

# Eins für alles?

## Monitoring und Profiling mit VisualVM

Thomas Much

[thomas@muchsoft.com](mailto:thomas@muchsoft.com)  
[www.muchsoft.com](http://www.muchsoft.com)



# Referent



- Dipl.-Inform. Thomas Much
- IT-Architekt, Softwareentwickler & Projektcoach
  - Java seit 1996, C++ seit 1993, davor ObjectPascal, Eiffel ...
- Programmiertrainer
  - seit 1997
- Autor
  - "Java für Mac OS X"  
Galileo Computing 2005  
630 Seiten, ISBN 3-89842-447-2



# Agenda

- Monitoring und Profiling
- JDK-Tools
- VisualVM
- VisualVM mit GlassFish (Sun Application Server)
- Fazit

# Monitoring und Profiling

# Software-Probleme (1)

- Arten
  - dauerhafte Performance-Probleme
  - über die Zeit anwachsende Performance-Probleme
  - sporadische Performance-Probleme
  - Überlast
  - gelegentliche Fehler
  - regelmäßige Fehler

## Software-Probleme (2)

- Gründe
  - Anwendungsfehler
  - HTTP-Session
  - Datenbankprobleme, schlechtes SQL
  - Netzwerküberlastung
  - ...
  
- Abhilfe: Monitoring und Profiling

# Monitoring und Profiling

- Monitoring
    - beschreibt Erfassung von Zuständen, die Beobachtung, Überwachung oder Steuerung eines Vorgangs (Prozesses) mittels technischer Hilfsmittel
  - Profiling
    - Einsatz von Programmwerkzeugen (Tools), die das Laufzeitverhalten von Software analysieren
- Monitoring entdeckt Probleme,  
Profiling geht ihnen auf den Grund.

# Performance-Profiling

Wie kann Performance-Profiling durchgeführt werden?

- Eigene Zeitmessung
  - `System.currentTimeMillis()`, `System.nanoTime()`
- JVM-Schnittstellen (JVMPi, JVMTI)
- Bytecode-Instrumentierung



# JVMPI / JVMTI

- JVMPI
    - JVM Profiler Interface
    - seit Java 1.1 dabei
    - aber immer nur "experimentell" mit zahlreichen Problemen
    - seit Java 5.0 deprecated, seit Java 6 nicht mehr dabei
  - JVMTI
    - JVM Tool Interface
    - seit Java 5.0 dabei
    - löst JVMPI (und Debugger-API JVMDI) ab
- zahlreiche Tools bauen darauf auf

# JDK-Tools

# JDK-Tools

- **jinfo, jps**
  - Prozess-Informationen und -Ids
- **jstack**
  - Prozess-Stacktrace
- **hprof**
  - Zeit- und Speicher-Profiling
- **jmap, jhat**
  - Speicher-Dumps, Histogramme, Auswertung als Web-Server
- **jstat, jstatd**
- **jconsole**

## jstat / jstatd

- `jstat -gcutil pid ms count` ↵

oder remote: `pid@host:port`

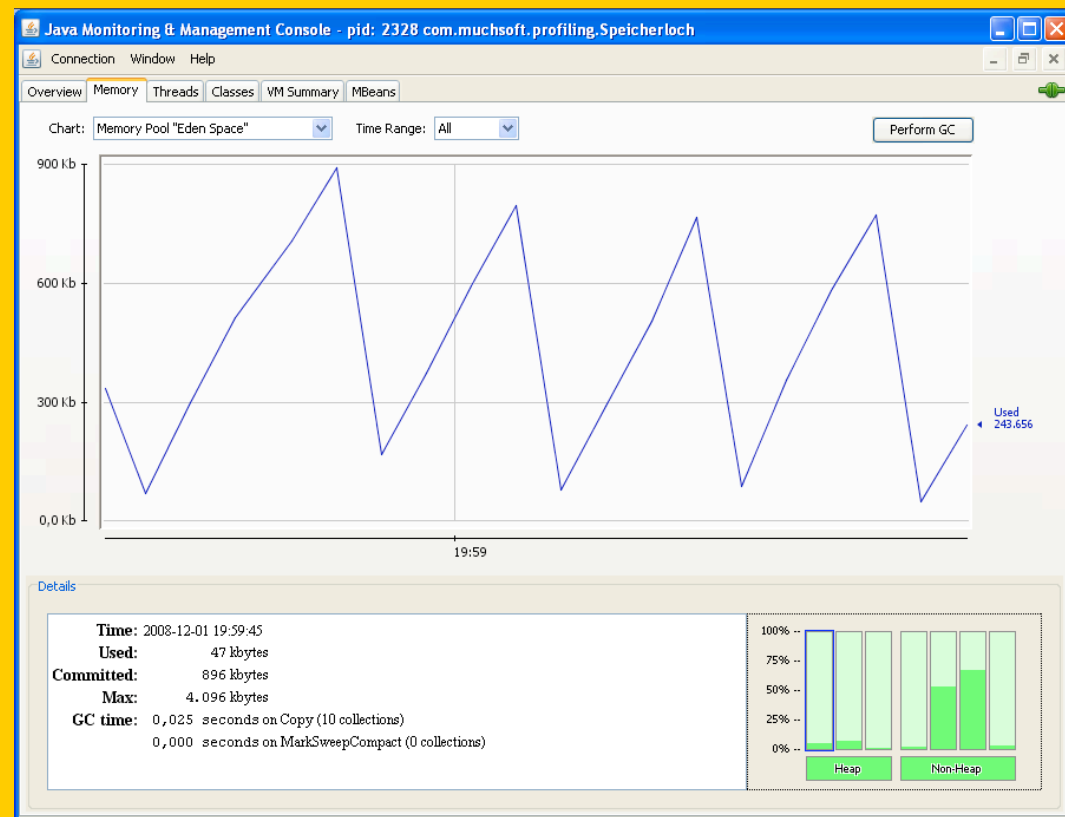
- `start rmiregistry` ↵  
`jstatd`  
`-J-Djava.security.policy=jstatd.all.policy` ↵

*jstatd.all.policy*

```
grant codebase "file:${java.home}/../lib/tools.jar" {  
    permission java.security.AllPermission;  
};
```

# JConsole

- seit JDK 5.0 fester Bestandteil
- Aufruf mit `jconsole`



# VisualVM

# VisualVM (1)

- Download von `<https://visualvm.dev.java.net/>`
- seit JDK 6u7 fester Bestandteil (!)
- NetBeans-RCP-Anwendung
  
- benötigt Java 6 als Runtime
  
- kann Anwendungen mit Java 1.4, 5.0 und 6 untersuchen
  - je neuer die Ziel-JVM ist, desto mehr Informationen stehen zur Verfügung



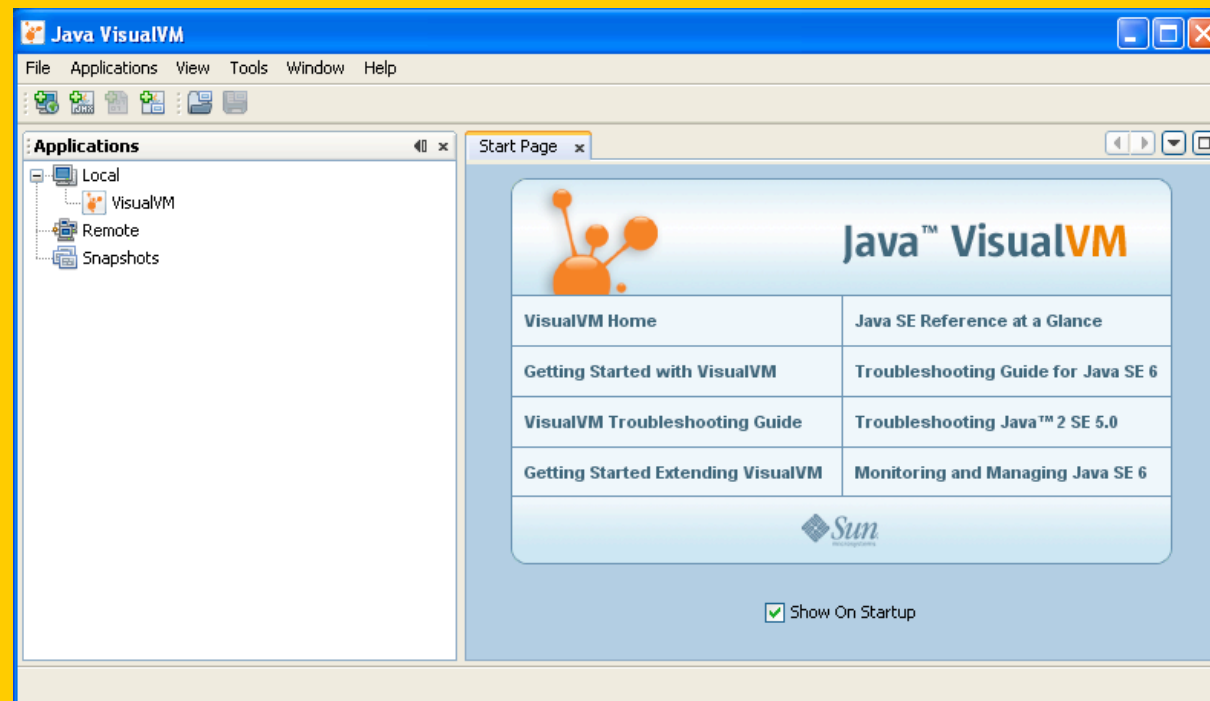
## VisualVM (2)

Feature	JDK 1.4.2 local/remote	JDK 5.0 local/remote	JDK 6 local	JDK 6 remote
Overview	✓	✓	✓	✓
System Properties			✓	
Monitor	✓	✓	✓	✓
Threads		✓	✓	✓
Profiler			✓	
Thread Dump			✓	
Heap Dump			✓	
Heap Dump on OOME			✓	
MBean Browser		✓	✓	✓
JConsole Wrapper		✓	✓	✓

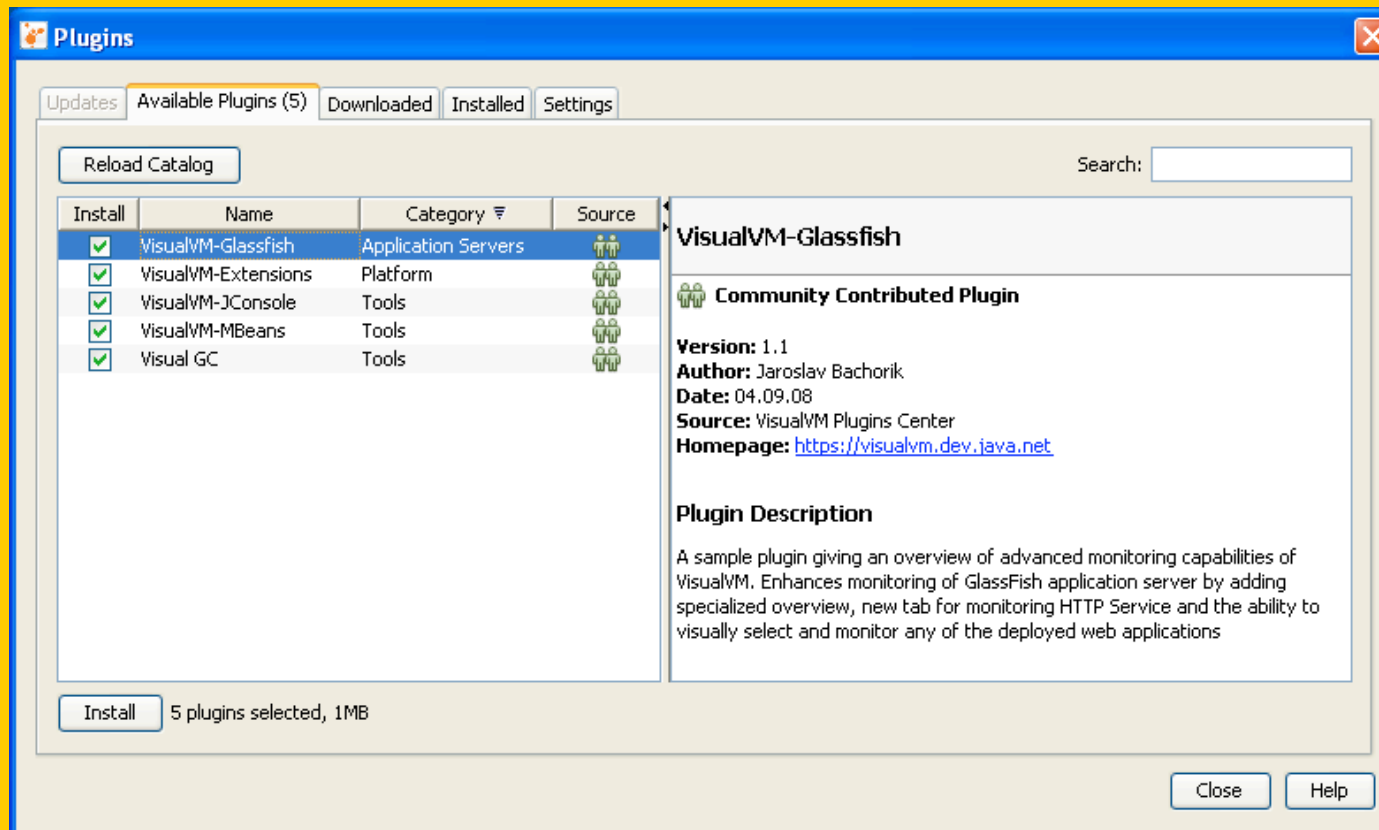


## VisualVM (3)

- Aufruf mit `jvisualvm`
- beim ersten Start erfolgt eine Kalibrierung



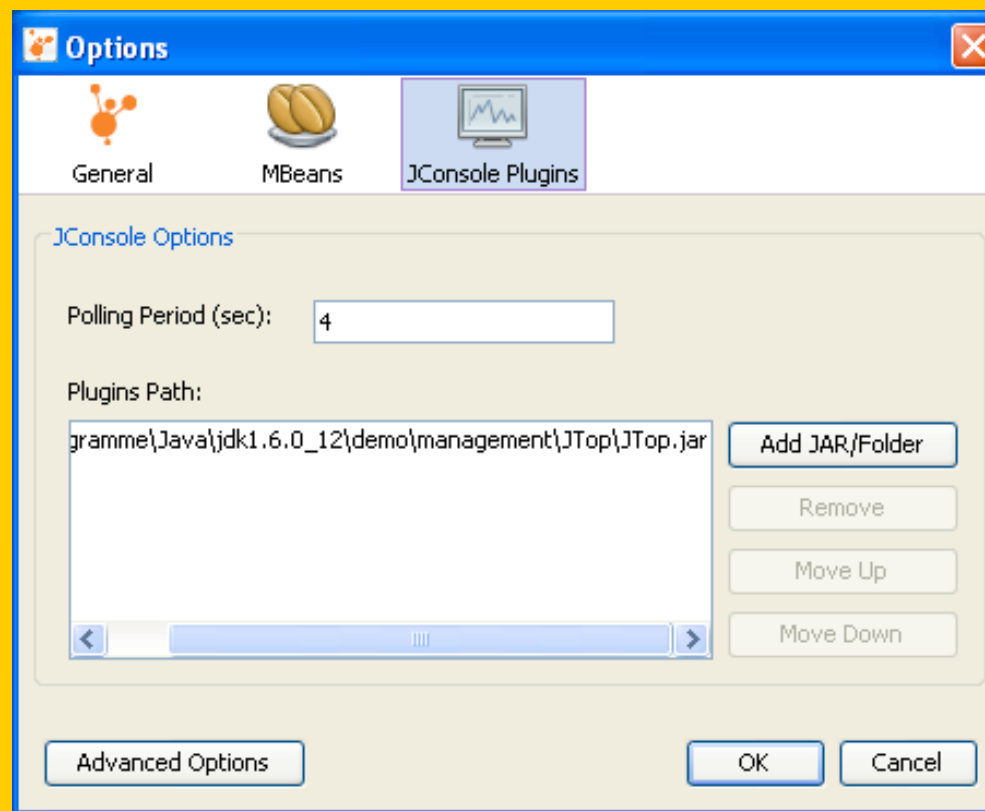
# VisualVM: Plugins



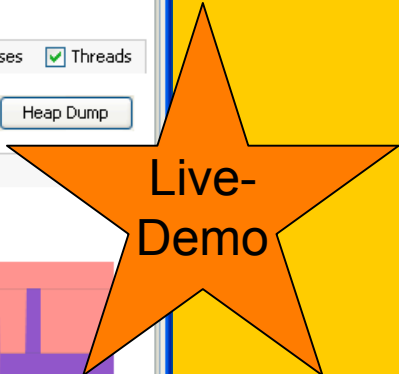
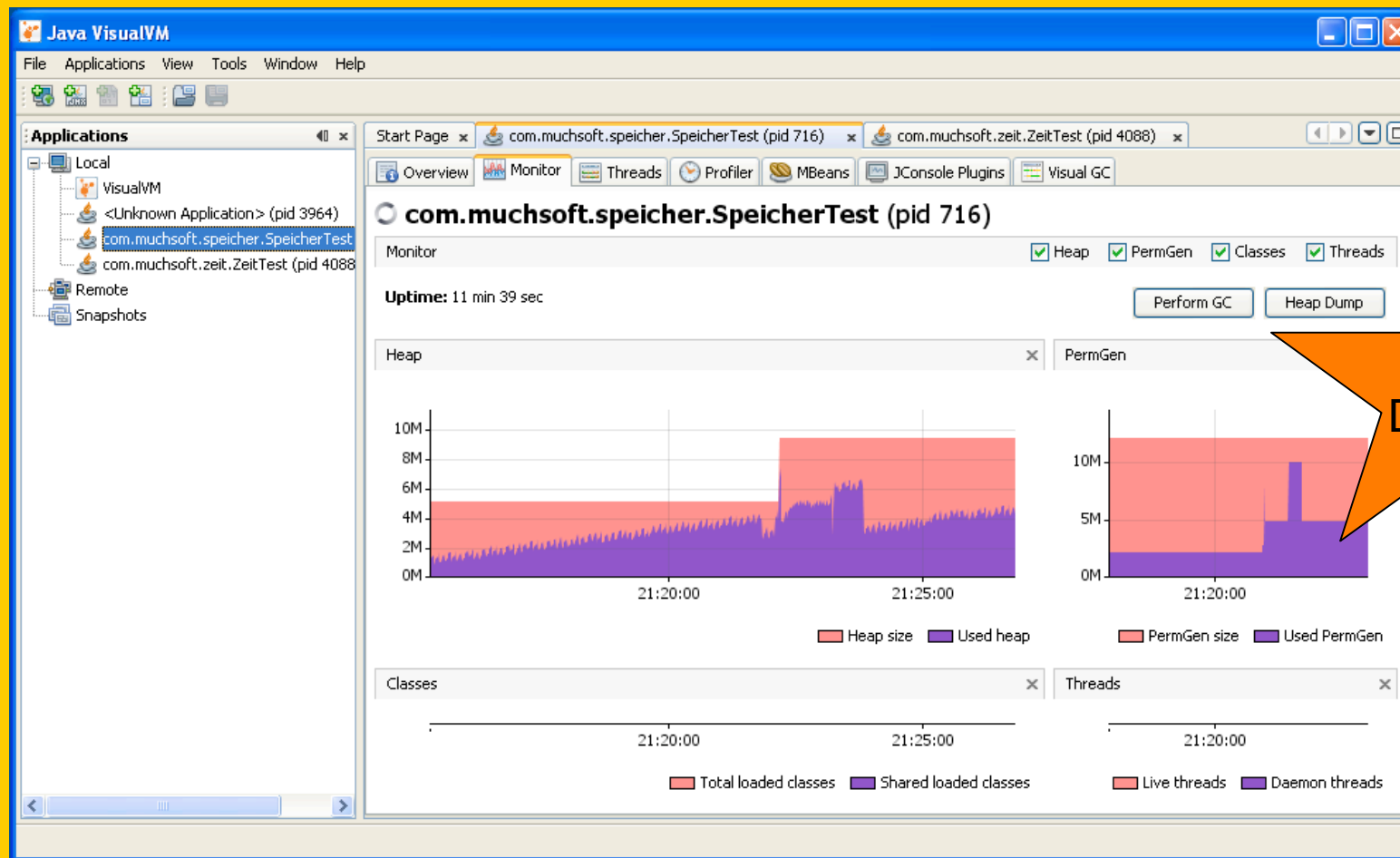
→ am besten alle Plugins installieren, vor allem VisualGC

# VisualVM: Optionen / JTop

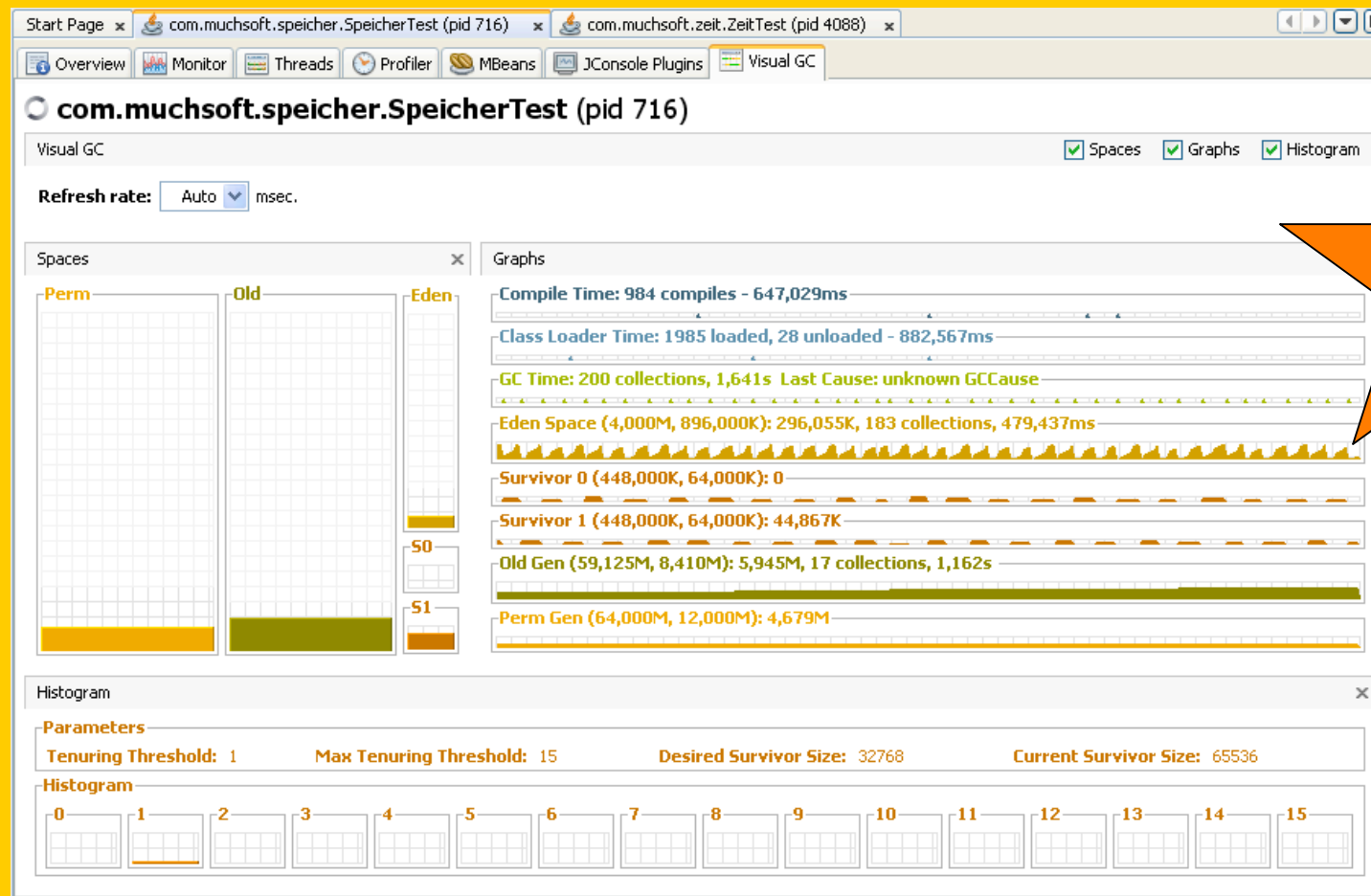
- mit dem JConsole-Plugin kann z.B. JTop installiert werden:



# VisualVM: Monitoring



# VisualVM: Visual GC



# VisualVM: Speicherlecks finden

Java VisualVM

File Applications View Tools Window Help

Applications

- Local
  - VisualVM
    - <Unknown Application> (pid 3964)
    - com.muchsoft.speicher.SpeicherTest
      - [heapdump] 09:27:33 PM
      - com.muchsoft.zeit.ZeitTest (pid 4088)
  - Remote
  - Snapshots

Start Page x com.muchsoft.speicher.SpeicherTest (pid 716) x com.muchsoft.zeit.ZeitTest (pid 4088) x

Overview Monitor Threads Profiler MBeans JConsole Plugins Visual GC [heapdump] 09:27:33 PM x

## com.muchsoft.speicher.SpeicherTest (pid 716)

Heap Dump

Summary Classes Instances

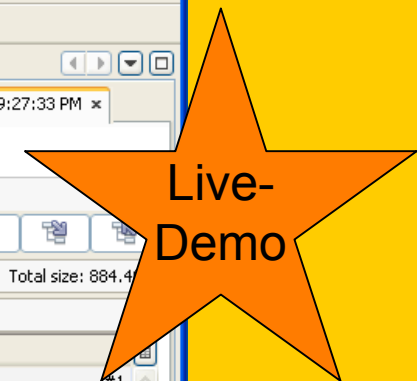
com.muchsoft.speicher.Speicherloch Instances: 73.700 | Instance size: 12 | Total size: 884.4

Instance	Field	Type	Value
<500 instances>	this	Speicherloch	#1
	jetzt	Date	#1

Field	Type	Value
this	Speicherloch	#1
	Object[]	#735 (88.256 items)

Array type | Object type | Primitive type | Static field | GC Root | Loop

Show Instance  
Show Nearest GC Root



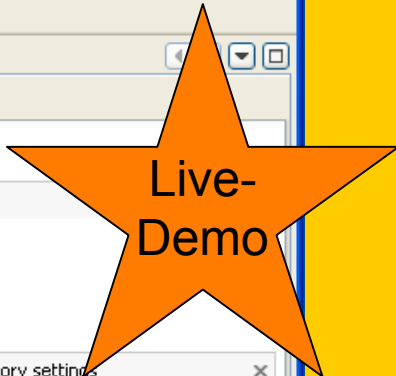
# VisualVM: Profiling

The screenshot shows the Java VisualVM Profiler interface. The main window displays the Profiler tab for the application `com.muchsoft.zeit.ZeitTest (pid 4088)`. The Profiler section shows the profile set to CPU, with a status of "Profiling running (24 methods instrumented)".

The Profiling results section shows a table of Hot Spots:

Hot Spots - Method	Self time [%]	Self time	Invoc...
<code>com.muchsoft.zeit.Langsam.langsam ()</code>	88,7%	192517 ms	192
<code>com.muchsoft.zeit.Mittel.mittel ()</code>	9,9%	21405 ms	193
<code>com.muchsoft.zeit.Schnell.schnell ()</code>	1,4%	3078 ms	192

The CPU settings section shows the "Start profiling from classes" field set to `com.muchsoft.zeit.**`. The "Profile new Runnables" checkbox is checked. The "Profile only classes" radio button is selected, and the "Do not profile classes" field is empty. The "Restore Defaults" button is visible at the bottom right of the CPU settings panel.



# VisualVM: Memory-Snapshots vergleichen

Java VisualVM

File Applications View Tools Window Help

Applications

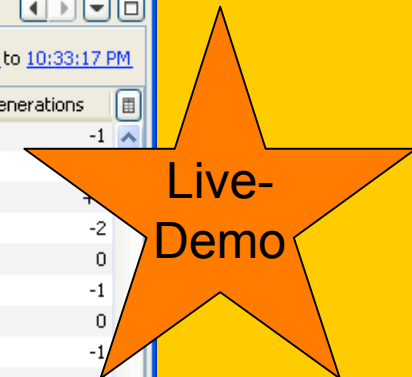
- Local
  - VisualVM
    - <Unknown Application> (pid 2548)
    - com.muchsoft.profiling.ThreadSleepTest (pid 2548)
  - Remote
  - Snapshots

... GC com.muchsoft.profiling.ThreadSleepTest (pid 2548) x Liveness Comparison x

Comparison of 10:32:33 PM to 10:33:17 PM

Class Name - Live Allocated Objects	Live Bytes	Live Bytes	Live Objects	Generations
java.io.ObjectStreamClass\$FieldReflector...	-168 B	-168 B	-7	-1
java.lang.String[]	-168 B	-168 B	-7	-1
java.lang.Long	-176 B	-176 B	-11	-1
sun.rmi.transport.ConnectionInputStream	-192 B	-192 B	-3	-2
java.util.TreeMap\$KeyIterator	-192 B	-192 B	-6	0
sun.rmi.transport.ConnectionOutputStream	-216 B	-216 B	-3	-1
java.io.ObjectInputStream\$HandleTable...	-224 B	-224 B	-4	0
java.lang.reflect.Method[]	-304 B	-304 B	-1	-1
java.lang.StringBuilder	-432 B	-432 B	-27	-1
java.lang.reflect.Field	-648 B	-648 B	-9	+1
java.lang.Class[]	-720 B	-720 B	-41	+2
int[]	-760 B	-760 B	-8	-1
java.io.ObjectStreamClass	-1.152 B	-1.152 B	-12	+4
java.lang.String	-1.680 B	-1.680 B	-70	+8
java.io.ObjectStreamClass\$WeakClassKey	-1.856 B	-1.856 B	-58	-1
java.lang.reflect.Method	-3.360 B	-3.360 B	-42	0
byte[]	-3.824 B	-3.824 B	-21	+5
char[]	-6.608 B	-6.608 B	-83	+8

[Class Name Filter]



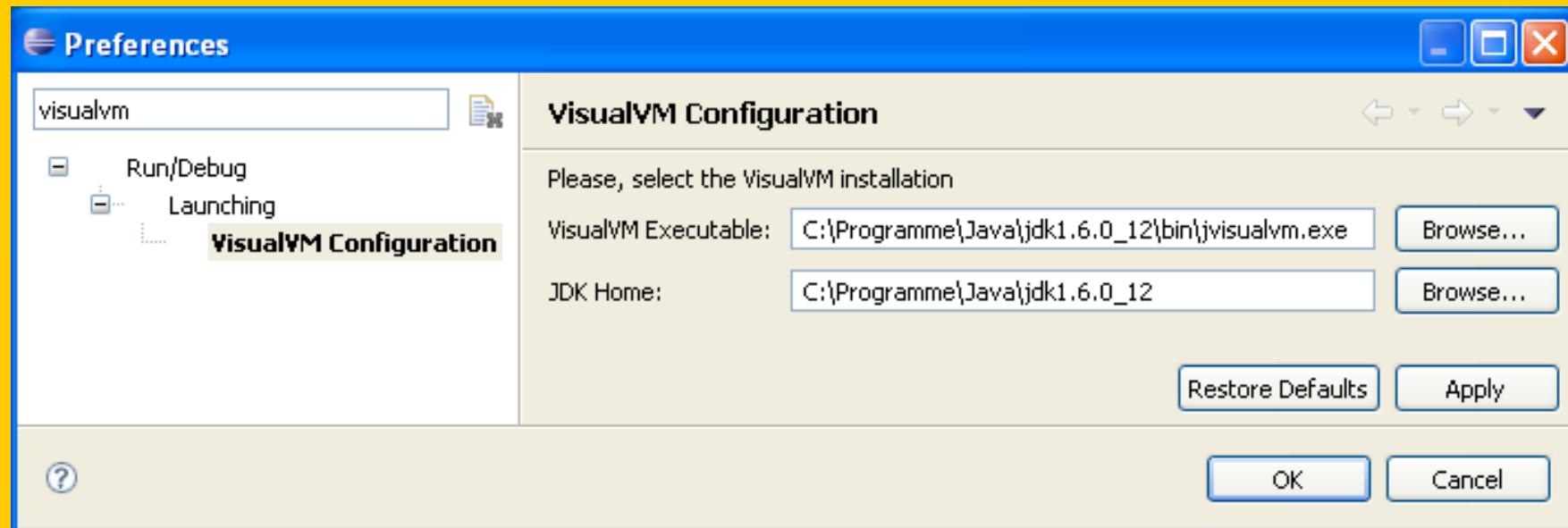


## VisualVM: Probleme

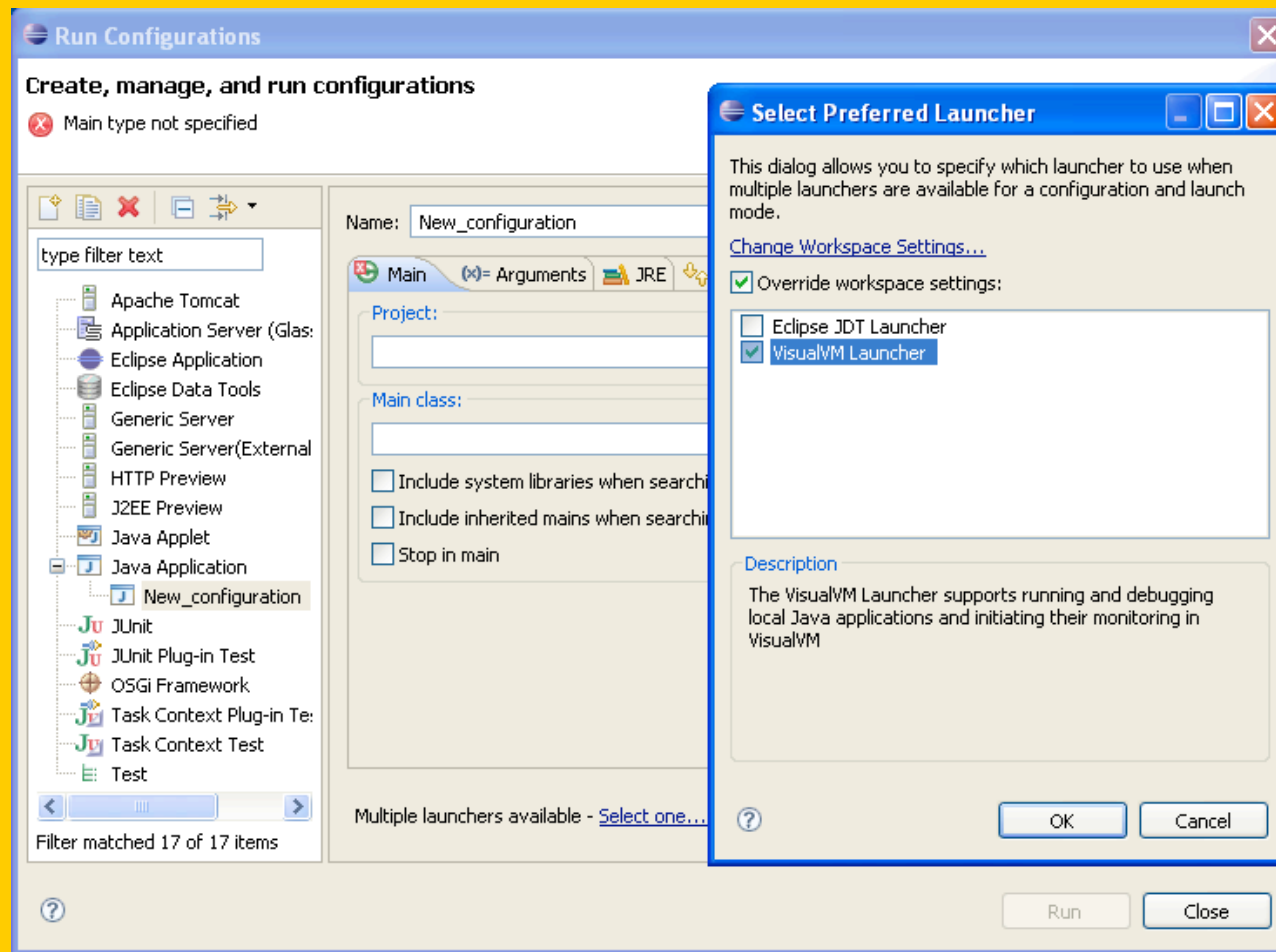
- VisualVM ist noch relativ neu, es gibt noch ein paar Probleme...  
siehe <https://visualvm.dev.java.net/troubleshooting.html>
- beispielsweise kann das Profiling lokale Ziel-JVMs zum Absturz bringen
  - Class-Sharing deaktivieren
  - Ziel-JVM mit `-Xshare:off` starten

# Eclipse: VisualVM-Plugin

- Download von  
<<https://visualvm.dev.java.net/eclipse-launcher.html>>
- Konfiguration:



# Eclipse: Anwendung+VisualVM starten

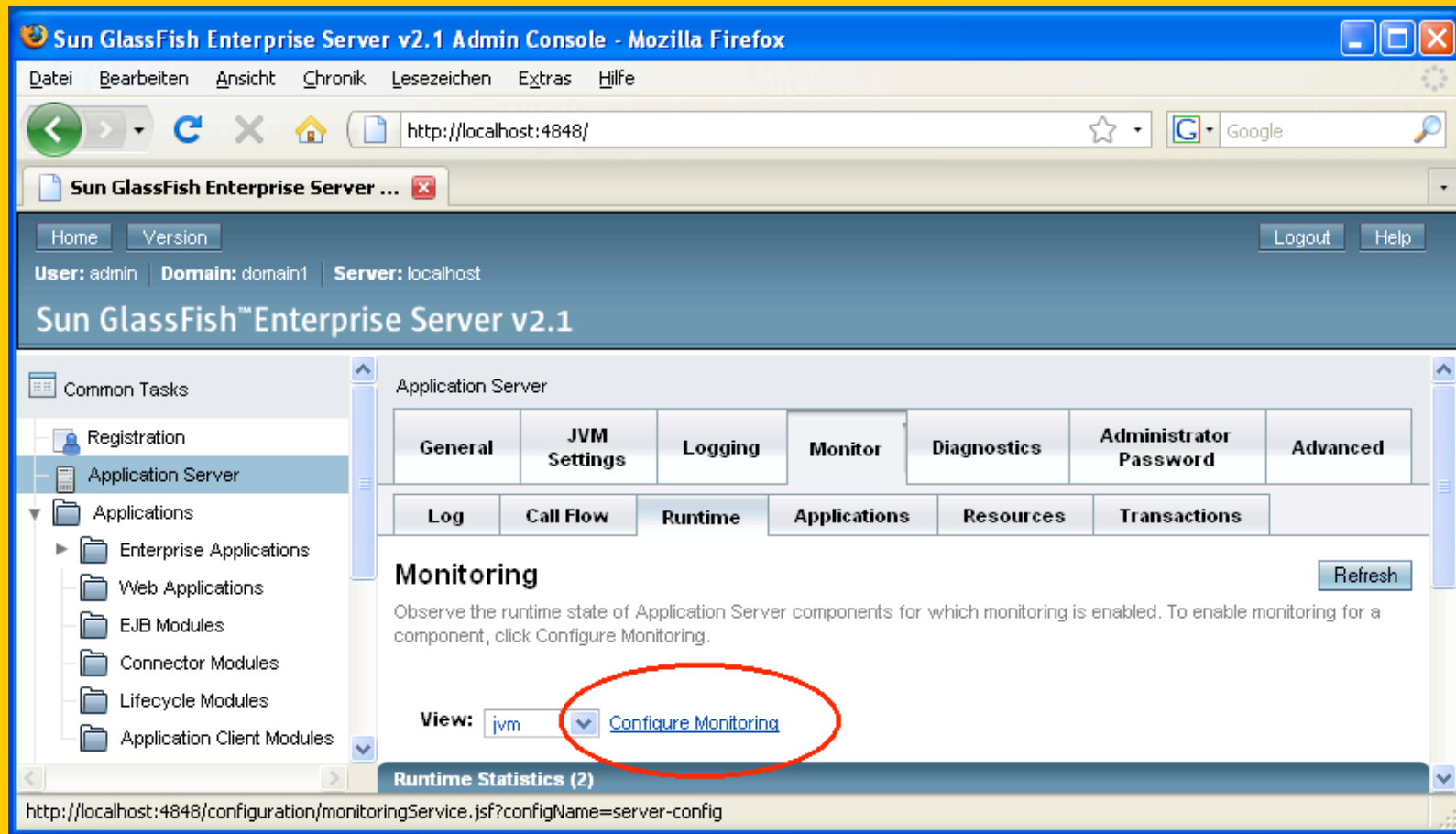


# VisualVM mit GlassFish

# VisualVM mit GlassFish

- GlassFish starten
- für Remote-Zugriff: `jstatd` starten
  - auf dem Server muss der RMI-Port (1099) offen sein
- VisualVM starten
  - GlassFish-Plugin verwenden
  - Monitoring in GlassFish aktivieren

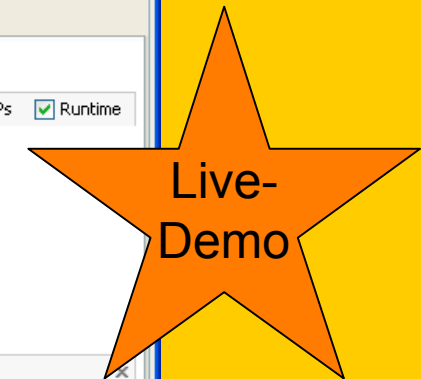
# GlassFish: Monitoring aktivieren



# GlassFish: Monitoring mit VisualVM

The screenshot shows the Java VisualVM interface. On the left, a tree view shows the application structure under 'Local' > 'VisualVM' > 'GlassFish/SJSAS (pid 3608)'. The 'webapp' is selected. The main window displays the 'webapp' overview, including the context path, document base, and working directory. Below this, there are two line graphs: 'Sessions Active' and 'Sessions Total', and a 'JSRs' graph. At the bottom, a table shows the performance of various servlets and web services.

Name	ErrorCount	ServiceTime	RequestCount	ProcessingTime
default	0	0.0	5	0
jsp	0	0.0	0	0
a1	0	2.7857142857142856	28	78
a2	0	385.5555555555554	9	3470



# VisualVM: Fazit



## VisualVM: Fazit

- **Positiv:**
  - einfach zu installieren und einzusetzen
  - Open Source, kostenfrei, fester Bestandteil vom JDK
  - soll laut Sun zur Standard-Management-Anwendung von Java-Applikationen werden
  - viel Potential!
- **Negativ:**
  - noch nicht 99%ig stabil
  - kein Remote-Profiling
  - keine Instrumentierung (Probes o.ä.)
  - hierfür TPTP oder kommerzielle Produkte erforderlich

# Fragen?

## Vielen Dank!

