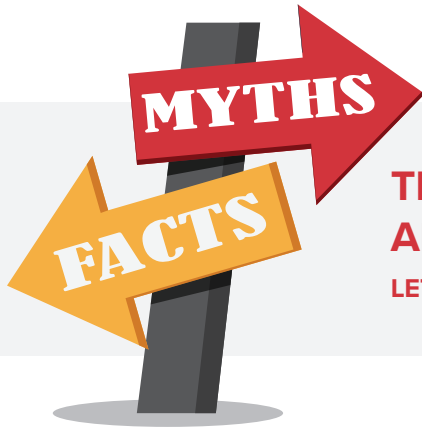
Max GC Time : 790 ms

GC Duration Time Range :

Duration (secs)	No. of GCs
0 - 0.1	103
0.1 - 0.2	1
0.2 - 0.3	1
0.3 - 0.4	1
0.7 - 0.8	1



# 3 POPULAR MYTHS ABOUT GARBAGE COLLECTION



**THERE ARE 3 HIGHLY POPULAR MYTHS ABOUT GARBAGE COLLECTION.**

**LET'S REVIEW THOSE MYTHS AND THE ACTUAL TRUTH BEHIND THEM.**

## MYTH #1: MINOR GC DON'T PAUSE THE APPLICATION

### MYTH

There are different types of Garbage Collection events: Minor GC event, Major GC event & Full GC event. It's been educated that Minor GC are harmless, as they don't pause the application. However Major/Full GC are dangerous because it pauses the application.

### TRUTH

This is a lie. 100% lie. Minor GC \*do pause\* the application. Minor GC pause times are comparatively lower than other GC events most of the times. Thus, it could have been educated as 'harmless'. However, in some cases, we have seen Minor GC take more time than all the Major/Full GC events. Thus, when tuning your application, pay proper attention towards Minor GC pause time Metrics as well.

## MYTH #2: SERIAL GC PERFORMANCE IS HORRIBLE

### MYTH

There are several types of Garbage Collection algorithms:

- Serial GC
- Parallel GC

### TRUTH

To validate this theory, we conducted a study on a major B2B travel application in production, which processes more than 70% of North America's leisure travel transactions.



- G1 GC
- Shenandoah GC

Each GC algorithm exhibits its unique performance characteristics. A false education industry has been making: ‘Serial GCs are not meant for serious applications.’ Serial GC performance characteristics are horrible. It should be used only during development time or in prototype applications.

We configured a couple of servers to use latest ‘G1 GC’ algorithm and couple of servers to use ‘Serial GC’ algorithm. We just used vanilla G1 GC and Vanilla Serial GC settings. Results turned out that Serial GC performance to be comparable (in fact slightly better than) to G1 GC algorithm. Of course, with proper tuning & parameters settings, G1 GC can be made to run better than Serial GC. The take away is, Serial GC is not as bad as it’s portrayed. We didn’t pass any additional GC tuning parameters.

Results turned out that Serial GC performance to be comparable (in fact slightly better than) to G1 GC algorithm. Of course, with proper tuning & parameters settings, G1 GC can be made to run better than Serial GC. The take away is, Serial GC is not as bad as it’s portrayed.

### MYTH #3: GARBAGE COLLECTION IS AUTOMATIC. I DON’T HAVE TO WORRY ABOUT IT.

#### MYTH

I have heard few developer friends saying: “Garbage Collection is Automatic. I don’t have to worry about it”.

#### TRUTH

First part is true, i.e. “Garbage Collection is Automatic” on all modern platforms – JVM (Java Virtual Machine), ART (Android Run Time)... But the second part is not so true, i.e. “I don’t have to worry about it”. Garbage Collection is automatic, but it’s not free. It comes with a price. In fact, the price can be \*very expensive\*. Poor Garbage Collection can lead to:

- Unpleasant user experience (SLA Breaches)
- Increase in the bill from cloud hosting providers
- Puts entire application availability under risk



# SYS TIME GREATER THAN USER TIME

Time taken by every single GC event is reported in the GC log. In every GC event, there are 'user', 'sys', and 'real' time. What does these time mean? What is the difference between them ?

- 'real' time is the total elapsed time of the GC event. This is basically the time that you see on the clock.
- 'user' time is the CPU time spent in user-mode code (outside the kernel)
- 'Sys' time is the amount of CPU time spent in the kernel. This means executing CPU time spent in system calls within the kernel, as opposed to library code, which is still running in user-space.

In normal (all most all) GC events, user time will be greater than sys time. It's because, in a GC event, most of the time is spent within the JVM code and only very less time is spent in the kernel. But in certain circumstances, you might see sys time to be greater than user time.

## Example

```
[Times: user=0.04 sys=0.35, real=0.42 secs]
```

Here you can notice the sys time to be 0.35 seconds which is considerably higher than user time 0.04 seconds.

If you observe multiple occurrences of this scenario in your GC log then it might be indicative of one of the following problems:

- 1. Operating system problem**
- 2. VM related problem**

### 1. Operating System problem

Operating System exceptions such as page faults, misaligned memory references, and floating-point exceptions consume large amount of system time. Make sure your OS is running with proper patches, upgrades, and sufficient CPU/memory/disk space .

### 2. VM related Problem

If your application is running in a virtualized environment, may be because of nature of the emulation sys time can be higher than user time. Make sure virtualized environment is NOT OVERLOADED with too many environments and also ensure that there are adequate resources available in the Virtual Machine for your application to run

# REAL TIME IS GREATER THAN USER AND SYS TIME



Time taken by every single GC event is reported in the GC log. In every GC event, there are 'user', 'sys', and 'real' time. What does these time mean? What is the difference between them ?

- 'real' time is the total elapsed time of the GC event. This is basically the time that you see in the clock.
- 'user' time is the CPU time spent in user-mode code (outside the kernel).
- 'Sys' time is the amount of CPU time spent in the kernel. This means executing CPU time spent in system calls within the kernel, as opposed to library code, which is still running in user-space.

In normal (all most all) GC events, real time will be less than user + sys time. It's because of multiple GC threads work concurrently to share the work load, thus real time will be less than user + sys time. Say user + sys time is 2 seconds. If 5 GC threads are concurrently working then real time should be somewhere in the neighbourhood of 400 milliseconds ( 2 seconds / 5 GC threads).

But in certain circumstances you might see real time to be greater than user + sys time.

## Example

```
[Times: user=0.20 sys=0.01, real=18.45 secs]
```

If you notice multiple occurrences of this scenario in your GC log then it might be indicative of one of the following problems:

- 1. Heavy I/O activity**
- 2. Lack of CPU**

### 1. Heavy I/O activity

When there is heavy I/O activity (i.e. networking/disk access/user interaction) on the server then real time tend to spike up to a great extent. As part of GC logging, your application makes a disk access. If there is a heavy I/O activity on the server then GC event might be stranded, causing spiked up real time.

Note: Even if your application is not causing the heavy I/O activity, the other process on the server may cause the heavy I/O activity leading to the high real time. Here is a brilliant article from LinkedIn engineers, explaining the GC problem they experienced because of high I/O activity.

You can monitor I/O activity on your server, using the sar (System Activity Report), in Unix.

## Example

```
sar -d -p 1
```

Above commands reports the reads/sec and writes/sec made to the device every 1 second. For more details on 'sar' command refer to this tutorial.

If you notice high I/O activity on your server then you can do one of the following to fix the problem:

- a. If high I/O activity is caused by your application, then optimize your application's I/O activity.
- b. Eliminate the processes which are causing high I/O activity on the server
- c. Move your application to a different server where I/O activity is less

## 2. Lack of CPU

If multiple processes are running on your server and if your application doesn't get enough CPU cycles to run, it will start to wait. When the application starts to wait then real time will be considerably higher than user + sys time.

You can use the commands like 'top' or monitoring tools (nagios, newRelic, AppDynamics...) to observe the CPU utilization on the server. If you notice CPU utilization to be very high and your process doesn't get enough cycles to run then you can do one of the following to fix the problem:

- a. Reduce the number of processes that is running on the server, so that your application gets fair chance to run
- b. Increase the CPU capacity – if you are in AWS cloud (or equivalent), move to a bigger instance type which has more CPU cores
- c. Move your application to a new server where there is adequate CPU capacity

# ELIMINATE CONSECUTIVE FULL GCs

# 8

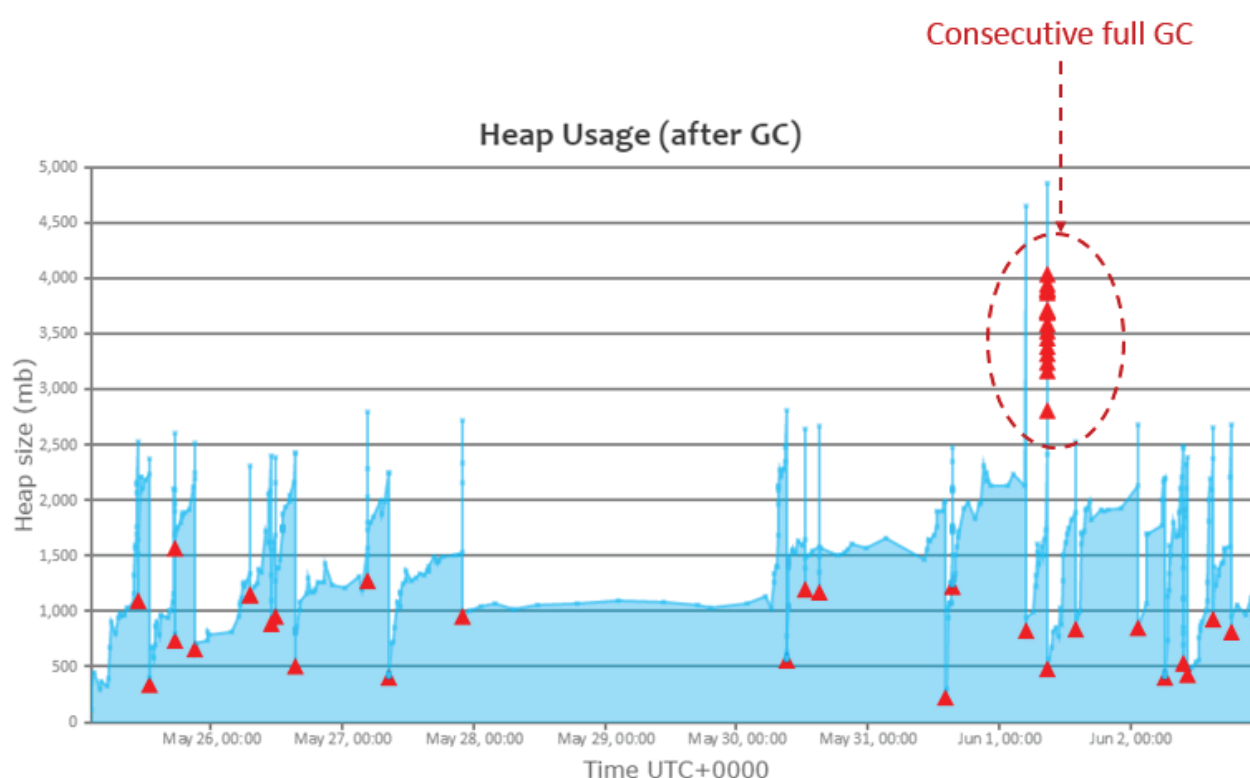
Full GC is an important event in the garbage collection process. During this full GC phase, garbage is collected from all the regions in the JVM heap (Young, Old, Perm, Metaspace). Full GC tends to evict more objects from memory, as it runs across all generations. A Full GC event has multiple phases. Certain phases of the Full GC event (like 'initial-mark', 'remark', 'cleanup',...) pauses all the application threads that are running the JVM. During this period, no customer transactions will be processed. JVM will use all the CPU cycles to perform garbage collection. Due to that CPU consumption will be quite high. Thus in general full GCs aren't desired. Needless to ask the desirability of consecutive full GCs. Consecutive full GCs will cause following problems:

- 1. CPU consumption will spike up.**
- 2. Because JVM is paused, the response time of your application transactions will degrade. Thus it will impact your SLAs and cause poor customer experience.**

In this article, let's learn about this consecutive full GCs – What is it? What causes it? How to fix it?

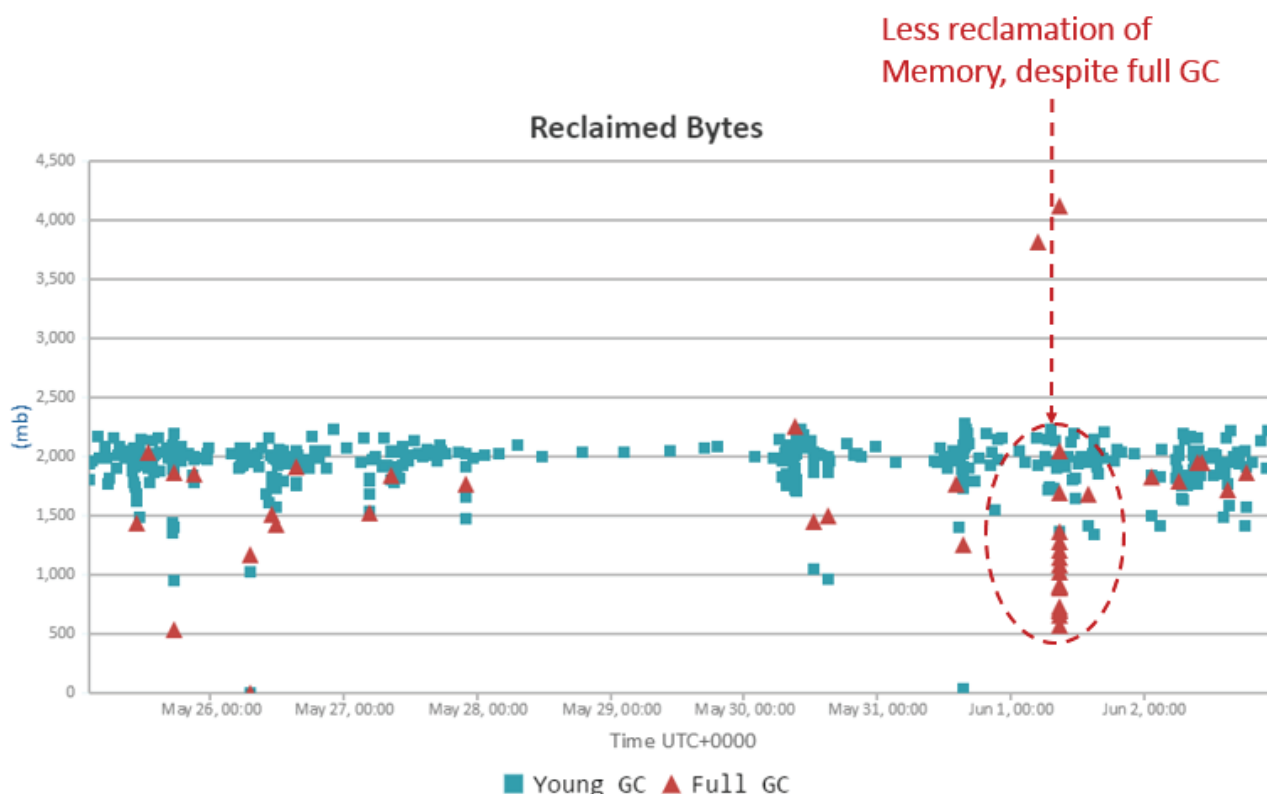
## What is Consecutive full GCs?

It's always easier to explain through an example. So let's examine the GC log of a real world application, which suffered from this consecutive full GC problem. Below are the graphs generated by GCeasy tool by analyzing garbage collection logs. Notice the highlighted portion of the first graph. You can see the full GCs to be consecutively running (red triangles in the graph indicates full GC). If full GC runs consecutively, then it's indicative of a problem.





Even though full GCs were consecutively running, it wasn't able to reclaim enough memory. You can observe it in the second graph (which shows the reclaimed bytes). In this graph you can see the memory reclaimed from these full GCs to be very less. It is because most of the objects in memory are in active use, thus JVM isn't able to reclaim enough memory.



## What causes Consecutive full GCs?

Consecutive Full GCs are caused because of one single reason: Under allocation of JVM heap size or Under allocation of Perm Gen/metaspase. It's indicative of the fact that application needs more memory than what you have allocated. In other words, you are trying to fit in a truck load of objects in a small compact car. So JVM has to work very hard to clean-up the created garbage, to fit in to a small compact car, to make room for actively used objects.

Now you might have a question, my application was running fine all along, why all of a sudden I see this consecutive Full GC problem? That's a valid question. Answer to this question could be one of the following:

1. Your application's tra꜒ volume has started to grow since the last time you have tuned the JVM heap size. May be your business is improving, more users have started to use your application.
2. During your peak volume time period, more objects would get created than normal time. May be you didn't tune your JVM for peak tra꜒ volume or your peak tra꜒ volume has increased since the last time you have tuned the JVM heap size.

## How to solve consecutive Full GCs?

Consecutive Full GCs can be solved through one of the following solution:

### 1. Increase JVM Heap Size

Since consecutive Full GCs runs due to lack of memory, increasing JVM heap size should solve the problem. Say suppose you have allocated JVM heap size to be 2.5GB, then try increasing it to 3 GB and see whether it resolves the problem. JVM heap size can be increased by passing the argument: “-Xmx”. Example:

```
-Xmx3G
```

This argument will set the JVM heap size to be 3 GB. If it still doesn't resolve the problem then try increasing the JVM heap size step by step. Over-allocation of JVM heap is also not good either, it might increase the GC pause time as well.

### 2. Increase Perm Gen/Metaspace Size

Some times full GCs can run consecutively, if Perm Gen or metaspace is under-allocated. In such circumstance just increase the Perm Gen/Metaspace size.

### 3. Add more JVM instances

Adding more JVM instances is an another solution to this problem. When you add more JVM instance, then tra꜀ volume will get distributed. The amount of tra꜀ volume handled by one single JVM instance will go down. If less tra꜀ volume is sent to a JVM instance , then less objects will be created. If less objects are created, then you will not run into the consecutive Full GC problems.

## Validating the fix

Irrespective of the approach you take to resolve the problem, validate the fix in the test environment before rolling out the change to production. Because any changes to JVM heap settings should be thoroughly tested & validated. To validate that problem doesn't resurface with new settings, study your GC log with GCeasy tool. It has the intelligence to spot and report whether the application is su꜀ering from consecutive full GC problem or not.

# ROTATING GC LOG FILES



Garbage Collection logs are essential artifacts to optimize application's performance and trouble shoot memory problems. Garbage Collection logs can be generated in a particular file path by passing the "-Xloggc" JVM argument.

```
Example: -Xloggc:/home/GCEASY/gc.log
```

But the challenge to this approach is: whenever the application is restarted, old GC log file will be over-ridden by the new GC log file as the file path is same (i.e. /home/GCEASY/gc.log).

Thus you wouldn't be able to analyze the old GC logs that existed before restarting the application. Especially if the application has crashed or had certain performance problems then, you need old GC Logs for analysis.

Because of the heat of the production problem, most of the time, IT/DevOps team forgets to back up the old GC log file; A classic problem that happens again & again, that we all are familiar :-). Most human errors can be mitigated through automation and this problem is no exception to it.

A simple strategy to mitigate this challenge is to write new GC log contents in a different file location. In this article 2 different strategies to do that are shared with you:

## 1. Suffix timestamp to GC Log file

If you can suffix the GC log file with the time stamp at which the JVM was restarted then, GC Log file locations will become unique. Then new GC logs will not over-ride the old GC logs. It can be achieved as shown below:

```
"-XX:+PrintGCDetails -XX:+PrintGCDateStamps -Xloggc:/home/GCEASY/gc-%t.log"
```

'%t' suffixes timestamp to the gc log file in the format: 'YYYY-MM-DD\_HH-MM-SS'. So generated GC log file name will start to look like: 'gc-2019-01-29\_20-41-47.log'

This strategy has one minor drawback:

### a. Growing file size

Suppose if you don't restart your JVMs, then GC log file size can be growing to huge size. Because in this strategy new GC log files are created only when you restart the JVM. But this is not a major concern in my opinion, because one GC event only occupies few bytes. So GC log file size will not grow beyond a manageable point for most applications.

## 2. Use -XX:+UseGCLogFileRotation

Another approach is to use the JVM system properties:

```
"-XX:+PrintGCDetails -XX:+PrintGCDateStamps -Xloggc:/home/GCEASY/gc.log -
XX:+UseGCLogFileRotation -XX:NumberOfGCLogFiles=5 -XX:GCLogFileSize=2M"
```

When '-XX:-UseGCLogFileRotation' is passed GC log rotation is enabled by the JVM itself.

'-XX:NumberOfGCLogFiles' sets the number of files to use when rotating logs, must be  $\geq 1$ . The rotated log files will use the following naming scheme, <filename>.0, <filename>.1, ..., <filename>.n-1.

'-XX:GCLogFileSize' defines the size of the log file at which point the log will be rotated, must be  $\geq 8K$

But this strategy has few challenges:

### a. Losing old GC Logs

Suppose if you had configured -XX:NumberOfGCLogFiles=5 then, over a period of time, 5 GC log files will be created:

```
gc.log.0 — oldest GC Log content
gc.log.1
gc.log.2
gc.log.3
gc.log.4 — latest GC Log content
```

Most recent GC log contents will be written to 'gc.log.4' and old GC log content will be present in 'gc.log.0'.

When the application starts to generate more GC logs than the configured '-XX:NumberOfGCLogFiles' in this case 5, then old GC log contents in gc.log.0 will be deleted. New GC events will be written to gc.log.0. It means you will end up not having all the generated GC logs. You will lose the visibility of all events.

### b. Mixed-up GC Logs

Suppose application has created 5 gc log files i.e.

```
gc.log.0
gc.log.1
gc.log.2
gc.log.3
gc.log.4
```

then, let's say you are restarting the application. Now new GC logs will be written to gc.log.0 file and old GC log content will be present in gc.log.1, gc.log.2, gc.log.3, gc.log.4 i.e.

**gc.log.0** — GC log file content after restart  
**gc.log.1** — GC log file content before restart  
**gc.log.2** — GC log file content before restart  
**gc.log.3** — GC log file content before restart  
**gc.log.4** — GC log file content before restart

So your new GC log contents get mixed up with old GC logs. Thus to mitigate this problem you might have to move all the old GC logs to a different folder before you restart the application.

### c. Forwarding GC logs to central location

In this approach, current active file to which GC logs are written is marked with extension “.current”. Example, if GC events are currently written to file ‘gc.log.3’ it would be named as: ‘gc.log.3.current’.

If you want to forward GC logs from each server to a central location, then most DevOps engineers uses ‘rsyslog’. However this file naming convention poses significant challenge to use ‘rsyslog’.

### d. Tooling

Now to analyze the GC log file using the GC tools such as (gceasy.io, GCViewer....), you will have to upload multiple GC log files instead of just one single GC Log file.

## Conclusion

You can debate on which approach you want to take for rotating GC log files, but don’t debate on whether to rotate the GC log files or not. It will come very handy when need comes. You never know when need comes.



# REDUCE LONG GC PAUSES

# 10

Long GC Pauses are undesirable for applications. It affects your SLAs; it results in poor customer experiences, and it causes severe damages to mission critical applications. Thus in this article, I have laid out key reasons that can cause long GC pauses and potential solutions to solve them.

## 1. High Object Creation Rate

If your application's object creation rate is very high, then to keep with it, garbage collection rate will also be very high. High garbage collection rate will increase the GC pause time as well. Thus, optimizing the application to create less number of objects is THE EFFECTIVE strategy to reduce long GC pauses. This might be a time-consuming exercise, but it is 100% worth doing. In order to optimize object creation rate in the application, you can consider using java profilers like JProfiler, YourKit, JVisualVM....). These profilers will report

- What are the objects that created?
- What is the rate at which these objects are created?
- What is the amount of space they are occupying in memory?
- Who is creating them?

*Tit-bit: How to figure out object creation rate?*

### ☒ Object Stats

These are good metrics to compare with previous baseline

Total created bytes ?	3.82 tb
Total promoted bytes ?	16.48 gb
Avg creation rate ?	8.83 mb/sec
Avg promotion rate ?	38 kb/sec

Upload your GC log to the Universal Garbage Collection log analyzer tool GCEasy. This tool will report the object creation rate. There will be field by name 'Avg creation rate' in the section 'Object Stats.' This field will report the object creation rate. Strive to keep this value lower always. See the image (which is an excerpt from the GCEasy generated report), showing the 'Avg creation rate' to be 8.83 mb.sec.

## 2. Undersized Young Generation

When young Generation is undersized, objects will be prematurely promoted to Old Generation. Collecting garbage from old generation takes more time than collecting it from young Generation. Thus increasing young generation size has a potential to reduce the long GC pauses. Young Generation can be increased setting either one of the two JVM arguments

-Xmn: specifies the size of the young generation

-XX:NewRatio: Specifies ratio between the old and young generation. For example, setting

-XX:NewRatio=3 means that the ratio between the old and young generation is 3:1. i.e. young generation will be fourth of the overall heap. i.e. if heap size is 2 GB, then young generation size would be 0.5 GB.

### 3. Choice of GC Algorithm

Choice of GC algorithm has a major influence on the GC pause time. Unless you are a GC expert or intend to become one or someone in your team is a GC expert – you can tune GC settings to obtain optimal GC pause time. Assume if you don't have GC expertise, then I would recommend using G1 GC algorithm, because of its auto-tuning capability. In G1 GC, you can set the GC pause time goal using the system property '-XX:MaxGCPauseMillis.' Example:

```
-XX:MaxGCPauseMillis=200
```

As per the above example, Maximum GC Pause time is set to 200 milliseconds. This is a soft goal, which JVM will try its best to meet it.

### 4. Process Swapping

Sometimes due to lack of memory (RAM), Operating system could be swapping your application from memory. Swapping is very expensive as it requires disk accesses which is much slower as compared to the physical memory access. In my humble opinion – no serious application in a production environment should be swapping. When process swaps, GC will take a long time to complete. Below is the script obtained from StackOverflow (thanks to the author) – which when executed will show all the process that are being swapped. Please make sure your process is not getting swapped.

```
#!/bin/bash
# Get current swap usage for all running processes
# Erik Ljungstrom 27/05/2011
# Modified by Mikko Rantalainen 2012-08-09
# Pipe the output to "sort -nk3" to get sorted output
# Modified by Marc Methot 2014-09-18
# removed the need for sudo

SUM=0
OVERALL=0
for DIR in `find /proc/ -maxdepth 1 -type d -regex "^/proc/[0-9]+"

```

If you find your process to be swapping then do one of the following:

- Allocate more RAM to the server
- Reduce the number of processes that running on the server, so that it can free up the memory (RAM).
- Reduce the heap size of your application (which I wouldn't recommend, as it can cause other side effects).

## 5. Less GC Threads

For every GC event reported in the GC log, user, sys and real time are printed. Example:

```
[Times: user=25.56 sys=0.35, real=20.48 secs]
```

To know the difference between each of these times, please read the article . (I highly encourage you to read the article, before continuing this section). If in the GC events you consistently notice that 'real' time isn't significantly lesser than the 'user' time – then it might be indicating that there aren't enough GC threads. Consider increasing the GC thread count. Say suppose 'user' time 25 seconds, and you have configured GC thread count to be 5, then real time should be close to 5 seconds (because  $25 \text{ seconds} / 5 \text{ threads} = 5 \text{ seconds}$ ).

**WARNING:** Adding too many GC threads will consume a lot of CPU and takes away a resource from your application. Thus you need to conduct thorough testing before increasing the GC thread count.

## 6. Background IO Traffic

If there is a heavy file system I/O activity (i.e. lot of reads and writes are happening) it can also cause long GC pauses. This heavy file system I/O activity may not be caused by your application. Maybe it is caused by another process that is running on the same server, still, can cause your application to suffer from long GC pauses. Here is a brilliant article from LinkedIn Engineers, which walks through this problem in detail.

When there is a heavy I/O activity, you will notice the 'real' time to be significantly more than 'user' time. Example:

```
[Times: user=0.20 sys=0.01, real=18.45 secs]
```

When this pattern happens, here are the potential solutions to solve it:

- If high I/O activity is caused by your application, then optimize it.
- Eliminate the processes which are causing high I/O activity on the server
- Move your application to a different server where I/O activity is less

*Tip-bit: How to monitor I/O activity?*

You can monitor I/O activity, using the sar (System Activity Report), in Unix. Example:

```
sar -d -p 1
```

Above commands reports the reads/sec and writes/sec made to the device every 1 second.

## 7. System.gc() calls

When System.gc() or Runtime.getRuntime().gc() method calls are invoked it will cause stop-the-world Full GCs. During stop-the-world full GCs, entire JVM is freezed (i.e. No user activities will be performed during period). System.gc() calls are made from one of the following sources:

- Your own application developers might be explicitly calling System.gc() method.
- It could be 3rd party libraries, frameworks, sometimes even application servers that you use could be invoking System.gc() method.

3. It could be triggered from external tools (like VisualVM) through use of JMX
4. If your application is using RMI, then RMI invokes `System.gc()` on a periodic interval. This interval can be configured using the following system properties:
  - `Dsun.rmi.dgc.server.gcInterval=n`
  - `Dsun.rmi.dgc.client.gcInterval=n`

Evaluate whether it's absolutely necessary to explicitly invoke `System.gc()`. If there is no need to then, please remove it. On the other hand, you can forcefully disable the `System.gc()` calls by passing the JVM argument: `'-XX:+DisableExplicitGC'`.

*Tit-bit: How to know whether `System.gc()` calls are explicitly called?*

### GC Causes ?

Cause	Count
Allocation Failure ?	242
System.gc() calls ?	4
Promotion Failure ?	1
GCLocker Initiated GC ?	2

Upload your GC log to the Universal Garbage Collection log analyzer tool GCEasy. This tool has a section called 'GC Causes.' If GC activity is triggered because of '`System.gc()`' calls then it will be reported in this section. See the image (which is an excerpt from the GCEasy generated report), showing that `System.gc()` was made 4 times during the lifetime of this application.

## 8. Large Heap size

Large heap size (`-Xmx`) can also cause long GC pauses. If heap size is quite high, then more garbage will be get accumulated in the heap. When Full GC is triggered to evict the all the accumulated garbage in the heap, it will take long time to complete. Logic is simple: If you have small can full of trash, it's going to be quick and easy to dispose them. On the other hand if you have truck load of trash, it's going to take more time to dispose them.

Suppose your JVMs heap size is 18GB, then consider having three 6 GB JVM instances, instead of one 18GB JVM. Small heap size has great potential to bring down the long GC pauses.

**CAUTION:** All of the above mentioned strategies should be rolled to production only after thorough testing & analysis. All strategies may not apply to your application. Improper usage of these strategies can result in negative results.

# WHICH GC TO USE?



	Serial GC	Parallel GC	CMS GC	G1 GC
<i>How it works?</i>	<a href="#">Refer here</a>	<a href="#">Refer here</a>	<a href="#">Refer here</a>	<a href="#">Refer here</a>
<i>How to enable?</i>	-XX:+UseSerialGC	XX: +UseParallelGC	XX:+UseConcMarkSweepGC	-XX:+UseG1GC
<i>CPU Impact</i>	best (least CPU consumption) ★	2nd best	High	High
<i>Total GC Pause time</i>	Worst	2nd best	Best ★	3rd best
<i>Max GC Pause time</i>	Worst	3rd best	2nd best	Best ★
<i>JDK versions</i>	all	from JDK 5	from JDK 6	from JDK 8

# HOW TO TAKE THREAD DUMPS? – 8 OPTIONS

# 12

Thread dumps are vital artifacts to diagnose CPU spikes, deadlocks, poor response times, memory problems, unresponsive applications, and other system problems. There are great online thread dump analysis tools such as <http://fastthread.io/>, which can analyse and spot problems. But to those tools you need provide proper thread dumps as input. Thus in this article, I have documented 8 different options to capture thread dumps.

## 1. jstack

'jstack' is an effective command line tool to capture thread dumps. jstack tool is shipped in JDK\_HOME/bin folder. Here is the command that you need to issue to capture thread dump:

```
jstack -l <pid> > <file-path>
```

where

pid: is the Process Id of the application, whose thread dump should be captured

file-path: is the file path where thread dump will be written in to.

Example:

```
jstack -l 37320 > /opt/tmp/threadDump.txt
```

As per the example thread dump of the process would be generated in /opt/tmp/threadDump.txt file.

Jstack tool is included in JDK since Java 5. If you are running in older version of java, consider using other options

## 2. Kill -3

In major enterprises for security reasons only JREs are installed on production machines. Since jstack and other tools are only part of JDK, you wouldn't be able to use jstack tool. In such circumstances 'kill -3' option can be used.

```
kill -3 <pid>
```

where

pid: is the Process Id of the application, whose thread dump should be captured

Example:

```
Kill -3 37320
```

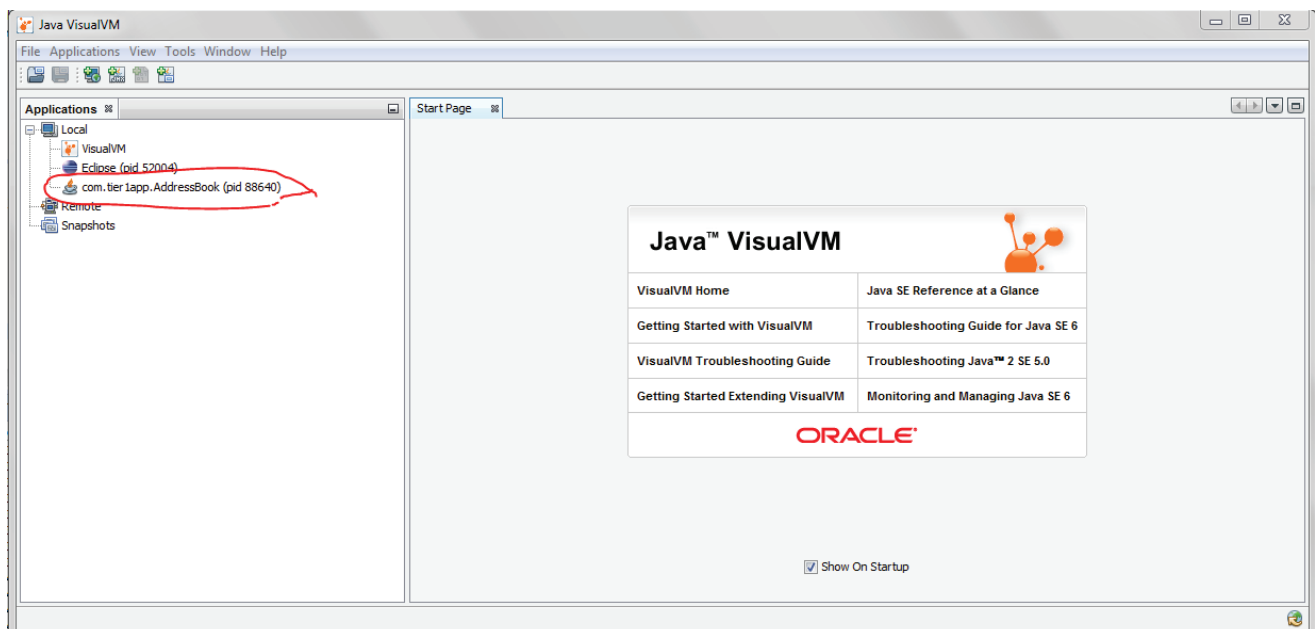
When 'kill -3' option is used thread dump is sent to the standard error stream. If you are running your application in tomcat, thread dump will be sent into <TOMCAT\_HOME>/logs/catalina.out file.

Note: To my knowledge this option is supported in most flavours of \*nix operating systems (Unix, Linux, HP-UX operating systems). Not sure about other Operating systems.

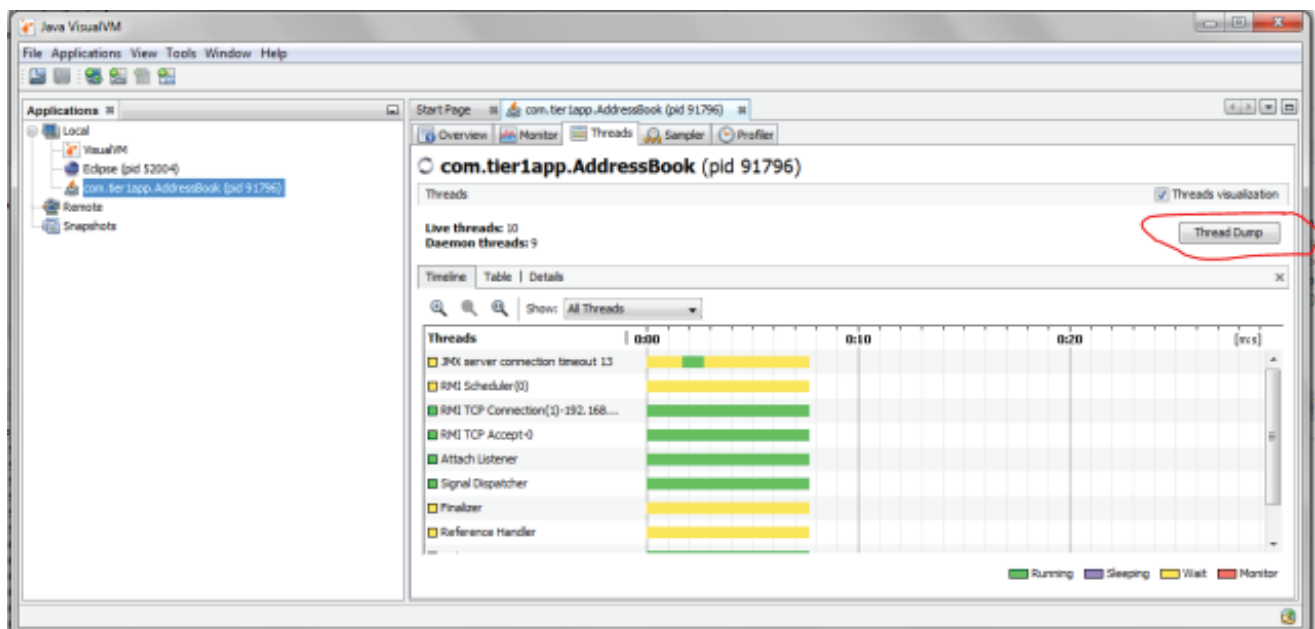
### 3. JVisualVM

Java VisualVM is a graphical user interface tool that provides detailed information about the applications while they are running on a specified Java Virtual Machine (JVM). It's located in JDK\_HOME\bin\jvisualvm.exe. It's part of Sun's JDK distribution since JDK 6 update 7.s

Launch the jvisualvm. On the left panel, you will notice all the java applications that are running on your machine. You need to select your application from the list (see the red color highlight in the below diagram). This tool also has the capability to capture thread dumps from the java processes that are running on the remote host as well.



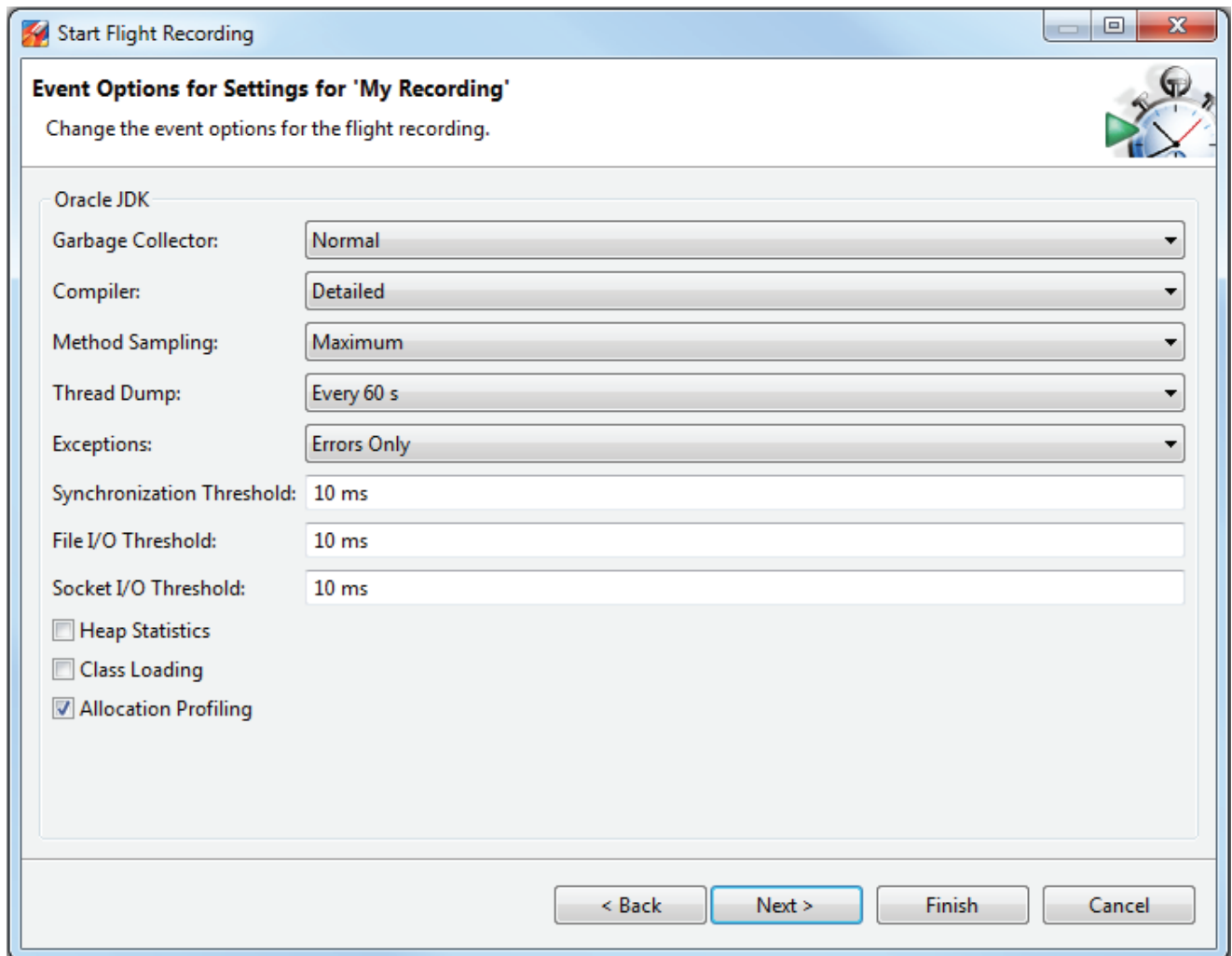
Now go to the “Threads” tab. Click on the “Thread Dump” button as shown in the below image. Now Thread dumps would be generated.



## 4. JMC

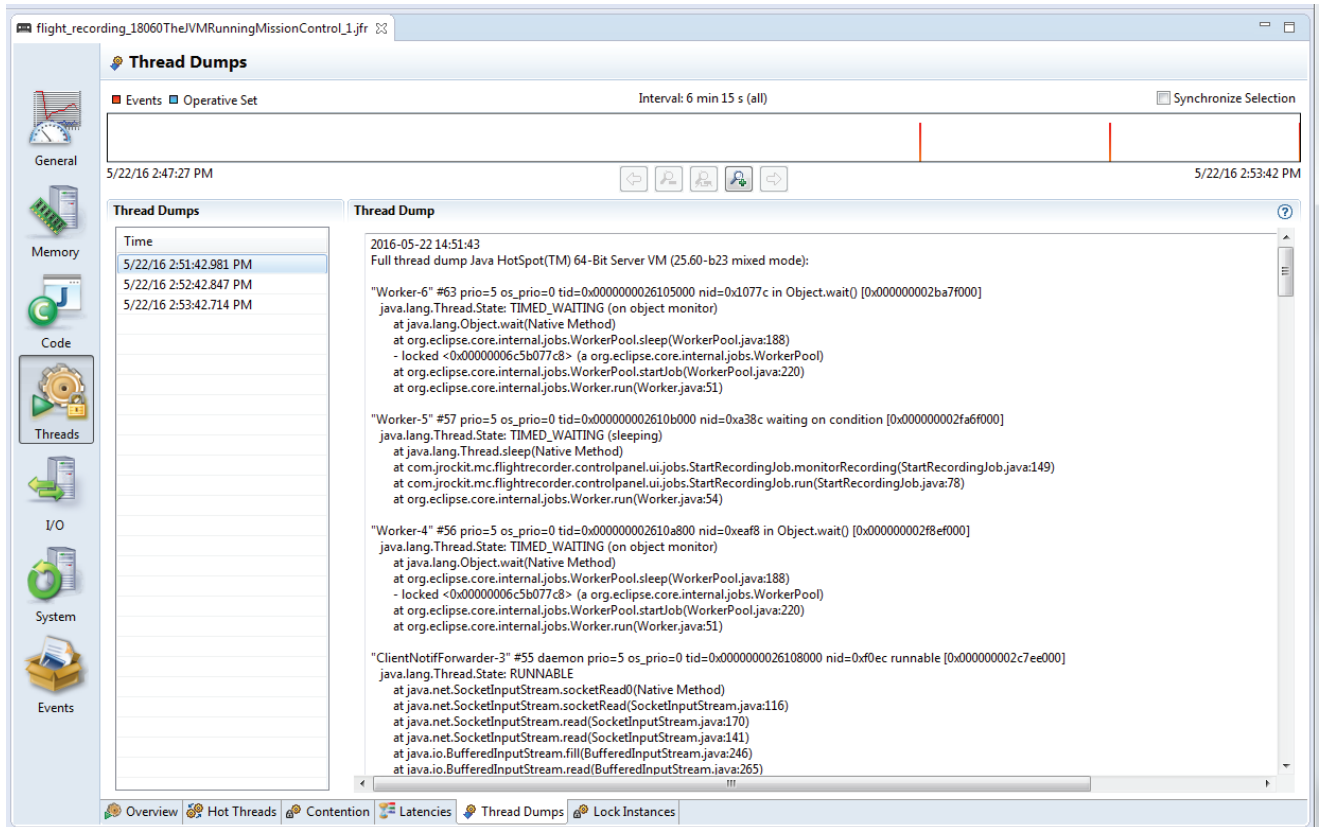
Java Mission Control (JMC) is a tool that collects and analyze data from Java applications running locally or deployed in production environments. This tool has been packaged into JDK since Oracle JDK 7 Update 40. This tool also provides an option to take thread dumps from the JVM. JMC tool is present in `JDK_HOME\bin\jmc.exe`

Once you launch the tool, you will see all the Java processes that are running on your local host. Note: JMC also can connect with java processes running on the remote host. Now on the left panel click on the “Flight Recorder” option that is listed below the Java process for which you want to take thread dumps. Now you will see the “Start Flight Recording” wizard, as shown in the below figure.



Here in the “Thread Dump” field, you can select the interval in which you want to capture thread dump. As per the above example, every 60 seconds thread dump will be captured. After the selection is complete start the Flight recorder. Once the recording is complete, you will see the thread dumps in the “Threads” panel, as shown in the figure next page.





## 5. Windows (Ctrl + Break)

This option will work only in Windows Operating system.

Select command line console window in which you have launched application.

Now on the console window issue the “Ctrl + Break” command.

This will generate thread dump. A thread dump will be printed on the console window itself.

**Note 1:** in several laptops (like my Lenovo T series) “Break” key is removed. In such circumstance, you have to google to find the equivalent keys for the “Break.” In my case, it turned out that “Function key + B” is the equivalent of “Break” key. Thus I had to use “Ctrl + Fn + B” to generate thread dump.

**Note 2:** But one disadvantage with the approach is thread dump will be printed on the windows console itself. Without getting the thread dump in a file format, it’s hard to use the thread dump analysis tools such as <http://fastthread.io>. Thus when you launch the application from the command line, redirect the output a text file i.e. Example if you are launching the application “SampleThreadProgram”, you would issue the command:

```
java -classpath . SampleThreadProgram
```

instead, launch the SampleThreadProgram like this

```
java -classpath . SampleThreadProgram > C:\workspacethreadDump.txt 2>&1
```

Thus when you issue “Ctrl + Break” thread dump will be sent to C:\workspacethreadDump.txtfile.

## 6. ThreadMXBean

Since JDK 1.5 ThreadMXBean has been introduced. This is the management interface for the thread system in the Java Virtual Machine. Using this interface also you can generate thread dumps. You only have to write few lines of code to generate thread dumps programmatically. Below is a skeleton implementation on ThreadMXBean implementation, which generates Thread dump from the application.

```
public void dumpThreadDump() {
    ThreadMXBean threadMxBean = ManagementFactory.getThreadMXBean();
    for (ThreadInfo ti : threadMxBean.dumpAllThreads(true, true)) {
        System.out.print(ti.toString());
    }
}
```

## 7. APM Tool – App Dynamics

Few Application Performance Monitoring tools provide options to generate thread dumps. If you are monitoring your application through App Dynamics (APM tool), below are the instructions to capture thread dump:

1. Create an action, selecting Diagnostics->Take a thread dump in the Create Action window.
2. Enter a name for the action, the number of samples to take, and the interval between the thread dumps in milliseconds.
3. If you want to require approval before the thread dump action can be started, check the Require approval before this Action checkbox and enter the email address of the individual or group that is authorized to approve the action. See Actions Requiring Approval for more information.
4. Click OK.

**Create Thread Dump**

Thread Dumps will be taken by the Java App Server Agent, and uploaded to the controller when completed. You can download them on the Events screen (for events that trigger Policies).

Name

Number of thread dumps  Maximum 50 ?

Interval in ms  Maximum 500 ms ?

Require approval before executing this Action ☐ ?

E-mail address for approver

[Configure Email / SMS settings](#)

Cancel OK

## 8. JCMD

The jcmd tool was introduced with Oracle's Java 7. It's useful in troubleshooting issues with JVM applications. It has various capabilities such as identifying java process ids, acquiring heap dumps, acquiring thread dumps, acquiring garbage collection statistics, ....

Using the below JCMD command you can generate thread dump:

```
jcmd <pid> Thread.print > <file-path>
```

where

pid: is the Process Id of the application, whose thread dump should be captured

file-path: is the file path where thread dump will be written in to.

Example:

```
jcmd 37320 Thread.print > /opt/tmp/threadDump.txt
```

As per the example thread dump of the process would be generated in /opt/tmp/threadDump.txt file.

## Conclusion

Even though 8 different options are listed to capture thread dumps, IMHO , 1. 'jstack' and 2. 'kill -3' and 8. 'jcmd' are the best ones. Because they are:

- a. Simple (straightforward, easy to implement)
- b. Universal (works in most of the cases despite OS, Java Vendor, JVM version, ...)

# THREAD DUMP ANALYSIS API

# 13

In this modern world, thread dumps are still analyzed in a tedious & manual mode i.e., you have to get hold of DevOps team, ask them to send you the thread dumps, then they will mail you the thread dumps, then you will upload the dumps in to a thread dump analysis tool, then you have to apply your intelligence to analyze it. There is no programmatic way to analyze thread dumps in a proactive manner. Thus to eliminate this hassle, fastthread.io is introducing a RESTful API to analyze thread dumps. With one line of CURL command, you can get your thread dumps analyzed instantly.

Here are a few use cases where this API can be extremely useful.

## Use case 1: Automatic Root cause Analysis

Most of the DevOps invokes a simple Http ping or APM tools to monitor the applications health. This ping is good to detect whether application is alive or not. APM tools are great at informing that application's CPU spiked up by 'x%', memory utilization increased by 'y%', response time dropped by 'z' milliseconds. It won't inform what caused the CPU to spike up, what caused memory utilization to increase, what caused the response time to degrade. If you can configure Cron job to capture thread dumps/GC logs on a periodic interval and invoke our REST API, we apply our intelligent patterns & machine learning algorithms to instantly identify the root cause of the problem.

**Advantage 1:** Whenever these sort of production problem happens, because of the heat of the moment, DevOps team recycles the servers with out capturing the thread dumps and GC logs. You need to capture thread dumps and GC logs at the moment when problem is happening, in order to diagnose the problem. In this new strategy you don't have to worry about it, because your cron job is capturing thread dumps/GC logs on a periodic intervals and invoking the REST API, all your thread dumps/GC Logs are archived in our servers.

**Advantage 2:** Unlike APM tools which claims to add less than 3% of overhead, where as in reality it adds multiple folds, beauty of this strategy is: It doesn't add any overhead (or negligible overhead). Because entire analysis of the thread dumps/GC logs are done on our servers and not on your production servers..

## Use case 2: Performance Tests

When you conduct performance tests, you might want to take thread dumps/GC logs on a periodic basis and get it analyzed through the API. In case if thread count goes beyond a threshold or if too many threads are WAITING or if any threads are BLOCKED for a prolonged period or lock isn't getting released or frequent full GC activities happening or GC pause time exceeds thresholds, it needs to get the visibility right then and there. It should be analyzed before code hits the production. In such circumstance this API will become very handy.

## Use case 3: Continuous Integration

As part of continuous integration it's highly encouraged to execute performance tests. Thread dumps/GC Logs should be captured and it can be analyzed using the API. If API reports any problems, then build can be failed. In this way, you can catch the performance degradation right during code commit time instead of catching it in performance labs or production.

## How to invoke Thread dump analysis API?

Invoking Thread dump analysis API is very extremely simple:

1. Register with us. We will email you the API key. This is a one-time setup process. Note: If you have purchased enterprise version with API, you don't have to register. API key will be provided to you as part of installation instruction.
2. Post HTTP request to `http://api.fastthread.io/fastthread-api?apiKey={API_KEY_SENT_IN_EMAIL}`
3. The body of the HTTP request should contain the Thread dump that needs to be analyzed. You can either send 1 thread dump or multiple thread dumps in the same request.
4. HTTP Response will be sent back in JSON format. JSON has several important stats about the Thread dump. Primary element to look in the JSON response is: "problem". API applies several intelligent thread dump analysis patterns and if it detects any issues, it will reported in this "problem" element.

## CURL command

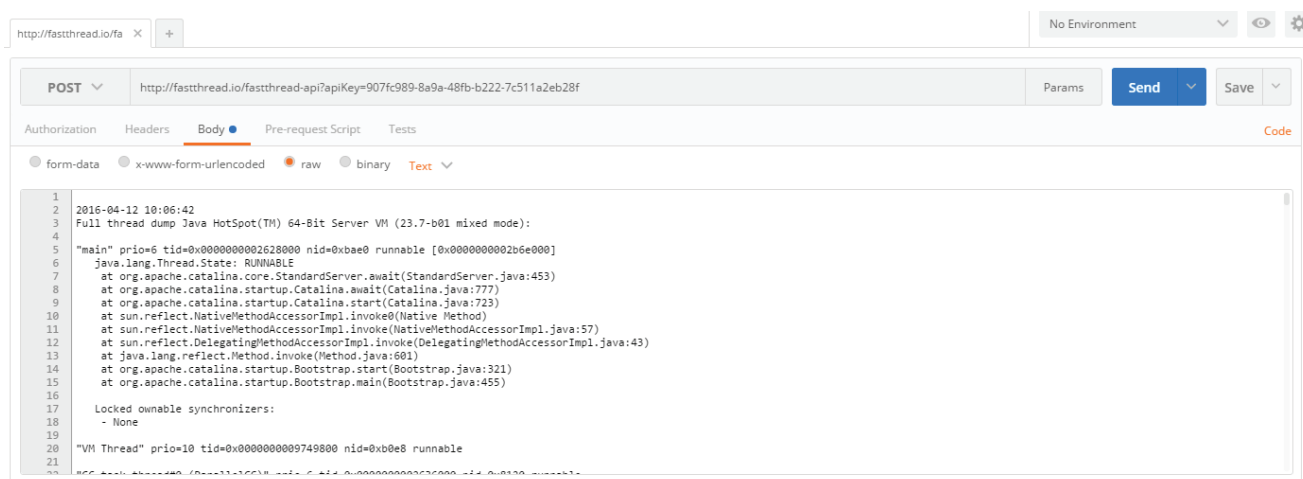
Assuming your Thread dump file is located in `./my-thread-dump.txt`, then CURL command to invoke the API is:

```
curl -X POST --data-binary @./my-thread-dump.txt
http://api.fastthread.io/fastthread-api?apiKey={API_KEY_SENT_IN_EMAIL} --header
"Content-Type:text"
```

It can't get any more simpler than that? Isn't it?

## Other Tools

You can also invoke the API using any web service client tools such as: SOAP UI, Postman Browser Plugin,.....



## Sample Response

```

{
  "problem": [
    {
      "level": "SEVERE",
      "description": "8 thread are looping on same lines of code. If threads loop infinitely on the
same lines of code, CPU consumption will start to spike up"
    }
  ],
  "threadsRemainingInWaitingState": [
    {
      "method": "java.lang.Object.wait(Native Method)",
      "threadCount": 3,
      "threads": "Reference Handler, Dispatcher-Thread-2, Finalizer"
    },
    {
      "method": "sun.misc.Unsafe.park(Native Method)",
      "threadCount": 2,
      "threads": "New Relic RPM Connection Service, New Relic Retransformer"
    }
  ],
  "threadDumpReport": [
    {
      "timestamp": "2016-03-03 10:37:28",
      "JVMTType": " 64-Bit Server VM (23.7-b01 mixed mode)",
      "threadState": [
        {
          "state": "RUNNABLE",
          "threadCount": 28,
          "threads": "Attach Listener, InvoiceGeneratedQC-A99-6, InvoiceGeneratedQC-H87-6,
InvoiceGeneratedQC-B85-9, InvoiceGeneratedQC-A99-6, InvoiceGeneratedQC-H87-6,
InvoiceGeneratedQC-H87-3, InvoiceGeneratedQC-H87-1, InvoiceGeneratedQC-B85-9, Service
Thread, C2 CompilerThread1, C2 CompilerThread0, Signal Dispatcher, main, VM Thread, GC
task thread#0 (ParallelGC), GC task thread#1 (ParallelGC), GC task thread#2 (ParallelGC), GC
task thread#3 (ParallelGC), GC task thread#4 (ParallelGC), GC task thread#5 (ParallelGC), GC
task thread#6 (ParallelGC), GC task thread#7 (ParallelGC), GC task thread#8 (ParallelGC), GC
task thread#9 (ParallelGC), GC task thread#10 (ParallelGC), GC task thread#11 (ParallelGC), GC
task thread#12 (ParallelGC)"
        },
        {
          "state": "WAITING",
          "threadCount": 6,
          "threads": "Dispatcher-Thread-2, New Relic RPM Connection Service, New Relic
Retransformer, Finalizer, Reference Handler, VM Periodic Task Thread"
        },
        {
          "state": "TIMED_WAITING",

```

# HOW TO IDENTIFY CRITICAL CODE PATH?

# 14

Before answering the question ‘How to identify critical code path?’ let me answer ‘Why identify critical code path?’

## Why identify critical code path?

There are a couple of answers to it:

1. Performance Optimization
2. Accurate Smoke Test

### Performance Optimization

In most applications, we have observed that less than 5% of application code accounts for more than 90% of code execution. Thus, if you can optimize this 5% of code, you can improve your entire application’s performance significantly. It’s the best ROI. You can save a significant amount of time in not analyzing the remaining 95% of code.

### Accurate Test Suite

You can write highly targeted unit tests to exercise the critical code path and make your application bullet-proof. In fact, these tests can act as your application’s smoke test. It can be integrated into your CI/CD pipeline. This sort of accurate test suite reduces over-all test execution time. It can also reduce the test data setup time in the backend systems.

Now let’s get back to our original question: How to identify critical code path? This is where thread dumps come handy.

## Thread dumps

Thread dumps are the snapshot of all threads running in the application at given moment. It contains detailed information about each thread including its stack trace. Below is the information provided for the “InvoiceGeneratedQC-A99-6” thread in the thread dump.

```

"InvoiceGeneratedQC-A99-6" prio=10 tid=0x00002b7cfc6fb000 nid=0x4479 runnable [0x00002b7d17ab8000]
java.lang.Thread.State: RUNNABLE
  9 at com.buggycompany.rt.util.ItinerarySegmentProcessor.setConnectingFlight(ItinerarySegmentProcessor.java:380)
    8 at com.buggycompany.rt.util.ItinerarySegmentProcessor.processTripType0(ItinerarySegmentProcessor.java:366)
      7 at com.buggycompany.rt.util.ItinerarySegmentProcessor.processItineraryTripType(ItinerarySegmentProcessor.java:254)
        6 at com.buggycompany.framework.concurrent.buggycompanyCallable.call(buggycompanyCallable.java:80)
          5 at java.util.concurrent.FutureTask$Sync.innerRun(FutureTask.java:334)
            4 at java.util.concurrent.FutureTask.run(FutureTask.java:166)
              3 at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1145)
                2 at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:615)
                  1 at java.lang.Thread.run(Thread.java:722)

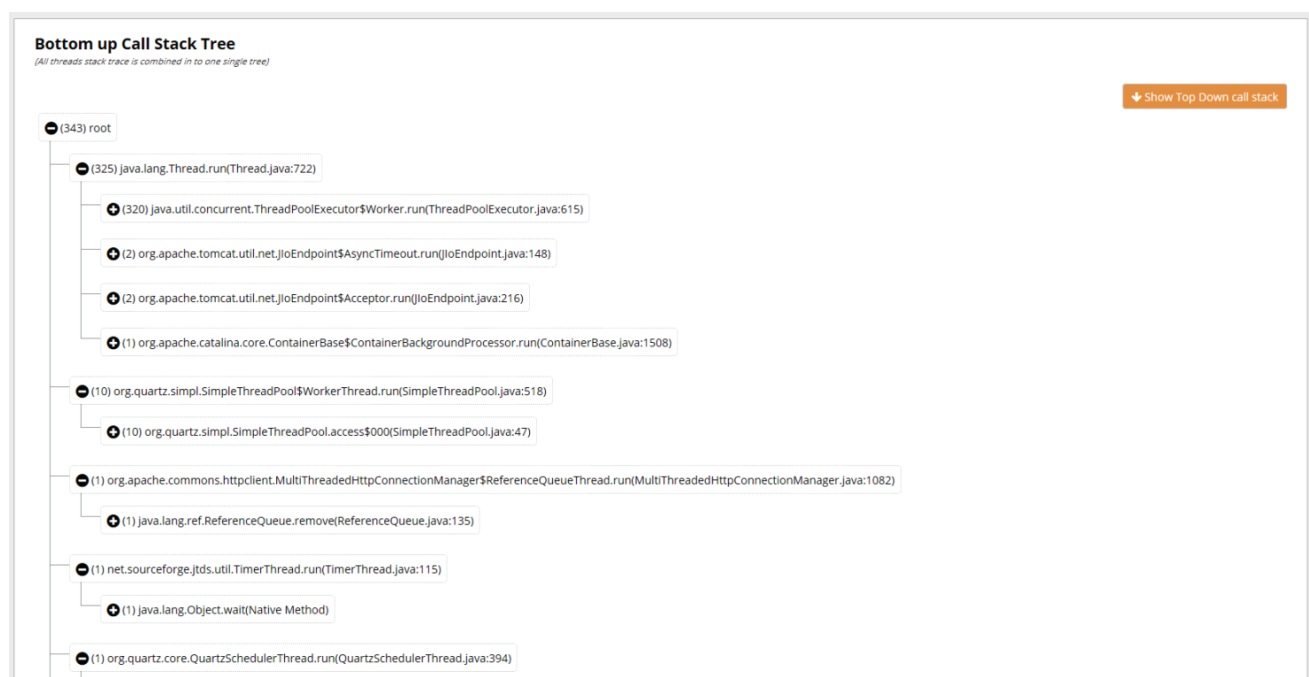
```

Above stack trace tells you the code execution path of the “InvoiceGeneratedQC-A99-6” thread. This thread has executed 9 methods in the order (1 – 9), as shown in Fig 1. Thread executed the `java.lang.Thread.run()` method first, then it went on to execute `java.util.concurrent.ThreadPoolExecutor$Worker.run()` method from there it went on to execute other methods until `com.buggycompany.rt.util.ItinerarySegmentProcessor.setConnectingFlight()`.

If multiple threads execute same code execution path, that code path becomes your critical path. You can group stack trace of all the threads, combine them together to form one single call stack tree & from there you can identify the critical code execution path.

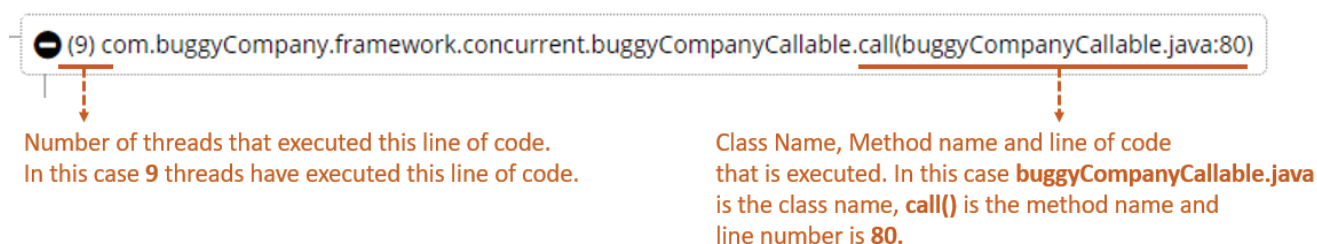
## Call Stack Tree

Tools like `fastThread.io` can group stack trace of all the threads and generate one single call stack tree. You can drill down & up on this tree to see the critical code path. Below is the sample call stack tree generated by ‘fastThread.io’.





Call Stack Tree shows you the class name, method name and line of the code that has been executed and a number of threads that have executed the line of code.



From the above element in the Call Stack Tree, you can identify that call() method in buggyCompanyCallable.java is executed by 9 threads. Since 9 threads are executing this method, it's a good candidate to be categorized as critical code path.

Note that Call Stack Tree will contain code from Java, external frameworks, and libraries because your application executes those codes as well. When writing focused tests or optimizing the performance of the critical code path, you can ignore or give low priority to that external code, as you have very little control over them.

## Best Practices:

### Capture thread dumps from production

It's extremely hard (if not impossible) to mirror production trac in a test environment. As you need to identify the critical code execution path in production, it's ideal to capture thread dumps from production environment instead of test or other lower environments.

### Capture thread dumps during peak volume time

It's ideal to capture thread dumps during peak volume time period. It's where you can observe maximum possibilities of application code exercise.

### Capture 3 – 5 thread dumps

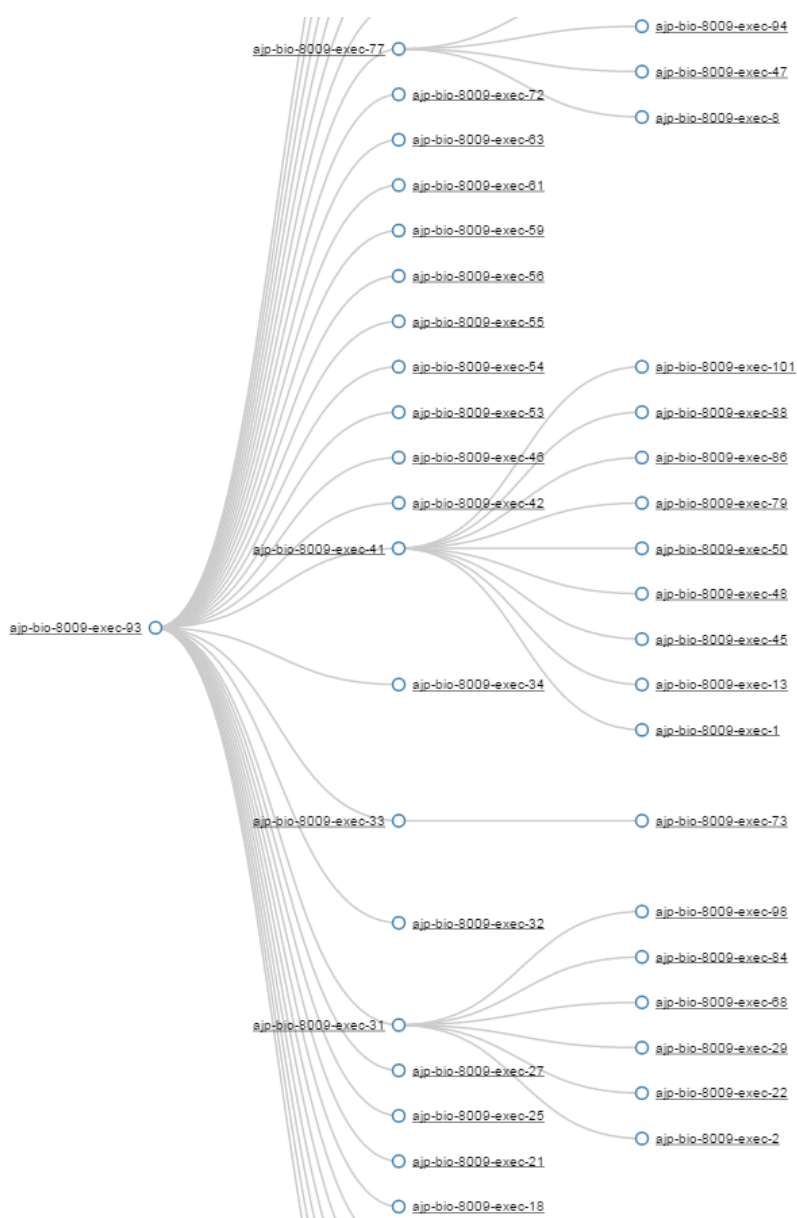
As thread dump is a snapshot of a particular point in time, it's ideal to capture at least 3 – 5 thread dumps to see few possibilities of code execution path. There are multiple options to capture thread dumps. Use your favorite option to capture the thread dump.

# WHAT'S THE DIFFERENCE BETWEEN BLOCKED, WAITING, AND TIMED\_WAITING? – EXPLAINED TO REAL-LIFE EXAMPLES

15

BLOCKED, WAITING, and TIMED\_WAITING are important thread states, but often confusing to many of us. One must have a proper understanding of both in order to analyze thread dumps. Using real-life examples, this article breaks down each state into simpler terms.

Any confusing concepts can be easily understood through examples rather than formal definitions given in Java doc. If they are real-life examples, it can be even more relatable. I would like to share some real-life examples which might help to understand these thread states.



## BLOCKED

Java doc formally defines BLOCKED state as: “A thread that is blocked waiting for a monitor lock is in this state.”

**Real-life example:** Today you are going for a job interview. This is your dream job, which you have been targeting for last few years. You woke up early in the morning, got ready, put on your best outfit, looking sharp in front of the mirror. Now you step out to your garage and realize that your wife has already taken the car. In this scenario, you only have one car, so what will happen? In real life, a fight may happen. However, you are BLOCKED because your wife has already taken the car. You won't be able to go to the interview.

This is the **BLOCKED** state. Explaining it in technical terms, you are the thread T1 and your wife is the thread T2 and lock is the car. T1 is BLOCKED on the lock (i.e. the car) because T2 has already acquired this lock.

**Titbit:** A Thread will enter into BLOCKED state when it's waiting for a monitor lock to enter a synchronized block/method or reenter a synchronized block/method after calling `Object#wait()` method.

## WAITING

Java doc formally defines WAITING state as: “A thread that is waiting indefinitely for another thread to perform a particular action is in this state.”

**Real-life example:** Let's say a few minutes later your wife comes back home with the car. Now you realize that the interview time is approaching, and there is a long distance to drive to get there. So, you put all the power on the gas pedal in the car. You drive at 100 mph when the allowed speed limit is only 60 mph. Your luck, a traffic cop sees you driving over the speed limit, and he pulls you over to the curb. Now you are entering into the WAITING state, my friend. You stop driving the car and sit idly in the car until the cop investigates you and then lets you go. Basically, until he lets you go, you are stuck in the WAITING state.

Explaining it in technical terms, you are thread T1 and the cop is thread T2. You released your lock (i.e. you stopped driving the car) and went into the WAITING state. Until the cop (i.e. T2) lets you go, you will be stuck in this **WAITING** state.

**Titbit:** A Thread will enter into WAITING state when it's calling one of the following methods:

`Object#wait()` with no timeout

`Thread#join()` with no timeout

`LockSupport#park()`

Thread that has called `Object.wait()` on an object is in WAITING state until another thread to call `Object.notify()` or `Object.notifyAll()` on that object. A thread that has called `Thread.join()` is in WAITING state for a specified thread to terminate.

## TIMED\_WAITING

Java doc formally defines TIMED\_WAITING state as: “A thread that is waiting for another thread to perform an action for up to a specified waiting time is in this state.”

**Real-life example:** Despite all the drama, you did extremely well in the interview, impressed everyone and got this high paying job. (Congratulations!) You come back home and tell your neighbor about this new job and how excited you are about it. Your friend says that he is also working in the same office building. He suggests that the two of you should drive together. You think it's a great idea. So on the first day of work, you go to his house. You stop your car in front of his house. You wait for 10 minutes, but your neighbor still doesn't come out. You go ahead and start driving to work, as you don't want to be delayed on your first day. Now this is **TIMED\_WAITING**.

Explaining it in technical terms, you are thread T1, and your neighbor is thread T2. You release the lock(i.e. stop driving the car) and wait up to 10 minutes. If your neighbor, T2, doesn't come out in 10 minutes, you start driving the car again.

**Titbit:** A Thread will enter into TIMED\_WAITING state when it's calling one of the following methods:

```
Thread#sleep()
Object#wait() with timeout
Thread#join() with timeout
LockSupport#parkNanos()
LockSupport#parkUntil()
```

## Conclusion

When someone is analyzing thread dumps, understanding these different thread states are critical. How many threads are in RUNNABLE, BLOCKED, WAITING, and TIMED\_WAITING states? Which threads are blocked? Who is blocking them? What object used for locking? These are some of the important metrics to be analyzed in thread dumps. These kinds of detailed thread dump analyses can easily be done through an online tool such as <http://fastthread.io/>

# THREAD DUMP ANALYSIS PATTERN – ATHLETE

16

## Description

Threads in 'runnable' state consume CPU. So when you are analyzing thread dumps for high CPU consumption, threads in 'runnable' state should be thoroughly reviewed. Typically in thread dumps several threads are classified in 'RUNNABLE' state. But in reality several of them wouldn't be actually running, rather they would be just waiting. But still, JVM classifies them in 'RUNNABLE' state. You need to learn to differentiate from really running threads with pretending/misleading RUNNABLE threads.

## Example

Below is the real world thread dump excerpt. In these stack traces, threads aren't actually in 'RUNNABLE' state. i.e. they are not actively executing any code. They are just waiting on the sockets to read or write. It's because JVM doesn't really know what thread is doing in a native method, JVM classifies them in 'RUNNABLE' state. Real running threads would consume CPU, whereas these threads are on I/O wait, which don't consume any CPU.

```
WorkerThread#27[10.2011.55:56893] - priority:10 - threadId:0x00002b59983cb000 -
nativeId:0x4482 - addressSpace:null - state:RUNNABLE
stackTrace:
java.lang.Thread.State: RUNNABLE
at java.net.SocketInputStream.socketRead0(Native Method)
at java.net.SocketInputStream.__AW_read(SocketInputStream.java:129)
at java.net.SocketInputStream.read(SocketInputStream.java)
at java.io.BufferedInputStream.fill(BufferedInputStream.java:218)
at java.io.BufferedInputStream.read(BufferedInputStream.java:237)
- locked (a java.io.BufferedInputStream)
at java.io.DataInputStream.readByte(DataInputStream.java:248)
at
org.jboss.jms.server.remoting.ServerSocketWrapper.checkConnection(ServerSocketWrapper.java
:94)
at org.jboss.remoting.transport.socket.ServerThread.acknowledge(ServerThread.java:857)
at org.jboss.remoting.transport.socket.ServerThread.dorun(ServerThread.java:585)
at org.jboss.remoting.transport.socket.ServerThread.run(ServerThread.java:234)
```

```

multicast receiver,GATES-RefData-Infinispan-Cluster,matcamupa67-vm-46719 - priority:10 -
threadId:0x00002b597c1ba800 - nativeId:0x3034 - addressSpace:null - state:RUNNABLE
stackTrace:
java.lang.Thread.State: RUNNABLE
at java.net.PlainDatagramSocketImpl.receive0(Native Method)
- locked (a java.net.PlainDatagramSocketImpl)
at java.net.PlainDatagramSocketImpl.receive(PlainDatagramSocketImpl.java:145)
- locked (a java.net.PlainDatagramSocketImpl)
at java.net.DatagramSocket.receive(DatagramSocket.java:725)
- locked (a java.net.DatagramPacket)
- locked (a java.net.MulticastSocket)
at org.jgroups.protocols.UDP$PacketReceiver.__AW__run(UDP.java:675)
at org.jgroups.protocols.UDP$PacketReceiver.run(UDP.java)
at java.lang.Thread.run(Thread.java:662)

```

### Why named as Athlete Pattern?

Wikipedia defines Athlete as a person who competes in one or more sports that involve physical strength, speed, and/or endurance. Athlete consumes high energy, similarly really RUNNABLE threads consumes high CPU, whereas pretending RUNNABLE threads don't.

# DEADLOCK



## Description

Wikipedia aptly defines deadlock is a situation in which two or more competing actions are each waiting for the other to finish, and thus neither ever does. If deadlock happens in a JVM, the only way to recover from the situation is to restart the JVM.

## Example

Here is a sample code which simulates deadlock condition in-between two threads:

```
package com.tier1app;

public class DeadLockSimulator {
    public static Object Lock1 = new Object();
    public static Object Lock2 = new Object();
    private static class FirstThread extends Thread {
        public void run() {
            synchronized (Lock1) {
                System.out.println("Threadolding lock 1...");
                try { Thread.sleep(10); } catch (Exception e) {}
                System.out.println("Threadaiting for lock 2...");
                synchronized (Lock2) {
                    System.out.println("Threadolding lock 1 & 2...");
                }
            }
        }
    }
    private static class SecondThread extends Thread {
        public void run() {
            synchronized (Lock2) {
                System.out.println("Threadolding lock 2...");
                try { Thread.sleep(10); } catch (Exception e) {}
                System.out.println("Threadaiting for lock 1...");
                synchronized (Lock1) {
                    System.out.println("Threadolding lock 1 & 2...");
                }
            }
        }
    }
}
```

In the above code following is the execution path of 'FirstThread':

1. Acquires lock on the Lock1 object
2. Sleeps for 10 seconds interval
3. Acquires lock on Lock2 object

Following is the execution path of 'SecondThread':

1. Acquires lock on the Lock2 object
2. Sleeps for 10 seconds interval
3. Acquires lock on Lock1 object

If you read the above execution path carefully, FirstThread after executing step #1, it would have moved on to step #2. When it is in step #2, SecondThread would have executed it's step #1. So by the time FirstThread wakes up and executes it's step #3 (i.e. trying to acquire Lock2), SecondThread would have already acquired lock on Lock2. Similarly, SecondThread executes it's step #3 (i.e. trying to acquire Lock1), FirstThread has already acquired lock on Lock1. Thus it results in classic deadlock situation. The only way to recover from this situation is to restart the JVM.

Thread dump captured on the above code would look like:

```
"Thread-1" prio=6 tid=0x0000000007319000 nid=0x7cd3c waiting for monitor entry
[0x0000000008a3f000]
  java.lang.Thread.State: BLOCKED (on object monitor)
  at com.tier1app.DeadLockSimulator$SecondThread.run(DeadLockSimulator.java:29)
  - waiting to lock 0x00000007ac3b1970 (a java.lang.Object)
  - locked 0x00000007ac3b1980 (a java.lang.Object)
  Locked ownable synchronizers:
    - None

"Thread-0" prio=6 tid=0x0000000007318800 nid=0x7da14 waiting for monitor entry
[0x000000000883f000]
  java.lang.Thread.State: BLOCKED (on object monitor)
  at com.tier1app.DeadLockSimulator$FirstThread.run(DeadLockSimulator.java:16)
  - waiting to lock 0x00000007ac3b1980 (a java.lang.Object)
  - locked 0x00000007ac3b1970 (a java.lang.Object)
  Locked ownable synchronizers:
    - None
```



# THREAD DUMP ANALYSIS PATTERN – TREADMILL

# 18

## Description

You might have experienced the application's CPU to spike up suddenly & spike wouldn't go down until JVM is recycled. You restart the JVM, after certain time period CPU consumption would once again start to spike up. Then you will have to recycle the JVM once again. Have you experienced it? If you have smile on your face now, then it's certain you would have experienced this problem.

This type of problem typically happens when thread spins on an infinite loop. A thread would be spinning infinitely when one of the issues described in this article happens.

To diagnose these sort of problems, you would have to capture 3 thread dumps in an interval of 10 seconds. In between those thread dumps, if there are threads

- a. on the same method (or one the same line of code)
- b. they are in 'RUNNABLE' state,

then those are the threads which are causing CPU to spike up. Investigating the stack trace of those threads will tell the exact method (or line of code), where threads are spinning. Fixing that particular method (or line of code) would resolve the problem.

## Example

HashMap isn't threaded safe implementation. When multiple threads try to access HashMap's get() and put() APIs concurrently it would cause threads go into infinite looping. This problem doesn't happen frequently, but it does happen.

Below is an excerpt from a thread dump which indicates the infinite looping that is happening in HashMap:

```
"Thread-0"; prio=6 tid=0x00000000b583000 nid=0x10adc runnable
[0x00000000cb6f000]
  java.lang.Thread.State: RUNNABLE
    at java.util.HashMap.put(HashMap.java:374)
    at
  com.tier1app.HashMapLooper$AddForeverThread.AddForever(NonTerminatingLooper.java:32)
    at com.tier1app.HashMapLooper$AddForeverThread.method2(NonTerminatingLooper.java:27)
    at com.tier1app.HashMapLooper$AddForeverThread.method1(NonTerminatingLooper.java:22)
    at com.tier1app.NonTerminatingLooper$LoopForeverThread.run(NonTerminatingLooper.java:16)
```

Across all the 3 thread dumps “Thread-0” was always exhibiting same stack trace. i.e. it was always in the `java.util.HashMap.put(HashMap.java:374)` method. This problem was addressed by replacing the `HashMap` with `ConcurrentHashMap`.

### **Why named as Treadmill?**

In Treadmill, one would keep running without moving forward. Similarly, when there is infinite looping, CPU consumption goes high without progress in the code execution path. Thus it's named as 'Treadmill' pattern.

# THREAD DUMP ANALYSIS PATTERN –ATHEROSCLEROSIS

19

## Description

If threads are blocking momentarily, then it's not a problem. However, if they are blocking for a prolonged period, then it's of concern. It's indicative of some problem in the application. Typically blocked threads would make application unresponsive.

Threads that remain in the same method and in 'BLOCKED' state between 3 threads dump which are captured in an interval of 10 seconds can turn out to be problematic ones. Studying the stack traces of such blocked threads would indicate the reasons why they are blocked. Reasons may include: deadlocks, circular deadlocks, another thread, would have acquired the locked and never released it, external SORs could have become unresponsive ...

## Example

Following is the excerpt of a thread dump that was captured from a major SOA application, which became unresponsive. The thread `ajp-bio-192.168.100.128-9022-exec-173` remained in BLOCKED state for 3 consecutive thread dumps which were captured in an interval of 10 seconds. Here goes the important parts of Stack trace of this thread:

```
ajp-bio-192.168.100.128-9022-exec-173
Stack Trace is:
java.lang.Thread.State: BLOCKED (on object monitor)
at
** *.sp.dao.impl.ReferenceNumberGeneratorDaoImpl.getNextItineraryReferenceNumber(Referen
ceNumberGeneratorDaoImpl.java:55)
- waiting to lock 0x00000006afaa5a60 (a
** *.sp.dao.impl.ReferenceNumberGeneratorDaoImpl)
at sun.reflect.GeneratedMethodAccessor3112.invoke(Unknown Source)
at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
at java.lang.reflect.Method.invoke(Method.java:601)
at
org.springframework.aop.support.AopUtils.invokeJoinpointUsingReflection(AopUtils.java:307)
:
:
at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1145)
at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:615)
at java.lang.Thread.run(Thread.java:722)
```

Here you can notice that ajp-bio-192.168.100.128-9022-exec-173 stuck in the method `**.*.sp.dao.impl.ReferenceNumberGeneratorDaoImpl.getNextItineraryReferenceNumber(ReferenceNumberGeneratorDaoImpl.java:55)`. This thread got stuck in this method, because another thread ajp-bio-192.168.100.128-9022-exec-84 after obtaining the lock `0x00000006afaa5a60`, it never returned back. Below goes the stack trace of ajp-bio-192.168.100.128-9022-exec-84 thread

## Why named as Atherosclerosis?

Atherosclerosis is a heart disease. Medically it's defined as the following: the inside walls of human arteries are normally smooth and flexible, allowing blood to flow through them easily. Fatty deposits, or plaques, may build up inside the arterial wall. These plaques narrow the artery and can reduce or even completely stop the flow of blood, leading to death.

Similarly if blocking of a thread prolongs and happens across multiple threads, then it would make the application unresponsive, eventually, it has to be killed.

# THREAD DUMP ANALYSIS PATTERN – TRAFFIC JAM

20

## Description

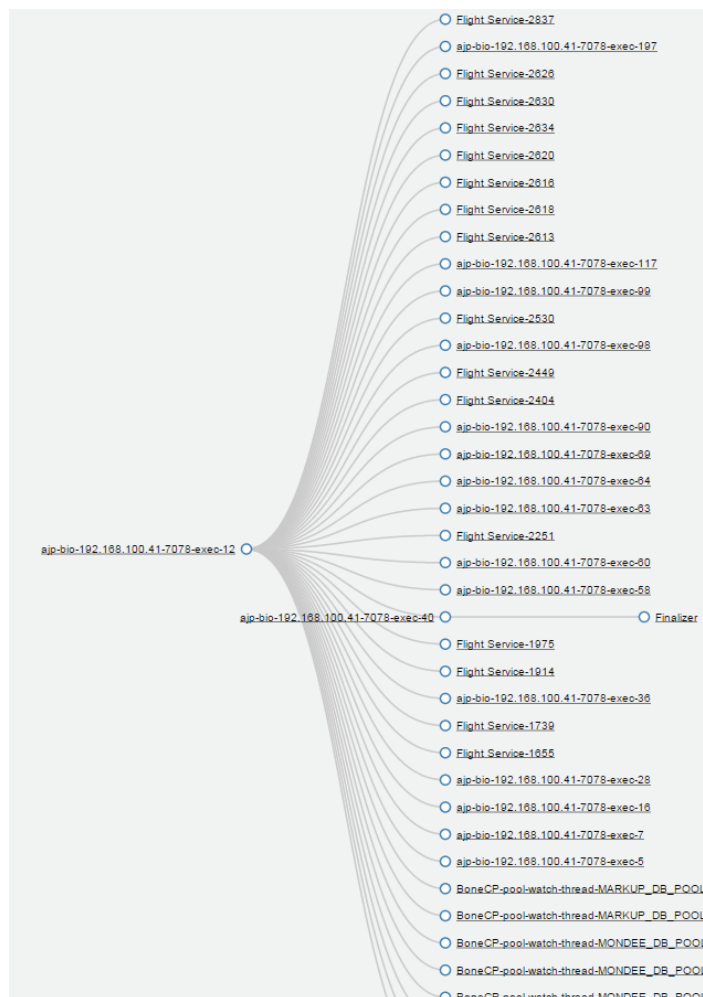
Thread-A could have acquired the lock-1 and then would never release it. Thread-B could have acquired lock-2 and waiting on this lock-1. Thread-C could be waiting to acquire lock-2. This kind of transitive blocks between threads can make entire application unresponsive. See the real-world example below.

## Example

Below is a real-world example taken from a major travel application. Here ‘Finalizer’ thread was waiting for a lock that was held by ‘ajp-bio-192.168.100.41-7078-exec-40’ thread.

ajp-bio-192.168.100.41-7078-exec-40 and several other threads were waiting for the lock which took place by ‘ajp-bio-192.168.100.41-7078-exec-12’ thread.

Thus ‘ajp-bio-192.168.100.41-7078-exec-12’ has transitively blocked 42 threads in total. This ripple effect caused the entire application to become unresponsive. Apparently, it turned out ‘ajp-bio-192.168.100.41-7078-exec-12’ was blocked indefinitely because of a bug in an APM monitoring agent (Ruxit). Upgrading the agent version to latest version resolved the problem. This is quite an irony because – APM monitoring agents are meant to prevent/isolate these sort of issues, but in this case, they themselves are causing the issue. It’s like a law enforcement breaking the laws.



## Why named as Traffic Jam?

Traffic Jam typically happens when there is an accident in the front. Due to that, all the cars that are following the front car will also get stranded. This is very analogous to the transitive blocks behavior described here.

# THREAD DUMP ANALYSIS PATTERN – REPETITIVE STRAIN INJURY (RSI)



## Description

When there is a performance bottleneck in the application, most of the threads will start to accumulate on that problematic bottleneck area. Those threads will have same stack trace. Thus whenever a significant number of threads exhibit identical/repetitive stack trace then those stack trace should be investigated. It may be indicative of performance problems.

Here are few such scenarios:

1. Say your SOR or external service is slowing down then a significant number of threads will start to wait for its response. In such circumstance, those threads will exhibit same stack trace.
2. Say a thread acquired a lock & it never released then, then several other threads which are in the same execution path will get into the blocked state, exhibiting same stack trace.
3. If a loop (for loop, while loop, do..while loop) condition doesn't terminate then several threads which execute that loop will exhibit the same stack trace.

When any of the above scenarios occurs application's performance, and availability will be questioned.

## Example

Below is an excerpt from a thread dump of a major B2B application. This application was running fine, but all of a sudden it became unresponsive. Thread dump from this application was captured. It revealed that 225 threads out of 400 threads were exhibiting same stack trace. Here goes that stack trace:

```

"ajp-bio-192.168.100.128-9022-exec-79" daemon prio=10 tid=0x00007f4d2001c000 nid=0x1d1c
waiting for monitor entry [0x00007f4ce91fa000]
  java.lang.Thread.State: BLOCKED (on object monitor)
  at
  com.xxxxxxxx.xx.xxx.xxx.ReferenceNumberGeneratorDaoImpl.getNextItineraryReferenceNumb
er(ReferenceNumberGeneratorDaoImpl.java:55)
  - waiting to lock 0x00000006afaa5a60
  (acom.xxxxxxxx.sp.dao.impl.ReferenceNumberGeneratorDaoImpl)
  at sun.reflect.GeneratedMethodAccessor3112.invoke(Unknown Source)
  at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
  at java.lang.reflect.Method.invoke(Method.java:601)
  at org.springframework.aop.support.AopUtils.invokeJoinpointUsingReflection(AopUtils.java:307)
  at
  org.springframework.aop.framework.ReflectiveMethodInvocation.invokeJoinpoint(ReflectiveMet
hodInvocation.java:182)
  at
  org.springframework.aop.framework.ReflectiveMethodInvocation.proceed(ReflectiveMethodInv
ocation.java:149)
  at
  org.springframework.orm.hibernate3.HibernateInterceptor.invoke(HibernateInterceptor.java:111)
  at
  org.springframework.aop.framework.ReflectiveMethodInvocation.proceed(ReflectiveMethodInv
ocation.java:171)
  at
  org.springframework.aop.framework.JdkDynamicAopProxy.invoke(JdkDynamicAopProxy.java:2
04)
  at com.sun.proxy.$Proxy36.getNextItineraryReferenceNumber(Unknown Source)
  at
  com.xxxxxxxx.xx.xxxxxxxx.xxx.ReferenceNumberGeneratorServiceImpl.getNextItineraryReferenc
eNumber(ReferenceNumberGeneratorServiceImpl.java:15)
  :
  :

```

From the stack trace, you can infer that thread was blocked and waiting for the lock on the object 0x00000006afaa5a60. 225 such threads were waiting to obtain lock on this same object. It's definitely a bad sign. It's a clear indication of thread starvation.

Apparently this lock was held by "ajp-bio-192.168.100.128-9022-exec-84". Below is the stack trace this thread. You can notice that this thread acquired the lock on the object 0x00000006afaa5a60, but after acquiring the lock, it got stuck waiting for response from the database. Apparently for this application database timeout wasn't set. Due to that this thread's database call never returned back. Due to that 225 other threads were stuck. Thus application became unresponsive.

After setting proper database time out value, this problem went away.

## Why named as RSI?

A Repetitive Strain Injury (RSI) happens when you exercise your body parts (hand, fingers, wrist, neck,...) repeatedly in a wrong posture. Similarly when there is a performance bottleneck, multiple threads will start to exhibit the stack trace again & again. Those stack trace should be studied in detail.



# THREAD DUMP ANALYSIS PATTERN – ADDITIVES

## 22

### Description

It's highly recommended to capture 3 threads dumps in an interval of 10 seconds to uncover any problem in the JVM. If in the 2nd and 3rd thread dump if additional threads start to go into a particular state, then those threads and their stack traces have to be studied in detail. It may or may not be indicative of certain problem in the application, but definitely, a good lead to follow.

### Example

This problem surfaced because of a thread leak in Oracle JDBC Driver when ONS feature was turned ON. This problem happened in an old version of Oracle JDBC Driver (almost in 2011). Because of the bug in the driver, under certain scenarios, it started to spawn tonnes of new threads. In every captured thread dump new threads in RUNNABLE stated with below stack trace got added. Around 1700 threads with the same stack trace got created.

```
Thread-6805 - priority:8 - threadId:0x07768000 - nativeId:10966 - addressSpace:null -  
state:RUNNABLE  
stackTrace:  
java.lang.Thread.State: RUNNABLE  
at java.net.SocketInputStream.socketRead0(Native Method)  
at java.net.SocketInputStream.read(SocketInputStream.java:155)  
at java.net.SocketInputStream.read(SocketInputStream.java:121)  
at oracle.ons.InputBuffer.readMoreData(InputBuffer.java:268)  
at oracle.ons.InputBuffer.getNextString(InputBuffer.java:223)  
at oracle.ons.ReceiverThread.run(ReceiverThread.java:266)
```

### Why named as additives?

Additives are defined as 'a substance added to something in small quantities'. Similarly, this pattern talks about addition of new threads to an existing state.

# THREAD DUMP ANALYSIS PATTERN – LEPRECHAUN TRAP

23

## Description

Objects that have `finalize()` method are treated differently during Garbage collection process than the ones which don't have them. During garbage collection phase, object with `finalize()` aren't immediately evicted from the memory. Instead as first step, those objects are added to an internal queue of `java.lang.ref.Finalizer` object. There is a low priority JVM thread by name 'Finalizer' that executes `finalize()` method of each object in the queue. Only after the execution of `finalize()` method, object becomes eligible for Garbage Collection. Because of poor implementation of `finalize()` method if Finalizer thread gets blocked then it will have a severe detrimental cascading effect on the JVM.

If Finalizer thread gets blocked then internal queue of `java.lang.ref.Finalize` will start to grow. It would cause JVM's memory consumption to grow rapidly. It would result in `OutOfMemoryError`, jeopardizing entire JVM's availability. Thus when analyzing thread dumps it's highly recommended to study the stack trace of Finalizer thread.

## Example

Here is a sample stack trace of a Finalizer thread which got blocked in a `finalize()` method:

```
"Finalizer" daemon prio=10 tid=0x000072dc32b000 nid=0x7a21 waiting for monitor entry
[0x000072cdcb6000]
  java.lang.Thread.State: BLOCKED (on object monitor)
  at net.sourceforge.jtds.jdbc.JtdsConnection.releaseTds(JtdsConnection.java:2024)
  - waiting to lock 0x00000007d50d98f0 (a net.sourceforge.jtds.jdbc.JtdsConnection)
  at net.sourceforge.jtds.jdbc.JtdsStatement.close(JtdsStatement.java:972)
  at net.sourceforge.jtds.jdbc.JtdsStatement.finalize(JtdsStatement.java:219)
  at java.lang.ref.Finalizer.invokeFinalizeMethod(Native Method)
  at java.lang.ref.Finalizer.runFinalizer(Finalizer.java:101)
  at java.lang.ref.Finalizer.access$100(Finalizer.java:32)
  at java.lang.ref.Finalizer$FinalizerThread.run(Finalizer.java:178)
```

Above stack trace was captured from a JVM which was using one of the older versions of JTDS JDBC Driver. Apparently this version of driver had an issue; you can see `finalize()` method in the `net.sourceforge.jtds.jdbc.JtdsStatement` object calling `JtdsConnection#releaseTds()` method. Apparently, this method got blocked and never returned back. Thus Finalizer thread got stuck indefinitely in the `JtdsConnection#releaseTds()` method. Due to that Finalizer thread wasn't able to work on the other objects that had `finalize()` method. Due to that application started to suffer from `OutOfMemoryError`. In the latest version of JTDS JDBC Driver this issue was fixed. Thus when you are implementing `finalize()` method be very careful.

## Why named as Leprechaun Trap?

Kids in western countries build Leprechaun Trap as part of St. Patrick's day celebration. Leprechaun is a fairy character, basically a very tiny old man, wearing a green coat & hat who is in search for gold coins. Kids build creative traps for this Laprechaun, luring him with gold coins. Similarly anxious Finalizer thread is always in search of objects that has finalize() method to execute them. In case if finalize() method is wrongly implemented, it can trap the Finalizer thread. Because of this similarity we have named it as Leprechaun Trap.

# STACKOVERFLOWERROR: CAUSES & SOLUTIONS

# 24

StackOverflowError is one of the common confronted JVM error. In this blog post, let's learn inner mechanics of thread stacks, reasons that can trigger StackOverflowError and potential solutions to address this error.

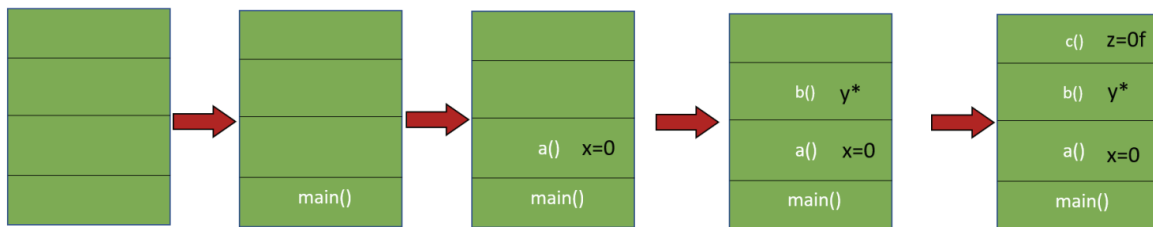
To gain deeper understanding into StackOverflowError, let's review this simple program:

```
public class SimpleExample {  
    public static void main(String args[]) {  
        a();  
    }  
    public static void a() {  
        int x = 0;  
        b();  
    }  
    public static void b() {  
        Car y = new Car();  
        c();  
    }  
    public static void c() {  
        float z = 0f;  
        System.out.println("Hello");  
    }  
}
```

This program is very simple with the following execution code:

- main() method is invoked first
- main() method invokes a() method. Inside a() method integer variable 'x' is initialized to value 0.
- a() method in turn invokes b() method. Inside b() method Car object is constructed and assigned to variable 'y'.
- b() method in turn invokes c() method. Inside c() method float variable 'z' is initialized to value 0.

Now let's review what happens behind the scenes when above simple program is executed. Each thread in the application has its own stack. Each stack has multiple stack frames. Thread adds the methods it's executing, primitive data types, object pointers, return values to its stack frame in the sequence order in which they are executed.



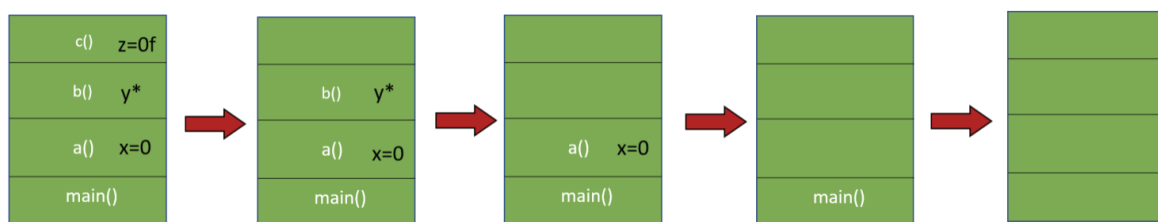
In step #1: main() method is pushed into the application thread's stack.

In step #2: a() method is pushed into application thread's stack. In a() method, primitive data type 'int' is defined with value 0 and assigned to variable x. This information is also pushed into the same stack frame. Note both data i.e. '0' and variable 'x' is pushed into thread's stack frame.

In step #3: b() method is pushed into thread's stack. In b() method, 'Car' object is created and assigned to variable 'y'. Crucial point to note here is 'Car' object is created in the heap and not in the thread's stack. Only Car object's reference i.e. y is stored in the thread's stack frame.

In step #4: c() method is pushed into thread's stack. In c() method, primitive data type 'float' is defined with value 0f and assigned to variable z. This information is also pushed into same stack frame. Note both data i.e. '0f' and variable 'z' is pushed into thread's stack frame.

Once each method's execution is completed, then method and the variables/object pointers which are stored in the stack frame are removed as shown in Fig 2.



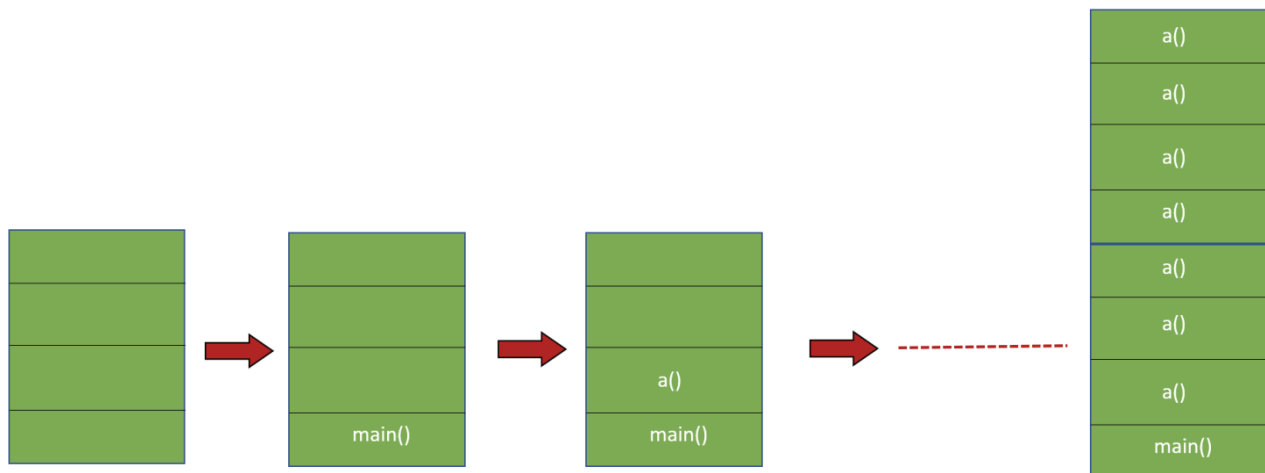
## What causes StackOverflowError?

As you can see thread's stack is storing methods it's executing, primitive datatypes, variables, object pointers, and return values. All of these consume memory. If thread's stack sizes grow beyond the allocated memory limit then StackOverflowError is thrown. Let's look at the below buggy program, which will result in StackOverflowError:

```
public class SOFDemo {
    public static void a() {
        // Buggy line. It will cause method a() to be called infinite
        number of times.
        a();
    }
    public static void main(String args[]) {
        a();
    }
}
```

In this program `main()` method invokes `a()` method. `a()` method recursively calls itself. This implementation will cause `a()` method to be invoked infinite number of times. In this circumstance `a()` method will be added to thread's stack frame infinite number of times. Thus, after a few thousand iterations thread's stack size limit would be exceeded. Once stack size limit is exceeded it will result in 'StackOverflowError':

```
Exception in thread "main" java.lang.StackOverflowError
  at com.buggyapp.stackoverflow.SOFDemo.a(SOFDemo.java:19)
  at com.buggyapp.stackoverflow.SOFDemo.a(SOFDemo.java:19)
  at com.buggyapp.stackoverflow.SOFDemo.a(SOFDemo.java:19)
  at com.buggyapp.stackoverflow.SOFDemo.a(SOFDemo.java:19)
  at com.buggyapp.stackoverflow.SOFDemo.a(SOFDemo.java:19)
  at com.buggyapp.stackoverflow.SOFDemo.a(SOFDemo.java:19)
  at com.buggyapp.stackoverflow.SOFDemo.a(SOFDemo.java:19)
  at com.buggyapp.stackoverflow.SOFDemo.a(SOFDemo.java:19)
  at com.buggyapp.stackoverflow.SOFDemo.a(SOFDemo.java:19)
  at com.buggyapp.stackoverflow.SOFDemo.a(SOFDemo.java:19)
  at com.buggyapp.stackoverflow.SOFDemo.a(SOFDemo.java:19)
  at com.buggyapp.stackoverflow.SOFDemo.a(SOFDemo.java:19)
```



## What are the solutions to StackOverflowError?

There are a couple of strategies to address StackOverflowError.

### 1. Fix the code

Most of the time because of a non-terminating recursive calls (as shown in the above example), threads stack size can grow to a large size. In those circumstances, you must fix the source code which is causing recursive looping. When 'StackOverflowError' is thrown, it will print the stack trace of the code that it was recursively executing. This code is a good pointer to start debugging and fixing the issue.

## 2. Increase Thread Stack Size (-Xss)

There might be legitimate reason where a threads stack size needs to be increased. May be thread has to execute a large number of methods, lot of local variables/created. In such circumstance, you can increase the thread's stack size using the JVM argument: '-Xss'. This argument needs to be passed when you start the application.Example:

```
-Xss2m
```

This will set the thread's stack size to 2 mb.

It might bring a question, what is the default thread's stack size? Default thread stack size varies based on your operating system, java version & vendor.

JVM version	Thread stack size
Sparc 32-bit JVM	512k
Sparc 64-bit JVM	1024k
x86 Solaris/Linux 32-bit JVM	320K
x86 Solaris/Linux 64-bit JVM	1024K
Windows 32-bit JVM	320K
Windows 64-bit JVM	1024K

# HOW TO TROUBLESHOOT CPU PROBLEMS?

# 25

Diagnosing and troubleshooting CPU problems in production that too in cloud environment can become tricky and tedious. Your application might have millions of lines of code, trying to identify the exact line of code that is causing the CPU to spike up, might be equivalent of finding a needle in the haystack. In this article, let's learn how to find that needle (i.e. CPU spiking line of code) in a matter of seconds/minutes.

To help readers better understand this troubleshooting technique, we built a sample application and deployed it into AWS EC2 instance. Once this application was launched, it caused CPU consumption to spike up to 199.1%. Now let's walk you through the steps that we followed while troubleshooting this problem. Basically, there are 3 simple steps:

1. Identify threads that consume CPU
2. Capture thread dumps
3. Identify lines of code that is causing CPU to spike up

## 1. Identify threads that are causing CPU to spike

In the EC2 instance, multiple processes could be running. The first step is to identify the process that is causing the CPU to spike up. Best way to do is to use the 'TOP' command that is present in \*nix flavor of operating systems.

Issue command 'top' from the console

```
$ top
```

This command will display all the processes that are running in the EC2 instance sorted by high CPU consuming processes displayed at the top. When we issued the command in the EC2 instance we were seeing the below output:

```
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
5	root	0	-20	0	0	0	S	0.0	0.0	0:00.00	kworker/0:0H
6	root	20	0	0	0	0	S	0.0	0.0	1:27.11	kworker/u30:0
7	root	20	0	0	0	0	S	0.0	0.0	3:22.96	rcu_sched
8	root	20	0	0	0	0	S	0.0	0.0	0:00.00	rcu_bh
9	root	RT	0	0	0	0	S	0.0	0.0	0:01.96	migration/0
10	root	RT	0	0	0	0	S	0.0	0.0	0:01.44	migration/1
11	root	20	0	0	0	0	S	0.0	0.0	0:14.63	ksoftirqd/1
13	root	0	-20	0	0	0	S	0.0	0.0	0:00.00	kworker/1:0H
14	root	20	0	0	0	0	S	0.0	0.0	0:00.00	kdevtmpfs
15	root	0	-20	0	0	0	S	0.0	0.0	0:00.00	netns
18	root	0	-20	0	0	0	S	0.0	0.0	0:00.00	perf
22	root	20	0	0	0	0	S	0.0	0.0	0:00.00	xenwatch
23	root	20	0	0	0	0	S	0.0	0.0	0:00.00	xenbus
134	root	20	0	0	0	0	S	0.0	0.0	0:03.71	khungtaskd
135	root	0	-20	0	0	0	S	0.0	0.0	0:00.00	writeback

Fig:'top' command issued from an AWS EC2 instance



From the output, you can notice process# 31294 to be consuming 199.1% of CPU. It's pretty high consumption. Ok, now we have identified the process in the EC2 instance which is causing the CPU to spike up. Next step is to identify the threads with in this process that is causing the CPU to spike up.

Issue command 'top -H -p {pid}' from the console. Example

```
$ top -H -p 31294
```

From the output you can notice:

This command will display all the threads that are causing the CPU to spike up in this particular 31294 process. When we issued this command in the EC2 instance, we were seeing the below output:

```
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
31296	ec2-user	20	0	4415m	22m	14m	S	0.0	0.3	0:00.00	java
31297	ec2-user	20	0	4415m	22m	14m	S	0.0	0.3	0:00.00	java
31298	ec2-user	20	0	4415m	22m	14m	S	0.0	0.3	0:00.00	java
31299	ec2-user	20	0	4415m	22m	14m	S	0.0	0.3	0:00.00	java
31300	ec2-user	20	0	4415m	22m	14m	S	0.0	0.3	0:00.00	java
31301	ec2-user	20	0	4415m	22m	14m	S	0.0	0.3	0:00.00	java
31302	ec2-user	20	0	4415m	22m	14m	S	0.0	0.3	0:00.00	java
31303	ec2-user	20	0	4415m	22m	14m	S	0.0	0.3	0:00.00	java
31304	ec2-user	20	0	4415m	22m	14m	S	0.0	0.3	0:00.00	java
31305	ec2-user	20	0	4415m	22m	14m	S	0.0	0.3	0:00.02	java

Fig:'top -H -p {pid}' command issued from an AWS EC2 instance

From the output you can notice:

- Thread Id 31306 consuming 69.3%of CPU
- Thread Id 31307 consuming 65.6%of CPU
- Thread Id 31308 consuming 64.0%of CPU
- Remaining all other threads consume negligible amount of CPU.

This is a good step forward, as we have identified the threads that are causing CPU to spike. As the next step, we need to capture thread dumps so that we can identify the lines of code that is causing the CPU to spike up.

## 2. Capture thread dumps

A thread dump is a snapshot of all threads that are present in the application. Thread state, stack trace (i.e. code path that thread is executing), thread Id related information of each thread in the application is reported in the thread dump.

There are 8 different options to capture thread dumps. You can choose the option that is convenient for you. One of the simplest options to take thread dump is to use tool 'jstack' which is packaged in JDK. This tool can be found in \$JAVA\_HOME/bin folder. Below is the command to capture thread dump:

```
jstack -l {pid} > {file-path}
```

where

pid: is the process Id of the application, whose thread dump should be captured

file-path: is the file path where thread dump will be written in to.

Example:

```
jstack-l 31294 > /opt/tmp/threadDump.txt
```

As per the example, thread dump of the process would be generated in /opt/tmp/threadDump.txt file.

### 3. Identify lines of code that is causing CPU to spike up

Next step is to analyze the thread dump to identify the lines of code that is causing the CPU to spike up. We would recommend analyzing thread dumps through fastThread, a free online thread dump analysis tool.

Now we uploaded captured thread dump to fastThread tool. Tool generated this beautiful visual report. Report has multiple sections. On the right top corner of the report, there is a search box. There we entered the Ids of the threads which were consuming high CPU. Basically, thread Ids that we identified in step #1 i.e. '31306,31307, 31308'.

fastThread tool displayed all these 3 threads stack trace as shown below.

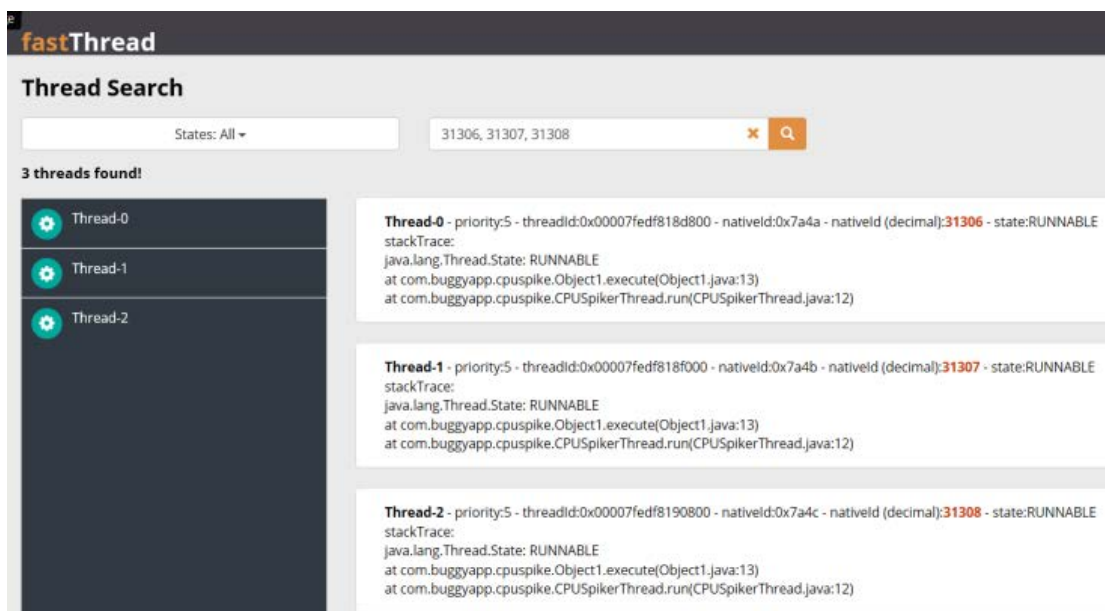


Fig: fastThread tool displaying CPU consuming thread

You can notice all the 3 threads to be in RUNNABLE state and executing this line of code:

```
com.buggyapp.cpuspike.Object1.execute(Object1.java:13)
```

Apparently following is the application source code

```
package com.buggyapp.cpuspike;
public class Object1 {
    public static void execute() {
        while (true) {
            doSomething();
        }
    }
    public static void doSomething() {
    }
}
```

You can see line #13 in object1.java to be 'doSomething()'. You can see that 'doSomething()' method to do nothing, but it is invoked an infinite number of times because of non-terminating while loop inline# 11. If a thread starts to a loop infinite number of times, then CPU will start to spike up. That is what exactly happening in this sample program. If non-terminating loop in line #11 is fixed, then this CPU spike problem will go away.

## Conclusion

To summarize first we need to use 'TOP' tool to identify the thread ids that are causing the CPU spike up, then we need to capture the thread dumps, next step is to analyze thread dumps to identify exact lines of code that is causing CPU to spike up. Enjoy troubleshooting, happy hacking!

# HEAP DUMP ANALYSIS API

26

Android, JVM Heap dump analysis doesn't have to be done manually (painfully) anymore. You can programmatically analyze Heap dumps through our REST API. Below are the few use cases where our heap dump analysis REST APIs used by major enterprises.

## Use Case 1: CI/CD pipeline

As part of continuous integration pipeline, several mature engineering organizations are executing performance tests. As part of this stress tests, they capture heap dumps from the application. Captured heap dumps are analyzed through our heap dump analysis REST API. If API identifies any memory vulnerabilities or certain values (like Object count, or wasted memory size) goes beyond a threshold, then the entire build is failed.

## Use Case 2: Production root cause analysis

Whenever application experiences any memory problems or `OutOfMemoryError`, then heap dumps are captured from the application to analyze. Captured heap dumps can be programmatically analyzed through heap dump analysis REST API. Root cause analysis can be instantly identified.

## Use Case 3: Analyzing multiple dumps instantly

Enterprises have multiple applications. It's hard to analyze each applications heap dump manually on periodic basis, both in production and test environment. It's both time consuming and tedious process. In such circumstances, they configure cron job that will capture heap dumps on a periodic basis and analyze heap dumps through our REST API. They configure alerts in case if heap dump thresholds drop below certain values.

## How to invoke Heap Dump Analysis API?

Invoking the Heap Dump API is very simple:

1. You will have to Register with us. We will email you the API key. This is the one-time setup process. Note: if you have purchased enterprise version with API, you don't have to worry about registration. The API Key will be provided to you as part of installation instructions.
2. POST HTTP request to `http://api.heaphero.io/analyze-hd-api?apiKey={API_KEY_SENT_IN_EMAIL}`
3. The body of the HTTP request should contain the Heap Dump that needs to be analyzed.
4. HTTP response will be sent back in JSON format.

```
Curl -X POST --data-binary @./my-heap-dump.hprof
http://api.heaphero.io/analyze-hd-api?apiKey={API_KEY_SENT_IN_EMAIL} --header
"Content-Type:text"
```

It cannot get any simpler than that? Isn't it?

Heap Dump are quite large in size. For fast and efficient processing, we recommend you to compress and send the heap dump files. When you are compressing the heap dump, you need to pass 'Content-Encoding' element in the HTTP Header element or in the URL parameter.

Say suppose you are compressing heap dump file in to 'zip' format, then you can invoke the API with HTTP header element

```
Curl -X POST --data-binary @./my-heap-dump.hprof
http://api.heaphero.io/analyze-hd-api?apiKey={API_KEY_SENT_IN_EMAIL} --header
"Content-Encoding:zip"
```

or you can also invoke the API with 'Content-Encoding' element in the URL parameter

```
Curl -X POST --data-binary @./my-heap-dump.hprof
http://api.heaphero.io/analyze-hd-api?apiKey={API_KEY_SENT_IN_EMAIL}&Content-Encoding=zip
```

We support following compression formats. You may use the one of your choice:

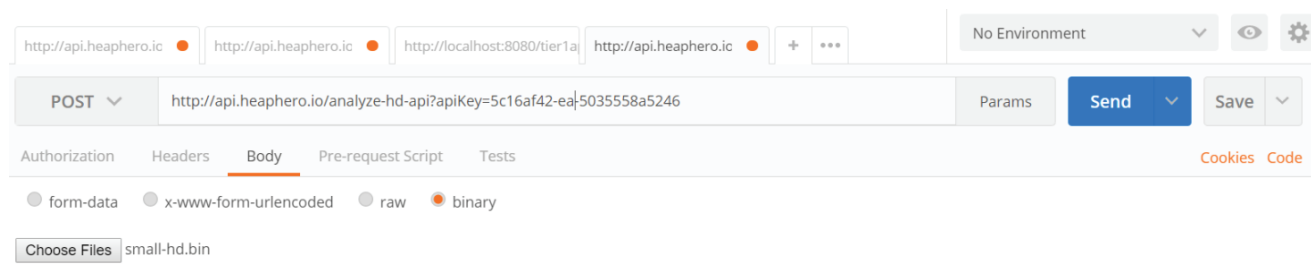
zip, gz, xz, z, lzma, deflate, sz, lz4, zstd, bz2, tar

Whatever compression format you used for compressing the heap dump should be passed in 'Content-Encoding' element.

## Postman

You can also invoke API using the POSTMAN, SOAP UI,... sort of tools. Below are the steps to invoke HeapHero API through Postman.

a. Select the 'POST' option b. Enter the URL to be 'http://api.heaphero.io/analyze-hd-api?apiKey={API\_KEY\_SENT\_IN\_EMAIL}' c. Select 'Body' option. d. Select 'binary' radio button e. Click on 'Choose Files' button and select your heap dump file f. Now click on 'Send' button. You will see the JSON API response



**Sample Response:**

```
{
  "totalSize": "235.94kb",
  "objectCount": "4,324",
  "classCount": "470",
  "threadCount": 3,
  "wastedMemoryPercentage": 0,
  "largeClasses": [
    {
      "name": "String",
      "percentage": "39.35761589403973",
      "size": "92.86kb",
      "count": "1,121"
    },
    {
      "name": "char[]",
      "percentage": "15.794701986754967",
      "size": "37.27kb",
      "count": "1,135"
    },
    {
      "name": "byte[]",
      "percentage": "10.341059602649006",
      "size": "24.4kb",
      "count": "8"
    },
    {
      "name": "Object[]",
      "percentage": "10.188741721854305",
      "size": "24.04kb",
      "count": "511"
    },
    {
      "name": "java.lang.reflect.Field",
      "percentage": "2.3543046357615895",
      "size": "5.55kb",
      "count": "79"
    }
  ],
  "largeObjects": [
    {
      "percentage": 17.589403,
      "size": "41.5kb"
    },
    {
      "objectName": "java.lang.System",
      "percentage": 10.370861,
```

# HOW TO CAPTURE HEAP DUMP FROM ANDROID APP? – 3 OPTIONS

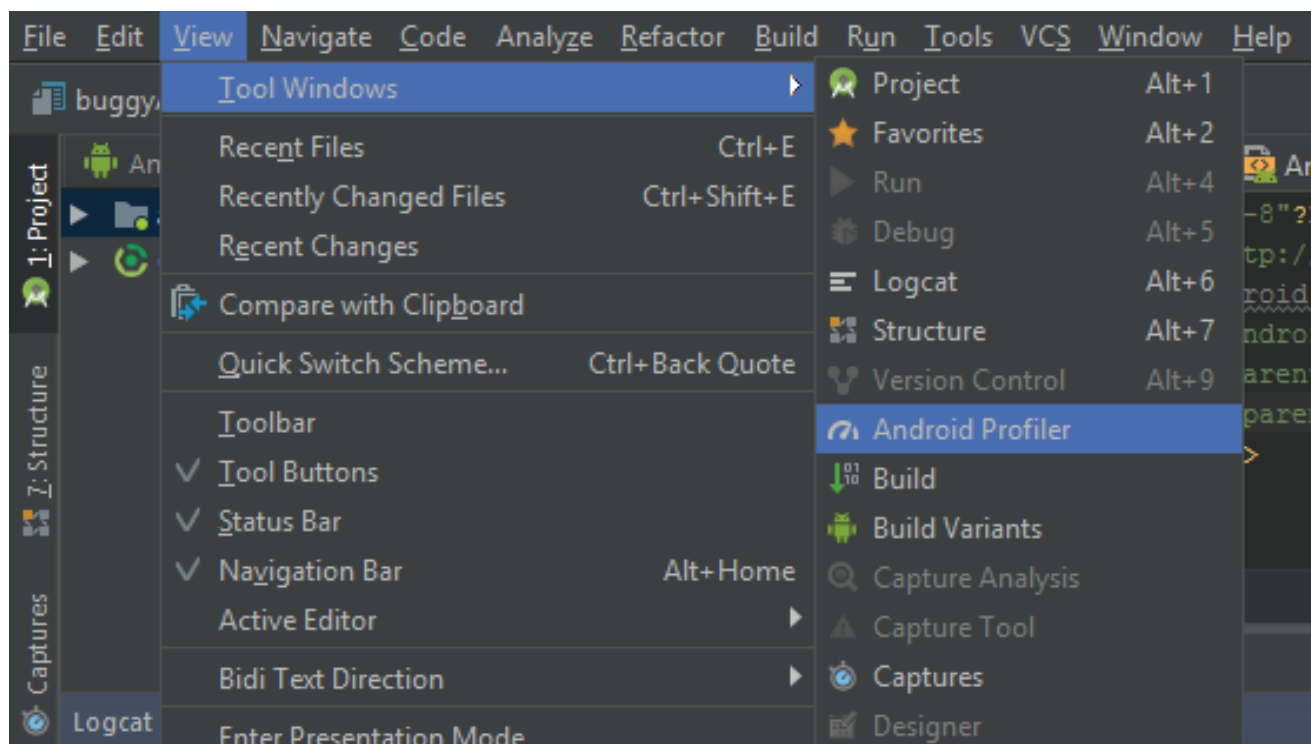
Heap Dumps are vital artifacts to diagnose memory-related problems such as memory leaks, Garbage Collection problems, and `java.lang.OutOfMemoryError`. They are also vital artifacts to optimize memory usage as well.

In this article, we have given few different options to capture Heap Dumps from Android Apps. Once you have captured heap dumps, you can use great tools like HeapHero and Android studio's heap analyzer to analyze heap dumps.

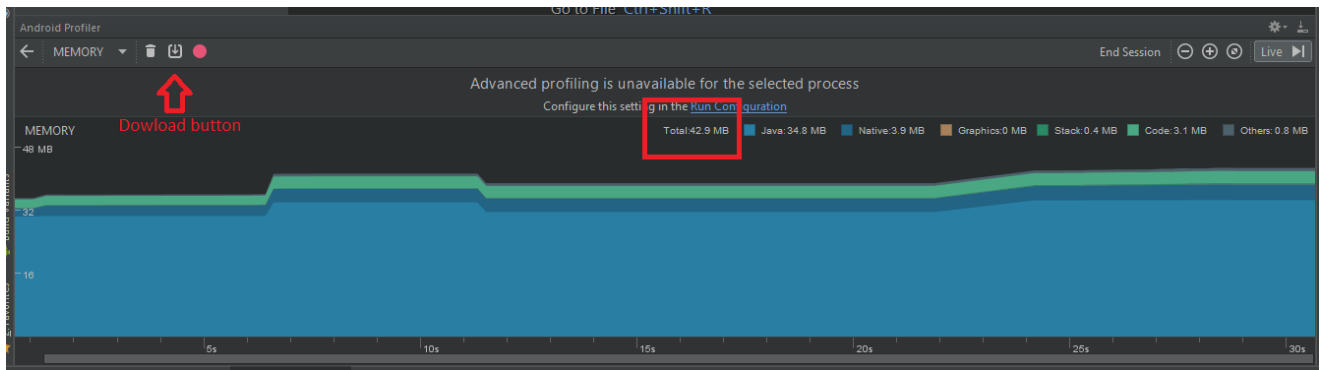
## 1. Memory Profiler

Below are the steps to capture heap dumps from Memory Profiler in Android studio:

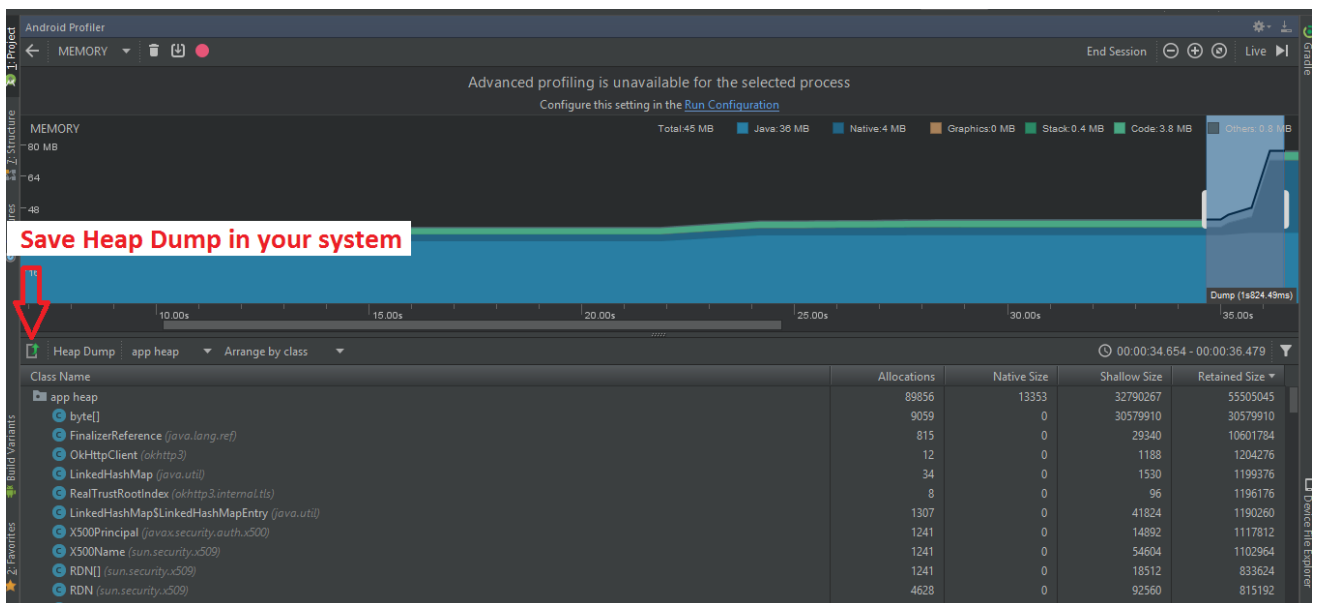
- a. Run the app and select the device you want to profile from Android Studio.
- b. In your Android studio, click on View >> Tool Windows >> Android Profiler



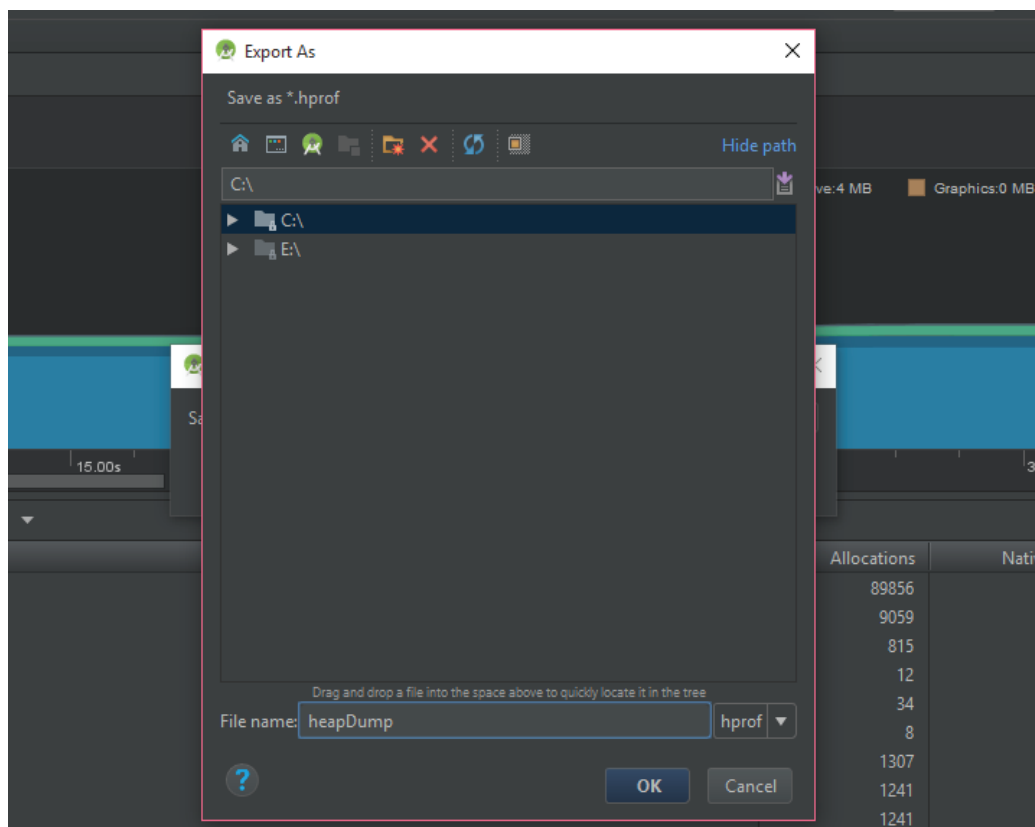
- c. There will be Memory timeline, which would be below the CPU timeline, but above the Network timeline. In this memory timeline, click on download button (highlighted in the below image) to generate heap dump from the Android app.



d. To store heap dump in your system, click on the highlighted icon in the below image.



e. Choose a location to save the generated heap dump file.





## 2. Android Debug Bridge (ADB)

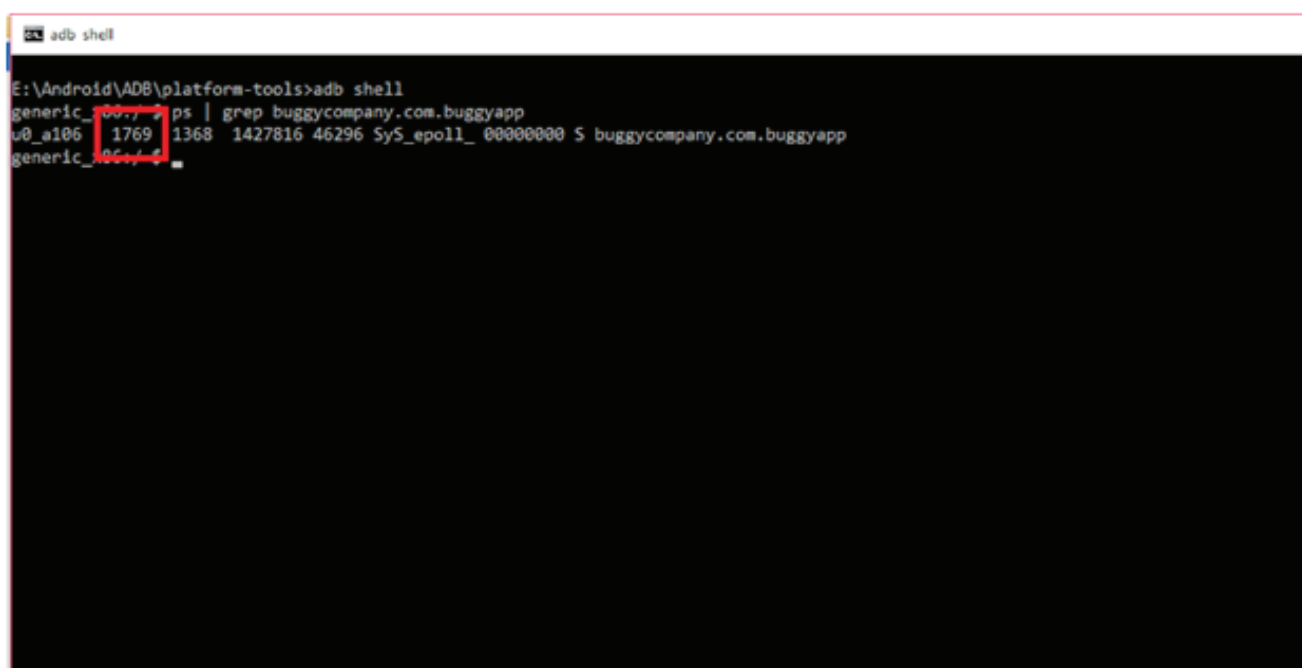
Android Debug Bridge is a command line tool which allows you to interact with a device. ADB provides a variety of device actions, such as installing and debugging apps. It also gives access to the Unix shell to run a variety of commands on the device. You can use this tool to generate android heap dumps. Launch ADB shell and follow the below steps:

### a. Identify your Android App's Process Id

First step is to identify your Android App's process Id. You can do that by issuing below command:

```
adb shell ps | grep <APP-NAME>
```

Above command will return details about the process. The second number will be the PID of your app. Please check the below screenshot.



### b. Create a Heap Dump:

```
adb shell am dumpheap <PID> <HEAP-DUMP-FILE-PATH>
```

PID: Your Android App Process Id

HEAP-DUMP-FILE-PATH: Location where heap dump file should be generated

Example:

```
adb shell am dumpheap 1769 /data/local/tmp/android.hprof
```

### c. Pull the file to your machine

Above step will generate the heap dump file in the device. For analysis, you need to pull the generated file to your machine. You do that by issuing below command:

```
adb pull <HEAP-DUMP-FILE-PATH>
```

HEAP-DUMP-FILE-PATH: Location where heap dump file

Example:

```
db pull /data/local/tmp/android.hprof
```

### 3. Capture Heap Dumps on OutOfMemoryError

If you place the below code in your application, it will capture heap dumps whenever your application receives an OutOfMemoryError.

```
public class CaptureHeapDumps extends Application {
    private static final String FILE_NAME = "heap-dump.hprof";
    @Override
    public void onCreate() {
        super.onCreate();

        Thread.currentThread().setUncaughtExceptionHandler(new OutOfMemoryExceptionHandler());
    }
    @NonNull
    private Thread.UncaughtExceptionHandler OutOfMemoryExceptionHandler() {
        return new Thread.UncaughtExceptionHandler() {
            @Override
            public void uncaughtException(Thread t, Throwable e) {
                String directory = getApplicationInfo().dataDir;
                String absolutePath = new File(directory, FILE_NAME)
                    .getAbsolutePath();
                try {
                    Debug.dumpHprofData(absolutePath);
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        };
    }
}
```

This would generate the heap dump file in this location: /data/user/0/appname/heap-dump.hprof

# DISAPPOINTING STORY ON MEMORY OPTIMIZATION

# 28

Not all stories need to be success stories. Reality is also not like that. We would like to share a true, disappointing story (but a phenomenal learning experience) that may be beneficial to you.

This is a story about optimizing memory utilization of a web application. This application was configured with a lot of memory (4GB) just to service handful of transactions/sec. Thus, we set out to study the memory utilization patterns of this application. We captured heap dumps of this application using 'jmap' tool. We uploaded the captured heap dump to HeapHero tool. HeapHero is a heap dump analysis tool just like Eclipse MAT, JProfiler, Yourkit. HeapHero tool profiled the memory and provided statistics on total classes, total objects, heap size, histogram view of large objects residing in the memory. On top of these traditional metrics, HeapHero reported the total amount of memory wasted due to inefficient programming practices. In modern computing, considerable amount memory is wasted because of inefficient programming practices such as: Duplicate object creation, suboptimal data type definitions (declaring 'double' and assigning only 'float' values), over allocation and underutilization of data structures and several other practices.

This application was no exception to it. HeapHero reported that application is wasting 56% of memory due to inefficient programming practices. Yes, it's eyebrow raising 56%. It reported that 30% of application's memory is wasted because of duplicate strings.

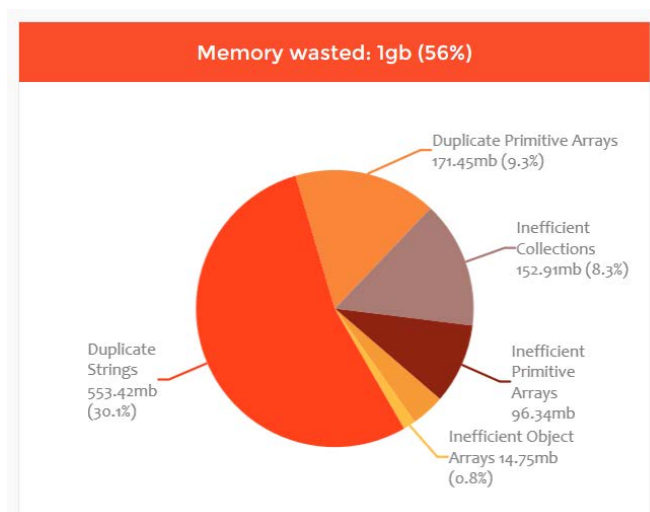


Fig: HeapHero tool reporting amount of memory wasted due to inefficient programming

## String Deduplication

Since Java 8 update 20 a new JVM argument '-XX:+UseStringDeduplication' was introduced. When an application is launched with this argument, JVM will eliminate the duplicate strings from the application's memory during garbage collection. However please be advised that '-XX:+UseStringDeduplication' argument will work only with G1 GC algorithm. You can activate G1 GC algorithm by passing '-XX:+UseG1GC'.

We got excited. We thought just by introducing ‘-XX:+UseG1GC -XX:+UseStringDeduplication’ JVM argument, we would be able to save 30% of memory without any code refactoring. Wow, isn’t it wonderful? To verify this theory, we conducted two different tests in our performance lab:

**Test 1:** Passing ‘-XX:+UseG1GC’

**Test 2:** Passing ‘-XX:+UseG1GC -XX:+UseStringDeduplication’

We enabled Garbage collection logs on the application to study the memory usage pattern. Analyzed Garbage Collection logs using the free online garbage collection log analysis tool – GCEasy. We were hoping that in the test run #2 we would be able to see 30% reduction in the memory consumption, because of elimination of duplicate strings. However, the reality was quite different. We didn’t see any difference in the memory usage. Both test runs were consistently showing the same amount of memory utilization. See the heap usage graphs generated by the GCEasy tool by analyzing the garbage collection logs.

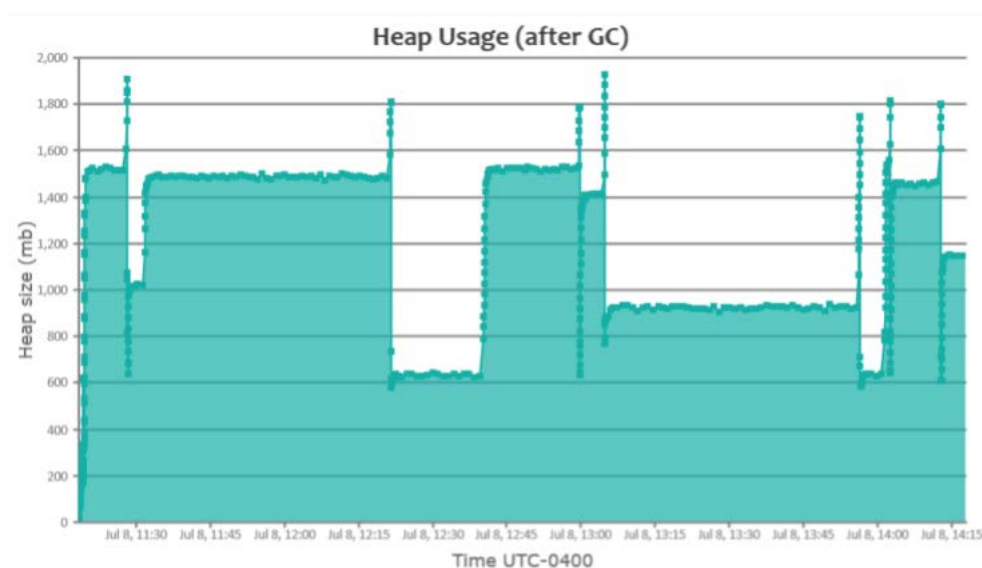


Fig: GCEasy Heap usage graph with ‘-XX:+UseG1GC’

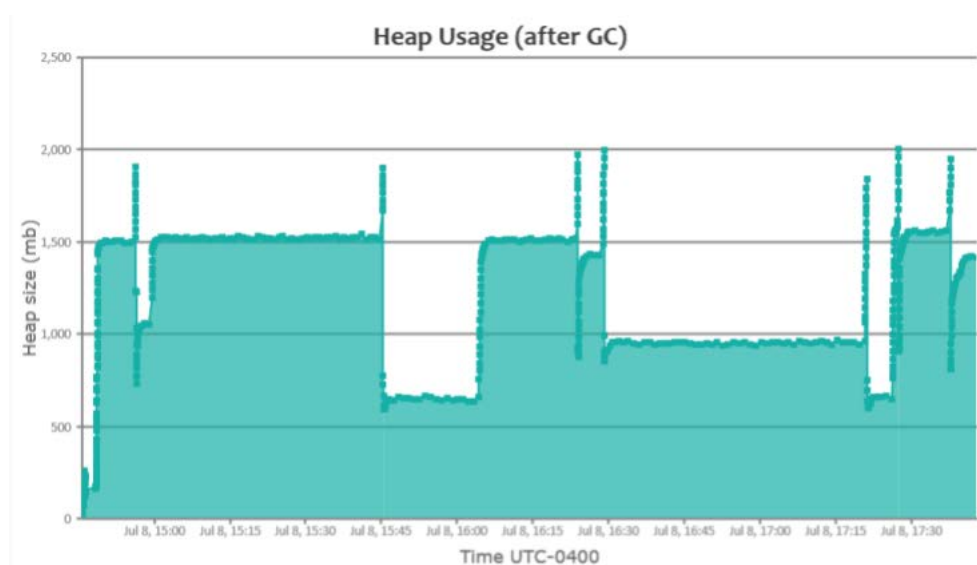


Fig: GCEasy heap usage graph with ‘-XX:+UseG1GC -XX:+UseStringDeduplication’

In Test run #1 heap usage hovering around 1500mb all through the test, in test run #2 also heap usage was hovering around 1500mb. Disappointingly we didn’t see the anticipated 30% reduction in the memory usage, despite introducing ‘-XX:+UseG1GC -XX:+UseStringDeduplication’ JVM arguments.

## Why there wasn't reduction in heap usage?

'Why there wasn't reduction in heap usage' – this question really puzzled us. Did we configure JVM arguments rightly? Doesn't '-XX:+UseStringDeduplication' do its job correctly? Is the analysis report from the GCeasy tool is correct? All these questions troubled our sleep. After detailed analysis, we figured out the bitter truth. Apparently '-XX:+UseStringDeduplication' will eliminate duplicate strings that are present in the old generation of the memory only. It will not eliminate duplicate strings in the young generation. Java memory has 3 primary regions: young generation, old generation, Metaspace. Newly created objects go into the young generation. Objects that survived for longer period are promoted into the old generation. JVM related objects and metadata information are stored in Metaspace. Thus stating in other words '-XX:+UseStringDeduplication' will only remove duplicate strings that are living for a longer period. Since this is a web application, most of the string objects were created and destroyed immediately. It was very clear from the following statistics reported in the GCeasy log analysis report:

Total created bytes ?	466.7 gb
Total promoted bytes ?	9.31 gb
Avg creation rate ?	44.93 mb/sec
Avg promotion rate ?	918 kb/sec

Fig: Object creation/promotion stats reported by GCeasy

Average object creation rate of this application is: 44.93 mb/sec, whereas average promotion rate (i.e. from young generation to old generation) is only 918 kb/sec. It's indicative that very small of the percentage of objects are long living. Even in these 918 kb/sec promoted objects, string objects are going to be a smaller portion. Thus the amount of duplicate strings removed by '-XX:+UseStringDeduplication' was very negligible. Thus, sadly we didn't see the expected reduction in memory.

## Conclusion

- (a). '-XX:+UseStringDeduplication' will be useful only if application has a lot of long-lived duplicate strings. It wouldn't yield fruitful results for applications when majority of the objects are short-lived. Unfortunately, most modern web applications, micro-service applications objects are short-lived.
- (b). Another famous option recommended in the industry to eliminate duplicate strings is to use `String#intern()` function. However, `String#intern()` isn't going to be useful for this application. Because in `String#intern()` you end up creating the string objects and then eliminating it right after. If a string is short-lived by nature, you don't need to do this step, as regular garbage collection process will eliminate the strings. Also, `String#intern()` has a possibility to add (very little) latency overhead to the transaction and CPU overhead.
- (c). Given the current situation best way to eliminate duplicate strings from the application is to refactor the code to make sure duplicate strings are not even created. HeapHero points out the code paths where a lot of duplicate of strings are created. Using those pointers, we are going to continue our journey to refactor the code to reduce memory consumption.

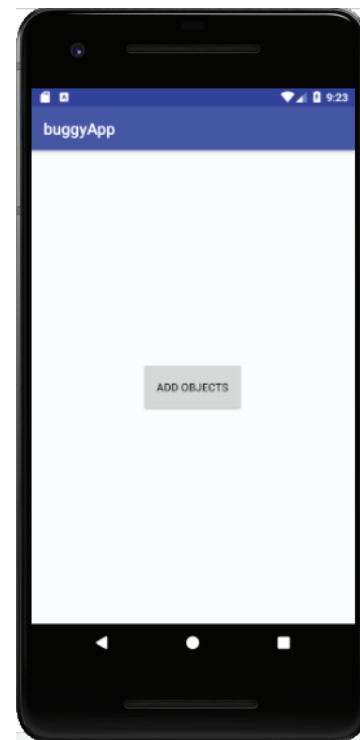
# HOW TO DIAGNOSE OUTOFMEMORYERROR IN ANDROID?

# 29

Diagnosing OutOfMemoryError in Android apps can become a tricky, tedious job. Here, we would like to show you an easy technique to troubleshoot OutOfMemoryError in Android apps.

## BuggyApp

To facilitate this exercise, we built a simple android application which would trigger OutOfMemoryError. We named this app 'BuggyApp'. Do you like the name? This app has just one page, which contains only one button: "Add Objects". When you click on this button, the app would start to add very large string objects to an array list in an infinite while loop. When large string objects are infinitely added to an array list, it will result in OutOfMemoryError.



Few seconds after you click on "Add Objects" button, application will crash with OutOfMemoryError, as shown below:

```
----- beginning of crash
05-28 14:45:42.796 29710-29710/buggycompany.com.buggyapp E/AndroidRuntime: FATAL EXCEPTION: main
Process: buggycompany.com.buggyapp, PID: 29710
java.lang.OutOfMemoryError: Failed to allocate a 2095496 byte allocation with 1762016 free bytes and 1720KB until OOM
  at java.util.Arrays.copyOf (Arrays.java:3352)
  at java.lang.AbstractStringBuilder.expandCapacity (AbstractStringBuilder.java:130)
  at java.lang.AbstractStringBuilder.ensureCapacityInternal (AbstractStringBuilder.java:114)
  at java.lang.AbstractStringBuilder.append (AbstractStringBuilder.java:417)
  at java.lang.StringBuilder.append (StringBuilder.java:133)
  at buggycompany.com.buggyapp.MainActivity$1$1.run (MainActivity.java:49)
  at android.os.Handler.handleCallback (Handler.java:751)
  at android.os.Handler.dispatchMessage (Handler.java:95)
  at android.os.Looper.loop (Looper.java:154)
  at android.app.ActivityThread.main (ActivityThread.java:6119) <1 internal call>
  at com.android.internal.os.ZygoteInit$MethodAndArgsCaller.run (ZygoteInit.java:886)
  at com.android.internal.os.ZygoteInit.main (ZygoteInit.java:776)
```

## How to diagnose OutOfMemoryError?

Now let's get to the interesting part – How to diagnose OutOfMemoryError? It's just two easy simple steps, my friend:

Capture Android Heap Dumps

Analyze Android Heap Dumps

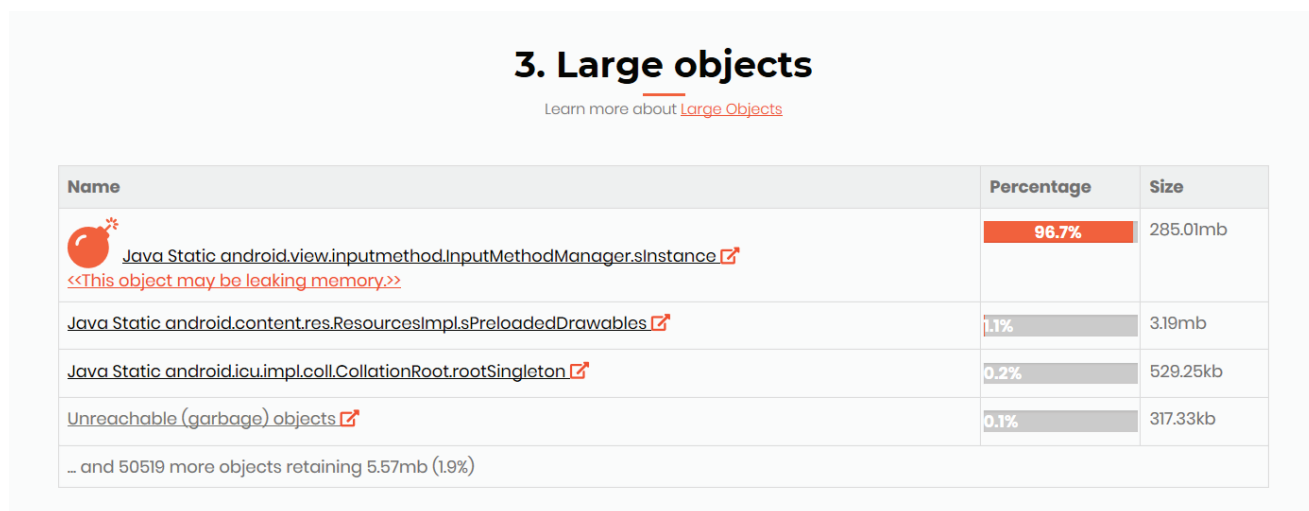
### 1. Capture Android Heap Dumps

First step is to capture heap dumps from the android app. Heap Dump is a snapshot of memory, which contains information about the objects in the memory, their references, their size... Here is an article which summarizes 3 different options to capture heap dumps from the android app . You can use any one of the options that is convenient for you to capture heap dumps. We used option #2 mentioned in the article to capture the heap dump from this 'BuggyApp'.

### 2. Analyze Android Heap Dumps

Second step is to analyze the captured heap dump. To analyze android heap dumps, we used the free online tool: HeapHero. This tool analyzes android heap dumps and points out potential memory leak suspects. Besides identifying memory leaks, HeapHero also identifies the amount of memory wasted due to poor programming practices and recommends solutions to fix the same. Since it's an online tool, you don't have to do any downloading, installation, setup.... All you need to do is to upload the heap dump file that was captured in step #1 to HeapHero.

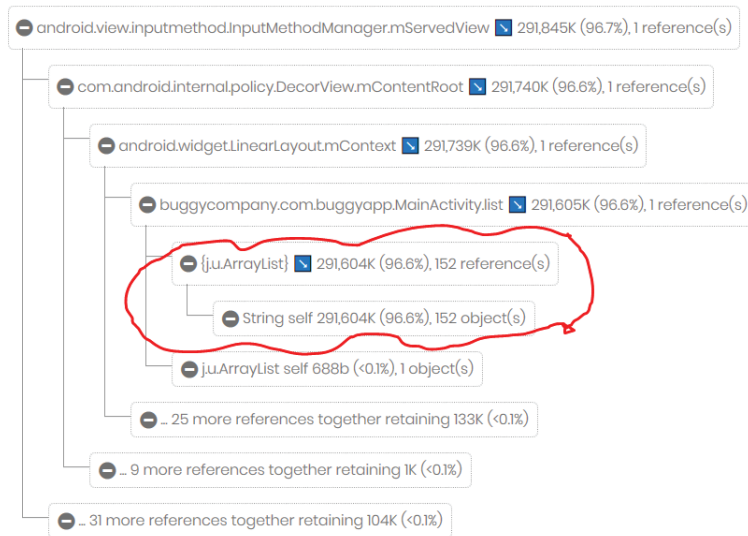
Now we uploaded the heap dump file captured in step #1 to HeapHero. HeapHero generated this beautiful report. In the report, there is a section: 'Large Objects'. This section reports all the large objects that are residing in the memory.



From the report, you could see the 'android.view.inputmethod.InputMethodManager.sInstance' object to occupy 96.7% of overall memory. This is a significant size for any object. In fact, HeapHero has in-built intelligence to identify potential memory leaking objects. Based on this intelligence, HeapHero also marked 'android.view.inputmethod.InputMethodManager.sInstance' object as potential memory leak suspect.

When we clicked on the 'android.view.inputmethod.InputMethodManager.sInstance' hyperlink in the report, it started to show stack trace/code path of the leaking objects.

## What does 285.01mb (96.7%) of Java Static android.view.inputmethod.InputMethodManager.sInstance contain?



You could see the 'buggycompany.com.buggyapp.MainActivity' object holding 'java.util.ArrayList' object, which in turn is holding on to large amount of string objects (96.6%). Ahaha, exact lines of code that we wrote in 'BuggyApp' to trigger OutOfMemoryError.

That's it my friend . With these two simple steps, you might be able to solve complex OutOfMemoryError and memory leak problems. We hope this article will help you to isolate the exact lines of code that are triggering memory problems in your app.



# HOW TO DIAGNOSE MEMORY LEAKS?

# 30

Memory leaks don't have to be hard/scary/tedious problem to solve if we can follow below mentioned 3 simple steps:

## Step 1: Capture baseline heap dump

You need to capture heap dump when it's in the healthy state. Start your application. Let it take real traffic for 10 minutes. At this point, capture heap dump. Heap Dump is basically the snapshot of your memory. It contains all objects that are residing in the memory, values stored in those objects, inbound & outbound references of those object. You can capture Heap dump using the following command:

```
jmap-dump:format=b,file=<file-path><pid>  
where  
pid: is the Java Process Id, whose heap dump should be captured  
file-path: is the file path where heap dump will be written in to.
```

If you don't want to use jmap for capturing heap dumps, here are several other options to capture heap dumps.

It's always better to capture heap dump in the production environment (unless in the test environment, you can mirror the exact production traffic pattern). The traffic type and its volume play a primary role in the type and number of objects created in the memory.

## Step 2: Capture troubled heap dump

After doing step #1, let the application run. Before application crashes, take another heap dump once again. Often times, it might be challenging to capture heap dumps before it crashes because we don't know when the application will crash. Is it after 30 minutes, 3 hours, 3 days? Thus, it's ideal to start your application with following JVM property:

```
-XX:+HeapDumpOnOutOfMemoryError -XX:HeapDumpPath=<file-path>  
file-path: is the file path where heap dump will be written in to.
```

This property will trigger heap dump right when the application experiences OutOfMemoryError.

### Step 3: Compare heap dumps

Objects which are causing memory leaks grow over the period. If you can identify objects whose size has grown between the heap dumps captured in step #1 and step #2, then those are the objects which are causing memory leak.

## Large objects

(Potential Memory Leak Suspects)

Name	Percentage	Size
Unreachable (garbage) objects <a href="#">🔗</a>	17.6%	41.5kb
Java Static java.lang.System.out <a href="#">🔗</a>	10.4%	24.47kb
Java Static sun.launcher.LauncherHelper.ostream <a href="#">🔗</a>	10.4%	24.47kb
Java Static java.nio.charset.Charset.standardProvider <a href="#">🔗</a>	8.4%	19.8kb
Java Static sun.launcher.LauncherHelper.scloder <a href="#">🔗</a>	5.8%	13.74kb
... and 233 more objects retaining 15.91kb (6.7%)		

[SHOW ALL RECORDS >>](#)

**HeapHero**
Home   Features   Why Us?   User Reviews   [BLOG](#)

### What does 13.74kb (5.8%) of Java Static sun.launcher.LauncherHelper.scloder contain?



You can consider using heap dump analyzer tool such as HeapHero.io for this purpose. When you load the heap dumps into HeapHero.io, it provides rich information about your application's memory. There is a "Large Objects" section (shown in Fig 1), which reports largest objects that are sitting in the memory. Compare this section between heap dumps captured in step #1 and step #2. If you notice any abnormal growth of objects, then they are the ones which are causing memory leak in your application. You can also click on any of the largest objects to see the children, grandchildren, great-grandchildren objects present in it. Screen shot of this section is shown in Fig #2.

# WHAT HAPPENS BEHIND THE SCENE – FINALIZE() METHOD

31

“What is the purpose of finalize() method?” is one of the often asked Java interview questions. The typical answer to it is: “Usual purpose of finalize() method is to perform cleanup actions before the object is discarded”. However, behind the scene, finalize() method are handled in a special way. A small mistake in finalize() method has the potential to jeopardize entire application’s availability. Let’s study it in detail.

## Behind the Scene

Objects that have “finalize()” method are treated differently during garbage collection process than the ones which don’t have. During garbage collection phase, objects with “finalize()” method aren’t immediately evicted from the memory. Instead, as the first step, those objects are added to an internal queue of ‘java.lang.ref.Finalizer’. For entire JVM, there is only one low priority JVM thread, by name, ‘Finalizer’ that executes “finalize()” method of each object in the queue. Only after the execution of “finalize()” method, the object becomes eligible for Garbage Collection. Assume if the application is producing a lot of objects which has “finalize()” method and the low priority “Finalizer” thread isn’t able to keep up with executing finalize() method, then significant amount of unfinalized objects will start to build up in the internal queue of ‘java.lang.ref.Finalizer’, which would result in significant amount of memory wastage.

Sometimes because of poor programming practice, “Finalizer” thread may start to WAIT or BLOCK while executing the “finalize()” method. If “Finalizer” thread starts to wait or block, then the number of unfinalized objects in the internal queue of ‘java.lang.ref.Finalizer’ will start to grow significantly, which would result in OutOfMemoryError, jeopardizing entire JVM’s availability.

## Example

To illustrate this theory, we wrote a simple sample program.

```
public class SampleObject {  
  
    public String data;  
  
    public SampleObject(String data) {  
  
        this.data = data;  
  
    }  
  
    @Override  
  
    public void finalize() {  
  
        try {  
  
            // Sleep for 1 minute.  
  
            Thread.currentThread().sleep(1 * 60 * 1000);  
  
        } catch (Exception e) {}  
  
    }  
}
```

Basically, 'main()' method of this class creates 'SampleObject' continuously. Interesting part of this program is "finalize()" method. This method puts the current executing thread (i.e. 'Finalizer' thread) to sleep for 1 minute. This example illustrates the poor implementation of "finalize()" method.

When we ran the above program with max heap size of 10 mb (i.e. -Xmx10M), it crashed with 'java.lang.OutOfMemoryError' after few seconds of launch.

This program crashed with 'java.lang.OutOfMemoryError' because: Only after the execution of 'finalize()' method, SampleObject can be evicted from the memory. Since 'Finalizer' thread is put to sleep, it couldn't execute the "finalize()" method at the rate in which 'main()' method was creating new 'SampleObject'. Thus memory got filled up and program resulted in 'java.lang.OutOfMemoryError'.

On the other hand, when we commented out "finalize()" method, program ran continuously without experiencing any 'java.lang.OutOfMemoryError'.

## How to diagnose this problem?


Your application might contain hundreds, thousands, millions of classes. It includes classes from 3rd party libraries and frameworks. Now the question become, how will you identify “finalize()” methods that are poorly implemented? This is a tough question to answer. This is where heap dump analysis tools like HeapHero.io might come handy.

When heap dump was captured from the above program and uploaded to HeapHero.io, it generated this beautiful report with several sections. Section that is of interest to us is: ‘Objects waiting for finalization’.

### Objects waiting for Finalization

Learn about [Objects waiting for Finalization](#)

Wasted Memory



7.66mb (97.2%)

**❓ What are the objects waiting for finalization?**

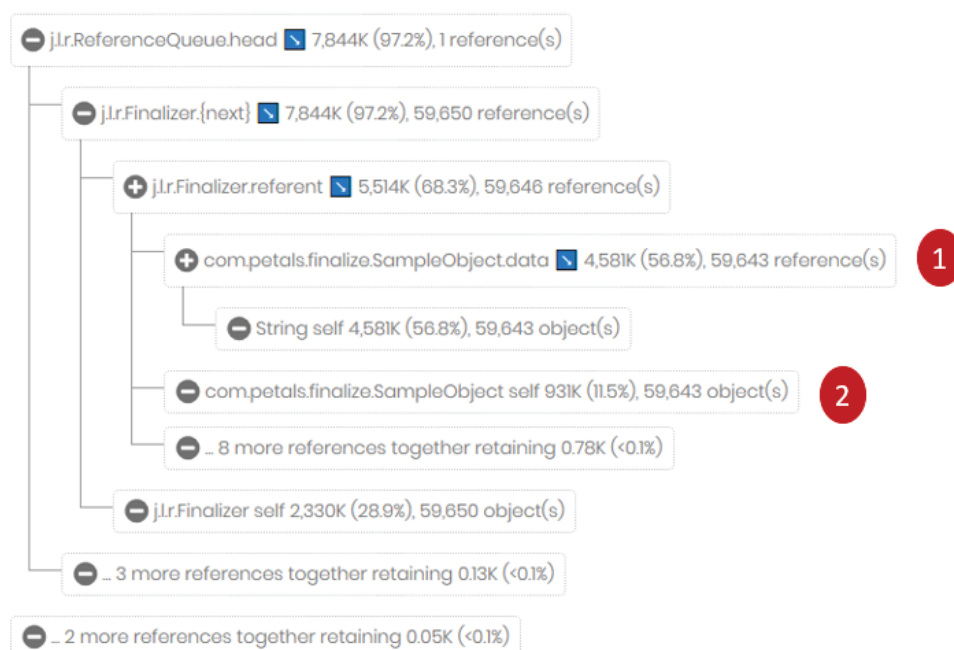
To see objects waiting for finalization, [click here](#).

**🔧 How to fix objects waiting for finalization?**

Please refer to our [recommendations](#).

This section of the report shows the amount of memory wasted due to objects waiting for finalization of your application. In this hypothetical example 7.66 MB i.e. 97.2% is the amount of memory that is wasted.

## Objects waiting for Finalization



When you click on the hyperlink given under 'What are the objects waiting for finalization?', you will be able to see the objects waiting to be finalized. Basically, you will be able to see the object tree of "j.l.r.ReferenceQueue" (note this is queue of 'java.lang.ref.Finalizer' object that holds the reference of all objects, whose finalize() method needs to be executed). If you drill down the tree it will show the objects that are sitting in the queue waiting to be finalized. Here you can see 2 types of objects that are sitting in the queue:

com.petals.finalize.SampleObject.data occupying 56.8% of memory

com.petals.finalize.SampleObject occupying 11.5% of memory

BINGO!! These are the objects that are created in our sample program

# TROUBLESHOOT OUTFOFMEMORY ERROR : UNABLE TO CREATE NEW NATIVE THREAD

There are 8 flavors of `java.lang.OutOfMemoryError`. In these 8 flavors

```
java.lang.OutOfMemoryError: unable to create new native thread
```

is one of the commonly occurring flavor. This type of

```
OutOfMemoryError
```

is generated when an application isn't able to create new threads. This error can surface because of following two reasons:

There is no room in the memory to accommodate new threads.  
The number of threads exceeds the Operating System limit.

## Solutions

There are 6 potential solutions to address this

```
java.lang.OutOfMemoryError: unable to create new native thread
```

error. Depending on what event is triggering this error, either one or a combination of the below mentioned solutions can be applied to resolve the problem.

### 1. Fix Thread Creation Rate

When you see

```
java.lang.OutOfMemoryError: unable to create new native thread,
```

you should diagnose whether the application has started to create more threads. You can use online thread dump analyzer tool such as <http://fastthread.io/> (which I would highly recommend), to see how many threads are created? What is the stack trace of those excessively created threads? Who is creating them? Once you know to these questions, it's easy to solution them. Check out the 'real world example' section of this article, which walks through a '

```
java.lang.OutOfMemoryError: unable to create new native thread'
```

problem experienced a major B2B travel application and how <http://fastthread.io/> tool was used to diagnose the problem.

## 2. Increase the Thread Limits Set at Operating System

The Operating System has limits for the number of threads that can be created. The limit can be found by issuing “

```
ulimit -u
```

” command. In certain servers, I have seen this value set to a low value such as 1024. It means totally only 1024 threads can be created in this machine. So if your application is creating more than 1024 threads, it's going to run into

```
java.lang.OutOfMemoryError: unable to create new native thread.
```

In such circumstances increase this limit.

## 3. Allocate More Memory to the Machine

If you don't see a high number of threads created and

```
ulimit -u
```

value is well ahead then it's indicative that your application has grown organically and needed more memory to create threads. In such circumstance, allocate more memory to the machine. It should solve the problem.

## 4. Reduce Heap Space

One very important point that even seasoned engineers forget is: threads are not created within the JVM heap. They are created outside the JVM heap. So if there is less room left in the RAM, after the JVM heap allocation, application will run into

So let's consider this example:

Overall RAM size	6 GB
Heap size (i.e. -Xms and -Xmx)	5 GB
Perm Gen size (i.e. -XX:MaxPermSize and -XX:MaxPermSize)	512 MB

As per this configuration 5.5 GB (i.e. 5 GB heap + 512 MB Perm Gen) is used by the JVM Heap and it leave only 0.5GB (i.e. 6 GB – 5.5GB) space. Note in this 0.5 GB space – kernel processes, other user processes and threads has to run. It may not be sufficient, and most likely the application will start to experience

```
java.lang.OutOfMemoryError: unable to create new native thread
```

To mitigate this problem, you can consider reducing the Heap Size from 5GB to 4GB (if your application can accommodate it without running into other memory bottlenecks).



## 5. Reduce Number of Processes

This solution is quite similar to 'Reduce Heap Space'. Let's look into this scenario, where you are running multiple processes on a server which is constrained by memory. Say:

Overall RAM size	32 GB
Number of Java Processes in the server	5
Heap size of each Java process	6 GB

It means in total all of the java processes heap is occupying 30 GB (i.e. 5 processes X 6 GB) of memory. It leaves only 2 GB for kernel processes, other user processes and threads to run. It may not be sufficient, and most likely the application will start to experience

```
java.lang.OutOfMemoryError: unable to create new native thread
```

In this circumstance, it's better to run only 4 java processes on one server. So that only 24 GB is occupied (4 processes X 6GB), and it leaves 8 GB (i.e. 32 GB – 24 GB) of memory. It might leave enough room for threads and run other processes to run.

## 6. Reduce Thread Stack Size (-Xss)

A thread occupies memory in RAM. So if each thread has high memory allocation, then overall memory consumption will also go higher. The default value of a thread's memory size depends on the JVM provider. In some cases it's 1mb. So if your application has 500 threads, then threads alone is going to occupy 500mb of space.

However, you can use the java system property `-Xss` to set the thread's memory size. Using this property, you can throttle down the memory size. Example if you configure `-Xss256k`, your threads will only consume 125mb of space (i.e. 500 threads X 256k). So by lowering `-Xss` size also, you might be able to eliminate

```
java.lang.OutOfMemoryError: unable to create new native thread
```

**CAUTION:** However if you configure `-Xss` to a very low value, you will start to experience `java.lang.StackOverflowError`. If you configure to even lower value, JVM will not even start.

## Real World Example

Now let me walk through a real world example of

```
java.lang.OutOfMemoryError: unable to create new native thread
```

which I diagnosed recently. This error was experienced by a major B2B travel application in North America. No recent production deployments were made to this application, but all of a sudden it started to throw

```
java.lang.OutOfMemoryError: unable to create new native thread
```

**Step 1:** As an initial step, we captured the thread dump from the application when it was experiencing this

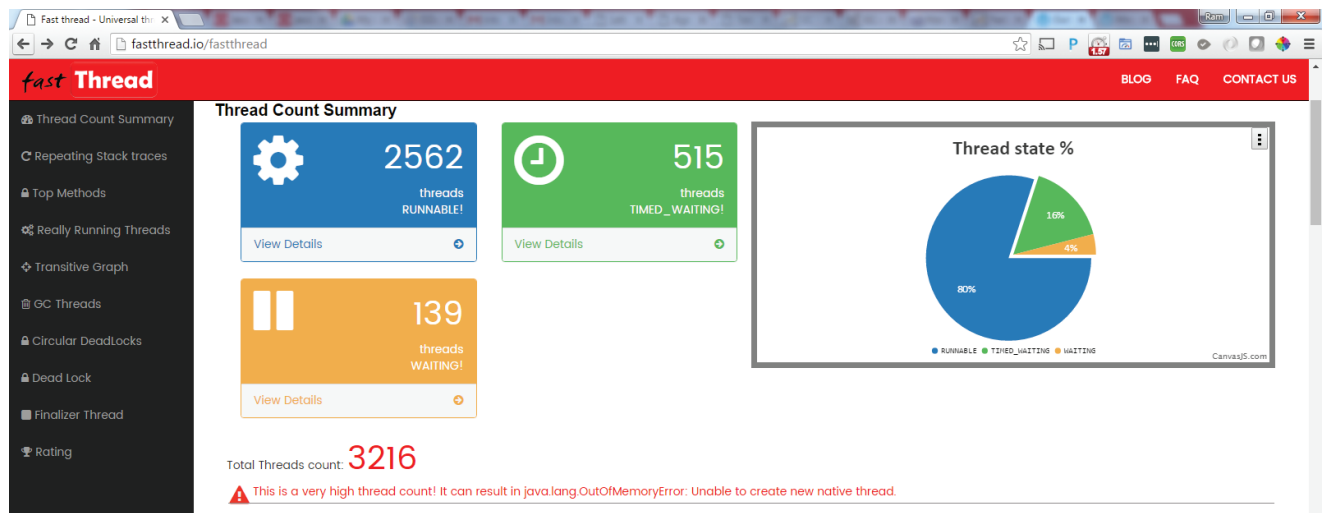
```
java.lang.OutOfMemoryError: unable to create new native thread
```

Then we uploaded the thread dump into online thread dump analyzer <http://fastthread.io/>.

**Step 2:** <http://fastthread.io/> tool reported that application had 3216 threads alive and rightly pointed that it can result in

```
java.lang.OutOfMemoryError: unable to create new native thread
```

3000+ threads was a very high thread count for this application, which is at least 6 times more than the regular period.



**Step 3:** Since now it's confirmed that excessive threads are causing

```
java.lang.OutOfMemoryError: unable to create new native thread
```

the next step is to identify what those excessively created threads are? And who is creating them? <http://fastthread.io/> tool has a section "Repeating Stack traces", in which threads with same stack traces are grouped together. In that section, we noticed that 2319 threads (i.e. 72%) are exhibiting same stack trace as shown in Fig 2.

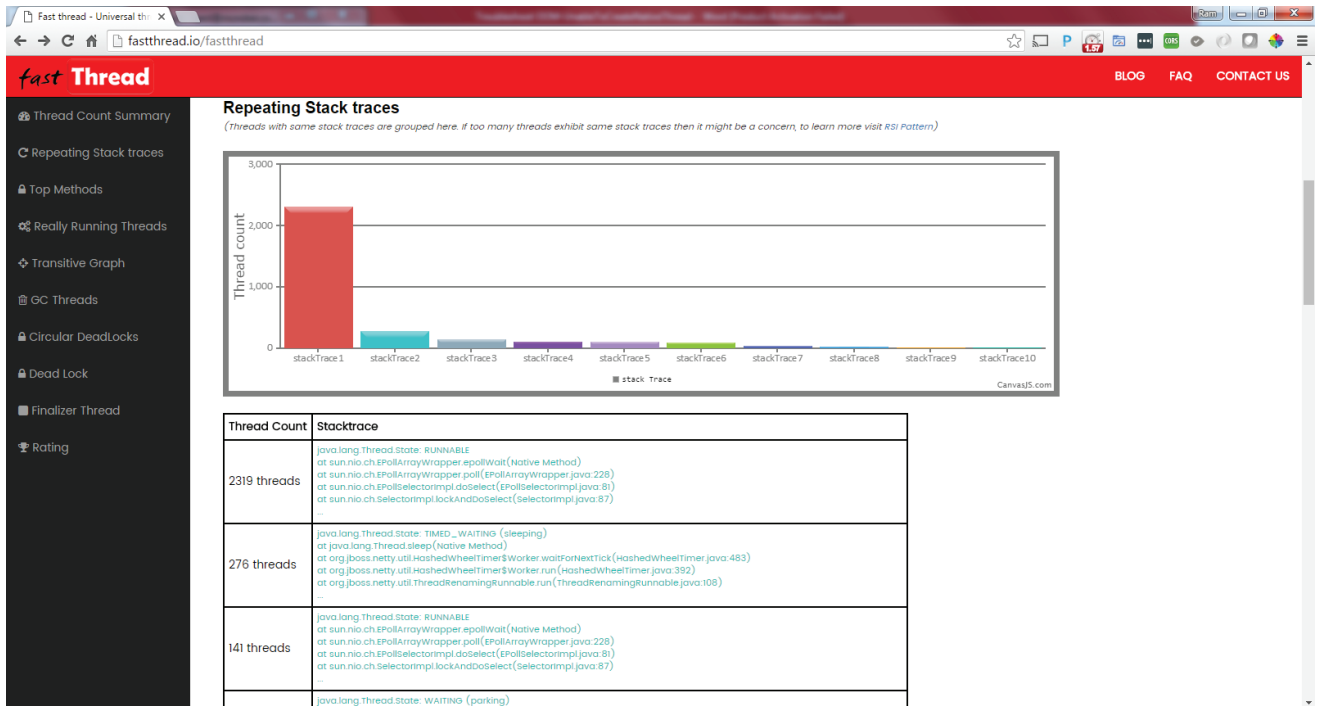


Fig 2: <http://fastthread.io/> tool showing group of threads which has same stack trace

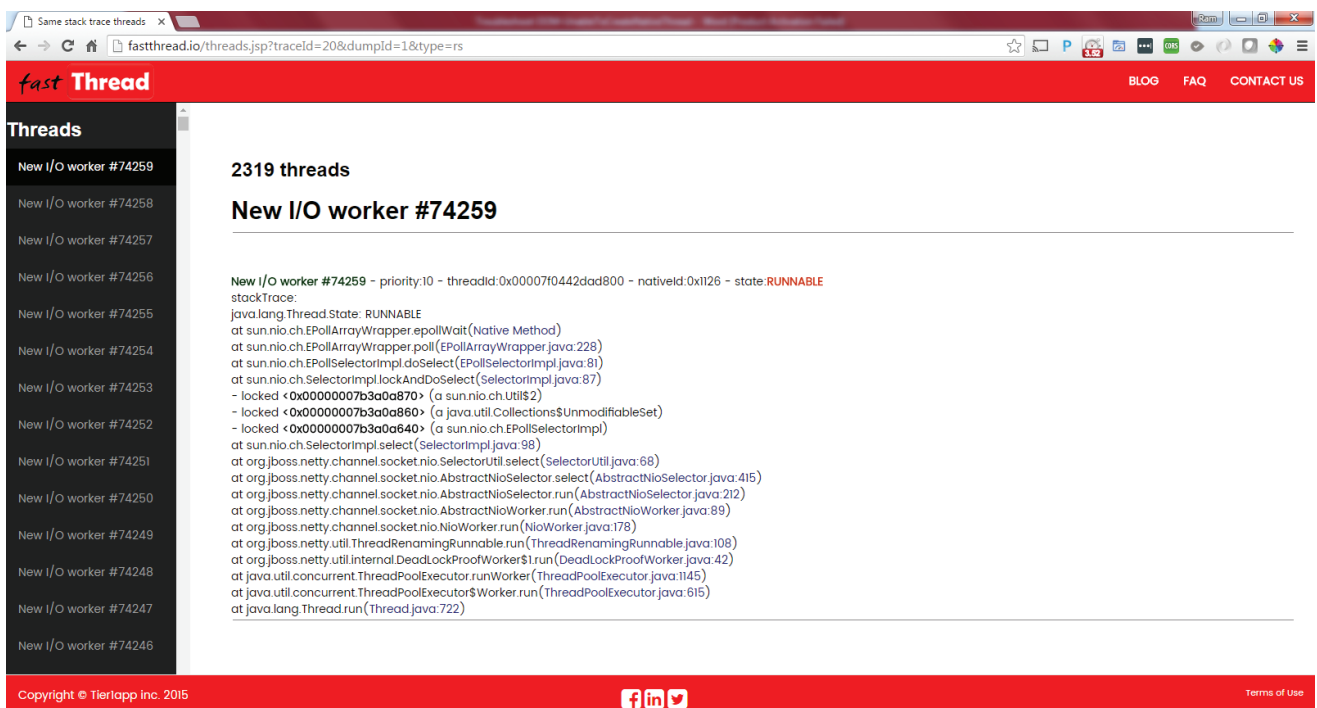


Fig 3: Individual Thread's stack trace as reported by the <http://fastthread.io/> tool

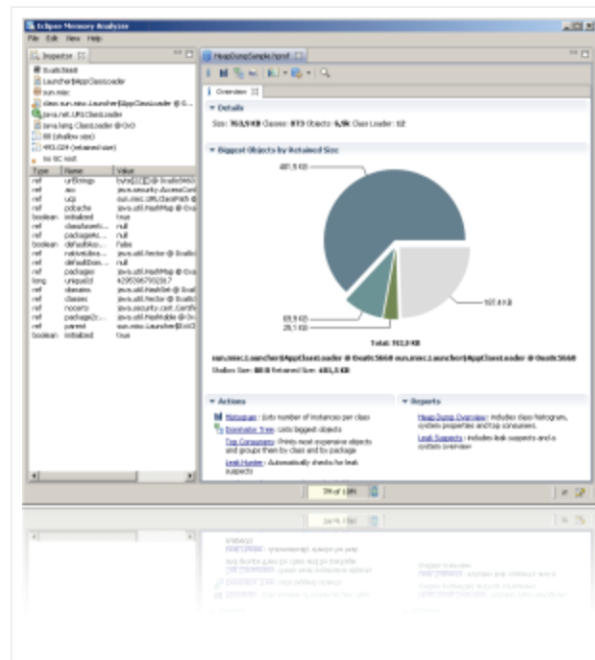
From the stack trace, we inferred that these threads created are by the Datastax driver. This application uses DataStax driver for connecting with Apache Cassandra NoSQL Database. So now the question becomes all of sudden why Datastax driver started to create so many threads? No upgrades were made to this driver. No recent deployments were made to the application. Why all of a sudden this problem started?

**Root cause:** Apparently the problem turned out that Apache Cassandra NoSQL DB was running into disk space issue on one of its nodes. This issue caused the Datastax driver to spawn thousands of threads. Thus it cascaded as 'java.lang.OutOfMemoryError: unable to create new native thread' error on the JVM side. When more space was allocated to Apache Cassandra NoSQL DB nodes, the problem got resolved.

# ECLIPSE MAT – TIDBITS

# 33

Eclipse MAT is a great JVM Memory Analysis tool. Here are few tidbits to use it effectively.



## 1. Use stand-alone version

Two versions of Eclipse MAT is available:

1. Stand-alone
2. Eclipse Plugin

Based on my personal experience, stand-alone version seems to work better and faster than plugin version. So I would highly recommend installing Stand-alone version.

## 2. Eclipse MAT – heap size

If you are analyzing a heap dump of size, say 2 GB, allocate at least 1 GB additional space for Eclipse MAT. If you can allocate more heap space, then it's more the merrier. You can allocate additional heap space for Eclipse MAT tool, by editing MemoryAnalyzer.ini file. This file is located in the same folder where MemoryAnalyzer.exe is present. To the MemoryAnalyzer.ini you will add -Xmx3g at the bottom.

## Examples

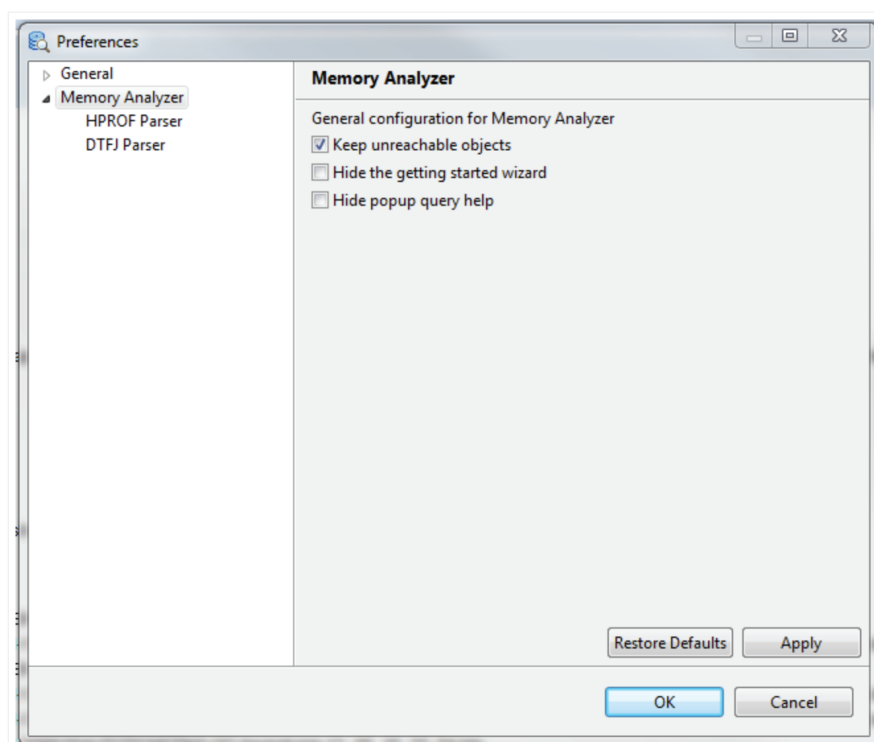
```
-startup
plugins/org.eclipse.equinox.launcher_1.3.0.v20140415-2008.jar
--launcher.library
plugins/org.eclipse.equinox.launcher.win32.win32.x86_64_1.1.200.v20140603-13
26
-vmargs
-Xmx3g
```

### 3. Enable 'keep unreachable objects'

From its reporting Eclipse MAT removes the object which it thinks as 'unreachable.' As 'unreachable' objects are eligible for garbage collection, MAT doesn't display them in the report. Eclipse MAT classifies Local variables in a method as 'unreachable objects'. Once thread exits the method, objects in local variables will be eligible for garbage collection.

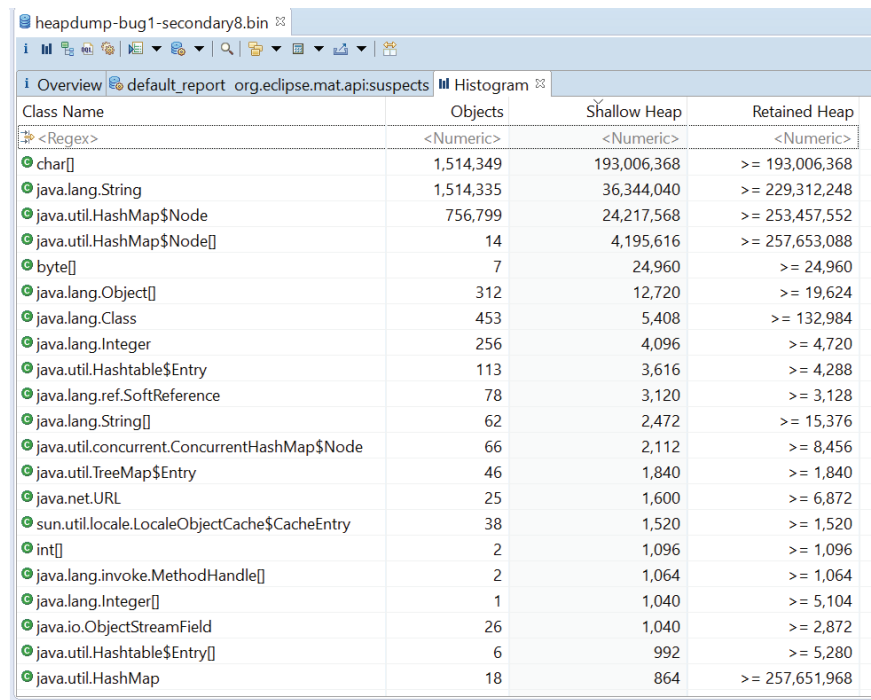
However, there are several cases where a thread will go into a 'BLOCKED' or prolonged 'WAITING', 'TIMED\_WAITING' state. In such circumstances local variables will be still alive in memory, occupying space. Since Eclipse MAT default settings don't show the unreachable objects, you will not get visibility into these objects. You can change the default settings in Eclipse MAT 1.4.0 version by:

1. Go to Window > Preferences ...
2. Click on 'Memory Analyzer'
3. Select 'Keep unreachable objects'
4. Click on 'OK' button



## 4. Smart Data Settings

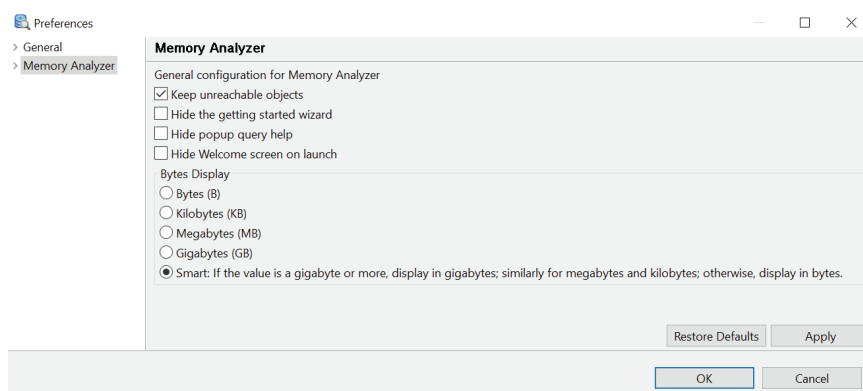
Eclipse MAT by default displays data in bytes. It's difficult to read large object sizes in bytes and digest it. Example Eclipse MAT prints object size like this: "193,006,368". It's much easier if this data can be displayed in KB, MB, GB i.e. "184.07 MB".



Class Name	Objects	Shallow Heap	Retained Heap
<Regex>	<Numeric>	<Numeric>	<Numeric>
char[]	1,514,349	193,006,368	>= 193,006,368
java.lang.String	1,514,335	36,344,040	>= 229,312,248
java.util.HashMap\$Node	756,799	24,217,568	>= 253,457,552
java.util.HashMap\$Node[]	14	4,195,616	>= 257,653,088
byte[]	7	24,960	>= 24,960
java.lang.Object[]	312	12,720	>= 19,624
java.lang.Class	453	5,408	>= 132,984
java.lang.Integer	256	4,096	>= 4,720
java.util.Hashtable\$Entry	113	3,616	>= 4,288
java.lang.ref.SoftReference	78	3,120	>= 3,128
java.lang.String[]	62	2,472	>= 15,376
java.util.concurrent.ConcurrentHashMap\$Node	66	2,112	>= 8,456
java.util.TreeMap\$Entry	46	1,840	>= 1,840
java.net.URL	25	1,600	>= 6,872
sun.util.locale.LocaleObjectCache\$CacheEntry	38	1,520	>= 1,520
int[]	2	1,096	>= 1,096
java.lang.invoke.MethodHandle[]	2	1,064	>= 1,064
java.lang.Integer[]	1	1,040	>= 5,104
java.io.ObjectStreamField	26	1,040	>= 2,872
java.util.Hashtable\$Entry[]	6	992	>= 5,280
java.util.HashMap	18	864	>= 257,651,968

Eclipse MAT provides an option to display object size in KB, MB, GB based on their appropriate size. It can be enabled by following below steps:

1. Go to Window > Preferences ...
2. Click on 'Memory Analyzer'
3. In the 'Bytes Display' section select 'Smart: If the value is a gigabyte or ...'
4. Click on 'OK' button



Once this setting change is made, all data will appear in much more readable KB, MB, GB format, as shown in below figure.

heapdump-bug1-secondary8.bin			
i Overview default_report org.eclipse.mat.api:suspects Histogram			
Class Name	Objects	Shallow Heap	Retained Heap
<Regex>	<Numeric>	<Numeric>	<Numeric>
char[]	1,514,349	184.07 MB	>= 184.07 MB
java.lang.String	1,514,335	34.66 MB	>= 218.69 MB
java.util.HashMap\$Node	756,799	23.10 MB	>= 241.72 MB
java.util.HashMap\$Node[]	14	4.00 MB	>= 245.72 MB
byte[]	7	24.38 KB	>= 24.38 KB
java.lang.Object[]	312	12.42 KB	>= 19.16 KB
java.lang.Class	453	5.28 KB	>= 129.87 KB
java.lang.Integer	256	4.00 KB	>= 4.61 KB
java.util.Hashtable\$Entry	113	3.53 KB	>= 4.19 KB
java.lang.ref.SoftReference	78	3.05 KB	>= 3.05 KB
java.lang.String[]	62	2.41 KB	>= 15.02 KB
java.util.concurrent.ConcurrentHashMa...	66	2.06 KB	>= 8.26 KB
java.util.TreeMap\$Entry	46	1.80 KB	>= 1.80 KB
java.net.URL	25	1.56 KB	>= 6.71 KB
sun.util.locale.LocaleObjectCache\$Cach...	38	1.48 KB	>= 1.48 KB
int[]	2	1.07 KB	>= 1.07 KB
java.lang.invoke.MethodHandle[]	2	1.04 KB	>= 1.04 KB
java.lang.Integer[]	1	1.02 KB	>= 4.98 KB
java.io.ObjectStreamField	26	1.02 KB	>= 2.80 KB
java.util.Hashtable\$Entry[]	6	992 B	>= 5.16 KB
java.util.HashMap	18	864 B	>= 245.72 MB

# SHALLOW HEAP, RETAINED HEAP

34

Eclipse MAT (Memory Analyzer Tool) is a powerful tool to analyze heap dumps. It comes quite handy when you are trying to debug memory related problems. In Eclipse MAT two types of object sizes are reported:

1. Shallow Heap
2. Retained Heap

In this article let's study the difference between them. Let's study how are they calculated?

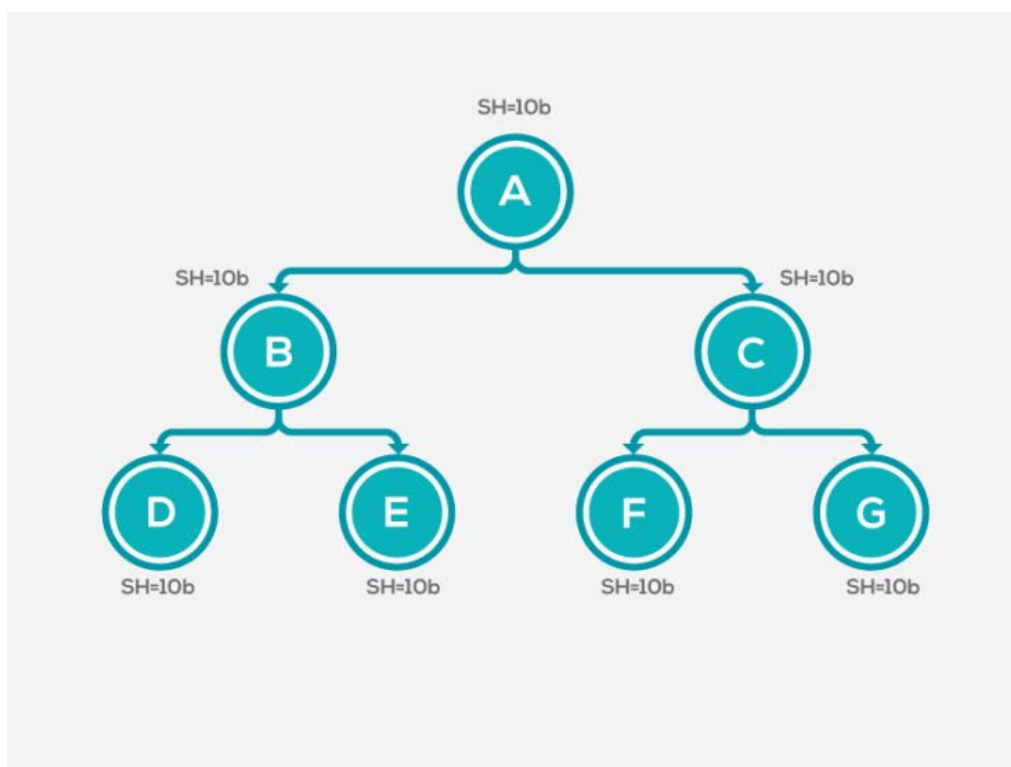


Fig 1: Objects in memory

It's easier to learn new concepts through example. Let's say your application's has object model as shown in Fig #1:

- Object A is holding reference to objects B and C.
- Object B is holding reference to objects D and E.
- Object C is holding reference to objects F and G.

Let's say each object occupies 10 bytes of memory. Now with this context let's begin our study.?



## Shallow Heap size

Shallow heap of an object is its size in the memory. Since in our example each object occupies 10 bytes, shallow heap size of each object is 10 bytes. Very simple.

## Retained Heap size of B

From the Fig #1 you can notice that object B is holding reference to objects D and E. So, if object B is garbage collected from memory, there will be no more active references to object D and E. It means D & E can also be garbage collected. Retained heap is the amount of memory that will be freed when the particular object is garbage collected. Thus, retained heap size of B is:

= B's shallow heap size + D's shallow heap size + E's shallow heap size

= 10 bytes + 10 bytes + 10 bytes

= 30 bytes

Thus, retained heap size of B is 30 bytes.

## Retained Heap size of C

Object C is holding reference to objects F and G. So, if object C is garbage collected from memory, there will be no more references to object F & G. It means F & G can also be garbage collected. Thus, retained heap size of C is:

= C's shallow heap size + F's shallow heap size + G's shallow heap size

= 10 bytes + 10 bytes + 10 bytes

= 30 bytes

Thus, retained heap size of C is 30 bytes as well

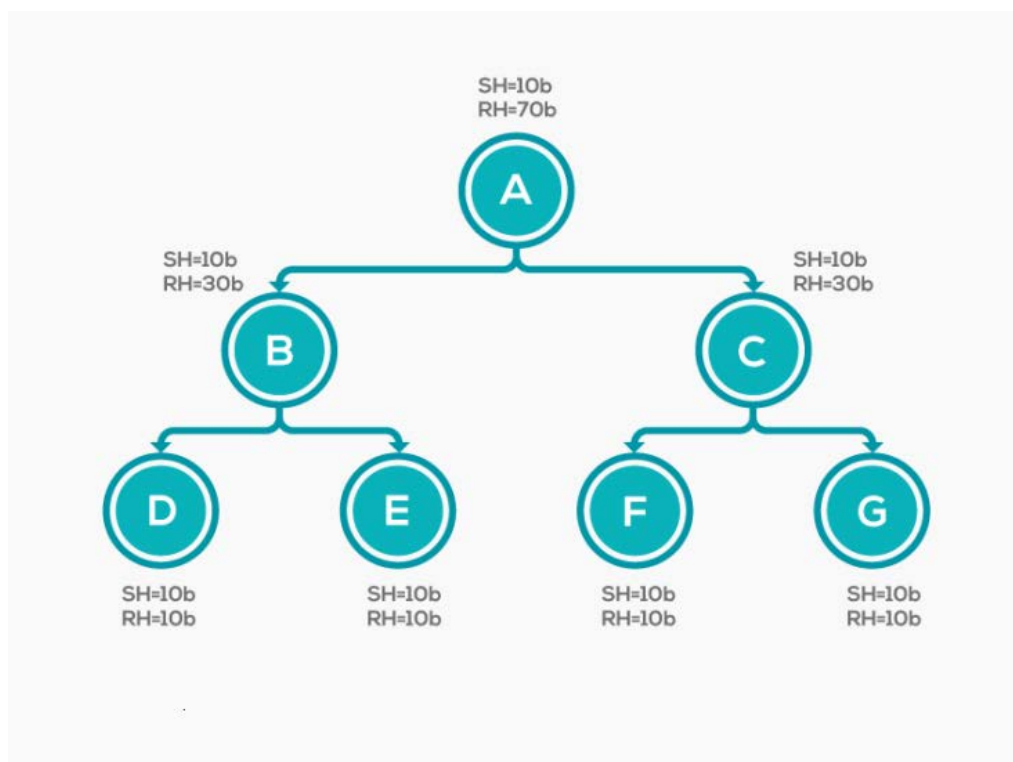


Fig 2: Objects Shallow and Retained Heap size

## Retained Heap size of A

Object A is holding reference to objects B and C, which in turn are holding references to objects D, E, F, G. Thus, if object A is garbage collected from memory, there will be no more reference to object B, C, D, E, F and G. With this understanding let's do retained heap size calculation of A.

Thus, retained heap size of A is:

= A's shallow heap size + B's shallow heap size + C's shallow heap size + D's shallow heap size + E's shallow heap size + F's shallow heap size + G's shallow heap size

= 10 bytes + 10 bytes + 10 bytes + 10 bytes + 10 bytes + 10 bytes + 10 bytes

= 70 bytes

Thus, retained heap size of A is 70 bytes.

## Retained heap size of D, E, F and G

Retained heap size of D is 10 bytes only i.e. their shallow size only. Because D don't hold any active reference to any other objects. Thus, if D gets garbage collected no other objects will be removed from memory. As per the same explanation objects E, F and G's retained heap size are also 10 bytes only.

## Let's make our study more interesting

Now let's make our study little bit more interesting, so that you will gain thorough understanding of shallow heap and retained heap size. Let's have object H starts to hold reference to B in the example. Note object B is already referenced by object A. Now two guys A and H are holding references to object B. In this circumstance let's study what will happen to our retained heap calculation.

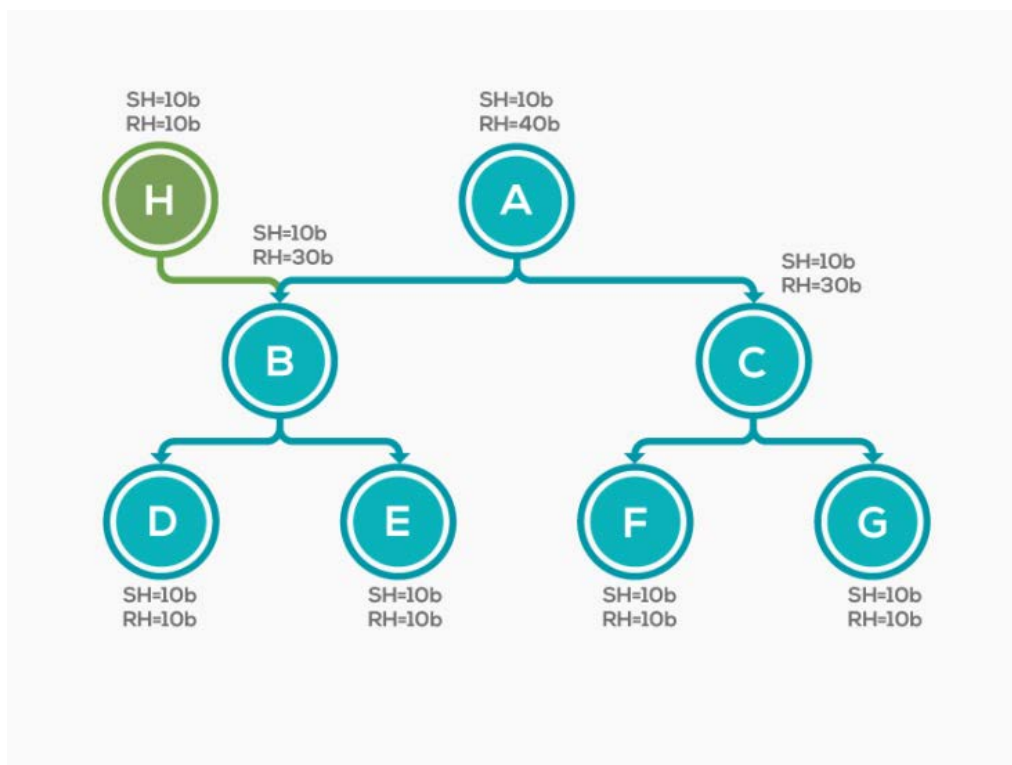


Fig 3: New reference to Object B

In this circumstance retained heap size of object A will go down to 40 bytes. Surprising? Puzzling? continue reading on. If object A gets garbage collected, then there will be no more reference to objects C, F and G only. Thus, only objects C, F and G will be garbage collected. On the other hand, objects B, D and E will continue to live in memory, because H is holding active reference to B. Thus B, D and E will not be removed from memory even when A gets garbage collected.

Thus, retained heap size of A is:

= A's shallow heap size + C's shallow heap size + F's shallow heap size + G's shallow heap size

= 10 bytes + 10 bytes + 10 bytes + 10 bytes

= 40 bytes.

Thus, retained heap size of A will become 40 bytes. All other objects retained heap size will remain undisturbed, because there is no change in their references.

Hope this article helped to clarify Shallow heap size and Retained heap size calculation in Eclipse MAT. You might also consider exploring HeapHero – another powerful heap dump analysis tool, which shows the amount of memory wasted due to inefficient programming practices such as duplication of objects, overallocation and underutilization of data structures, suboptimal data type definitions,....

# OUTOFMEMORYERROR

# 35

NO.	OutOfMemoryError Cause	Solution
1	Java heap space <ol style="list-style-type: none"> <li>Object could not be allocated in the Java heap</li> <li>Increase in Traffic volume</li> <li>Application is unintentionally holding references to objects which prevents the objects from being garbage collected</li> <li>Application makes excessive use of finalizers. Finalizer objects aren't GCed immediately. Finalizers are executed by a daemon thread that services the finalization queue. Sometimes finalizer thread cannot keep up, with the finalization queue.</li> </ol>	<ol style="list-style-type: none"> <li>Increase Heap size '-Xmx'.</li> <li>Fix memory leak in the application</li> </ol>
2	GC overhead limit exceeded <ol style="list-style-type: none"> <li>Java process is spending more than 98% of its time doing garbage collection and recovering less than 2% of the heap and has been doing so far the last 5 (compile time constant) consecutive garbage collection</li> </ol>	<ol style="list-style-type: none"> <li>Increase heap size '-Xmx'</li> <li>GC Overhead limit exceeded can be turned off with '-XX:-UseGCOverheadLimit'</li> <li>Fix the memory leak in the application</li> </ol>
3	Requested array size exceeds VM limit <ol style="list-style-type: none"> <li>Application attempted to allocate an array that is larger than the heap size</li> </ol>	<ol style="list-style-type: none"> <li>Increase heap size '-Xmx'</li> <li>Fix bug in application. code attempting to create a huge array</li> </ol>
4	Permgen Space <ol style="list-style-type: none"> <li>Permgen space contains:               <ol style="list-style-type: none"> <li>Names, Fields, methods of the classes</li> <li>Object arrays and type arrays associated with a class</li> <li>Just In Time compiler optimizations</li> </ol>               When this space runs out of space this error is thrown             </li> </ol>	<ol style="list-style-type: none"> <li>Increase Permgen size '-XX:MaxPermSize'</li> <li>Application redeployment without restarting can cause this issues. So restart JVM.</li> </ol>

5	Metaspace	<p>1. From Java 8 Permgen replaced by Metaspace. Class metadata is allocated in native memory (referred as metaspace). If metaspace is exhausted then this error is thrown</p>	<p>1. If '-XX:MaxMetaSpaceSize', has been set on the command-line, increase its value.</p> <p>2. Remove '-XX:MaxMetsSpaceSize'</p> <p>3. Reducing the size of the Java heap will make more space available for MetaSpace.</p> <p>4. Allocate more memory to the server</p> <p>5. Could be bug in application. Fix it.</p>
6	Unable to create new native thread	<p>1. There isn't sufficient memory to create new threads. Threads are created in native memory. It indicates there isn't sufficient native memory space</p>	<p>1. Allocate more memory to the machine</p> <p>2. Reduce Java Heap Space</p> <p>3. Fix thread leak in the application.</p> <p>4. Increase the limits at the OS level. <code>ulimit -a</code></p> <p>5. Reduce thread stack size with <code>-Xss</code> parameter</p>
7	Kill process or sacrifice child	<p>1. Kernel Job – Out of Memory Killer. Will kill processes under extremely low memory conditions</p>	<p>1. Migrate process to different machine.</p> <p>2. Add more memory to machine</p>
8	Reason stack_trace_with_native_method	<p>1. Native method encountered allocation failure</p> <p>2. a stack trace is printed in which the top frame is a native method</p>	<p>1. Use OS native utilities to diagnose</p>

# VIRTUAL MACHINE ERROR

# 36

Java.lang.VirtualMachineError is thrown when Java virtual machine encounters any internal error or resource limitation which prevents it from functioning. It's a self-defensive mechanism employed by JVM to prevent entire application from crashing. In this article let's discuss different types of VirtualMachineError, their characteristics, reasons why they get triggered and potential solutions to fix them.

## Types of VirtualMachineError

There are four different types of VirtualMachineError:

- OutOfMemoryError
- StackOverflowError
- InternalError
- UnknownError

Let's review these types in detail in this section

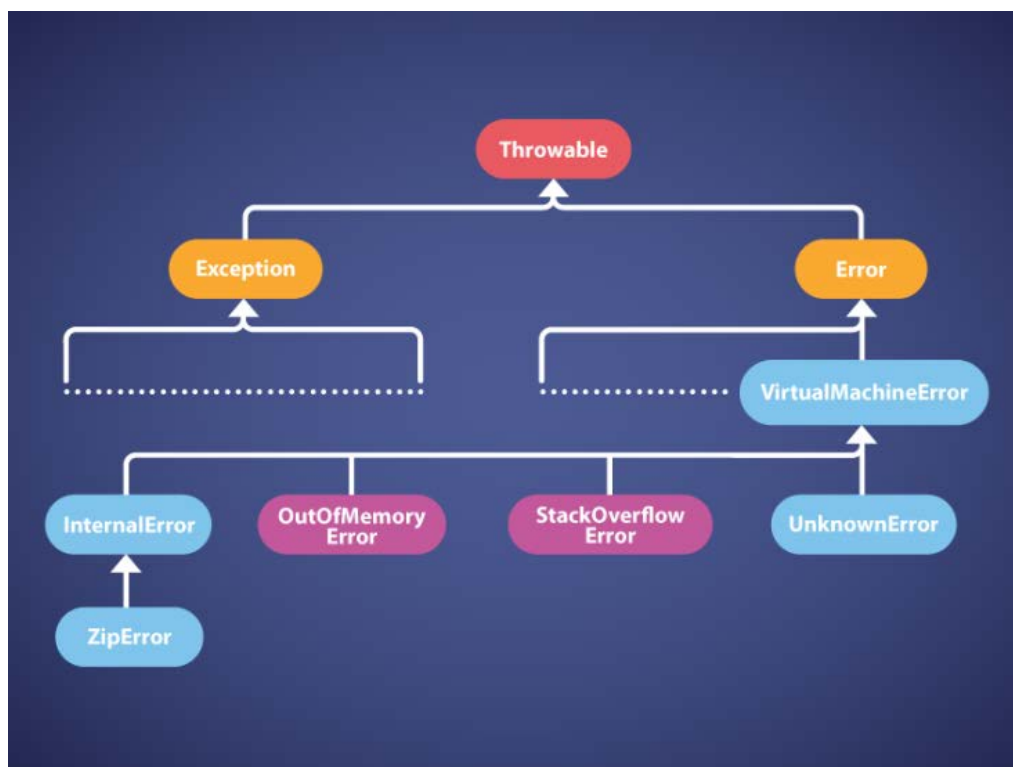


Fig: Java Throwable class hierarchy

## # a. OutOfMemoryError

Just like OMG (Oh My God) acronym, OOM (OutOfMemoryError) is quite popular among DevOps community :-). Most DevOps engineers think that there is one OutOfMemoryError. But there are 8 different flavors of OutOfMemoryError:

1. java.lang.OutOfMemoryError: Java heap space
2. java.lang.OutOfMemoryError: GC Overhead limit exceeded
3. java.lang.OutOfMemoryError: Requested array size exceeds VM limit
4. java.lang.OutOfMemoryError: Permgen space
5. java.lang.OutOfMemoryError: Metaspace
6. java.lang.OutOfMemoryError: Unable to create new native thread
7. java.lang.OutOfMemoryError: Kill process or sacrifice child
8. java.lang.OutOfMemoryError: reason stack\_trace\_with\_native\_method

Each flavor is triggered for different reasons. Similarly, solutions are also different for each flavor of OutOfMemoryError. Here is a beautiful one-page document that summarizes all different flavors of OutOfMemoryError, their causes and solutions.

In general, OutOfMemoryError can be diagnosed and fixed by analyzing Garbage Collection logs and Heap Dumps. Since analyzing Garbage Collection logs manually can be tedious, you may consider using free tools like: GCeasy, HP Jmeter, IBM GC analyzer. Similarly to analyze heap dumps, you may consider using free tools like: HeapHero, Eclipse MAT.

## # b. StackOverflowError

Thread's stack is storing information about the methods it's executing, primitive datatype values, local variables, object pointers, and return values. All of them consume memory. If thread's stack sizes grow beyond the allocated memory limit, then java.lang.StackOverflowError is thrown. This problem typically happens when a thread recursively invokes same function again and again as a result of a bug in the executing program. More details on how to debug StackOverflowError and all possible solutions to fix it can be found in this article.

## # c. InternalError

java.lang.InternalError is thrown by JVM when there is a:

- a. Fault in the software implementing the virtual machine,
- b. Fault in the underlying host system software
- c. Fault in the hardware.

But rarely you will encounter InternalError. To understand what specific scenarios may cause InternalError, you may search for 'InternalError' string in Oracle's Java Bug database. At the time of writing this article (Dec' 20, 2018), there are only 200 defects reported for this error in Oracle java bug database. Most of them are fixed.

## # d. UnknownError

java.lang.UnknownError is thrown when an exception or error has occurred, but the Java virtual machine is unable to report the actual exception or error. Seldom you will see UnknownError. In fact, when searching for 'UnknownError' in Oracle Java Bug database at the time of writing this article (Dec' 20, 2018), there are only 2 defects found reported.

## Characteristics

VirtualMachineError has couple of primary characteristics:

1. Unchecked Exception
2. Synchronous & asynchronous delivery

Let's discuss these two characteristics in this section.

### (1). Unchecked Exception

There are two types of Exceptions:

1. Checked exceptions
2. Unchecked exceptions

Exceptions which are checked at compile time called Checked Exception. If some methods in your code throws a checked exception, then the method must either handle the exception or it must specify the exception using throws keyword. Examples of the checked exceptions are: IOException, SQLException, DataAccessException, ClassNotFoundException...

Unchecked exceptions do not have this requirement. They don't have to be caught or declared thrown. All types of VirtualMachineError are unchecked exceptions.

### (2). Synchronous & asynchronous Delivery

Exceptions can be thrown in two modes:

1. Synchronous
2. Asynchronous

Synchronous exceptions happen at a specific program statement, no matter, how many number of times program is executed in similar environment. Example of synchronous exceptions are NullPointerException, ArrayIndexOutOfBoundsException, etc....

Asynchronous exceptions can happen at any point in time and it can happen in any part of program statement. There will be no consistency where it can be thrown. All the VirtualMachineError are thrown asynchronously, but sometimes they can also be thrown synchronously. StackOverflowError may be thrown synchronously by method invocation as well as asynchronously due to native method execution or Java Virtual Machine resource limitations. Similarly, OutOfMemoryError may be thrown synchronously during object creation, array creation, class initialization, and boxing conversion, as well as asynchronously.



# REMOTE DEBUGGING JAVA APPLICATIONS

37

Few problems might happen only on test or production servers. It may not be reproducible in your local machine. In those circumstances you want to connect your IDE to the remote test (or production) servers and do remote debugging.

Java applications can be remotely debugged by following these two simple steps:

Pass remote debugging arguments to JVM

Configure IDE

Let's review these two steps in this article.

## Step 1: Pass remote debugging arguments to JVM

Typically, you would launch your java application like this:

```
java -jar app.jar
```

To enable remote debugging you need to pass these additional arguments:

```
java -Xdebug -Xrunjdwp:server=y,transport=dt_socket,address=8000,suspend=n -jar app.jar
```

**-Xdebug:** Enables JVM to do remote debugging.

**-Xrunjdwp:** Specifies the connectivity details:

**Transport:** Configures transport between application and debugger. It can have 2 values: 'dt\_socket' or 'shmem'. 'dt\_socket' instructs to socket interface. 'shmem' will instruct application and debugger interace through shared memory region, which is useful only when both application and debugger are running on same machine.

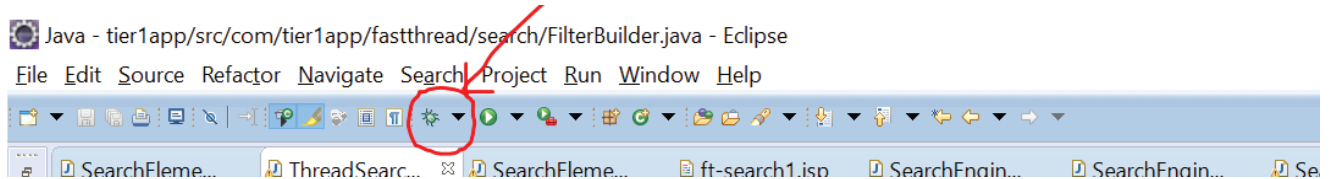
**Address:** Port which will be opened by the application for remote debugging.

**Suspend:** It can have two values. 'y' means application will be suspended until any remote debugger is connected to the application. 'n' means application will not be suspended even if no remote debugger is connected to the application.

## Step 2: Configure IDE

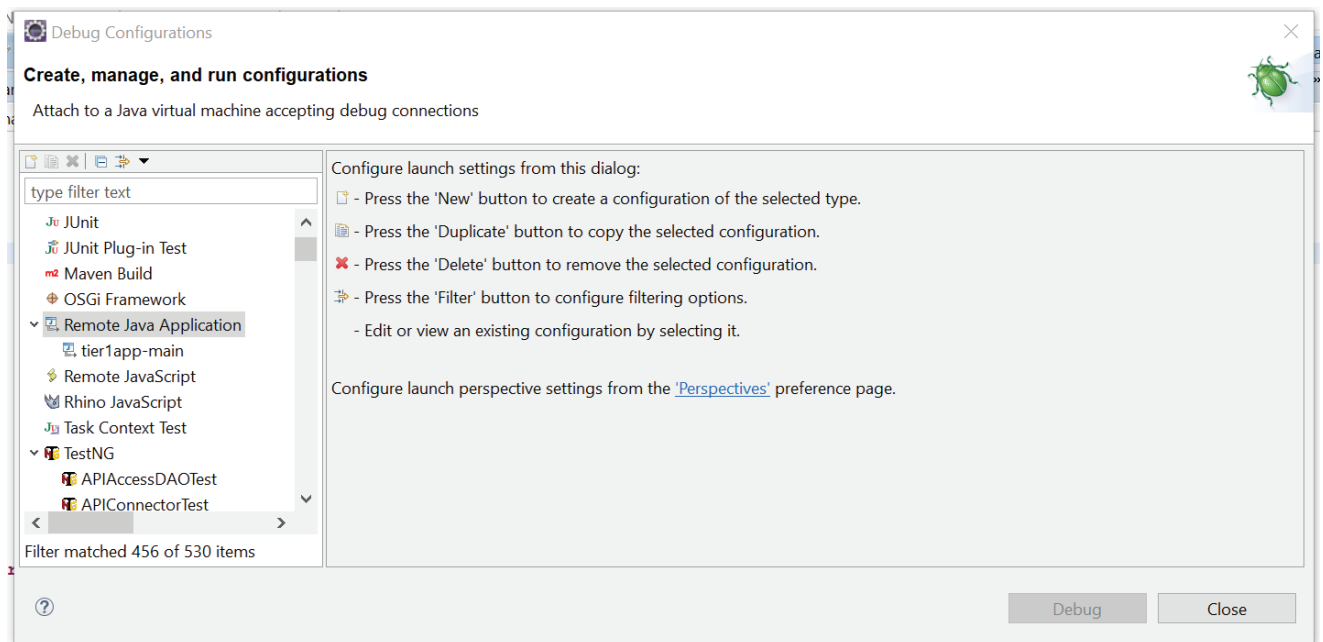
Below are the steps to configure Eclipse IDE to connect to your remote application:

- (1). Click on the Debug menu icon

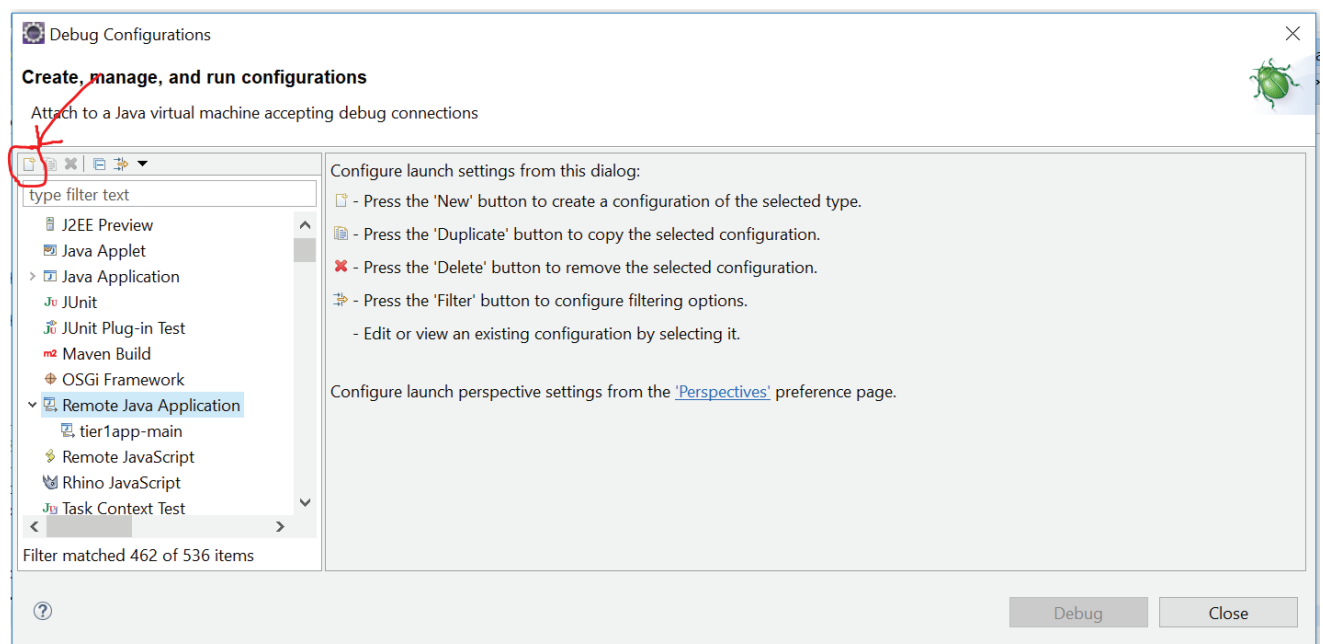


- (2). Click on 'Debug Configurations...' menu item

- (3). In the left panel select 'Remote Java Application'

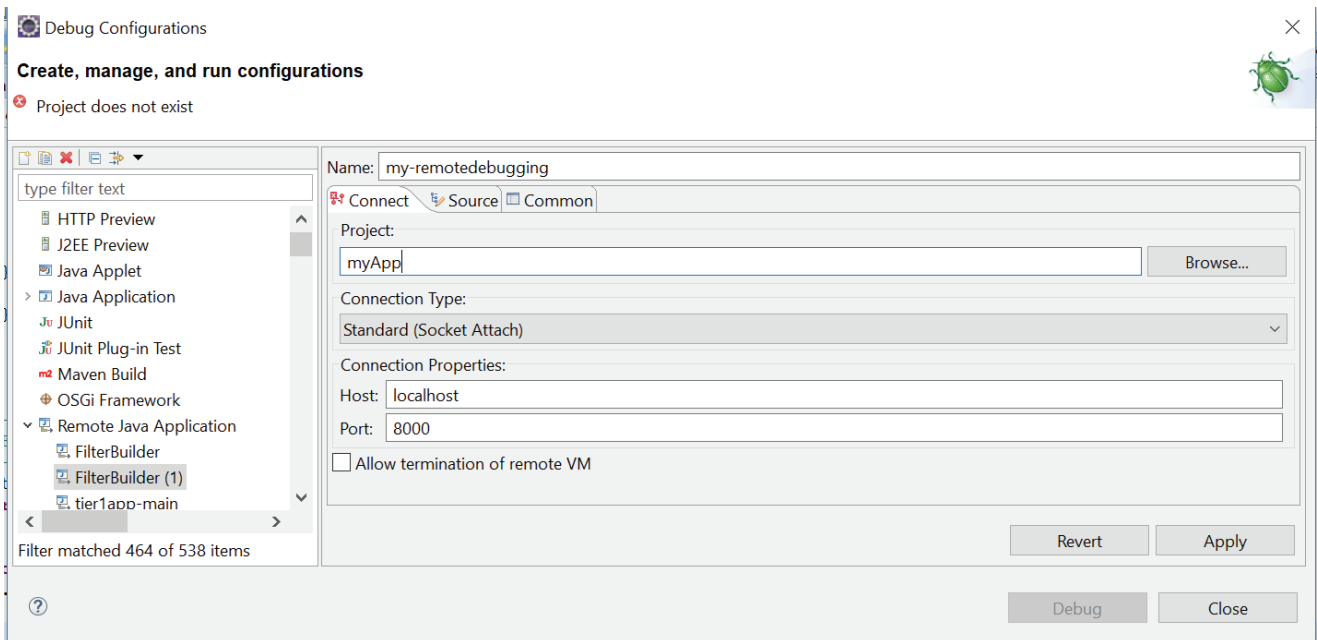


- (4). Press the 'New' button



(5). Now you need enter project and connectivity details:

- a. In the 'Name' field you can enter any name. Example: myapp-remotedebugging
- b. In the 'Project' field select your applications source code that you want to debug.
- c. In the 'Host' field enter the hostname in which your application is running.



(6). After entering all these details click on the 'Debug' button.

That's it. Now you are all set for doing remote debugging. Wish you 'Happy Debugging'. Hopefully it's not that painful.

Warning:

Don't keep remote debugging JVM arguments ON always, as it has following downsides:

- a. Remote debugging mode disables several optimizations that JVM does to application to optimize the performance. All those optimizations will be lost.
- b. Remote debugging opens up a port. It's a security risk, as anyone who can hit the server can initiate remote debugging.