Testing in DevOps

DOu – Certified Tester in DevOps (CTD)
Exercise Solutions

HO-2.4.2(HO-0)
Exercise - Demonstrate how to integrate a memory leak detection tool in the DevOps pipeline

- Capture the memory leak detection by using Visual VM & MAT tool and show the results

HO-2.4.2(HO-0)
Exercise Solution - Demonstrate how to integrate a memory leak detection tool in the DevOps pipeline

- Pre-requisite → Java 8 , Eclipse IDE for java developers
- Get the code:
  - Open the browser and go to https://github.com/login
  - Login to your account
  - Launch URL https://github.com/umangsaltuniv/memory-leak-java
  - Click Fork at right top section
  - “memory-leak-java" repository will be added on your GitHub account
  - After forking “memory-leak-java” repository on GitHub Web, Click “Code” button
  - Click “Download ZIP”
  - Code will be downloaded on your local machine
  - Unzip the folder

HO-2.4.2(HO-0)
Setup Visual VM in Eclipse

- jvisualvm.exe is available in below location:
- C:\Program Files\Java\jdk1.8.0_191\bin
- Launch below url in browser
- https://visualvm.github.io/idesupport.html
- Click “plugin” link under Eclipse section to download it, unzip it(keep it in any location e.g. D:/Softwares/visualvm_launcher_u3_eclipse/
- Launch Eclipse > Go to Help > Click Install New Software... > Click Add > Enter Name(e.g. VsiualVM) and Enter URL:

  file:/D:/Softwares/visualvm_launcher_u3_eclipse/
- Select the Uncategorized check box > Next > Next > Accept T&C > Finish > Install anyway > Restart Eclipse
- Eclipse > Window > Preferences > expand Run/Debug > Expand Launching > VisualVMConfiguration:
- browser **jvisualvm.exe** under VisualVM Executable box
- JDK Home - C:\Program Files\Java\jdk1.8.0_191
- Click Apply & Close

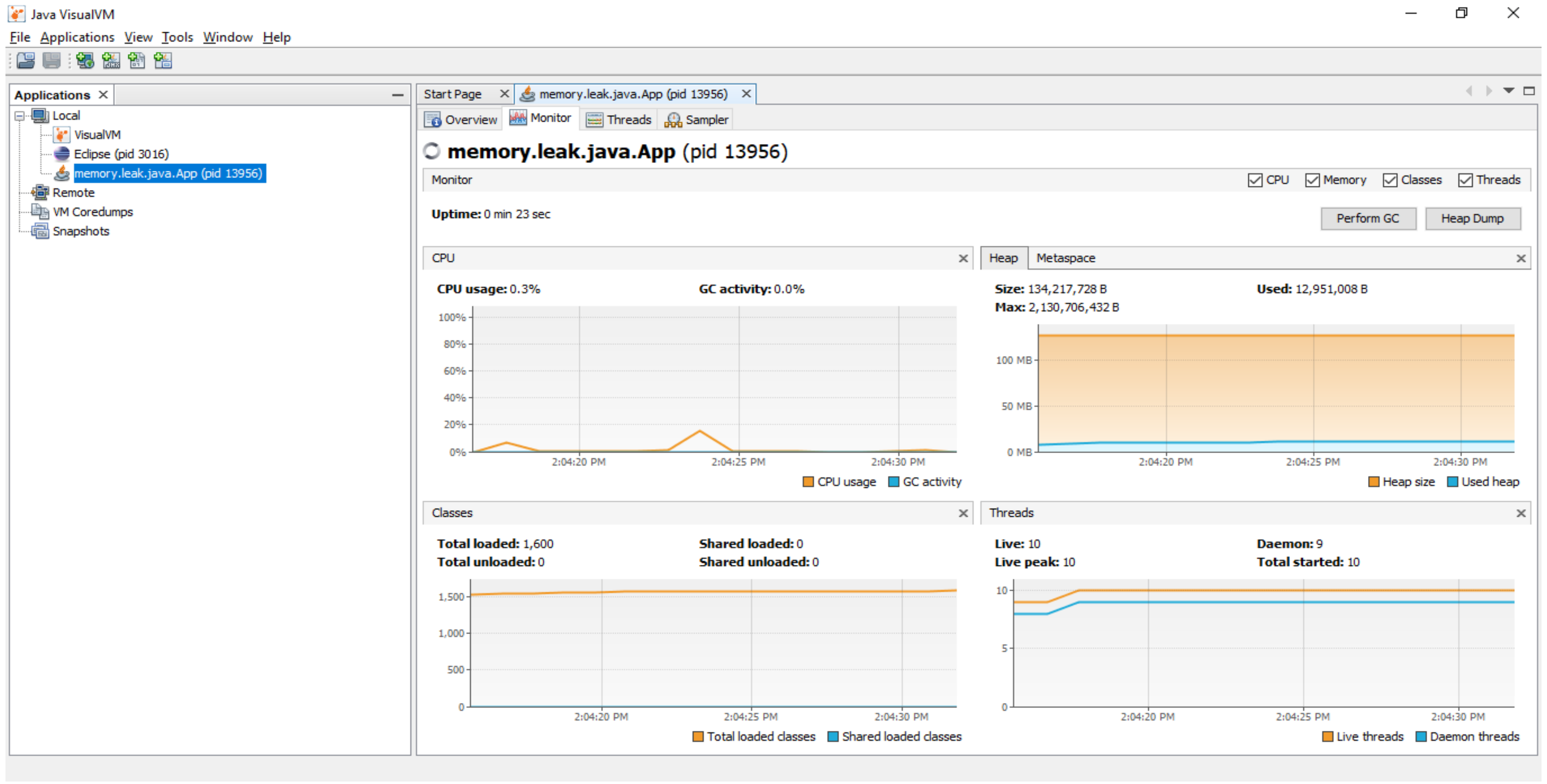
HO-2.4.2(HO-0)
Setup code in Eclipse

- Launch Eclipse
- Create java project
  - File > New > Java project > Enter the project name > Finish
- Create java package
  - Select project name > Right click > New > Package > Enter the package name(e.g.,emory.leak.java) > Finish
- Copy & paste the java classes(App.java & User.java) in package from local machine(where you have downloaded from GitHub)
- Add 2 breakpoints in App.java(e.g at system.out.println steps) by clicking at left hand side of the code line
- Run App.java as debug
- Select "Use configuration specific settings" checkbox > click VisualVM Launcher > OK > Click Debug

  Note: Above step's option will be displayed only first time
- Visual VM UI will be launched by running app code
- Note: If Visual VM does not launch then perform below steps:
- Click debug icon at top > Debug Configurations > select your java app(if app not visible then run the app as debug then follow these steps) > Go to main tab > click "Select other" link at bottom > Select "Use configuration specific settings" checkbox > click VisualVM Launcher > OK > Click Debug

HO-2.4.2(HO-0)
Visual VM Analysis

- Select your app > Click Monitor > See Heap usage graph



memory.leak.java.App (pid 13956)

Uptime: 0 min 23 sec

CPU usage: 0.3% GC activity: 0.0%

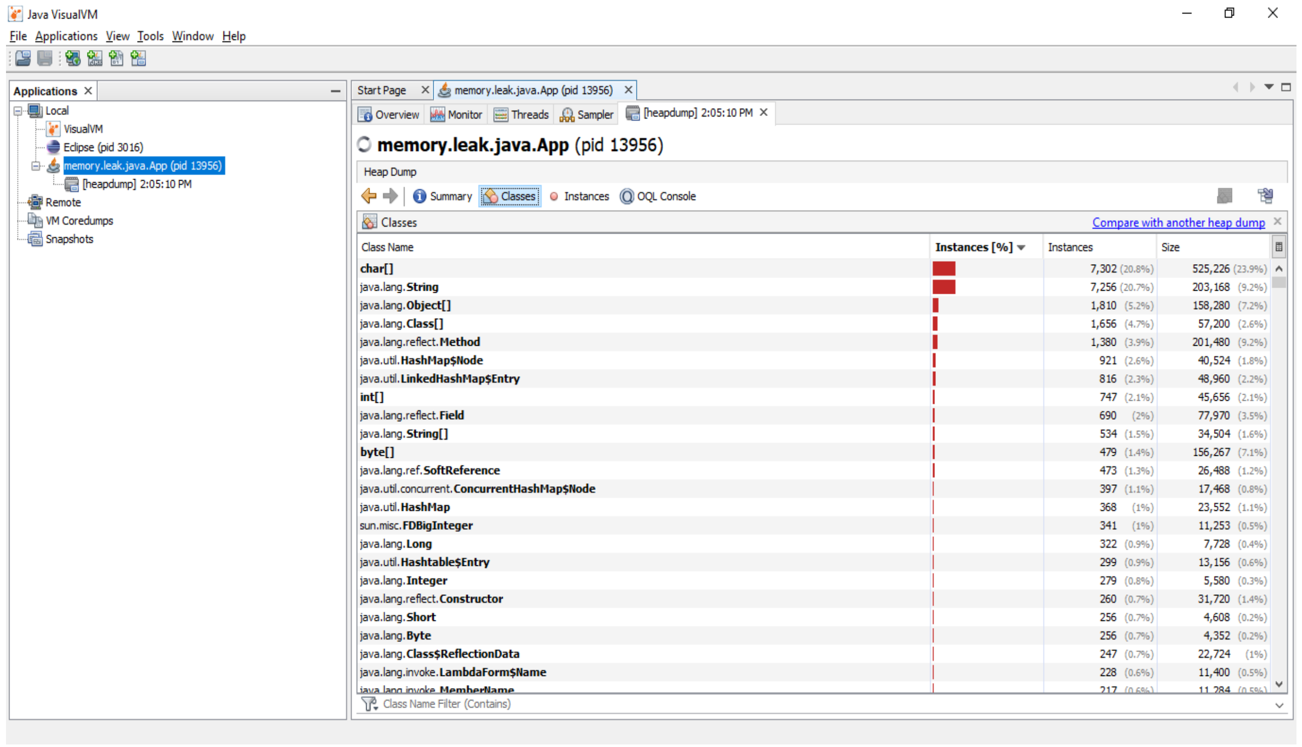
Size: 134,217,728 B Used: 12,951,008 B
Max: 2,130,706,432 B

Total loaded: 1,600 Shared loaded: 0
Total unloaded: 0 Shared unloaded: 0

Live: 10 Daemon: 9
Live peak: 10 Total started: 10

HO-2.4.2(HO-0)
Visual VM Analysis

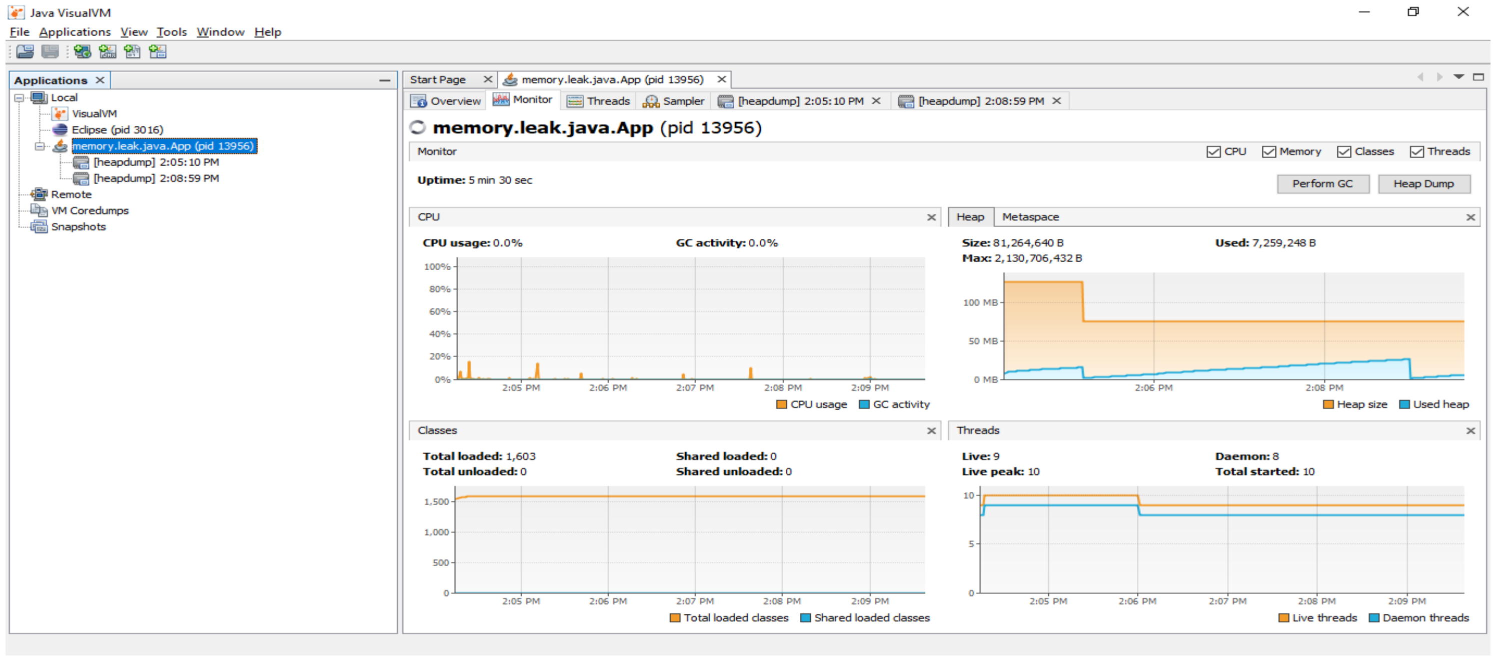
- Click Heap Dump > Heap Dump will be created at left > Open it > Go to Classes > See the memory leaks in red bar for other things



Class Name	Instances [%]	Instances	Size
char[]	🟥	7,302 (20.8%)	525,226 (23.9%)
java.lang.**String**	🟥	7,256 (20.7%)	203,168 (9.2%)
java.lang.**Object[]**	🟥	1,810 (5.2%)	158,280 (7.2%)
java.lang.**Class[]**	🟥	1,656 (4.7%)	57,200 (2.6%)
java.lang.reflect.**Method**	🟥	1,380 (3.9%)	201,480 (9.2%)
java.util.**HashMap$Node**	🟥	921 (2.6%)	40,524 (1.8%)
java.util.**LinkedHashMap$Entry**	🟥	816 (2.3%)	48,960 (2.2%)
int[]	🟥	747 (2.1%)	45,656 (2.1%)
java.lang.reflect.**Field**	🟥	690 (2%)	77,970 (3.5%)
java.lang.**String[]**	🟥	534 (1.5%)	34,504 (1.6%)
byte[]	🟥	479 (1.4%)	156,267 (7.1%)
java.lang.ref.**SoftReference**	🟥	473 (1.3%)	26,488 (1.2%)
java.util.concurrent.**ConcurrentHashMap$Node**	🟥	397 (1.1%)	17,468 (0.8%)
java.util.**HashMap**	🟥	368 (1%)	23,552 (1.1%)
sun.misc.**FDBigInteger**	🟥	341 (1%)	11,253 (0.5%)
java.lang.**Long**	🟥	322 (0.9%)	7,728 (0.4%)
java.util.**Hashtable$Entry**	🟥	299 (0.9%)	13,156 (0.6%)
java.lang.**Integer**	🟥	279 (0.8%)	5,580 (0.3%)
java.lang.reflect.**Constructor**	🟥	260 (0.7%)	31,720 (1.4%)
java.lang.**Short**	🟥	256 (0.7%)	4,608 (0.2%)
java.lang.**Byte**	🟥	256 (0.7%)	4,352 (0.2%)
java.lang.**Class$ReflectionData**	🟥	247 (0.7%)	22,724 (1%)
java.lang.invoke.**LambdaForm$Name**	🟥	228 (0.6%)	11,400 (0.5%)
java.lang.invoke.**MemberName**	🟥	217 (0.6%)	11,284 (0.5%)

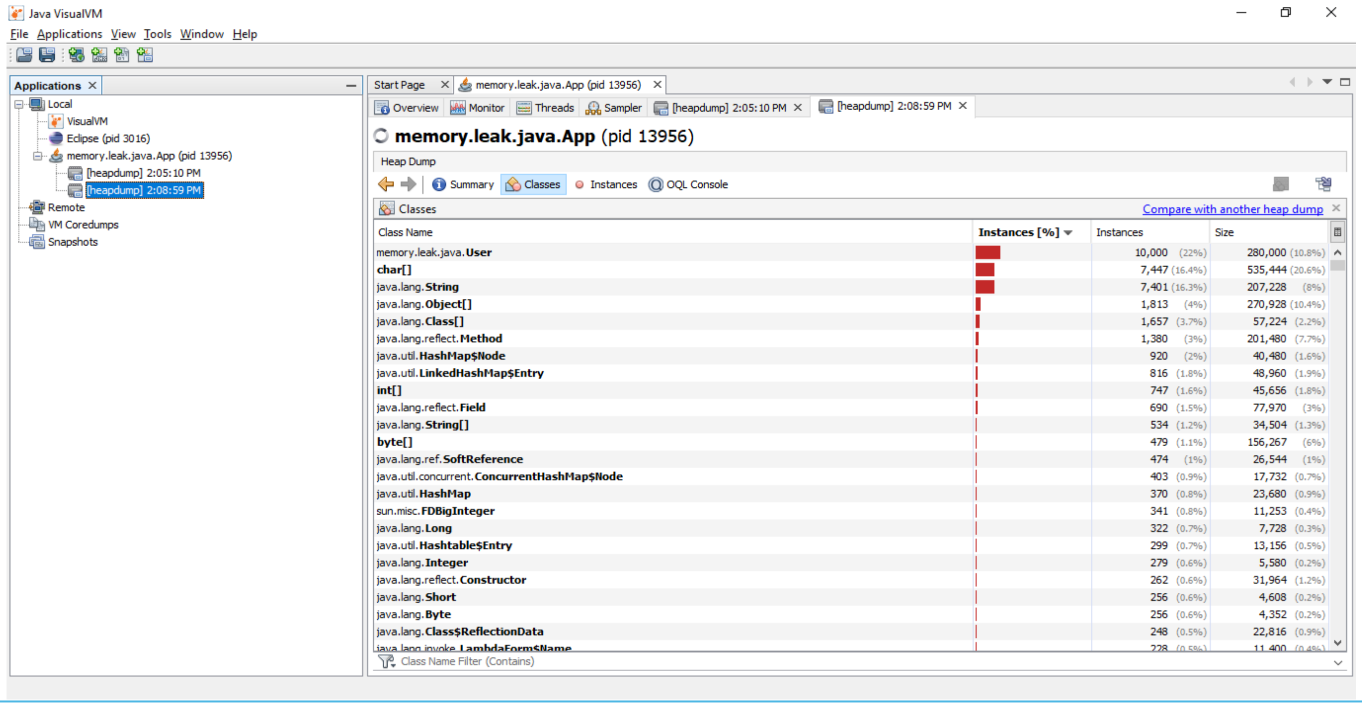
HO-2.4.2(HO-0)
Visual VM Analysis

- Now go to eclipse and click resume button(adjacent to red terminate button), code control will move to 2nd breakpoint
- Click Monitor > See Heap usage graph > heap usage will be increased



HO-2.4.2(HO-0)
Visual VM Analysis

- Click Heap Dump > Heap Dump will be created at left > Open it > Go to Classes > See the memory leaks in red bar for our app this time
- Save both Heap Dump files by right clicking on it



memory.leak.java.App (pid 13956)

Classes

Class Name	Instances [%]	Instances	Size
memory.leak.java.**User**		10,000 (22%)	280,000 (10.8%)
char[]		7,447 (16.4%)	535,444 (20.6%)
java.lang.**String**		7,401 (16.3%)	207,228 (8%)
java.lang.**Object[]**		1,813 (4%)	270,928 (10.4%)
java.lang.**Class[]**		1,657 (3.7%)	57,224 (2.2%)
java.lang.reflect.**Method**		1,380 (3%)	201,480 (7.7%)
java.util.**HashMap$Node**		920 (2%)	40,480 (1.6%)
java.util.**LinkedHashMap$Entry**		816 (1.8%)	48,960 (1.9%)
int[]		747 (1.6%)	45,656 (1.8%)
java.lang.reflect.**Field**		690 (1.5%)	77,970 (3%)
java.lang.**String[]**		534 (1.2%)	34,504 (1.3%)
byte[]		479 (1.1%)	156,267 (6%)
java.lang.ref.**SoftReference**		474 (1%)	26,544 (1%)
java.util.concurrent.**ConcurrentHashMap$Node**		403 (0.9%)	17,732 (0.7%)
java.util.**HashMap**		370 (0.8%)	23,680 (0.9%)
sun.misc.**FDBigInteger**		341 (0.8%)	11,253 (0.4%)
java.lang.**Long**		322 (0.7%)	7,728 (0.3%)
java.util.**Hashtable$Entry**		299 (0.7%)	13,156 (0.5%)
java.lang.**Integer**		279 (0.6%)	5,580 (0.2%)
java.lang.reflect.**Constructor**		262 (0.6%)	31,964 (1.2%)
java.lang.**Short**		256 (0.6%)	4,608 (0.2%)
java.lang.**Byte**		256 (0.6%)	4,352 (0.2%)
java.lang.**Class$ReflectionData**		248 (0.5%)	22,816 (0.9%)
java.lang.invoke.**LambdaForm$Name**		228 (0.5%)	11,400 (0.4%)

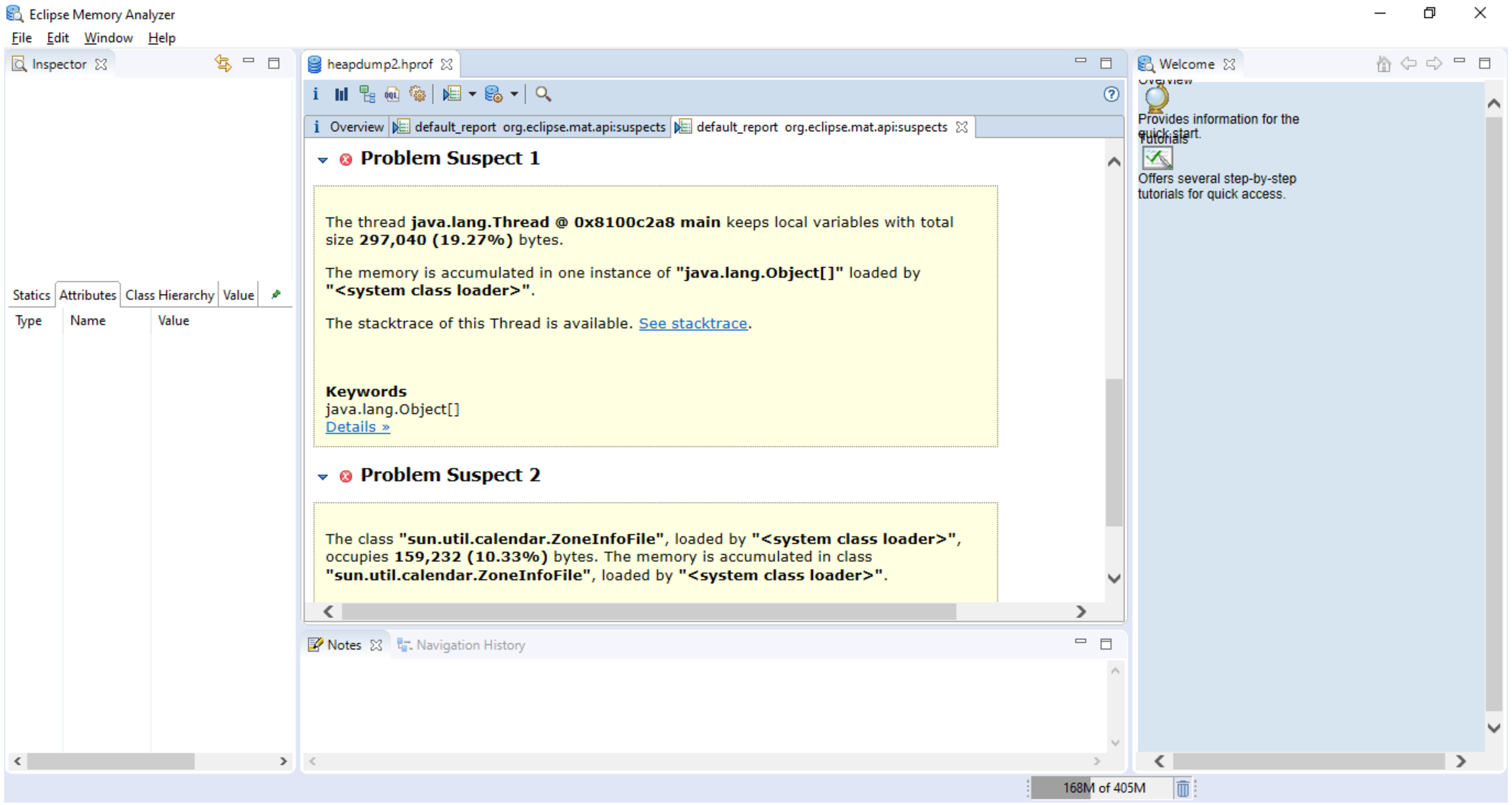
HO-2.4.2(HO-0)
Setup MAT tool

- Launch below link in browser
- https://www.eclipse.org/mat/downloads.php
- Click Windows (x86_64)
- Click Download
- Unzip the folder

HO-2.4.2(HO-0)
Evaluate memory leaks results with memory analyzer tool (MAT)

- Go inside mat folder > Launch MemoryAnalyzer.exe
- Click Open a Heap Dump
- Browser heap dump file2
- Select Leak Suspects Report option
- Finish
- Click Leak suspects link & evaluate the result

HO-2.4.2(HO-0)
Evaluate memory leaks results with memory analyzer tool (MAT)


Eclipse Memory Analyzer
File Edit Window Help
Inspector
heapdump2.hprof
Welcome
Overview
default_report org.eclipse.mat.api:suspects
default_report org.eclipse.mat.api:suspects
Statics
Attributes
Class Hierarchy
Value
Type
Name
Value
Provides information for the quick start.
Tutorials
Offers several step-by-step tutorials for quick access.
Notes
Navigation History
168M of 405M

Problem Suspect 1

The thread **java.lang.Thread @ 0x8100c2a8 main** keeps local variables with total size **297,040 (19.27%)** bytes.

The memory is accumulated in one instance of **"java.lang.Object[]"** loaded by **"<system class loader>"**.

The stacktrace of this Thread is available. See stacktrace.

Keywords
java.lang.Object[]
Details »

Problem Suspect 2

The class **"sun.util.calendar.ZoneInfoFile"**, loaded by **"<system class loader>"**, occupies **159,232 (10.33%)** bytes. The memory is accumulated in class **"sun.util.calendar.ZoneInfoFile"**, loaded by **"<system class loader>"**.