# **Memory Profiling using Visual VM**

What type of profiling is most important?

Clear answer: memory profiling!

The speed of your application typically is something that you "feel" throughout your whole development process. You get to know, when your application runs slow by daily testing it during development. So, the CPU performance of your application should not be something to get too much surprised, when changing from test environment to production environment.

With memory, things are different: you do not "feel" the memory footprint of your application during development:

- You do not get to know memory leaks because you typically restart your system too often.
- You have so much memory locally available that you typically do not run into memory problems while testing the system in a single-user-way.

Consequence: memory profiling is a MUST during development. Switching on a productive system without having checked the memory behavior of the application before is not a too good idea.

# Challenges

#### What is the typical memory footprint per user?

At a certain point of time your system administrator will ask you: how much memory should I reserve for your application? The response is quite simple:

- You need to tell, what the fix memory size is that is used by your system
- You need to tell, what the typical memory size per user is

Typically, it does not make sense to do some in-depth-profiling (e.g. on object instance basis) to gain these responses. It's better to start some simple heuristic approach by observing the Java heap space.

Measuring is something very simple: you can access the information also through the "System"-class in Java or you can use a tool like Visual VM. The only thing you have to make sure is to start the garbage collector before measuring in order to get reproducible and reliable results.

#### Are there memory leaks?

This is something which is more challenging. Making sure that your application does not have memory leaks.

There is no other way than in-depth-profiling to find this out. The memory heap space is not accurate enough to find out if your application leaks or not. Of course you may gain some feeling, but nothing more.

So, here you need explicit tooling - luckily there is a free tool "Visual VM" that is made available by Oracle that helps you a lot.

Once again: when your application's memory leaks, then the only way to find details is by measuring - not by thinking!!! The tool will tell you the truth about what happens inside your application.

# Setting up the Visual VM environment

You can download Visual VM from <a href="http://visualvm.java.net/">http://visualvm.java.net/</a>.

After starting Visual VM you see a tree on the left showing the avaiable Java processes on the right:



When having started the server part of CaptainCasa Enterprise Client it should be visible as "Tomcat" in the left tree.

We experienced sometimes problems, the result being that the list of Java processed did not show "our" Tomcat. If so, check the following:

- Start the server from a "real" Java-JDK, not from a Java-JRE. By default CaptainCasa only comes with a Java-JRE...
- Start the Visual VM as administrator (right mouse click on visualvm.exe, select "run as administrator".

## Observing the heap space

Double click on the Tomcat instance withing the tree on the left, then on the right select the "Monitor"-tab.

Elie Applications View Lools Window Help         Image: Constraint of the state of the stat	🔐 VisualVM 1.3.4		i x
Applications #       Image: Start Page: Monitor Imeads @ Sampler O Profiler         Image: Control of Update (pid 3328)       Image: Control of Update (pid 3328)         Image: Control of Total (pid 1124)       Image: Control of Total (pid 1124)         Image: Control of Total (pid 1124)       Image: Control of Total (pid 1124)         Image: Control of Total (pid 1124)       Image: Control of Total (pid 1124)         Image: Control of Total (pid 1124)       Image: Control of Total (pid 1124)         Image: Control of Total (pid 1124)       Image: Control of Total (pid 1124)         Image: Control of Total (pid 1124)       Image: Control of Total (pid 1124)         Image: Control of Total (pid 1124)       Image: Control of Total (pid 1124)         Image: Control of Total (pid 1124)       Image: Control of Total (pid 1124)         Image: Control of Total (pid 1124)       Image: Control of Total (pid 1124)         Image: Control of Total (pid 1124)       Image: Control of Total (pid 1124)         Image: Control of Total (pid 1124)       Image: Control of Total (pid 124)         Image: Control of Total (pid 124)       Image: Control of Total (pid 124)         Image: Control of Total (pid 124)       Image: Control of Total (pid 124)         Image: Control of Total (pid 124)       Image: Control of Total (pid 124)         Image: Control of Total (pid 124)       Image: Controtal (pid 124)         Image:	File Applications View Tools Window Help		
Applications III     StartPage III (2010)     Icol           Icol </th <th></th> <th></th> <th></th>			
Image: Single Color       Image: Single Color<	Applications %	Start Page 🕷 💥 Tomcat (pid 1124) 🕷	
Edges (pd 338)     Grg.hcqldb.Serve (pd 756)     Grmcat (pd 1124)     Monitor     Uptime: 9 min 24 sec     Perform GC     Heap Dump     Heap PermGen     x     Stapshots     Used: 35.895.056 8     Hax: 558.70.912 8     J     J     Monitor     J     J     Stapshots     Stapshots     J     Stapshots     Stapshots     J     Stapshots     Stapshots     Stapshots     J     Stapshots     J     Stapshots     Stapshots     Stapshots     J     Stapshots     Stapshots     J     Stapshots	VisualVM	G Overview We Monitor E Threads W Sampler (V) Profiler	
Monitor (a) disconse (a) discon	Eclipse (pid 3328)	O Tomcat (pid 1124)	
Uptime:         9 min 24 sec         Perform GC         Heap Dump           Heap Demole         x           Size:         124.348.800 B         Used:         35.895.056 B           Heap Demole         x           Size:         124.348.800 B         Used:         35.895.056 B           How Perform GC         Heap Dump         x           Size:         124.348.800 B         Used:         35.895.056 B           100 MB         98%         98%         100 MB         100 MB           100 MB	org.hsqldb.Server (pid 7636)	Monitor	hreads
Heap         PermGen         x           Size: 134.348.800 B         Used: 35.895.056 B         Used: 35.895.056 B           125 MB         98%         98%         0           100 MB         98%         0	Tomcat (pid 1124)	Uptime: 9 min 24 sec Perform GC Heap D	ump
Size: 134.348.800 B       Used: 35.895.056 B         Max: 536.870.912 B       98%         100 MB       98%         75 MB       98%         50 MB       98%         25 MB       98%	- Snapshots	Heap PermGen	×
Max: 536.870.912 B         115 MB       98%       0000         100 MB       0000		Size: 134.348.800 B Used: 35.895.056 B	
115 MB-     98%     98%     98%       100 MB-     98%     98%       75 MB-     100     100       50 MB-     100     100       25 MB-     100     100		Max: 536.870.912 B	
100 MB 10		125 MB-	
		98%	
75 MB-     25 MB-     25 MB-     20 MB- <th></th> <th>100 MB-</th> <th></th>		100 MB-	
75 MB- 50 MB- 25 MB- 25 MB-			
50 MB- 25 MB- 25 MB-		75 MB-	
25 MB-		50 MB-	
25 MB-			
		25 MB-	
0 MB 1443120 1443325 1443340 1443345 1443340 1443346		0 MB 14:33:20 14:33:25 14:33:30 14:33:35 14:33:40	14:33:45
Heap size 🛄 Used heap		📕 Heap size 🔳 Use	d heap
		16	

You can see the allocated memory of the Java VM - both the currently used one and the current maximum. Only the bottom blue line is the one to be interested in - the height of the orange one is defined by the "-Xmx...m -Xms...m" options that you pass to the Java process when starting "java.exe".

Now start a CaptainCasa client to work against your Tomcat and begin to user your application. ...and: do not panic...:



By just doing some simple things the memory will dramatically increase (e.g. the used heap space gains 25 MBytes). - This is just normal when working with Java...!

You have to click "Perform GC" (garbace collector) within the tool in order to obtain comparable results.

# Finding memory leaks

#### Starting the profiler

For finding memory leaks you have to do some in-depth-profiling. In Visual VM there is a corresponding "Profiler"-tab:

The typical first thing to do - before starting the profiling session - is to open the settings

area and take a look at the "Memory Settings":

	Settings
Memory Stop	
	CPU settings Memory settings ×
	Profile object allocations
	Profile object allocations and GC
	Track every 10 (a) object allocations
	Record allocations stack traces

By default the counting of object instance if done for every  $10^{th}$  object allocation - in order to save runtime performance. - But: we want to explicitly measure memory on single instance level, so we recommend to count every instance. Set the corresponding number to "1":

		💟 Settings
Memory Stop		
	CPU settings Memory settings	x
	Profile object allocations	
	Profile object allocations and GC	
	Track every 1 📄 object allocations	
	Record allocations stack traces	

Now click the "Memory" button and - wait! ...wait! ...wait!

ừ VisualVM 1.3.4		
File Applications View Tools Window Help		
: 🐸 📖 : 📽 🏡 🏫 😤		
Applications ×	Start Page 🛛 🎇 Tomcat (pid 1124) 🕷	
Applications       Image: Construction of the second	Start Page       Image: Tomcat (pid 1124)         Image: Overview       Monitor         Image: Tomcat (pid 1124)         Profiler         Profile:       OCPU         Image: Tomcat (pid 1124)         Status:       profiling inactive         Profiling results	Ettings
		3

You may observe the Tomcat's DOS box to see that something is going on:

2 zstart - Verknüpfung	J
*** probably it has been unloaded recently *** Profiler engine warning: class sun.reflect.GeneratedSerializationConstructor *** Requested classloader: sun.reflect.DelegatingClassLoader@99c4cc, its class =	
*** probably it has been unloaded recently *** Profiler engine warning: class sun.reflect.GeneratedConstructorAccessor6 tha	
*** Requested classloader: sun.reflect.DelegatingClassLoader@136c55, its class = *** Profiler engine warning: target VM cannot load class to instrument sun.refle	
**** probably it has been unloaded recently **** Profiler engine warning: class sun.reflect.GeneratedConstructorAccessor7 tha	
*** Profiler engine warning: target VM cannot load class to instrument sun.refle *** probably it has been unloaded recently	
Profiler Agent: Redefining 100 classes at idx 0, out of total 3462 Profiler Agent: Redefining 100 classes at idx 100, out of total 3462	
Profiler Agent: Redefining 100 classes at 1dx 200, out of total 3462 Profiler Agent: Redefining 100 classes at idx 300, out of total 3462 Profiler Agent: Redefining 100 classes at idx 400, out of total 3462	
Profiler Agent: Redefining 100 classes at idx 500, out of total 3462 Profiler Agent: Redefining 100 classes at idx 600, out of total 3462	
Profiler Agent: Redefining 100 classes at idx 200, out of total 3462 Profiler Agent: Redefining 100 classes at idx 800, out of total 3462 Profiler Agent: Redefining 100 classes at idx 900, out of total 3462	
Profiler Agent: Redefining 100 classes at idx 1000, out of total 3462 Profiler Agent: Redefining 100 classes at idx 1100, out of total 3462	
Profiler Agent: Redefining 100 classes at idx 1200, out of total 3462 Profiler Agent: Redefining 100 classes at idx 1300, out of total 3462 Pupfiler Agent: Redefining 100 classes at idx 1400, out of total 3462	
Profiler Agent: Redefining 100 classes at idx 1500, out of total 3462 Profiler Agent: Redefining 100 classes at idx 1600, out of total 3462	
Profiler Agent: Redefining 100 classes at idx 1700, out of total 3462 Profiler Agent: Redefining 100 classes at idx 1800, out of total 3462	
Profiler Agent: Redefining 100 classes at idx 2000, out of total 3462 Profiler Agent: Redefining 100 classes at idx 2000, out of total 3462 Profiler Agent: Redefining 100 classes at idx 2100, out of total 3462	
Profiler Agent: Redefining 100 classes at idx 2200, out of total 3462 Profiler Agent: Redefining 100 classes at idx 2300, out of total 3462	
Profiler Hgent: Redefining 100 classes at idx 2400, out of total 3462	
	J

Visual VM is instrumenting the classes that are loaded within the Java process. This means the byte code of the classes that is loaded is updated - and corresponding hooks are inserted into the code that take over the counting of object instances.

We are not sure, if this is correct, ...but: we made the experience that you should not touch Visual VM during the instrumentation phase. Otherwise we sometimes waited ages for a feed back and sometimes had to shut down Visual VM.

After pressing the garbage collector icon you should see a list of the currently loaded objects:

VisualVM 1.3.4							x
File Applications View Tools Window Help							
: 🖴 📖 : 🐝 🏡 🛍 😫							
Applications %	Start Page 🕺 🎇 Tomcat (pid 1124) 🕺					4	
	🐻 Overview 🗰 Monitor 🚍 Threads 🔬 Sampler 🕑 Profiler						
Edinas (aid 2229)	O Tomcat (pid 1124)						
eclipse (pid 5526)							
and baldb Server (pid 7636)	Profiler					E 5	ettings
······································	Profile: OCPU Memory Stop						
😓 org.echt.client.ztest.Test6Dialog (pid 5944)							
em Remote	Status: profiling running (7.752 classes instrumented, tracking each 10	0th object)					
Snapshots	Profiling regulte						
	🍭 💫 🛅 🕪   🚍 Snapshot   🔯 🔯						
	Class Name - Live Allocated Objects	Live Bytes 💌	Live Bytes		Live Objects	Generations	
	java.lang.reflect. Field		288 B	(11%)	4 (6,6%)		1 🔺
	char[]		288 B	(11%)	3 (4,9%)		1
	java.lang.reflect. Method		240 B	(9,1%)	3 (4,9%)		1
	java.lang.reflect.Constructor		128 B	(4,9%)	2 (3,3%)		1
	java.net.SocksSocketImpl		120 B	(4,6%)	1 (1,6%)		1
	java.lang. Thread		112 B	(4,3%)	1 (1,6%)		1
	java.lang.Class[]		104 B	(4%)	6 (9,8%)		1
	java.util.HashMap\$Entry[]		80 B	(3%)	1 (1,6%)		1
	java.lang.ThreadLocal\$ThreadLocalMap\$Entry[]		80 B	(3%)	1 (1,6%)		1
	short[]		72 B	(2,7%)	1 (1,6%)		1
	java.util.Hashtable\$Entry[]		56 B	(2,1%)	1 (1,6%)		1
	java.lang. String		48 B	(1,8%)	2 (3,3%)		1 -
	[Class Name Filter]						-
	Live Results Class History						
	L						

Tpyically we switch off the auto-refresh (icon on the very left), because observing the instances regularly without collecting garbage does not make too much sense.

### Typical procedure

The absolute number of object instances that is shown in the list is typically not too much relevant. You want to find deltas (leaks!).

In order to find the deltas you first have to think about some application procedure which you start, process and close - and where you expect that after running the procedure all of your allocated objects are released.

This may be a whole session (e.g. start client, do something, close client) or may be some activity within your running application (e.g. from workplace: start workpage, process workpage, close workpage).

The way of measuring is:

- (Switch off the automatic refreshing.)
- In Visual VM start the garbage collector, by pressing the corresponding icon.
- In Visual VM reset the results by pressing the "Reset collected results" icon. Now all delta counters are reset.
- Perform your application procedure one or several times.
- In Visual VM press start the garbage collector again.

You now will see a list of objects that are currently loaded. You can filter the list by class name:

🔐 VisualVM 1.3.4							
File Applications View Tools Window Help							
: 🔛 📖 : 😪 🛞 🐘 🐏							
Applications %	art Page al 👷 Tomcat (pid 1124) al 👘 👘 Profiler						
VisualVM     Scipse (pid 3328)     Gon, echt.clent.page.PageBrowser (pid 4872)     Gon, Isqidb.Server (pid 7636)     Work (pid 126)     Gon and (pid 1126)     Gon and (pid 1126)							
	Profiler					Settings	
	Profile:         CPU         Memory         Stop           Status:         profiling running (7.767 dasses instrumented, tracking ea	ich 10th object)					
	Profiling results						
	🍓 🖏 📷 🕪 📄 Snapshot 🛛 🔯						
	Class Name - Live Allocated Objects	Live Bytes	Live Bytes		Live Objects	Generations 👻 🔳	
	workplace.DemoBudget\$BudgetLine	1	400 B	(0,5%)	5 (0,2%)	1 ^	
	workplace. Dispatcher	Í	112 B	(0,1%)	1 (0%)	1	
	org.ecInt.workplace.Workpage		80 B	(0,1%)	1 (0%)	1	
	org.ecInt.workplace.WorkpageContainer\$WorkpageInfo		40 B	(0%)	1 (0%)	1	
	workplace. DemoBudget		40 B	(0%)	1 (0%)	1	
	workplace. DemoBase\$MyParser		0 B	(0%)	0 (0%)	0	
	org.ecInt.workplace.IWorkpageLifecycleListener[]		0 B	(0%)	0 (0%)	0	
	🚛 workplace.					- - -	
	Live Results Class History						
						3	

In the columns on the right you see the number of instances that do exist, and you see the delta from the point of time on when you last time hit the "Reset results" icon.

Now you can see which objects are not cleaned up correctly. But: you only see the number of objects - you do not see where the objects are referenced (and as consequence are NOT garbage collected).

### Using the heap dump for single instance analysis

This is the time now for starting the heap dump tool:

The heap dump is collected by pressing the right mouse button on the Tomcat Java process:

	(pid 7050)
sorg.ecInt.clier	Open
Remote Snapshots	Sample Profile
	Thread Dump Heap Dump
	Profiler Snapshot Application Snapshot
	Enable Heap Dump on OOME
	Properties

In the heap dump view switch to the "Classes"-view and filter the class that you are interested in:

ど VisualVM 1.3.4					• ×	
File Applications View Tools Window Help						
Applications %	Start Page 🛛 🎇 Tomcat (pid 1124) 🕺					
- Local	🙀 Overview 🙀 Monitor 🖼 Threads 🔐 Sampler 🚫 Profiler 📰 [heapdur	np] 03:39:10 PM ×				
	C Tomont (rid 1124)	L				
Eclipse (pid 3328)						
<ul> <li>g org.ecirit.cient.page.Pagebrowser (pid 4672)</li> <li>d org.hsaldb.Server (pid 7636)</li> </ul>	Heap Dump					
⊕ · 🞇 Tomcat (pid 1124)	🔶 🕂 🚯 Summary 🔂 Classes] 💿 Instances 🔘 OQL Console			0	12	
🚽 🎂 org.ecInt.client.ztest.Test6Dialog (pid 5944)	S Classes		Compare wit	h another heap	dump ×	
Remote	Class Name	Class Name Instances [%]  Instances [%]				
	workplace. DemoBudget\$BudgetLine		<b>46</b> (0%)	3.220	(0%) 🔺	
	workplace. DemoIntro\$ItemAction		3 (0%)	36	(0%)	
	workplace. DemoBudget		1 (0%)	36	(0%)	
	workplace. DemoIntro		1 (0%)	48	(0%)	
	workplace.DemoBase\$MyParser		0 (0%)	0	(0%)	
	workplace. DemoLabelVertical		0 (0%)	0	(0%)	
	workplace. DemoCombofield		0 (0%)	0	(0%)	
	workplace. DemoMinispread		0 (0%)	0	(0%)	
	workplace. DemoBase		0 (0%)	0	(0%)	
	workplace. DemoLabel		0 (0%)	0	(0%)	
	workplace. DemoMinispread\$MyRow		0 (0%)	0	(0%)	
	workplace. DemoLabelSequence		0 (0%)	0	(0%)	
	workplace. DemoHelloWorld		0 (0%)	0	(0%)	
	workplace. DemoBasicControls		0 (0%)	0	(0%)	
	workplace. DemoFlush		0 (0%)	0	(0%)	
					Ψ.	
	workplace.Demo				- 🕲	
					2	

Select one class with the right mouse button and view the instances:

🔐 VisualVM 1.3.4						X
File Applications View Tools Window Help						
Applications %	Start Page 🛛 💥 Tomcat (pi	d 1124) ⊗			4	
🖓 🖓 Local	🐻 Overview 🚮 Monitor 🗮	Threads	Sampler 🚫 Profiler 🔚 [heapo	dump] 03:39:10 PM ×		
	O Townert (wid 11)	24)				
Eclipse (pid 3328)	O Tomcat (pid 11.	24)				
org.echt.client.page.PageBrowser (pid 4872)	Heap Dump					
org.hsqldb.Server (pid 7636)     Org.act (pid 1124)	🔶 🔶 🚯 Summary 🔝 C	lasses 🛛 😔 Ins	tances 🔘 OQL Console			18
🦢 🎂 org.echt.client.ztest.Test6Dialog (pid 5944)	workplace.DemoBudge	t\$BudgetLine	e Ins	tances: 46   Instance size: 70   Total size	: 3.220   Compute Retains	ed Sizes
En Constant	5 Instances	×	😤 Fields			×
,time Snapshots	Instance	m	Field	Turce	Value	
			o this	Press Predect the dest time	Value	
	<b>○</b> #1	Â	this of thirth	DemoBudgetsBudgetLine		#0 ^
	0 #2	_		beeless	(	#1 E
	0 #4	-	m_enabled	Doolean		aise
	0 #5	=	m_dreds	Arraydiet	#*	.4/9
	0 #5	_	m parentNode	DemoBudget\$BudgetLine	#3	#1
	0 #7		m requestfocus	long		0 -
	0 #8		References	long		~
	0 #9		References			<u> </u>
			Field	Туре	Value	
	Q #11		<ul> <li>this</li> </ul>	DemoBudget\$BudgetLine		#6 🔺
	0 #12		🗄 🔍 🕒 m_parentNode	DemoBudget\$BudgetLine		#46
	♥ #13		⊕ • ● m_parentNode	DemoBudget\$BudgetLine		#43 E
	0 #14		🗄 😔 m_parentNode	DemoBudget\$BudgetLine		#44
			🗈 🥥 m_parentNode	DemoBudget\$BudgetLine		#45
	● #16		⊕ ⊚ m_parentNode	DemoBudget\$BudgetLine		#40
	A #17		🖬 🎯 m parentNode	DemoBudaet\$BudaetLine		#41 👻
			Array type	Object type	atic field   🖻 GC Root   🕃	Loop
						3

You now can check each instance - and e.g. the objects that are referencing the instance.

### Visual VM Summary

When it comes to memory profiling Visual VM provides functions:

- You can observe the heap size (=> Monitoring)
- You can count object instances and can simply observer changes (=> Profiling)
- You can drill down to the level of single instances (=> Dump)