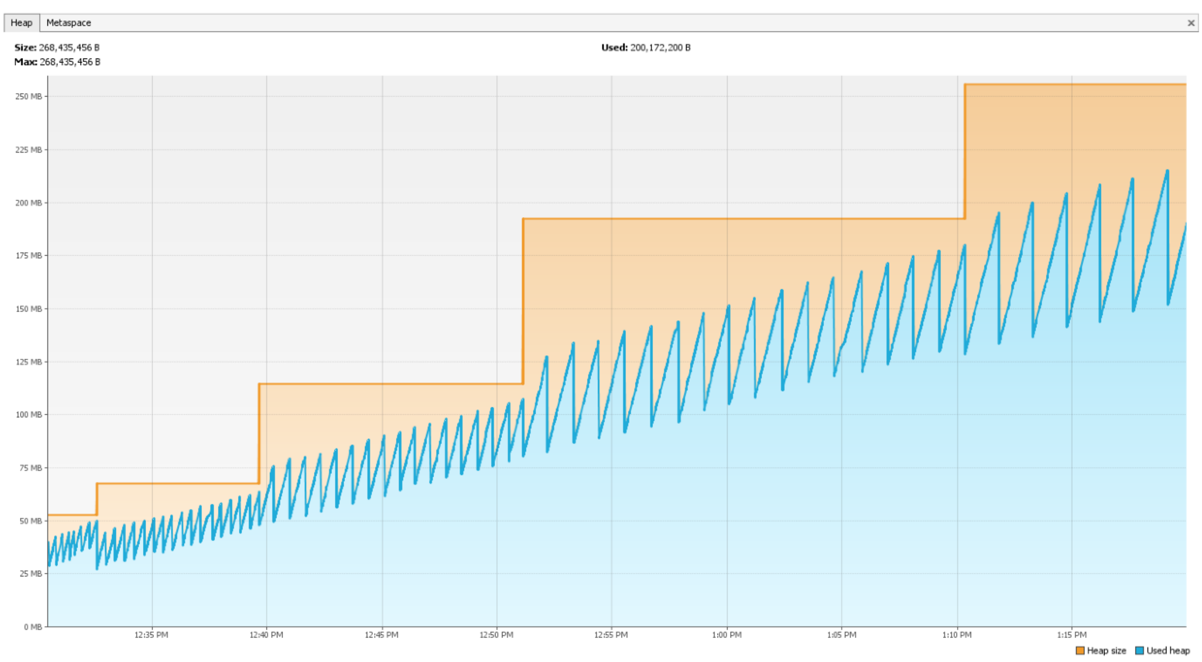
Simplify the Heap Dump Analysis with techniques used by Experts

OutofMemory issues are something many of us have faced in production. It's frustrating when the only fix seems to be restarting the Java process, knowing it's just a temporary solution. For many teams, this becomes a routine—restarting the application daily or weekly to avoid those dreaded errors. But the real question is: why does this happen, and how can we fix it for good?

It's not that we don't want to solve the problem permanently. Often, it's just a lack of understanding about how the JVM works, why OutofMemory errors are triggered, and what's really causing them.

That's why I want to break it down in a way that anyone can understand. Using the open-source tool **Eclipse MAT**, I'll show you the techniques that experts use to analyze heap dumps and get to the root cause of OutOfMemory issues.



In just a few simple steps, you can go from repeatedly restarting your app to permanently fixing these problems–just like the pros do.

When your Java application processes requests, computes data, or handles business logic, it stores data as objects in the Heap memory temporarily. Once a request is completed, those objects should no longer be needed, and that's where the Garbage Collector comes in. It's responsible for cleaning up these unused objects to free up space in the Heap.

However, if too many objects are created and not cleaned up (or if there's a memory leak), the Heap fills up. This is when the JVM can no longer allocate memory for new objects, triggering an **OutOfMemoryError.**

Alright, now let's get into the real fun—figuring out the reasons for OutOfMemoryError:

1. **Allocation of large objects**: When your application tries to allocate a single large object that exceeds the available heap space, you'll get an OutOfMemoryError.
2. **Memory leaks:** This happens when your application keeps allocating a high number of small objects but fails to clean them up properly, leaving them hanging in the Heap.

The first step is collecting a **Heap Dump.** You can do this using jcmd or any other tool, and in many cases, your application will generate a heap dump automatically when an OutOfMemoryError occurs. The heap dump file is essentially a snapshot of your application's memory at the time the error occurred, which we'll analyze to find the root cause.

I'll be using the **Eclipse MAT tool** for this analysis, which will help us find the issue in just a few steps. You can download the tool here: Eclipse MAT Tool.

Our goal with the heap dump analysis is simple:

1. Identify **which objects** are taking up most of the heap memory.

2. Figure out **which application packages** are responsible for the high number of object allocations.

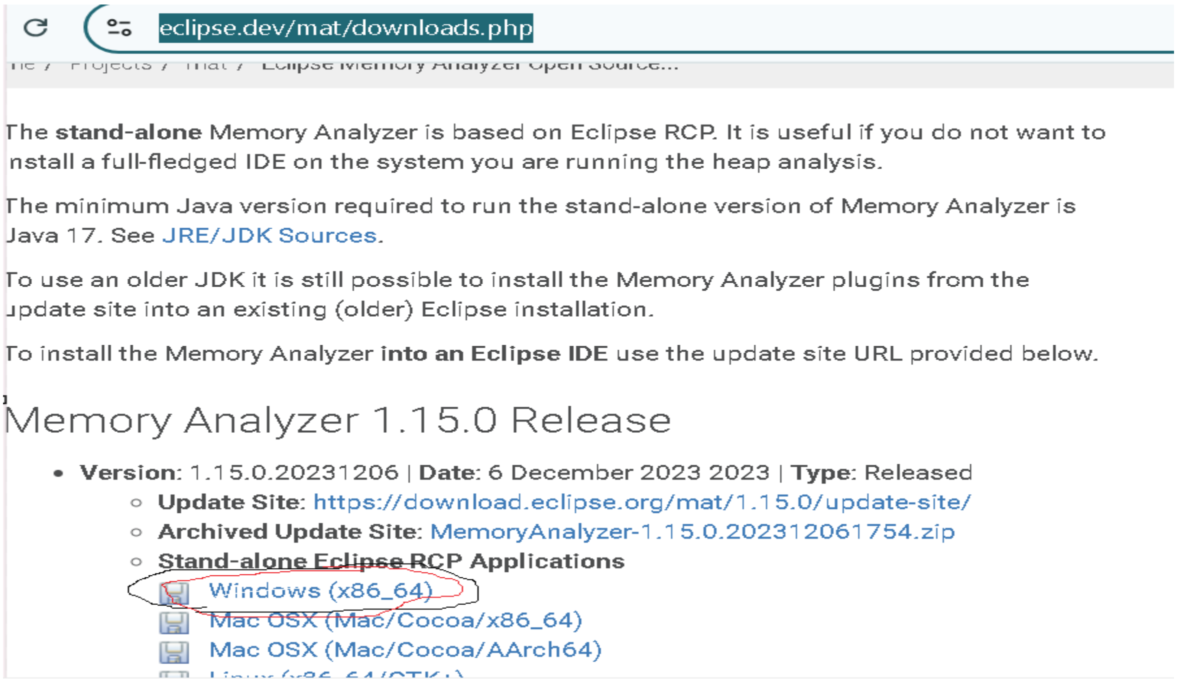
First thing first, collect the Heap Dump using **jcmd** or any tool , make sure you collect the heap dump, or your application generates the heap dump automatically when OutOfMemory issue occurs.

Heap dump file look like this

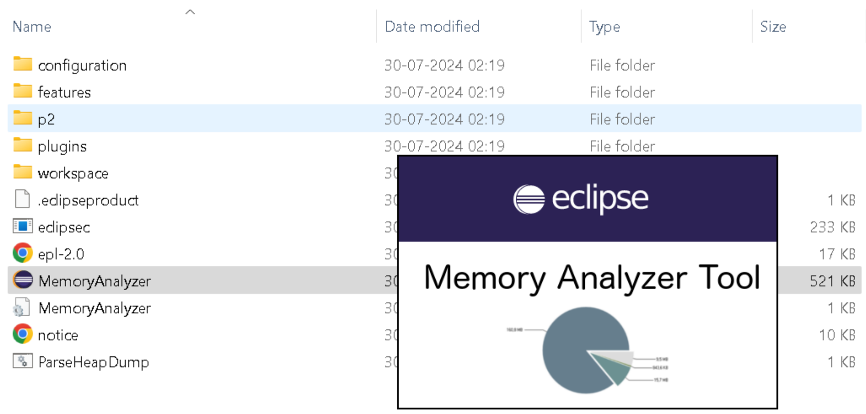
Name	Date modified	Type	Size
heapdump1.hprof	20-06-2024 03:52	HPROF File	6,27,477 KB

I am going to use Eclipse MAT tool to analyse this file and find the root cause in just few steps

Download Eclipse MAT tool : **https://eclipse.dev/mat/downloads.php**



Okay , now open the MAT tool



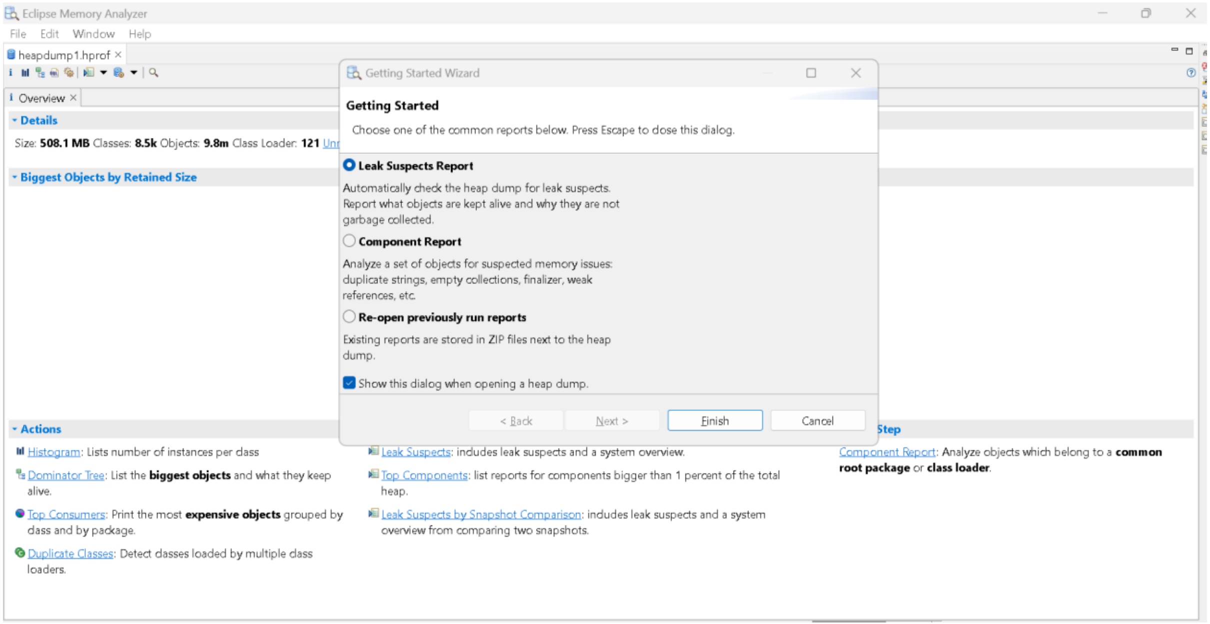
Import the heap dump: Go to File > Acquire heap dump.

You'll now see a lot of statistics and deep-dive object details about your heap. You will get a lot of statistics as it gathers deep-dive object details about your heap. This is where many people confuse , because it provides more deeper aspect of the objects and so they might not know what to check and how to find the root cause.

We'll take the **expert approach** to make sense of it.

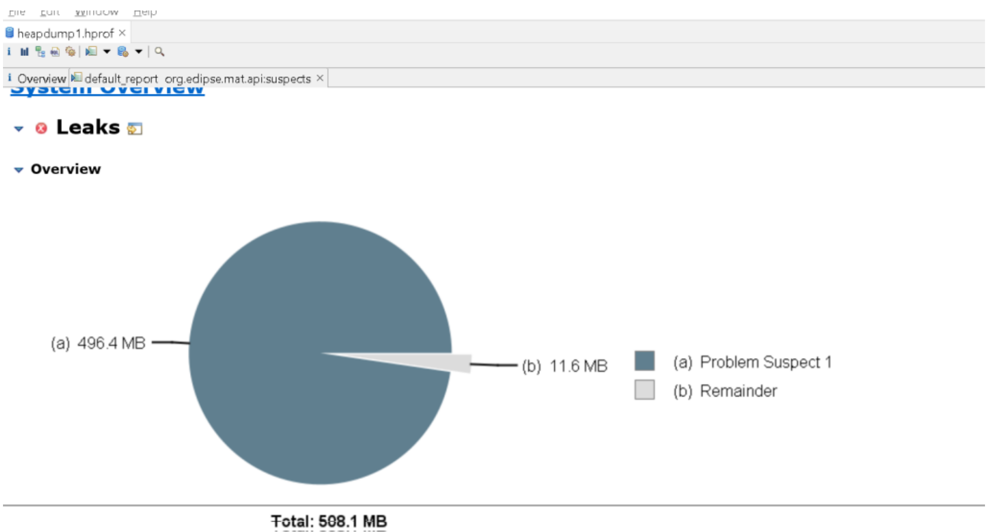
- Identify **which objects** are taking up most of the heap memory.
- Figure out **which application packages** are responsible for the high number of object allocations.

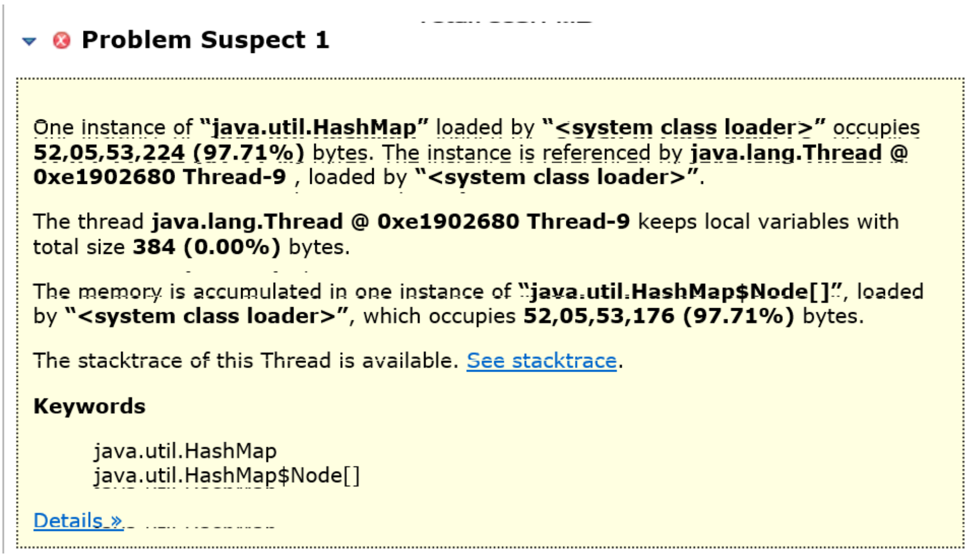
Keep this mind, this should be our goal always when you analyze the Heap Dump



Leak Suspects:

Let's check the Leak suspect report





Looking at the above report, it shows **java.util.hashmap** consumes 97% of heap memory. This is internal java classes code.

Don't get confused by looking at this, how could we fix this internal Java code ?

Well, here's the key: while we can't modify the internal workings of Java classes like HashMap, but we do have control over **how our application code interacts with them.**

Think of it this way: in almost every scenario, your **application code** is responsible for invoking these internal Java classes. It might be a **custom class** or method written by one of your developers, such as a collection of objects from your application. Typically, this application code belongs to your own package, which could follow a structure like com.appname.*.*.

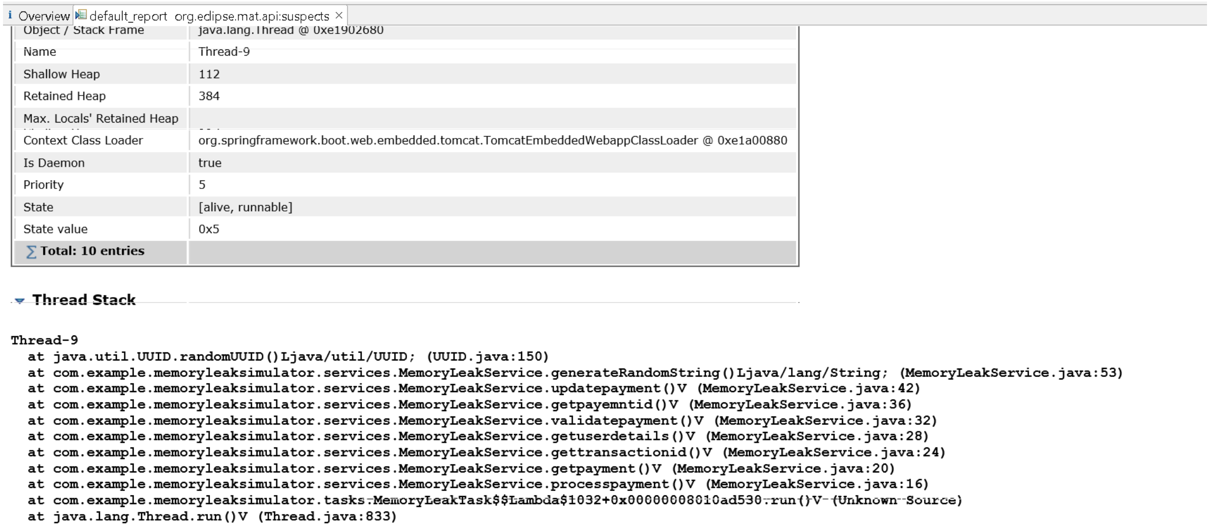
So, where do we focus our attention?

Instead of focusing on the java.util.hashmap which is just a symptom, we need to trace back to the **application-level code** that initiated the call to the HashMap. In essence, you need to determine **which part of your application code** is responsible for the heavy use of this HashMap.

Here's how you can approach it:

In same leak-suspect screen, expand the screen and find the application code invoked HashMap.

From Thread stack – I could see some application code but still its unclear whether this is problematic code caused the OutOfMemory issue.



Object / Stack Frame	java.lang.Thread @ 0xe1902680
Name	Thread-9
Shallow Heap	112
Retained Heap	384
Max. Locals' Retained Heap	
Context Class Loader	org.springframework.boot.web.embedded.tomcat.TomcatEmbeddedWebappClassLoader @ 0xe1a00880
Is Daemon	true
Priority	5
State	[alive, runnable]
State value	0x5
∑ Total: 10 entries	

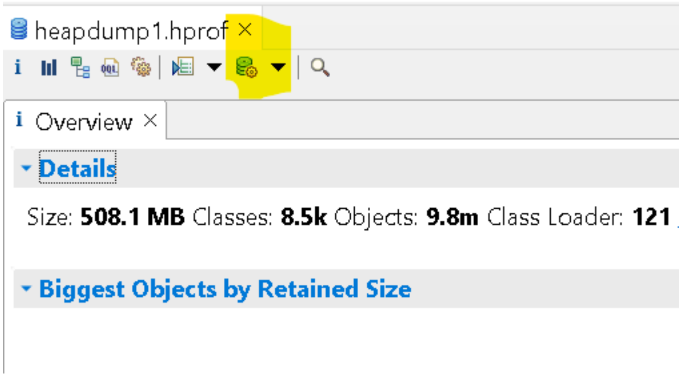
Thread Stack

```
Thread-9
  at java.util.UUID.randomUUID()Ljava/util/UUID; (UUID.java:150)
  at com.example.memoryleaksimulator.services.MemoryLeakService.generateRandomString()Ljava/lang/String; (MemoryLeakService.java:53)
  at com.example.memoryleaksimulator.services.MemoryLeakService.updatepayment()V (MemoryLeakService.java:42)
  at com.example.memoryleaksimulator.services.MemoryLeakService.getpayemntid()V (MemoryLeakService.java:36)
  at com.example.memoryleaksimulator.services.MemoryLeakService.validatepayment()V (MemoryLeakService.java:32)
  at com.example.memoryleaksimulator.services.MemoryLeakService.getuserdetails()V (MemoryLeakService.java:28)
  at com.example.memoryleaksimulator.services.MemoryLeakService.gettransactionid()V (MemoryLeakService.java:24)
  at com.example.memoryleaksimulator.services.MemoryLeakService.getpayment()V (MemoryLeakService.java:20)
  at com.example.memoryleaksimulator.services.MemoryLeakService.processpayment()V (MemoryLeakService.java:16)
  at com.example.memoryleaksimulator.tasks.MemoryLeakTask$$Lambda$1032+0x00000008010ad530.run()V (Unknown Source)
  at java.lang.Thread.run()V (Thread.java:833)
```

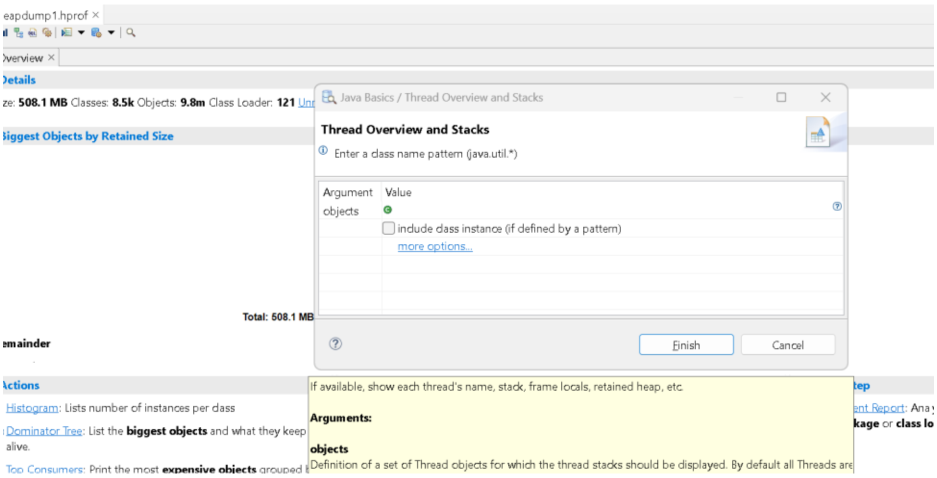
To Make it simple, I am going to show you 2 quick ways to find the root cause.

1. Thread Overview and stack trace
2. Dominator Tree

1.Thread Overview and stack trace



Click Java Basics -→ Thread overview and stacks and Click Finish



3.You can see all thread stack trace here, now you can expand each thread and find the highest object allocated Thread from the below list.

Overview | thread_overview

Object / Stack Frame	Name	Shallow Heap	Retained Heap	Max. Locals' Retained Heap	Context Class Loader	Is Daemc
<Regex>	<Regex>	<Numeric>	<Numeric>	<Numeric>	<Regex>	<Regex>
java.lang.Thread @ 0xe19693e0	RMI TCP Connection(3)-192.168.0.100	112	18,736		jdk.internal.loader.ClassLoaders$AppClassLoader @ 0xe1549...	true
java.lang.Thread @ 0xe1a4d820	RMI TCP Connection(4)-192.168.0.100	112	18,672			true
java.lang.Thread @ 0xe1548f18	RMI TCP Connection(6)-192.168.0.100	112	18,600		jdk.internal.loader.ClassLoaders$AppClassLoader @ 0xe1549...	true
org.apache.tomcat.util.threads.TaskThread @ 0xe159eac0	http-nio-8081-exec-2	120	1,160		org.springframework.boot.loader.launch.LaunchedClassLoad...	true
java.lang.Thread @ 0xf47eae20	Thread-81	112	1,112		org.springframework.boot.web.embedded.tomcat.TomcatE...	true
java.lang.Thread @ 0xf48fa418	Thread-72	112	1,112		org.springframework.boot.web.embedded.tomcat.TomcatE...	true
org.apache.tomcat.util.threads.TaskThread @ 0xe11ffd50	http-nio-8081-exec-5	120	1,000		org.springframework.boot.loader.launch.LaunchedClassLoad...	true
org.apache.tomcat.util.threads.TaskThread @ 0xe1547f30	http-nio-8081-exec-1	120	1,000		org.springframework.boot.loader.launch.LaunchedClassLoad...	true
org.apache.tomcat.util.threads.TaskThread @ 0xe15482a8	http-nio-8081-exec-7	120	1,000		org.springframework.boot.loader.launch.LaunchedClassLoad...	true
org.apache.tomcat.util.threads.TaskThread @ 0xe154846(	http-nio-8081-exec-9	120	1,000		org.springframework.boot.loader.launch.LaunchedClassLoad...	true
org.apache.tomcat.util.threads.TaskThread @ 0xe159ed9(	http-nio-8081-exec-6	120	1,000		org.springframework.boot.loader.launch.LaunchedClassLoad...	true
org.apache.tomcat.util.threads.TaskThread @ 0xe159ef48	http-nio-8081-exec-8	120	1,000		org.springframework.boot.loader.launch.LaunchedClassLoad...	true
org.apache.tomcat.util.threads.TaskThread @ 0xe159f100	http-nio-8081-exec-10	120	1,000		org.springframework.boot.loader.launch.LaunchedClassLoad...	true
org.apache.tomcat.util.threads.TaskThread @ 0xe193dbe(	http-nio-8081-exec-4	120	1,000		org.springframework.boot.loader.launch.LaunchedClassLoad...	true
org.apache.tomcat.util.threads.TaskThread @ 0xe196922{	http-nio-8081-exec-3	120	1,000		org.springframework.boot.loader.launch.LaunchedClassLoad...	true
java.lang.Thread @ 0xe1a88a90	Thread-18	112	616		org.springframework.boot.web.embedded.tomcat.TomcatE...	true
java.lang.Thread @ 0xe1548908	RMI TCP Accept-0	112	568		jdk.internal.loader.ClassLoaders$AppClassLoader @ 0xe1549...	true
java.lang.Thread @ 0xe1a88910	Thread-14	112	544		org.springframework.boot.web.embedded.tomcat.TomcatE...	true
java.lang.Thread @ 0xec3776b0	Thread-48	112	544		org.springframework.boot.web.embedded.tomcat.TomcatE...	true
java.lang.Thread @ 0xfca08d48	Thread-42	112	544		org.springframework.boot.web.embedded.tomcat.TomcatE...	true
java.lang.Thread @ 0xffeef650	Thread-105	112	544		org.springframework.boot.web.embedded.tomcat.TomcatE...	true
java.lang.Thread @ 0xe196bc00	Thread-13	112	424		org.springframework.boot.web.embedded.tomcat.TomcatE...	true
java.lang.Thread @ 0xe2affa78	Thread-78	112	424		org.springframework.boot.web.embedded.tomcat.TomcatE...	true
java.lang.Thread @ 0xe31ffa38	Thread-74	112	424		org.springframework.boot.web.embedded.tomcat.TomcatE...	true
java.lang.Thread @ 0xe695bb18	Thread-36	112	424		org.springframework.boot.web.embedded.tomcat.TomcatE...	true
java.lang.Thread @ 0xe69f9330	Thread-39	112	424		org.springframework.boot.web.embedded.tomcat.TomcatE...	true

As you can see here, I was able to find the thread "Thread-18" is responsible for OOM error. The problematic application code is com.example.memoryleak.generateRandomString was storing 525MB of objects due to which OutOfMemory error has occurred.(refer highlighted in yellow)

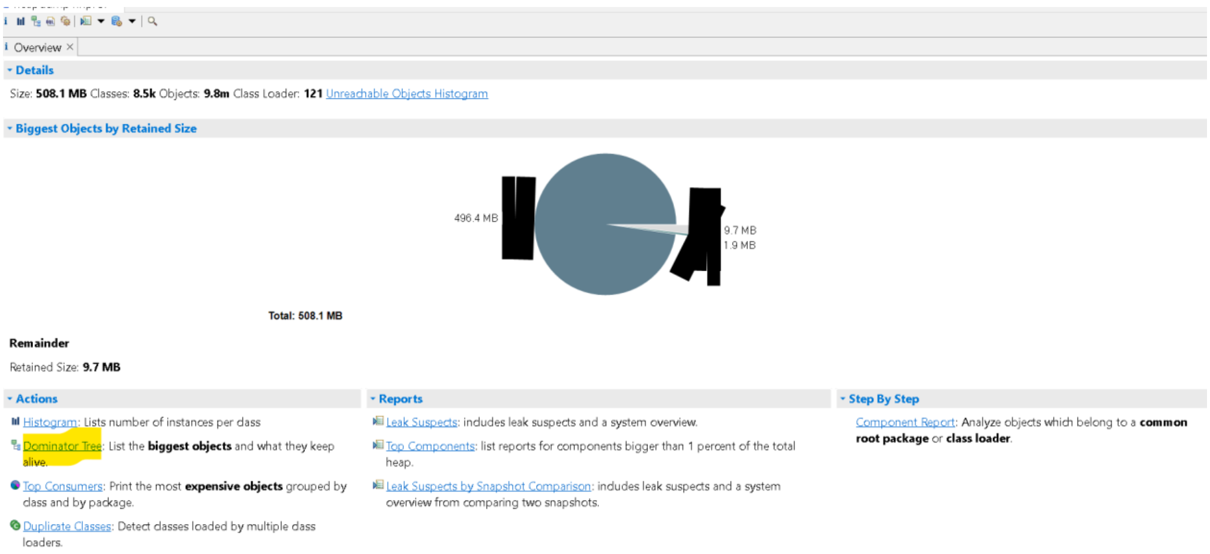
Overview | thread_overview

Object / Stack Frame	Name	Shallow Heap	Retained Heap	Max. Locals' Retained Heap	Context Class Lo
org.apache.tomcat.util.threads.TaskThread @ 0xe15482a8	http-nio-8081-exec-7	120	1,000		org.springframe
org.apache.tomcat.util.threads.TaskThread @ 0xe1548460	http-nio-8081-exec-9	120	1,000		org.springframe
org.apache.tomcat.util.threads.TaskThread @ 0xe159ed90	http-nio-8081-exec-6	120	1,000		org.springframe
org.apache.tomcat.util.threads.TaskThread @ 0xe159ef48	http-nio-8081-exec-8	120	1,000		org.springframe
org.apache.tomcat.util.threads.TaskThread @ 0xe159f100	http-nio-8081-exec-10	120	1,000		org.springframe
org.apache.tomcat.util.threads.TaskThread @ 0xe193dbe0	http-nio-8081-exec-4	120	1,000		org.springframe
org.apache.tomcat.util.threads.TaskThread @ 0xe1969228	http-nio-8081-exec-3	120	1,000		org.springframe
java.lang.Thread @ 0xe1a88a90	Thread-18	112	616		org.springframe
at java.lang.OutOfMemoryError.<init>()V (OutOfMemoryError.java:48)					
at sun.security.provider.HashDrbg.generateAlgorithm([B[B)V (HashDrbg.java:234)				736	
at sun.security.provider.AbstractDrbg.engineNextBytes([BLjava/security/SecureRandomParameters;)V (Abstract					
at sun.security.provider.AbstractDrbg.engineNextBytes([B)V (AbstractDrbg.java:334)					
at sun.security.provider.DRBG.engineNextBytes([B)V (DRBG.java:235)					
at java.security.SecureRandom.nextBytes([B)V (SecureRandom.java:758)					
at java.util.UUID.randomUUID()Ljava/util/UUID; (UUID.java:151)				32	
at com.example.memoryleaksimulator.services.MemoryLeakService.generateRandomString()Ljava/lang/String;					
at com.example.memoryleaksimulator.services.MemoryLeakService.updatepayment()V (MemoryLeakService.jav				52,05,53,224	
at com.example.memoryleaksimulator.services.MemoryLeakService.getpayemntid()V (MemoryLeakService.java:				16	
at com.example.memoryleaksimulator.services.MemoryLeakService.validatepayment()V (MemoryLeakService.ja				16	
at com.example.memoryleaksimulator.services.MemoryLeakService.getuserdetails()V (MemoryLeakService.java:				16	
at com.example.memoryleaksimulator.services.MemoryLeakService.gettransactionid()V (MemoryLeakService.jav				16	
at com.example.memoryleaksimulator.services.MemoryLeakService.getpayment()V (MemoryLeakService.java:2(				16	
at com.example.memoryleaksimulator.services.MemoryLeakService.processpayment()V (MemoryLeakService.ja				16	
at com.example.memoryleaksimulator.tasks.MemoryLeakTask$$Lambda$1032+0x00000008010ad530.run()V (				16	
at java.lang.Thread.run()V (Thread.java:833)				616	
Total: 17 entries					
java.lang.Thread @ 0xe1548908	RMI TCP Accept-0	112	568		jdk.internal.loade

Great! We've successfully identified the root cause of the **OutOfMemoryError**. Now we can share these findings with the developer to address and resolve the issue. Happy ending indeed! 🎉

2.Dominator Tree

Let's explore a second approach to finding the root cause of the **OutOfMemoryError** in a different application. This time, we'll utilize the **Dominator Tree** to trace and identify the underlying issue. By analyzing the dominant objects in memory, we can pinpoint the areas where the most memory is being consumed, helping us find the root cause of the memory leak.



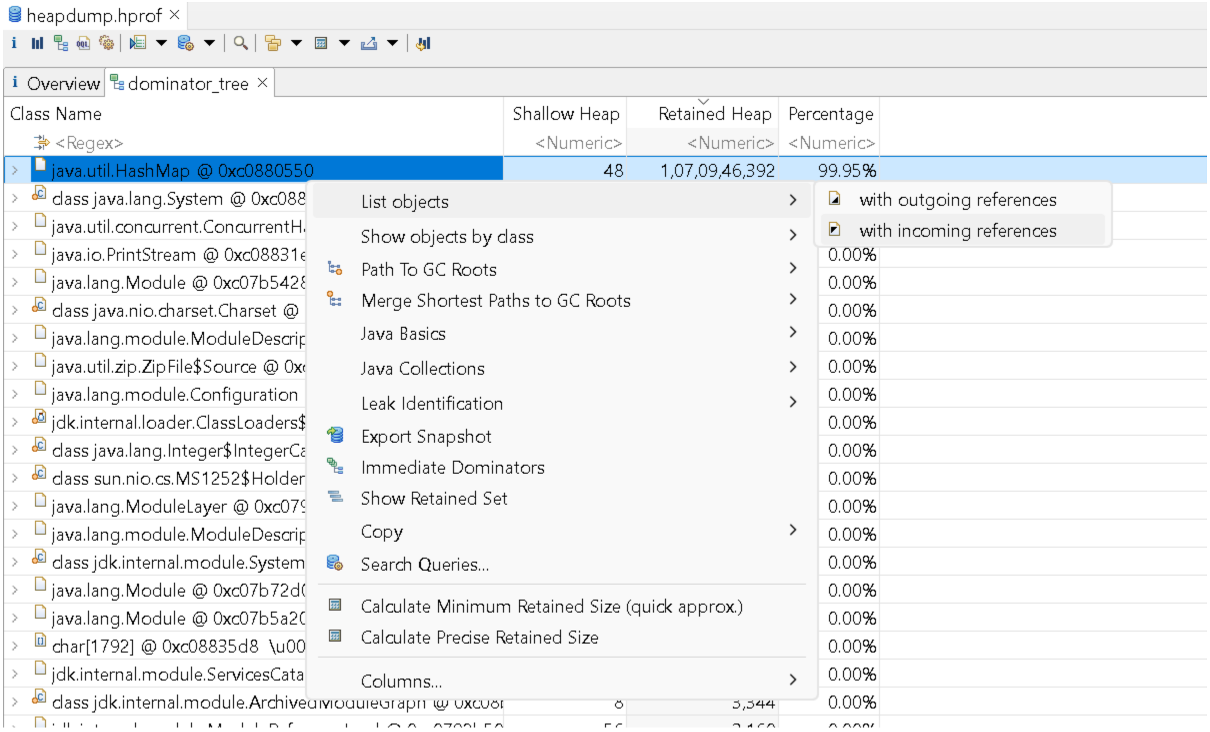
In the **Dominator Tree**, you can observe that the **java.util.HashMap** object is holding around 99% of the objects. However, this alone doesn't indicate which part of the application code is responsible for allocating this HashMap. To identify the source, we need to dive deeper into the code and analyze further.

Overview | dominator_tree

Class Name	Shallow Heap	Retained Heap	Percentage
<Regex>	<Numeric>	<Numeric>	<Numeric>
java.util.HashMap @ 0xc0880550	48	1,07,09,46,392	99.95%
class java.lang.System @ 0xc0880d28 **System Class**	56	35,944	0.00%
java.util.concurrent.ConcurrentHashMap @ 0xc08985e8	64	34,064	0.00%
java.io.PrintStream @ 0xc08831e0	40	25,112	0.00%
java.lang.Module @ 0xc07b5428	48	18,320	0.00%
class java.nio.charset.Charset @ 0xc08c4438 **System Clas**	32	14,320	0.00%
java.lang.module.ModuleDescriptor @ 0xc0792b88	64	7,944	0.00%
java.util.zip.ZipFile$Source @ 0xc03f88f0	80	7,632	0.00%
java.lang.module.Configuration @ 0xc0791928	40	6,800	0.00%
jdk.internal.loader.ClassLoaders$AppClassLoader @ 0xc0:	96	5,800	0.00%
class java.lang.Integer$IntegerCache @ 0xc08c6528 **Syste**	24	5,168	0.00%
class sun.nio.cs.MS1252$Holder @ 0xc08b5298 **System (**	24	4,744	0.00%
java.lang.ModuleLayer @ 0xc0791900	40	4,352	0.00%
java.lang.module.ModuleDescriptor @ 0xc07a2658	64	4,256	0.00%
class jdk.internal.module.SystemModules$default @ 0xc0	8	4,128	0.00%
java.lang.Module @ 0xc07b72d0	48	3,904	0.00%
java.lang.Module @ 0xc07b5a20	48	3,776	0.00%
char[1792] @ 0xc08835d8 \u0000\u0001\u0002\u0003\	3,600	3,600	0.00%
jdk.internal.module.ServicesCatalog @ 0xc07c0528	16	3,456	0.00%
class jdk.internal.module.ArchivedModuleGraph @ 0xc08l	8	3,344	0.00%
jdk.internal.module.ModuleReferenceImpl @ 0xc0792b50	56	3,160	0.00%
java.lang.OutOfMemoryError[4] @ 0xc03f70a0 **JNI Globa**	32	2,912	0.00%
java.lang.Module @ 0xc07b5df0	48	2,800	0.00%
class jdk.internal.loader.NativeLibraries$LibraryPaths @ 0x	8	2,720	0.00%
class jdk.internal.misc.VM @ 0xc08c5b60 **System Class**	96	2,672	0.00%

Right click → List Objects -→ with Incoming references

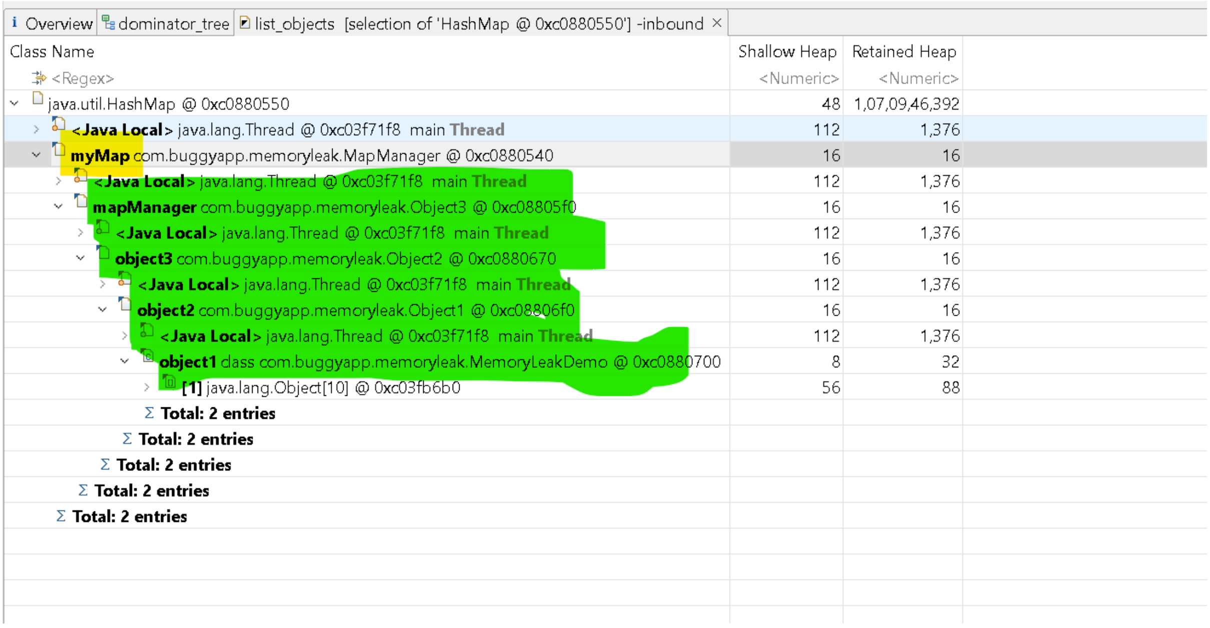
Using the incoming references , we can find who called the java.util.hashmap internal code.



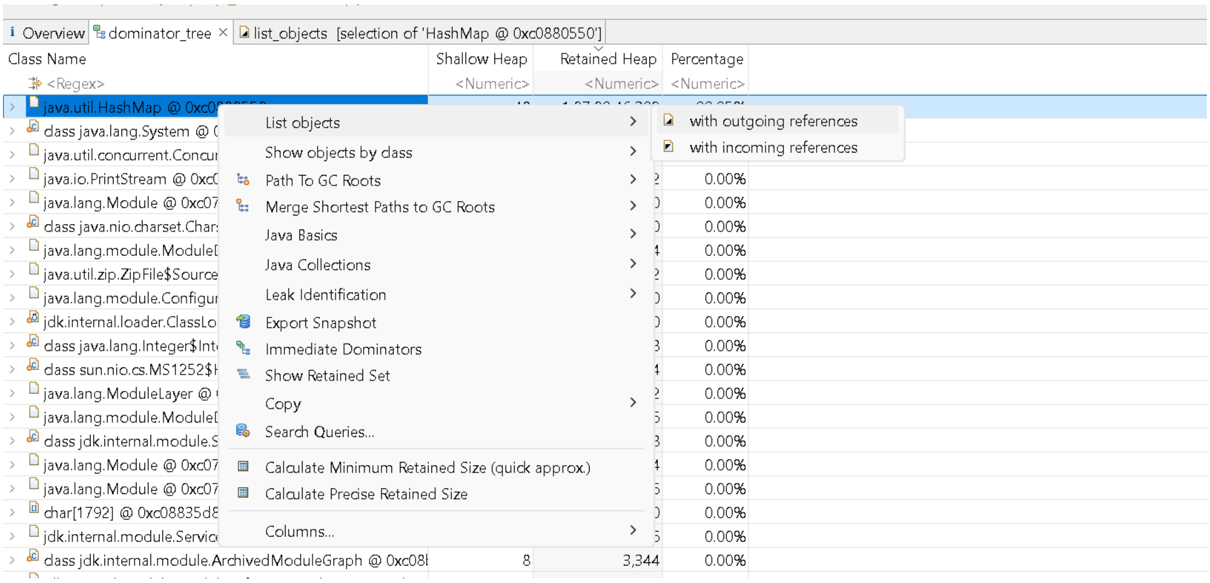
In the **Incoming References** screen, we need to keep expanding the classes until we locate the application package code. As shown below, I was able to find the application package code `**com.buggyapp.memoryleak.MapManager**`, where the object in question is named `myMap`.

When reviewing the flow from bottom to top, the com.buggyapp.memoryleak.MemoryLeakDemo class calls the Object1 method, which then calls Object2, followed by Object3, and eventually reaches the MapManager method. This is where the myMap object is created, which is holding around 1GB of data.

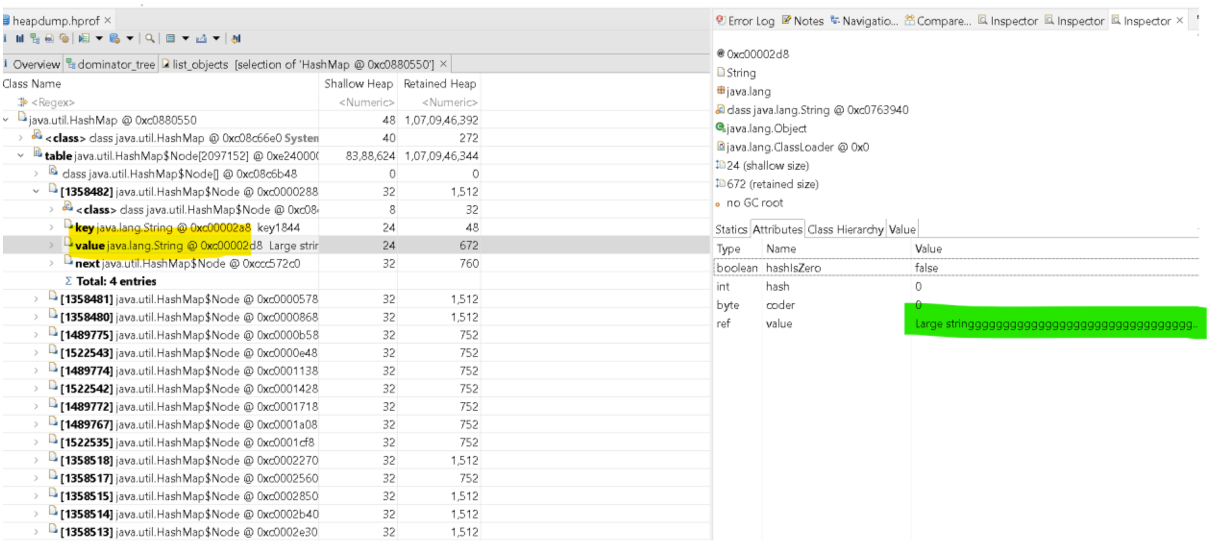
The issue originates in the MapManager class during the creation of this object, and we need to inform the developer to address and fix the problem.



If you're interested in seeing what the 1GB object contains, we can also inspect it using the **Outgoing Reference** option.



Continue expanding the tree and clicking on the highlighted values (marked in yellow). In the inspector panel on the right-hand side, you'll notice that 1GB of memory is being consumed by an object storing data labeled as "Large Stringgggggggggggggggggggggggggggg."



That's it .. Hope this helps to simplify the Heap dump analysis.