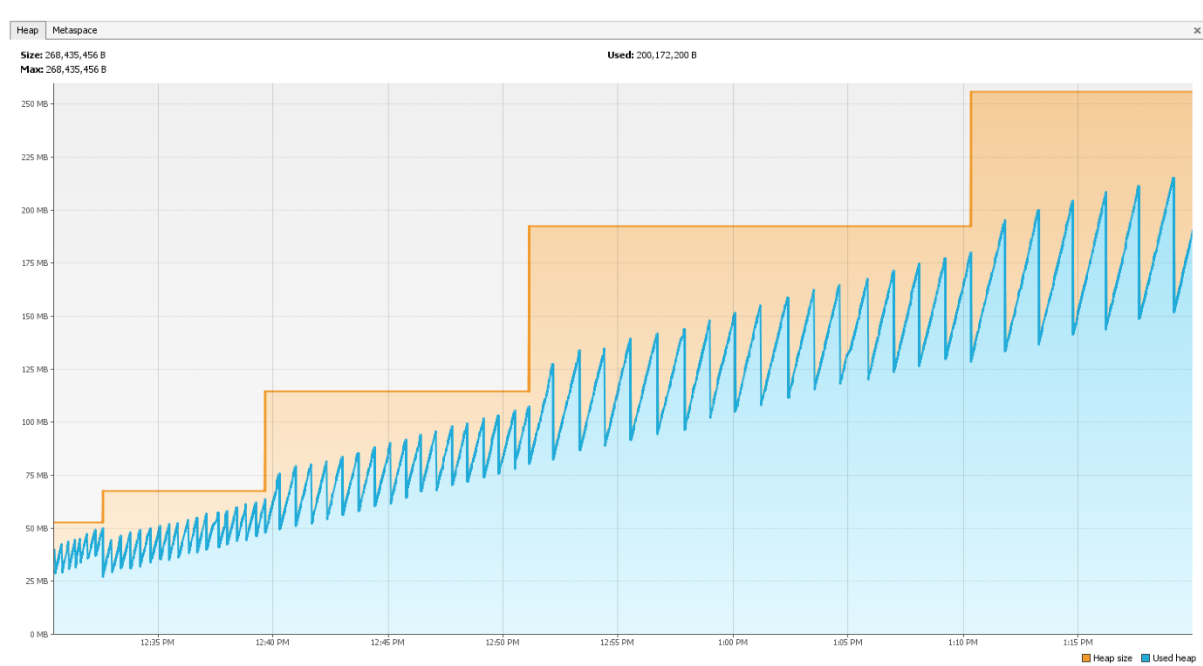


Simplify the Heap Dump Analysis with techniques used by Experts

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>

eclipse.dev/mat/downloads.php

The **stand-alone** Memory Analyzer is based on Eclipse RCP. It is useful if you do not want to install a full-fledged IDE on the system you are running the heap analysis.

The minimum Java version required to run the stand-alone version of Memory Analyzer is Java 17. See [JRE/JDK Sources](#).

To use an older JDK it is still possible to install the Memory Analyzer plugins from the update site into an existing (older) Eclipse installation.

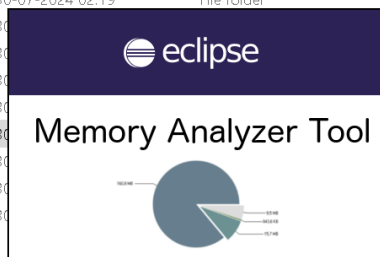
To install the Memory Analyzer **into an Eclipse IDE** use the update site URL provided below.

Memory Analyzer 1.15.0 Release

- **Version:** 1.15.0.20231206 | **Date:** 6 December 2023 | **Type:** Released
 - **Update Site:** <https://download.eclipse.org/mat/1.15.0/update-site/>
 - **Archived Update Site:** [MemoryAnalyzer-1.15.0.202312061754.zip](#)
 - **Stand-alone Eclipse RCP Applications**
 - Windows (x86_64)
 - Mac OSX (Mac/Cocoa/x86_64)
 - Mac OSX (Mac/Cocoa/AArch64)
 - Linux (x86_64/GTK+)

Okay , now open the MAT tool

Name	Date modified	Type	Size
configuration	30-07-2024 02:19	File folder	
features	30-07-2024 02:19	File folder	
p2	30-07-2024 02:19	File folder	
plugins	30-07-2024 02:19	File folder	
workspace	30-07-2024 02:19	File folder	
.edipseproduct	30-07-2024 02:19	File	1 KB
edipsec	30-07-2024 02:19	File	233 KB
epl-2.0	30-07-2024 02:19	File	17 KB
MemoryAnalyzer	30-07-2024 02:19	File	521 KB
MemoryAnalyzer	30-07-2024 02:19	File	1 KB
notice	30-07-2024 02:19	File	10 KB
ParseHeapDump	30-07-2024 02:19	File	1 KB



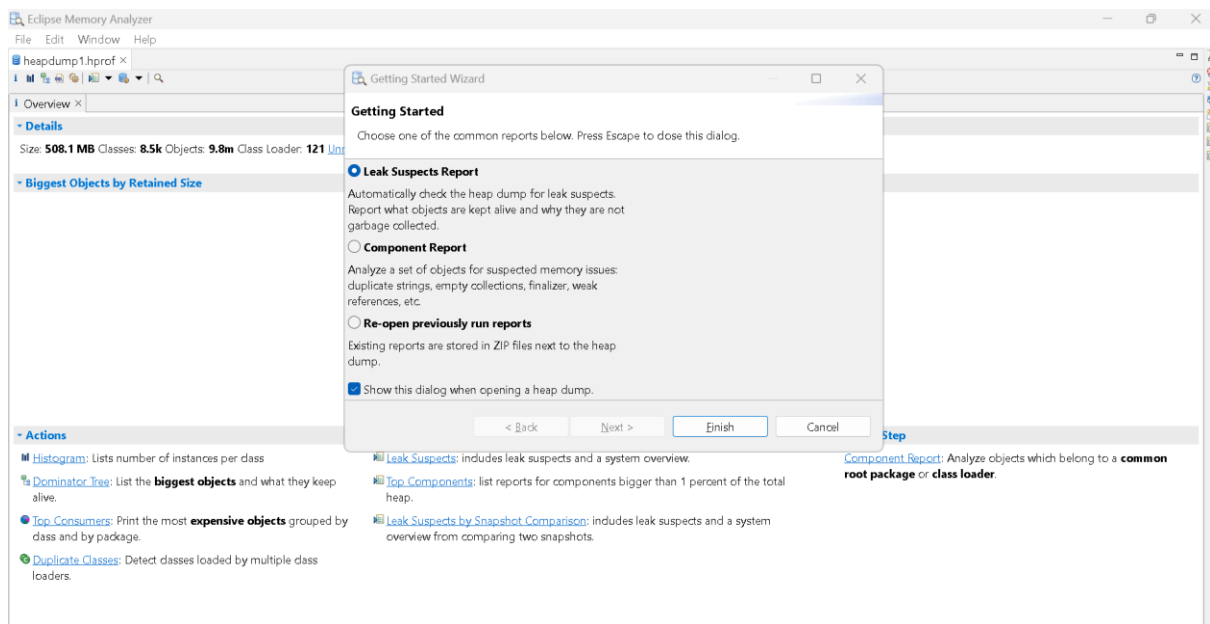
Import the heap dump: Go to File > Acquire heap dump.

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.

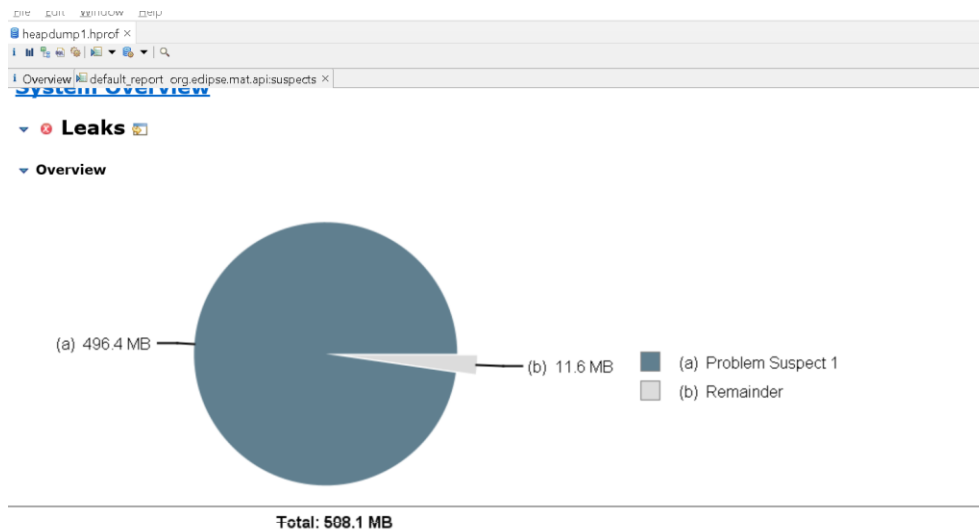
- Identify **which objects** are taking up most of the heap memory.
- Figure out **which application packages** are responsible for the high number of object allocations.

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



Leak Suspects:

Let's check the Leak suspect report



▼ Problem Suspect 1

One instance of "**java.util.HashMap**" loaded by "**<system class loader>**" occupies **52,05,53,224 (97.71%)** bytes. The instance is referenced by **java.lang.Thread @ 0xe1902680 Thread-9**, loaded by "**<system class loader>**".

The thread **java.lang.Thread @ 0xe1902680 Thread-9** keeps local variables with total size **384 (0.00%)** bytes.

The memory is accumulated in one instance of "**java.util.HashMap\$Node[]**", loaded by "**<system class loader>**", which occupies **52,05,53,176 (97.71%)** bytes.

The stacktrace of this Thread is available. [See stacktrace.](#)

Keywords

java.util.HashMap
java.util.HashMap\$Node[]

[Details >](#)

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.

Overview	default_report org.eclipse.mat.api:suspects x
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.getpaymentid()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+0x00000000000010ad530.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

heapdump1.hprof

Overview

Details

Size: **508.1 MB** Classes: **8.5k** Objects: **9.8m** Class Loader: **121**

Biggest Objects by Retained Size

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

Thread Overview and Stacks

Enter a class name pattern (java.util.*)

Argument	Value
objects	...

☐ include class instance (if defined by a pattern)

[more options...](#)

Total: 508.1 MB

Finish Cancel

Actions

Histogram: Lists number of instances per class

Dominator Tree: List the biggest objects and what they keep alive.

Top Consumers: Print the most expensive objects around

If available, show each thread's name, stack, frame locals, retained heap, etc.

Arguments:

objects

Definition of a set of Thread objects for which the thread stacks should be displayed. By default all Threads are

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.

Object / Stack Frame	Name	Shallow Heap	Retained Heap	Max Locals	Retained Heap	Context Class Loader	Is Daemon
<Regex>	<Regex>	<Numeric>	<Numeric>		<Numeric>	<Regex>	<Regex>
java.lang.Thread @ 0xe19593e0	RMI TCP Connection(3)-192.168.0.100	112	18,736			jdk.internal.loader.ClassLoaders\$AppClassLoader @ 0xe1549...	true
java.lang.Thread @ 0xe154d820	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 @ 0xe159ead0	http-nio-8081-exec-2	120	1,160			org.springframework.boot.loader.launch.LaunchedClassLoad...	true
java.lang.Thread @ 0xf47ee20	Thread-81	112	1,112			org.springframework.boot.web.embedded.tomcat.TomcatE...	true
java.lang.Thread @ 0xf48fa18	Thread-72	112	1,112			org.springframework.boot.web.embedded.tomcat.TomcatE...	true
org.apache.tomcat.util.threads.TaskThread @ 0xe11ff050	http-nio-8081-exec-5	120	1,000			org.springframework.boot.loader.launch.LaunchedClassLoad...	true
org.apache.tomcat.util.threads.TaskThread @ 0xe1547f90	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 @ 0xe154846c	http-nio-8081-exec-9	120	1,000			org.springframework.boot.loader.launch.LaunchedClassLoad...	true
org.apache.tomcat.util.threads.TaskThread @ 0xe159ed9c	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 @ 0xe193dbe0	http-nio-8081-exec-4	120	1,000			org.springframework.boot.loader.launch.LaunchedClassLoad...	true
org.apache.tomcat.util.threads.TaskThread @ 0xe1969228	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 @ 0xfceef650	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

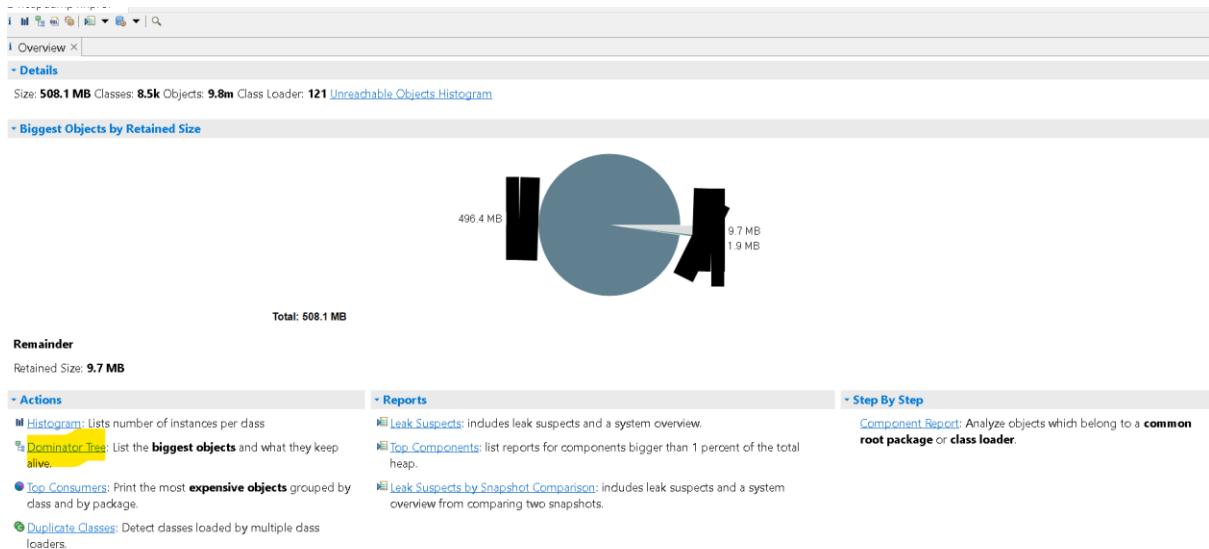
As you can see here, I was able to find the thread “Thread-18” is responsible for OOM error. The problematic application code is `com.example.memoryleak.generateRandomString` was storing 525MB of objects due to which `OutOfMemory` error has occurred. (refer highlighted in yellow)

Object / Stack Frame	Name	Shallow Heap	Retained Heap	Max Locals	Retained Heap	Context Class Loader
org.apache.tomcat.util.threads.TaskThread @ 0xe15482a8	http-nio-8081-exec-7	120	1,000			org.springframework
org.apache.tomcat.util.threads.TaskThread @ 0xe1548460	http-nio-8081-exec-9	120	1,000			org.springframework
org.apache.tomcat.util.threads.TaskThread @ 0xe159ed90	http-nio-8081-exec-6	120	1,000			org.springframework
org.apache.tomcat.util.threads.TaskThread @ 0xe159ef48	http-nio-8081-exec-8	120	1,000			org.springframework
org.apache.tomcat.util.threads.TaskThread @ 0xe159f100	http-nio-8081-exec-10	120	1,000			org.springframework
org.apache.tomcat.util.threads.TaskThread @ 0xe193dbe0	http-nio-8081-exec-4	120	1,000			org.springframework
org.apache.tomcat.util.threads.TaskThread @ 0xe1969228	http-nio-8081-exec-3	120	1,000			org.springframework
java.lang.Thread @ 0xe1a88a90	Thread-18	112	616			org.springframework
at java.lang.OutOfMemoryError.<init>(V (OutOfMemoryError.java:48)						
at sun.security.provider.HashDrbg.generateAlgorithm(BB)V (HashDrbg.java:234)					736	
at sun.security.provider.AbstractDrbg.engineNextBytes(B)V (AbstractDrbg.java:234)						
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(V) (UUID.java:151)					32	
at com.example.memoryleak.simulator.services.MemoryLeakService.generateRandomString(V) (MemoryLeakService.java:21)					52,055,324	
at com.example.memoryleak.simulator.services.MemoryLeakService.updatePayment(V) (MemoryLeakService.java:21)					16	
at com.example.memoryleak.simulator.services.MemoryLeakService.getPaymentId(V) (MemoryLeakService.java:21)					16	
at com.example.memoryleak.simulator.services.MemoryLeakService.validatePayment(V) (MemoryLeakService.java:21)					16	
at com.example.memoryleak.simulator.services.MemoryLeakService.getUserDetails(V) (MemoryLeakService.java:21)					16	
at com.example.memoryleak.simulator.services.MemoryLeakService.getTransactionId(V) (MemoryLeakService.java:21)					16	
at com.example.memoryleak.simulator.services.MemoryLeakService.getPayment(V) (MemoryLeakService.java:21)					16	
at com.example.memoryleak.simulator.services.MemoryLeakService.processPayment(V) (MemoryLeakService.java:21)					16	
at com.example.memoryleak.simulator.tasks.MemoryLeakTask.lambda\$run\$0(V) (MemoryLeakTask.java:33)					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.loader

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.

i Overview dominator_tree x				
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 Class	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 @ 0xc03f70a0	96	5,800	0.00%
>	class java.lang.Integer\$IntegerCache @ 0xc08c6528 System Class	24	5,168	0.00%
>	class sun.nio.cs.MS1252\$Holder @ 0xc08b5298 System Class	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 @ 0xc07b72d0	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\u0004\u0005\u0006\u0007\u0008\u0009\u000a\u000b\u000c\u000d\u000e\u000f\u0010\u0011\u0012\u0013\u0014\u0015\u0016\u0017\u0018\u0019\u001a\u001b\u001c\u001d\u001e\u001f\u0020\u0021\u0022\u0023\u0024\u0025\u0026\u0027\u0028\u0029\u002a\u002b\u002c\u002d\u002e\u002f\u0030\u0031\u0032\u0033\u0034\u0035\u0036\u0037\u0038\u0039\u003a\u003b\u003c\u003d\u003e\u003f\u0040\u0041\u0042\u0043\u0044\u0045\u0046\u0047\u0048\u0049\u004a\u004b\u004c\u004d\u004e\u004f\u0050\u0051\u0052\u0053\u0054\u0055\u0056\u0057\u0058\u0059\u005a\u005b\u005c\u005d\u005e\u005f\u0060\u0061\u0062\u0063\u0064\u0065\u0066\u0067\u0068\u0069\u006a\u006b\u006c\u006d\u006e\u006f\u0070\u0071\u0072\u0073\u0074\u0075\u0076\u0077\u0078\u0079\u007a\u007b\u007c\u007d\u007e\u007f\u0080\u0081\u0082\u0083\u0084\u0085\u0086\u0087\u0088\u0089\u008a\u008b\u008c\u008d\u008e\u008f\u0090\u0091\u0092\u0093\u0094\u0095\u0096\u0097\u0098\u0099\u009a\u009b\u009c\u009d\u009e\u009f\u00a0\u00a1\u00a2\u00a3\u00a4\u00a5\u00a6\u00a7\u00a8\u00a9\u00aa\u00ab\u00ac\u00ad\u00ae\u00af\u00b0\u00b1\u00b2\u00b3\u00b4\u00b5\u00b6\u00b7\u00b8\u00b9\u00ba\u00bb\u00bc\u00bd\u00be\u00bf\u00c0\u00c1\u00c2\u00c3\u00c4\u00c5\u00c6\u00c7\u00c8\u00c9\u00ca\u00cb\u00cc\u00cd\u00ce\u00cf\u00d0\u00d1\u00d2\u00d3\u00d4\u00d5\u00d6\u00d7\u00d8\u00d9\u00da\u00db\u00dc\u00dd\u00de\u00df\u00e0\u00e1\u00e2\u00e3\u00e4\u00e5\u00e6\u00e7\u00e8\u00e9\u00ea\u00eb\u00ec\u00ed\u00ie\u00ef\u00f0\u00f1\u00f2\u00f3\u00f4\u00f5\u00f6\u00f7\u00f8\u00f9\u00fa\u00fb\u00fc\u00fd\u00fe\u00ff\u0100\u0101\u0102\u0103\u0104\u0105\u0106\u0107\u0108\u0109\u010a\u010b\u010c\u010d\u010e\u010f\u0110\u0111\u0112\u0113\u0114\u0115\u0116\u0117\u0118\u0119\u011a\u011b\u011c\u011d\u011e\u011f\u0120\u0121\u0122\u0123\u0124\u0125\u0126\u0127\u0128\u0129\u012a\u012b\u012c\u012d\u012e\u012f\u0130\u0131\u0132\u0133\u0134\u0135\u0136\u0137\u0138\u0139\u013a\u013b\u013c\u013d\u013e\u013f\u0140\u0141\u0142\u0143\u0144\u0145\u0146\u0147\u0148\u0149\u014a\u014b\u014c\u014d\u014e\u014f\u0150\u0151\u0152\u0153\u0154\u0155\u0156\u0157\u0158\u0159\u015a\u015b\u015c\u015d\u015e\u015f\u0160\u0161\u0162\u0163\u0164\u0165\u0166\u0167\u0168\u0169\u016a\u016b\u016c\u016d\u016e\u016f\u0170\u0171\u0172\u0173\u0174\u0175\u0176\u0177\u0178\u0179\u017a\u017b\u017c\u017d\u017e\u017f\u0180\u0181\u0182\u0183\u0184\u0185\u0186\u0187\u0188\u0189\u018a\u018b\u018c\u018d\u018e\u018f\u0190\u0191\u0192\u0193\u0194\u0195\u0196\u0197\u0198\u0199\u019a\u019b\u019c\u019d\u019e\u019f\u01a0\u01a1\u01a2\u01a3\u01a4\u01a5\u01a6\u01a7\u01a8\u01a9\u01aa\u01ab\u01ac\u01ad\u01ae\u01af\u01b0\u01b1\u01b2\u01b3\u01b4\u01b5\u01b6\u01b7\u01b8\u01b9\u01ba\u01bb\u01bc\u01bd\u01be\u01bf\u01c0\u01c1\u01c2\u01c3\u01c4\u01c5\u01c6\u01c7\u01c8\u01c9\u01ca\u01cb\u01cc\u01cd\u01ce\u01cf\u01d0\u01d1\u01d2\u01d3\u01d4\u01d5\u01d6\u01d7\u01d8\u01d9\u01da\u01db\u01dc\u01dd\u01de\u01df\u01e0\u01e1\u01e2\u01e3\u01e4\u01e5\u01e6\u01e7\u01e8\u01e9\u01ea\u01eb\u01ec\u01ed\u01ee\u01ef\u01f0\u01f1\u01f2\u01f3\u01f4\u01f5\u01f6\u01f7\u01f8\u01f9\u01fa\u01fb\u01fc\u01fd\u01fe\u01ff\u0200\u0201\u0202\u0203\u0204\u0205\u0206\u0207\u0208\u0209\u020a\u020b\u020c\u020d\u020e\u020f\u0210\u0211\u0212\u0213\u0214\u0215\u0216\u0217\u0218\u0219\u021a\u021b\u021c\u021d\u021e\u021f\u0220\u0221\u0222\u0223\u0224\u0225\u0226\u0227\u0228\u0229\u022a\u022b\u022c\u022d\u022e\u022f\u0230\u0231\u0232\u0233\u0234\u0235\u0236\u0237\u0238\u0239\u023a\u023b\u023c\u023d\u023e\u023f\u0240\u0241\u0242\u0243\u0244\u0245\u0246\u0247\u0248\u0249\u024a\u024b\u024c\u024d\u024e\u024f\u0250\u0251\u0252\u0253\u0254\u0255\u0256\u0257\u0258\u0259\u025a\u025b\u025c\u025d\u025e\u025f\u0260\u0261\u0262\u0263\u0264\u0265\u0266\u0267\u0268\u0269\u026a\u026b\u026c\u026d\u026e\u026f\u0270\u0271\u0272\u0273\u0274\u0275\u0276\u0277\u0278\u0279\u027a\u027b\u027c\u027d\u027e\u027f\u0280\u0281\u0282\u0283\u0284\u0285\u0286\u0287\u0288\u0289\u028a\u028b\u028c\u028d\u028e\u028f\u0290\u0291\u0292\u0293\u0294\u0295\u0296\u0297\u0298\u0299\u029a\u029b\u029c\u029d\u029e\u029f\u02a0\u02a1\u02a2\u02a3\u02a4\u02a5\u02a6\u02a7\u02a8\u02a9\u02aa\u02ab\u02ac\u02ad\u02ae\u02af\u02b0\u02b1\u02b2\u02b3\u02b4\u02b5\u02b6\u02b7\u02b8\u02b9\u02ba\u02bb\u02bc\u02bd\u02be\u02bf\u02c0\u02c1\u02c2\u02c3\u02c4\u02c5\u02c6\u02c7\u02c8\u02c9\u02ca\u02cb\u02cc\u02cd\u02ce\u02cf\u02d0\u02d1\u02d2\u02d3\u02d4\u02d5\u02d6\u02d7\u02d8\u02d9\u02da\u02db\u02dc\u02dd\u02de\u02df\u02e0\u02e1\u02e2\u02e3\u02e4\u02e5\u02e6\u02e7\u02e8\u02e9\u02ea\u02eb\u02ec\u02ed\u02ee\u02ef\u02f0\u02f1\u02f2\u02f3\u02f4\u02f5\u02f6\u02f7\u02f8\u02f9\u02fa\u02fb\u02fc\u02fd\u02fe\u02ff\u0280\u0281\u0282\u0283\u0284\u0285\u0286\u0287\u0288\u0289\u028a\u028b\u028c\u028d\u028e\u028f\u0290\u0291\u0292\u0293\u0294\u0295\u0296\u0297\u0298\u0299\u029a\u029b\u029c\u029d\u029e\u029f\u02a0\u02a1\u02a2\u02a3\u02a4\u02a5\u02a6\u02a7\u02a8\u02a9\u02aa\u02ab\u02ac\u02ad\u02ae\u02af\u02b0\u02b1\u02b2\u02b3\u02b4\u02b5\u02b6\u02b7\u02b8\u02b9\u02ba\u02bb\u02bc\u02bd\u02be\u02bf\u02c0\u02c1\u02c2\u02c3\u02c4\u02c5\u02c6\u02c7\u02c8\u02c9\u02ca\u02cb\u02cc\u02cd\u02ce\u02cf\u02d0\u02d1\u02d2\u02d3\u02d4\u02d5\u02d6\u02d7\u02d8\u02d9\u02da\u02db\u02dc\u02dd\u02de\u02df\u02e0\u02e1\u02e2\u02e3\u02e4\u02e5\u02e6\u02e7\u02e8\u02e9\u02ea\u02eb\u02ec\u02ed\u02ee\u02ef\u02f0\u02f1\u02f2\u02f3\u02f4\u02f5\u02f6\u02f7\u02f8\u02f9\u02fa\u02fb\u02fc\u02fd\u02fe\u02ff\u0280\u0281\u0282\u0283\u0284\u0285\u0286\u0287\u0288\u0289\u028a\u028b\u028c\u028d\u028e\u028f\u0290\u0291\u0292\u0293\u0294\u0295\u0296\u0297\u0298\u0299\u029a\u029b\u029c\u029d\u029e\u029f\u02a0\u02a1\u02a2\u02a3\u02a4\u02a5\u02a6\u02a7\u02a8\u02a9\u02aa\u02ab\u02ac\u02ad\u02ae\u02af\u02b0\u02b1\u02b2\u02b3\u02b4\u02b5\u02b6\u02b7\u02b8\u02b9\u02ba\u02bb\u02bc\u02bd\u02be\u02bf\u02c0\u02c1\u02c2\u02c3\u02c4\u02c5\u02c6\u02c7\u02c8\u02c9\u02ca\u02cb\u02cc\u02cd\u02ce\u02cf\u02d0\u02d1\u02d2\u02d3\u02d4\u02d5\u02d6\u02d7\u02d8\u02d9\u02da\u02db\u02dc\u02dd\u02de\u02df\u02e0\u02e1\u02e2\u02e3\u02e4\u02e5\u02e6\u02e7\u02e8\u02e9\u02ea\u02eb\u02ec\u02ed\u02ee\u02ef\u02f0\u02f1\u02f2\u02f3\u02f4\u02f5\u02f6\u02f7\u02f8\u02f9\u02fa\u02fb\u02fc\u02fd\u02fe\u02ff\u0280\u0281\u0282\u0283\u0284\u0285\u0286\u0287\u0288\u0289\u028a\u028b\u028c\u028d\u028e\u028f\u0290\u0291\u0292\u0293\u0294\u0295\u0296\u0297\u0298\u0299\u029a\u029b\u029c\u029d\u029e\u029f\u02a0\u02a1\u02a2\u02a3\u02a4\u02a5\u02a6\u02a7\u02a8\u02a9\u02aa\u02ab\u02ac\u02ad\u02ae\u02af\u02b0\u02b1\u02b2\u02b3\u02b4\u02b5\u02b6\u02b7\u02b8\u02b9\u02ba\u02bb\u02bc\u02bd\u02be\u02bf\u02c0\u02c1\u02c2\u02c3\u02c4\u02c5\u02c6\u02c7\u02c8\u02c9\u02ca\u02cb\u02cc\u02cd\u02ce\u02cf\u02d0\u02d1\u02d2\u02d3\u02d4\u02d5\u02d6\u02d7\u02d8\u02d9\u02da\u02db\u02dc\u02dd\u02de\u02df\u02e0\u02e1\u02e2\u02e3\u02e4\u02e5\u02e6\u02e7\u02e8\u02e9\u02ea\u02eb\u02ec\u02ed\u02ee\u02ef\u02f0\u02f1\u02f2\u02f3\u02f4\u02f5\u02f6\u02f7\u02f8\u02f9\u02fa\u02fb\u02fc\u02fd\u02fe\u02ff\u0280\u0281\u0282\u0283\u0284\u0285\u0286\u0287\u0288\u0289\u028a\u028b\u028c\u028d\u028e\u028f\u0290\u0291\u0292\u0293\u0294\u0295\u0296\u0297\u0298\u0299\u029a\u029b\u029c\u029d\u029e\u029f\u02a0\u02a1\u02a2\u02a3\u02a4\u02a5\u02a6\u02a7\u02a8\u02a9\u02aa\u02ab\u02ac\u02ad\u02ae\u02af\u02b0\u02b1\u02b2\u02b3\u02b4\u02b5\u02b6\u02b7\u02b8\u02b9\u02ba\u02bb\u02bc\u02bd\u02be\u02bf\u02c0\u02c1\u02c2\u02c3\u02c4\u02c5\u02c6\u02c7\u02c8\u02c9\u02ca\u02cb\u02cc\u02cd\u02ce\u02cf\u02d0\u02d1\u02d2\u02d3\u02d4\u02d5\u02d6\u02d7\u02d8\u02d9\u02da\u02db\u02dc\u02dd\u02de\u02df\u02e0\u02e1\u02e2\u02e3\u02e4\u02e5\u02e6\u02e7\u02e8\u02e9\u02ea\u02eb\u02ec\u02ed\u02ee\u02ef\u02f0\u02f1\u02f2\u02f3\u02f4\u02f5\u02f6\u02f7\u02f8\u02f9\u02fa\u02fb\u02fc\u02fd\u02fe\u02ff\u0280\u0281\u0282\u0283\u0284\u0285\u0286\u0287\u0288\u0289\u028a\u028b\u028c\u028d\u028e\u028f\u0290\u0291\u0292\u0293\u0294\u0295\u0296\u0297\u0298\u0299\u029a\u029b\u029c\u029d\u029e\u029f\u02a0\u02a1\u02a2\u02a3\u02a4\u02a5\u02a6\u02a7\u02a8\u02a9\u02aa\u02ab\u02ac\u02ad\u02ae\u02af\u02b0\u02b1\u02b2\u02b3\u02b4\u02b5\u02b6\u02b7\u02b8\u02b9\u02ba\u02bb\u02bc\u02bd\u02be\u02bf\u02c0\u02c1\u02c2\u02c3\u02c4\u02c5\u02c6\u02c7\u02c8\u02c9\u02ca\u02cb\u02cc\u02cd\u02ce\u02cf\u02d0\u02d1\u02d2\u02d3\u02d4\u02d5\u02d6\u02d7\u02d8\u02d9\u02da\u02db\u02dc\u02dd\u02de\u02df\u02e0\u02e1\u02e2\u02e3\u02e4\u02e5\u02e6\u02e7\u02e8\u02e9\u02ea\u02eb\u02ec\u02ed\u02ee\u02ef\u02f0\u02f1\u02f2\u02f3\u02f4\u02f5\u02f6\u02f7\u02f8\u02f9\u02fa\u02fb\u02fc\u02fd\u02fe\u02ff\u0280\u0281\u0282\u0283\u0284\u0285\u0286\u0287\u0288\u0289\u028a\u028b\u028c\u028d\u028e\u028f\u0290\u0291\u0292\u0293\u0294\u0295\u0296\u0297\u0298\u0299\u029a\u029b\u029c\u029d\u029e\u029f\u02a0\u02a1\u02a2\u02a3\u02a4\u02a5\u02a6\u02a7\u02a8\u02a9\u02aa\u02ab\u02ac\u02ad\u02ae\u02af\u02b0\u02b1\u02b2\u02b3\u02b4\u02b5\u02b6\u02b7\u02b8\u02b9\u02ba\u02bb\u02bc\u02bd\u02be\u02bf\u02c0\u02c1\u02c2\u02c3\u02c4\u02c5\u02c6\u02c7\u02c8\u02c9\u02ca\u02cb\u02cc\u02cd\u02ce\u02cf\u02d0\u02d1\u02d2\u02d3\u02d4\u02d5\u02d6\u02d7\u02d8\u02d9\u02da\u02db\u02dc\u02dd\u02de\u02df\u02e0\u02e1\u02e2\u02e3\u02e4\u02e5\u02e6\u02e7\u02e8\u02e9\u02ea\u02eb\u02ec\u02ed\u02ee\u02ef\u02f0\u02f1\u02f2\u02f3\u02f4\u02f5\u02f6\u02f7\u02f8\u02f9\u02fa\u02fb\u02fc\u02fd\u02fe\u02ff\u0280\u0281\u0282\u0283\u0284\u0285\u0286\u0287\u0288\u0289\u028a\u028b\u028c\u028d\u028e\u028f\u0290\u0291\u0292\u0293\u0294\u0295\u0296\u0297\u0298\u0299\u029a\u029b\u029c\u029d\u029e\u029f\u02a0\u02a1\u02a2\u02a3\u02a4\u02a5\u02a6\u02a7\u02a8\u02a9\u02aa\u02ab\u02ac\u02ad\u02ae\u02af\u02b0\u02b1\u02b2\u02b3\u02b4\u02b5\u02b6\u02b7\u02b8\u02b9\u02ba\u02bb\u02bc\u02bd\u02be\u02bf\u02c0\u02c1\u02c2\u02c3\u02c4\u02c5\u02c6\u02c7\u02c8\u02c9\u02ca\u02cb\u02cc\u02cd\u02ce\u02cf\u02d0\u02d1\u02d2\u02d3\u02d4\u02d5\u02d6\u02d7\u02d8\u02d9\u02da\u02db\u02dc\u02dd\u02de\u02df\u02e0\u02e1\u02e2\u02e3\u02e4\u02e5\u02e6\u02e7\u02e8\u02e9\u02ea\u02eb\u02ec\u02ed\u02ee\u02ef\u02f0\u02f1\u02f2\u02f3\u02f4\u02f5\u02f6\u02f7\u02f8\u02f9\u02fa\u02fb\u02fc\u02fd\u02fe\u02ff\u0280\u0281\u0282\u0283\u0284\u0285\u0286\u0287\u0288\u0289\u028a\u028b\u028c\u028d\u028e\u028f\u0290\u0291\u0292\u0293\u0294\u0295\u0296\u0297\u0298\u0299\u029a\u029b\u029c\u029d\u029e\u029f\u02a0\u02a1\u02a2\u02a3\u02a4\u02a5\u02a6\u02a7\u02a8\u02a9\u02aa\u02ab\u02ac\u02ad\u02ae\u02af\u02b0\u02b1\u02b2\u02b3\u02b4\u02b5\u02b6\u02b7\u02b8\u02b9\u02ba\u02bb\u02bc\u02bd\u02be\u02bf\u02c0\u02c1\u02c2\u02c3\u02c4\u02c5\u02c6\u02c7\u02c8\u02c9\u02ca\u02cb\u02cc\u02cd\u02ce\u02cf\u02d0\u02d1\u02d2\u02d3\u02d4\u02d5\u02d6\u02d7\u02d8\u02d9\u02da\u02db\u02dc\u02dd\u02de\u02df\u02e0\u02e1\u02e2\u02e3\u02e4\u02e5\u02e6\u02e7\u02e8\u02e9\u02ea\u02eb\u02ec\u02ed\u02ee\u02ef\u02f0\u02f1\u02f2\u02f3\u02f4\u02f5\u02f6\u02f7\u02f8\u02f9\u02fa\u02fb\u02fc\u02fd\u02fe\u02ff\u0280\u0281\u0282\u0283\u0284\u0285\u0286\u0287\u0288\u0289\u028a\u028b\u028c\u028d\u028e\u028f\u0290\u0291\u0292\u0293\u0294\u0295\u0296\u0297\u0298\u0299\u029a\u029b\u029c\u029d\u029e\u029f\u02a0\u02a1\u02a2\u02a3\u02a4\u02a5\u02a6\u02a7\u02a8\u02a9\u02aa\u02ab\u02ac\u02ad\u02ae\u02af\u02b0\u02b1\u02b2\u02b3\u02b4\u02b5\u02b6\u02b7\u02b8\u02b9\u02ba\u02bb\u02bc\u02bd\u02be\u02bf\u02c0\u02c1\u02c2\u02c3\u02c4\u02c5\u02c6\u02c7\u02c8\u02c9\u02ca\u02cb\u02cc\u02cd\u02ce\u02cf\u02d0\u02d1\u02d2\u02d3\u02d4\u02d5\u02d6\u02d7\u02d8\u02d9\u02da\u02db\u02dc\u02dd\u02de\u02df\u02e0\u02e1\u02e2\u02e3\u02e4\u02e5\u02e6\u02e7\u02e8\u02e9\u02ea\u02eb\u02ec\u02ed\u02ee\u02ef\u02f0\u02f1\u02f2\u02f3\u02f4\u02f5\u02f6\u02f7\u02f8\u02f9\u02fa\u02fb\u02fc\u02fd\u02fe\u02ff\u0280\u0281\u0282\u0283\u0284\u0285\u0286\u0287\u0288\u0289\u028a\u028b\u028c\u028d\u028e\u028f\u0290\u0291\u0292\u0293\u0294\u0295\u0296\u0297\u0298\u0299\u029a\u029b\u029c\u029d\u029e\u029f\u02a0\u02a1\u02a2\u02a3\u02a4\u02a5\u02a6\u02a7\u02a8\u02a9\u02aa\u02ab\u02ac\u02ad\u02ae\u02af\u02b0\u02b1\u02b2\u02b3\u02b4\u02b5\u02b6\u02b7\u02b8\u02b9\u02ba\u02bb\u02bc\u02bd\u02be\u02bf\u02c0\u02c1\u02c2\u02c3\u02c4\u02c5\u02c6\u02c7\u02c8\u02c9\u02ca\u02cb\u02cc\u02cd\u02ce\u02cf\u02d0\u02d1\u02d2\u02d3\u02d4\u02d5\u02d6\u02d7\u02d8\u02d9\u02da\u02db\u02dc\u02dd\u02de\u02df\u02e0\u02e1\u02e2\u02e3\u02e4\u02e5\u02e6\u02e7\u02e8\u02e9\u02ea\u02eb\u02ec\u02ed\u02ee\u02ef\u02f0\u02f1\u02f2\u02f3\u02f4\u02f5\u02f6\u02f7\u02f8\u02f9\u02fa\u02fb\u02fc\u02fd\u02fe\u02ff\u0280\u0281\u0282\u0283\u0284\u0285\u0286\u0287\u0288\u0289\u028a\u028b\u028c\u028d\u028e\u028f\u0290\u0291\u0292\u0293\u0294\u0295\u0296\u0297\u0298\u0299\u029a\u029b\u029c\u029d\u029e\u029f\u02a0\u02a1\u02a2\u02a3\u02a4\u02a5\u02a6\u02a7\u02a8\u02a9\u02aa\u02ab\u02ac\u02ad\u02ae\u02af\u02b0\u02b1\u02b2\u02b3\u02b4\u02b5\u02b6\u02b7\u02b8\u02b9\u02ba\u02bb\u02bc\u02bd\u02be\u02bf\u02c0\u02c1\u02c2\u02c3\u02c4\u02c5\u02c6\u02c7\u02c8\u02c9\u02ca\u02cb\u02cc\u02cd\u02ce\u02cf\u02d0\u02d1\u02d2\u02d3\u02d4\u02d5\u02d6\u02d7\u02d8\u02d9\u02da\u02db\u02dc\u02dd\u02de\u02df\u02e0\u02e1\u02e2\u02e3\u02e4\u02e5\u02e6\u02e7\u02e8\u02e9\u02ea\u02eb\u02ec\u02ed\u02ee\u02ef\u02f0\u02f1\u02f2\u02f3\u02f4\u02f5\u02f6\u02f7\u02f8\u02f9\u02fa\u02fb\u02fc\u02fd\u02fe\u02ff\u0280\u0281\u0282\u0283\u0284\u0285\u0286\u0287\u0288\u0289\u028a\u028b\u028c\u028d\u028e\u028f\u0290\u0291\u0292\u0293\u0294\u0295\u0296\u0297\u0298\u0299\u029a\u029b\u029c\u029d\u029e\u029f\u02a0\u02a1\u02a2\u02a3\u02a4\u02a5\u02a6\u02a7\u02a8\u02a9\u02aa\u02ab\u02ac\u02ad\u02ae\u02af\u02b0\u02b1\u02b2\u02b3\u02b4\u02b5\u02b6\u02b7\u02b8\u02b9\u02ba\u02bb\u02bc\u02bd\u02be\u02bf\u02c0\u02c1\u02c2\u02c3\u02c4\u02c5\u02c6\u02c7\u02c8\u02c9\u02ca\u02cb\u02cc\u02cd\u02ce\u02cf\u02d0\u02d1\u02d2\u02d3\u0			

heapdump.hprof x

Overview | dominator_tree x

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 @ 0xc0880550			
java.util.concurrent.ConcurrentHashMap @ 0xc0880550			
java.io.PrintStream @ 0xc08831e0			
java.lang.Module @ 0xc07b5428			
class java.nio.charset.Charset @ 0xc0880550			
java.lang.module.ModuleDescriptor @ 0xc0880550			
java.util.zip.ZipFile\$Source @ 0xc0880550			
java.lang.module.Configuration @ 0xc0880550			
jdk.internal.loader.ClassLoaders\$AppClassLoader @ 0xc0880550			
class java.lang.Integer\$IntegerCache @ 0xc0880550			
class sun.nio.cs.MS1252\$Holder @ 0xc0880550			
java.lang.ModuleLayer @ 0xc0790550			
java.lang.module.ModuleDescriptor @ 0xc0880550			
class jdk.internal.module.SystemModuleLayer @ 0xc0880550			
java.lang.Module @ 0xc07b72d0			
java.lang.Module @ 0xc07b5a20			
char[1792] @ 0xc08835d8 \u0000			
jdk.internal.module.ServicesCatalog @ 0xc0880550			
class jdk.internal.module.ArchivedModuleGraph @ 0xc0880550			
...			

Context Menu:

- with outgoing references
- with incoming references
- List objects
- Show objects by class
- Path To GC Roots
- Merge Shortest Paths to GC Roots
- Java Basics
- Java Collections
- Leak Identification
- Export Snapshot
- Immediate Dominators
- Show Retained Set
- Copy
- Search Queries...
- Calculate Minimum Retained Size (quick approx.)
- Calculate Precise Retained Size
- Columns...

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.

Overview		dominator_tree	list_objects [selection of 'HashMap @ 0xc0880550'] -inbound ×		
Class Name				Shallow Heap	Retained Heap
<Regex>				<Numeric>	<Numeric>
java.util.HashMap @ 0xc0880550				48	1,07,09,46,392
<Java Local> java.lang.Thread @ 0xc03f71f8 main Thread				112	1,376
myMap com.buggyapp.memoryleak.MapManager @ 0xc0880540				16	16
<Java Local> java.lang.Thread @ 0xc03f71f8 main Thread				112	1,376
mapManager com.buggyapp.memoryleak.Object3 @ 0xc08805f0				16	16
<Java Local> java.lang.Thread @ 0xc03f71f8 main Thread				112	1,376
object3 com.buggyapp.memoryleak.Object2 @ 0xc0880670				16	16
<Java Local> java.lang.Thread @ 0xc03f71f8 main Thread				112	1,376
object2 com.buggyapp.memoryleak.Object1 @ 0xc08806f0				16	16
<Java Local> java.lang.Thread @ 0xc03f71f8 main Thread				112	1,376
object1 class com.buggyapp.memoryleak.MemoryLeakDemo @ 0xc0880700				8	32
[1] java.lang.Object[10] @ 0xc03fb6b0				56	88
Σ Total: 2 entries					
Σ Total: 2 entries					
Σ Total: 2 entries					
Σ Total: 2 entries					
Σ Total: 2 entries					

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

The screenshot shows the IntelliJ IDEA interface with the 'list_objects' tool window open. The tool window displays a table of objects with columns for Class Name, Shallow Heap, Retained Heap, and Percentage. The selected object is 'java.util.HashMap @ 0xc0880550'. A context menu is open over the selected object, showing options like 'List objects', 'Show objects by class', 'Path To GC Roots', 'Merge Shortest Paths to GC Roots', 'Java Basics', 'Java Collections', 'Leak Identification', 'Export Snapshot', 'Immediate Dominators', 'Show Retained Set', 'Copy', 'Search Queries...', 'Calculate Minimum Retained Size (quick approx.)', 'Calculate Precise Retained Size', and 'Columns...'. The 'List objects' option is highlighted.

Class Name	Shallow Heap	Retained Heap	Percentage
java.util.HashMap @ 0xc0880550	1,072,160	1,072,160	20.00%
class java.lang.System @ 0xc0880550	0	0	0.00%
java.util.concurrent.ConcurrentHashMap @ 0xc0880550	0	0	0.00%
java.io.PrintStream @ 0xc0880550	0	0	0.00%
java.lang.Module @ 0xc0880550	0	0	0.00%
class java.nio.charset.Charset @ 0xc0880550	0	0	0.00%
java.lang.module.ModuleInfo @ 0xc0880550	0	0	0.00%
java.util.zip.ZipFile\$Source @ 0xc0880550	0	0	0.00%
java.lang.module.Configuration @ 0xc0880550	0	0	0.00%
jdk.internal.loader.ClassLoader @ 0xc0880550	0	0	0.00%
class java.lang.Integer @ 0xc0880550	0	0	0.00%
class sun.nio.cs.MS1252\$Encoder @ 0xc0880550	0	0	0.00%
java.lang.ModuleLayer @ 0xc0880550	0	0	0.00%
java.lang.module.ModuleInfo @ 0xc0880550	0	0	0.00%
class jdk.internal.module.Service @ 0xc0880550	0	0	0.00%
java.lang.Module @ 0xc0880550	0	0	0.00%
java.lang.Module @ 0xc0880550	0	0	0.00%
char[1792] @ 0xc08835d8	0	0	0.00%
jdk.internal.module.Service @ 0xc0880550	0	0	0.00%
class jdk.internal.module.ArchivedModuleGraph @ 0xc0880550	8	3,344	0.00%

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 Stringggggggggggggggggggggggggggggggg."

The screenshot displays the Eclipse IDE's Memory Analyzer (MAT) interface. The left pane shows the 'dominator_tree' view for a selection of 'HashMap @ 0xc0880550'. The middle pane shows a list of objects with columns for Class Name, Shallow Heap, and Retained Heap. The right pane shows the 'Inspector' view for the selected object, displaying its type as 'String' and its value as 'Large string'.

Class Name	Shallow Heap	Retained Heap
java.util.HashMap @ 0xc0880550	48	1,07,09,46,392
<class> class java.util.HashMap @ 0xc08c6e0 System	40	272
table java.util.HashMap\$Node[2097152] @ 0xc240000	83,88,624	1,07,09,46,344
class java.util.HashMap\$Node[] @ 0xc08c6b48	0	0
[1358482] java.util.HashMap\$Node @ 0xc0000288	32	1,512
<class> class java.util.HashMap\$Node @ 0xc08c6b48	8	32
key java.lang.String @ 0xc0000008 key1844	24	48
value java.lang.String @ 0xc00002d8 Large string	24	672
next java.util.HashMap\$Node @ 0xc0c572c0	32	760
Total: 4 entries		
[1358481] java.util.HashMap\$Node @ 0xc0000578	32	1,512
[1358480] java.util.HashMap\$Node @ 0xc0000868	32	1,512
[1489775] java.util.HashMap\$Node @ 0xc0000b58	32	752
[1522543] java.util.HashMap\$Node @ 0xc0000e48	32	752
[1489774] java.util.HashMap\$Node @ 0xc0001138	32	752
[1522542] java.util.HashMap\$Node @ 0xc0001428	32	752
[1489772] java.util.HashMap\$Node @ 0xc0001718	32	752
[1489767] java.util.HashMap\$Node @ 0xc0001a08	32	752
[1522535] java.util.HashMap\$Node @ 0xc0001cf8	32	752
[1358518] java.util.HashMap\$Node @ 0xc0002270	32	1,512
[1358517] java.util.HashMap\$Node @ 0xc0002560	32	752
[1358515] java.util.HashMap\$Node @ 0xc0002850	32	1,512
[1358514] java.util.HashMap\$Node @ 0xc0002b40	32	1,512
[1358513] java.util.HashMap\$Node @ 0xc0002e30	32	1,512

Type	Name	Value
boolean	hashIsZero	false
int	hash	0
byte	coder	6
ref	value	Large string

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

