Java memory profiler user guide

Robert Olofsson, robert.olofsson@khelekore.org

May 25, 2006

"We should forget about small efficiencies, say about 97% of the time: Premature optimization is the root of all evil."

Donald Knuth

"Make it work before you make it work fast."

Jon Bentley

"I feel the need – the need for speed."

Tom Cruise in Top Gun

1 Getting and installing jmp

JMP can be downloaded from the jmp web page found at `http://www.khelekore.org/jmp/`. Both source and binary versions of jmp are available. For the latest and greatest the source install is recomended. Source installation can however be a bit problematic, especially under windows so the binaries can be nice.

1.1 Source instalation

After downloading jmp unpack it and compile it. A standard installation where everything needed is found should look like this.

```
tar -xvzf jmp-0.50.2.tar.gz
cd jmp-0.50.2
./configure
make
make install[1]
```

JMP uses the GTK+/2.x libraries unless the –enable-noui switch is added to configure.

GTK+/2.x is available from `http://www.gtk.org/` if you don't have it already (most linux distributions have the needed libraries from start). If you have the choice of runtime and development packages, you need to install both to build jmp.

If you are running windows I would suggest that you grab one of the prebuilt binaries for jmp.dll. If you want to compile jmp under windows you should first install cygwin and a GTK+-runtime, then you should be able to install jmp as above.

1.2 Contacting the jmp project

The best way to contact the users or developers of jmp is probably to use the public mailing lists.

- `jmp-devel@khelekore.org` for patches, bug reports, feature requests etc.
- `jmp-user@khelekore.org` for usage questions, success stories and other jmp related questions.

2 Testing jmp

2.1 Unix systems

For the jvm to be able to find the libjmp.so[2] you either need to place it under some directory where the linker will look (/usr/lib and /usr/local/lib are probably searched) or add the directory where libjmp.so is placed to the LD_LIBRARY_PATH. If you run bash you do:

```
export LD_LIBRARY_PATH=/path/to/jmp/lib:$LD_LIBRARY_PATH
```

2.2 Windows system

Make sure that both jmp.dll and the GTK+ runtimes can be found in your PATH. Also if you run windows 95/98/ME you really should:

```
set PANGO_WIN32_NO_UNISCRIBE=1
```

Since the text handling in GTK+ for thoose platforms will work much faster[3] with this option.

If you can not change the PATH, you can try to put jmp in the WINDOWS or WINNT folder or even the directory you are starting java from

[1]make install may require root privileges

[2]If you run Linux and/or Solaris the name will be libjmp.so, if you happen to run HP-UX, AIX or some other dialect the name may be something else, the name is not really important as long as the shared library can be found by the jvm.

[3]It may even be that jmp is unusable without this option.

2.3 JMP help and first examples

Ok, so you have compiled and installed jmp, how do you test it? The first test is to run the

```
java -Xrunjmp:help
```

If the output you get starts with:

```
jmp/0.50.2 initializing...
```

Followed by a lot of other text (actually a quick help for jmp) you seem to have jmp working. The version number may of course vary, 0.50.2 was the latest stable release when this document was written.

If jmp seems to work I suggest that you find some small test program and start it under jmp to familiarize yourself with the extra controls and windows jmp provides. Try something simple such as the hello world program:

```
java -Xrunjmp HelloWorld
```

If your program exits quickly you may only see the jmp windows flash before closing, if so try adding something like a

```
System.in.read ();
```

to your java code before if finishes.

3 Profiling

Before you start to profile you need to figure out what it is you want to know. Running jmp with everything turned on will waste a lot of time. Profiling is a time consuming operation. Basically there are three available options when profiling:

- Profiling objects to find memory leaks and find causes for heavy memory usage
- Profiling methods to find out where your program spends time
- Inspect threads to find out why your program is blocked

Even if you start jmp with a minimal set of profiling options turned on, they can be enabled when your program has reached a state suitable for profiling, as is done in example 1.

4 Startup options

JMP can take a number of startup options. The most important ones are:

help which will make jmp show a short description of the options available.

nogui which will make jmp run without any user interface. Can be useful for automated unit tests.

dumpdir="directory name" which will tell jmp where to write the data and heap files it outputs.

dumptimer="seconds" which will tell jmp to dump its data at a fixed intervall.

filter=[[+-][filtertype:]][name] adds an initial filter for "name" It is possible to give several filter to include more than one package. If you want several filters you have to write them as multiple options. See section 5.3 for more information of filters. An example is:

"filter=package1,filter=some.package,filter=-package:some.package.sub".

Use "+" for and inclusive filter, use "-" for an exclude filter. "+" is the default and may be left out.

The available filtertypes are "class", "package", "recursive" and "all".

The name is either the class name (for filtertype "class"), a package name (for filtertype "package" and recursive") or any free text (for the filtertype "all").

Filters apply in the ordering given on the command line.

threadtime will make jmp use thread time instead of absolute time. This option is currently only available under linux.

noobjects will make jmp start with object profiling off.

nomethods will make jmp start with method profiling off.

nomonitors will make jmp start with monitor profiling off.

allocfollowsfilter will make jmp group object allocations into filtered methods.

simulator will tell jmp to skip some of the initialization that requires a fully functional jvm. Used for testing.

Several options can be given at once and should be comma separated.

4.1 Examples

Here are a few examples of how jmp can be run for different profiling tasks. If you actually ***read***[4] the help text and understand the examples given here your profiling will be easier and *much* faster

1. java -Xrunjmp:noobjects,nomethods,nomonitors rabbit.proxy.Proxy
   This will start jmp with a minimum of tracing. This is very useful if you have an initialization that takes much time. Once your application has reached a suitable state for profiling you enable the profiling you need.

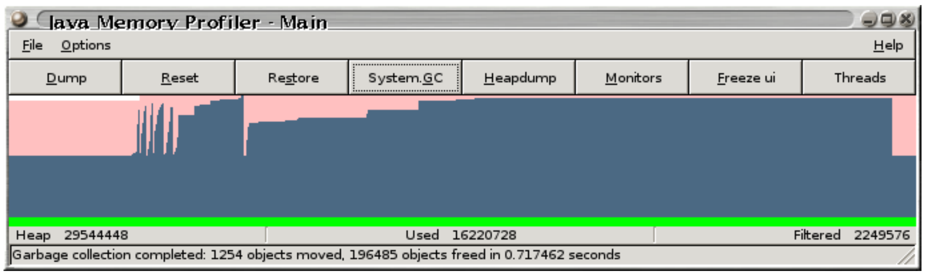
2. java -Xrunjmp:noobjects,nomonitors,filter=rabbit rabbit.proxy.Proxy
   This will start jmp with only method tracing on. An initial filter for classes in rabbit and sub packages is also set.

[4]Well nobody reads the documentation so why should you?

3. java -Xrunjmp:nomethods,nomonitors,filter=rabbit rabbit.proxy.Proxy
   This will start jmp with only object tracing on. Filter as above.

4. java -Xrunjmp:nomonitors,filter=rabbit rabbit.proxy.Proxy
   This will start jmp with object and method tracing. Filter as above.

5 JMP Main window



Heap 29544448 Used 16220728 Filtered 2249576

Garbage collection completed: 1254 objects moved, 196485 objects freed in 0.717462 seconds

The main window for jmp has a few buttons, a menu bar with some entries, a heap graph and a status bar. The buttons give easy access to the standard operations used when profiling. The heap graph gives a good overview of how much memory the jvm is currently using both in graphics and in text form.

5.0.1 Buttonbar

Dump this button creates a file with the current status of the jvm. The file contains all threads with complete stack traces[5], all classes with an instance count that is not zero and also all methods that have a method time that is not zero. The file created will only hold information that passes the current filter for methods and classes.

Reset this button temporarily sets the instance count to zero for all classes and also set all method times to zero. This can be very useful if you want to profile a specific operation after running the profiler for some time. No data is thrown away by pressing reset, all data is saved and can be brought back by the restore button. After reset has been pressed all objects in the string list and object list from "show alloc'ed" will be objects created after the last reset only.

Restore this button restores the values from all previous resets. That is it is **not** possible to store several levels of information, only one level is stored even though the reset button may give the impression of several levels. After restore has been pressed all objects in the string list and object list from "show alloc'ed" will contain all objects in the jvm.

System.GC force the jvm to do a full garbage collect. According to the jvmpi specification this actually cause the garbage collector to run (if you have read the java API you know that the System.GC call is only a hint to the jvm).

Heapdump ask the jvm for a heap dump, when jmp parses the heap it will build up the instance owner information. The heapdump will also be saved in a text file that can be analyzed later on. [6]

Monitors brings up information about current monitors, this may be used to detect deadlocks. See section 9 for more information about monitors.

[5]Only if method tracing is enabled

[6]Note: at least the SUN jvm will allocate a big block of data for the heap dump. This means that memory usage will increase until the next GC occurs.

This button will ask the jvm for a monitor dump so you do not need any profiling enabled for this button to work.

Freeze ui tells jmp to stop updating its user interface. Profiling will still continue, but no updates will be shown. Pressing this button once more will resume the ui updates. Updating the jmp user interface takes time so this button will make profiling run faster. This button can also be used if you want to see the results of an operation for a longer time, but still want the profiler and program to continue running in the background.

Threads brings up the thread window. This window shows all threads in the jvm and it is possible to inspect the stack for each thread. See section 8 for more information.

5.0.2 Menubar

File

Dump performs the same function as the dump-button.

Reset counters performs the same function as the reset-button.

Restore counters performs the same function as the restore-button.

System GC performs the same function as the System.GC-button.

Heapdump performs the same function as the heapdump-button.

Monitors performs the same function as the Monitors-button. See section 9 for more information.

Freezed ui performs the same function as the freeze ui-button.

Threads performs the same function as the Threads-button. See section 8 for more information.

Options

Filter brings up a dialog for setting the current filter. See section 5.3 for more information.

Events brings up a dialog for selecting which profiling events jmp will enable.

Help

About brings up a dialog showing the authors of jmp.

5.1 Heap graph

The heap graph shows memory usage over time. The graph has three colors, normally pink, blue and green. Green is used to show the currently filtered set, blue shows total memory usage[7] and pink shows how much memory is allocated for the java heap.

[7]Relative to last reset.

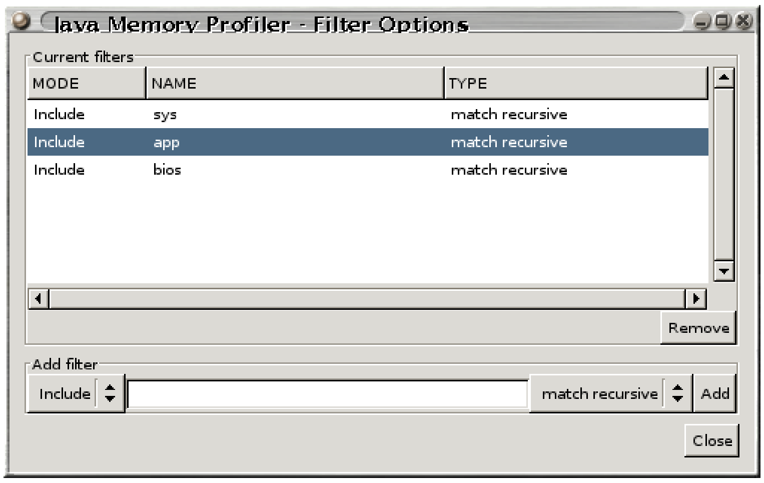
Under the heap graph there are three information boxes that gives exact sizes for the heap size in bytes, the number of used bytes and the number of bytes used by classes in the currently filtered set.

The graph is rescaled as needed and by watching it it should be quite easy to see if you have a memory leak (that is if the graph keeps increasing). You can also see if you allocate a lot of temporary objects by checking if your heap graph shows a saw tooth characteristics. Heavy allocation of temporary objects meand that you have a lot of garbage collection going on. Heavy garbage collection is generally not good for performance.

5.2 Status bar

The status bar will show some information about what happens inside the jvm. A typical message that will be displayed in the status bar from time to time is the garbage collection summary. Other information found here is the name of the files that are created when jmp writes a data dump file or performs a heap dump.

5.3 Filter



MODE	NAME	TYPE
Include	sys	match recursive
Include	app	match recursive
Include	bios	match recursive

Normally you will filter the data shown to only include the classes you are interested in. Showing all the classes loaded will be slower and show much noice. The filter dialog is available in the options menu. It is possible to set up several filters, each filter can include or exclude the matched classes. In jmp there are 4 type of filters:

Match class this mode jmp will only show instances and methods from the specified class.

Match package this mode will make jmp show all classes and methods from one specific package.

Match recursive this mode will make jmp show all classes and methods from one specific package and all of the sub packages of it. For example a recursive filter of “java.awt” will also match “java.awt.event.FocusEvent”.

Match all in this mode no filtering is done, all classes loaded are shown.

Filters are evaluated in reverse order order (last filter is evaluated first). Normal usage is to have one or two include recursive filters for the package(s) you want to see data about. It is however possible to add one filter that matches all and then add an exclude <somepackage> to show data about every class except thoose in somepackage.

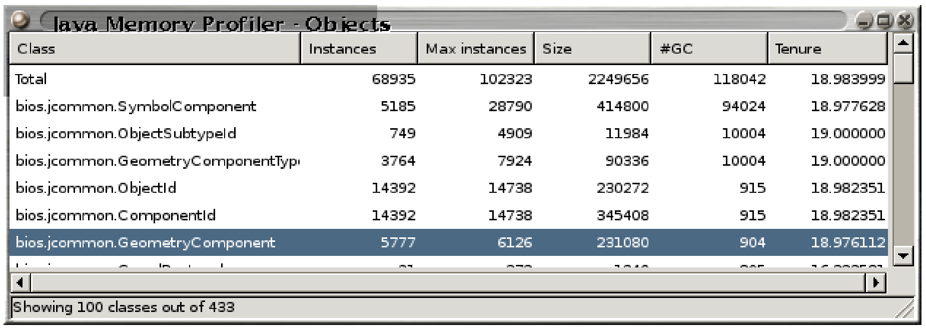
The filter menues found in some popup menues will remove all filters and install one new filter, the type of the installed filter will be either of match class, match recursive or match all.

5.4 Profiling events

The event dialog makes it possible to enable and dissable profiling events during runtime. Running with full profiling takes much time, so only enable what you need.

- Enable object tracing if you want to find memory leaks. This will give you a correct instance count and togheter with a heap dump you will know which objects that hold the references to the objects that should have been garbage collected.
- Enable method tracing if you want to find performance hot spots. This will show you how much time each method takes and how many times it has been called.
- Enable monitor tracing if you want to find out about deadlocks and contention.

6 Profiling objects



Java Memory Profiler - Objects

Class	Instances	Max instances	Size	#GC	Tenure
Total	68935	102323	2249656	118042	18.983999
bios.jcommon.SymbolComponent	5185	28790	414800	94024	18.977628
bios.jcommon.ObjectSubtypeId	749	4909	11984	10004	19.000000
bios.jcommon.GeometryComponentTyp	3764	7924	90336	10004	19.000000
bios.jcommon.ObjectId	14392	14738	230272	915	18.982351
bios.jcommon.ComponentId	14392	14738	345408	915	18.982351
bios.jcommon.GeometryComponent	5777	6126	231080	904	18.976112

Showing 100 classes out of 433

6.1 Class window

This window holds a table that shows quick information about the classes in the jvm. Normally you have it filtered so the view will not show full information, only what you want to see. By clicking on a column header will make jmp sort the classes according to that column. Only classes that have an instance count that is not zero is shown, a negative instance count is possible after the reset button has been pressed and garbage collection has taken place. There is one special row in this table, it has the "class name" **Total** and that row is a summary of the current view. Total is not real class, it is only shown that way in this view. The meaning of Total is to give a quick overview of how much memory the currently selected set uses.

Class is the full name of the class, including the package the class belongs to.

Instances is the current number of currently allocated instances of this class. Note that jmp does not check if the instances are reachable(live) or not. That is all instances that have not yet been garbage collected are also counted.

Max instances is the maximum number of simultianous instances of this class. Note that jmp does not check if the instances are reachable(live) or not. That is all instances that have not yet been garbage collected are also counted.

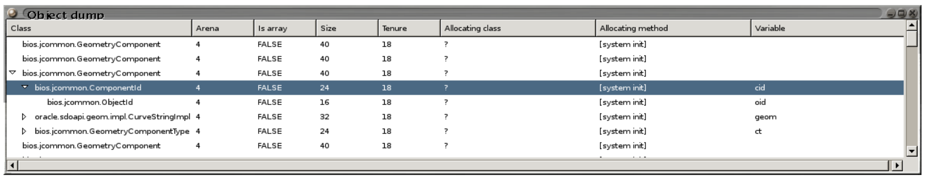
Size is the current size, in bytes, of all instances of this class. Note that jmp does not check if the instances are reachable(live) or not. That is all instances that have not yet been garbage collected are also counted.

#GC is the number of instances of this class that have been garbage collected. If you create a lot of short lived instances this column will show a high count. Big numbers in this column are usually worth an investigation.

Tenure is the average number of garbage collections that objects of this class has survived.

By right clicking on a row you will get a popup menu with some additional options that may be useful for finding memory leaks or understanding the memory usage.

6.2 Show alloc'ed instances



Class	Arena	Is array	Size	Tenure	Allocating class	Allocating method	Variable
bios.jcommon.GeometryComponent	4	FALSE	40	18	?	[system init]	
bios.jcommon.GeometryComponent	4	FALSE	40	18	?	[system init]	
▽ bios.jcommon.GeometryComponent	4	FALSE	40	18	?	[system init]	
▼ bios.jcommon.ComponentId	4	FALSE	24	18	?	[system init]	cid
bios.jcommon.ObjectId	4	FALSE	16	18	?	[system init]	oid
▷ oracle.sdoapi.geom.impl.CurveStringImpl	4	FALSE	32	18	?	[system init]	geom
▷ bios.jcommon.GeometryComponentType	4	FALSE	24	18	?	[system init]	ct
bios.jcommon.GeometryComponent	4	FALSE	40	18	?	[system init]	

This operation brings up a window with all instances of this class that have not yet been garbage collected. In this window you can see where each instance was allocated (if known, that is if method tracing and object tracing was enabled when the object was created).

By selecting an object and right-clicking you will get a popup menu with two options, the first option, "inspect instance", find all non null references in the object and expand the instance tree with them. Inspection of an instance is done at the request time so when the window shows up for the first time it will have a simple list in it, only by manually selecting the "inspect instance" will you get a tree. For the second option "owned object statistics" see below, section 6.2.1.

Only objects that are created after the last reset operation was performed are shown. Perform a restore if you want to see all objects in the jvm.

Class the name of the class. When this dialog is opened from the class list all instances should be of the same type, but if the dialog is opened from the method window (see section 7.2) or some instances are inspected then the instances may be of different classes. When you have expanded some nodes the class column will hold many different types. q

Arena which memory arena the object currently lives in. Depending on garbage collector for your jvm the arenas may be "short lived objects" and "long lived objects". How many different arenas that are used by the jvm can vary much depending on which garbage collector that has been selected[8].

Is array if the object is an array or not, TRUE means array, FALSE means normal instance.

Size the size of the instance or array. Instances of the same class should normally have the same size, but if they are arrays the size will vary. For objects of different classes the size column will be different.

Tenure is the number of garbage collections that this instance has survived.

Allocating class the class that created the instance. This information is only available if both method profiling and object profiling was enabled at the time the instance was allocated. If allocating class is unknown a "?" will be shown instead.

Allocating method the name of the method that created the instance. This information is only available if both method profiling and object profiling was enabled at the time the instance was allocated. If allocatinng method is unknown "[system init]" will be shown instead.

[8]Due to limitations in SUN's jvm all the arena column rarely holds any information.

Variable is the name of the variable (or index for arrays) holding this instance. This column is only given for objects that have been found by inspecting a node higher up in the tree.

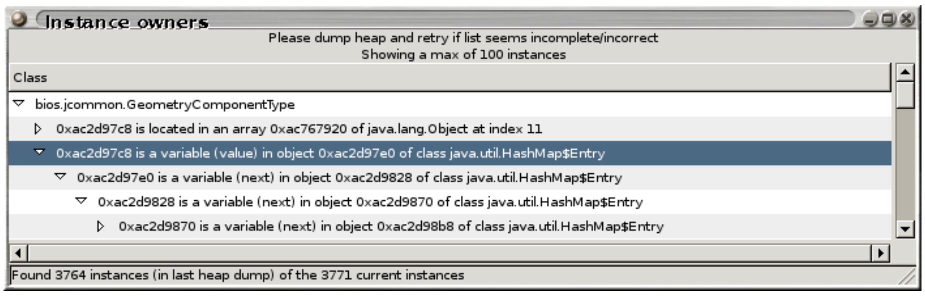
6.2.1 Owned object statistics

By selecting this option jmp will find all objects below a given object and show how much data the selected object keep alive. The window show the number of instances of each class found and the size of all instances. Circular references and/or multiple references are handled nicely.

This is a quick way to find out how much memory a single object keeps alive. Note however that other instances may also keep the same objects alive, so do not believe that you will free all of the data shown if you can remove the selected instance.

This function inspects the objects upon request so no heap dump information is needed, however asking the jvm for the information may take some time if you select the object in the top of a big data structure.

6.3 Show object owners



```
Please dump heap and retry if list seems incomplete/incorrect
Showing a max of 100 instances

bios.jcommon.GeometryComponentType
  0xac2d97c8 is located in an array 0xac767920 of java.lang.Object at index 11
  0xac2d97c8 is a variable (value) in object 0xac2d97e0 of class java.util.HashMap$Entry
    0xac2d97e0 is a variable (next) in object 0xac2d9828 of class java.util.HashMap$Entry
      0xac2d9828 is a variable (next) in object 0xac2d9870 of class java.util.HashMap$Entry
        0xac2d9870 is a variable (next) in object 0xac2d98b8 of class java.util.HashMap$Entry

Found 3764 instances (in last heap dump) of the 3771 current instances
```

During heap dumps jmp stores the owner for each object. This information is available in the popup menu reached by right-clicking on a class. If no heap has been dumped the first time this information is requested, jmp will automatically request on heap dump. Since a heap dump takes time this may cause a delay before this window is shown for the first time.

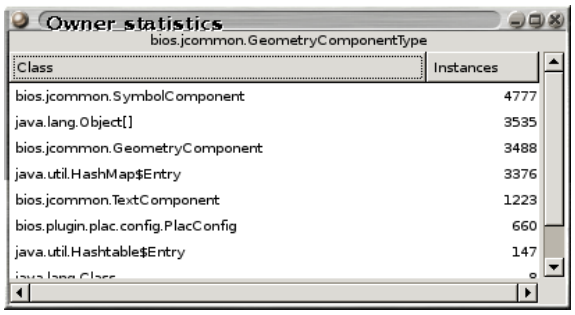
In the window a tree will show who the owner(s) of each instance are. Following the tree will show you why your objects **won't** be garbage collected.

The information is only gathered during heap dumps so if a lot of garbage collection has happend after the last heap dump the information may not be complete. For best result run the garbage collector, get a heap dump and then directly show instance owners. The status bar may give indications if you need to perform a new heap dump.

Currently jmp will only show owners up to 7 level higher than the given object. In the future it may be possible to manually expand nodes above this. Expansion will also stop when jmp finds a static reference. When jmp encounters a circular list, it will not go back to an already visited object.

Right clicking on a line will give an option to show the other instances owned by the given object. This means that once you have located the object you think holds to many references it is easy to check if that really is true. The dialog is that will open is the same type of dialog as the one found in section 6.2

6.4 Show object owners statistics


Owner statistics

bios.jcommon.GeometryComponentType

Class	Instances
bios.jcommon.SymbolComponent	4777
java.lang.Object[]	3535
bios.jcommon.GeometryComponent	3488
java.util.HashMap$Entry	3376
bios.jcommon.TextComponent	1223
bios.plugin.plac.config.PlacConfig	660
java.util.Hashtable$Entry	147

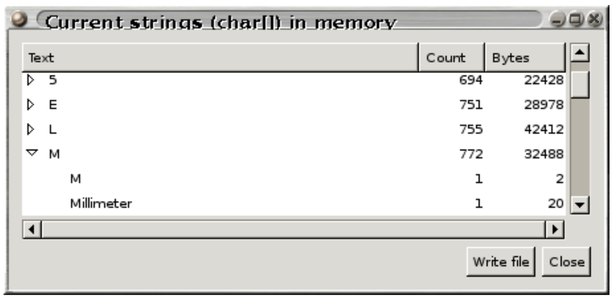
This is a quick way to find out a summary of which objects it is that keeps the references to a given class. . As with the object owners, this option requires a recent heap dump to show accurate information and jmp will automatically request a heap dump if no such information has been gathered.

The statistics is only gathered for direct references, no high order collection is done, use the show object owners dialog to find out such information.

6.5 Inspect class

This function brings up a window with some data about the class. The data displayed contains the super class, the implemented interfaces, the static variables and the instance variables in the class.

6.6 Show strings


Current strings (char[]) in memory

Text	Count	Bytes
▷ 5	694	22428
▷ E	751	28978
▷ L	755	42412
▽ M	772	32488
M	1	2
Millimeter	1	20

Since a very common cause of high memory usage is duplicate strings it is good to check if there are many duplicates and this method brings up a window where it is easy to find strings that occur more than once.

It is very easy to generate duplicate strings, reading data from files or a database will quite often do it. One way to reduce the memory usage is to make sure that you do not keep duplicate strings in memory. If you find that you have many duplicate strings then you may want to add a string cache[9] that given a string either finds it in the cache and returns the cached string or adds it to the cache for furhter usage.

The window contains a tree where each root node holds a string (to identify the starting character of the strings under it), a count of the number of strings

[9]String.intern() may be what you want, however if you want more control I suggest you use your own cache based on a WeakHashMap.

and the number of bytes used byt the strings under that node. The leafs of the tree are the individual strings.

JMP/0.50.1 has a bug that makes the count for all individual strings be 1, duplicated strings will appear as different lines in the tree.

Only strings created after the last reset will be shown. Call restore to show all strings in the jvm.

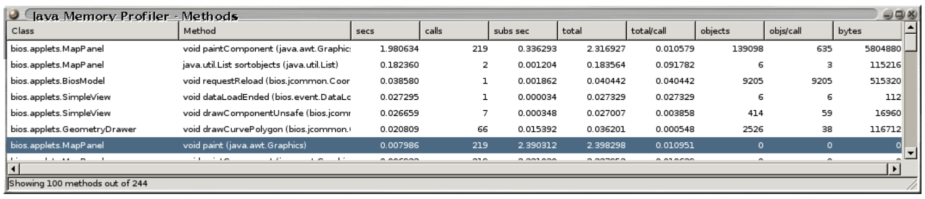
The window has a button to write a file with all the char[] for later inspection. The file will be written in the UTF-8 encoding to handle all java strings.

There is also a button to close the window.

6.7 Filter

The filter menu entries is a fast and simple way of setting a recursive filter for all of the (super) packages or the selected class. There is also an option "no filter" that will remove the current filter. If you need more options for filtering you can open the filter dialog from the main window (found in the menus as options->filter), see section 5.3 for more information about what types of filters jmp can have.

7 Profiling methods



Java Memory Profiler - Methods

Class	Method	secs	calls	subs sec	total	total/call	objects	objs/call	bytes
bios.applets.MapPanel	void paintComponent (java.awt.Graphic:	1.980634	219	0.336293	2.316927	0.010579	139098	635	5804880
bios.applets.MapPanel	java.util.List sortobjects (java.util.List)	0.182360	2	0.001204	0.183564	0.091782	6	3	115216
bios.applets.BiosModel	void requestReload (bios.jcommon.Coor	0.038580	1	0.001862	0.040442	0.040442	9205	9205	515320
bios.applets.SimpleView	void dataLoadEnded (bios.event.DataLc	0.027295	1	0.000034	0.027329	0.027329	6	6	112
bios.applets.SimpleView	void drawComponentUnsafe (bios.jcomr	0.026659	7	0.000348	0.027007	0.003858	414	59	16960
bios.applets.GeometryDrawer	void drawCurvePolygon (bios.jcommon.(	0.020809	66	0.015392	0.036201	0.000548	2526	38	116712
bios.applets.MapPanel	void paint (java.awt.Graphics)	0.007986	219	2.390312	2.398298	0.010951	0	0	0

Showing 100 methods out of 244

7.1 Method window

In the methods window you will find information about all the methods that have been called[10] when method profiling has been enabled (and some columns also need object profiling to show interesting values).

Class is the class of the method.

Method is the signature of the method. JMP tries to convert the JNI signature into something human readable.

Secs is the time spent in this method alone. Note: only time when method profiling has been enabled is counted.

Calls is the number of times this method has been called. Note: only time when method profiling has been enabled is counted.

Subs sec is the time spent in methods called from this method has taken. If your method is recursive it may show a big time here. Note: only time when method profiling has been enabled is counted.

Total is the sum of the columns secs and subs sec. Note: only time when method profiling has been enabled is counted.

Total/call is the average total time spent per call to the given method

Objects is the number of objects this method has allocated. Note this column only shows interesting values if both method and object profiling has been enabled.

Objs/call is the average number of objects allocated in each invocation of the given method. Note this column only shows interesting values if both method and object profiling has been enabled.

bytes is the number of bytes this method has allocated. Note this column only shows interesting values if both method and object profiling has been enabled.

The standard sort order of this table is based on the column secs, but by clicking on any table header you will sort on that column.

Some people think that the default sort order should be total, but that would mean that the main-method was on top always and we already know that the program takes time, if we did not we would not profile it.

[10]This information is accurate, but see section 12 on why it sometimes will show odd results.

Sorting on either class or method will sort on first class and then on method.

By right clicking on a row you will get a popup with some additional operations.

7.2 Show alloc'ed instances

This is the same window as can be reached from the class window described in section 6.2. The objects shown will all have the allocating class and method set to the selected row in the method window. Objects that have been garbage collected since they were allocated will not show up in the list. This means that you can use this option to find out if your method allocated short lived or long lived objects. Make sure that your long lived objects are the ones you intended.

7.3 Show called methods (down)

This will bring up a call graph as a tree. Currently only the class and method names will be shown. The tree will start in the selected method and show methods that were called from it.

Class the name of the class that the called method is in.

Method the signature of the method that were called.

It is quite common to find a few methods that are not expected here, mostly constructors of different kinds (the methods that have a signature of "void <init> (...)") and class initialization (methods that have a signature of "void <clinit> ()").

7.4 Show callee methods (up)

This will bring up a call graph as a tree. Currently only the class and method names will be shown. The tree will start in the selected method and show the methods that called it.

Class the name of the class that the called method is in.

Method the signature of the method that were called.

7.5 Show callee methods

This will bring up a call graph as a tree, similar to the "show called methods", The difference is that this show the callee, that is the methods that called the selected method.

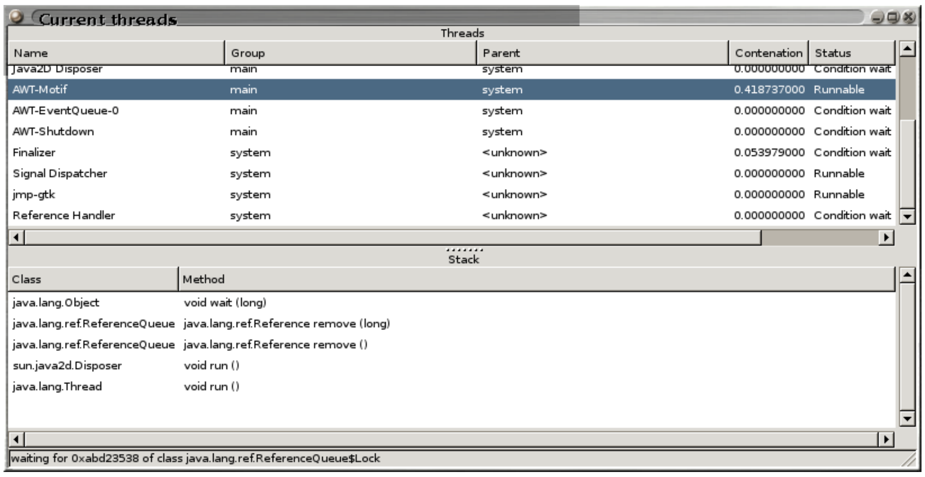
7.6 Method info

This window will show some extra information about each method like source file, JNI signature and line numbers for the start and end of the method.

7.7 Filter

This filter works exactly as the filter on objects, see section 5.3 and 6.7 for more information.

8 Current thread status


Current threads

Threads

Name	Group	Parent	Contenation	Status
Java2D Disposer	main	system	0.000000000	Condition wait
AWT-Motif	main	system	0.418737000	Runnable
AWT-EventQueue-0	main	system	0.000000000	Condition wait
AWT-Shutdown	main	system	0.000000000	Condition wait
Finalizer	system	<unknown>	0.053979000	Condition wait
Signal Dispatcher	system	<unknown>	0.000000000	Runnable
jmp-gtk	system	<unknown>	0.000000000	Runnable
Reference Handler	system	<unknown>	0.000000000	Condition wait

Stack

Class	Method
java.lang.Object	void wait (long)
java.lang.ref.ReferenceQueue	java.lang.ref.Reference remove (long)
java.lang.ref.ReferenceQueue	java.lang.ref.Reference remove ()
sun.java2d.Disposer	void run ()
java.lang.Thread	void run ()

waiting for 0xabd23538 of class java.lang.ref.ReferenceQueue$Lock

8.1 Thread window

The thread window is a two paned window that show the current threads and a stack trace for a specific thread and moment. When jmp is running there will be a thread named “jmp-gtk”, this is the thread that jmp uses to update its user interface.

8.1.1 Current threads

At the top of the window is a list of current threads. By clicking on a column header you can select the sort order for this table.

Name is the name of each thread.

Group is the name of the thread group. If the group is not known the group name will be shown as <unknown>

Parent is the name of the thread group that this thread was created in. Some threads are started without a parent thread. Thoose threads will have the parent <unknown>

Contention is the total time this thread has spent waiting for a monitor. Time is only counted when monitor profiling is enabled. If a thread has high contention it is probably a good idea to investigate why and try deserialize the critical paths.

Status shows what the thread is doing. Status can be one of:

- **Condition wait** means that the thread is waiting, normally this includes sleep (), read () and other blocking operations.
- **Monitor wait** means that the thread is waiting for a monitor. Heavy contention is not good for performance.
- **Runnable** means that the thread is either on the CPU performing work or waiting for its turn on the CPU.

When a thread has been suspended or interrrupted it will also have flags to indicate the status:

(S) means that the thread has been suspended.

(I) means that the thread has been interruped.

Time is the total time the thread has been running. This column will only show sane values when jmp is run with the threadtime argument, which means that it currently only works under linux.

8.1.2 Inspecting one threads stack

When you click on a thread its current stack will be shown in the bottom part of the thread window. Note that this will only work if you have method tracing enabled. Some threads, like the jmp-gtk and the Signal Dispatcher thread will never show information since they are only executing native code. The method at the top of the table is the method that the thread is currently executing.

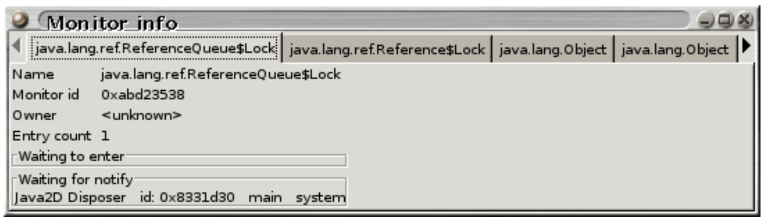
Class is the name of the class that holds the method.

Method is the human readable method name.

8.1.3 Statusbar

The statusbar will show some information about the currently selected thread. If monitor profiling is enabled and the selected thread is waiting on an object the id and class of that object will be shown.

9 Monitor window


Monitor info
java.lang.ref.ReferenceQueue$Lock
java.lang.ref.Reference$Lock
java.lang.Object
java.lang.Object

```
Name          java.lang.ref.ReferenceQueue$Lock
Monitor id    0xabd23538
Owner         <unknown>
Entry count   1
Waiting to enter
Waiting for notify
Java2D Disposer   id: 0x8331d30   main   system
```

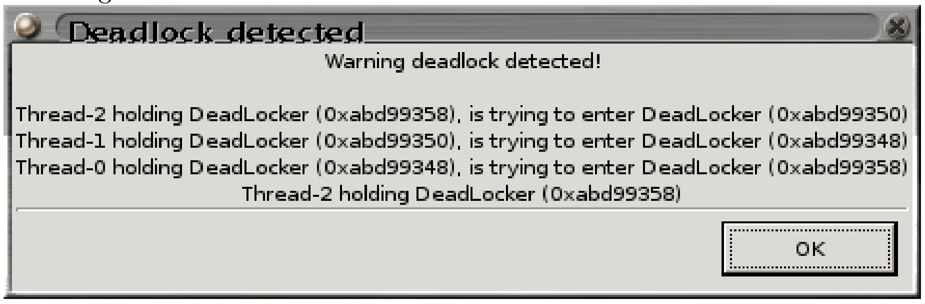
The monitor window lets you see which monitors that are currently active. At the top of this tabbed pane you select the monitor you want to inspect. It is always possible to show the monitors, when this dialog is opened a monitor dump is performed so no profiling needs to be enabled if all you want is to check the monitors in a deadlock situation.

The information for each monitor shows the name[11] and id of the monitor. You also see which thread that currently holds this monitor. If you need more information about the thread use the thread window.

There are two lists of threads, the first list shows which threads that are waiting to enter a block synchronized on the monitor and the second list shows which threads that are waiting for a notify on the given monitor. Both lists show the thread name, the thread id, the thread group and the thread parent. Again if you want more information about the threads look in the threads window.

9.1 Deadlock detection

JMP has a semi-automatic deadlock detector, now what does that mean? It means that when you open the monitor dialog jmp will search for a deadlock situation. If a deadlock is detected a warning dialog is shown. It will look like this image.



```
Warning deadlock detected!

Thread-2 holding DeadLocker (0xabd99358), is trying to enter DeadLocker (0xabd99350)
Thread-1 holding DeadLocker (0xabd99350), is trying to enter DeadLocker (0xabd99348)
Thread-0 holding DeadLocker (0xabd99348), is trying to enter DeadLocker (0xabd99358)
Thread-2 holding DeadLocker (0xabd99358)
```

In the dialog we can see the circular dependancies of the three threads that are in a deadlock situation.

[11]The name is either the class name for a java monitor or the string name given native monitor.

10 Java api

JMP has a java API that can be used to control jmp from inside java. You will still need to start your java program with -Xrunjmp to use jmp though. This API is quite new and it only has limited functionality, but the idea is that it should be possible to control profiling options from normal java code for example from inside unit tests.

Controlling jmp from inside java code can be as simple as (this example uses features first found in jmp/0.29)

```
JMPController.runGC ();               // run deterministic GC
JMPController.runDataDump ();         // get a data dump
JMPController.enableObjectEvents (); // start profiling
runTestOfCode ();                     // run actual test
JMPController.disableObjectEvents ();// end profiling
JMPController.runGC ();               // clean up
JMPController.runDataDump ();         // get a data dump
```

To find more information about the java api consult the javadoc.

To use the java api you will need the jmp.JMPController.class in your classpath. This class is not found in all the binaries, so you may have to download it from the JMPs web site.

11 File formats

11.1 Dump file format

JMP normally writes data dumps to files named *jmp_dump-<number>.txt* where number always starts at 0 and is incremented for each dump. The name may be set with the startup option dumpfile. Files are written in the current directory and successive runs overwrite previous files without asking[12].

The dump files that jmp produces contain much of the information found in the user interface. Basically the file is tab separated, making it easy to import the data into your favorite spreadsheat program. Dump files have three sections and depending on what profiling options are turned on when the file is generated the sections may be empty. The first section contains a list of all the threads in the system and the stack for each of the thread. The second section is a list of all the (filtered) classes with name, instance count and the other information found in the class window. The third section contains (filtered) method information, much like the method window.

11.2 Heap dump file format

JMP writes the heap dump data to files named *jmp_heap_dump_<number>* where number always starts at 0 and is incremented for each dump. In the future it should be possible to set the name of the files with a startup option. Files are written in the current directory and successive runs overwrite previous files without asking[13].

[12]This is a security problem, but you do not run the profiler as root do you?
[13]This is a security problem, but you do not run the profiler as root do you?

Each time that jmp parses a heap dump it records the data into a file that can be analyzed later on, but be warned, it is much easier to use the GUI to inspect the heap after a heap dump. You will need to know a bit of JNI and understand what the types have for letters (I for int, J long...).

11.3 String dump file format

JMP writes the string dump data to files named *jmp_string_dump-<number>.txt* where number starts with 0 and is incremented for each dump file written. In the future it should be possible to set the name of the files with a startup option. Files are written in the current directory and successive runs overwrite previous files without asking.

The string dump files contain data on the format "'<string>'\t'<count>"'. Remember that the strings may contain newlines. The file is UTF-8 encoded.

12 Inlining and other features

Hotspot and probably other jvms inline methods to achieve faster code. This feature may be problematic when trying to profile a program. Sometimes your methods won't seem to be called at all (they have been inlined into someplace else) and sometimes your methods will seem to allocate much more objects than you can see (inlined methods allocations are added to the outer method).

12.1 Bugs

JMP has bugs that may cause it to crash, normally taking the whole jvm with it. When you encounter a crash please try to build a debug-enabled version and report to the jmp-devel mailing list, see the **README** supplied with jmp for how to enable debugging. Running with less profiling events enabled can also make jmp more stable.

There are also bugs in at least SUN's jvms (at least up to 1.4.0_02) that may cause it to crash at random if you have method tracing enabled.

13 JVM simulator

In the source code for jmp you will find a small simulator that can be used for testing jmp and other jvmpi based profilers. To build and run a test of jmp go into the jvmsimulator directory and do:

```
make jvmsimulator
jvmsimulator libjmp.so additional_arguments
```

The additional arguments are the same as the one from argument found in "java -Xrunjmp:additional_arguments". If you happen to develop another profiler you can test it by changing libjmp.so to the name of that profiler. To test jmp you **must** give at least the argument *simulator* to tell jmp to not evaluate the jvm version (that requires java method calls and that can not yet be done in the simulator).