COM1032 Operating Systems
Lab 7
Memory Management

Purpose:

The purpose of this lab session is to familiarise yourself with the Operating System Memory Management processes, and investigation methods. You will run Memory Profiler and generate, save, and inspect data.

Aim

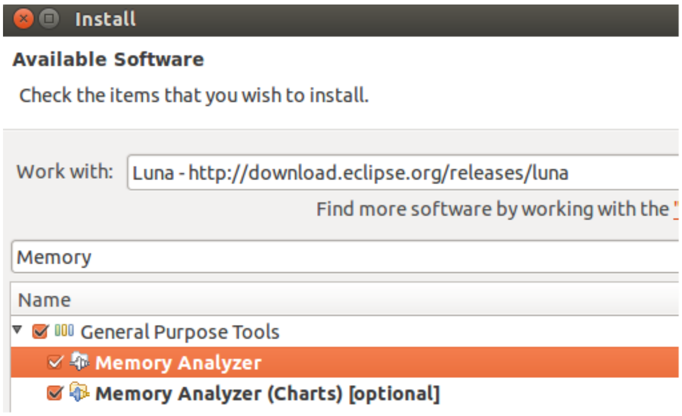
By the end of the lab you will be able to:

- Use Memory Profiler to collect data about your application.
- View Memory Profiler reports.
- Dump the Java heap and inspect it.

Eclipse Memory Analyser Tool (MAT):

Task 1: Install the tool

The Eclipse Memory Analyser tool (MAT) is a fast and feature-rich heap dump analyser that helps you find memory leaks and analyse high memory consumption issues. Install Eclipse MAT via the Help Menu → Install New Software… menu entry. Select the update site of your release from the drop-down box and once its content is downloaded, select General Purpose Tools and its sub-entries Memory Analyser and Memory Analyser (Charts). In the opened dialog, in the "Work with" section on top, click to Add, and add the Luna release using the repository: http://download.eclipse.org/releases/luna. Click Ok to go back to the first dialog and wait for existing software to refresh. In the "General Purpose Tools" choose the two packages shown in the figure below.



Task 2: Create an Example Application

Create the Java project and create the following class:

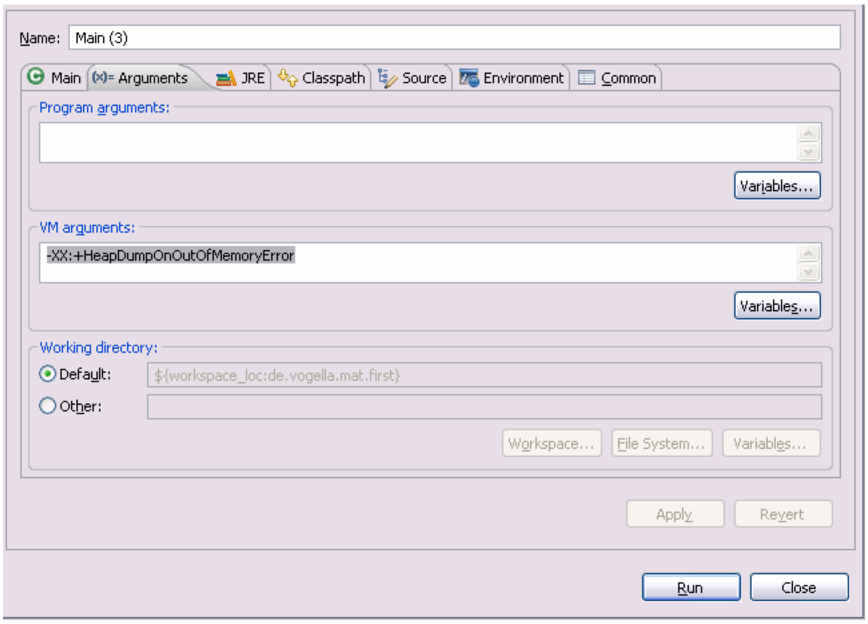
```
import java.util.ArrayList;
import java.util.List;
public class Main {
    /**
     * @param args
     */
    public static void main(String[] args) {
        List<String> list = new ArrayList<String>();
        while (true)
            list.add("OutOfMemoryError soon");
    }
}
```

Task 3. Acquire a Heap Dump

A *heap dump* is a snapshot of the complete Java object graph on a Java application at a certain point in time. It is stored in a binary format called HPROF (Heap Memory Profiling). It includes all objects, fields, primitive types and object references.

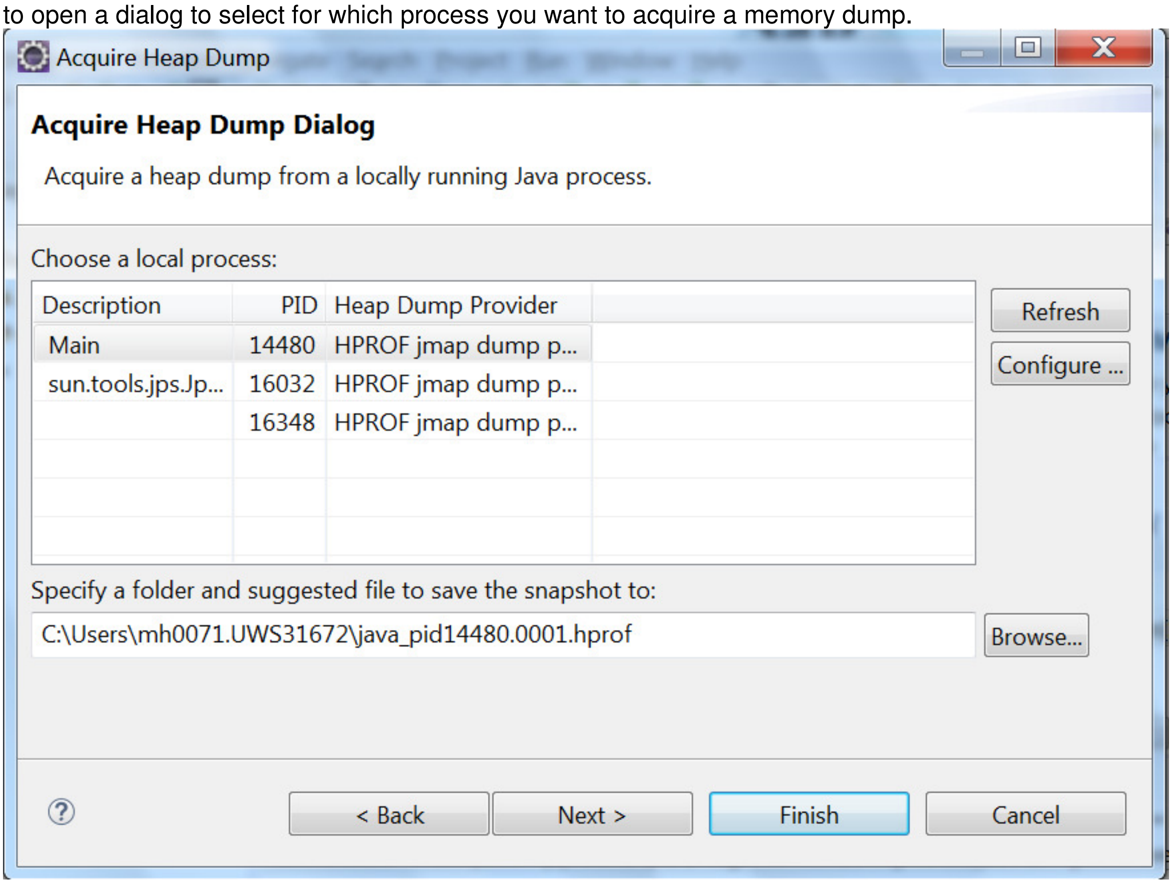
Get Heap Dump on an OutOfMemoryError

It is possible to instruct the JVM to create automatically a heap dump in case that it runs out of memory, i.e. in case of a OutOfMemoryError error. To instruct the JVM to create a heap dump in such a situation, start your Java application with the -XX:+HeapDumpOnOutOfMemoryError option by adding the following line to the arguments of your run or debug configuration of your application:



The heap dump is written to the work directory.

To attach to a running process: use the File Menu ⟶ New ⟶ Other ⟶ Other ⟶ Heap Dump menu entry to open a dialog to select for which process you want to acquire a memory dump.

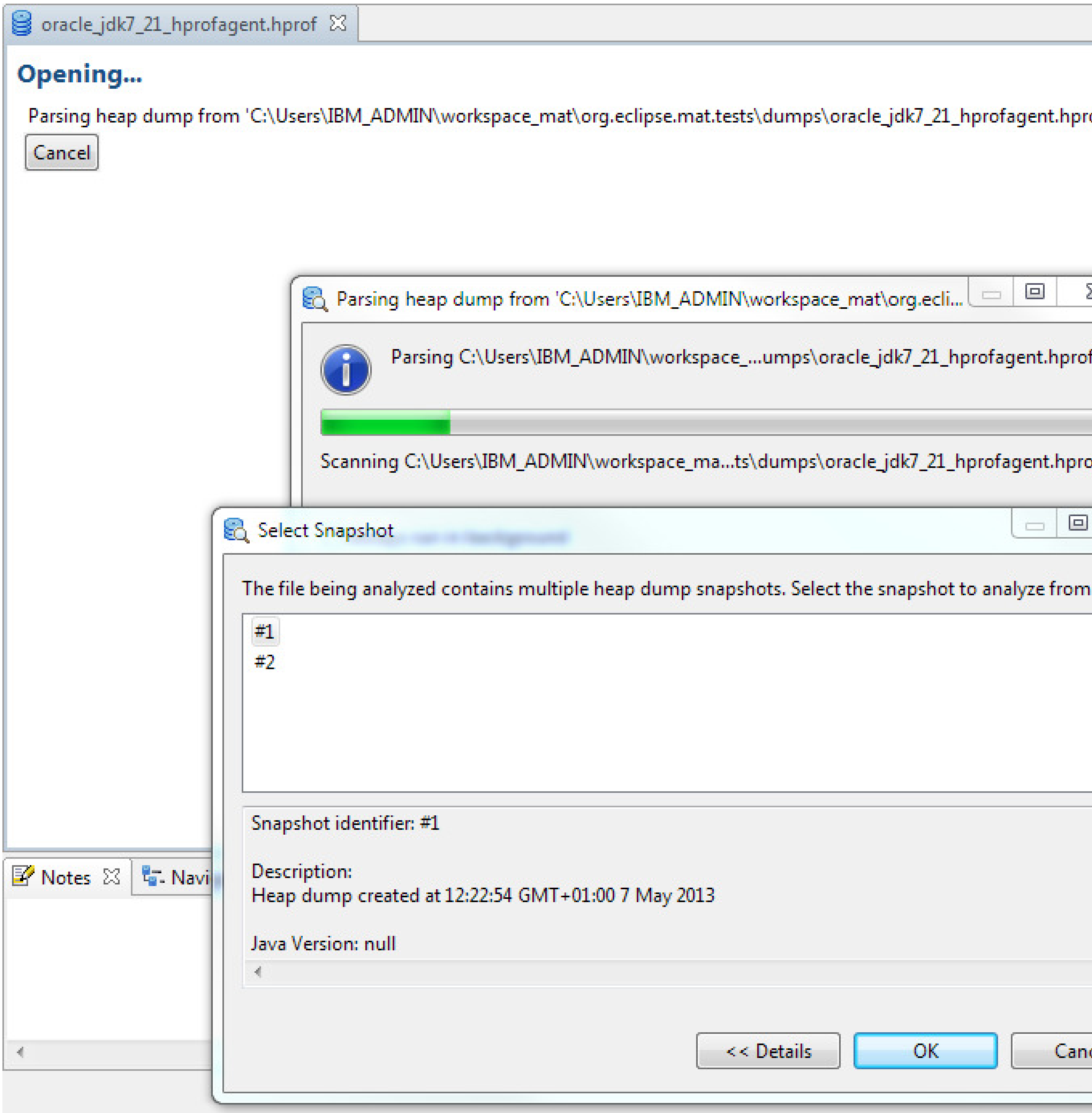


Depending on the concrete execution environment the pre-installed heap dump providers may work with their default settings and in this case a list of running Java processes should appear: To make selection easier, the order of the Java processes can be altered by clicking on the column titles for **pid** or **Heap Dump Provider**.

One can now select from which process a heap dump should be acquired, provide a preferred location for the heap dump and press **Finish** to acquire the dump. Some of the heap dump providers may allow (or require) additional parameters (e.g. type of the heap dump) to be set. This can be done by using **Next** button to get to the Configuring Heap Dump Provider Arguments page of the wizard. Check the online documentation: https://help.eclipse.org/2019-12/index.jsp?topic=%2Forg.eclipse.mat.ui.help%2Ftasks%2Facquiringheapdump.html.

Run the project you created in Task 2. It crashes and writes a heap dump.

Open the heap dump in MAT and get familiar with using the MAT tooling.

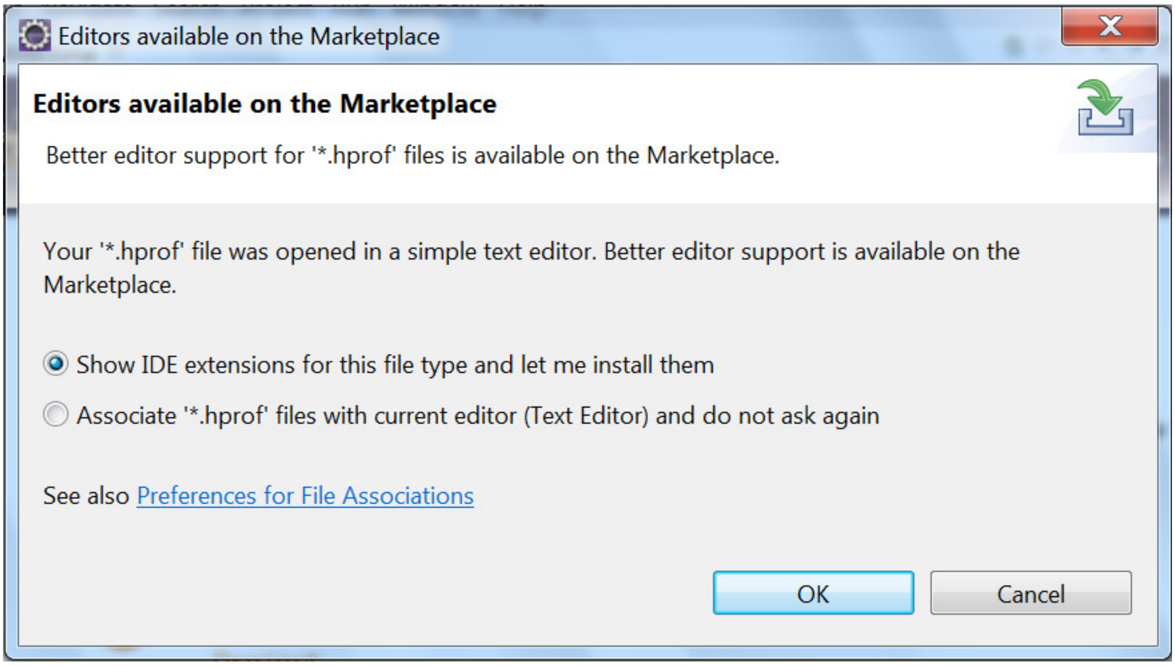


The index files generated have a component in the file name from the snapshot identifier, so the index files from each snapshot can be distinguished. This means that multiple snapshots from one heap dump file can be examined in Memory Analyzer simultaneously. The heap dump history for the file remembers the last snapshot selected for that file, though when the snapshot is reopened via the history the index file is also shown in the history. To open another snapshot in the dump, close the first snapshot, then reopen the heap dump file using the File menu and another snapshot can be chosen to be parsed. The first snapshot can then be reopened using the index file in the history, and both snapshots can be viewed at once.

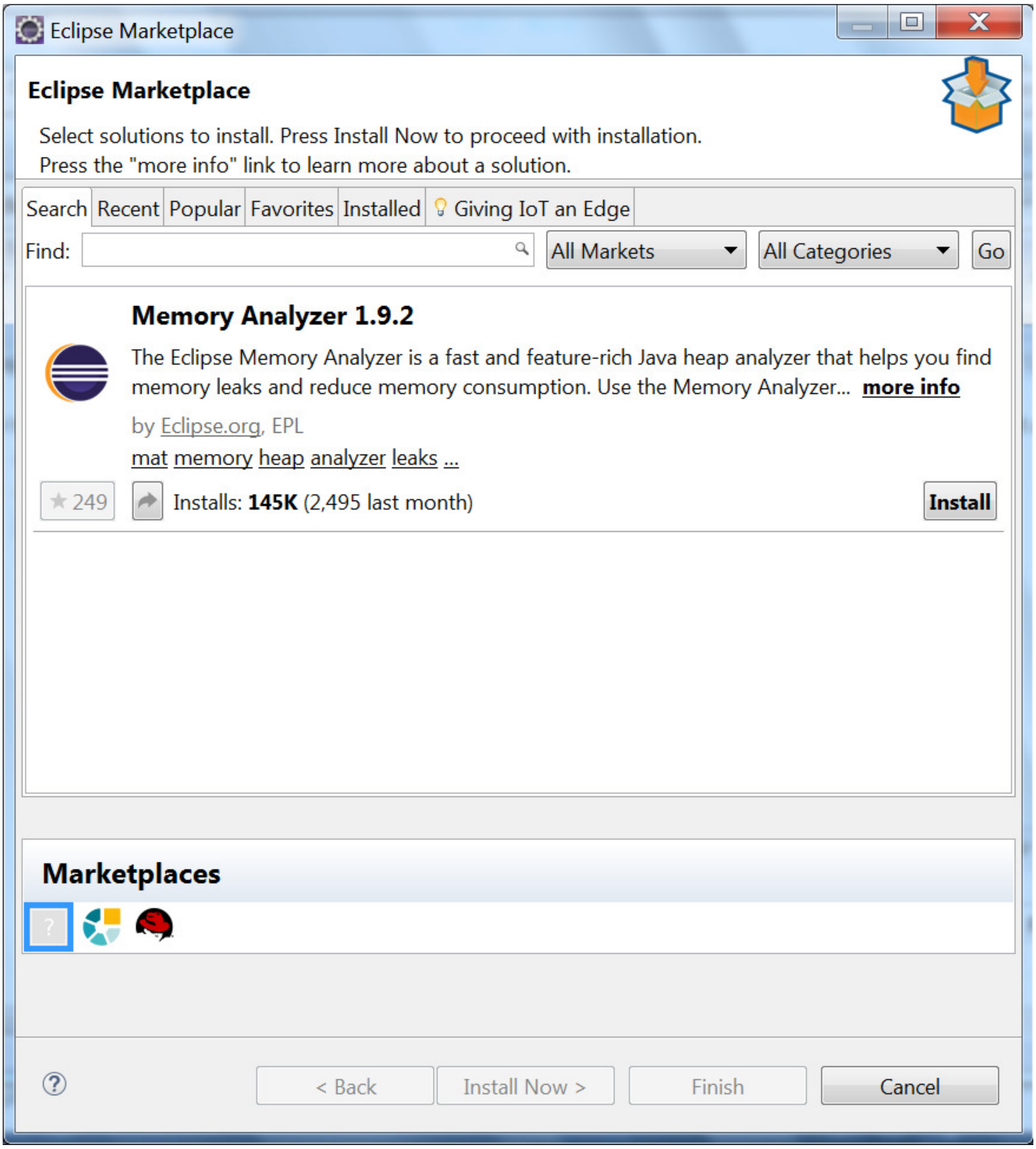
Task 4: Reviewing a heap dump

After a new heap dump with the .hprof ending has been created, you can open it via a double-click in Eclipse. If you used MAT to create the heap dump, it should be opened automatically.

You may need to refresh your project (F5 on the project). Double-click the file and select the Leak Suspects Report. You might see the following screen:



Click OK to install MAT if Task 1 was unsuccessful, or needs update:

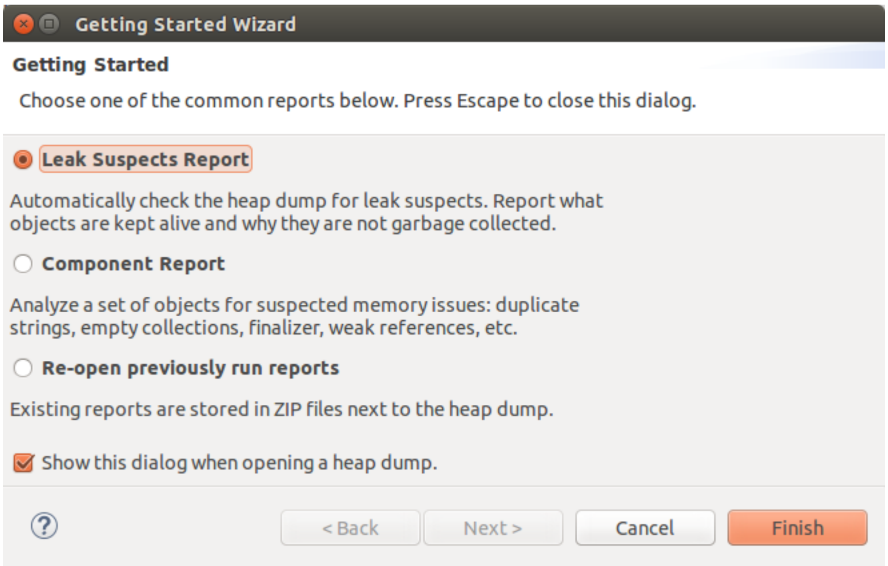


Then restart Eclipse. Double click the HPROF file again. You might get an Eclipse Internal Error because the heap size is much smaller than 2.5 GB dump file you are trying to open. You solve this by editing your eclipse.ini file to increase the vm argument to -Xmx1024m or higher. In Windows: 1) Set higher xmx here : Control Panel > Java > Java tab > View.. > Runtime parameters. [20gb just for the load]

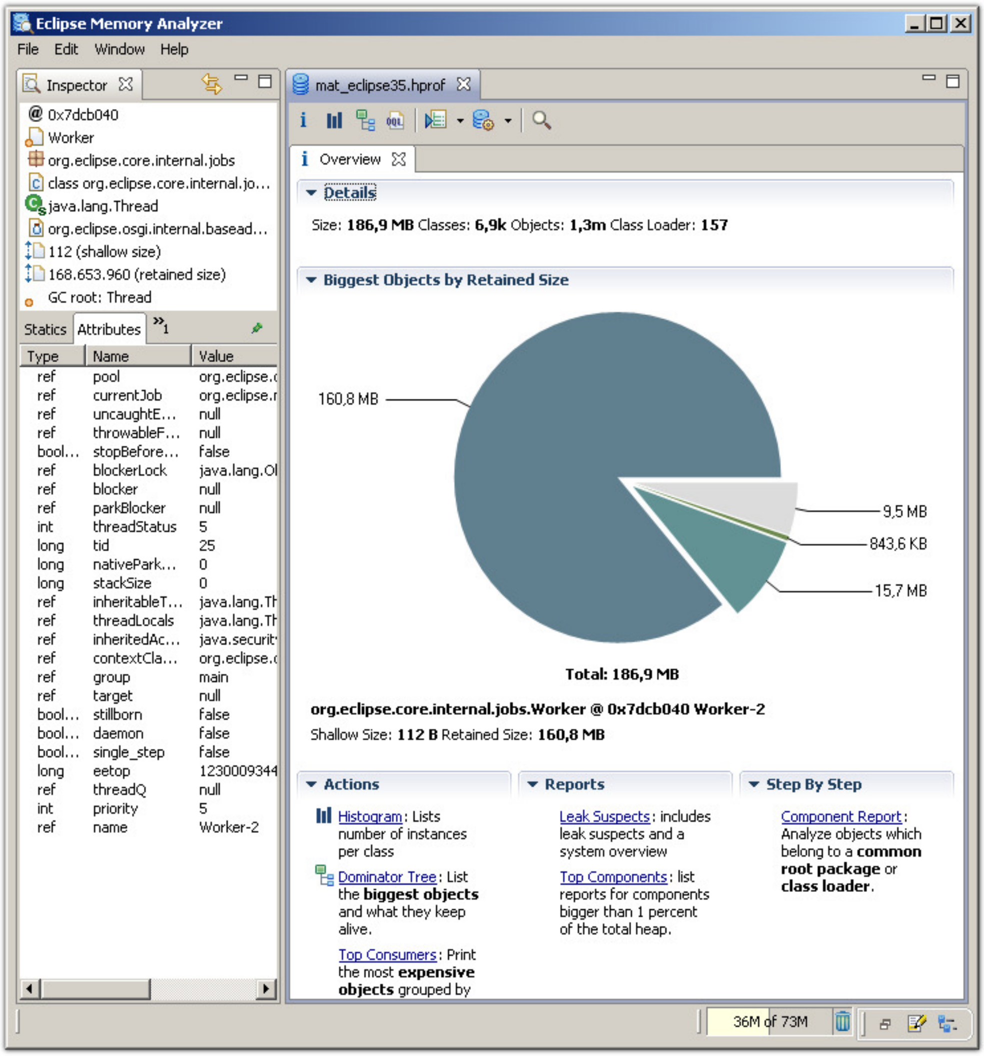
1. open the Eclipse.ini in the folder where Eclipse executable is located.
2. change the default -Xmx1024m to a larger size, -Xmx20g worked for me.

Note that on OS X, to increase the memory allocated to MAT, you need to right-click Memory Analyzer.app or Eclipse.app and show the package contents. The MemoryAnalyzer.ini or Eclipse.ini file is under /Contents/MacOS/.

Now you can choose from the following options:



The overview page allows you to start the analysis of the heap dump. The dominator tree gives quickly an overview of the used objects.



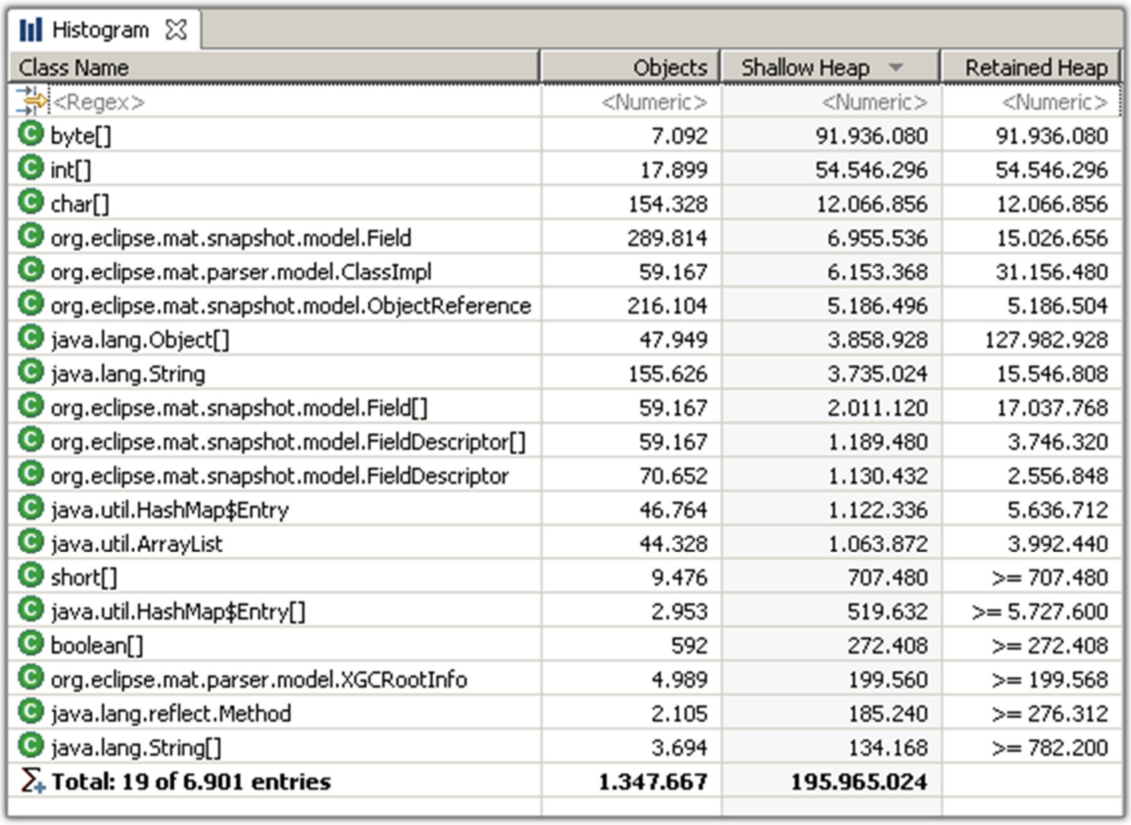
On the right, you'll find the size of the dump and the number of classes, objects and class loaders.

If the total size of the dump is much smaller than the size of the file it is possible that the heap dump contained many 'garbage' objects which would be discarded at the next garbage collection. See the unreachable objects query to examine these 'garbage' objects.

Right below, the pie chart gives an impression on the biggest objects in the dump. Move your mouse over a slice to see the details of the objects in the object inspector on the left. Click on any slice to drill down and follow for example the outgoing references.

Step 3 - The Histogram

Select the *histogram* from the tool bar to list the number of instances per class, the shallow size and the retained size .

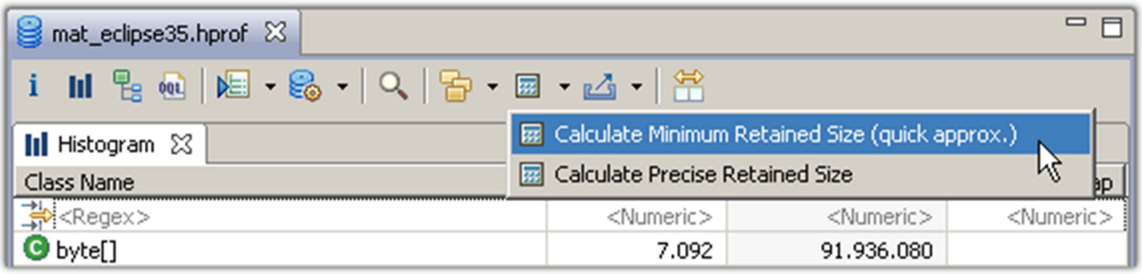


Histogram

Class Name	Objects	Shallow Heap	Retained Heap
<Regex>	<Numeric>	<Numeric>	<Numeric>
byte[]	7.092	91.936.080	91.936.080
int[]	17.899	54.546.296	54.546.296
char[]	154.328	12.066.856	12.066.856
org.eclipse.mat.snapshot.model.Field	289.814	6.955.536	15.026.656
org.eclipse.mat.parser.model.ClassImpl	59.167	6.153.368	31.156.480
org.eclipse.mat.snapshot.model.ObjectReference	216.104	5.186.496	5.186.504
java.lang.Object[]	47.949	3.858.928	127.982.928
java.lang.String	155.626	3.735.024	15.546.808
org.eclipse.mat.snapshot.model.Field[]	59.167	2.011.120	17.037.768
org.eclipse.mat.snapshot.model.FieldDescriptor[]	59.167	1.189.480	3.746.320
org.eclipse.mat.snapshot.model.FieldDescriptor	70.652	1.130.432	2.556.848
java.util.HashMap$Entry	46.764	1.122.336	5.636.712
java.util.ArrayList	44.328	1.063.872	3.992.440
short[]	9.476	707.480	>= 707.480
java.util.HashMap$Entry[]	2.953	519.632	>= 5.727.600
boolean[]	592	272.408	>= 272.408
org.eclipse.mat.parser.model.XGCRootInfo	4.989	199.560	>= 199.568
java.lang.reflect.Method	2.105	185.240	>= 276.312
java.lang.String[]	3.694	134.168	>= 782.200
Total: 19 of 6.901 entries	**1.347.667**	**195.965.024**	

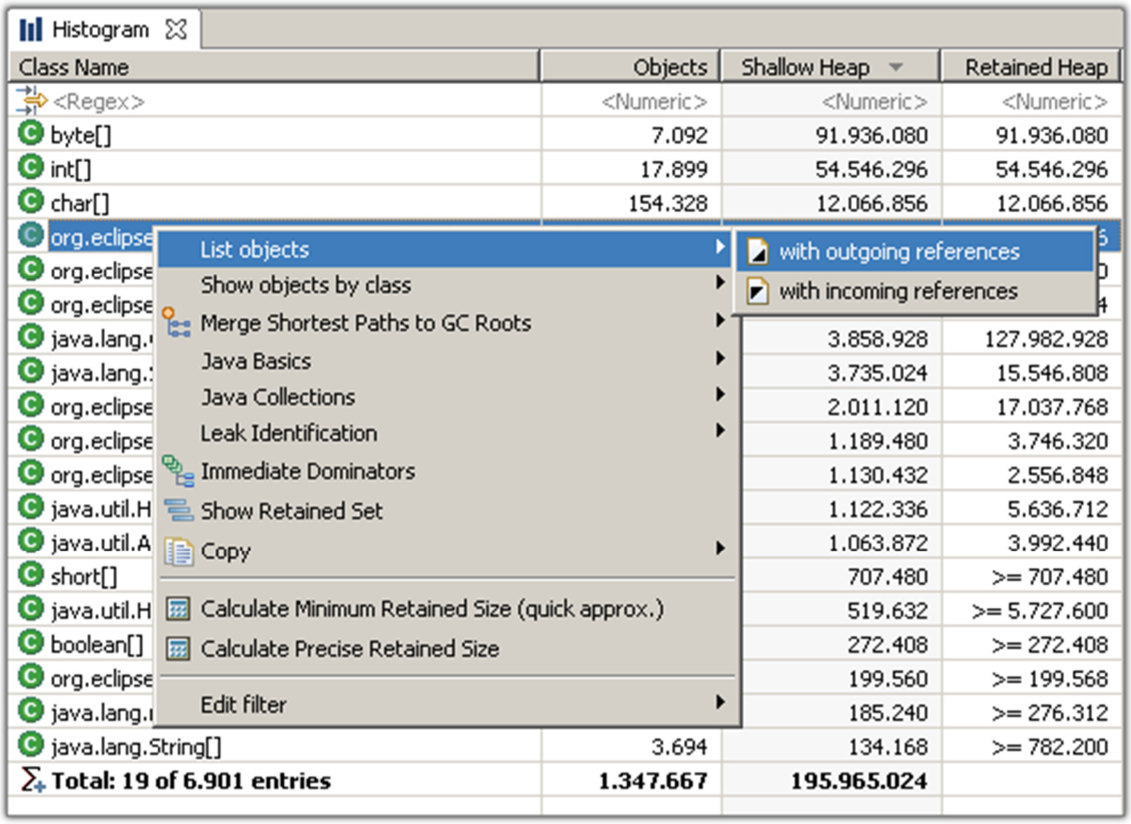
The Memory Analyzer displays by default the retained size of individual objects. However, the retained size of a set of objects - in this case all instances of a particular class - needs to be calculated.

To approximate the retained sizes for all rows, pick icon from the tool bar. Alternatively, select a couple rows and use the context menu.

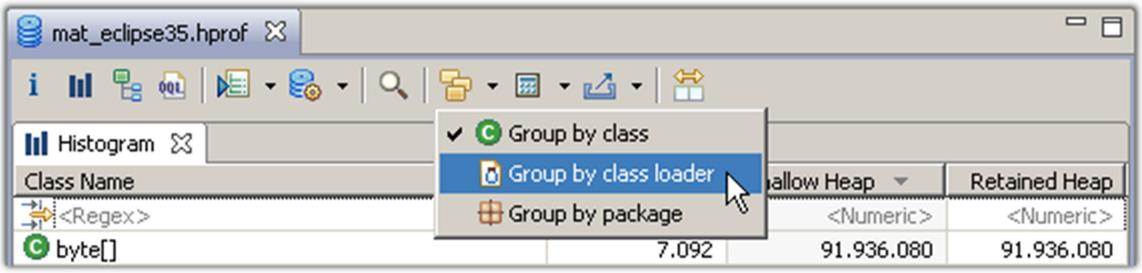


Using the **context menu** , you can drill-down into the set of objects which the selected row represents. For example, you can list the objects with outgoing or incoming references. Or group the objects by the value of an attribute. Or group the collections by their size. Or or or...

One thing that makes the Memory Analyzer so powerful is the fact that one can run any action on any set of objects. Just drill down and slice your objects the way you need them.



Another important feature is the facility to **group any histogram by class loader, packages or superclass** .



Any decent application loads different components by different class loaders. The Memory Analyzer attaches a meaningful label to the class loader - in the case of OSGi bundles it is the bundle id. Therefore it becomes a lot easier to divide the heap dump into smaller parts.

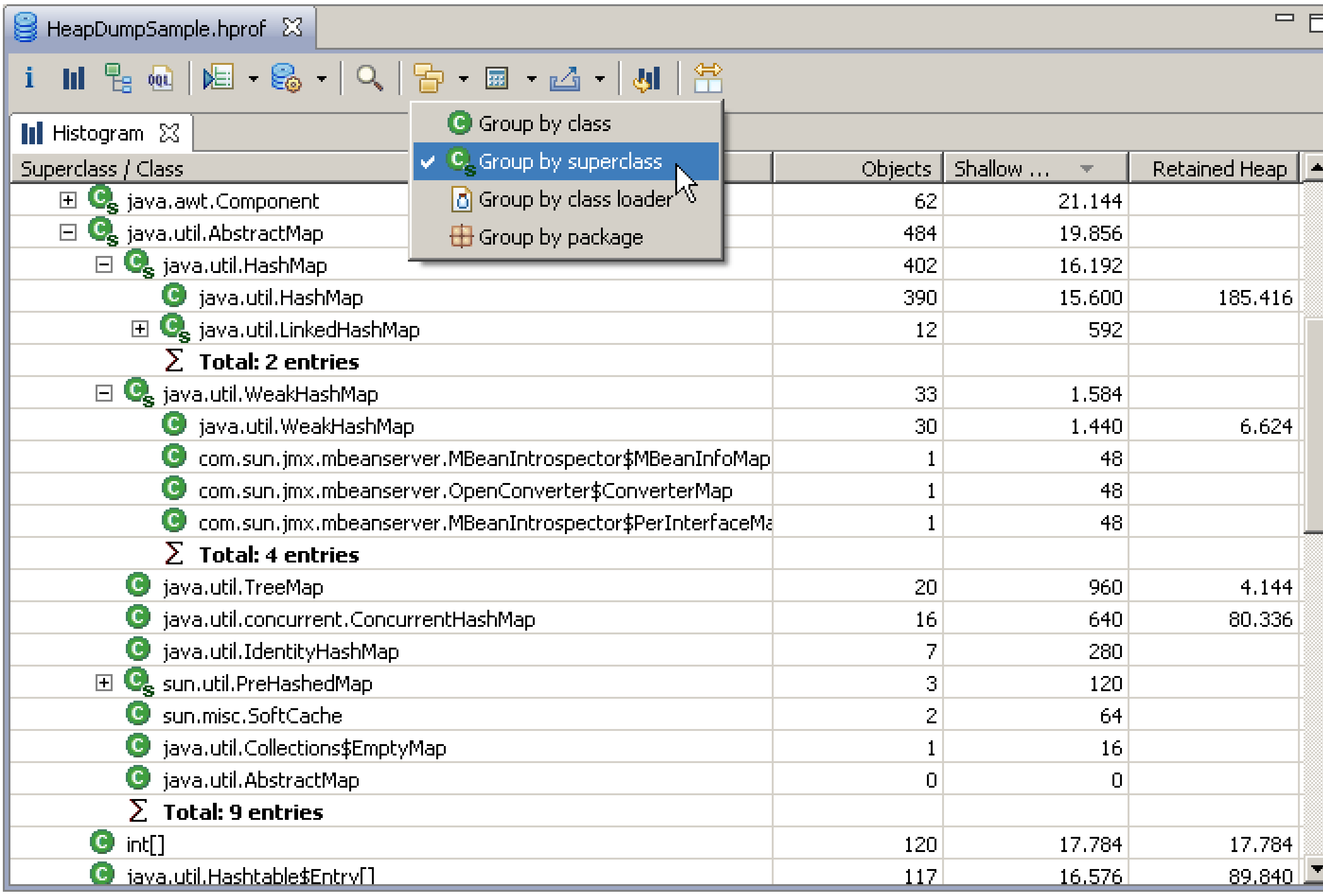
More: Analyze Class Loader

Class Loader / Class	Objects	Shallow Heap	Retained Heap
<Regex>	<Numeric>	<Numeric>	<Numeric>
<system class loader>	540.757	171.403.096	>= 172.638.432
org.eclipse.mat.api	695.484	16.486.560	>= 20.083.560
org.eclipse.mat.parser	70.526	6.498.488	>= 143.406.608
org.eclipse.emf.ecore	12.118	526.024	>= 642.560
Equinox Startup Class Loader	11.580	402.744	>= 3.028.216
org.eclipse.equinox.registry	6.080	277.632	>= 900.328
org.eclipse.swt	2.215	85.040	>= 156.512
org.eclipse.ui.workbench	1.842	65.712	>= 438.016
org.eclipse.jface	1.974	49.808	>= 212.224
org.eclipse.mat.report	1.059	36.128	>= 94.749.136
org.eclipse.core.commands	620	27.232	>= 62.832
org.eclipse.emf.common	782	27.120	>= 112.104
com.ibm.icu	506	17.936	>= 1.242.992
org.eclipse.equinox.common	801	17.880	>= 51.928
org.eclipse.birt.chart.engine	355	13.000	>= 158.152
org.eclipse.core.resources	188	6.752	>= 73.912
org.eclipse.update.configurator	130	4.336	>= 112.040
org.eclipse.core.expressions	151	4.128	>= 19.048
org.eclipse.mat.ui	88	2.848	>= 44.672
Total: 19 of 151 entries	**1.347.667**	**195.965.024**	

Grouping the histogram by packages allows to drill-down along the Java package hierarchy.

Package / Class	Objects	Shallow ...	Retained Heap
<Regex>	<Numeric>	<Numeric>	<Numeric>
byte[]	7.092	91.936.080	91.936.080
int[]	17.899	54.546.296	54.546.296
org	806.042	24.539.016	>= 174.053.832
eclipse	804.832	24.507.752	>= 174.024.568
osgi	1.202	31.120	>= 86.536
framework	1.067	25.600	>= 37.472
util	110	4.848	>= 43.488
service	25	672	>= 5.576
Total: 3 entries			
mozilla	7	128	>= 136
w3c	1	16	16
omg	0	0	0
xml	0	0	0
Total: 6 entries			
char[]	154.328	12.066.856	12.066.856
java	347.242	11.738.112	>= 115.757.992
lang	235.103	8.461.248	>= 112.800.984
Object[]	47.949	3.858.928	127.982.928
String	155.626	3.735.024	15.546.808
reflect	4.628	353.904	>= 582.568

Grouping the histogram by superclass provides an easy way to find for example all the subclasses of java.util.AbstractMap, etc...



Superclass / Class	Objects	Shallow ...	Retained Heap
java.awt.Component	62	21.144	
java.util.AbstractMap	484	19.856	
java.util.HashMap	402	16.192	
java.util.HashMap	390	15.600	185.416
java.util.LinkedHashMap	12	592	
Total: 2 entries			
java.util.WeakHashMap	33	1.584	
java.util.WeakHashMap	30	1.440	6.624
com.sun.jmx.mbeanserver.MBeanIntrospector$MBeanInfoMap	1	48	
com.sun.jmx.mbeanserver.OpenConverter$ConverterMap	1	48	
com.sun.jmx.mbeanserver.MBeanIntrospector$PerInterfaceMa	1	48	
Total: 4 entries			
java.util.TreeMap	20	960	4.144
java.util.concurrent.ConcurrentHashMap	16	640	80.336
java.util.IdentityHashMap	7	280	
sun.util.PreHashedMap	3	120	
sun.misc.SoftCache	2	64	
java.util.Collections$EmptyMap	1	16	
java.util.AbstractMap	0	0	
Total: 9 entries			
int[]	120	17.784	17.784
java.util.Hashtable$Entry[]	117	16.576	89.840

Step 4 - The Dominator Tree

The dominator tree displays the biggest objects in the heap dump. The next level of the tree lists those objects that would be garbage collected if all incoming references to the parent node were removed.

The dominator tree is a powerful tool to investigate which objects keep which other objects alive. Again, the tree can be grouped by class loader (e.g. components) and packages to ease the analysis.

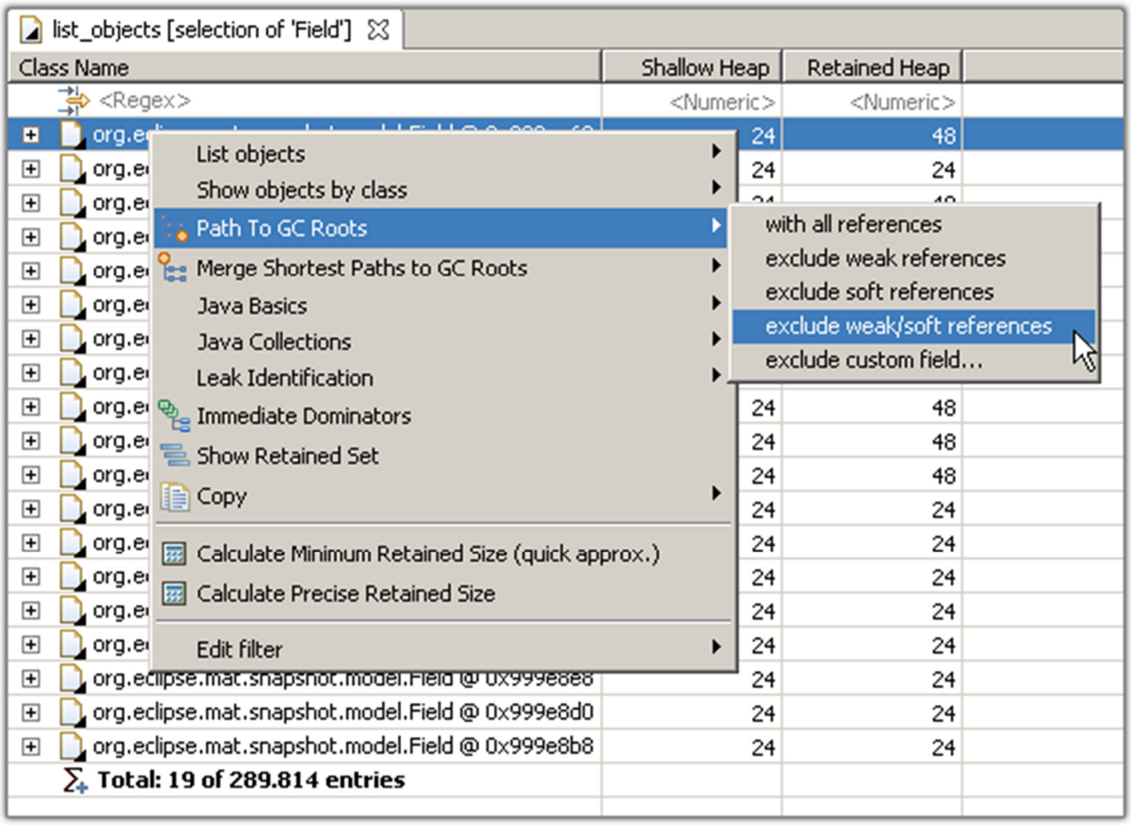
dominator_tree

Class Name	Shallow Heap	Retained...	Percentage
<Regex>	<Numeric>	<Numeric>	<Numeric>
org.eclipse.core.internal.jobs.Worker @ 0x7dcb040 W	112	168.653.960	86,06%
int[4112974] @ 0x12070a20 **Unknown**	16.451.912	16.451.912	8,40%
org.eclipse.core.internal.registry.RegistryObjectManag	64	863.880	0,44%
org.eclipse.osgi.internal.baseadaptor.DefaultClassLoad	72	735.408	0,38%
sun.net.www.protocol.jar.URLJarFile @ 0x7924978	64	344.856	0,18%
class com.ibm.icu.text.RuleBasedCollator @ 0x43e3c30	256	258.072	0,13%
org.eclipse.mat.query.registry.QueryRegistry @ 0x7e0	32	180.320	0,09%
org.eclipse.osgi.internal.resolver.SystemState @ 0x79	80	171.216	0,09%
org.eclipse.osgi.internal.baseadaptor.DefaultClassLoad	72	157.432	0,08%
org.eclipse.osgi.internal.baseadaptor.DefaultClassLoad	72	130.520	0,07%
org.eclipse.osgi.internal.baseadaptor.DefaultClassLoad	72	112.264	0,06%
java.util.jar.JarFile @ 0x78e9d40	48	106.416	0,05%
sun.util.resources.TimeZoneNames @ 0x9cfdc80	40	100.816	0,05%
class com.ibm.icu.text.UnicodeSet @ 0x43e9e018	64	97.480	0,05%
class java.io.ObjectStreamClass$Caches @ 0x43f721c0	16	81.808	0,04%
org.eclipse.osgi.internal.baseadaptor.DefaultClassLoad	72	66.056	0,03%
class com.sun.org.apache.xerces.internal.util.XMLChar	40	65.592	0,03%
class org.eclipse.update.internal.configurator.SiteEntry	16	56.664	0,03%
org.eclipse.emf.ecore.xml.type.impl.XMLTypePackageIr	336	55.160	0,03%
Total: 19 of 44.690 entries			

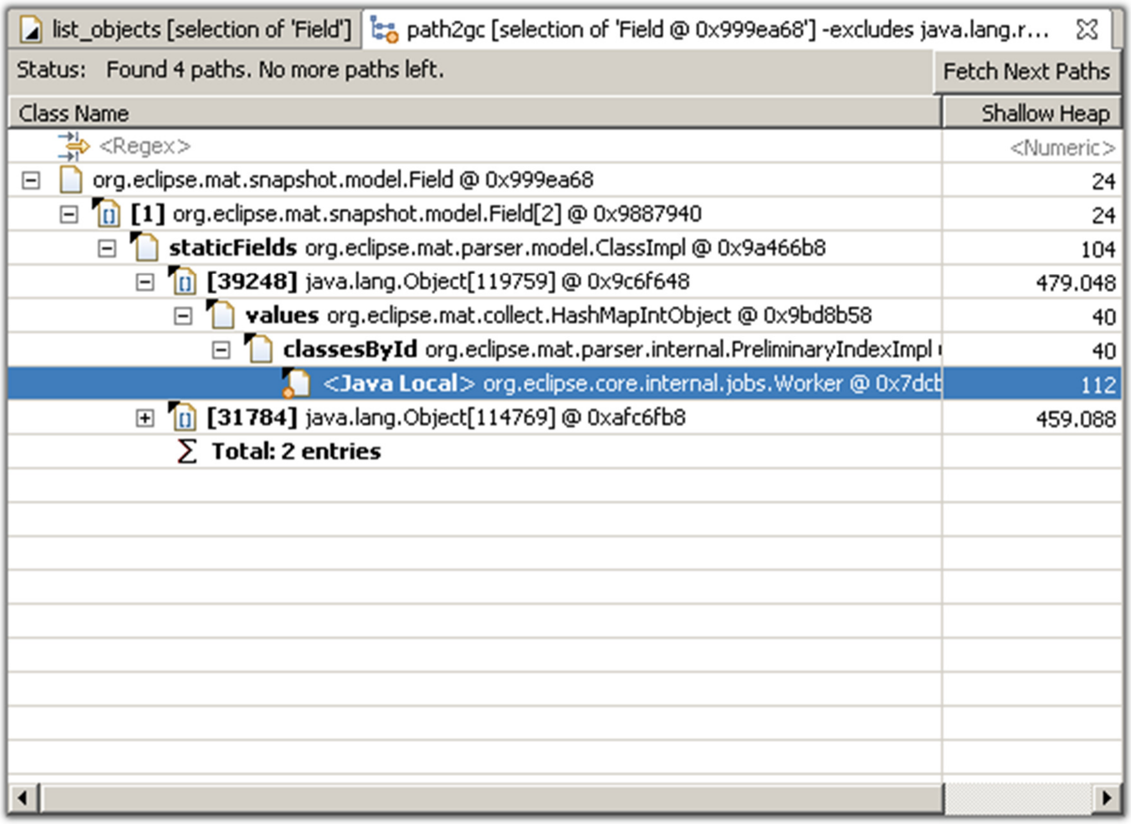
Step 5 - Path to GC Roots

Garbage Collections Roots (GC roots) are objects that are kept alive by the Virtual Machines itself. These include for example the thread objects of the threads currently running, objects currently on the call stack and classes loaded by the system class loader.

The (reverse) reference chain from an object to a GC root - the so called path to GC roots - explains why the object cannot be garbage collected. The path helps solving the classical memory leak in Java: those leaks exist because an object is still referenced even though the program logic will not access the object anymore.

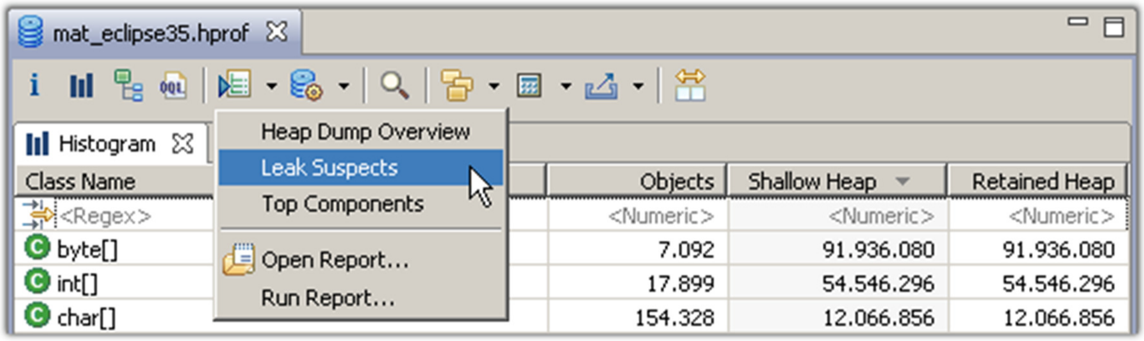


Initially, the GC root reached by the shortest path is selected.



Step 6 - The Leak Report

The Memory Analyzer can inspect the heap dump for leak suspects, e.g. objects or set of objects which are suspiciously big.



Learn more in this blog posting: Automated Heap Dump Analysis: Finding Memory Leaks with One Click http://memoryanalyzer.blogspot.com/2008/05/automated-heap-dump-analysis-finding.html .

Using Memory Analyzer to analyse problems

Memory Analyzer can diagnose OutOfMemoryErrors by looking for areas of the application that are either leaking memory or have a footprint requirement that's too large for the available memory. Memory Analyzer does automatic leak detection and generates a Leak Suspects report.

The additional data that's available in the HPROF and IBM system dumps, particularly the field names and field values — along with the capabilities of the Inspector view and **Object Query Language (OQL)** — also make it possible to diagnose a wider range of problem types than "What's using all of the memory?". For example, you can ascertain the occupancy and load factor of collections to see if they are efficiently sized, or look at the hostname and port associated with a ConnectException to see what connection the application was trying to create. For more information on OQL, please check https://help.eclipse.org/2020-03/index.jsp?topic=%2Forg.eclipse.mat.ui.help%2Freference%2Foqlsyntax.html.

Summary

- Use Memory Profiler to observe how your app uses memory over time. Look for patterns that indicate memory leaks.
- Use Java heap dumps to identify which classes allocate large amounts of memory.
- Record allocations over time to observe how apps allocate memory and where in your code the allocation is happening.

Final Coursework Hints:

In your final Coursework, you will need incremental development of subsystems in a modular way, and get them to interface together.

https://blogs.oracle.com/java/introduction-to-modular-development

As you finish every step, run the MAT to identify any memory leaks, and what is stopping the garbage collection from working, and any logical problem that you might be observing.

You can add these analysis steps to your report and document how you solved the various problems.

https://help.eclipse.org/2020-03/index.jsp?topic=%2Forg.eclipse.mat.ui.help%2Fgettingstarted%2Fbasictutorial.html

https://help.eclipse.org/2020-03/index.jsp?topic=%2Forg.eclipse.mat.ui.help%2Fgettingstarted%2Fbasictutorial.html

From the theoretical concepts discussed in the lecture, consider adding a memory management unit (MMU) functions to allocate memory in the heap. If you choose to implement an MMU, you will need to define the instructions to dynamically allocate and free memory in the instruction set that your OS Simulator supports. You might choose to include an automatic garbage collection that free allocated memory automatically without specifying instructions to do this. You might design the suitable times in which it is triggered. You might employ concepts such as paging, segmentation, swapping, and virtual memory. You might also consider the memory requirements by a process as a resource request that is considered in your resource allocation based scheduling and deadlock prevention algorithm implementation.

Exercise 2:

Assuming that a system has a 32-bit virtual address, write a Java program that is passed (1) the size of a page and (2) the virtual address. Your program will report the page number and offset of the given virtual address with the specified page size. Page sizes must be specified as a power of 2 and within the range 1024 —16384 (inclusive). Assuming such a program is named Address, it would run as follows:

```
java Address 4096 19986
```

and the correct output would appear as:

```
The address 19986 contains:
page number = 4
offset = 3602
```