Memory Analysis using Eclipse Memory Analyzer and VisualVM Tools

Mehak Agarwal, Texas State University
Trupti Mhaisdhune, Texas State University

1 INTRODUCTION

1.1 Overview of Tools

We have worked with two tools Eclipse Memory Analyzer Tool and Visual VM for analysis of issues related to memory leak and high memory consumption.

1.1.1 Eclipse Memory Analyzer tool(MAT) :

The Eclipse Memory Analyzer is a fast and feature-rich Java heap analyzer that helps you find memory leaks and reduce memory consumption. The Memory Analyzer provides a general-purpose toolkit to analyze Java heap dumps. Besides heap walking and fast calculation of retained sizes, the Eclipse tool reports leak suspects and memory consumption anti-patterns. The main area of application is OutOfMemoryError and high memory consumption. It helps us to see who is preventing the Garbage Collector from collecting objects, run a report to automatically extract leak suspects. It shows the biggest objects in the application i.e. objects having the largest memory consumption. It also helps to find the responsible objects, analysis of threads, analysis of collection usage, analyzing finalizer and details of soft references.

1.1.2 Visual VM :

VisualVM is used by the developer to monitor and troubleshoot application build in Java platform. VisualVM generates and analyzes heap dumps, track down memory leaks, it also performs and monitors garbage collection. VisualVM provides a visual interface that gives details about java application while they are running on JVM where one can monitor the memory required by an application running on JVM. It also allows to take a snapshot of heap dump and save it for monitoring heaps. Heap dump snapshot give information of all object in the JVM heap at a certain point of time. JVM allocates the memory for objects from the heap for all class instances, then garbage collector reclaims the heap memory if the object is no longer used and no references to that object are present. So, examining this heap with the VisualVM helps to locate the object and their references in the source file. VisualVM includes profiler which allows analyzing the runtime usage of memory and CPU.

1.2 Overview of analyzed problem

We have analyzed problems in Java applications which are related to memory leak and high memory consumption. Application faults which occur at runtime due to memory issues are very complex and cannot be uncovered easily. Small application with memory leak might not be a big issue if the operating system has enough memory to run it. But for applications which run continuously on the system and consumes additional memory over time might cause some serious problems for an application or the whole system. Applications with memory leak and high memory consumption have a performance impact and are not reliable to use, they have very high chances of crashing, these applications bring down overall performance of the system on which they are running. We tried to simulate these problems through some codes which are explained later in the report. Although these problems are easy to find in these sample codes, when they are a part of bigger real-world complex problems then they cannot be uncovered by just going through the source code. So, we have tried to find out the issues by using the Eclipse MAT and VisualVM tools described above.

2 OBSERVATIONS

2.1 Installing the tools

2.1.1 Eclipse Memory Analyzer:

Eclipse memory analyzer can be installed as a stand-alone Application and can also be installed as a plugin with the full-fledged eclipse IDE.

2.1.1.1 Steps to install as a plugin:

- Go to Help -> Install New Software in Eclipse. Then add http://download.eclipse.org/mat/1.7/update-site/ as the location of Repository in the dialog box.
- Then follow the Dialog box as shown below. Accept the License agreement and restart Eclipse.

2.1.1.2 Installing the tool as Standalone Application:

- Download zip file as shown in the snapshot.
- Extract the zip folder.
- Launch the application by double clicking the exe file \MemoryAnalyzer-1.7.0.20170613-win32.win32.x86_64\-mat\MemoryAnalyzer.

2.1.2 VisualVM:

VisualVM is open source tool available in two distribution one is at GitHub and second as a JDK tool.

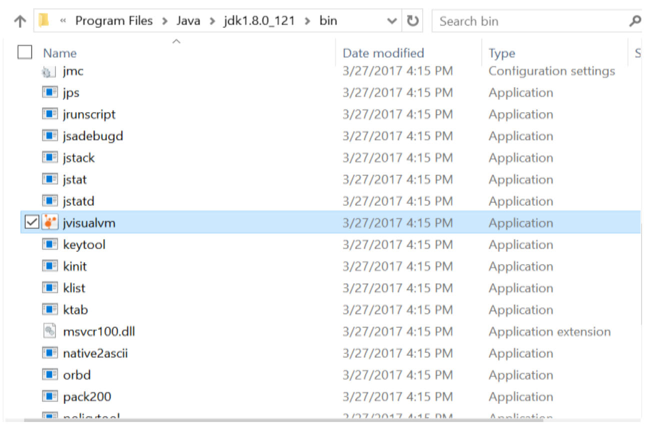
2.1.2.1 Installing VisualVM from Github:

- Download the VisualVM installer from the link: https://visualvm.github.io/-download.html

- Extract the downloaded installer to an empty directory on your local system. After extraction executable VisualVM file is available in the bin folder.

2.1.2.2 Installing VisualVM as JDK tool:

- VisualVM is automatically available after JDK is installed in local system and JDK can be downloaded from http://www.oracle.com/ technetwork/ java/ javase/downloads/jdk9-downloads-3848520.html
- The executable file for Visual VM could be found on <JDK installation folder>/bin as displayed.



Name	Date modified	Type
jmc	3/27/2017 4:15 PM	Configuration settings
jps	3/27/2017 4:15 PM	Application
jrunscript	3/27/2017 4:15 PM	Application
jsadebugd	3/27/2017 4:15 PM	Application
jstack	3/27/2017 4:15 PM	Application
jstat	3/27/2017 4:15 PM	Application
jstatd	3/27/2017 4:15 PM	Application
jvisualvm	3/27/2017 4:15 PM	Application
keytool	3/27/2017 4:15 PM	Application
kinit	3/27/2017 4:15 PM	Application
klist	3/27/2017 4:15 PM	Application
ktab	3/27/2017 4:15 PM	Application
msvcr100.dll	3/27/2017 4:15 PM	Application extension
native2ascii	3/27/2017 4:15 PM	Application
orbd	3/27/2017 4:15 PM	Application
pack200	3/27/2017 4:15 PM	Application

Figure 1: Locate VisualVM in jdk installation directory

2.2 Learning the tools

2.2.1 Eclipse MAT:

EMAT works with Heap Dump. A heap dump is a snapshot of the memory of a Java process at a certain time. There are different formats for persisting this data, and depending on the format it may contain different pieces of information, but in general, the snapshot contains information about the java objects and classes in the heap at the moment the snapshot was triggered. Usually, a full GC is triggered before the heap dump is written so it contains information about the remaining objects.

The Memory Analyzer is able to work with HPROF binary heap dumps, IBM system dumps (after preprocessing them), and IBM portable heap dumps (PHD) from a variety of platforms.

Typical information which can be found in heap dumps (one more - depending on the heap dump type) is: All Objects (Class, fields, primitive values and references), all classes (Classloader, superclass, static fields), Garbage Collection Roots, Objects defined to be reachable by the JVM, Thread Stacks (the call-stacks of threads at the moment of the snapshot, and per-frame information about local objects).

A heap dump does not contain allocation information, so it cannot resolve questions like who had created the objects and where they have been created. Such a heap dump contains information about all Java objects alive at a given point in time. All current Java Virtual Machines can write heap dumps, but the exact steps depend on vendor, version and operation system.

2.2.1.1 Acquiring Heap Dump from Memory Analyzer:

If the Java process from which the heap dump is to be acquired is on the same machine as the Memory Analyzer, it is possible to acquire a heap dump directly from the Memory Analyzer. Dumps acquired this way are directly parsed and opened in the tool.

Acquiring the heap dump is a VM specific. To trigger a heap dump from Memory Analyzer open the File > Acquire Heap Dump... menu item.

We can select a process for which heap dump should be acquired, provide a preferred location for the heap dump and press Finish to acquire the dump. Some of the heap dump providers may allow (or require) additional parameters (e.g. type of the heap dump) to be set. This can be done by going to the Next page of the wizard.

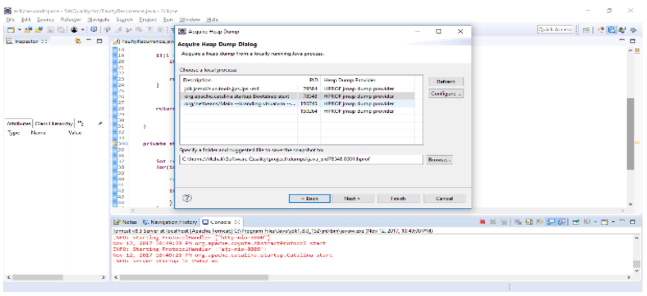


Figure 2: Acquiring heap dump using Memory Analyzer

If you are using Memory Analyzer installed in Eclipse rather than a stand-alone Memory Analyzer, first open the 'Memory Analysis' perspective using: Window > Perspective > Open Perspective > Other ... > Memory Analysis.

MAT uses this heap dump as an input and parses and process it to generate reports and charts/table to give detailed visual information about the Java heap of an application.

2.2.1.2 Working with EMAT:

Overview Tab on the right, you'll find the size of the dump and the number of classes, objects and class loaders. Right below, the pie chart gives an impression on the biggest objects in the dump. Move your mouse over a slice to see the details of the objects in the object inspector on the left. Click on any slice to drill down and follow for example the outgoing references.

The Histogram lists the number of instances per class, the shallow size and the retained size. The Memory Analyzer displays by default the retained size of individual objects.

Using the **context menu**, one can drill-down into the set of objects which the selected row represents. For example, you can list the objects with outgoing or incoming references. Or group the objects by the value of an attribute. Or group the collections by their size.

The Dominator Tree displays the biggest objects in the heap dump. The next level of the tree lists those objects that would-be garbage collected if all incoming references to the parent node were removed. The dominator tree is a powerful tool to investigate which objects keep which other objects alive. Again, the tree can be grouped by class loader (e.g. components) and packages to ease the analysis.

MAT also generates Leak suspects report, which helps us in finding root cause of memory leak

2.2.2 Visual VM:

2.2.2.1 Starting VisualVM:

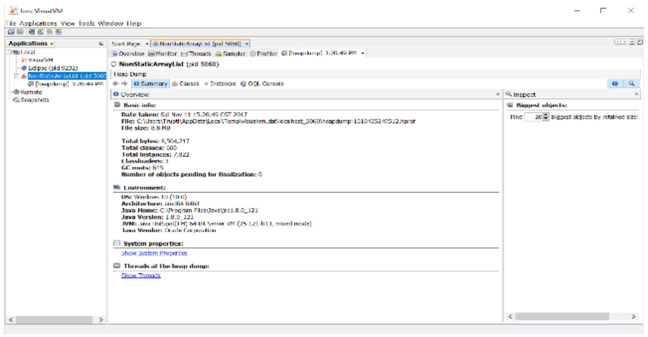
- VisualVM can be started by running VisualVM.exe Application from<VisualVM installer folder>/ bin or <JDK installation folder>/bin.
- After starting VisualVM, Window opens with Application window on left side and main window on right side.

2.2.2.2 Application window:

Application window on left enables us to quickly view the java applications running on local as well as remote system. Remote applications can be made available simply by providing hostname or IP address by right click on a remote node.

2.2.2.3 Generating heap dumps:

To generate heap dump, right-click the targeted java application in the applications window and select Heap Dump. It will then generate new heap dump for java application and open it in the main window at the left side.



NonStaticArrayList (pid 5060)

Heap Dump

Basic info:

Date taken: Sat Nov 11 15:20:49 CST 2017
File: C:\Users\Trupti\AppData\Local\Temp\visualvm.dat\localhost_5060\heapdump-1510435249512.hprof
File size: 8.9 MB

Total bytes: 8,504,217
Total classes: 660
Total instances: 7,822
Classloaders: 3
GC roots: 615
Number of objects pending for finalization: 0

Environment:

OS: Windows 10 (10.0)
Architecture: amd64 64bit
Java Home: C:\Program Files\Java\jre1.8.0_121
Java Version: 1.8.0_121
JVM: Java HotSpot(TM) 64-Bit Server VM (25.121-b13, mixed mode)
Java Vendor: Oracle Corporation

Figure 3: Summary view of heap dump

After generating heap dump, summary view of heap dumps opens which provides information about size, number of class instances etc.

The class view provides a list of the classes, the number and percentage of instances referenced by that class, and the size of classes. Instances of a particular class can be viewed by double click on that class or right-clicking the name and choosing Show in Instances View.

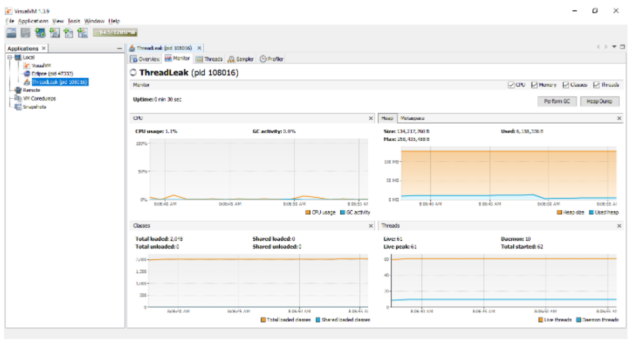


Figure 4: Heap dump, Thread Dump

The Instance view tab shows all object instances for a selected class. By selecting an instance from the Instance pane, the fields of that class and references to that class is displayed.

The heap size and used heap for running application can be viewed in monitor tab as shown in figure 4.

3 ANALYZING THE PROBLEM USING TOOLS

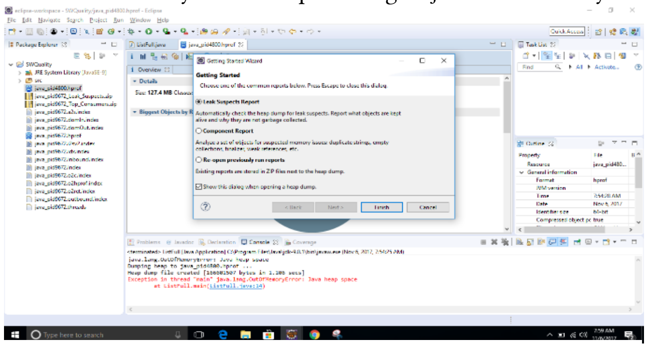
Figure 4: Monitoring Heap memory of running application

We have studied both the tools on four sample programs which simulate memory leak and high memory consumption issues related to Java Application.

3.1 Sample 1

ListFull.java

It Continuously adds Unique String objects in an ArrayList



```
<terminated> ListFull [Java Application] C:\Program Files\Java\jdk-9.0.1\bin\javaw.exe (Nov 6, 2017, 7:54:25 AM)
java.lang.OutOfMemoryError: Java heap space
Dumping heap to java_pid4800.hprof ...
Heap dump file created [108081507 bytes in 1.108 secs]
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
	at ListFull.main(ListFull.java:14)
```

Getting Started Wizard

Getting Started

Leak Suspects Report

Automatically check the heap dump for leak suspects. Report what objects are kept alive and why they are not garbage collected.

Component Report

Analyze a set of objects for suspected memory issues: duplicate strings, empty collections, finalizer, weak references, etc.

Re-open previously run reports

Existing reports are stored in ZIP files next to the heap dump.

Show this dialog when opening a heap dump.

Figure 5: Generating heap dump for sample 1

until Out of memory error is thrown by the application and application gets terminated.

Using MAT:

When this sample code is executed, it got terminated with OutOfMemoryError and heap dump was generated and stored in file system. Heap dump was then imported into MAT.

Following VM Argument was provided to this sample code execution to enable heap dump generation and to restrict heap size to 128 MB. -XX:+HeapDumpOnOutOfMemoryError -

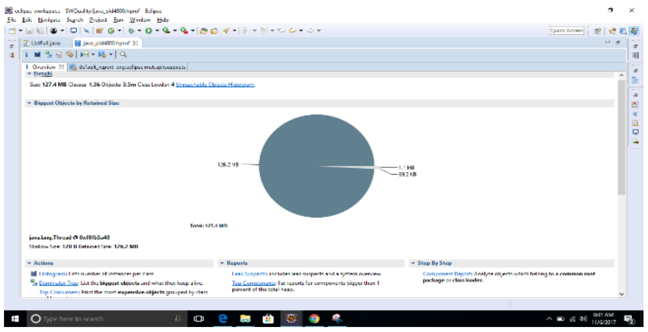


Figure 6: Pie Chart showing the biggest object in dump

Xms128m -Xmx128m. Once dump import is completed, MAT gives the option to generate any specific report for analysis. We selected Leak Suspects report and clicked finished.

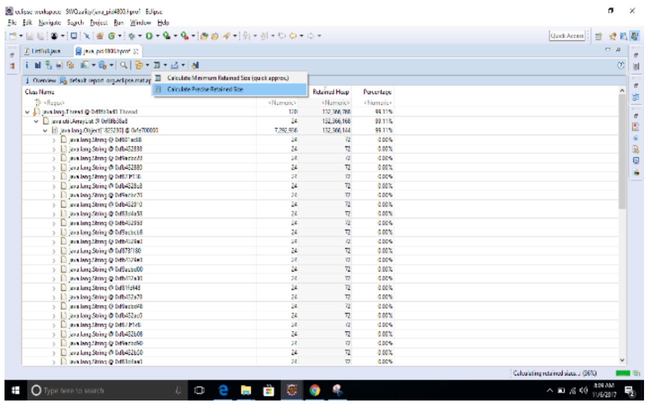


Figure 7: Dominator tree for the heap dump

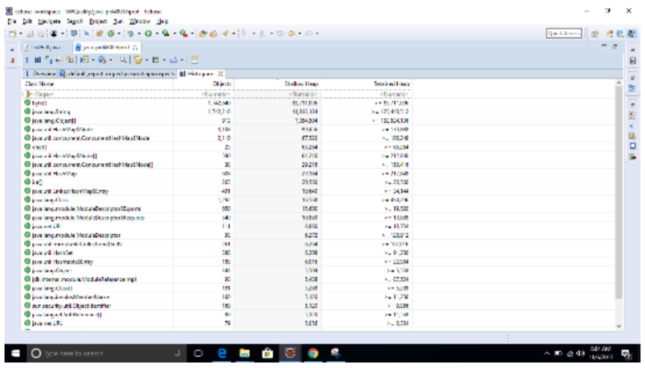
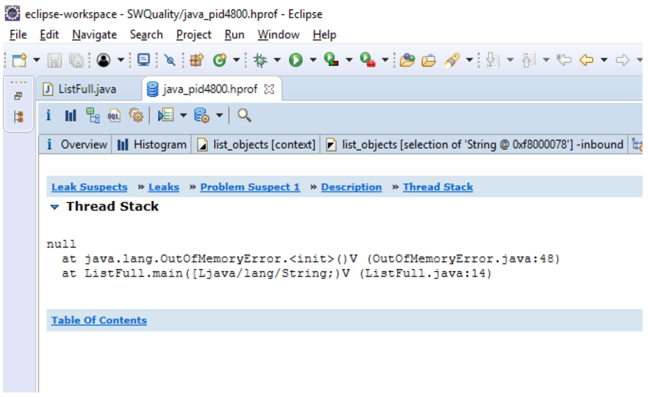


Figure 8: Histogram generated from the heap dump

Using Memory Analyzer we can drill down and slice the objects to do further analysis. As per the histogram snapshot shown in Figure **8** its evident that String Objects are consuming maximum heap space.

Using the dominator tree as in the Figure 7 it can be seen that an ArrayList object is consuming maximum heap memory by storing a large number of string objects.



Leak Suspects » Leaks » Problem Suspect 1 » Description » Thread Stack

Thread Stack

```
null
  at java.lang.OutOfMemoryError.<init>()V (OutOfMemoryError.java:48)
  at ListFull.main([Ljava/lang/String;)V (ListFull.java:14)
```

Table Of Contents

Figure 9: Thread Stack generated by MAT

We also analyzed the Thread stack as it tells exact cause of the OutOfMemory error, which here is the accumulation of a large number of String objects in an ArrayList in the Main method of a ListFull java class.

Using VisualVM:

VisualVM has a visualizer that enables you to easily browse heap dumps. Same Heap dump was loaded into VisualVM using menu option File -> Load and then select the heap dump file. VisualVM opens the heap dump in a new tab and creates a node for the heap dump under the application node in the Applications window.

The Classes view displays a list of classes and the number and percentage of instances referenced by that class. As per snapshot,

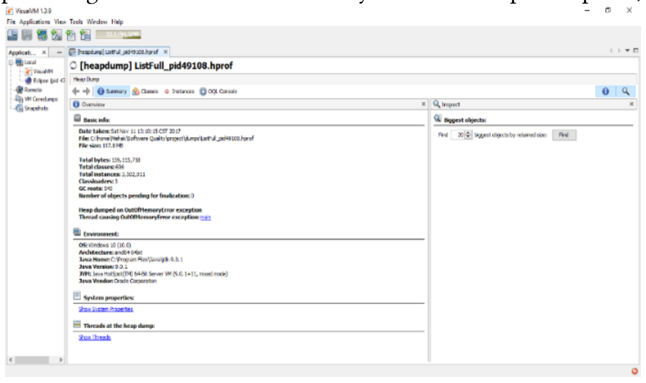


Figure 10: Heap dump summery for sample 1

we can see that String objects are consuming maximum heap space.

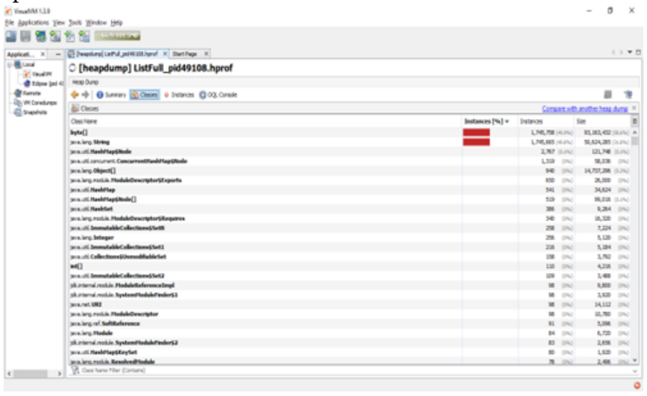


Figure 11: Class view for sample 1

The Instance view displays object instances for a selected class.

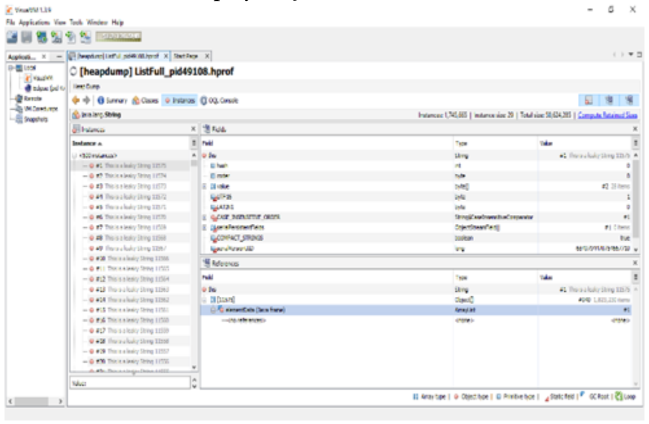


Figure 12: Instance view for sample 1

Select one suspect object i.e "This is a leak String11575", go to reference view and navigate to the object which stores these object. "elementData" variable of type ArrayList is holding these string objects, by right-clicking elementData->Show in threads.

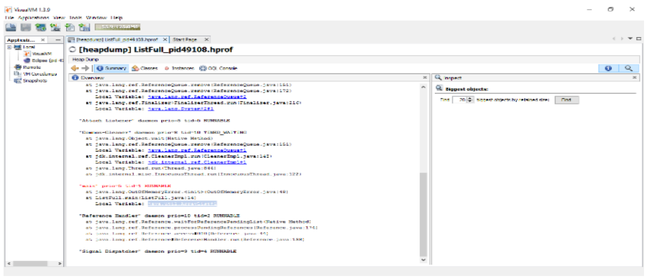


Figure 13: Thread stack for sample 1

This thread stack showed us exact source code location where these objects are being added.

3.2. Sample 2

ThreadLeak.java

It Creates 50 non-daemon threads which are not terminated properly and prevents stopping of application.

Using MAT:

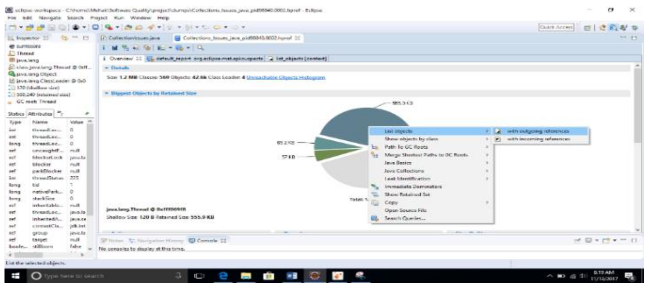


Figure 14: Options available for each slice of Pie chart

We acquired a heap dump when main thread died and worker threads were still running and preventing the application from

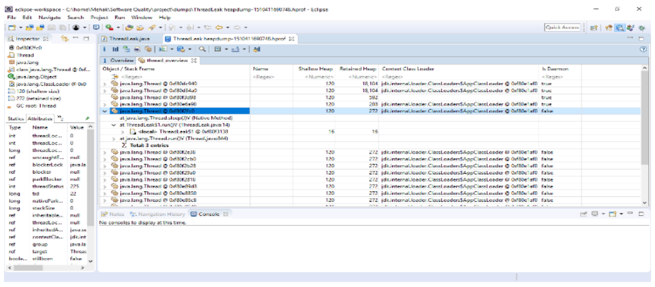


Figure 15: Thread Overview for sample 2

stopping.

When same heap dump was loaded, its thread overview tab presented detail of all the threads which were still running. All non-daemon threads were the worker thread and then as per the snapshot, we found the exact place in the code where these were generated.

Using VisualVM:

The Same heap dump is loaded in VisualVM and navigated to "Threads at the heap dump:" section of Summary tab, it gives detail of each and every thread running in the application, which helped in finding the exact location in the source code where these threads were executing.

We also loaded the running process in VisualVM and Monitor tab presented real-time information of CPU, Memory, classes, and GC of the process.

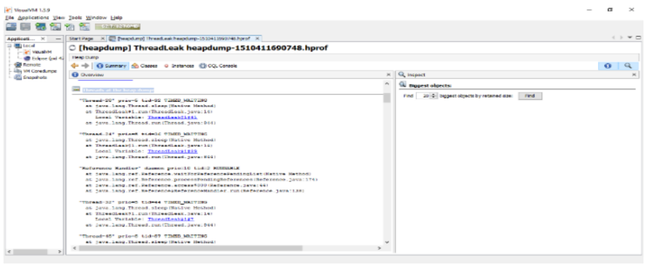


Figure 16: Threads at heap dump in sample 2

Threads tab present visual view of all the threads in the application, User worker threads were identified and were in

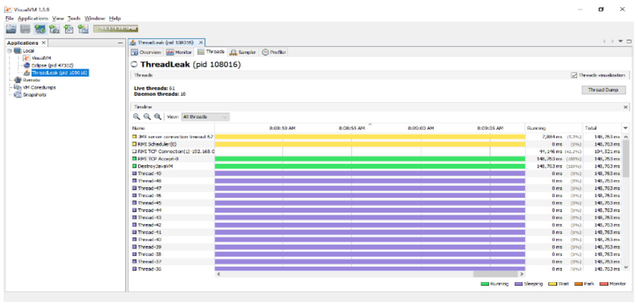


Figure 17: Status of threads for sample 2

sleeping state.

We can select any of the user thread and view its thread dump, which helps us in finding the exact location in the source code where threads are currently executing.

3.3 Sample 3

CollectionIssues.java

It creates 2 Map Object, First Map stores String as keys and List of Integers as value, some of the values in this Map are an empty list. Second Map stores custom keys to simulate map key collision issue. A timer was added to delay the processing and to avoid getting OutOfMemoryError and after some time heap dump was acquired.

Using Eclipse MAT:

Heap dump was loaded in Eclipse MAT. Overview tab pointed that main thread object is consuming the maximum amount of heap space. We clicked the same section of the pie chart and then navigated to "List outgoing references" as shown in Figure 18, which shows both the Map objects with other relevant information like shallow Heap size and retained heap size.

On Each Map object, we have a number of option to explore the contents and properties of these object. As shown in snapshot

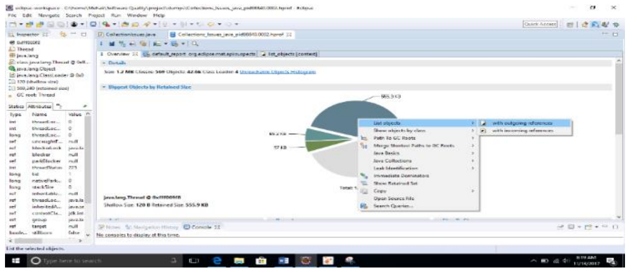
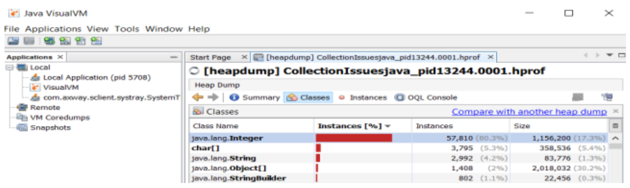


Figure 18: Analyzing the outgoing references of objects

"Java Collections" option has sub-functions like Array Fill Ratio, Arrays Grouped By Size, Collections Grouped by Size, Extract Hash values etc. These functions are very convenient to investigate collections issue or if collections have poor performance due to hash code collisions.Similarly, Java Basics have functions like Find Strings, Thread details, Group by Value etc. Group by value here helps in finding out that there are multiple Integer objects with same numeric values stored in various lists in a map value. Since Integer objects are immutable and they can be shared safely so there multiple Integer objects with the same value are consuming lots of heap memory and this heap memory consumption can be avoided.

Using VisualVM:



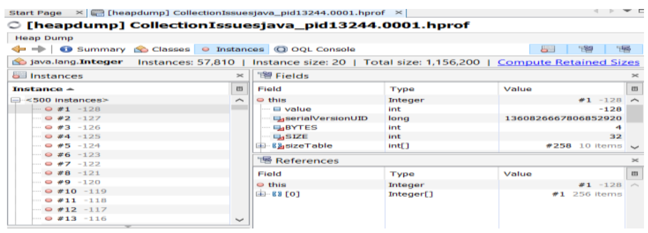
[heapdump] CollectionIssuesjava_pid13244.0001.hprof

Class Name	Instances [%]	Instances	Size
java.lang.Integer		57,810 (80.3%)	1,156,200 (17.3%)
char[]		3,795 (5.3%)	358,536 (5.4%)
java.lang.String		2,992 (4.2%)	83,776 (1.3%)
java.lang.Object[]		1,408 (2%)	2,018,032 (30.2%)
java.lang.StringBuilder		802 (1.1%)	22,456 (0.3%)

Figure 19: Class view for sample 3

When same heap dump is loaded in Visual VM then Classes Tab shows that Integer objects are consuming the maximum amount of heap space.

Double click on Integer row opens Instances view which shows the detail of each and every Integer object in heap. We can then navigate to References view, which shows the detail of fields in the current Integer object. This information is not very relevant since we are interested in finding out that hows these objects are created. Visual VM helps in navigating references in one direction that is from parent to child so the only way to find this issue is if we start from parent Map Object.



Start Page | [heapdump] CollectionIssuesjava_pid13244.0001.hprof

[heapdump] CollectionIssuesjava_pid13244.0001.hprof

Heap Dump

Summary | Classes | Instances | OQL Console

java.lang.Integer Instances: 57,810 | Instance size: 20 | Total size: 1,156,200 | Compute Retained Sizes

Instances

Instance
<500 instances>
#1 -128
#2 -127
#3 -126
#4 -125
#5 -124
#6 -123
#7 -122
#8 -121
#9 -120
#10 -119
#11 -118
#12 -117
#13 -116

Fields

Field	Type	Value
this	Integer	#1 -128
value	int	-128
serialVersionUID	long	1360826667806852920
BYTES	int	4
SIZE	int	32
sizeTable	int[]	#258 10 items

References

Field	Type	Value
this	Integer	#1 -128
[0]	Integer[]	#1 256 items

Figure 20: instances view for sample 3

Also, there are no specific functions to find duplicate values in a map or to extract hash values from a Map keys to find out a hash key collision on a map. From Inspect view we can find biggest objects in a heap and then from these parent objects we can navigate to each and every Integer child object to find out that there are multiple integer objects stored in list object and due to which heap consumption has increased, but it's a kind of manual process for exploring the child objects.

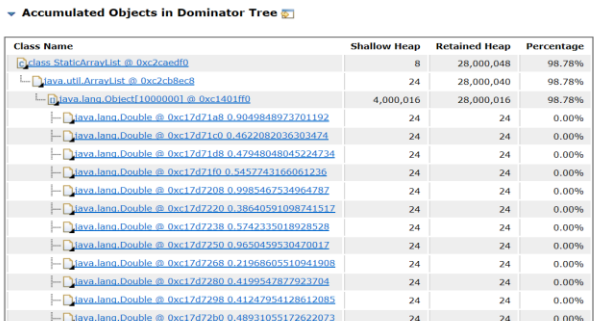
3.4 Sample 4

StaticArrayList.java

This program creates a heavy object with a static field. An ArrayList is created as a static field, which will never be collected by the JVM Garbage Collector, even after the calculations it was used for are done.

Using Eclipse MAT :

When heap of this sample code was generated and loaded in eclipse MAT, it showed that the static field list is not garbage collected and is consuming lot of heap space. Since this is a static field and its scope is global and it will remain in the heap. We can find the responsible object by looking at the histogram or by navigating the main thread. Also after drilling down to the dominator tree for that object it gave detailed idea about the class referring to that object



Accumulated Objects in Dominator Tree

Class Name	Shallow Heap	Retained Heap	Percentage
class StaticArrayList @ 0xc2caedf0	8	28,000,048	98.78%
java.util.ArrayList @ 0xc2cb8ec8	24	28,000,040	98.78%
java.lang.Object[1000000] @ 0xc1401ff0	4,000,016	28,000,016	98.78%
java.lang.Double @ 0xc17d71a8 0.9049848973701192	24	24	0.00%
java.lang.Double @ 0xc17d71c0 0.4622082036303474	24	24	0.00%
java.lang.Double @ 0xc17d71d8 0.47948048045224734	24	24	0.00%
java.lang.Double @ 0xc17d71f0 0.5457743166061236	24	24	0.00%
java.lang.Double @ 0xc17d7208 0.9985467534964787	24	24	0.00%
java.lang.Double @ 0xc17d7220 0.38640591098741517	24	24	0.00%
java.lang.Double @ 0xc17d7238 0.5742335018928528	24	24	0.00%
java.lang.Double @ 0xc17d7250 0.9650459530470017	24	24	0.00%
java.lang.Double @ 0xc17d7268 0.21968605510941908	24	24	0.00%
java.lang.Double @ 0xc17d7280 0.4199547877923704	24	24	0.00%
java.lang.Double @ 0xc17d7298 0.41247954128612085	24	24	0.00%
java.lang.Double @ 0xc17d72b0 0.48931055172622073	24	24	0.00%

Figure 21: Dominator tree for sample 4

Using VisualVM:

With VisualVM we monitored the memory usage by StaticArrayList.java program. We came to notice that as soon as we create the ArrayList object the used memory suddenly increase and never comes down after that point.

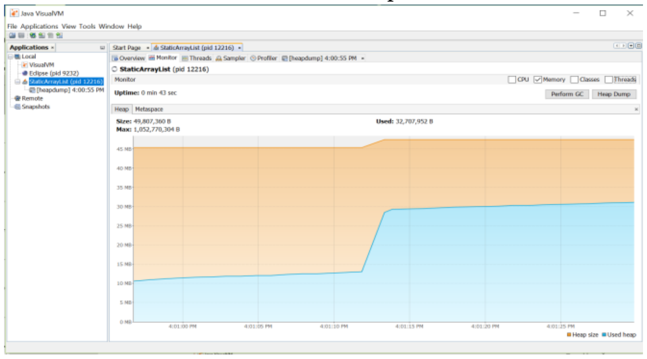
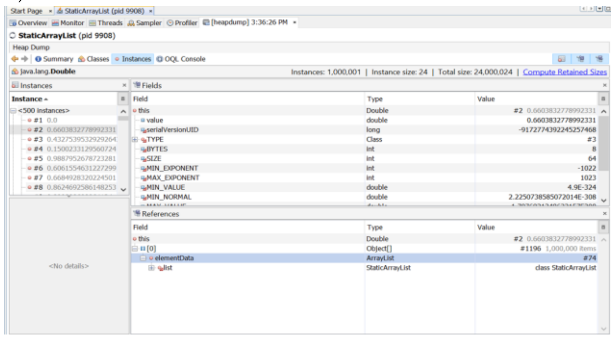


Figure 22: Memory monitoring for sample 4

So we created heap dump at that point of time to gather more information about classes and instances. Here we observed that double class is taking almost 99% of space heap. After analyzing in details about double class it was easy to get the exact location where instances of this class are referred (i.e StaticArrayList class).



Start Page | StaticArrayList (pid 9908)

Overview | Monitor | Threads | Sampler | Profiler | [heapdump] 3:36:26 PM

StaticArrayList (pid 9908)

Heap Dump

Summary | Classes | Instances | OQL Console

java.lang.Double Instances: 1,000,001 | Instance size: 24 | Total size: 24,000,024 | Compute Retained Sizes

Instances

Instance
<500 instances>
#1 0.0
#2 0.6603832778992331
#3 0.4327539532929264
#4 0.1500233129560724
#5 0.9887952678723281
#6 0.6061554631227299
#7 0.6684928320224501
#8 0.8624692586148253

<No details>

Fields

Field	Type	Value
this	Double	#2 0.6603832778992331
value	double	0.6603832778992331
serialVersionUID	long	-9172774392245257468
TYPE	Class	#3
BYTES	int	8
SIZE	int	64
MIN_EXPONENT	int	-1022
MAX_EXPONENT	int	1023
MIN_VALUE	double	4.9E-324
MIN_NORMAL	double	2.2250738585072014E-308

References

Field	Type	Value
this	Double	#2 0.6603832778992331
[0]	Object[]	#1196 1,000,000 items
elementData	ArrayList	#74
list	StaticArrayList	class StaticArrayList

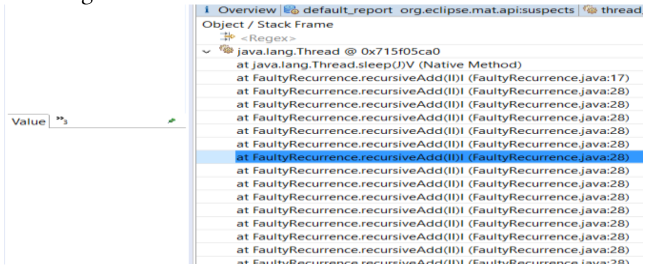
Figure 23: Instances view for sample 4

3.5 Sample 5

FaultyReccurence.java

It calculates the sum of a given Fibonacci series recursively using two approaches one recursively and other is in a loop.

Using MAT:

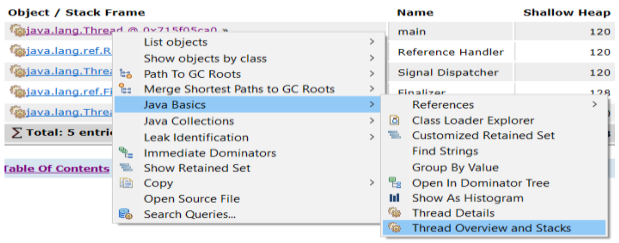


```
Object / Stack Frame
⌄ java.lang.Thread @ 0x715f05ca0
    at java.lang.Thread.sleep(J)V (Native Method)
    at FaultyRecurrence.recursiveAdd(II)I (FaultyRecurrence.java:17)
    at FaultyRecurrence.recursiveAdd(II)I (FaultyRecurrence.java:28)
    at FaultyRecurrence.recursiveAdd(II)I (FaultyRecurrence.java:28)
    at FaultyRecurrence.recursiveAdd(II)I (FaultyRecurrence.java:28)
    at FaultyRecurrence.recursiveAdd(II)I (FaultyRecurrence.java:28)
    at FaultyRecurrence.recursiveAdd(II)I (FaultyRecurrence.java:28)
    at FaultyRecurrence.recursiveAdd(II)I (FaultyRecurrence.java:28)
    at FaultyRecurrence.recursiveAdd(II)I (FaultyRecurrence.java:28)
    at FaultyRecurrence.recursiveAdd(II)I (FaultyRecurrence.java:28)
    at FaultyRecurrence.recursiveAdd(II)I (FaultyRecurrence.java:28)
    at FaultyRecurrence.recursiveAdd(II)I (FaultyRecurrence.java:28)
    at FaultyRecurrence.recursiveAdd(II)I (FaultyRecurrence.java:28)
    at FaultyRecurrence.recursiveAdd(II)I (FaultyRecurrence.java:28)
    at FaultyRecurrence.recursiveAdd(II)I (FaultyRecurrence.java:28)
```

Figure 24: Stack trace generated in Eclipse MAT

We started recursive method and added some delay in code to slow down processing and to get heap dump before StackOverflow error is thrown. Heap dump was loaded in MAT and Leak Suspects report was generated as shown in below snapshot.

We can see that the main non-daemon thread has maximum



Object / Stack Frame	Name	Shallow Heap
java.lang.Thread @ 0x715f05ca0	main	120
java.lang.ref.R	Reference Handler	120
java.lang.Threa	Signal Dispatcher	120
java.lang.ref.Fi	Finalizer	128
java.lang.Threa		
Σ Total: 5 entri		

Figure 25: Step to generate Thread Stack trace using eclipse MAT

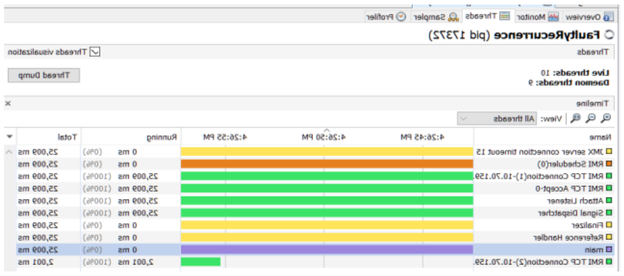
retained heap size and it will grow further with each recursive call since in each call number of local variables are created.

When main thread is selected, we got a number of options to explore the thread further. Thread Overview and Stacks function give detail of exact call stack flow of the execution. But there is no feature to get values of a local variable in each call stack.

Using VisualVM:

A heap dump was acquired by VisualVM when the program was running, Thread View shows main thread is running and then thread dump was generated which listed exact call stack of the main thread this calls stack is similar to the call stack shown by MAT and here too we cannot see the local stack variables value.

But VisualVM has an option for Profiling and by using it we can profile our code to find out the exact portion of code which was being executed and consuming resources.



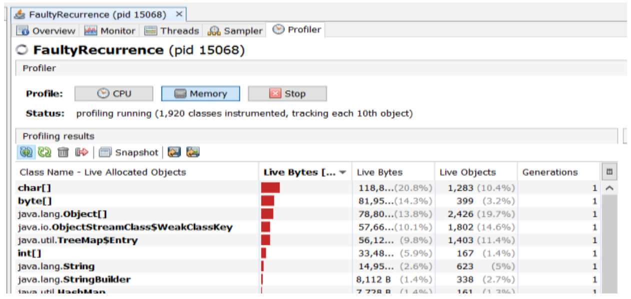
FaultyRecurrence (pid 17372)

Threads

Live threads: 10
Daemon threads: 9

Name	Running		Total
JMX server connection timeout 15	0 ms	(0%)	25,009 ms
RMI Scheduler(0)	0 ms	(0%)	25,009 ms
RMI TCP Connection(1)-10.70.159.	25,009 ms	(100%)	25,009 ms
RMI TCP Accept-0	25,009 ms	(100%)	25,009 ms
Attach Listener	25,009 ms	(100%)	25,009 ms
Signal Dispatcher	25,009 ms	(100%)	25,009 ms
Finalizer	0 ms	(0%)	25,009 ms
Reference Handler	0 ms	(0%)	25,009 ms
main	0 ms	(0%)	25,009 ms
RMI TCP Connection(2)-10.70.159.	2,001 ms	(100%)	2,001 ms

Figure 26: Status of threads for sample 5



FaultyRecurrence (pid 15068)

Profiler

Profile:

Status: profiling running (1,920 classes instrumented, tracking each 10th object)

Profiling results

Class Name - Live Allocated Objects	Live Bytes [...	Live Bytes	Live Objects	Generations
char[]		118,8...(20.8%)	1,283 (10.4%)	1
byte[]		81,95...(14.3%)	399 (3.2%)	1
java.lang.Object[]		78,80...(13.8%)	2,426 (19.7%)	1
java.io.ObjectStreamClass$WeakClassKey		57,66...(10.1%)	1,802 (14.6%)	1
java.util.TreeMap$Entry		56,12... (9.8%)	1,403 (11.4%)	1
int[]		33,48... (5.9%)	167 (1.4%)	1
java.lang.String		14,95... (2.6%)	623 (5%)	1
java.lang.StringBuilder		8,112 B (1.4%)	338 (2.7%)	1
java.util.HashMap		7,728 B (1.4%)	161 (1.3%)	1

Figure 27: Memory profiling for sample 5

4 ANALYSIS OF RESULTS

Both the tools can successfully drill down to the possible location in the source code that may cause memory leaks in the application.

4.1. Comparison:

Memory analyzer and VisualVM both provides the visual interaction to analyze the heap dump generated from running application. Both tools provide information about total heap size, used size, number of classes and their instances loaded in heap, size if each instance, biggest object.

4.1.1 Advantages of Memory Analyzer over VisualVM

MAT has special functions for collections analysis like fill ratio, hash code collision ratio, soft reference statistics etc. but these kinds of special functions are not available in VisualVM. Also Eclipse MAT gives the Shallow heap size for objects along with the Retained size for objects.

4.1.2 Advantage of VisualVM over memory analyzer

VisualVM enables you to monitor the heap memory usage at runtime. By monitoring heap usage, one can create heap dump at any time if there is any sudden change in heap memory usage. Other than memory usage VisualVM also provides the ability to monitor GC activity and CPU usage. Garbage collection activity can be performed from the VisualVM interface.

4.2 Strengths:

Eclipse Memory Analyzer:

- Eclipse Memory Analyzer is a powerful tool to find memory leaks and reduce memory consumption.
- Memory Analyzer provides a good understanding of memory usage as well as analysis summaries. One thing that makes the Memory Analyzer so powerful is the fact that one can run any action on any set of objects. Just drill down and slice your objects the way you need them.
- This tool provides object tree browsing which allows understanding the reference relationships between Java objects, interactions and the memory requirements of that objects.
- It has a feature Path to GC roots, which explains why a particular object cannot be garbage collected.
- It also provides various queries for analyzing java collections.

VisualVM:

- It provides good graphical representation.
- It provides a better view of the used heap vs. heap size while the application is running. Additionally, a heap dump can be taken at any time to provide a snapshot of CPU or memory usage
- The Threads tab provides a running timeline and shows all live threads by color code to indicate the state (running, sleeping etc.) of threads
- It also provides sampler and profiler tab.
- It gives a functionality to monitor memory usage of an application running on the remote system.

4.3 Weakness:

Eclipse Memory analyzer:

- It shows the objects having largest memory consumption and leak suspects but does not show the exact name of the variable as in the source code.
- It does not give the details about local variables.

VisualVM:

- The documentation for this tool does not explain in details how to analyze the generated heap dump.
- It does not show the exact errors causing memory leaks

4.4 Suggested improvements

Even though these tools Eclipse Memory Analyzer and VisualVM are used to find out the possible reference objects causing memory leaks these tools does not give exact variables and reasons that causes memory problems. So, developers have to analyze these problems manually to correct it. It could be a better improvement if these tools give mains cause for problem and suggestions about how those problems could be solved.

5 CONCLUSION

Eclipse Memory Analyzer and VisualVM both are valuable tools which can provide the programmer with deep understanding of application in terms of CPU performance, memory usage, and threads. Using the Memory Analyzer, we can analyze productive heap dumps with hundreds of millions of objects and calculate the retained sizes of objects, see who is preventing the Garbage Collector from collecting objects, run a report to automatically extract leak suspects and analyze collections and threads. With VisualVM you can see all running java processes and you can connect to them, take heap dumps, analyze CPU and memory performance.

Appendix

Link for sample java files:
https://github.com/btrupti/Software-Quality-Project/tree/master/Software-Quality-Project

REFERENCES

[1] Eclipse Memory Analyzer Documentation:
https://www.eclipse.org/mat/documentation/

[2] VisualVM Documentation:
https://visualvm.github.io/documentation.html