

Software Design and Construction

159.251

Software Diagnostics

Amjed Tahir
a.tahir@massey.ac.nz

Original author: Jens Dietrich

References

[PP] Andrew Hunt and David Thomas:
[The Pragmatic Programmer: From Journeyman to Master.](#)
Addison-Wesley, Oct 1999.

Additional Readings

Lars Vogel: Java Debugging with Eclipse.

<http://www.vogella.com/articles/EclipseDebugging/article.html>

Summary


- dealing with exceptions
- debugging
- profiling
- monitoring

The First Computer Bug Ever Found

9/9

0800 Antan started
 1000 " stopped - antan ✓
 1300 (032) MP-MC 1.58264000
 (033) PRO 2 2.130476415
 conv 2.130676415
 Relays 6-2 in 033 failed special speed test
 in relay " " test.
 Relays changed

1100 Started Cosine Tape (Sine check)
 1525 Started Multy Adder Test.

1545  Relay #70 Panel F
 (moth) in relay.

First actual case of bug being found.

1630 Antan started.
 1700 closed down.

Relay 3145
 Relay 3376

Debugging - History

While she [Rear Admiral Grace Murray Hopper - the inventor of COBOL] was working on a Mark II Computer at Harvard University in 1947, her associates discovered a moth stuck in a relay and thereby impeding operation, whereupon she remarked that they were "debugging" the system. Though the term *computer bug* cannot be definitively attributed to Admiral Hopper, she did bring the term into popularity. The remains of the moth can be found in the group's log book at the Smithsonian Institution's National Museum of American History in Washington, D.C.

The Psychology of Debugging

- debugging is problem solving
- focus on finding the bug, not blaming people
- "that's impossible" is not possible
- use a debugger (not **System.out** . . .)
- read error messages
- switch in debug mode: set log level to DEBUG, enable assertions, enable compiler warnings
- try to reproduce the bug in a test case

Debuggability as Design Goal

- it is important to keep **debuggability** in mind when developing software
- using **logging** supports debugging
- since logging is expensive, log levels must be used
- exceptions should be **chained** using causes
- **assertions** and test cases can be used to localise faults
- in general, modular designs are easier to debug

Reading Exceptions

- many bugs are manifested as exceptions or errors
- the general term covering both exceptions and errors is **throwables**
- i.e., in Java, Throwable is the superclass of both Exception and Error (all in java.lang)
- throwables have a message and a **stack trace**
- the stack trace shows the method invocations leading to the throwable
- these invocations have line numbers!
- IDEs like Eclipse turn them into hyperlinks!

Printing Stack Traces

- console:

```
exception.printStackTrace()
```

(default out is **System.err**)

```
exception.printStackTrace(PrintStream out)
```

- logging (log4j):

```
logger.error("a message", exception)
```

```
logger.warn("a message", exception)
```

.. (equivalent methods for other log levels)

Exception Chaining

- often, an exception is caused by another exception
- with exception chaining, a reference to the **causing exception** can be retained
- when printing the stack trace, the stack trace of the parent exception should be printed as well
- this supports better diagnosis
- the “try with resource” syntax provides a more concise syntax for some nested exception patterns (<https://docs.oracle.com/javase/tutorial/essential/exceptions/tryResourceClose.html>)

Example: StudentReader

```
public static List<Student> readStudents(String f) throws StudentDataStoreException {
    File file = new File(f);
    FileInputStream in = null;
    try {
        in = new FileInputStream(file);
        XMLDecoder decoder = new XMLDecoder(in);
        return (List<Student>) decoder.readObject();
    } catch (FileNotFoundException exception) {
        throw new StudentDataStoreException("cannot read students from " + f, exception);
    }
    finally {
        try {
            if (in!=null) in.close();
        } catch (IOException exception) {
            throw new StudentDataStoreException(
                "cannot read students from " + f, exception);
        }
    }
}
```

an IO exception causes this exception - pass the reference to the constructor

Example: StudentReader (ctd)

```
public static void main(String[] args) throws StudentDataStoreException {  
    StudentDataStore.readStudents("students.txt");  
}
```

try to read student
instances from non-xml file

```
Exception in thread "main" nz.ac.massey.sdc.debugging1.StudentDataStoreException: cannot read  
students from students.txt  
    at nz.ac.massey.sdc.debugging1.StudentDataStore.readStudents(StudentDataStore.java:17)  
    at nz.ac.massey.sdc.debugging1.ReadSomeStudents.main(ReadSomeStudents.java:10)
```

```
Caused by: java.io.FileNotFoundException: students.txt (No such file or directory)  
    at java.io.FileInputStream.open(Native Method)  
    at java.io.FileInputStream.<init>(FileInputStream.java:120)  
    at nz.ac.massey.sdc.debugging1.StudentDataStore.readStudents(StudentDataStore.java:13)  
    ... 1 more
```

line numbers

stack trace of causing
exception is appended

Debuggers

- debuggers are tools to debug code
- they support the step-by-step execution of programs, and the suspension of program execution at breakpoints
- debuggers support the inspection of program state (objects within the scope of the execution)
- some debuggers also support the direct manipulation of program state

Breakpoints

- breakpoints are added to lines of source code
- the execution of a program will be interrupted at a breakpoint
- in Eclipse: toggle breakpoints by clicking on left margin in code editor
- often, the program execution should only stop if additional conditions are satisfied

Conditional Breakpoints

- the Eclipse compiler supports the following breakpoint properties:
 - **boolean conditions** - the program only stops if the condition evaluates to true
 - **hit count** - if set to N, the program will only stop when the breakpoint is encountered for the N^{th} time
 - **condition change** - the program will stop if the value of an expression changes

Debugging Scope

- debugging requires source code
- this means that it is not possible to debug into code from external libraries
- however, often this is required to establish where a problem originate from
- possible solution if source code for libraries is available (Eclipse):
 - include sources directly in project
 - create separate project for library, and reference it in the build path settings
 - use Java Source Attachment (in properties of referenced libraries)

Remote Debugging

- many debuggers can debug remote programs
- use case: debug touchscreen interaction in a web application
 - run application on tablet
 - run debugger on desktop
 - example remote (JavaScript) debugger: opera dragonfly

Profiling

- dynamic analysis of programs: run and observe
- measure use of resources, such as
 - memory
 - cpu
 - threads
- generate views and reports to help the software engineer to understand and optimise program performance
- related: memory dumps for post mortem static analysis

Java Profilers (selection)

- VisualVM is part of the JDK and includes a profiler
- JConsole is a similar older tool
- The NetBeans IDE includes a profiler
- JProfiler is an excellent commercial profiler
- The Eclipse Test and Performance Tool Platform (TPTP) includes a profiler plugin - it is however not clear how active this project is
- the Eclipse memory analyser (MAT) is a Eclipse plugin to visualise heap dumps

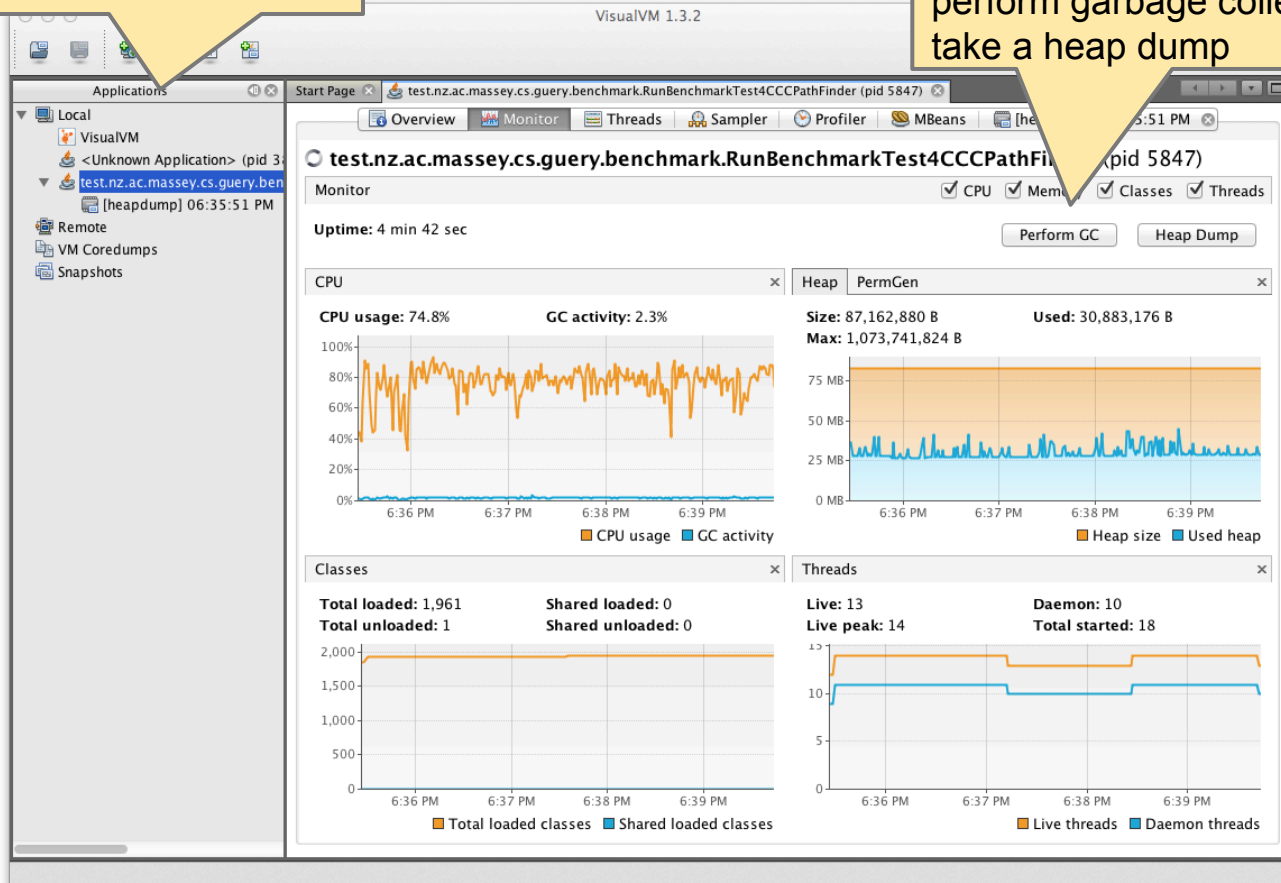
VisualVM

- free, included in the Java Development Kit (JDK), start with `jvisualvm`
- can connect to local and remote JVM applications
- is extensible through plugins
- can interact with JMX objects (discussed later)
- note that VisualVM only connects to running applications - it does not start applications
- easy solution: add a "warmup loop" to application to delay startup (e.g., `Thread.sleep(10000)`)
- other profilers (JProfiler) can directly start applications
- alternative tool: Java Mission Control (start with `jmc`)

VisualVM Monitor

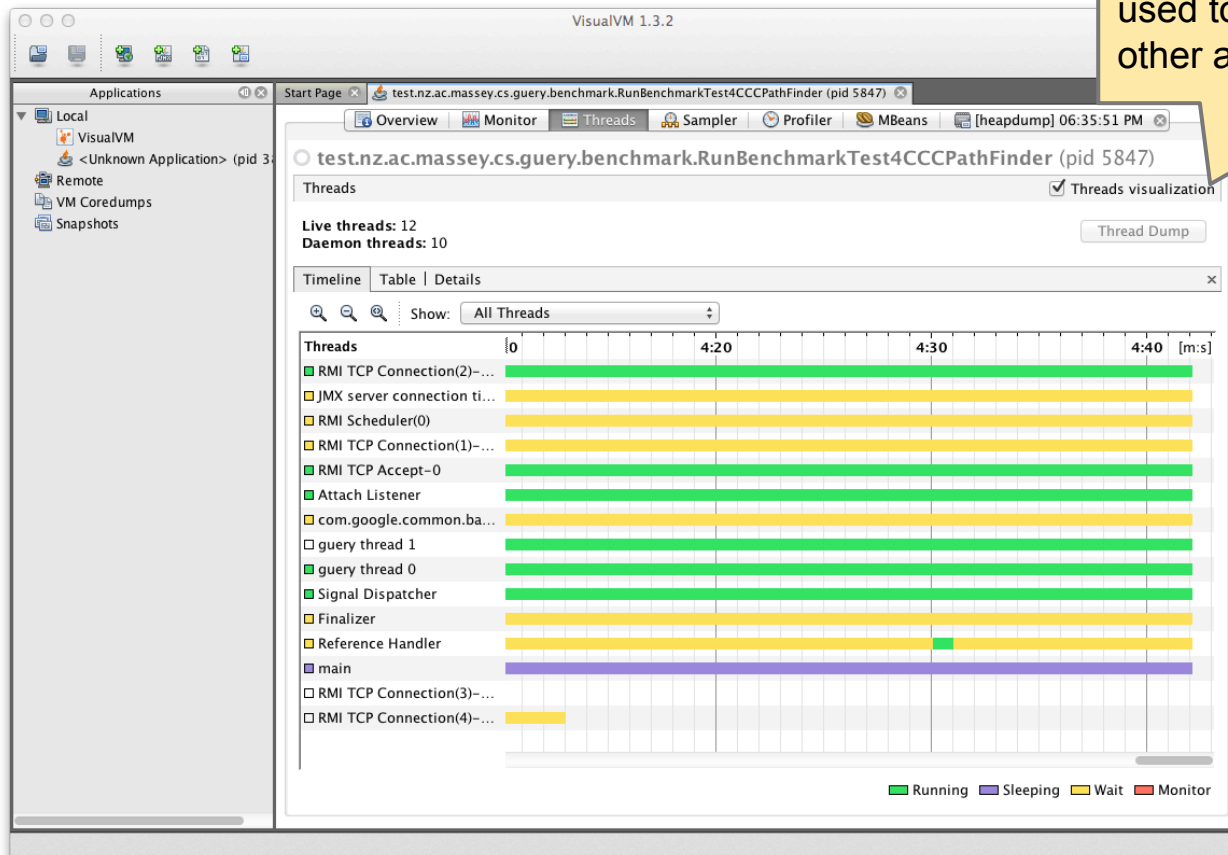
running local and remote JVMs are listed here, to connect VisualVM simply select and click one

overview: basic information about CPU, memory usage, loaded classes and threads, and actions to perform garbage collection and to take a heap dump



Thread Info

application threads with colours encoding status (running, sleeping, wait, monitor): can be used to detect deadlocks and other anomalies



Sampling

- non-invasive technique: the profiler polls the applications periodically by inserting interrupts, and creates reports from these data
- fast but not accurate - generates a statistical approximation
- low overhead, non-invasive - byte code is not changed
- VisualVM supports CPU and memory sampling
- cpu sampling: records how much time was used to execute methods
- memory sampling: records number of instances and bytes used by those instances per class

VisualVM Memory Sampling Example

VisualVM 1.3.2

Applications: Local, Remote, VM Coredumps, Snapshots

test.nz.ac.massey.cs.guery.benchmark.RunBenchmarkTest4CCCPathFinder (pid 12611)

Overview | Monitor | Threads | **Sampler** | Profiler | MBeans

test.nz.ac.massey.cs.guery.benchmark.RunBenchmarkTest4CCCPathFinder (pid 12611)

Sampler

Sample: CPU | Memory | Stop

Status: CPU sampling in progress

CPU samples

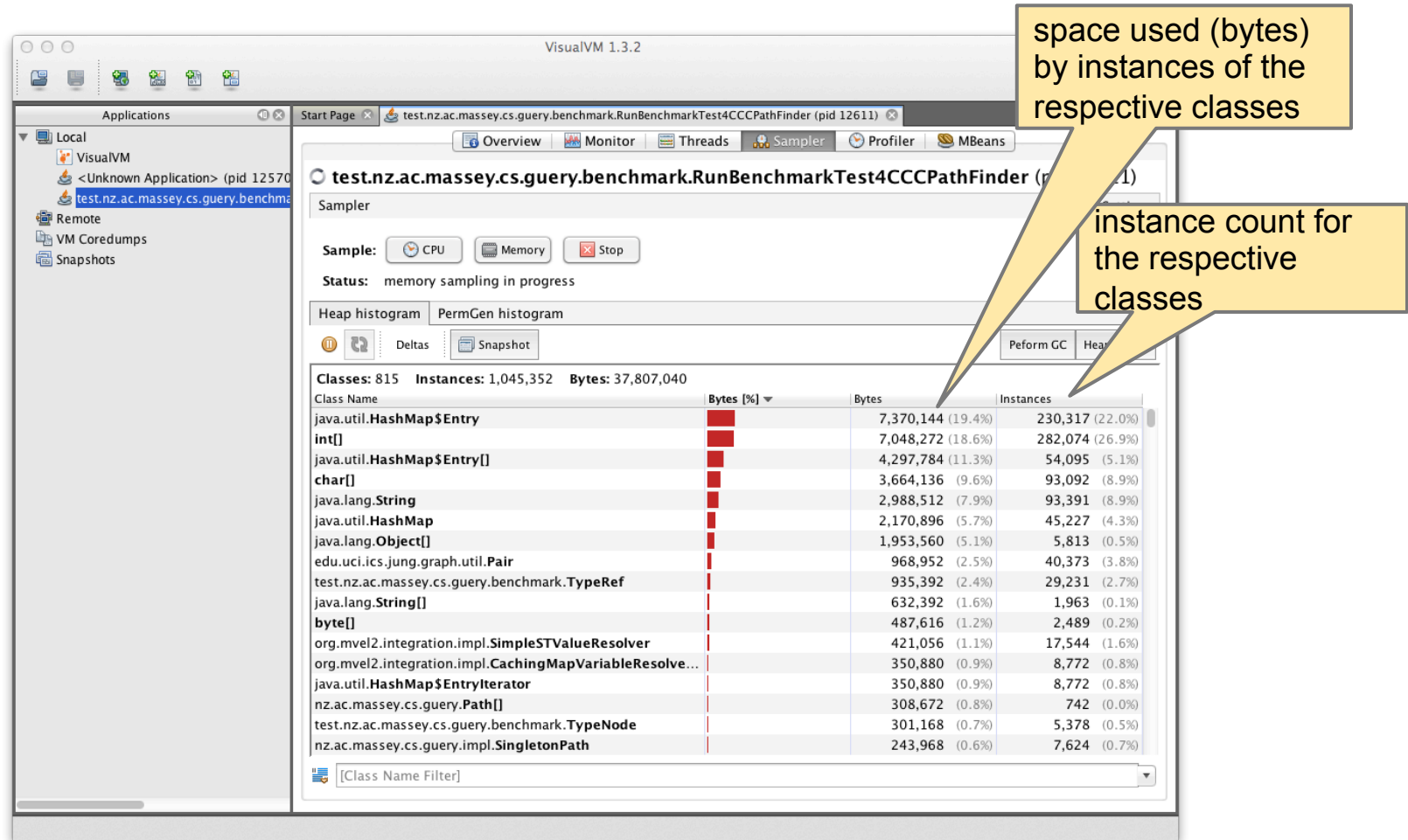
Snapshot

Hot Spots - Method	Self time [%]	Self time	Self time (CPU)
ASMAccessorImpl_11749295501345511794700.getValue ()	0.000 ms (0%)	0.000 ms	0.000 ms
ASMAccessorImpl_2324842291345511794720.getValue ()	0.000 ms (0%)	0.000 ms	0.000 ms
ASMAccessorImpl_3723648201345511794730.getValue ()	0.000 ms (0%)	0.000 ms	0.000 ms
ASMAccessorImpl_5681823821345511794730.getValue ()	0.000 ms (0%)	0.000 ms	0.000 ms
com.google.common.base.internal.Finalizer.run ()	0.000 ms (0%)	0.000 ms	0.000 ms
com.google.common.collect.AbstractIterator.hasNext ()	318 ms (1%)	318 ms	318 ms
com.google.common.collect.AbstractIterator.tryToComputeNext ()	0.000 ms (0%)	0.000 ms	0.000 ms
com.google.common.collect.Iterators\$7.computeNext ()	550 ms (1.7%)	550 ms	550 ms
com.google.common.collect.Iterators\$8.hasNext ()	0.000 ms (0%)	0.000 ms	0.000 ms
com.google.common.collect.Iterators\$8.next ()	0.000 ms (0%)	0.000 ms	0.000 ms
edu.uci.ics.jung.graph.DirectedSparseGraph.containsEdge ()	296 ms (0.9%)	296 ms	296 ms
edu.uci.ics.jung.graph.DirectedSparseGraph.containsVertex ()	277 ms (0.8%)	277 ms	277 ms
edu.uci.ics.jung.graph.DirectedSparseGraph.getDest ()	106 ms (0.3%)	106 ms	106 ms
edu.uci.ics.jung.graph.DirectedSparseGraph.getOutEdges ()	0.000 ms (0%)	0.000 ms	0.000 ms
edu.uci.ics.jung.graph.DirectedSparseGraph.getSource ()	296 ms (0.9%)	296 ms	296 ms
nz.ac.massey.cs.guery.AbstractGraphAdapter.getOutEdges ()	364 ms (1.1%)	364 ms	364 ms
nz.ac.massey.cs.guery.PathConstraint\$1.apply ()	269 ms (0.8%)	269 ms	269 ms
nz.ac.massey.cs.guery.PathConstraint.getPossibleTargets ()	0.000 ms (0%)	0.000 ms	0.000 ms

[Method Name Filter]

program profiled: <http://goo.gl/i3RMY>

VisualVM CPU Sampling Example

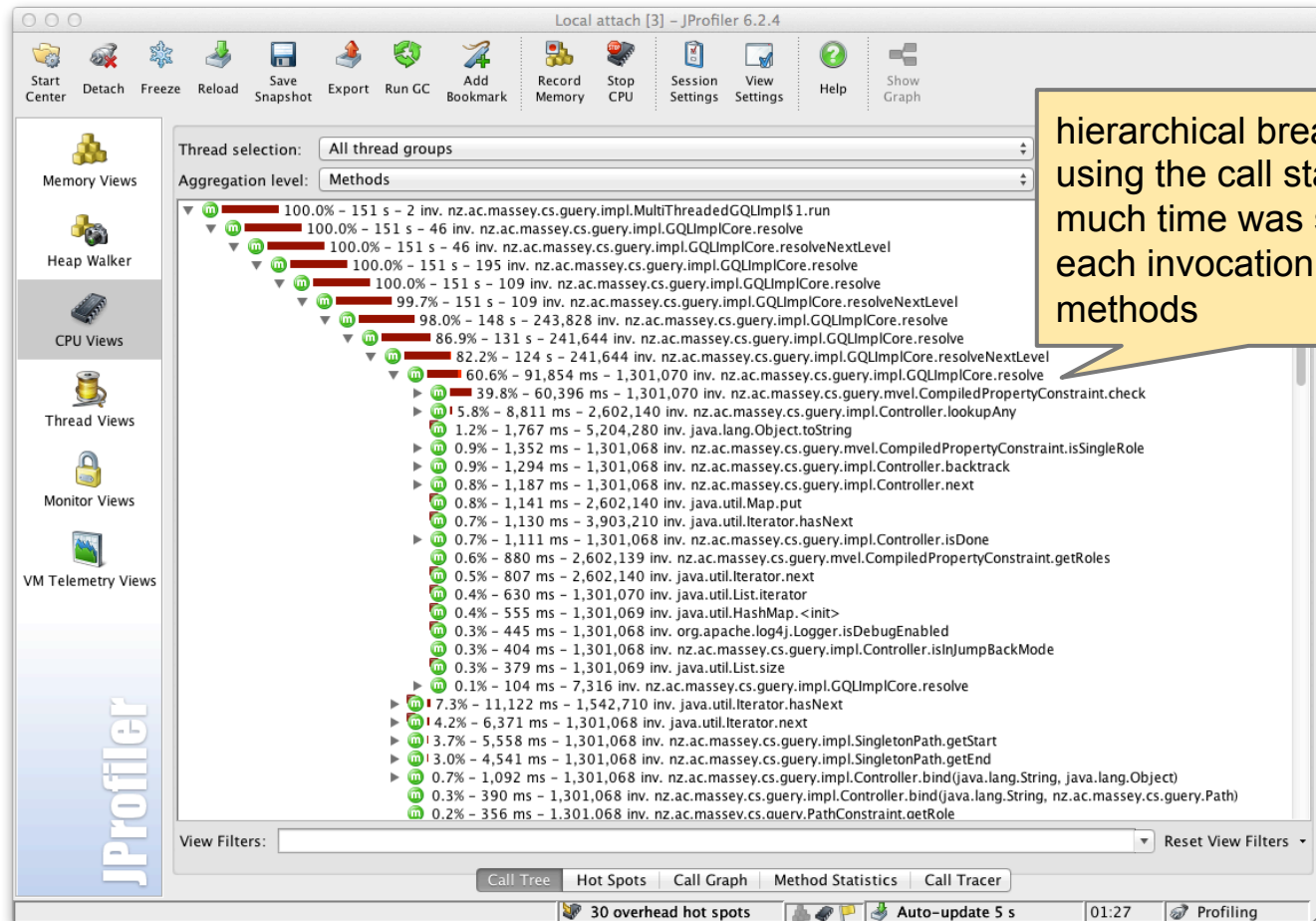


program profiled: <http://goo.gl/i3RMY>

Profiling (vs Sampling)

- invasive technique
- code must be instrumented (probes are inserted) to gather data, i.e. byte code is changed
- this can add significant overhead
- however, profiling can gather more data, and is more precise
- in particular, method invocations can be associated with callees (methods invoking methods), adding more contextual information
- a challenge is to keep profiling transparent - this often fails for large and complex systems, in particular if these systems use instrumentation and custom classloaders (example: application servers)

Profiling View with Call Stack (taken with JProfiler)



JVM Memory Basics

- the JVM divides memory into two sections (simplified): heap and stack
- in the heap part, objects are stored - including the classes themselves and arrays
- the heap is divided into several pools to optimise garbage collections (e.g., permgen pool holds classes - classes are usually not GCed)
- methods and local variables are stored in the stack

Common problems: StackOverflow

- when a method is invoked, a stack frame is created for the method invocation (with values for parameters and local variables)
- when calling methods recursively, too many frames are created and the stack is running out of space
- this creates a **java.lang.StackOverflowError**
- note that this is an error, not an exception - the JVM will exit!

Fixing StackOverflows

- rewrite logic to use non-recursive algorithms
- check that recursion has a termination condition
- increase stack size with JVM option **-Xss<size>**
 - the default size is 512k
 - **-Xss1m** increases this to 1MB

Common Problems: OutOfMemory

- the garbage collector (GC) will try to free memory by removing unreferenced objects
- the GC runs in the background, applications usually do not directly interact with it
- when too many objects are created that cannot be garbage collected, a **java.lang.OutOfMemoryError** is created
- this is also called a **memory leak**
- note that this is an error, not an exception - the JVM will exit!

Fixing OutOfMemories

- check program logic
- reuse objects instead of creating new ones (use object pools)
- reduce the number of static references: classes are usually not GCed, and therefore objects they reference are protected
- use weak or soft references (usually not used directly, but via data structures such as `java.util.WeakHashMap`)
- increase heap size with JVM option `-Xmx<size>`
 - `-Xmx1g` increases this to 1GB

Heap Dumps

- heap dumps can be taken by VisualVM and other profiling tools and saved for later static analysis
- static analysis - program is not running
- in VisualVM: HeapDump button in most views
- VisualVM can load heap dumps for analysis