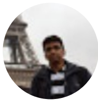
Basic Concepts of Java Heap Dump Analysis with MAT



Isuru Perera
Jul 2 · 14 min read

Eclipse Memory Analyzer Tool (MAT) is by far the best tool to analyze Java Heap Dumps. A heap dump is a snapshot of the heap memory of a Java process at a given time. The snapshot mainly consists of Java objects and classes.

Let's go through some of the basic concepts of Java heap dump analysis with MAT. In order to explain the concepts clearly, I developed a very simple Java sample application.

You can clone the `https://github.com/chrishantha/sample-java-programs` repository and use Apache Maven to build. The sample application used to explain all concepts are in `memoryref` directory.

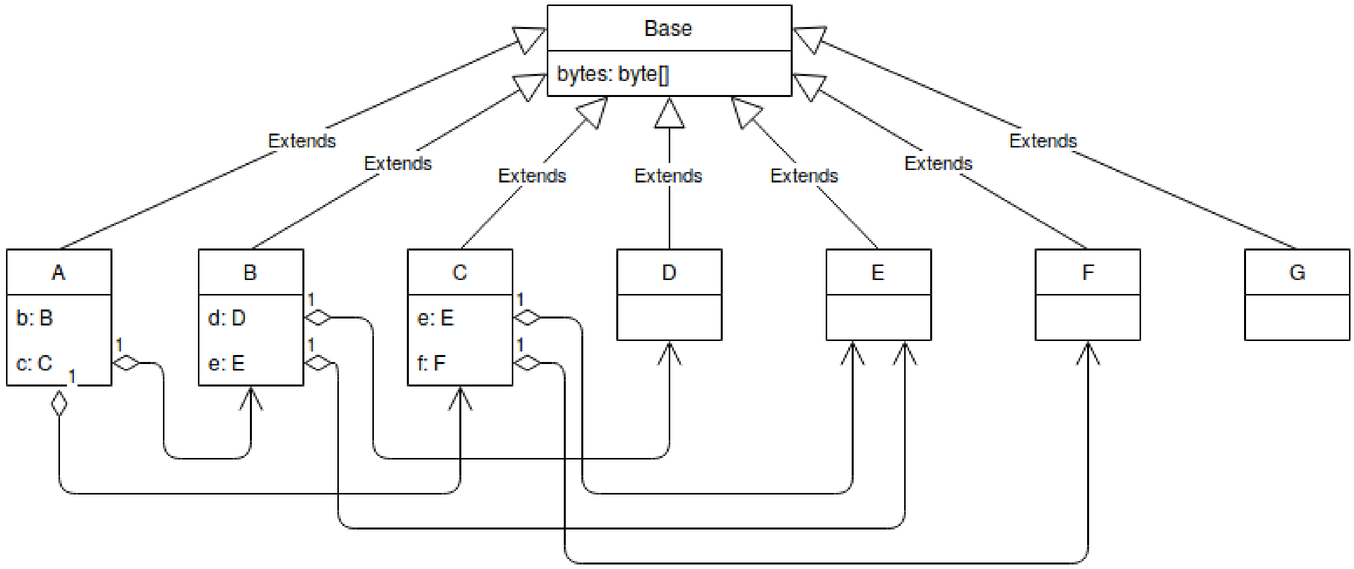
chrishantha/sample-java-programs
Sample Java programs to demonstrate performance issues - chrishantha/sample-java-...
github.com

```
git clone https://github.com/chrishantha/sample-java-programs
cd sample-java-programs
mvn clean install
cd memoryref
```

Please note that I'm using Oracle JDK 8 for all examples.

The sample application

The sample application mainly uses the classes named as `A`, `B`, `C`, `D`, `E`, `F`, and `G` to explain the basic concepts. Each of the previously mentioned classes extend to `Base` class, which creates a significantly large byte array in the constructor (so that the objects will consume more memory in run-time).

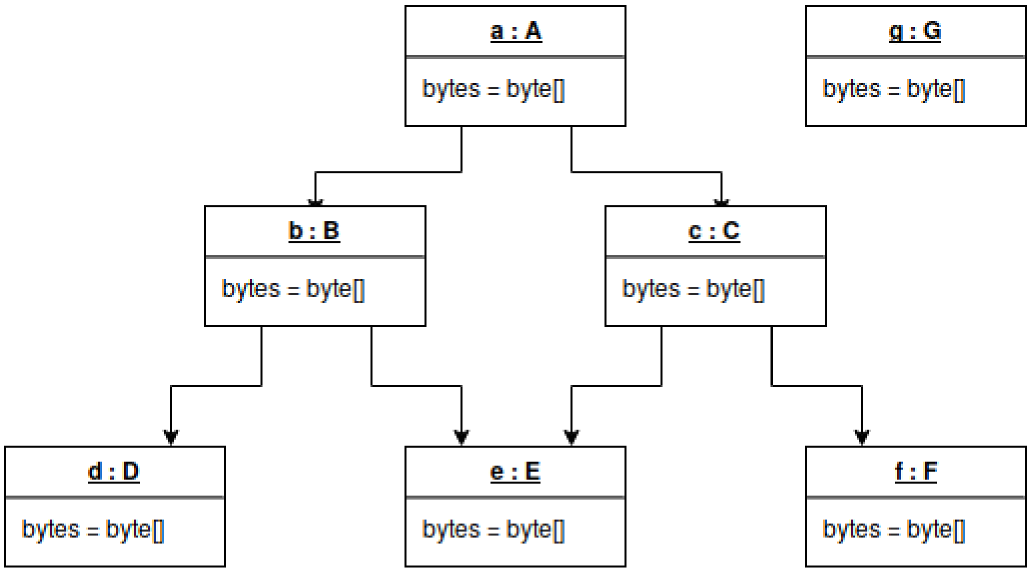


Class Diagram

The sample application creates an object from class `A` in the main thread and another object from class `G` in a daemon thread.

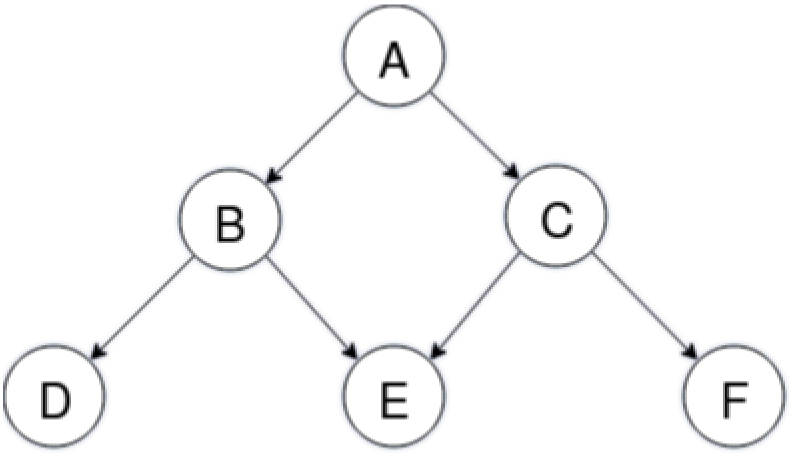
The instance created from class `A` references the instances created from other classes as shown below.

The class `G` does not refer any other classes.



Object Diagram

Perhaps, following diagram to show the references among objects will be more clear.



Let's refer the source code to understand each class in detail.

Following `Base` class creates a large byte array as explained above. The classes `A`, `B`, `C`, `D`, `E`, `F`, and `G` extends this `Base` class.

```
public abstract class Base {

    private final byte[] bytes;
    private static final Random random = new Random();

    Base() {
        bytes = new byte[(10 * (1 + random.nextInt(10))) * 1024 *
1024];
    }

    public long getSize() {
        return JavaAgent.getObjectSize(this) +
JavaAgent.getObjectSize(bytes);
    }
}
```

An instance created by class `A` will refer the instances of classes `B` and `C`.

```
public class A extends Base {

    final B b;
    final C c;

    A() {
        b = new B();
        c = new C(b);
    }

    @Override
    public long getSize() {
```

```
        return super.getSize() + b.getSize() + c.getSize();
    }
}
```

An instance created by class `B` will refer the instances of classes `D` and `E`.

```
public class B extends Base {

    final D d;
    final E e;

    B() {
        d = new D();
        e = new E();
    }

    @Override
    public int getSize() {
        return super.getSize() + d.getSize() + e.getSize();
    }
}
```

An instance created by class `C` will refer the instances of classes `E` and `F`. It's important to note that the reference of the object `e` is a copy from the `B` class.

```
public class C extends Base {

    final E e;
    final F f;

    C(B b) {
        this.e = b.e;
        f = new F();
    }

    @Override
    public int getSize() {
        return super.getSize() + f.getSize();
    }

}
```

The `D`, `E`, `F` and `G` classes just extend the `Base` class.

```
public class D extends Base {

}

public class E extends Base {

}

public class F extends Base {

}

public class G extends Base {

}
```

Following is the main application. As explained earlier, the main thread creates a new instance of class `A` and another background thread instantiates an object from class `G`.

```
public class MemoryRefApplication implements SampleApplication {

    @Override
    public void start() {
        A a = new A();
        long size = a.getSize();
        System.out.format("The retained heap size of object A is %d
bytes (~%d MiB).%n",
                size, (size / (1024 * 1024)));
        long objectSize = JavaAgent.getObjectSize(a);
        if (objectSize > 0) {
            System.out.format("The shallow heap size of object A is
%d bytes.%n", objectSize);
        } else {
            System.out.println("WARNING: Java Agent is not
initialized properly.");
        }
        Thread backgroundThread = new Thread() {

            private long getSize() {
                G g = new G();
                return g.getSize();
            }

            @Override
            public void run() {
                long size = getSize();
                System.out.format("The size of object allocated
within the background thread was %d bytes (~%d MiB).%n",
                        size, (size / (1024 * 1024)));
                try {
```

```
                    Thread.sleep(Long.MAX_VALUE);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        };
        backgroundThread.setName("Background Thread");
        backgroundThread.setDaemon(true);
        backgroundThread.start();
        try {
            System.out.println("Press Enter to exit.");
            System.in.read();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    @Override
    public String toString() {
        return "MemoryRefApplication";
    }
}
```

`JavaAgent` class is used to get an object's size using the Java Instrumentation API.

```
public class JavaAgent {
    private static volatile Instrumentation instrumentation;

    public static void premain(final String agentArgs, final
Instrumentation instrumentation) {
        JavaAgent.instrumentation = instrumentation;
    }

    public static long getObjectSize(final Object object) {
        if (instrumentation == null) {
            return -1L;
        }
        return instrumentation.getObjectSize(object);
    }
}
```

Running the sample application

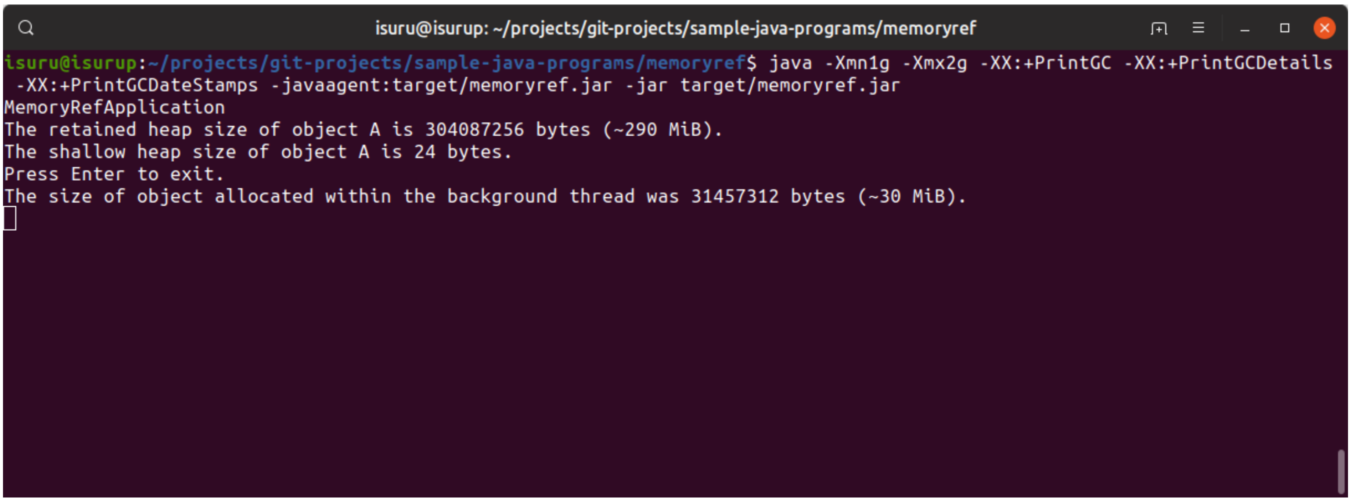
The application creates instances of `A`, `B`, `C`, `D`, `E`, `F`, and `G` classes. Each object may take at least 100 Mebibytes. We need at least 700MiB for Eden space to create the mentioned objects (Eden is a part of Young Generation as explained in my previous blog post). Therefore, I specified 1GB for Young Generation (leaving enough room for survivor spaces)

I also enabled GC logs to monitor GC activity.

Following is the command I executed.

```
java -Xmn1g -Xmx2g -XX:+PrintGC -XX:+PrintGCDetails -
XX:+PrintGCDateStamps -javaagent:target/memoryref.jar -jar
target/memoryref.jar
```

Following is the sample output.




```
MemoryRefApplication
The retained heap size of object A is 304087256 bytes (~290 MiB).
The shallow heap size of object A is 24 bytes.
Press Enter to exit.
The size of object allocated within the background thread was
31457312 bytes (~30 MiB).
```

You can see that the retained heap and shallow size of object `A` is printed. The size of the object allocated within the background thread is also printed.

Now we have a sample application to go through the basic concepts of Java heap dump analysis.

Getting a memory snapshot (heap dump)

The `jmap` command is the best way to get a heap dump of a Java application whenever you want.

You can just execute `jmap` command to see all options.

```
$ jmap
Usage:
    jmap [option] <pid>
        (to connect to running process)
    jmap [option] <executable <core>
        (to connect to a core file)
    jmap [option] [server_id@]<remote server IP or hostname>
        (to connect to remote debug server)

where <option> is one of:
    <none>               to print same info as Solaris pmap
    -heap                to print java heap summary
    -histo[:live]        to print histogram of java object heap; if
the "live"
                         suboption is specified, only count live
objects
    -clstats             to print class loader statistics
    -finalizerinfo       to print information on objects awaiting
finalization
    -dump:<dump-options> to dump java heap in hprof binary format
                         dump-options:
                           live         dump only live objects; if
not specified,
                                        all objects in the heap are
dumped.
                           format=b     binary format
                           file=<file>  dump heap to <file>
                         Example: jmap -
dump:live,format=b,file=heap.bin <pid>
    -F                   force. Use with -dump:<dump-options> <pid>
or -histo
                         to force a heap dump or histogram when
<pid> does not
                         respond. The "live" suboption is not
supported
                         in this mode.
    -h | -help           to print this help message
    -J<flag>             to pass <flag> directly to the runtime
system
```

I used `-dump` option to get a heap dump.

```
mkdir heap1
jmap -dump:file=heap1/heap.hprof $(pgrep -f memoryref)
```

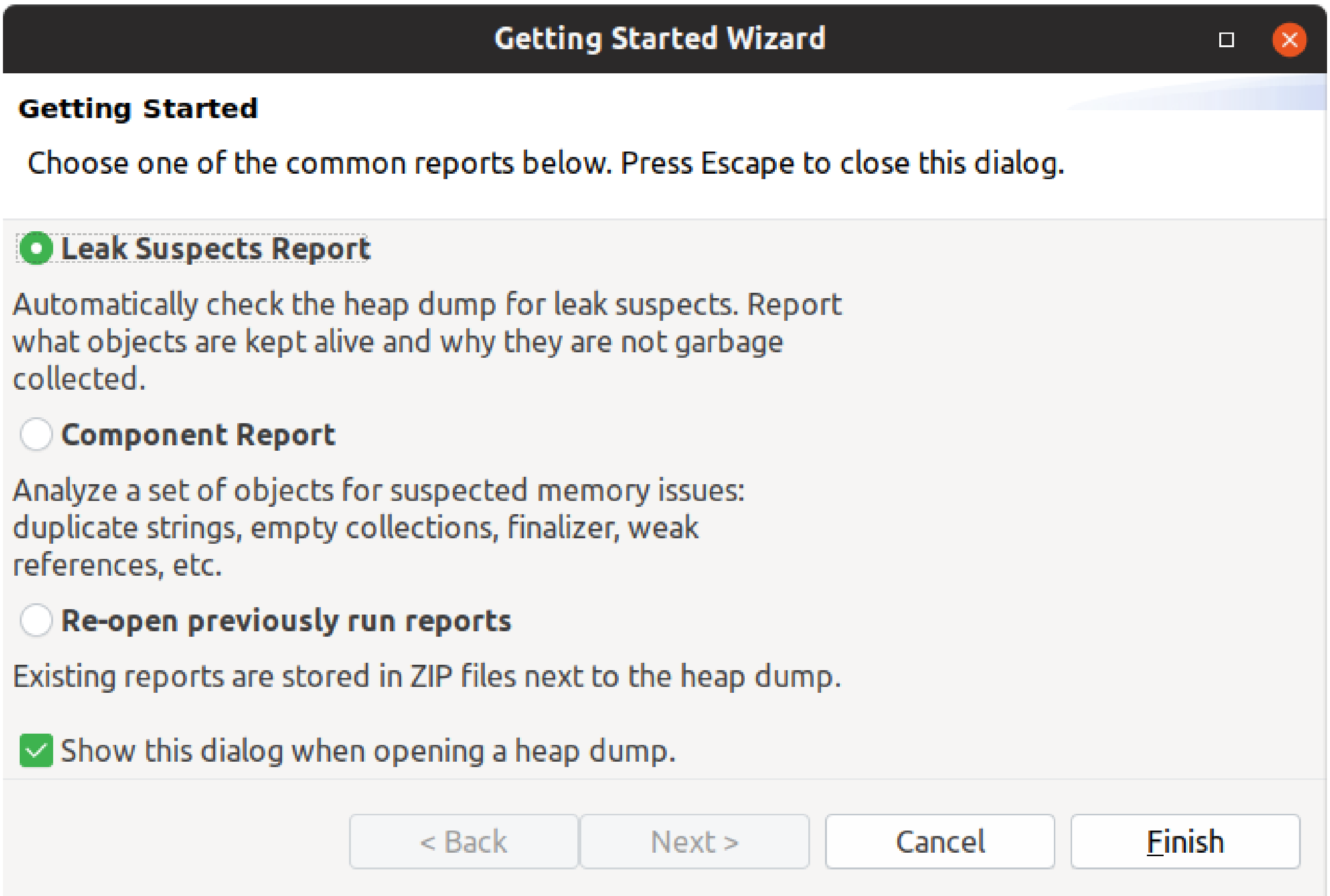
Note that I did not use `live` option here.

When we use the `live` option, a full GC will be triggered in order to sweep away unreachable objects and then dump only live objects.

Analyzing the heap dump

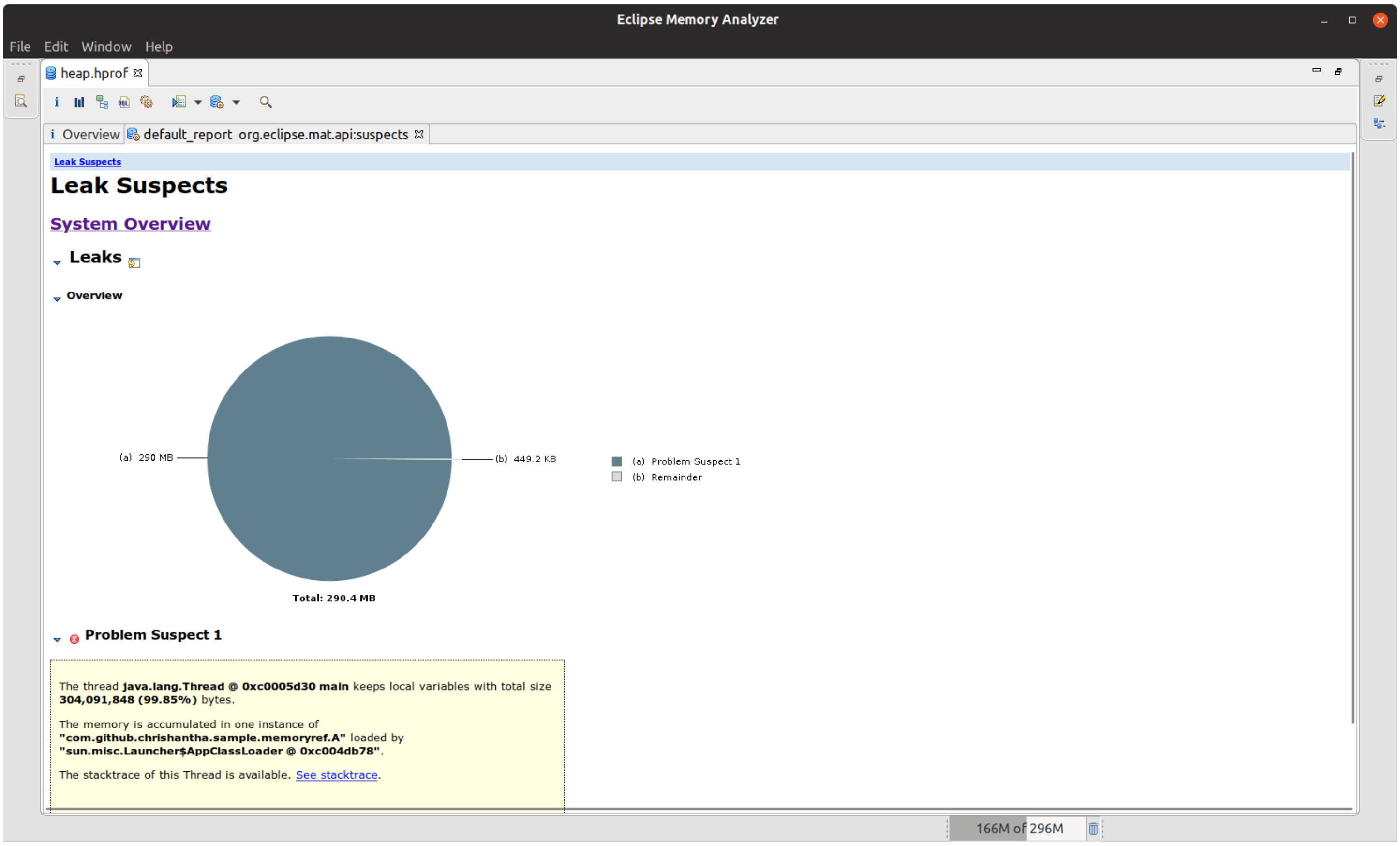
Let's analyze the heap dump using Eclipse MAT.

By default, when you open a heap dump using Eclipse MAT, the Getting Started wizard will give you options to generate "Leak Suspects Report" or "Component Report".



I generated a "Leak Suspects Report".

Leak Suspects Report



> *The thread **java.lang.Thread @ 0xc0005d30 main** keeps local variables with total size **304,091,848 (99.85%)** bytes.*
>
> *The memory is accumulated in one instance of **"com.github.chrishantha.sample.memoryref.A"** loaded by **"sun.misc.Launcher$AppClassLoader @ 0xc004db78"**.*

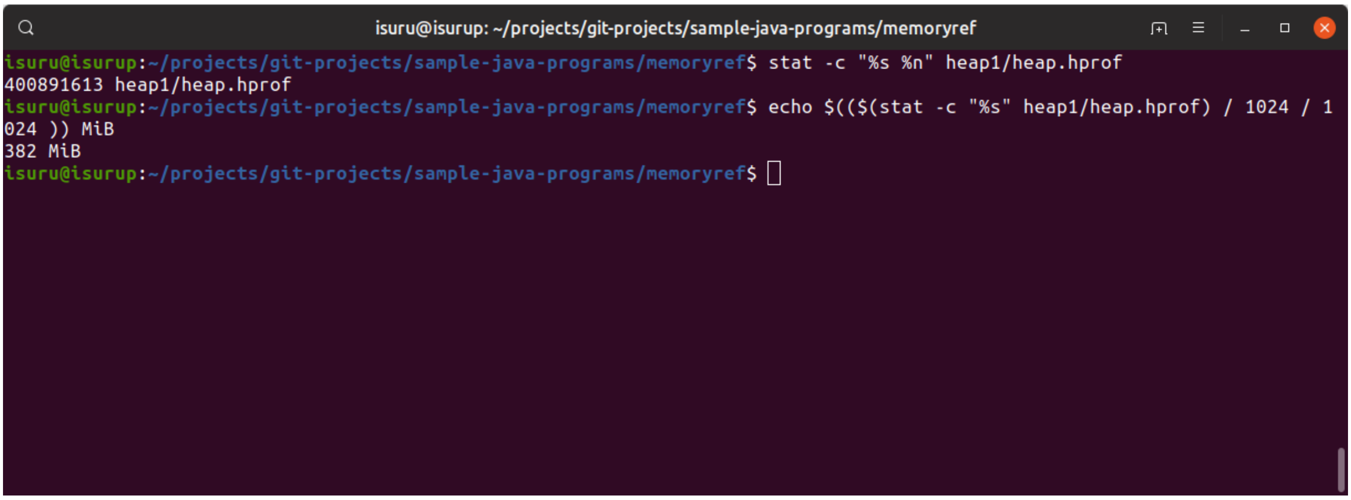
This report showed that main threads keeps local variables with total size **304,091,848 (99.85%)** bytes. This is slightly greater than the object A's size of **304,087,256** bytes. Total heap size shown in the report is **290.4 MB**

This is an interesting observation. The main thread is taking **99.85%** of the heap size. Why? What happened to the instance created in the background thread?

It seems that the object created in the background thread is not counted towards the total heap size.

What is the heap dump file size?

```
stat -c "%s %n" heap1/heap.hprof
echo $(($(stat -c "%s" heap1/heap.hprof) / 1024 / 1024 )) MiB
```



The heap dump file size is actually **382 MiB**. There is a reason why object created in the background thread is not counted towards the total heap size.

Reachability

So, any basic Garbage Collection algorithm mainly has two steps.

1. Mark all live objects. These objects are reachable from Garbage Collection roots.
2. Sweep away all unreachable objects.

So, basically there are two types of objects in memory.

1. **Reachable**: Any object with a direct path from a Garbage Collection root.
2. **Unreachable**: Any object with no direct path from a Garbage Collection root.

Reachability and Garbage Collection roots

What are Garbage Collection roots? The GC roots are the objects accessible from outside the heap. The GC algorithms build a tree of live objects starting from these GC roots.

Following are some GC roots.

1. **System Class**: Class loaded by bootstrap/system class loader.
2. **Thread Block**: Objects referred to from currently active thread blocks. (Basically all objects in active thread blocks when a GC is happening are GC roots)
3. **Thread**: Active Threads
4. **Java Local**: All local variables (parameters, objects or methods in thread stacks)

5. **JNI Local**: Local variables in native code

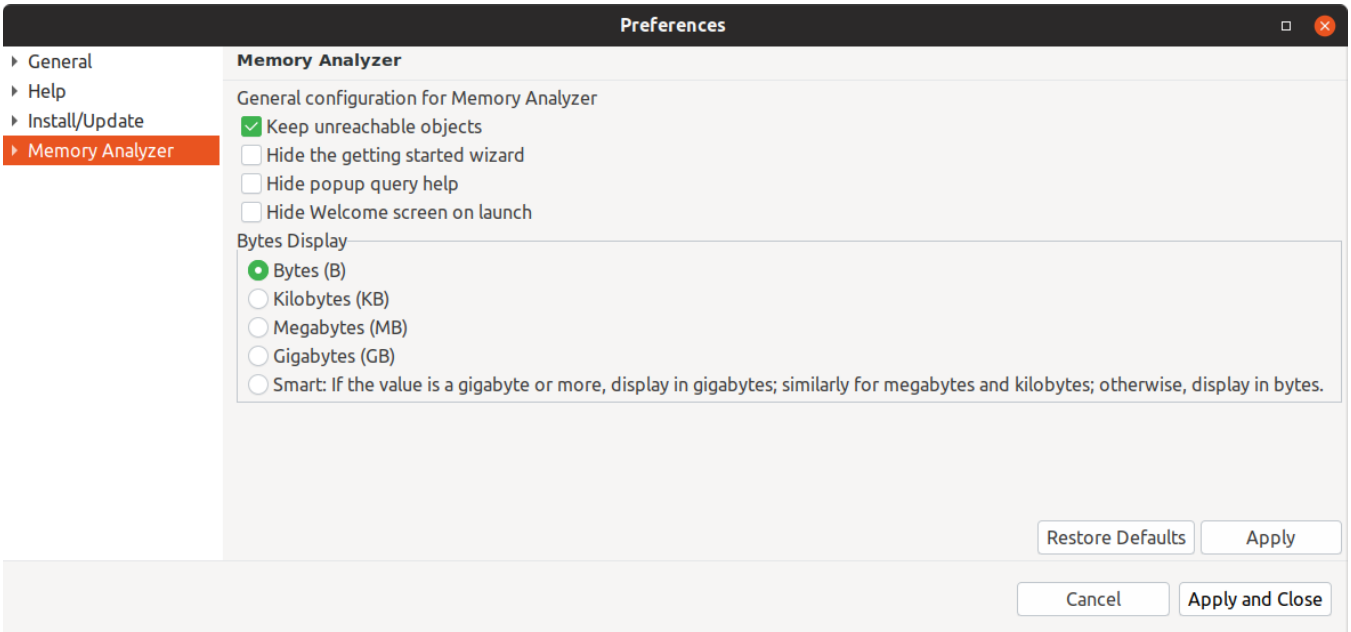
6. **JNI Global**: Global variables in native code

So, let's get back to our problem. I did not use the live option and therefore, there was no GC event as well.

What happened was that the Eclipse MAT removed the unreachable objects.

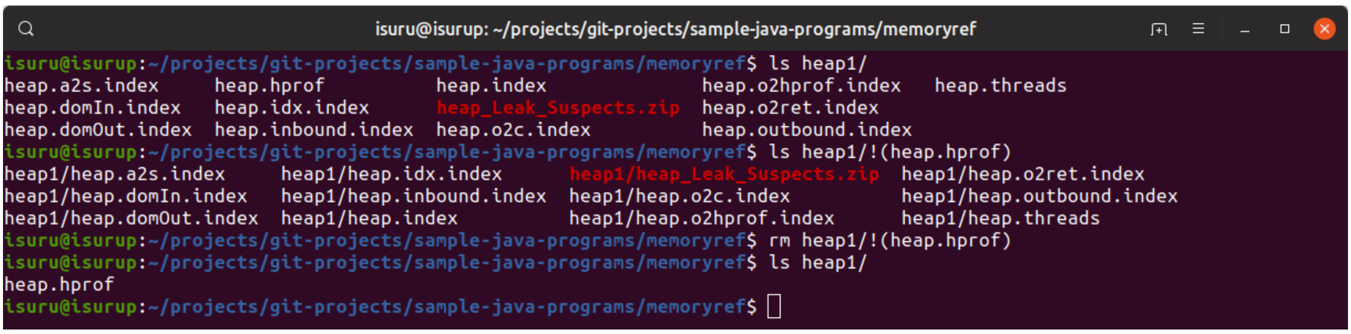
So, if the heap dump file size and the total heap size shown in Eclipse MAT is significantly different, the Eclipse MAT might have actually removed any unreachable objects.

We can turn off this behavior, by setting "Keep unreachable objects" option in *Eclipse MAT -> Window -> Preferences -> Memory Analyzer*.



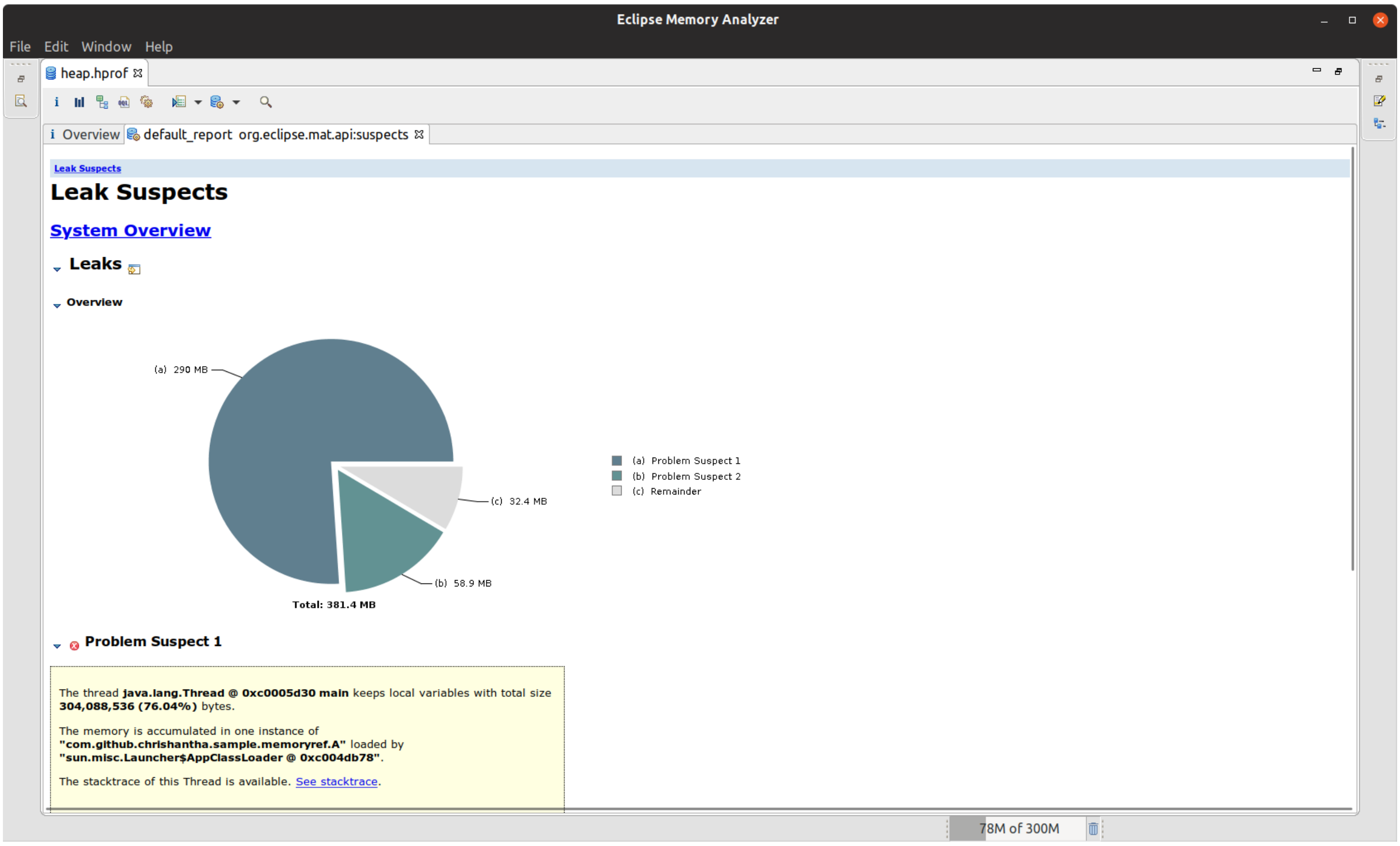
Let's open the heap dump again with this option.

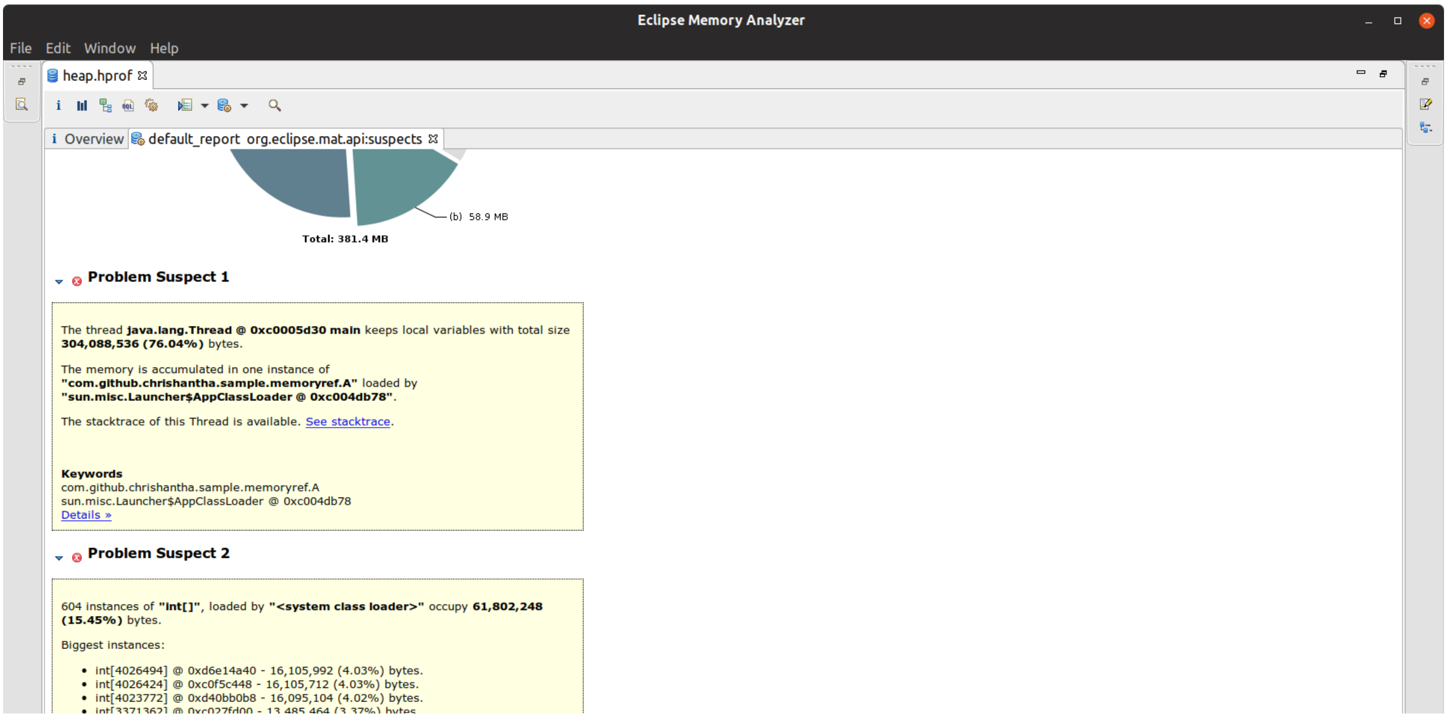
Before opening the same heap dump again, we need to remove the reports, indexes, etc. created by Eclipse MAT.

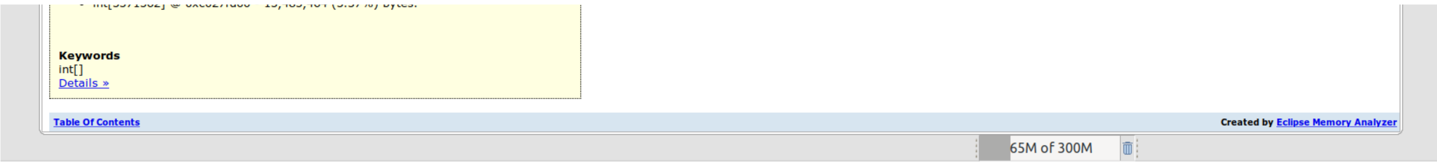


I used the command “ `rm heap1/!(heap.hprof)` ” to delete the Leak Suspects report, indexes, etc. (Be careful when you use `rm` command and make sure you use it wisely)

I generated “Leak Suspects Report” again.







After setting the option in Eclipse MAT to “Keep unreachable objects”, we can see the that total heap size is actually closer to the heap dump file size.

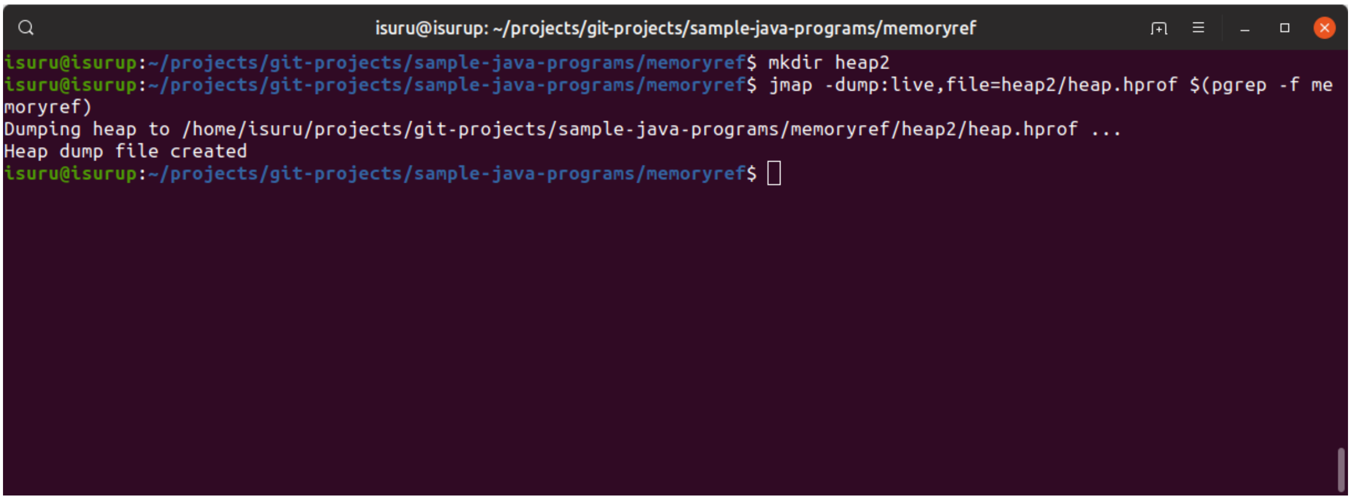
> ***Problem Suspect 1***
>
> *The thread **java.lang.Thread @ 0xc0005d30 main** keeps local variables with total size **304,088,536 (76.04%)** bytes.*
>
> *The memory is accumulated in one instance of **“com.github.chrishantha.sample.memoryref.A”** loaded by **“sun.misc.Launcher$AppClassLoader @ 0xc004db78”**.*
>
> ***Problem Suspect 2***
>
> *604 instances of **“int[]”**, loaded by **“<system class loader>”** occupy **61,802,248 (15.45%)** bytes.*
>
> *Biggest instances:*
>
> *int[4026494] @ 0xd6e14a40–16,105,992 (4.03%) bytes.*
>
> *int[4026424] @ 0xc0f5c448–16,105,712 (4.03%) bytes.*
>
> *int[4023772] @ 0xd40bb0b8–16,095,104 (4.02%) bytes.*
>
> *int[3371362] @ 0xc027fd00–13,485,464 (3.37%) bytes.*

The objects shown under “Problem Suspect 2” are actually the unreachable objects and those objects are eligible for garbage collection.

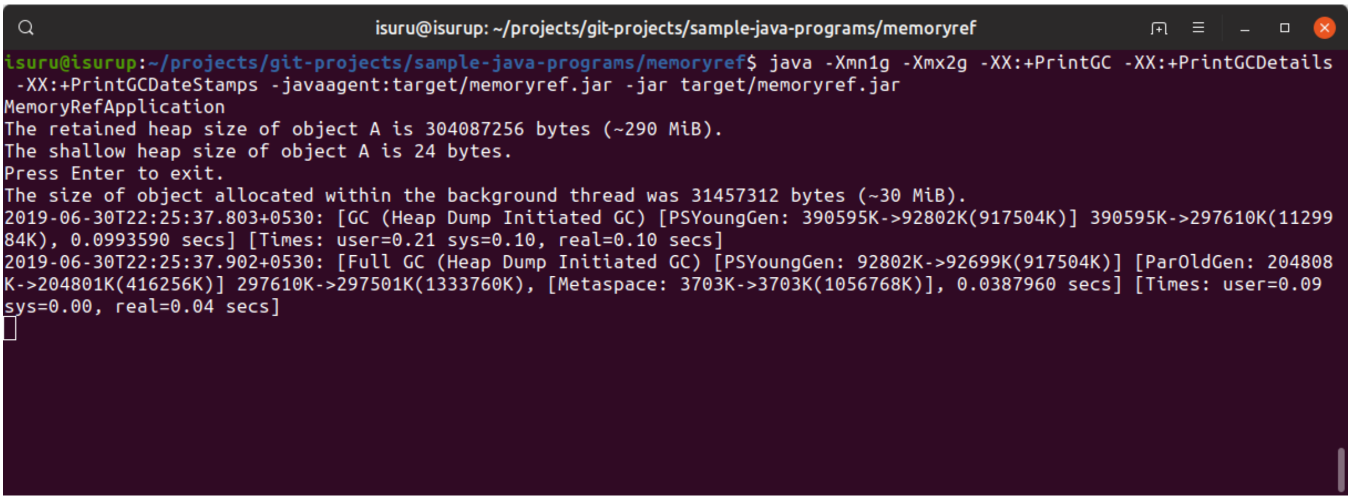
So, Eclipse MAT is actually doing us a favor by removing unreachable objects by default. We usually don’t need to spend time with any unreachable objects.

I got another heap dump with `live` option.

```
mkdir heap2
jmap -dump:live,file=heap2/heap.hprof $(pgrep -f memoryref)
```

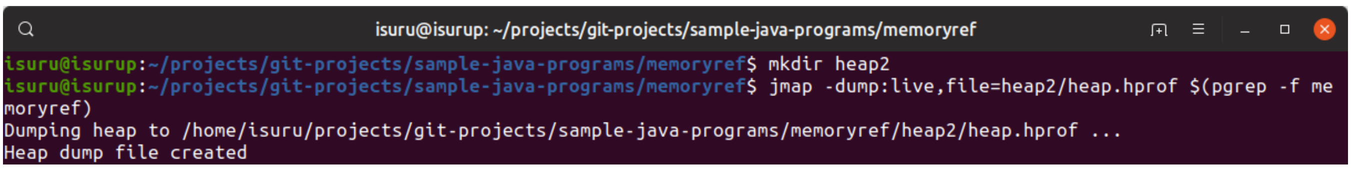


When the `jmap` command was executed, a full GC event was triggered in the application.




```
2019-06-30T22:25:37.803+0530: [GC (Heap Dump Initiated GC)
[PSYoungGen: 390595K->92802K(917504K)] 390595K->297610K(1129984K),
0.0993590 secs] [Times: user=0.21 sys=0.10, real=0.10 secs]
2019-06-30T22:25:37.902+0530: [Full GC (Heap Dump Initiated GC)
[PSYoungGen: 92802K->92699K(917504K)] [ParOldGen: 204808K-
>204801K(416256K)] 297610K->297501K(1333760K), [Metaspace: 3703K-
>3703K(1056768K)], 0.0387960 secs] [Times: user=0.09 sys=0.00,
real=0.04 secs]
```

Here, we can see the cause of GC events as "Heap Dump Initiated GC".



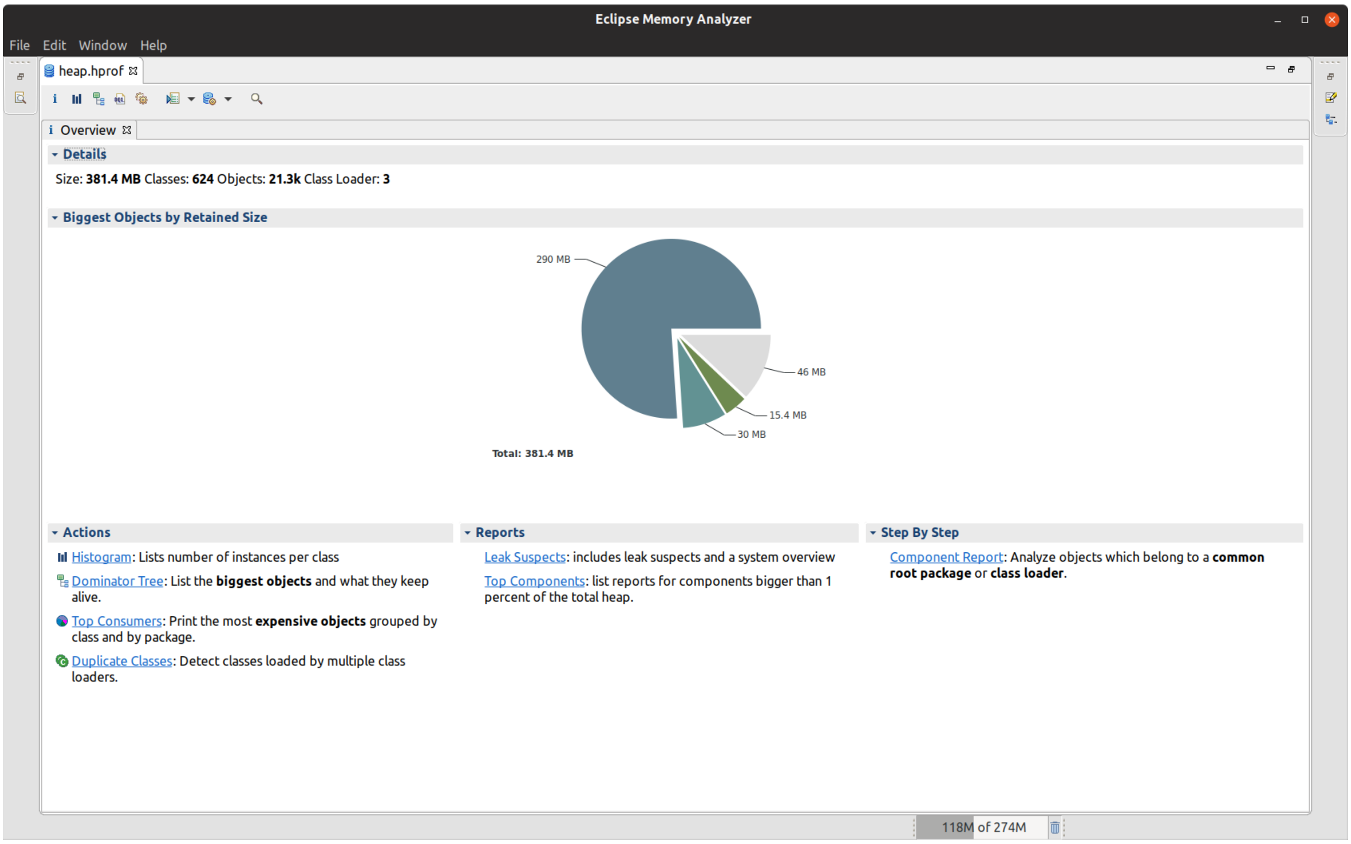
```
isuru@isurup:~/projects/git-projects/sample-java-programs/memoryref$ stat -c "%s %n" heap2/heap.hprof
305408801 heap2/heap.hprof
isuru@isurup:~/projects/git-projects/sample-java-programs/memoryref$ echo $(($(stat -c "%s" heap2/heap.hprof) / 1024 / 1
024 )) MiB
291 MiB
isuru@isurup:~/projects/git-projects/sample-java-programs/memoryref$
```

The new heap dump file size is **291 MiB**, which was the total heap size when all unreachable objects were removed.

Heap Dump Overview

Let's continue to go through the basic concepts with `heap1/heap.hprof` analysis.

The default view of Eclipse MAT after opening a heap dump is the "Overview" pane.



Overview pane

The pie chart shows the biggest objects by **retained size.** From Leak Suspects report, we know that 290MB is the retained size for the main thread.

From the heap dump overview page, we can do following actions.

1. **Histogram**: Lists number of instances per class

2. **Dominator Tree**: List the biggest objects and what they keep alive.
3. **Top Consumers**: Print the most expensive objects grouped by class and by package.
4. **Duplicate Classes**: Detect classes loaded by multiple class loaders.

These actions will help to figure out which objects actually take more memory.

Shallow vs Retained Heap

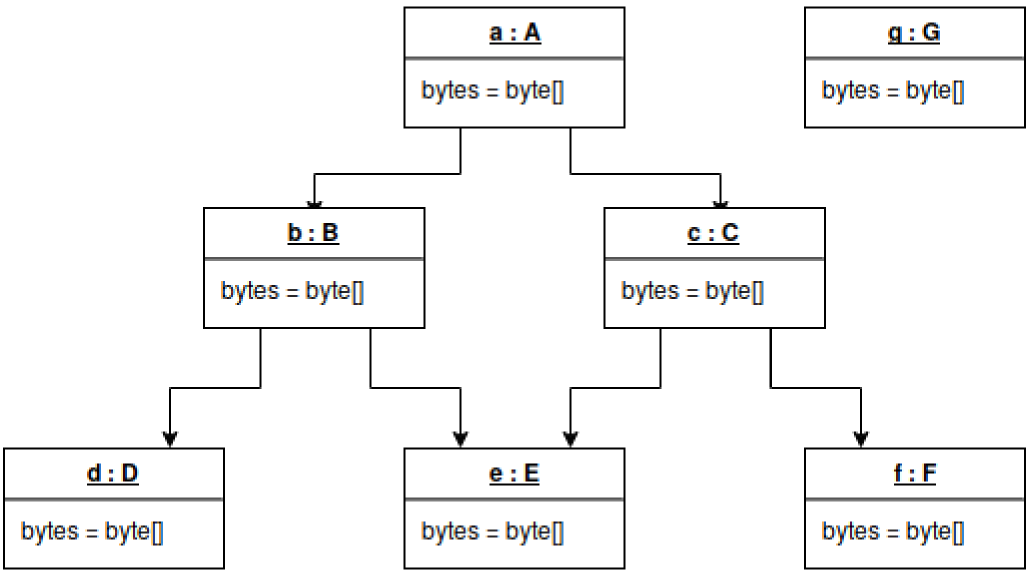
We talked about shallow and retained heap sizes earlier. Let's go through the definitions first.

Shallow heap is the memory consumed by one object. The actual memory consumed by one object depends on the underlying architecture.

Retained Heap of an object is the sum of shallow sizes of all objects in the retained set of that object.

Retained set of an object is the set of objects, which would be removed by GC when that particular object is garbage collected.

For example, let's see the object diagram of object `a` of type `A`. (This is the same image shown above)



Object diagram

Retained set of object `a` is the set of objects which would be removed by GC when object `a` is garbage collected. So, the retained set of the object `a` is `{b, c, d, e, f}`.

The retained heap of object `a` is the sum shallow heap sizes of `a` , `b` , `c` , `d` , `e` , and `f` including the shallow heap sizes of byte arrays in each object.

In order to find out the retained heap and shallow heap sizes for the biggest objects, we can use the **Dominator Tree** in Eclipse MAT.

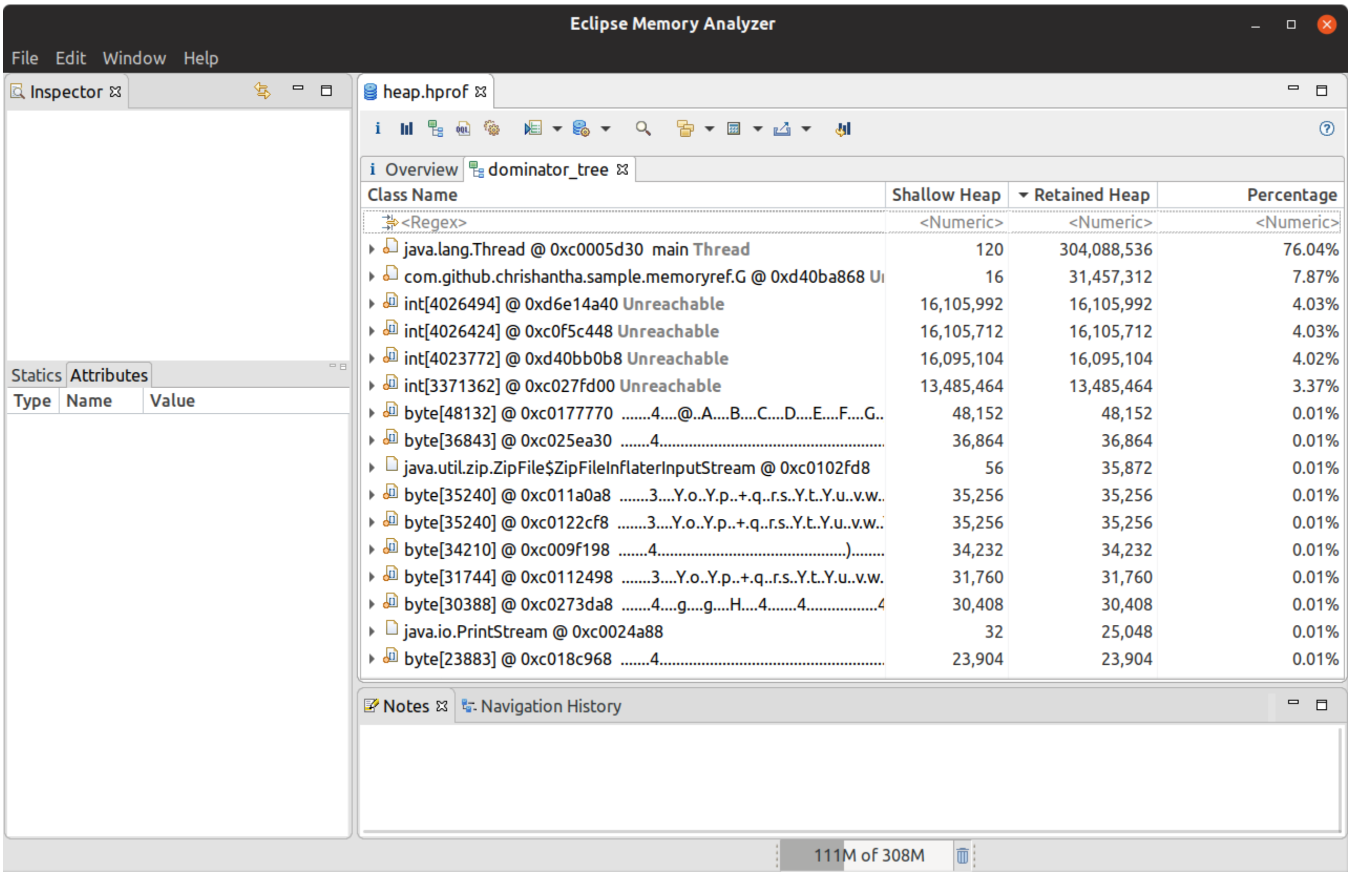
The Dominator Tree

The Dominator Tree is produced from the object graph in the heap dump. The “ Dominator Tree” can be viewed from the “Heap Dump Overview” page.

> *An object x **dominates** an object y if every path in the object graph from the start (or the root) node to y must go through x.*

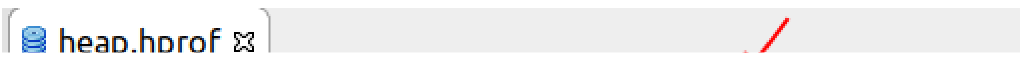
Above *informal* definition is taken from Eclipse MAT help page.

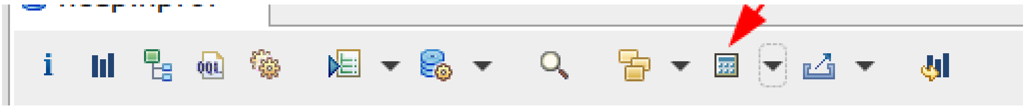
Since the tree is created based on object references, we can easily identify dependencies of objects.



Dominator Tree

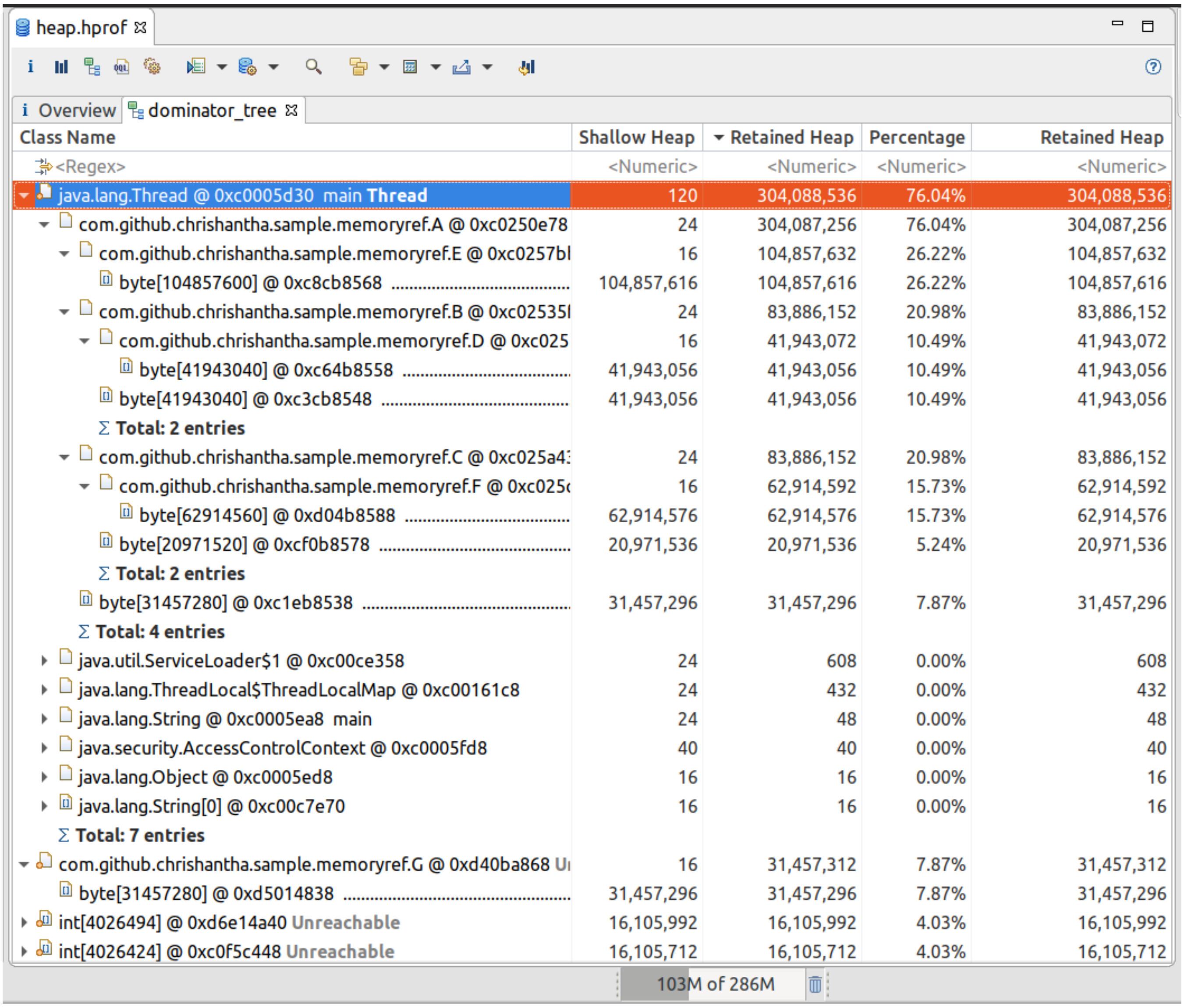
Here, we again see that main thread is the biggest object.





Calculating Retained Size

I calculated the precise retained heap sizes using the option shown in above screenshot.



heap.hprof

Overview | dominator_tree

Class Name	Shallow Heap	Retained Heap	Percentage	Retained Heap
<Regex>	<Numeric>	<Numeric>	<Numeric>	<Numeric>
java.lang.Thread @ 0xc0005d30 main **Thread**	120	304,088,536	76.04%	304,088,536
com.github.chrishantha.sample.memoryref.A @ 0xc0250e78	24	304,087,256	76.04%	304,087,256
com.github.chrishantha.sample.memoryref.E @ 0xc0257bl	16	104,857,632	26.22%	104,857,632
byte[104857600] @ 0xc8cb8568	104,857,616	104,857,616	26.22%	104,857,616
com.github.chrishantha.sample.memoryref.B @ 0xc02535l	24	83,886,152	20.98%	83,886,152
com.github.chrishantha.sample.memoryref.D @ 0xc025	16	41,943,072	10.49%	41,943,072
byte[41943040] @ 0xc64b8558	41,943,056	41,943,056	10.49%	41,943,056
byte[41943040] @ 0xc3cb8548	41,943,056	41,943,056	10.49%	41,943,056
Σ **Total: 2 entries**				
com.github.chrishantha.sample.memoryref.C @ 0xc025a4:	24	83,886,152	20.98%	83,886,152
com.github.chrishantha.sample.memoryref.F @ 0xc025(	16	62,914,592	15.73%	62,914,592
byte[62914560] @ 0xd04b8588	62,914,576	62,914,576	15.73%	62,914,576
byte[20971520] @ 0xcf0b8578	20,971,536	20,971,536	5.24%	20,971,536
Σ **Total: 2 entries**				
byte[31457280] @ 0xc1eb8538	31,457,296	31,457,296	7.87%	31,457,296
Σ **Total: 4 entries**				
java.util.ServiceLoader$1 @ 0xc00ce358	24	608	0.00%	608
java.lang.ThreadLocal$ThreadLocalMap @ 0xc00161c8	24	432	0.00%	432
java.lang.String @ 0xc0005ea8 main	24	48	0.00%	48
java.security.AccessControlContext @ 0xc0005fd8	40	40	0.00%	40
java.lang.Object @ 0xc0005ed8	16	16	0.00%	16
java.lang.String[0] @ 0xc00c7e70	16	16	0.00%	16
Σ **Total: 7 entries**				
com.github.chrishantha.sample.memoryref.G @ 0xd40ba868 Ui	16	31,457,312	7.87%	31,457,312
byte[31457280] @ 0xd5014838	31,457,296	31,457,296	7.87%	31,457,296
int[4026494] @ 0xd6e14a40 Unreachable	16,105,992	16,105,992	4.03%	16,105,992
int[4026424] @ 0xc0f5c448 Unreachable	16,105,712	16,105,712	4.03%	16,105,712

103M of 286M

Expanded Tree View

Here, the main thread dominates the object `a`. The object `a` dominates objects `b`, `c`, and `e`. Note that the **immediate dominator** of object `e` is `a`, since the object `e` is referenced in both `b` and `c` objects. This shows that the edges in the dominator tree do not directly correspond to object references from the object graph.

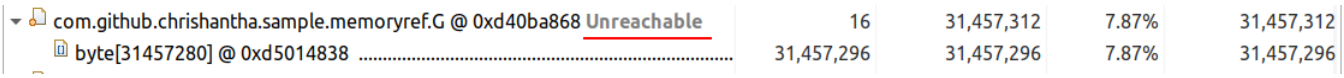
Following was the output of the application.

```
MemoryRefApplication
The retained heap size of object A is 304087256 bytes (~290 MiB).
```

```
The shallow heap size of object A is 24 bytes.
Press Enter to exit.
The size of object allocated within the background thread was
31457312 bytes (~30 MiB).
```

Here, we can see that the application has also calculated shallow and retained heap sizes precisely. I knew how the dominator tree would look like :) Basically, I didn't calculate the object size of `e` in class `C` since it was also shared in class `B`.

Dominator Tree also shows that the instance of class `G` is unreachable.



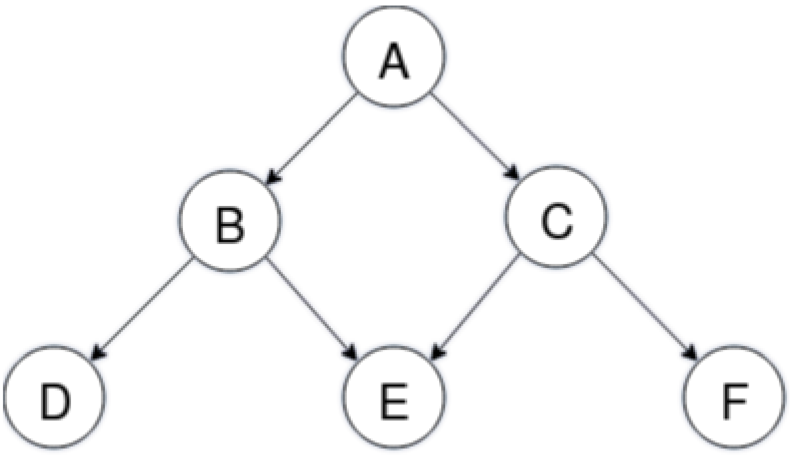
Unreachable object

Incoming and Outgoing References

In Eclipse MAT, we can see the incoming and outgoing references for a given object by right clicking an object and selecting *List Objects -> with outgoing references / with incoming references*

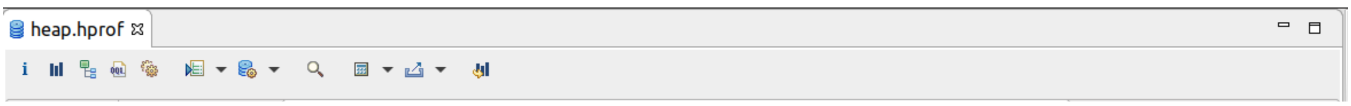
As explained earlier, the dominator tree is built out of the object graph, but the edges of the tree do not directly correspond to object references.

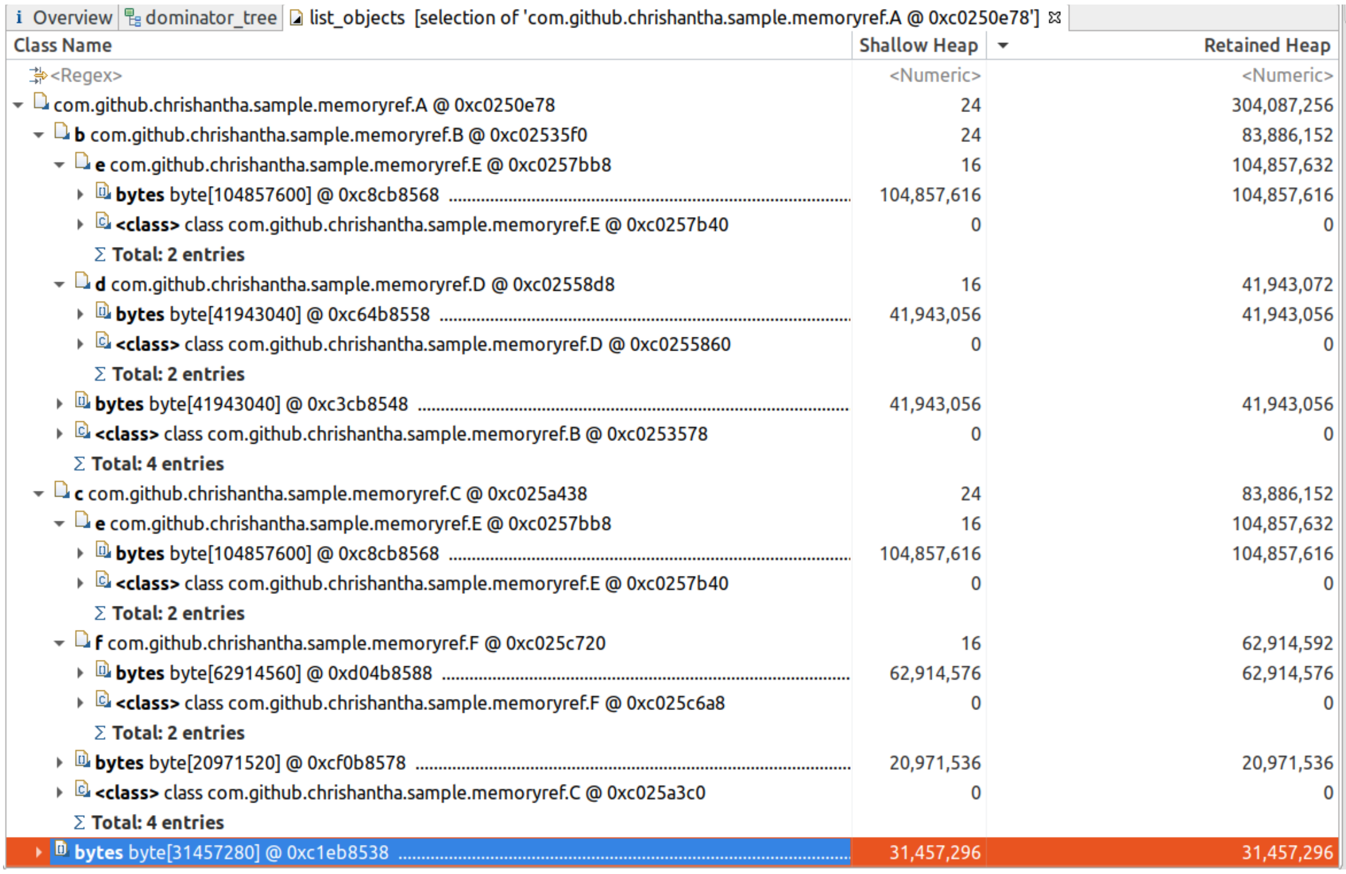
The object references shown here should be same as follows.



Object outgoing references

Let's look at incoming and outgoing references for some classes in sample application.

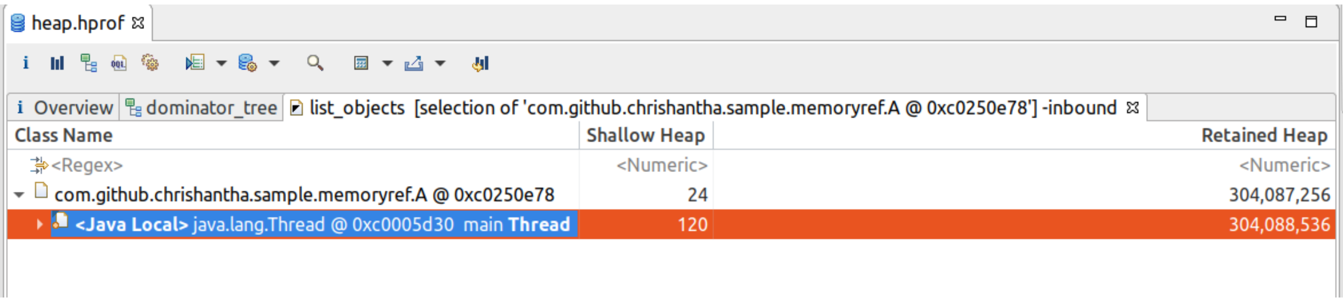




i Overview | dominator_tree | list_objects [selection of 'com.github.chrishantha.sample.memoryref.A @ 0xc0250e78']

Class Name	Shallow Heap	Retained Heap
<Regex>	<Numeric>	<Numeric>
com.github.chrishantha.sample.memoryref.A @ 0xc0250e78	24	304,087,256
b com.github.chrishantha.sample.memoryref.B @ 0xc02535f0	24	83,886,152
e com.github.chrishantha.sample.memoryref.E @ 0xc0257bb8	16	104,857,632
bytes byte[104857600] @ 0xc8cb8568	104,857,616	104,857,616
<class> class com.github.chrishantha.sample.memoryref.E @ 0xc0257b40	0	0
Σ **Total: 2 entries**		
d com.github.chrishantha.sample.memoryref.D @ 0xc02558d8	16	41,943,072
bytes byte[41943040] @ 0xc64b8558	41,943,056	41,943,056
<class> class com.github.chrishantha.sample.memoryref.D @ 0xc0255860	0	0
Σ **Total: 2 entries**		
bytes byte[41943040] @ 0xc3cb8548	41,943,056	41,943,056
<class> class com.github.chrishantha.sample.memoryref.B @ 0xc0253578	0	0
Σ **Total: 4 entries**		
c com.github.chrishantha.sample.memoryref.C @ 0xc025a438	24	83,886,152
e com.github.chrishantha.sample.memoryref.E @ 0xc0257bb8	16	104,857,632
bytes byte[104857600] @ 0xc8cb8568	104,857,616	104,857,616
<class> class com.github.chrishantha.sample.memoryref.E @ 0xc0257b40	0	0
Σ **Total: 2 entries**		
f com.github.chrishantha.sample.memoryref.F @ 0xc025c720	16	62,914,592
bytes byte[62914560] @ 0xd04b8588	62,914,576	62,914,576
<class> class com.github.chrishantha.sample.memoryref.F @ 0xc025c6a8	0	0
Σ **Total: 2 entries**		
bytes byte[20971520] @ 0xcf0b8578	20,971,536	20,971,536
<class> class com.github.chrishantha.sample.memoryref.C @ 0xc025a3c0	0	0
Σ **Total: 4 entries**		
bytes byte[31457280] @ 0xc1eb8538	31,457,296	31,457,296

Outgoing references for class "A"



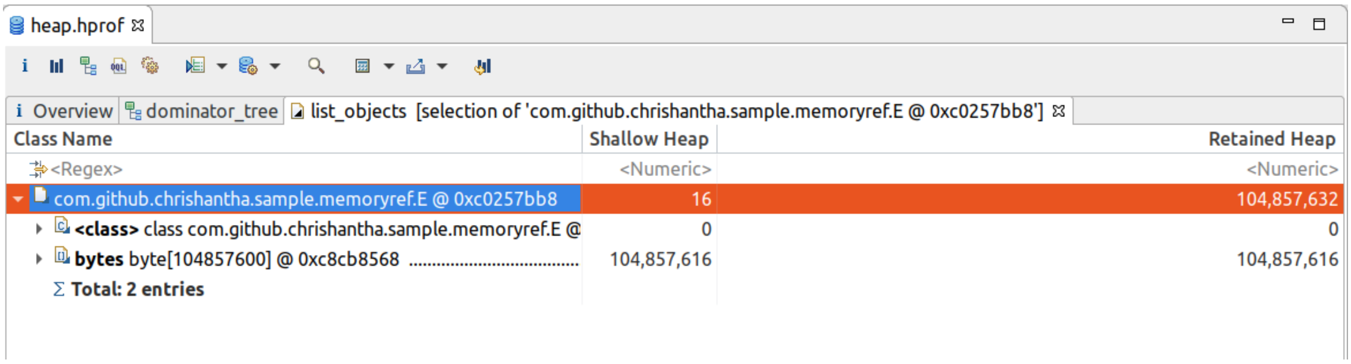
heap.hprof

i Overview | dominator_tree | list_objects [selection of 'com.github.chrishantha.sample.memoryref.A @ 0xc0250e78'] -inbound

Class Name	Shallow Heap	Retained Heap
<Regex>	<Numeric>	<Numeric>
com.github.chrishantha.sample.memoryref.A @ 0xc0250e78	24	304,087,256
<Java Local> java.lang.Thread @ 0xc0005d30 main **Thread**	120	304,088,536

Incoming references for class "A"

Earlier, we saw that object `a` dominates object `e`. Let's look at the references of object `e`.

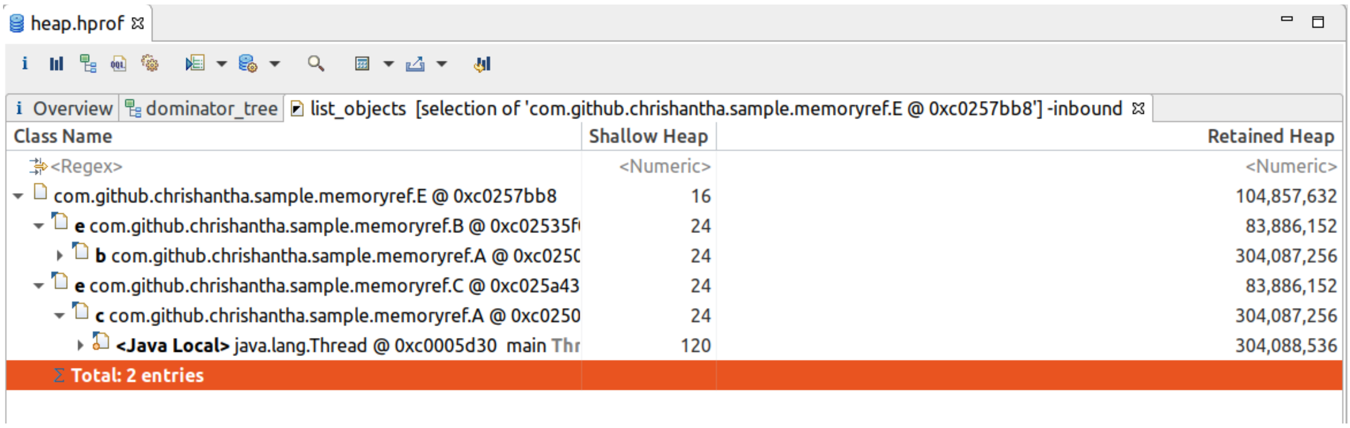


heap.hprof

i Overview | dominator_tree | list_objects [selection of 'com.github.chrishantha.sample.memoryref.E @ 0xc0257bb8']

Class Name	Shallow Heap	Retained Heap
<Regex>	<Numeric>	<Numeric>
com.github.chrishantha.sample.memoryref.E @ 0xc0257bb8	16	104,857,632
<class> class com.github.chrishantha.sample.memoryref.E @	0	0
bytes byte[104857600] @ 0xc8cb8568	104,857,616	104,857,616
Σ **Total: 2 entries**		

Outgoing references for class "E"



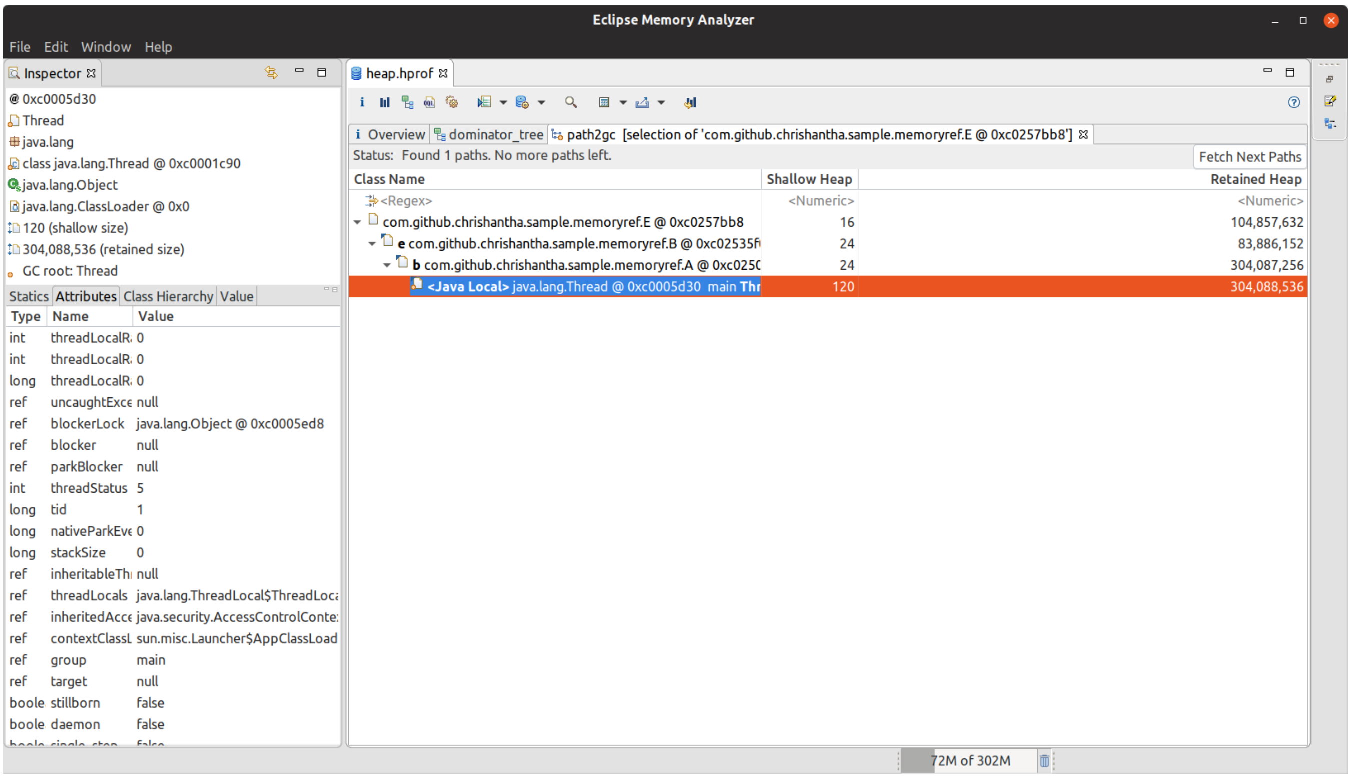
Incoming references for class "E"

Incoming references for class `E` are from classes `B` and `C` even though the immediate dominator of object `e` is object `a`.

Finding Paths to the GC Roots

Another useful feature in Eclipse MAT is the ability to find paths to the GC roots.

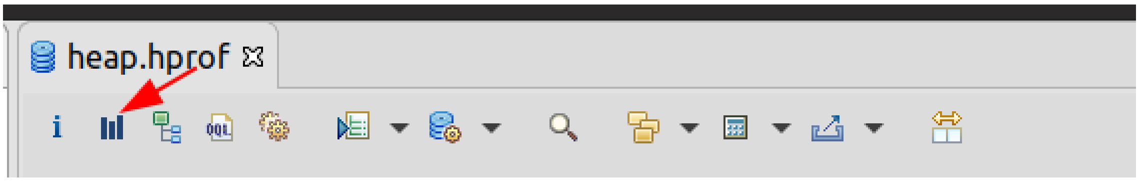
Following is the path to GC roots from class `E`. This view was opened by right clicking the class E in dominator tree and selecting *Path To GC Roots -> with all references*



We can see that the GC root of the instance of class `E` is the main thread.

The Histogram

Histogram shows the number of instances per class. It’s by default in descending order of retained size. You can open the Histogram from overview pane or by clicking the icon shown below.



Creating a histogram

Histogram is useful to find the retained heap size by class and find out how many instances per class.

In the sample application, there is only one object for classes `A` to `G`.

Histogram

Conclusion

In this story, I explained several basic concepts of heap dump analysis with Eclipse Memory Analyzer using a sample application I developed. I covered some basics of generating heap dumps, reachability, GC roots, shallow vs. retained heap, and dominator tree.

There are many more things that I haven't covered. For example, the Object Query Language (OQL). The OQL is an SQL-like language. When comparing with SQL, we can consider classes as tables, objects as rows and fields as columns. I didn't use OQL with the sample application, but there are many cases that OQL will be very useful. Eclipse MAT's help is the best place to start learning OQL.

Heapdump Java Eclipse Garbage Collection Heap