哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

Chapter 8: Software Construction for Performance

8.2 Dynamic Program Analysis Methods and Tools

Ming Liu

April 26, 2019

Outline

- **Program Profiling: Concepts and Approaches**
- **Program Profiling Tools**
- **Summary**

1 Program Profiling: Concepts and Approaches

(1) What is program profiling?

What is Profiling?

- **Profiling is a form of dynamic program analysis that measures, for example, the space (memory) or time complexity of a program, the usage of particular instructions, or the frequency and duration of function calls.**
  - Most commonly, profiling information serves to aid program optimization.
- **Program analysis tools are extremely important for understanding program behavior.**
  - Computer architects need such tools to evaluate how well programs will perform on new architectures.
  - Software writers need tools to analyze their programs and identify critical sections of code.
  - Compiler writers often use such tools to find out how well their instruction scheduling or branch prediction algorithm is performing...

What is Profiling?

- **Allows you to learn:**
  - Where your program is spending its time
  - What functions called what other functions
- **Can show you which pieces of your program are slower than you expected.**
  - might be candidates for rewriting
- **Are functions are being called more or less often than expected?**
- **This may help you spot bugs that had otherwise been unnoticed.**

(2) Profiler

Profiler

- **Profiling is achieved by instrumenting either the program source code or its binary executable form using a tool called a profiler (or code profiler).**

- **Use information collected during the actual execution of a program**
- **Can be used on programs that are too large or too complex to analyze by reading the source**
- **How your program is run affects the information that shows up in the profile data**
  - Don't use some feature of your program while it is being profiled, no profile information will be generated for that feature!

Profiler types based on output

- **Flat profiler**
  - Flat profilers compute the average call times, from the calls, and do not break down the call times based on the callee or the context.
- **Call-graph profiler**
  - Call graph profilers show the call times, and frequencies of the functions, and also the call-chains involved based on the callee. In some tools full context is not preserved.
- **Input-sensitive profiler**
  - Input-sensitive profilers add a further dimension to flat or call-graph profilers by relating performance measures to features of the input workloads, such as input size or input values. They generate charts that characterize how an application's performance scales as a function of its input.

(3) Profiling Approaches

Profiling Techniques Overview

- **Profilers may use a number of different techniques, such as event-based, statistical, instrumented, and simulation methods.**

- **Profilers use a wide variety of techniques to collect data, including:**
  - Hardware interrupts
  - Code instrumentation
  - Instruction set simulation
  - Operating system hooks
  - Performance counters

Profiling Techniques Overview

- **Profilers use one of three techniques for profiling programs:**
  - Insertion
  - Sampling
  - Instrumented virtual machine (java)

- **Why is it important to understand how a profiler works?**
  - Each different technique has its own pros & cons
  - Different profilers may give different results

(1) Insertion/Instrumentation

- **This technique effectively adds instructions to the target program to collect the required information.**

- **Source code insertion**
  - Profiling code goes in with the source
  - Easy to do, e.g. use very simple code-insertion techniques
    System.out.println(System.getCurrentTimeMillis());

- **Object code insertion**
  - Profiling code goes into the .o (C++) or .class (Java) files
  - Can be done statically or dynamically
  - Hard to do (modified class loader)

Insertion Pros & Cons

- **Insertion Pros:**
  - Can be used across a variety of platforms
  - Accurate (in some ways)
    - Can't easily do memory profiling
- **Insertion Cons:**
  - Requires recompilation or relinking of the app
  - Profiling code may affect performance
    - difficult to calculate exact impact

(2) Sampling

- **In sampling, the processor or VM is monitored and at regular intervals an interrupt executes and saves a "snapshot" of the processor state**
- **This data is then compared with the program's layout in memory to get an idea of where the program was at each sample**

Sampling Pros & Cons

- **Sampling Pros:**
  - No modification of app is necessary
- **Sampling Cons:**
  - A definite time/accuracy trade-off
    - a high sample rate is accurate, but takes a lot of time
  - Very small methods will almost always be missed
    - if a small method is called frequently and you have are unlucky, small but expensive methods may never show up
  - Sampling cannot easily monitor memory usage

(3) Instrumented VM

- **Another way to collect information is to instrument the Java VM**
- **Using this technique each and every VM instruction can be monitored (highly accurate)**

- **Pros:**
  - The most accurate technique
  - Can monitor memory usage data as well as time data
  - Can easily be extended to allow remote profiling
- **Cons:**
  - The instrumented VM is platform-specific

(4) Data granularity in profilers

(1) Event-based profilers

- **Based on their data granularity (how profilers collect information), they are classified into event based or statistical profilers.**

- **Event-based profilers**
  - Java: the JVMTI (JVM Tools Interface) API, formerly JVMPI (JVM Profiling Interface), provides hooks to profilers, for trapping events like calls, class-load, unload, thread enter leave.
  - Python: Python profiling includes the profile module, hotshot (which is call-graph based), and using the 'sys.setprofile' function to trap events like c_{call,return,exception}, python_{call,return,exception}.

(2) Statistical profilers

- **Some profilers operate by sampling. A sampling profiler probes the target program's call stack at regular intervals using operating system interrupts.**
- **Sampling profiles are typically less numerically accurate and specific, but allow the target program to run at near full speed.**
  - The resulting data are not exact, but a statistical approximation. "The actual amount of error is usually more than one sampling period. In fact, if a value is n times the sampling period, the expected error in it is the square-root of n sampling periods."
  - In practice, sampling profilers can often provide a more accurate picture of the target program's execution than other approaches, as they are not as intrusive to the target program, and thus don't have as many side effects (such as on memory caches or instruction decoding pipelines).

(3) Instrumentation

- **This technique effectively adds instructions to the target program to collect the required information.**
  - Note that instrumenting a program can cause performance changes, and may in some cases lead to inaccurate results and/or heisenbugs. The effect will depend on what information is being collected, and on the level of detail required.
  - For example, adding code to count every procedure/routine call will probably have less effect than counting how many times each statement is obeyed.
  - A few computers have special hardware to collect information; in this case the impact on the program is minimal.
  - See “Insertion” for details.

(4) Interpreter instrumentation

- **Interpreter debug options can enable the collection of performance metrics as the interpreter encounters each target statement.**
- **A bytecode, control table or JIT interpreters are three examples that usually have complete control over execution of the target code, thus enabling extremely comprehensive data collection opportunities.**
  - Hypervisor/Simulator
  - Hypervisor: Data are collected by running the (usually) unmodified program under a hypervisor. Example: SIMMON
  - Simulator and Hypervisor: Data collected interactively and selectively by running the unmodified program under an Instruction Set Simulator.

2 Program Profiling Tools

Profiling Tools for Java

- **Command-line tools (in JDK)**
- **JConsole (in JDK)**
- **Visual VM (in JDK)**
- **Memory Analyzer (MAT)**

(1) Command-line profiling tools

jps

- **Java command line tools for profiling are installed in the bin sub-directory of the JDK home installed directory.**
- **jps**
  - The jps tool lists running applications of the instrumented JVMs on the target system. If jps is run without specifying a hostid, it will look for instrumented JVMs on the local host.
  - The jps command uses the java launcher to find the class name and arguments passed to the main method.
  - E.g.(重新设计)

```
C:\Documents and Settings\Administrator>jps -m
7988 Main --log-config-file D:\tools\squirrel-sql-3.2.1\log4j.properties
--squir
rel-home D:\tools\squirrel-sql-3.2.1
7456 Jps -m
```

Jstat

- **Jstat**
  - The jstat tool displays performance statistics for an instrumented HotSpot Java virtual machine (JVM).
  - Jstat can show Java application's heap usage and GC in great detail.

```
C:\Documents and Settings\Administrator>jstat -class -t 2972 1000 2
Timestamp       Loaded  Bytes  Unloaded  Bytes     Time
1395.6            2375  2683.8        7    6.2       3.45
1396.6            2375  2683.8        7    6.2       3.45
```

Jinfo

- **Jinfo**
  - jinfo prints Java configuration information for a given Java process or core file or a remote debug server. Configuration information includes Java System properties and Java virtual machine command line flags.

```
C:\Documents and Settings\Administrator>jinfo -flag MaxTenuringThreshold
2972
-XX:MaxTenuringThreshold=15
```

显示是否打印 GC 详细信息：

```
C:\Documents and Settings\Administrator>jinfo -flag PrintGCDetails  2972
-XX:-PrintGCDetails
```

Jmap

- **Jmap**
  - jmap prints shared object memory maps or heap memory details of a given process or core file or a remote debug server.

下例使用 jmap 生成 PID 为 2972 的 Java 程序的对象统计信息，并输出到 s.txt 文件中：

```
jmap -histo 2972 >c:\s.txt
```

输出文件有如下结构：

```
num     #instances         #bytes  class name
----------------------------------------------
   1:          4983       6057848  [I
   2:         20929       2473080  <constMethodKlass>
.....
1932:             1             8  sun.java2d.pipe.AlphaColorPipe
1933:             1             8  sun.reflect.GeneratedMethodAccessor64
Total        230478      22043360
```

可以看到，这个输出显示了内存中的实例数量和合计。

jhat

- **jhat**
  - The jhat command parses a java heap dump file and launches a webserver.
  - jhat enables you to browse heap dumps using your favorite webbrowser.
  - jhat supports pre-designed queries (such as 'show all instances of a known class "Foo"') as well as OQL (Object Query Language) - a SQL-like query language to query heap dumps.
  - There are several ways to generate a java heap dump:
    - Use jmap -dump option to obtain a heap dump at runtime;
    - Use jconsole option to obtain a heap dump via HotSpotDiagnosticMXBean at runtime;
    - Heap dump will be generated when OutOfMemoryError is thrown by specifying -XX:+HeapDumpOnOutOfMemoryError VM option;
    - Use hprof.

jhat

- **jhat**

使用 jhat 工具可以用于分析 Java 应用程序的堆快照内容。以前文中 jmap 的输出堆文件 heap.hprof 为例：

```
C:\Documents and Settings\Administrator>jhat c:\heap.hprof
Reading from c:\heap.hprof...
Dump file created Thu Mar 22 22:27:19 CST 2012
Snapshot read, resolving...
Resolving 133055 objects...
Chasing references, expect 26 dots..........................
Eliminating duplicate references..........................
Snapshot resolved.
Started HTTP server on port 7000
Server is ready.
```

jhat 在分析完成后，使用 HTTP 服务器展示其分析结果。在浏览器中访问 http://127.0.0.1:7000，结果如图 6.19 所示。

Jstack

- **Jstack**
  - jstack prints Java stack traces of Java threads for a given Java process or core file or a remote debug server.
  - For each Java frame, the full class name, method name, 'bci' (byte code index) and line number, if available, are printed.
  - With the -m option, jstack prints both Java and native frames of all threads along with the 'pc' (program counter).
  - For each native frame, the closest native symbol to 'pc', if available, is printed. C++ mangled names are not demangled.

Jstack

- **Jstack（一个用jstack寻找死锁的例子）**

下例演示了一个简单的死锁，两个线程分别占用south锁和north锁，并同时请求对方占用的锁，导致死锁发生。

```
public class DeadLock extends Thread{
    protected Object myDirect;
    static ReentrantLock south = new ReentrantLock();
    static ReentrantLock north = new ReentrantLock();

    public DeadLock(Object obj){
        this.myDirect=obj;
        if(myDirect==south){
            this.setName("south");
        }
        if(myDirect==north){
            this.setName("north");
        }
    }
    @Override
    public void run() {
        if (myDirect == south) {
            try {
                north.lockInterruptibly();          //占用 north
                try {
                    Thread.sleep(500);              //等待 north 启动
                } catch (Exception e) {
                    e.printStackTrace();
                }
                south.lockInterruptibly();          //占用 south
                System.out.println("car to south has passed");
            } catch (InterruptedException e1) {
                System.out.println("car to south is killed");
            }finally{
                if(north.isHeldByCurrentThread())
                    north.unlock();
                if(south.isHeldByCurrentThread())
                    south.unlock();
            }

        }
        if (myDirect == north) {
            try {
                south.lockInterruptibly();          //占用 south
                try {
                    Thread.sleep(500);
                } catch (Exception e) {
                    e.printStackTrace();
                }
                north.lockInterruptibly();          //占用 north
                System.out.println("car to north has passed");
            } catch (InterruptedException e1) {
                System.out.println("car to north is killed");
            }finally{
                if(north.isHeldByCurrentThread())
                    north.unlock();
                if(south.isHeldByCurrentThread())
                    south.unlock();
            }

        }
    }

    public static void main(String[] args) throws InterruptedException
        DeadLock car2south = new DeadLock(south);       //2 个线程死锁
        DeadLock car2north = new DeadLock(north);
        car2south.start();
        car2north.start();
        Thread.sleep(1000);
    }
}
```

使用jstack工具打印上例的输出，部分结果如下：

```
"north" prio=6 tid=0x02bb9c00 nid=0x2b4 waiting on condition [0x02f5f000]
   java.lang.Thread.State: WAITING (parking)
    at sun.misc.Unsafe.park(Native Method)
    - parking to wait for  <0x22a19948> (a java.util.concurrent.locks.
    ReentrantLock$NonfairSync)
    at java.util.concurrent.locks.LockSupport.park(Unknown Source)
    at java.util.concurrent.locks.AbstractQueuedSynchronizer.parkAnd-
    CheckInterrupt(Unknown Source)
```

Jstack

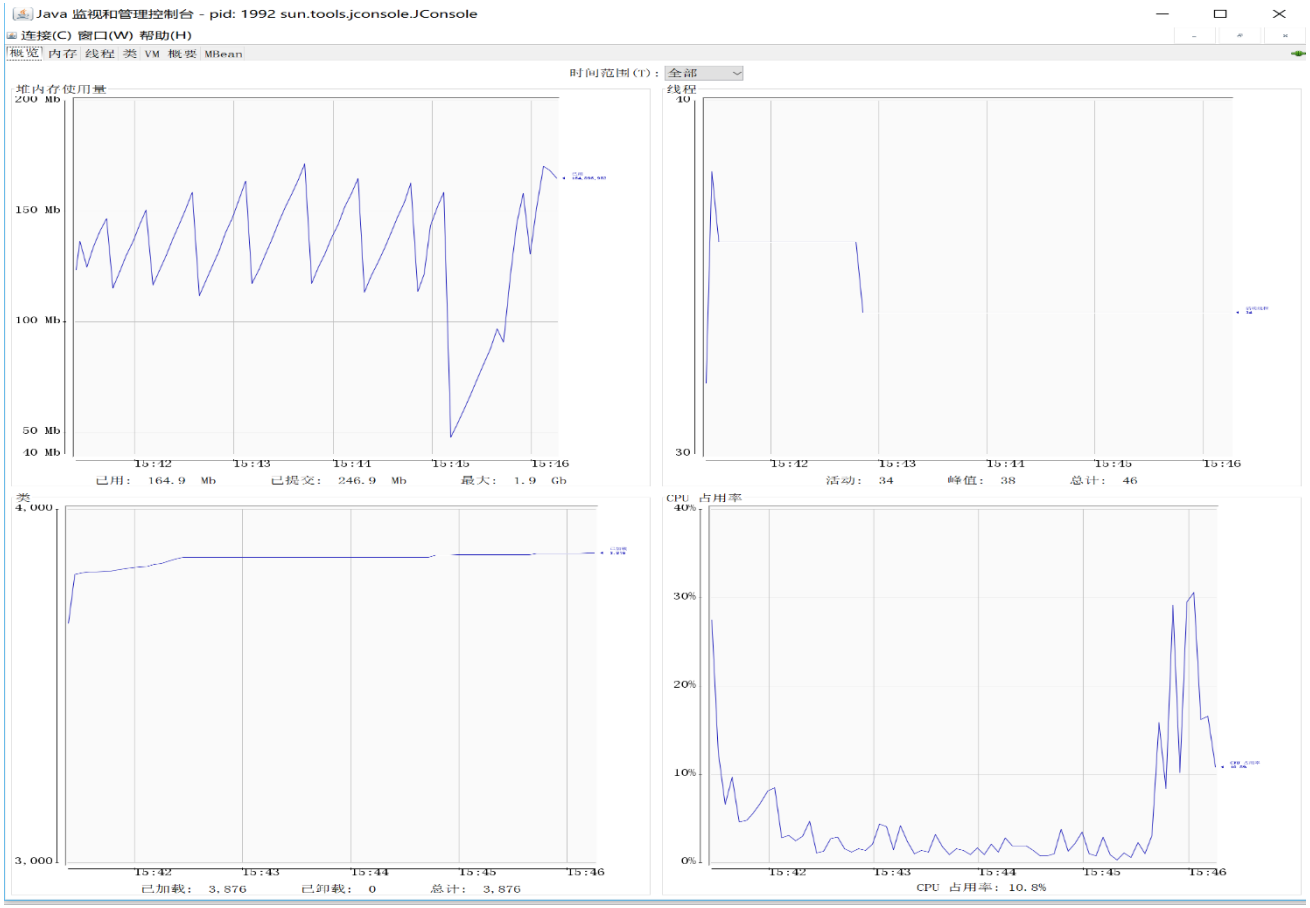
- **Jstack**

```
    at java.util.concurrent.locks.AbstractQueuedSynchronizer.doAcquire-
    Interruptibly(Unknown Source)
    at java.util.concurrent.locks.AbstractQueuedSynchronizer.acquireInt-
    erruptibly(Unknown Source)
    at java.util.concurrent.locks.ReentrantLock.lockInterruptibly-
    (Unknown Source)
    at javatuning.ch6.toolscheck.DeadLock.run(DeadLock.java:49)

  Locked ownable synchronizers:
    - <0x22a19920> (a java.util.concurrent.locks.ReentrantLock$Nonfair-
    Sync)

"south" prio=6 tid=0x02bb9000 nid=0x750 waiting on condition [0x02f0f000]
  java.lang.Thread.State: WAITING (parking)
    at sun.misc.Unsafe.park(Native Method)
    - parking to wait for  <0x22a19920> (a java.util.concurrent.locks.
    ReentrantLock$NonfairSync)
    at java.util.concurrent.locks.LockSupport.park(Unknown Source)
    at java.util.concurrent.locks.AbstractQueuedSynchronizer.parkAnd-
    CheckInterrupt(Unknown Source)
    at java.util.concurrent.locks.AbstractQueuedSynchronizer.doAcquire-
    Interruptibly(Unknown Source)
    at java.util.concurrent.locks.AbstractQueuedSynchronizer.acquire-
    Interruptibly(Unknown Source)
    at java.util.concurrent.locks.ReentrantLock.lockInterruptibly-
    (Unknown Source)
    at javatuning.ch6.toolscheck.DeadLock.run(DeadLock.java:29)

  Locked ownable synchronizers:
    - <0x22a19948> (a java.util.concurrent.locks.ReentrantLock$Nonfair-
    Sync)

//省略部分输出

Found one Java-level deadlock:          //找到死锁
=============================
"north":                                //死锁线程名字
  waiting for ownable synchronizer 0x22a19948, (a java.util.concurrent.
  locks.ReentrantLock$NonfairSync),
  which is held by "south"
"south":                                //死锁线程名字
  waiting for ownable synchronizer 0x22a19920, (a java.util.concurrent.
  locks.ReentrantLock$NonfairSync),
  which is held by "north"

//省略部分输出
```

(2) JConsole

JConsole

- **The JConsole graphical user interface is a monitoring tool that provides information about the performance and resource consumption of applications running on the Java platform.**



JConsole

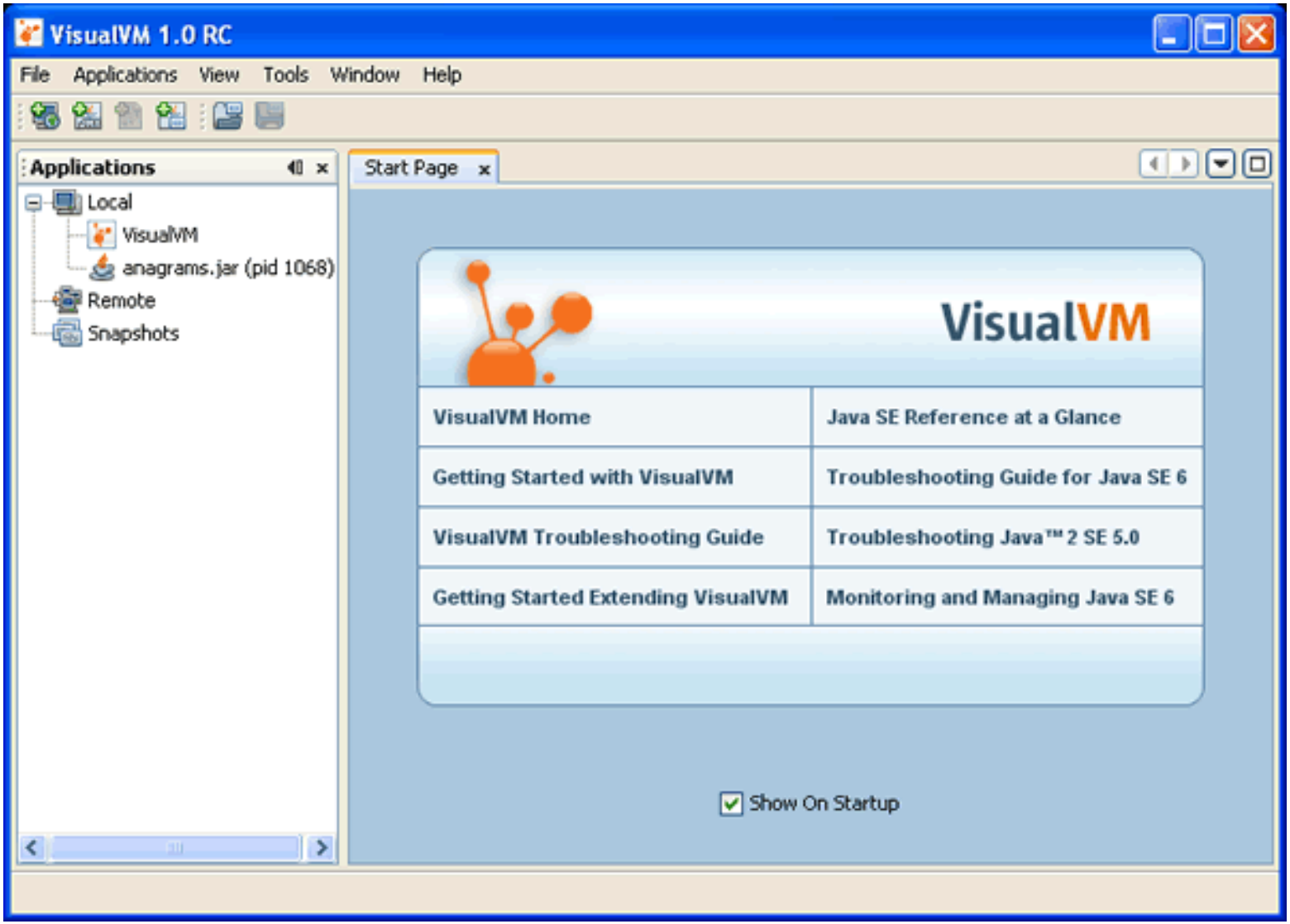
- **Starting JConsole**
  - The jconsole executable can be found in JDK_HOME/bin, where JDK_HOME is the directory in which the Java Development Kit (JDK) is installed. If this directory is in your system path, you can start JConsole by simply typing jconsole in a command (shell) prompt. Otherwise, you have to type the full path to the executable file.
- **Command Syntax**
  - You can use JConsole to monitor both local applications, namely those running on the same system as JConsole, as well as remote applications, namely those running on other systems.
  - Note - Using JConsole to monitor a local application is useful for development and for creating prototypes, but is not recommended for production environments, because JConsole itself consumes significant system resources. Remote monitoring is recommended to isolate the JConsole application from the platform being monitored.
- http://docs.oracle.com/javase/7/docs/technotes/guides/management/jconsole.html

(3) Visual VM

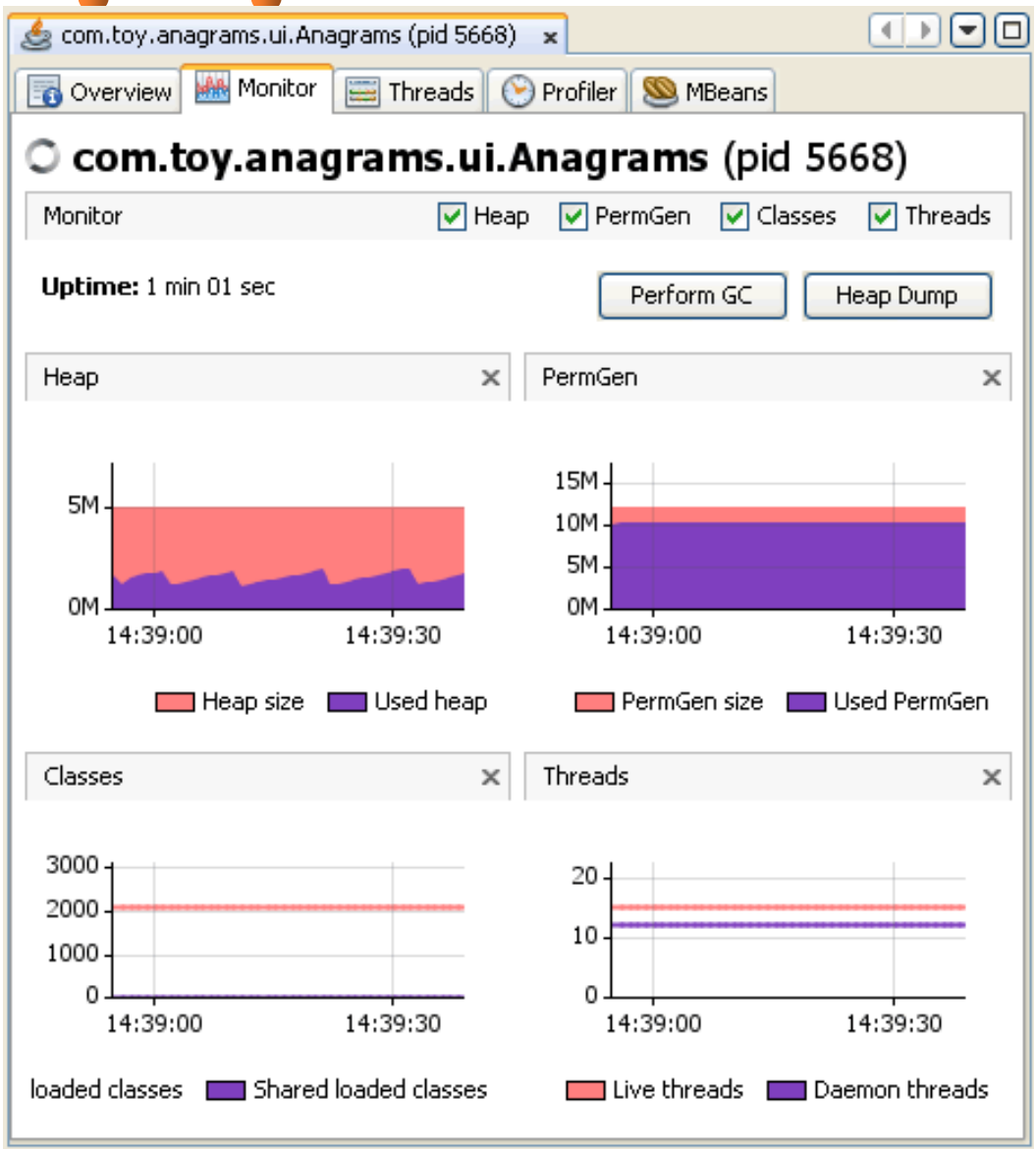
Visual VM

- **VisualVM is a tool that provides a visual interface for viewing detailed information about Java technology-based applications (Java applications) while they are running on a Java Virtual Machine (JVM).**
- **VisualVM organizes data about the JVM software that is retrieved by the Java Development Kit (JDK) tools and presents the information in a way that enables you to quickly view data on multiple Java applications. You can view data on local applications and applications that are running on remote hosts. You can also capture data about JVM software instances and save the data to your local system, and view the data later or share the data with others.**
- **In order to benefit from all Java VisualVM's features, you should run the Java Platform, Standard Edition (Java SE) version 6 or later.**
- **It can replace most of Java command-line tools, even the Jconsole tool.**

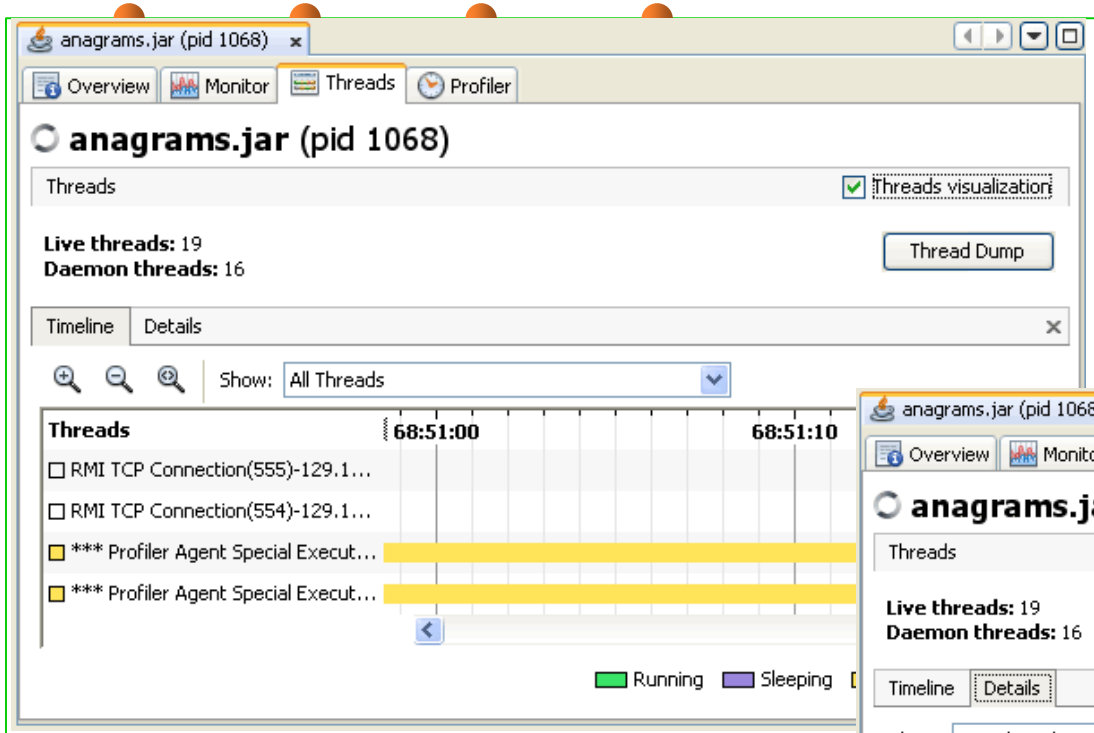
Visual VM-starting

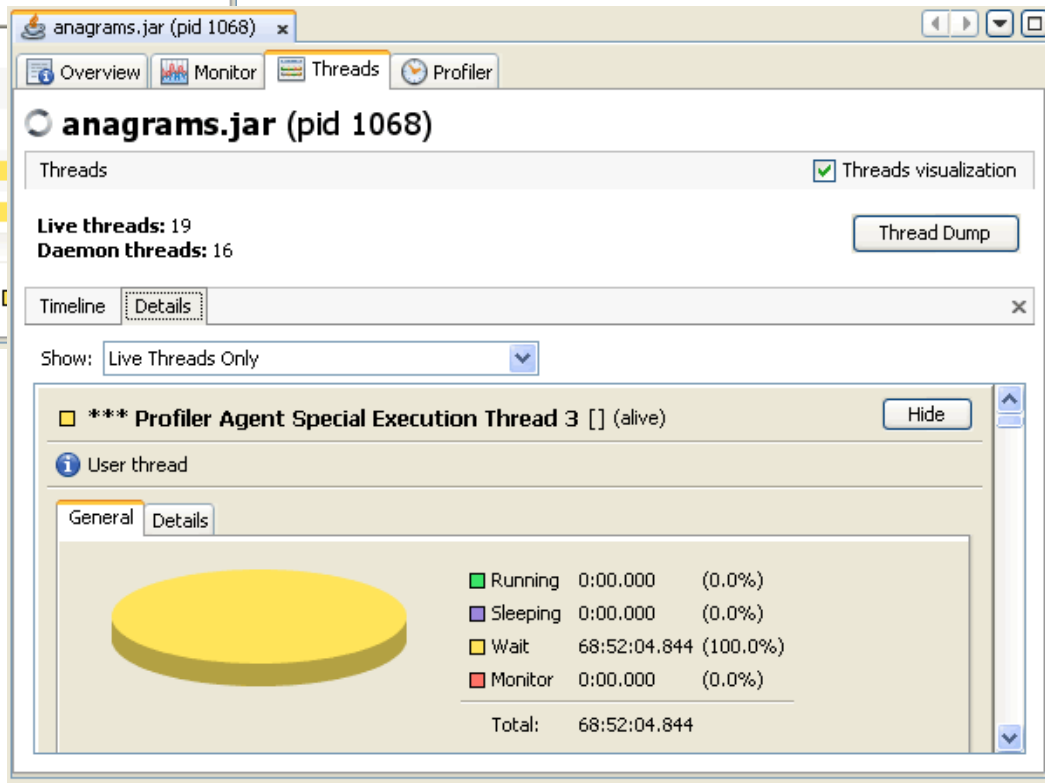


Visual VM -Monitoring an Application

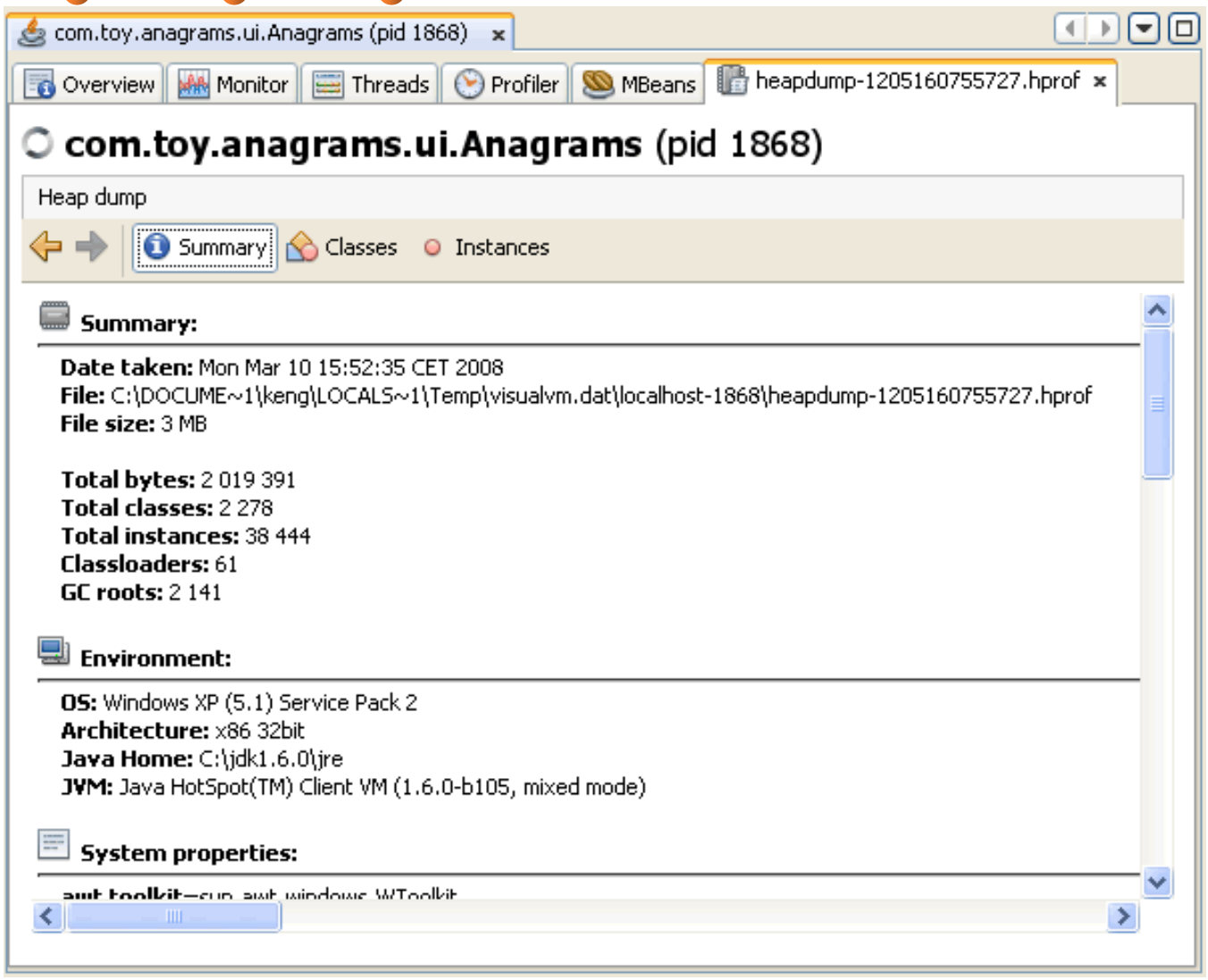


Visual VM -Monitoring Thread Activity



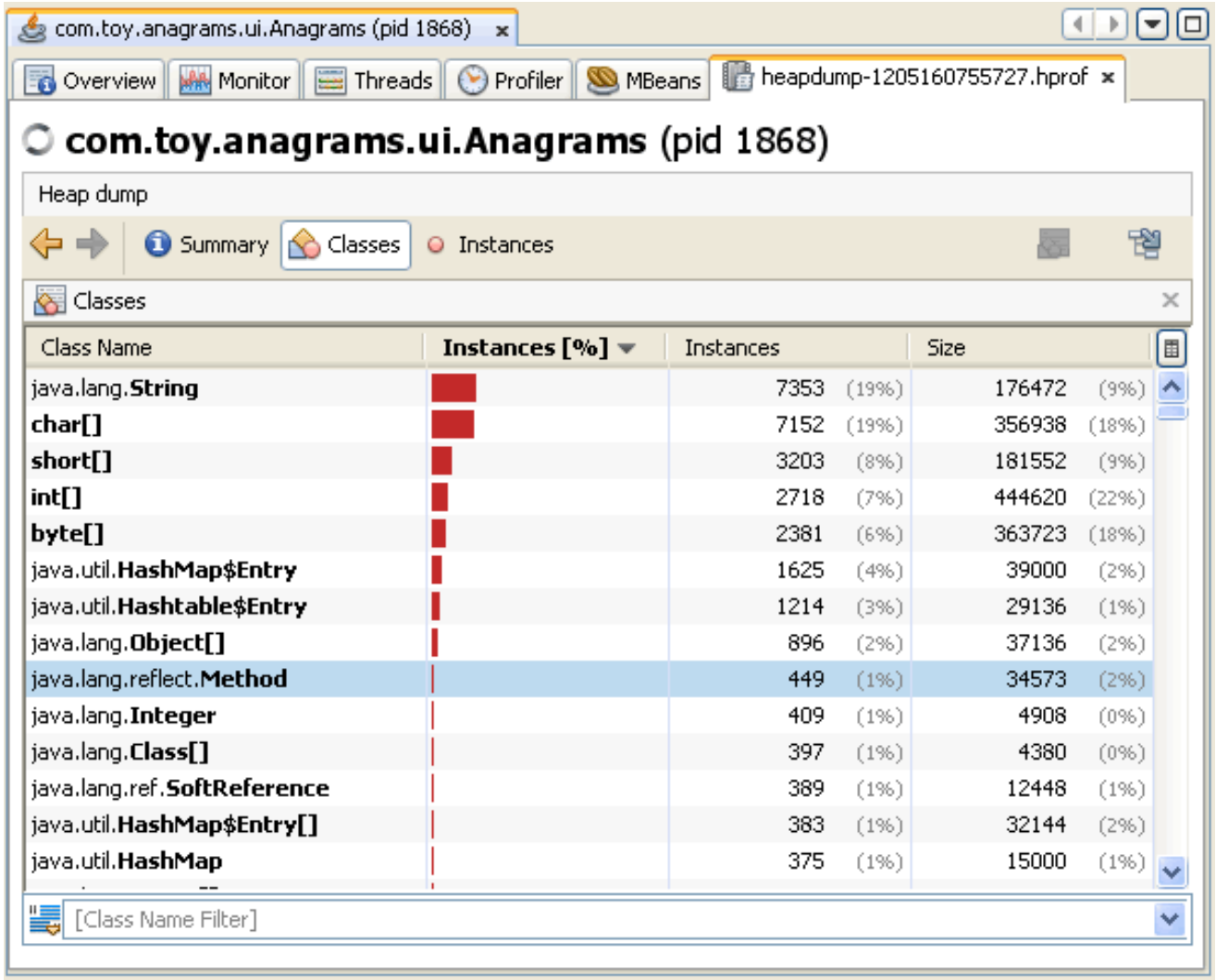


Visual VM -Browsing a Heap Dump



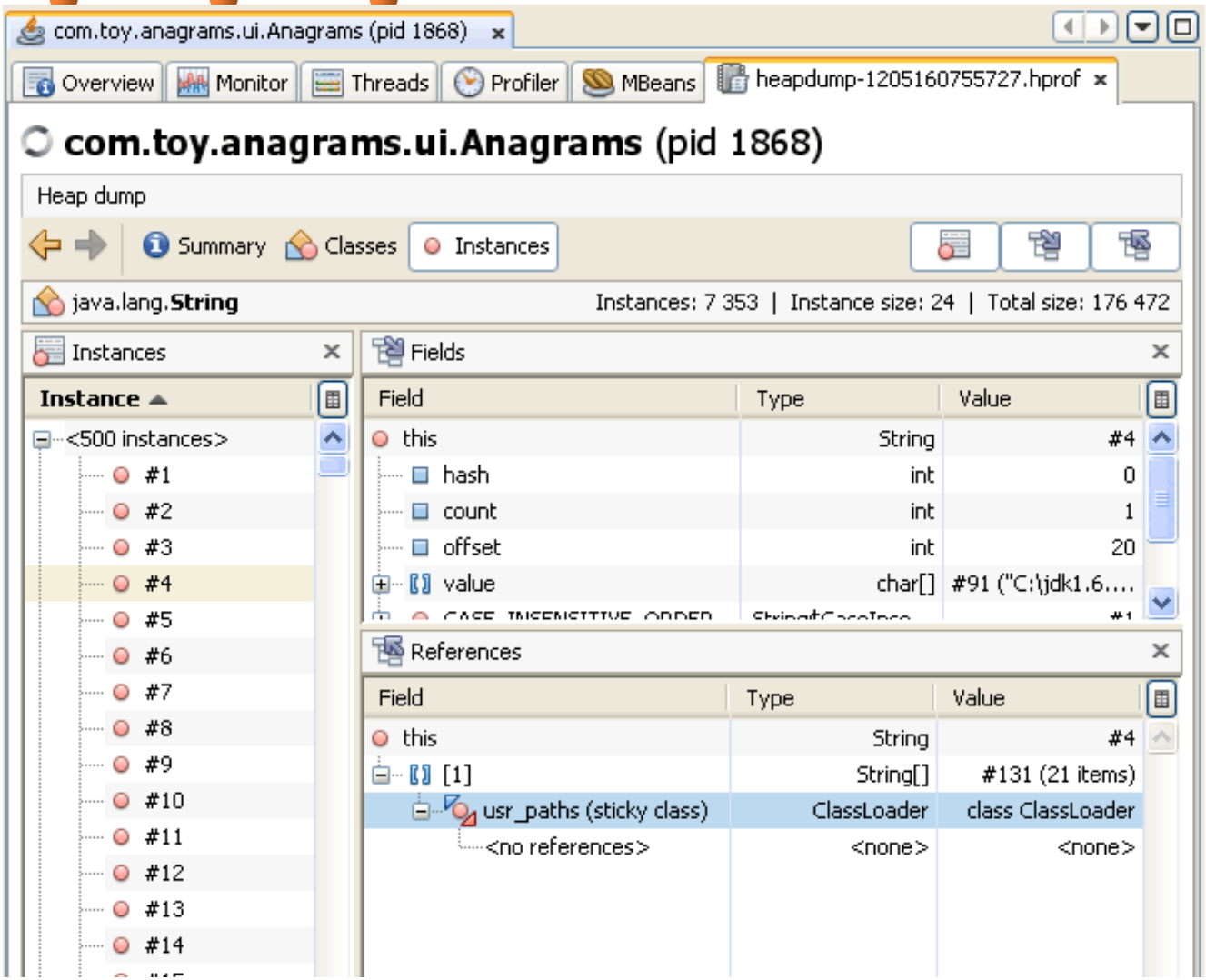
Summary View

Visual VM -Browsing a Heap Dump



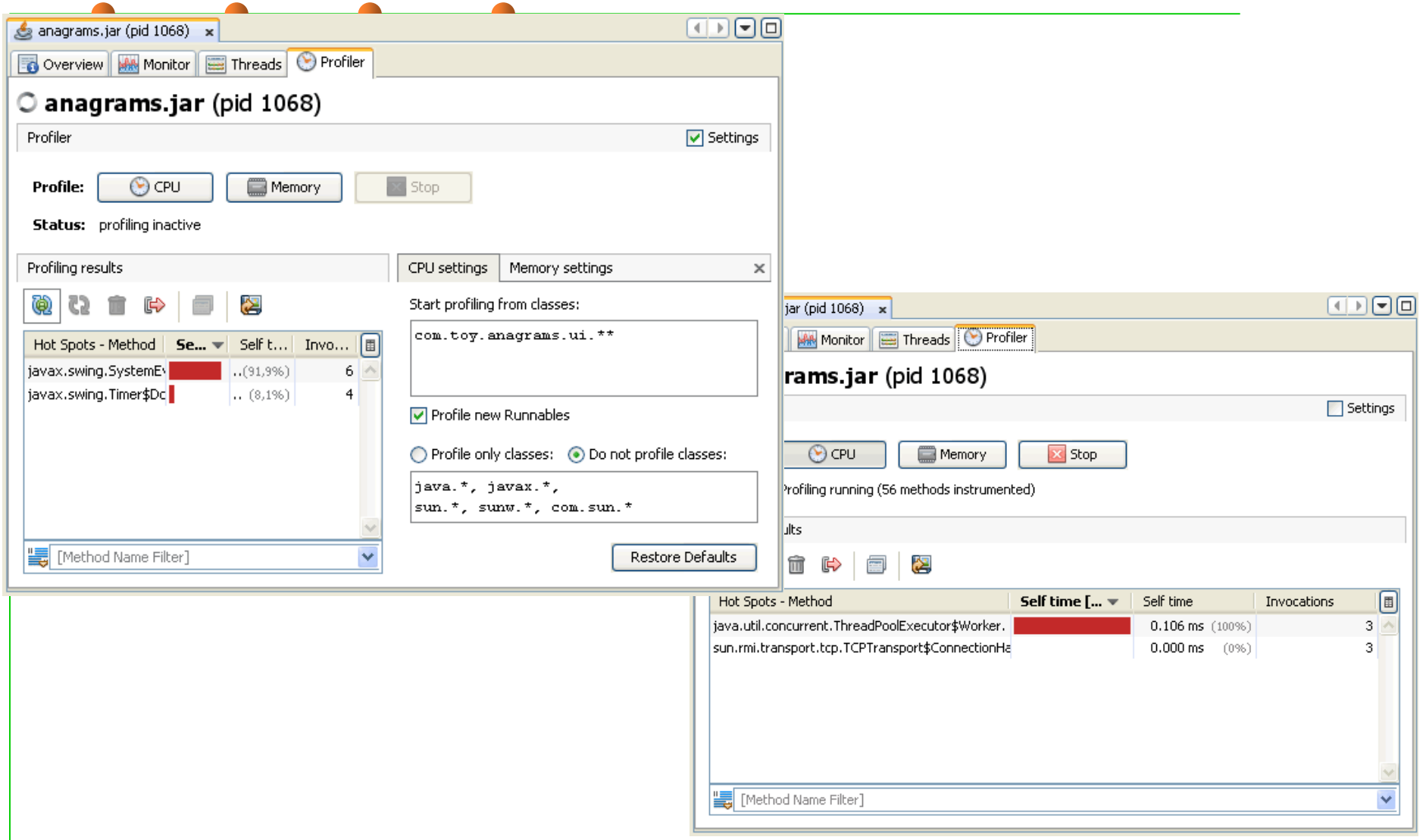
Class View

Visual VM -Browsing a Heap Dump



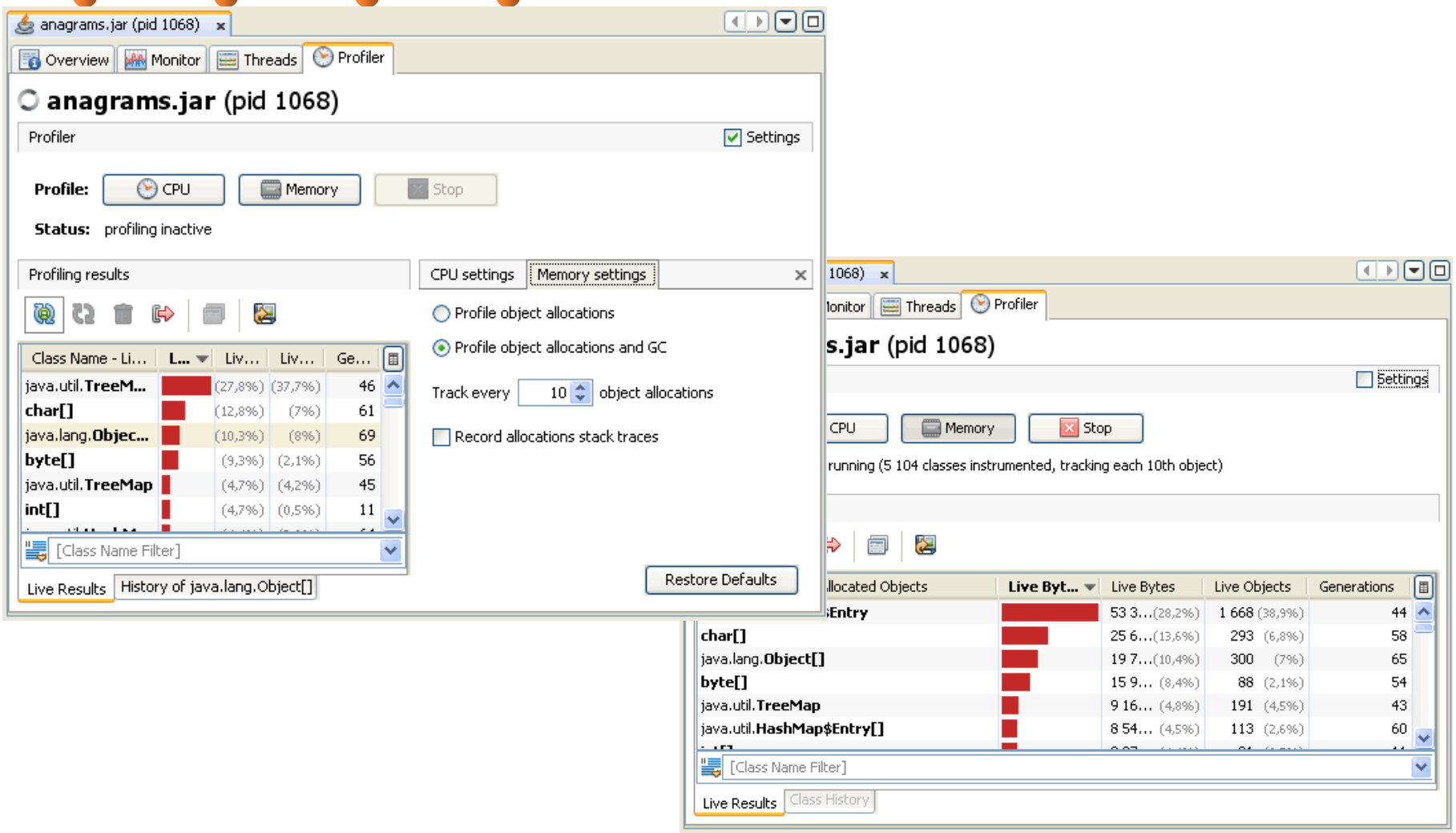
instances View

Visual VM -Profiling applications



CPU Profiling

Visual VM -Profiling applications



Memory Profiling

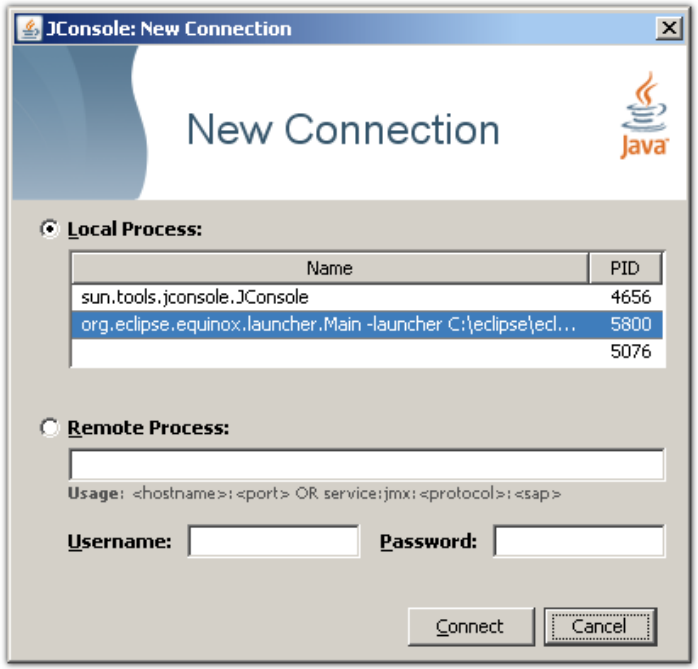
(4) Memory Analyzer (MAT)

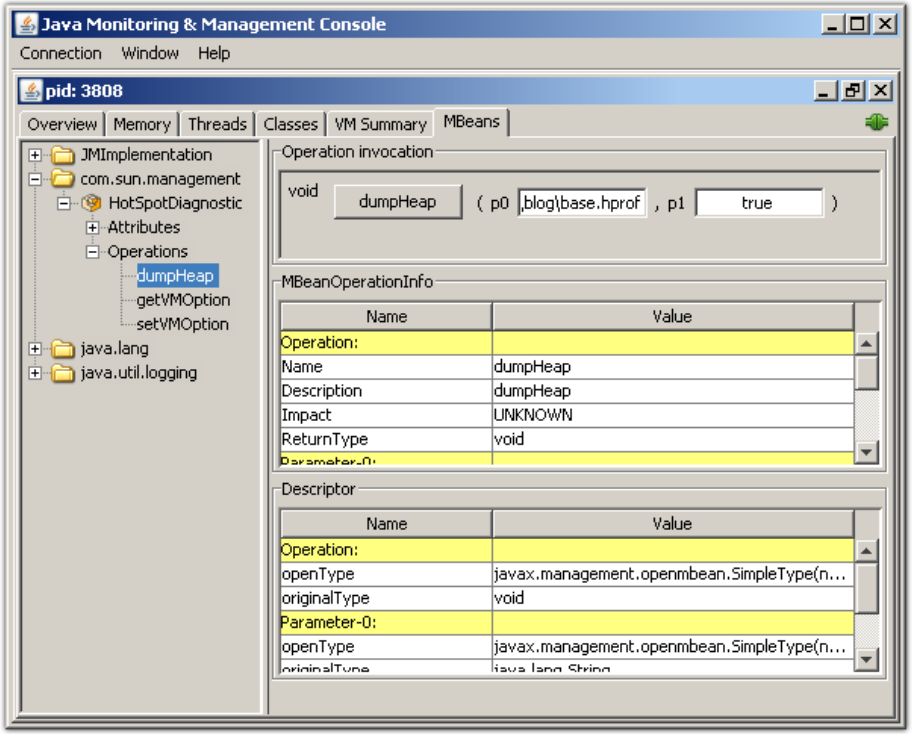
Memory Analyzer (MAT)

- **The Eclipse Memory Analyzer is a fast and feature-rich Java heap analyzer that helps you find memory leaks and reduce memory consumption.**
- **Use the Memory Analyzer to analyze productive heap dumps with hundreds of millions of objects, quickly calculate the retained sizes of objects, see who is preventing the Garbage Collector from collecting objects, run a report to automatically extract leak suspects.**
- **http://www.eclipse.org/mat/**
- **MAT can run in stand-alone manner or run as a plug-in of eclipse.**

Memory Analyzer (MAT)--dumpHeap

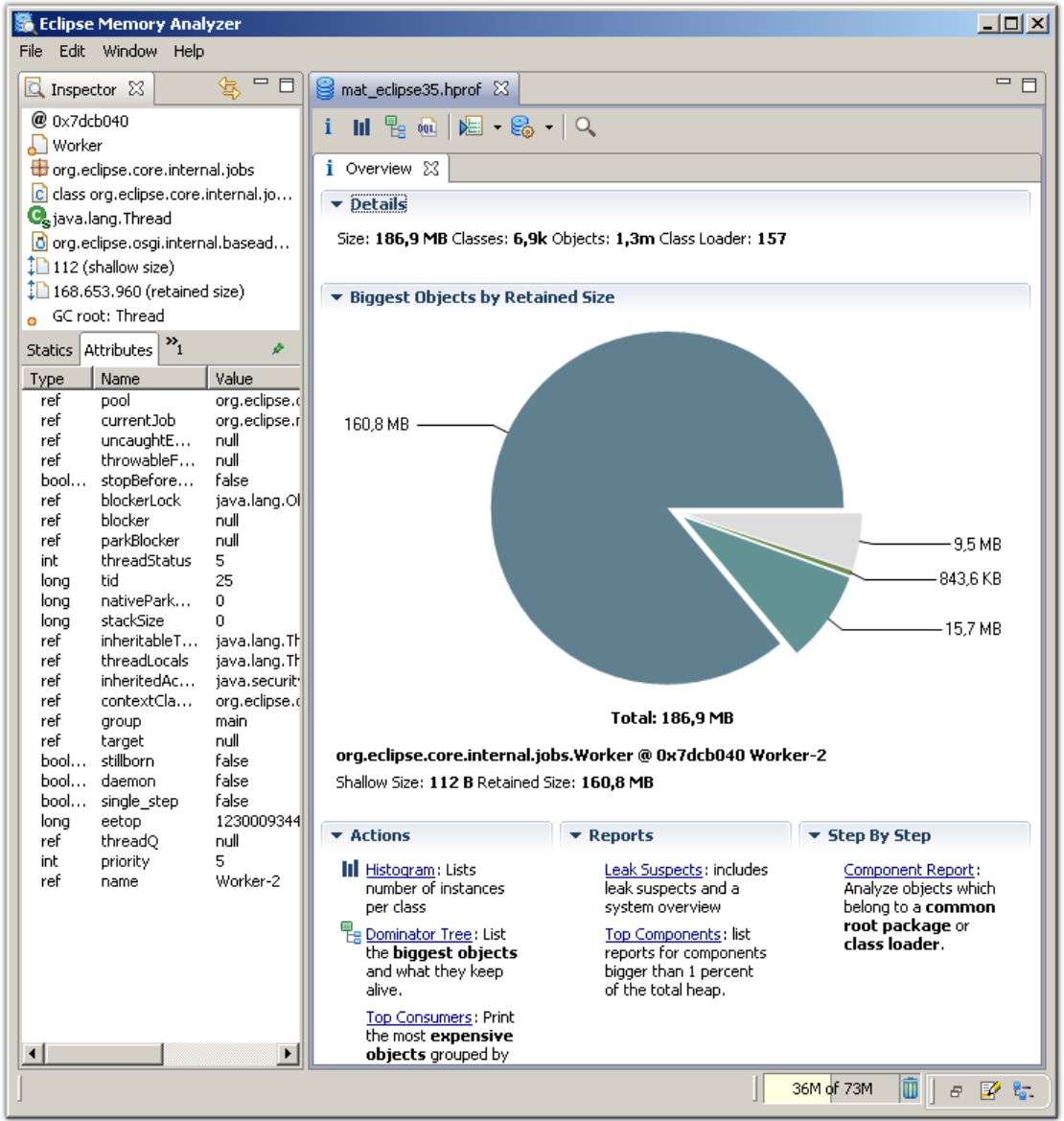
- **The Memory Analyzer works with heap dumps.**
  - Such a heap dump contains information about all Java objects alive at a given point in time.
  - All current Java Virtual Machines can write heap dumps, but the exact steps depend on vendor, version and operation system.
- **Using Jconsole to dump heap**





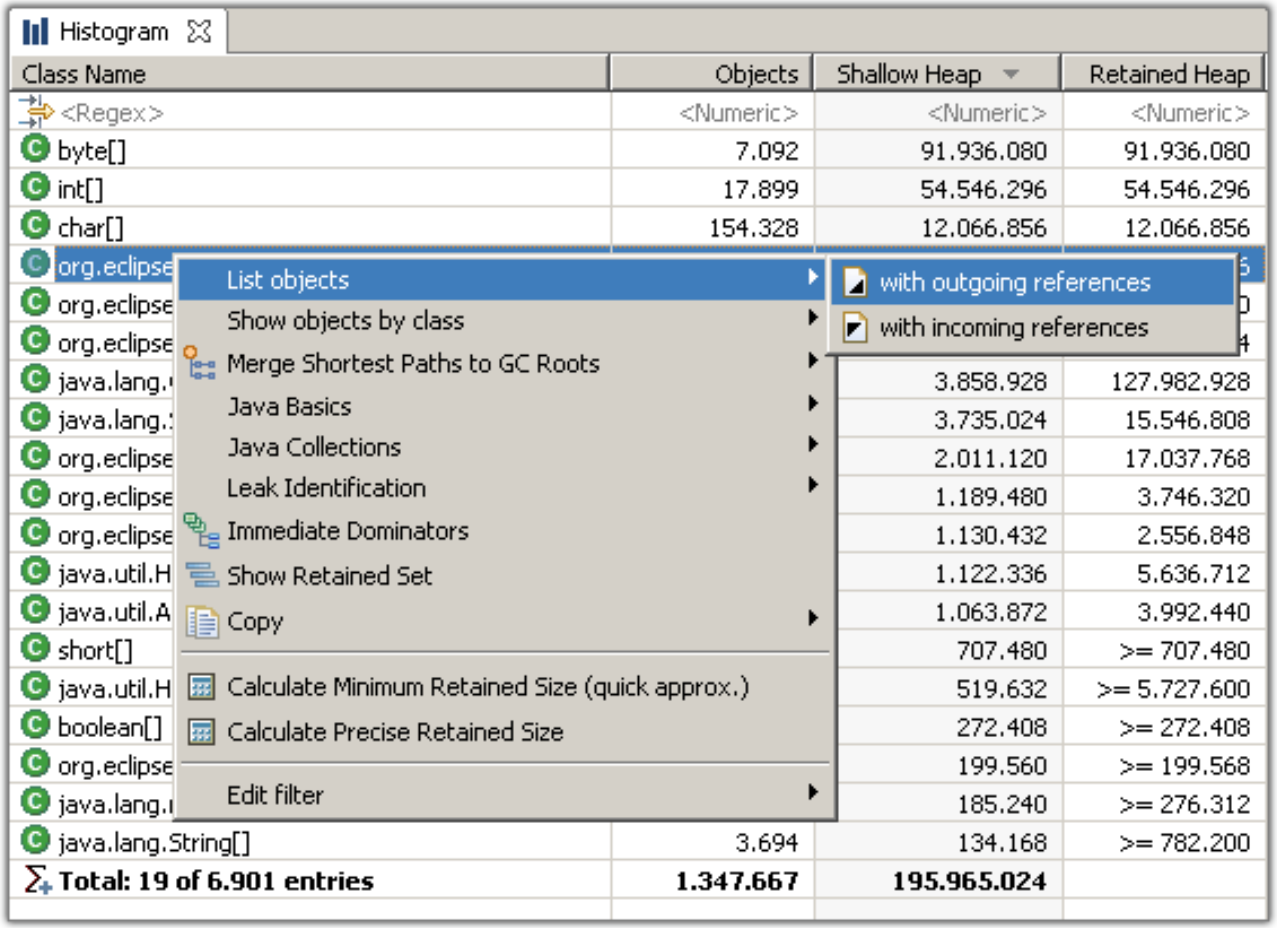
Memory Analyzer (MAT)

- **Overview page.**



Memory Analyzer (MAT) -- histogram

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

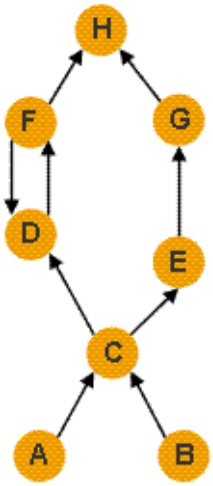


Memory Analyzer (MAT) -- histogram

- **Shallow heap is the memory consumed by one object. An object needs 32 or 64 bits (depending on the OS architecture) per reference, 4 bytes per Integer, 8 bytes per Long, etc. Depending on the heap dump format the size may be adjusted (e.g. aligned to 8, etc...) to model better the real consumption of the VM.**
- **Retained set of X is the set of objects which would be removed by GC when X is garbage collected.**
- **Retained heap of X is the sum of shallow sizes of all objects in the retained set of X, i.e. memory kept alive by X.**
- **Generally speaking, shallow heap of an object is its size in the heap and retained size of the same object is the amount of heap memory that will be freed when the object is garbage collected.**

Memory Analyzer (MAT) -- histogram

- **The retained set for a leading set of objects, such as all objects of a particular class or all objects of all classes loaded by a particular class loader or simply a bunch of arbitrary objects, is the set of objects that is released if all objects of that leading set become unaccessible.**
- **The retained set includes these objects as well as all other objects only accessible through these objects. The retained size is the total heap size of all objects contained in the retained set.**

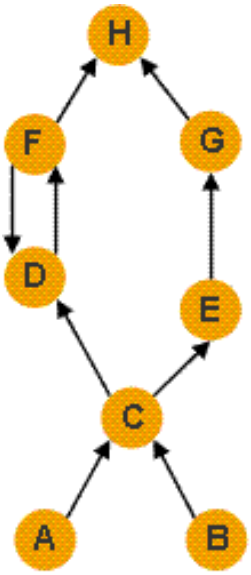


A and **B** are **garbage collection roots**, e.g. method parameters, locally created objects, objects used for wait(), notify() or synchronized(), etc.

Leading Set	Retained Set
E	E,G
C	C,D,E,F,G,H
A,B	A,B,C,D,E,F,G,H

Memory Analyzer (MAT) -- histogram

- **The Minimum Retained Size gives a good (under)estimation of the retained size which is calculated ways faster than the exact retained size of a set of objects. It only depends on the number of objects in the inspected set, not the number of objects in the heap dump.**

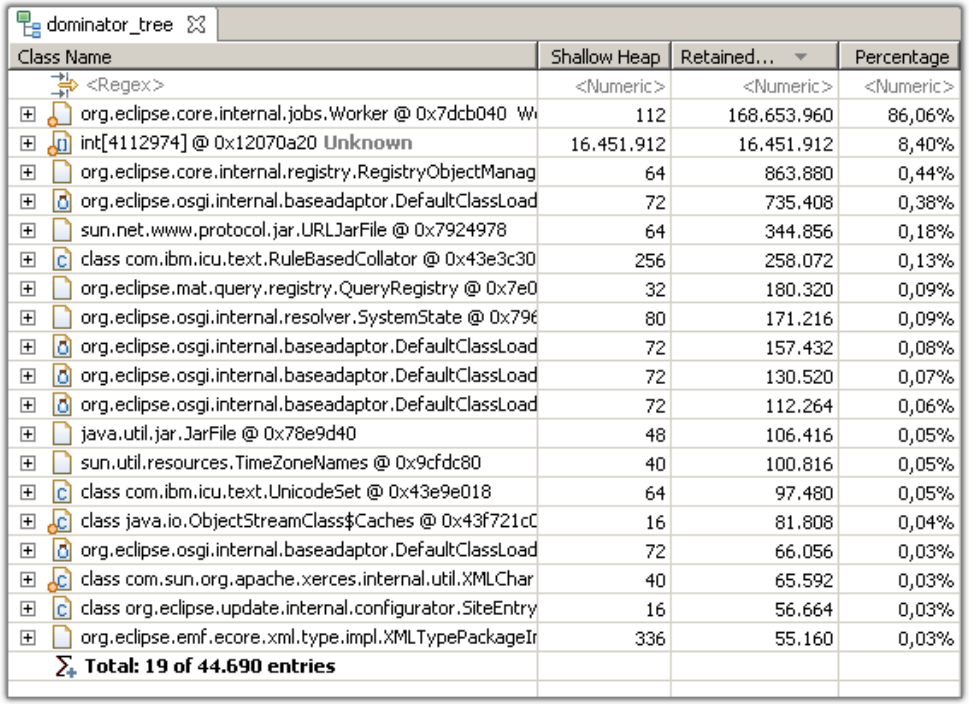


A and **B** are **garbage collection roots**, e.g. method parameters, locally created objects, objects used for wait(), notify() or synchronized(), etc.

Leading Set	Retained Set
E	E,G
C	C,D,E,F,G,H
A,B	A,B,C,D,E,F,G,H

Memory Analyzer (MAT)- 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

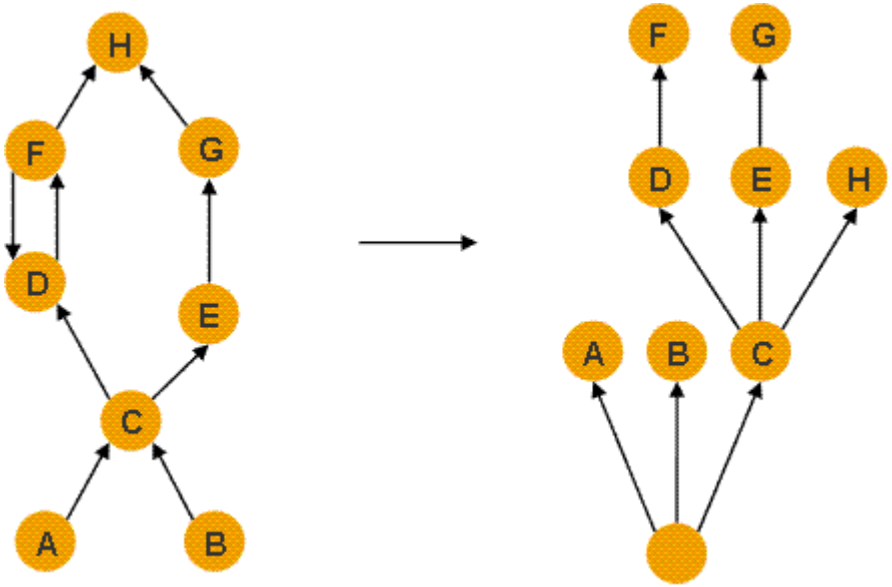
Class Name	Shallow Heap	Retained...	Percentage
<Regex>	<Numeric>	<Numeric>	<Numeric>
org.eclipse.core.internal.jobs.Worker @ 0x7dcb040 W	112	168.653.960	86,06%
int[4112974] @ 0x12070a20 Unknown	16.451.912	16.451.912	8,40%
org.eclipse.core.internal.registry.RegistryObjectManag	64	863.880	0,44%
org.eclipse.osgi.internal.baseadaptor.DefaultClassLoad	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 @ 0x796	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			

Memory Analyzer (MAT)- dominator tree

- **Dominator Tree**
  - MAT provides a dominator tree of the object graph. The transformation of the object reference graph into a dominator tree allows you to easily identify the biggest chunks of retained memory and the keep-alive dependencies among objects. Bellow is an informal definition of the terms.
- **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.**
- **The immediate dominator x of some object y is the dominator closest to the object y.**
- **A dominator tree is built out of the object graph. In the dominator tree each object is the immediate dominator of its children, so dependencies between the objects are easily identified.**

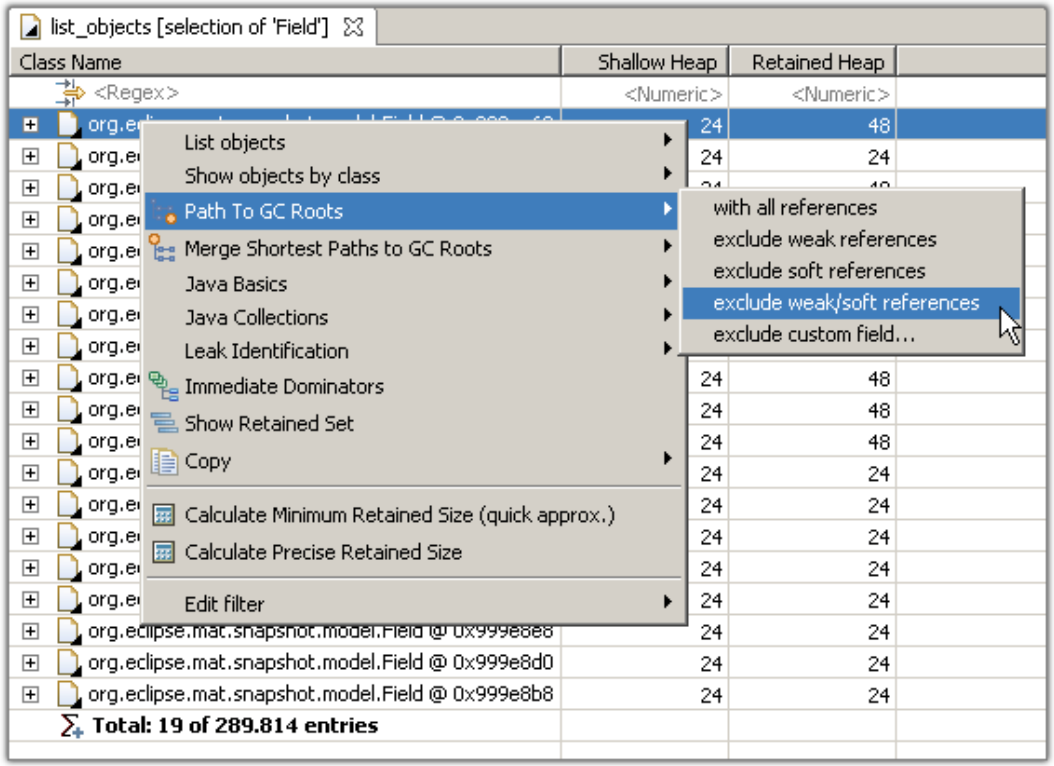
Memory Analyzer (MAT)- dominator tree

- **The dominator tree has the following important properties:**
  - The objects belonging to the sub-tree of x (i.e. the objects dominated by x) represent the retained set of x .
  - If x is the immediate dominator of y , then the immediate dominator of x also dominates y , and so on.
  - The edges in the dominator tree do not directly correspond to object references from the object graph.



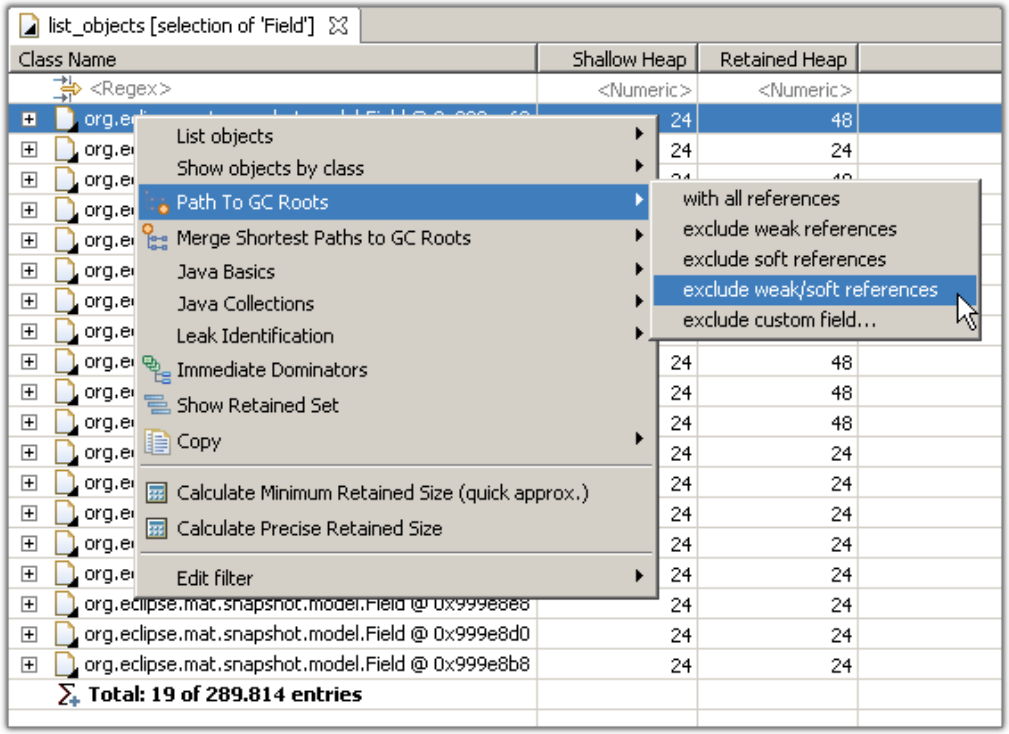
Memory Analyzer (MAT)- Path to GC Roots

- **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.**



Memory Analyzer (MAT)- Path to GC Roots

- **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**



哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

The end

April 26, 2019