BEA JRockit Mission Control™®

BEA JRockit Runtime Analyzer

Mission Control version 2.1®

Document Revised: April, 2007

Contents

Methods and Call Trace Information

Garbage Collection Events Information

General Garbage Collector Information

Java Heap Content Information

Getting Familiar with the Events Table Tab

Getting Familiar with the Stack Trace Tree Overview Tab

Adding Comments to a Recording

CHAPTER 1

Welcome to the BEA JRockit Runtime Analyzer (JRA)

The JRA is a JVM profiler and a Java application profiler that is especially designed for BEA JRockit. It has been around for quite some time within the JRockit development team, and was originally created to let the JRockit developers find good ways to optimize the JVM based on real-world applications, but it has proven very useful to developers outside of BEA for solving problems in both production and development.

This section is divided into the following topics:

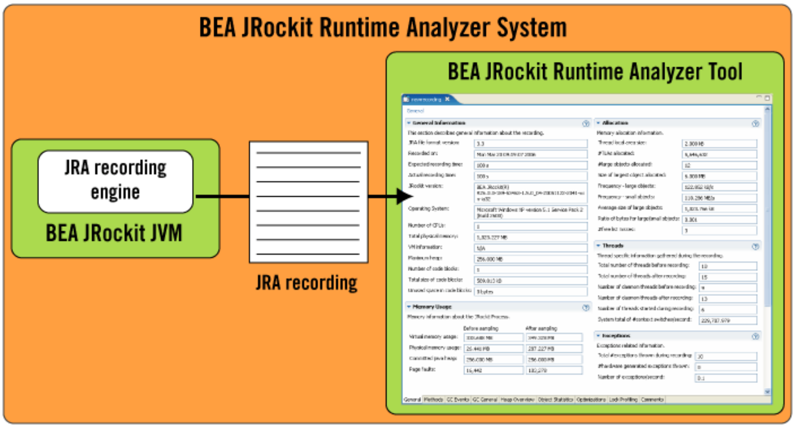
- How Does the JRA System Work?
- What is a JRA recording?
- What is the JRA tool?

How Does the JRA System Work?

The JRA system consists of two parts (see Figure 1-1): one part inside the JRockit JVM that collects data and saves it, and an analysis tool that visualizes the information to make it useful to the end-users. The JRockit-internal part produces a recording of the system's runtime behavior during a user specified period of time, typically a few minutes. The recording results in an XML file that JRockit writes to disk and automatically launches in the JRA tool once the recording is complete (this behavior is valid for JRockit 5.0; for JRockit 1.4, the file is saved to disk and you need to locate it first to open it).

The recording is a great way to share how JRockit has worked with your application. You can also use several recordings to compare and contrast how different command line options change the behavior of your application, for example, by creating before-and-after recordings. When sending trouble reports to the BEA JRockit support department, you are required to attach a JRA recording to your trouble report. The recording is analyzed “offline” by the Runtime Analyzer tool.

Figure 1-1 The BEA JRockit Runtime Analyzer System



The recording engine uses several sources of information including the JRockit Hot Spot Detector (also used by the optimization engine to decide what methods to optimize), the operating system, the JRockit Memory System (most notably the garbage collector), and the JRockit lock profiler, if enabled.

What is a JRA recording?

It is a collection of data about the JVM that can be used to analyze the doings and happenings of JRockit. It is also a “flight recording” of what has happened in the JVM when running the JRA.

What is the JRA tool?

The JRockit Runtime Analyzer tool is a Java application that parses a previously produced JRA recording and visualizes the data. This is a convenient way to analyze the data offline. The size of the compressed recording is on the order of a few hundred kilobytes, so a system administrator can easily make a recording of a deployed system and send it to the JVM or application developer who probably is in a better position to analyze it.

The JRA tool shows a top list of the hottest methods where you can select a method and see its call tree, i.e. its predecessors (what other methods have called this method) and successors (what methods the selected method will call). A percentage for each branch indicates how common a given path is.

As for memory management, there is a graph of the varying heap usage and pause times for the garbage collections. Detailed information about each GC shows exactly how much memory was released in a collection. There are also pie charts showing the distributions in size of free memory blocks and the distribution of occupied memory in small and large object chunks.

CHAPTER 2

Using the JRA Tool

How to use the JRA Tool is divided into the following topics:

CHAPTER 3

Getting Started with the BEA JRockit Runtime Analyzer Tool

Before you can view how your application is behaves, you need to create a JRA recording, i.e. collect data from your application. The JRockit recording engine produces a recording of the system's runtime behavior during a specified period of time, typically a few minutes. The recording results in an XML file that opens automatically in the JRA tool upon completion (for JRockit 1.4, the XML file is saved to the disk where JRockit is running). The file can be analyzed "offline" by the Runtime Analyzer tool.

This section describes how you start a recording and how you setup the JRA tool to suit your needs.

The following topics will be covered:

- Creating a Recording
- About JRA Overhead when Recording
- Opening a JRA File
- Comparing and Contrasting JRA Recordings
- Customizing Your JRA Tool

Creating a Recording

There are several ways to start a JRA recording:

- To start a recording within Mission Control

- To start a recording with jrcmd
- Starting a Recording from the JRockit Command Line

Note: If you are running Mission Control on a Windows system, you need to be a member of the **Administrators** or the **Performance Logs** user groups to be able to create a JRA recording. The typical error message, for not being part of either of these groups, can look like this:

```
[perf ] Failed to init virtual size counter:
```

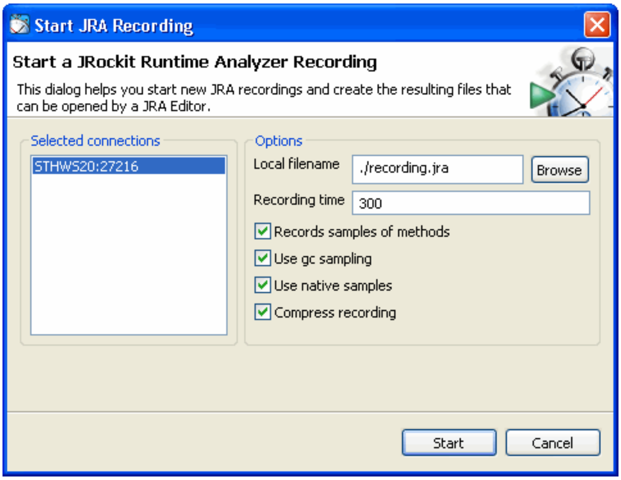
To start a recording within Mission Control

1. Start your Java application with JRockit and add the `-Xmanagement` option to the command line.
2. Start the JRockit Browser and connect to the JRockit instance you just started.
3. Make sure that your application is running and is under load.

   If you run the application without load, the data captured from that application will not show where there is room for improvements.
4. Click the **Start JRA recording** button.

   The JRA Recording dialog box appears (Figure 3-1).

Figure 3-1 JRA Recording Dialog Box



5. Select the connection you want to record.

6. Type a descriptive name for the recording in the **Local filename** field.

   The file is created in the current directory of the BEA JRockit process, unless you specify a different path. If an old file already exists, it will be overwritten by the new recording.

7. Set a time for the length of the recording (in seconds) in the **Recording time** field.

   Note: If you set a time that is too short, e.g. shorter than 30 seconds, you will probably not get enough sample data for the recording to be meaningful.

8. Select none, one, or all of the following sampling options:

   – **Records samples of methods**—records samples of methods

   – **Use gc sampling**—records garbage collection events

   – **Use native sampling**—records samples of native code

   – **Compress recording**—compresses recording to a zip file

   The **Selected JRockits** field displays which JRockit you will create your recording from.

9. Click **Finish**.

   The JRA recording progress window appears. When the recording is finished, it loads in the JRA tool.

To start a recording with jrcmd

1. Make sure that your application is running and is under load.

   If you run the application without stress, the data captured from that application will not show where there is room for improvements.

2. Use one of the following commands to initiate a recording:

   Windows platforms:

```
bin\jrcmd.exe <pid> jrarecording time=<jrarecording time>
filename=<filename>
```

   Unix platforms:

```
bin/jrcmd <pid> jrarecording time=<jrarecording time> filename=<filename>
```

   Where the arguments are:

   – `jrarecording time`—the duration of the recording in seconds (a good length is 300 seconds, i.e., five minutes).

– `filename`—the name of the file you want to save the recording to (for example `jrarecording.xml.zip`). The file will be created in the current directory of the JRockit process. It will be overwritten if it already exists.

For example:
`bin\jrcmd.exe <pid> jrarecording time=300 filename=c:\temp\jra.xml.zip`
Starts a JRA recording of 300s and stores the result in the specified file.

After the recording is initiated, BEA JRockit prints a message indicating that the recording has started. When the recording is done, it will print another message; it is now safe to shut down your application.

Starting a Recording from the JRockit Command Line

Use the `-XXjra` command in combination with an option listed in Table 3-1, for example, `-XXjra:recordingtime` to specify the duration of the recording.

Table 3-1 Command Line Startup Options

Option	Description
`delay`	Amount of time, in seconds, to wait before recording starts.
`recordingtime`	Duration, in seconds, for the recording. This is an optional parameter. If you don't use it, the default is 60 seconds)
`filename`	The name of recording file. This is an optional parameter. If you don't use it, the default is `jrarecording.xml`.
`sampletime`	The time, in milliseconds, between samples. Do not use this parameter unless you are familiar with how it works. This is an optional parameter.
`nativesamples`	Displays method samples in native code; that is, you will see the names of functions written in C-code. This is an optional parameter.
`methodtraces`	You can set this to false to disable the stack trace collection that otherwise happens for each sample. The default value is true.
`tracedepth`	Sets the number of frames that will be captured when collecting stack traces. Possible value are 0 through 16. The default value is 16.

Note: Setting `methodtraces` to false can still result in some stack traces being captured. These stack traces are captured as part of JRockit's dynamic optimizations and will have a depth of 3. If optimizations are turned off (`-Xnoopt`) these traces will not be captured.

You can view the startup options that you have set in the JRA recording, see Viewing VM Arguments Information. Listing 3-1 shows an example of how you can setup a JRA recording.

Listing 3-1 An example of using the `-XXjra` startup command:

```
-XXjra:delay=10,recordingtime=100,filename=jrarecording2.xml
```

would result in a recording that:

- Commenced ten seconds after JRockit started (`delay=10`).
- Lasted 100 seconds (`recordingtime=100`).
- Was written to a file called `jrarecording2.xml` (`filename=jrarecording2.xml`).

About JRA Overhead when Recording

The overhead while recording is very low—typically less than two percent. However, since JRA is forcing a full garbage collection at the beginning and at the end of the recording to generate the heap histogram data, there may be a spike at the beginning and at the end of a recording.

Opening a JRA File

Once you have created a JRA file, you can open it in the JRA tool to view what has happened in your application and in JRockit.

Note: If you have previously viewed a JRA recording in the JRA tool, it will automatically load when you open JRockit Mission Control.

To open a JRA file by dragging and dropping

1. Locate the JRA recording on your system.
2. Drag and drop the file to JRockit Mission Control.

To open a JRA file from JRockit Mission Control

1. In JRockit Mission Control, click **File** > **Open file** > **Open JRA Recording**.
2. Locate and select the recorded file and click **Open**.

3. Click **OK**.

   The **JRA General** tab now opens and you can view the data in the recording (see Figure 4-1).

   Note: If you have opened a recording that has been recorded with an older version of the JRA, some fields may not have any relevant data, since that data was impossible to obtain. That data will appear as "N/A".

Comparing and Contrasting JRA Recordings

The new JRA is excellent to use for comparing and contrasting recording. For example, you want to try different startup options in JRockit to see how they affect the running of your application.

To compare and contrast JRA recordings

1. Create two recordings, one for each setting you wish to try.
2. Open both recordings and lay them out in the JRA tool next to each other, by simply dragging the name tab to the toolbar in the JRA tool window.

Customizing Your JRA Tool

The following can be set to change the way you view a recording:

- Changing Table Settings
- Filtering Information
- Collapsing and Expanding a View
- Changing View of a Tab

Changing Table Settings

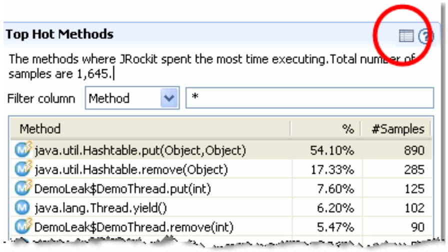
The JRA tool lists a lot of information in different tables. These tables can be customized to display information of your choice. You can also preset the width of the columns in the tables.

Note: You need to change the settings per table, i.e. there is no global change to all tables since they contain different types of information depending on the tab you are looking at.

To change the settings of the table

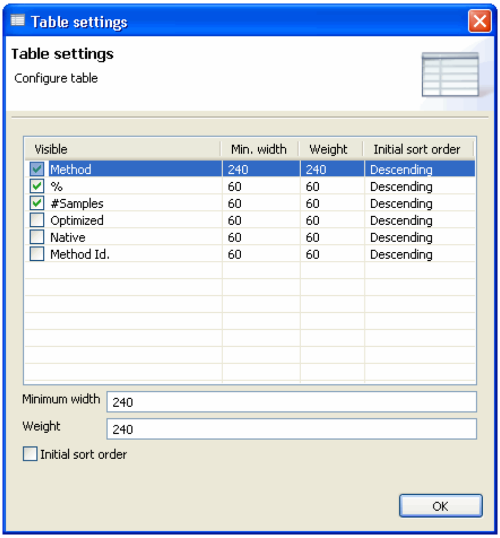
1. Click the **Table settings** button (see Figure 3-2).

Figure 3-2 Table settings button



A **Table settings** window appears (see Figure 3-3).

Figure 3-3 Table settings Window



2. Select what you want displayed in the table.
3. Set the **Min. width** and **Weight** of the column (optional) to a pixel value of your choice.
4. Select **Initial sort order** for a table item that you want the table to be sorted by.
5. Click **OK**.

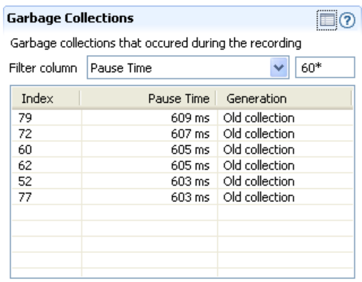
Filtering Information

Some of the information tables can contain lengths of data that can be hard to scroll through. Instead of scrolling through the long tables, you can filter for the information that you are interested in viewing.

To filter information

1. Select a table column name for which you want to filter the information. In this example, Figure 3-4, **Pause Time** was selected.
2. Enter a number or text for the information you want to see. In this example, Figure 3-4, ***60**** was used to see all Pause Times that contains a value starting with 6 and 0 (zero).

Figure 3-4 Filtering information



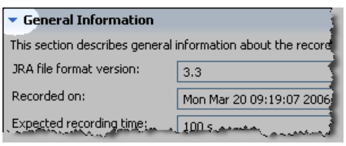
Collapsing and Expanding a View

Sometimes the information on a tab can be cumbersome to work with, then it is good to collapse the view of some fields.

To collapse/expand a view

- Click on the small arrow next to a description field (see highlight in Figure 3-5) to collapse the view of the General Information field.

Figure 3-5 Collapsing a view



Changing to view less values by right clicking a field. The next time you start the JRA tool, you will not see the specific field.

Changing View of a Tab

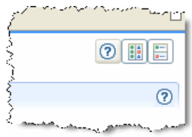
Sometimes the method names are hard to view in the default horizontal layout, therefore, you might want to change the layout to a vertical view instead.

To change the layout of a tab

- Click either the **Horizontal layout** or the **Vertical layout** button in the right hand corner of the tab that you are viewing (see Figure 3-6).

Note: Not all tabs have this functionality.

Figure 3-6 Horizontal and Vertical layout buttons



General Information in JRA Recording

The JRA recording contains a lot of data about the application's behavior, information about JRockit itself, such as JRockit version and which commands were used at the startup of JRockit, etc. All that information is displayed on the **General tab** in the JRA tool. As soon as you have made a recording, your JRA tool automatically opens the recording and displays general information on the **General** tab.

For recordings that have been generated with a JRockit that is older than R26.4, you might still be able to open them up in this version of the JRA tool; however, some fields may be blank, since older versions of JRockit did not have the same recording capabilities as this newer release.

Note: Only text fields that require extra explanations have been covered in this documentation.

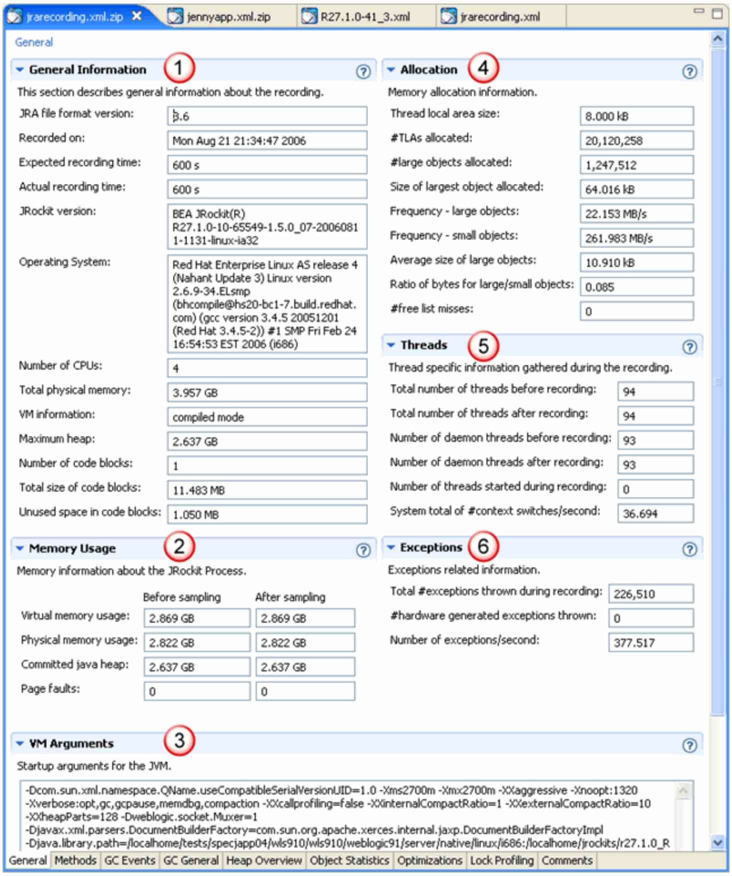
This section is divided into the following topics:

- Getting Familiar with the General Tab
- Viewing General Information
- Viewing Memory Usage Information
- Viewing VM Arguments Information
- Viewing Memory Allocation Information
- Viewing Threads Information
- Viewing Exceptions Information

Getting Familiar with the General Tab

The **General** tab (see Figure 4-1) contains information on both JRockit, your system, and your application.

Figure 4-1 The General tab



The **General** tab is divided into the following sections:

1. **General Information**—contains all general information about the JVM, operating system, recording time, etc.

2. **Memory Usage**—contains information on how JRockit is using the memory.

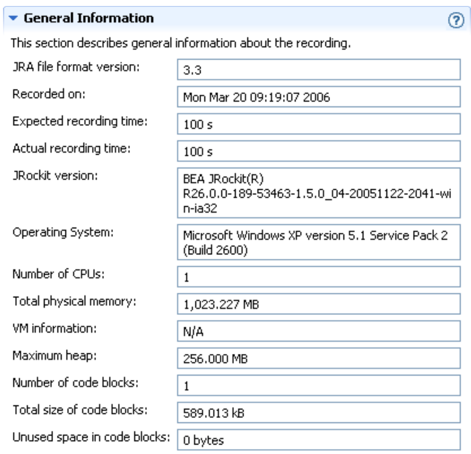
3. **VM Arguments**—lists all startup options that were used.
4. **Allocation**—contains information on how your application allocates memory on the Java heap.
5. **Threads**—contains information on thread usage.
6. **Exceptions**—contains exceptions related information.

Viewing General Information

This section displays (see Figure 4-2) information about the JRockit version, the operating system version, number of CPUs that has been used during the recording, etc.

- The value **Actual recording time** can differ from expected recording time, e.g. if the application that runs on BEA JRockit finished while a recording was still in progress.
- The **Maximum heap** size is set with a JRockit command-line option.
- The **VM information** can be information regarding the garbage collection that has been used.
- The value **Number of codeblocks** is a JVM internal value. All generated code is divided into (non-heap) memory blocks called code blocks.

Figure 4-2 General Information section

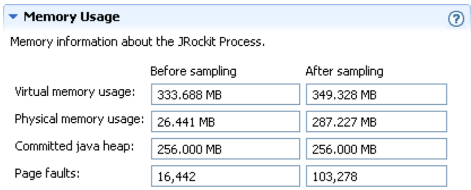


Viewing Memory Usage Information

This section (see Figure 4-3) shows a snapshot of the memory usage before and after the recording.

- The value **Committed java heap** was the current total heap size at the beginning and the end of the recording. It is less than or equal to the maximum heap size.

Figure 4-3 Memory Usage section



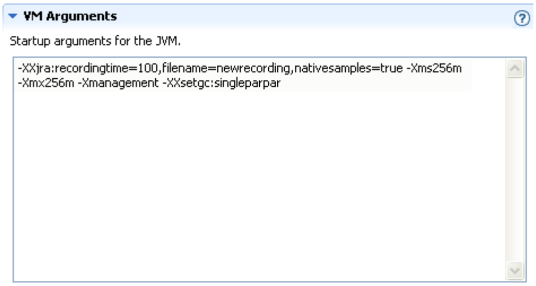
Viewing VM Arguments Information

This section displays (see Figure 4-4) the different command-line options that were used when starting JRockit. The options that have been used in the example are the following:

- The JRA recording time (`XXjra`) has been set (100 seconds).
- The name of the recorded file has been set (`filename`) and native sampling has been switched on.
- The initial, minimum and maximum Java heap has been set (`-Xms` and `-Xmx`)
- The management server is started (`-Xmanagement`)
- The default dynamic garbage collector has been turned off and the static parallel garbage collector is used instead (`-XXsetgc`)

There are many more command-line options that can be set. For comprehensive information on the different command-line options, please see the BEA JRockit Reference Manual.

Figure 4-4 VM Arguments

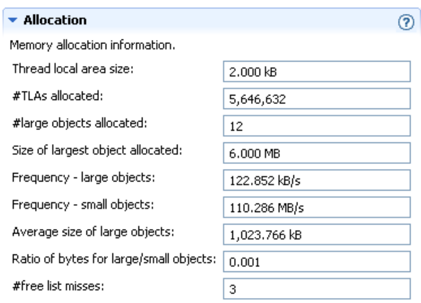


Viewing Memory Allocation Information

This section displays (see Figure 4-5) information about how JRockit is allocating memory on the Java heap.

- The **Thread local area (TLA) size** is a JRockit internal value. It is a small memory area, local to a thread, where the JVM can allocate small objects without having to take the heap lock.
- **Ratio of bytes for large/small objects**. Per default, JRockit considers an object to be large if it is larger than the thread local area size; it is small if it would normally fit in a thread local area. Large objects are always allocated in the old space (second generation) of the heap, never in the nursery.
- The **Number (#) free list misses** is a JRockit internal value. JRockit has a list of free memory blocks on the Java heap. During allocation, an object is normally put in the first free block on the "free list." If it does not fit there, JRockit will try the next block, and the next, etc. Each block where the code block did not fit is considered a "free list miss."

Figure 4-5 Allocation section

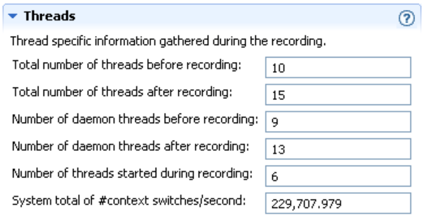


Viewing Threads Information

This section displays (see Figure 4-6) information on the number of Java threads that existed both before and after the recording.

- The value of **Number of deamon threads before/after recording** is the number of deamon threads. A deamon thread is a thread that runs in the background to support the runtime environment, for example, a garbage collector thread. The JVM exists when all non-daemon threads have completed.
- The value **System total of #** *(number)* **context switches per second** is fetched from the operating system. An unusually high context switch value compared to other applications may indicate contention in your application.

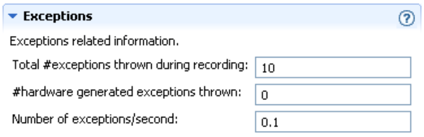
Figure 4-6 Threads section



Viewing Exceptions Information

This section displays (see Figure 4-7) information on the total number of Java exceptions that are thrown during a recording. This includes both caught and uncaught exceptions. Excessive exception throwing can be a performance problem. Hardware generated exceptions are originating from a "trap" in the hardware and are usually the most "expensive" kinds of exceptions.

Figure 4-7 Exceptions information



CHAPTER 5

Methods and Call Trace Information

Methods where JRockit spends most of its time are called hot. Once you have identified such a method, you might want to investigate it to see if it is a "bottleneck" for the application or not. The way that BEA JRockit collects method information is via a sampling thread that is called the hotspot detector. It uses statistical sampling to find Java methods that are candidates for optimization. The samples are collected by iterating through the Java threads in the virtual machine and suspending them one at a time. The current instruction pointer of the suspended thread is used to lookup in which Java method the thread is currently executing. The invocation count of the method is incremented and the method is added to a queue of methods to be optimized if the invocation count exceeds a certain threshold.

The JRA recording system makes use of the hotspot detector by setting it to a high sampling frequency during the recording and directing the samples to the .jra file.

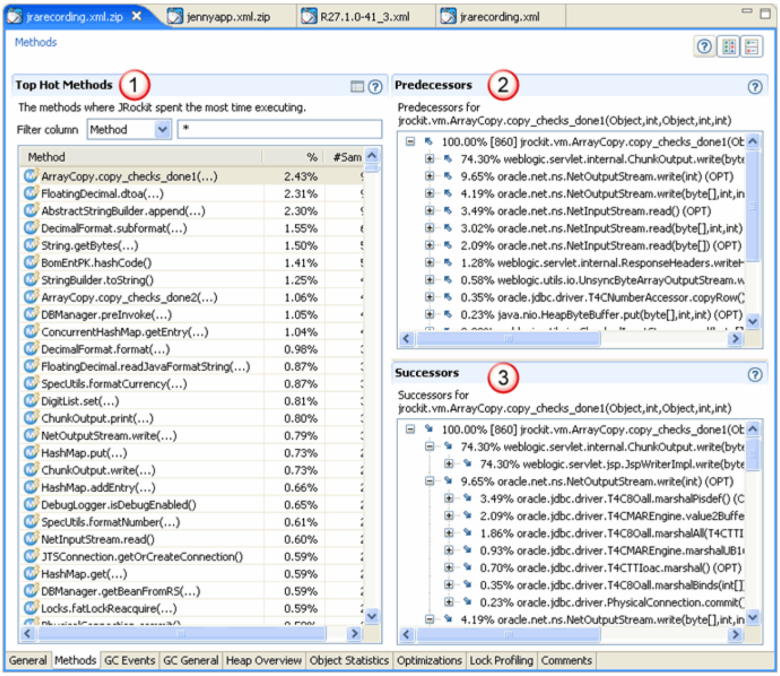
This section is divided into the following topics:

- Getting Familiar with the Methods Tab
- Viewing Hot Methods
- Viewing Predecessors and Successors

Getting Familiar with the Methods Tab

The **Methods tab** is divided into three sections (see Figure 5-1) that lists the top hot methods with its predecessors and successors during the recording.

Figure 5-1 The Methods tab



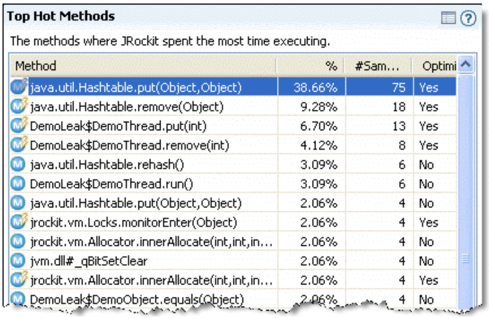
The **Methods** tab is divided into the following sections:

1. **Top Hot Methods**—a listing of the top hot methods. Click on the different table headings to get a different sort order.
2. **Predecessors**—a listing of all preceding methods to the method that you have selected in the **Top Hot Methods** list. If you have selected many methods, there will not be any information shown in this section.
3. **Successors**—a listing of all succeeding methods to the method that you have selected in the **Top Hot Methods** list. If you have selected many methods, there will not be any information shown in this section.

Viewing Hot Methods

The method sampling in JRockit is based on CPU sampling. This requires that you put load on the system to get any samples. The **Top Hot Methods** lists (see Figure 5-2) all methods sampled during the recording and sorts them with the most sampled method s first. These are the methods where most of JRockit's time is spent.

Figure 5-2 Top Hot Methods shown


Top Hot Methods

The methods where JRockit spent the most time executing.

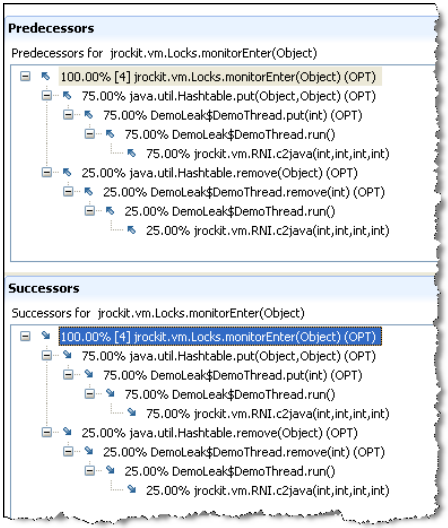
Method	%	#Sam...	Optimi
java.util.Hashtable.put(Object,Object)	38.66%	75	Yes
java.util.Hashtable.remove(Object)	9.28%	18	Yes
DemoLeak$DemoThread.put(int)	6.70%	13	Yes
DemoLeak$DemoThread.remove(int)	4.12%	8	Yes
java.util.Hashtable.rehash()	3.09%	6	No
DemoLeak$DemoThread.run()	3.09%	6	No
java.util.Hashtable.put(Object,Object)	2.06%	4	No
jrockit.vm.Locks.monitorEnter(Object)	2.06%	4	Yes
jrockit.vm.Allocator.innerAllocate(int,int,in...	2.06%	4	No
jvm.dll#_qBitSetClear	2.06%	4	No
jrockit.vm.Allocator.innerAllocate(int,int,in...	2.06%	4	Yes
DemoLeak$DemoObject.equals(Object)	2.06%	4	No

If your recording has native sampling enabled during the recording, you can see methods prefixed by `jvm`, which are native methods in the JVM.

Viewing Predecessors and Successors

By selecting a method in the **Top Hot Methods** list, you can see its sampled **Predecessors** and **Successors** (see Figure 5-3) in the tree views to the right (or below if you have changed the layout of the tab). These are the methods that call the selected method and the methods the selected method calls.

Figure 5-3 Viewing Predecessors and Successors



The number within brackets is the number of sampled call traces of which the method is part. The percentage shows how common a particular path is in the method tree. If you see methods that are called a lot from JRockit, you might want to investigate if that method is causing your application to run slower than necessary.

Garbage Collection Events Information

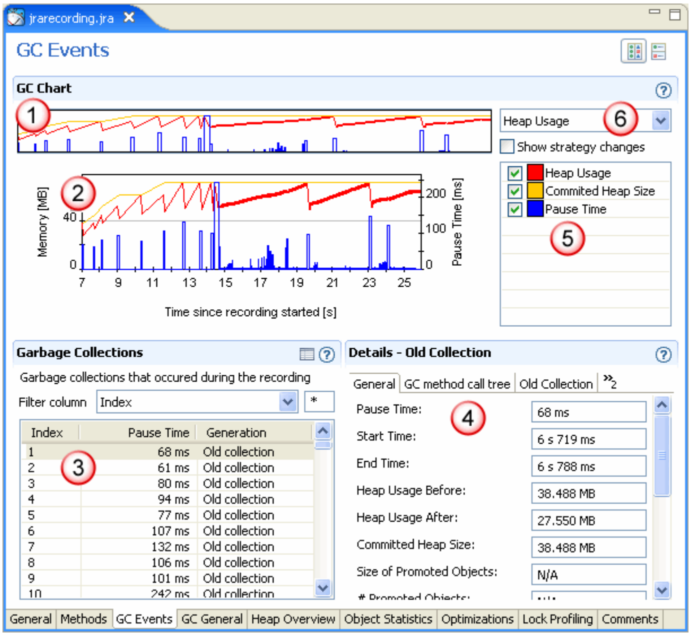
The **GC Events** tab shows detailed information about each garbage collection (GC) event that has occurred. The tab contains a graph for Java heap usage before and after each garbage collection as well as detailed garbage collection information for each collection.

This section is divided into the following topics:

Getting Familiar with the GC Events Tab

The **GC Events tab** is divided into six sections (see Figure 6-1) that pictures how the garbage collector has performed during the recording.

Figure 6-1 The GC Events tab



The **GC Events** tab is divided into the following sections:

1. **GC Events Overview** chart—this chart shows the entire recording in its full length (when you initially open your recording). You can use this to refocus the **Heap Usage** graph, see Changing Focus on Heap Usage Chart.

2. **Heap Usage** graph—this graph shows heap usage compared to pause times and how that varies during the recording. If you have selected a specific area in the GC Events Overview, you will only see that section of the recording. You can change the graph content in the **Heap Usage** drop-down list (marked 6 in Figure 6-1) to get a graphical view of the references and finalizers after each old collection.

3. **Garbage Collections** events—this list shows all garbage collection events that have taken place during the recording. When you click on a specific event, you will see a corresponding flag in the **Heap Usage** graph for that particular event, see Viewing Specifics about Garbage Collections.

4. **Details**—this section contains all the details about the specific garbage collection round. When you select a garbage collection in the **Garbage Collection** list, the tabs in the **Details** section changes depending on if you have selected an old collection or a young collection.

5. **Chart Configuration**—this section allows you to change the appearance on the active chart.
6. **Heap Usage**—this list allows you to toggle the view on the **Heap Usage** chart to view References and finalizers. It shows different types of reference counts after each collection.

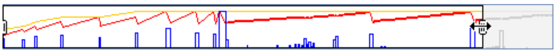
Changing Focus on Heap Usage Chart

Depending on how long your JRA recording is, the **Heap Usage** chart can be quite cumbersome to view in full mode; therefore, you can refocus the chart. by dragging the handles on the slide bar to the section of the recording that you want to view. Once you have set the side on the slide bar, you can slide that section to the position of the chart that you are interested in studying.

To change focus on the Heap Usage chart

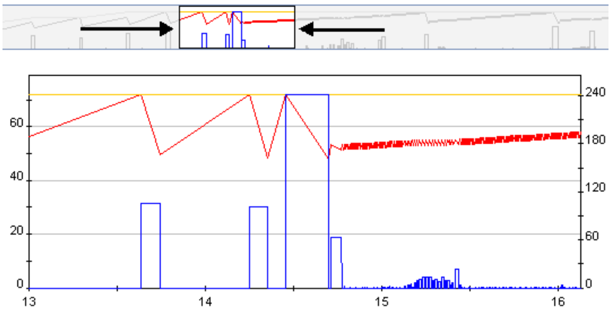
1. Click and drag the handles on both sides on the **GC Events Overview** chart (see Figure 6-2).

Figure 6-2 The GC Events overview zoom function



2. Drag the **GC Events** overview chart into the desired position for the **Heap Usage** chart (see Figure 6-3).

Figure 6-3 The GC Events overview chart



Viewing Specifics about Garbage Collections

The **Garbage Collections** section on the **GC Events** tab is a list of all garbage collections that have taken place during the recording. It lists all garbage collection events during the recording, provided that the garbage collection sampling was enabled. A garbage collection can be an *old* *collection*, which is a garbage collection in the old space of the Java heap or a *young collection*, which is a garbage collection in the young space (nursery).

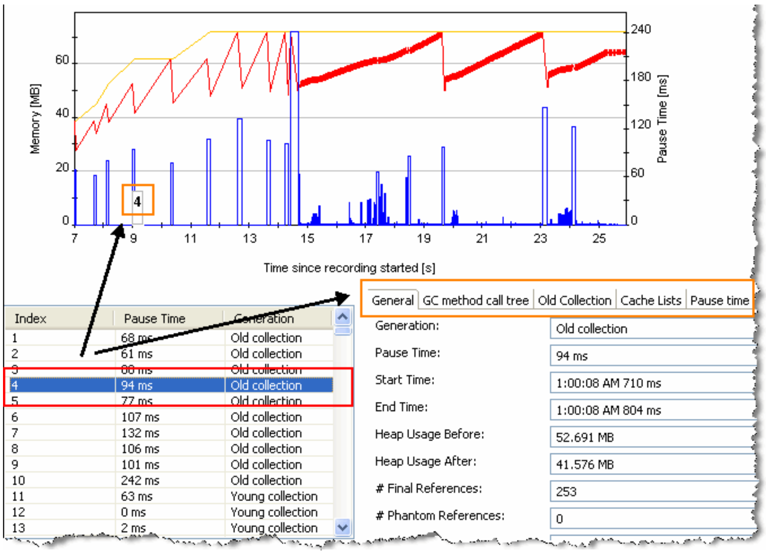
To view one garbage collection in the Heap Usage chart and Details section

1. Scroll in the **Garbage Collection** list to the garbage collection you want to view.
2. Click on that garbage collection.

   The garbage collection index number is now visible in the **GC Chart** and the **Details** section has also changed to show all the specifics about that garbage collection.

   The **Details** section changes name depending on if you selected an old collection or a young collection (see Figure 6-4).

Figure 6-4 Viewing garbage collection in Heap Usage chart and Details section

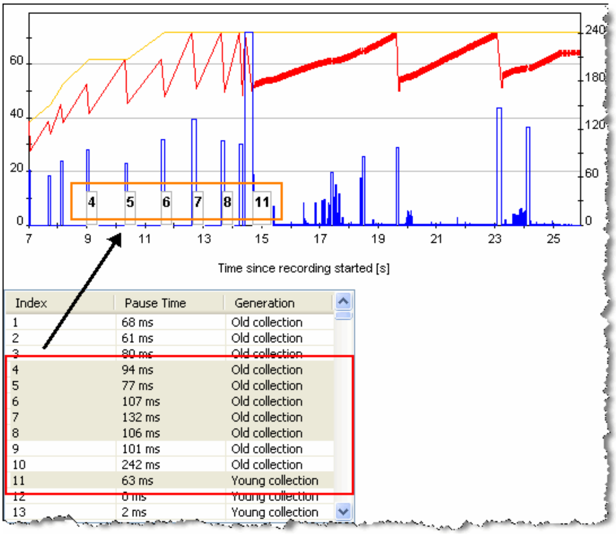


To view many garbage collections in the Heap Usage chart

1. Scroll in the **Garbage Collections** list.
2. Click and hold either the **Shift** key or **Ctrl** key to select multiple collections.

   The garbage collection index numbers are now visible in the **GC Chart** (see Figure 6-5).

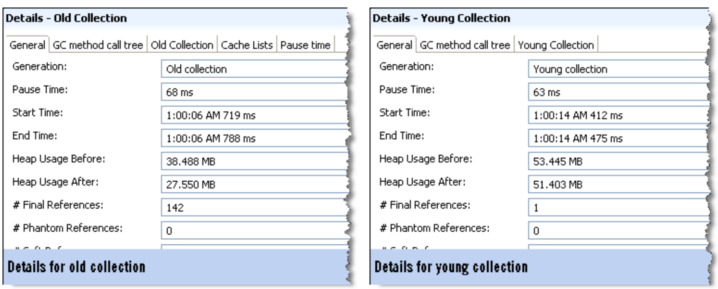
Figure 6-5 Viewing multiple garbage collections in Heap Usage chart



Viewing the Detailed Information About the Garbage Collection

When you select a garbage collection, the **Details** section of the **GC Events** tab changes name to either **Details - Old Collection** or **Details - Young Collection** depending on the type of garbage collection you have selected (this is only possible when selecting one garbage collection). You will also see different sets of tabs that contain specific information about the garbage collection that you have selected (see Figure 6-6).

Figure 6-6 Tab differences when viewing old and young collections



Each one of these tabs are described here. As much of the information in the tabs are fairly self-explanatory, those types of details will not be covered in the documentation.

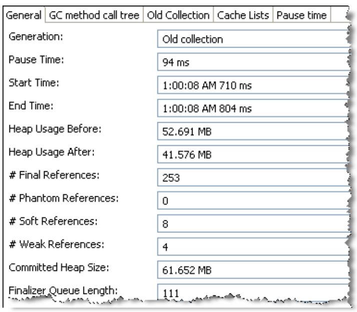
This section describes the following tabs:

- Viewing Information on the General Garbage Collection Tab
- Viewing Information on the GC Method Call Tree Tab
- Viewing Information on the Old/Young Collection Tab
- Viewing Information on the Cache Lists Tab (Only valid for old collections)
- The Pause Time Tab

Viewing Information on the General Garbage Collection Tab

The **General** tab (see Figure 6-7) displays information such as start time and end time of the garbage collection.

Figure 6-7 The General garbage collection tab



- **Generation**—Indicates whether the garbage collector performed an old or young collection (see the Memory Management user guide for more information on generational garbage collection). If a parallel garbage collector has been used, there will be only old collections in the **Garbage Collections** list.
- **Pause Time**—the time in milliseconds that the garbage collector stops all threads in JRockit. This is not the same as end time-start time in the case of a concurrent garbage collector.
- **Start/End Time**—the times when the garbage collection started and ended, counted in milliseconds from when JRockit started.

- **Heap Usage Before/After**—the used heap size before or after the garbage collection.
- **Number of References**—there are different types of references collected during a recording. For information on what a reference is, see the Memory Management user guide.
- **Committed Heap Size**—the total size of the heap (used plus unused memory) after the garbage collection.
- **Finalizer Queue Length (and Before)**—the finalizer queue length.

Viewing Information on the GC Method Call Tree Tab

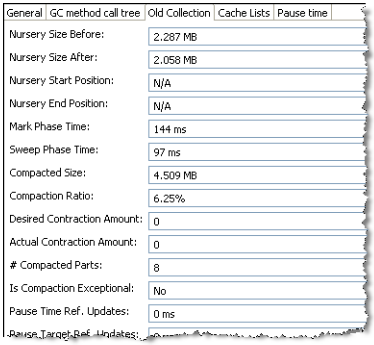
The **GC Method Call Tree** tab (see Figure 6-8) shows an aggregation of the call traces of the threads triggering a garbage collection.

Figure 6-8 The GC Method Call Tree tab

Viewing Information on the Old/Young Collection Tab

The name of this tab is dynamically changed when you select a garbage collection instance in the **Garbage Collections** section. Here you find information about nursery, mark and sweep pause times, etc. (see Figure 6-9).

Figure 6-9 The Old/Young Collection tab



- **Nursery Size Before/After**—indicates the size of the young space on the heap before and after the garbage collection (in some cases the nursery size can increase).

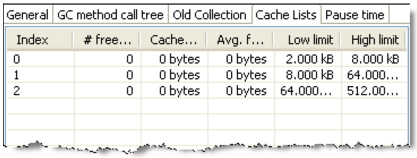
The information below is only valid for old collections:

- **Nursery Start/End Position**—the starting and ending position in the memory address of nursery.
- **Mark/Sweep Phase Time**—the time spent in the marking and sweep phases, measured in milliseconds.
- **Compacted Size**—the size of the heap that has been compacted in the garbage collection.
- **Compaction Ratio**—the ratio of heap size before and after the compaction, measured in percent.
- **Desired/Actual evacuation**—the desired evacuation is the size of the area on the Java heap that you want to evacuate and the actual evacuation is the size of the area that JRockit managed to evacuate. The value for actual evacuation can be smaller than the desired due to temporarily pinned objects (objects that are not allowed to be moved during garbage collection). The evacuation takes place during compaction or shrinking of the Java heap.
- **GC Reason**—indicates the reason for doing this garbage collection.

Viewing Information on the Cache Lists Tab (Only valid for old collections)

The **Cache Lists** tab (see Figure 6-10) the specification for the different cache lists. Each cache list contains settings for upper and lower cache size.

Figure 6-10 The Cache Lists tab



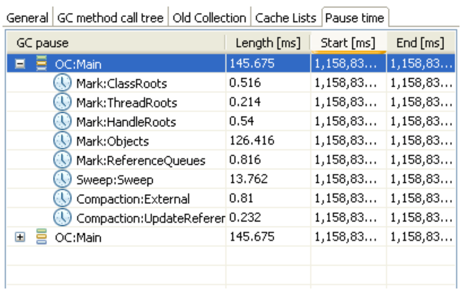
General | GC method call tree | Old Collection | Cache Lists | Pause time

Index	# free...	Cache...	Avg. f...	Low limit	High limit
0	0	0 bytes	0 bytes	2.000 kB	8.000 kB
1	0	0 bytes	0 bytes	8.000 kB	64.000...
2	0	0 bytes	0 bytes	64.000...	512.00...

- **Index**—this is the identification number for the cache list.
- **#free blocks**—the number of free blocks in the cache list.
- **Cache size**—the total size of this cache list.
- **Avg free block size**—the average size of each free memory block in the cache list.
- **Low limit**—the lower limit of a free memory block. There will be no smaller memory block than this in the selected cache list.
- **High limit**—the upper limit of a free memory block. There will be no larger memory blocks than this in the selected cache list.

The Pause Time Tab

The information under the **Pause Time** tab is mainly intended for BEA JRockit internal use when you have sent a JRA recording for analysis to the JRockit engineering team.



General | GC method call tree | Old Collection | Cache Lists | Pause time

GC pause	Length [ms]	Start [ms]	End [ms]
OC:Main	145.675	1,158,83...	1,158,83...
Mark:ClassRoots	0.516	1,158,83...	1,158,83...
Mark:ThreadRoots	0.214	1,158,83...	1,158,83...
Mark:HandleRoots	0.54	1,158,83...	1,158,83...
Mark:Objects	126.416	1,158,83...	1,158,83...
Mark:ReferenceQueues	0.816	1,158,83...	1,158,83...
Sweep:Sweep	13.762	1,158,83...	1,158,83...
Compaction:External	0.81	1,158,83...	1,158,83...
Compaction:UpdateReferer	0.232	1,158,83...	1,158,83...
OC:Main	145.675	1,158,83...	1,158,83...

- **GC Pause**—this column displays the names of the pauses (the main entry in the tree structure). If you are running a parallel garbage collector, then there will only be one pause per garbage collection. For the concurrent garbage collector, there can be several pauses during one garbage collection. The pauses consists of pause parts that can help the JRockit engineering staff to analyze why certain pauses are longer than others.

  Note: During a pause, the application is standing still.

- **Length**—this is the length, measured in milliseconds, of the pause.
- **Start/End**—this is the absolute time, measured in milliseconds, since January 1, 1970.

General Garbage Collector Information

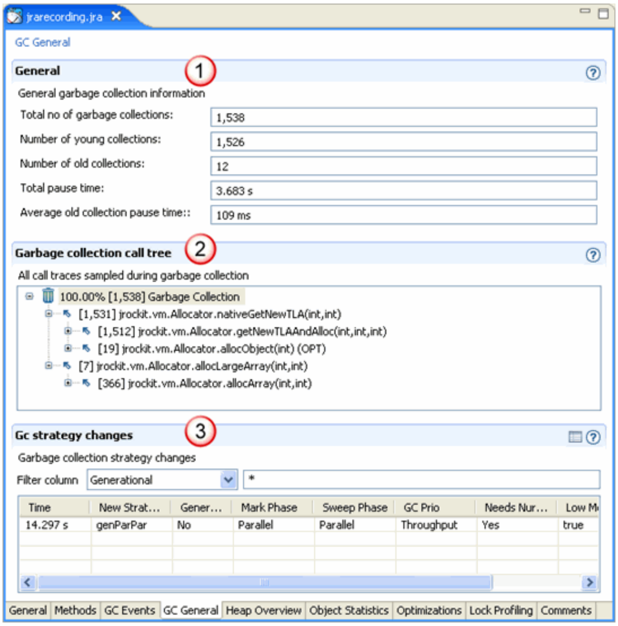
The **GC General** tab shows an overview of information about all garbage collections (GC) that took place during the recording. The information includes, amongst other, the total number of pause times and when and how the garbage collector has changed strategy.

This section is divided into the following topics:

Getting Familiar with the GC General Tab

The GC General tab (see Figure 7-1) is divided into three sections that gives you information about the garbage collection at a glance.

Figure 7-1 The GC General tab



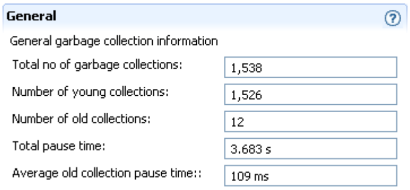
The **GC General** tab is divided into the following sections:

1. **General**—this section shows overall statistics about the garbage collections during the entire JRA recording.
2. **Garbage Collection Call Tree**—this section is a collection of all call traces that were sampled for all garbage collections for the JRA recording.
3. **GC Strategy Changes**—this section lists when a garbage collection strategy took place and how it changed.

Viewing General Garbage Collection Information

The **General** section (see Figure 7-2) shows general garbage collection information such as the total number of garbage collections during the recording and the duration of all pause times. You can use this information to, for example, see whether your application is coming down to desired pause time averages or not.

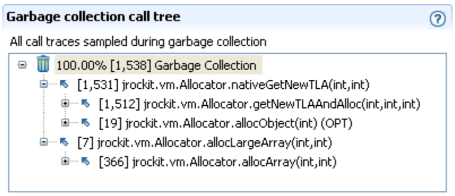
Figure 7-2 General Garbage Collection Information



Viewing Garbage Collection Call Tree Information

The **Garbage Collection Call Tree** section (see Figure 7-3) shows all call traces during the recording that triggered a garbage collection. The number within the brackets (next to the garbage bin icon) is the total number of garbage collection rounds that were performed during the JRA recording. Expand the call tree to see in which methods the garbage collection has taken place.

Figure 7-3 Garbage Collection Call Tree Information

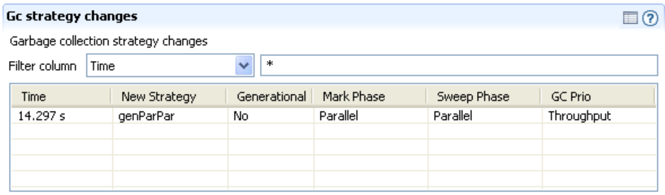


Viewing Garbage Collection Strategy Changes Information

The **Garbage Collection Strategy Changes** section (see Figure 7-4) shows when the garbage collector has changed strategy, for example, JRockit has been set to run for best throughput (`-Xgcprio:throughput`, **GC Prio** in Figure 7-4), then JRockit changes strategy in runtime to best reach this goal (New Strategy). The strategy change can, for example, be from `singleParPar` to `genParPar`. The strategy changes are listed under **New Strategy**. The old strategies are listed under **Generational**, **Mark Phase**, and **Sweep Phase**.

Note: These strategy changes only happen if you are running JRockit with the default garbage collector option, `-Xgcprio`.

Figure 7-4 Garbage Collection Strategy Changes Information



In the example seen in Figure 7-4, there has been one strategy change for the garbage collector.

Java Heap Content Information

The **Heap Overview** tab gives a quick overview of what the memory in the Java heap consists of in you application. You get a quick overview of how the heap looked at the end of the recording and also compiled information about the status of the heap during the entire recording.

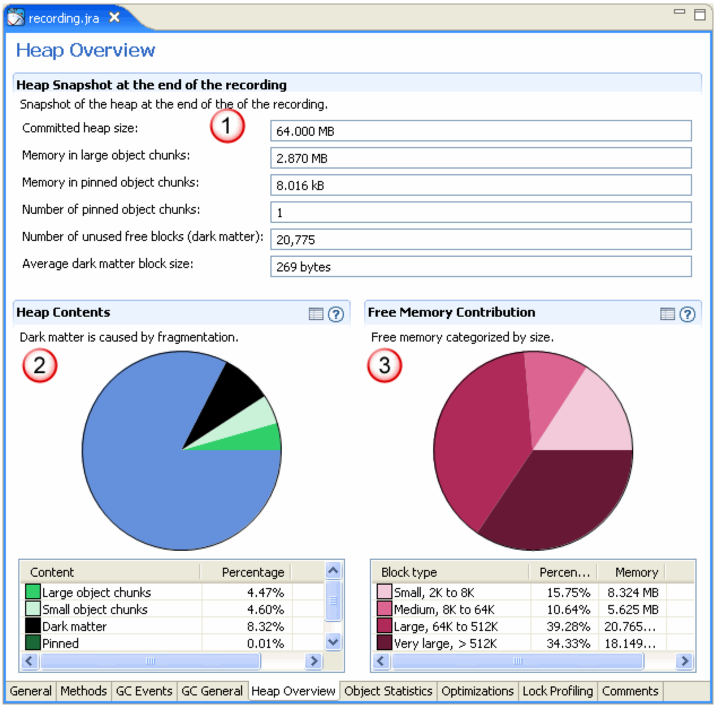
This section contains the following topics:

- Getting Familiar with the Heap Overview Tab
- Viewing the Heap Snapshot at the End of the Recording Information
- Viewing the Heap Contents Information
- Viewing the Free Memory Contribution Information

Getting Familiar with the Heap Overview Tab

The **Heap Overview** tab is divided into three sections (see Figure 8-1).

Figure 8-1 The Heap Overview tab



The **Heap Overview** tab is divided into the following sections:

1. **Heap Snapshot at the End of the Recording**—this section contains all the specifics about your heap at a glance.
2. **Heap Contents**—this graph gives a visual overview of the distribution of different sizes of objects. The table below the graph gives the exact data for each category of memory.
3. **Free Memory Contribution**—this graph gives a visual overview of the distribution of the different chunks of free memory that there is on the heap. The table below the graph gives the exact data for each category of memory.

Viewing the Heap Snapshot at the End of the Recording Information

When the JRA stops recording, it calculates the value of the committed heap size, which is how much heap the application has been allowed to use. This size can be set by the `-xmx` flag.

The memory that is considered **large object chunks**, is the total amount of memory on the heap that the Java application is allowed to use for large objects (64 KB to 512 kB).

The memory for the **pinned object chunks** is the amount of memory that is occupied by pinned objects. A pinned object is both referenced by another object in the application and is not allowed to be moved for compaction purposes, for example, i/o buffers that are accessed from native methods (native i/o). The **number of pinned object chunks** shows a value of how many object that are pinned.

Dark matter is memory that is free, but cannot be used due to the physical layout of the memory chunk (i.e. it might be too small for the application to allocate). Dark matter can cause fragmentation on the disk.

Viewing the Heap Contents Information

The **Heap Contents** pie chart gives a graphic overview of the distribution of objects on the heap. The color coding helps you determine how much of the heap that consists of large, small, and pinned object chunks as well as how much memory is considered dark and how much is free. The amount of dark matter indicates how much space in the Java heap that is wasted due to fragmentation of the Java heap. It is normal to have a certain amount of dark matter on the heap.

For information on how to minimize the dark matter, see Minimize Dark Matter in the BEA JRockit Configuration and Tuning Guide.

Below the chart, there is a table that lists all objects with the exact data: memory in MB and percentage that they occupy of the heap.

Viewing the Free Memory Contribution Information

The **Free Memory Contribution** pie chart gives a graphic overview of how the free memory is distributed in free blocks of different sizes on the Java heap. The block sizes are categorized by the following entities: small, medium, large, and very large. The block sizes are multiples of the minimum block size set at startup (default 2kB). You set the minimum block size with the option `-XXminblocksize`. Below are the multiples used for the different block sizes:

- **Small**: 1–4
- **Medium**: 4–32
- **Large**: 32–256
- **Very large**: 256 and up

Object Statistics Information

The **Object Statistics** tab (see Figure 9-1) displays the most common types and classes occupying the Java heap at the beginning and at the end of the JRA recording.

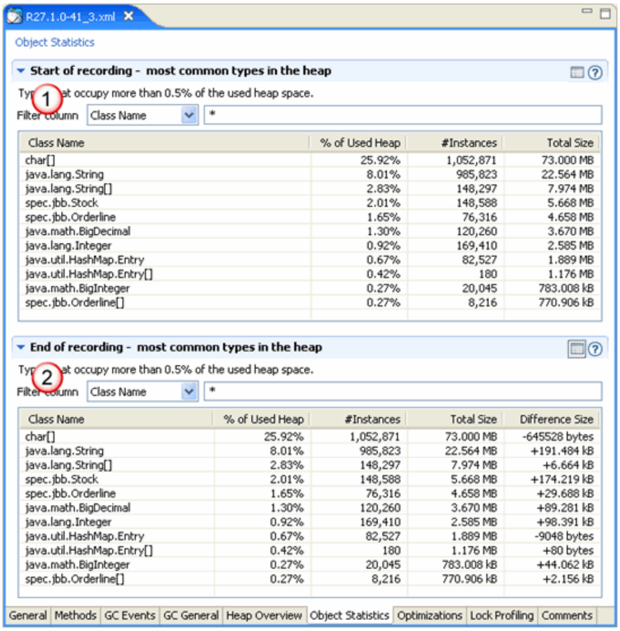
This section is divided into the following topics:

- Getting Familiar with the Object Statistics Tab
- Viewing Start of Recording Information
- Viewing End of Recording Information

Getting Familiar with the Object Statistics Tab

At the beginning and end of a recording session, snapshots are taken of the most common types and classes of object types that occupy the Java heap, that is, the types which instances in total occupy the most memory. The results are shown on the **Object Statistics tab** (see Figure 9-1). Abnormal results in the object statistics might help you detect the existence of a memory leak in your application.

Figure 9-1 The Object Statistics tab



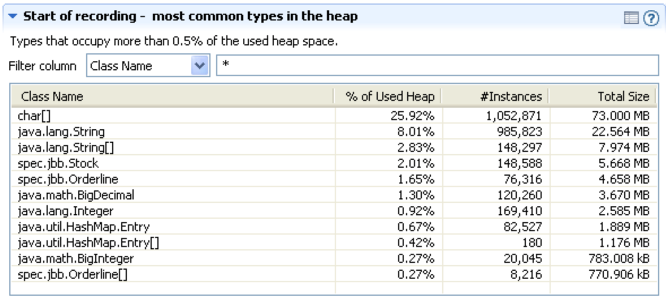
The **Object Statistics** tab is divided into the following sections:

1. **Start of Recording**—this section lists the most common types on the heap at the beginning of the recording.
2. **End of Recording**—this section lists the most common types on the heap at the end of the recording.

Viewing Start of Recording Information

When the JRA starts a recording it looks at the Java heap to see which types occupy the most memory in the used heap space. That information is listed under the **Start of Recording** section (see Figure 9-2).

Figure 9-2 Start of Recording section



Start of recording - most common types in the heap

Types that occupy more than 0.5% of the used heap space.

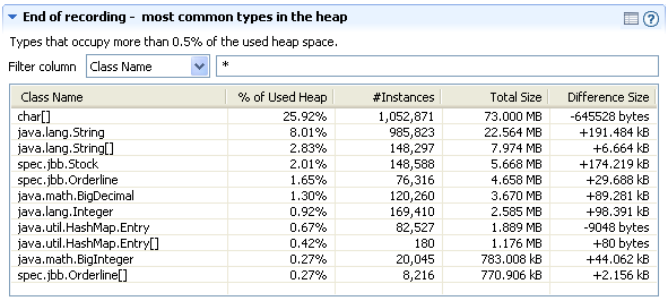
Filter column Class Name *

Class Name	% of Used Heap	#Instances	Total Size
char[]	25.92%	1,052,871	73.000 MB
java.lang.String	8.01%	985,823	22.564 MB
java.lang.String[]	2.83%	148,297	7.974 MB
spec.jbb.Stock	2.01%	148,588	5.668 MB
spec.jbb.Orderline	1.65%	76,316	4.658 MB
java.math.BigDecimal	1.30%	120,260	3.670 MB
java.lang.Integer	0.92%	169,410	2.585 MB
java.util.HashMap.Entry	0.67%	82,527	1.889 MB
java.util.HashMap.Entry[]	0.42%	180	1.176 MB
java.math.BigInteger	0.27%	20,045	783.008 kB
spec.jbb.Orderline[]	0.27%	8,216	770.906 kB

Viewing End of Recording Information

Right before the JRA stops a recording it looks at the Java heap to see which types occupy the most memory in the used heap space. That information is listed under the **End of Recording** section (see Figure 9-3).

Figure 9-3 End of Recording section



End of recording - most common types in the heap

Types that occupy more than 0.5% of the used heap space.

Filter column Class Name *

Class Name	% of Used Heap	#Instances	Total Size	Difference Size
char[]	25.92%	1,052,871	73.000 MB	-645528 bytes
java.lang.String	8.01%	985,823	22.564 MB	+191.484 kB
java.lang.String[]	2.83%	148,297	7.974 MB	+6.664 kB
spec.jbb.Stock	2.01%	148,588	5.668 MB	+174.219 kB
spec.jbb.Orderline	1.65%	76,316	4.658 MB	+29.688 kB
java.math.BigDecimal	1.30%	120,260	3.670 MB	+89.281 kB
java.lang.Integer	0.92%	169,410	2.585 MB	+98.391 kB
java.util.HashMap.Entry	0.67%	82,527	1.889 MB	-9048 bytes
java.util.HashMap.Entry[]	0.42%	180	1.176 MB	+80 bytes
java.math.BigInteger	0.27%	20,045	783.008 kB	+44.062 kB
spec.jbb.Orderline[]	0.27%	8,216	770.906 kB	+2.156 kB

Code Optimization Information

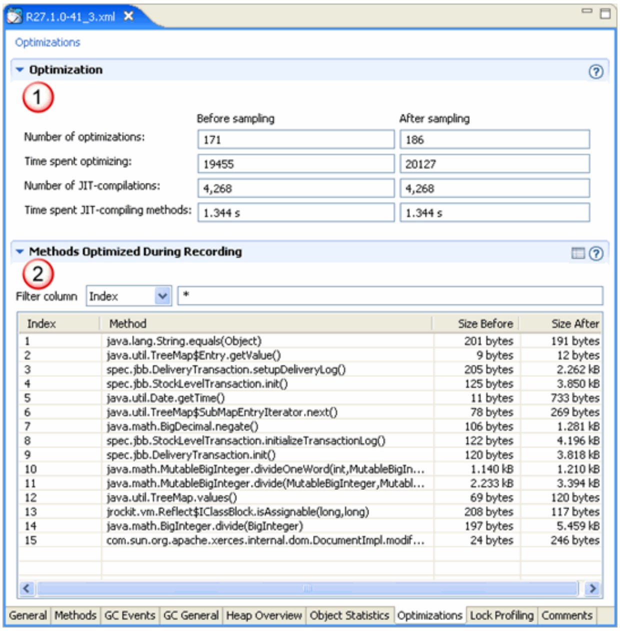
The **Optimizations** tab (see Figure 10-1) displays the methods that were optimized by the adaptive optimization system in JRockit during the recording.

This section is divided into the following topics:

Getting Familiar with the Optimizations Tab

The JRA records all optimization events that occur during the course of the recording. JRockit uses JIT compilation for the initial conversion to machine code. The most commonly used methods are then further optimized during the application run. This information is then displayed in the Optimizations tab (see Figure 10-1).

Figure 10-1 Optimizations tab



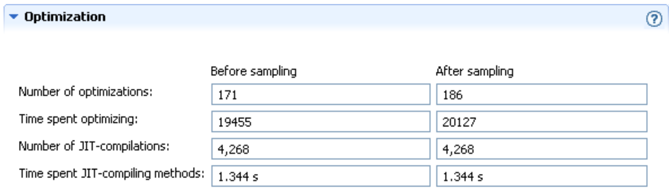
The **Optimizations** tab is divided into the following sections:

1. **Optimization**—this section displays the before and after scenario of the optimizations that have taken place.
2. **Methods Optimized During Recording**—this section displays which methods that have been optimized during the recording, i.e. this is necessarily not a full list of all optimizations that are performed for your application.

Viewing Optimization Information

The **Optimizations** section (see Figure 10-2) contains information on how many optimizations have taken place and the total duration of the optimizations. You can also see how many JIT compilations have been performed and the time JRockit took to compile those. For more information on JIT compilation, see the Introduction to BEA JRockit JDK.

Figure 10-2 Optimization section

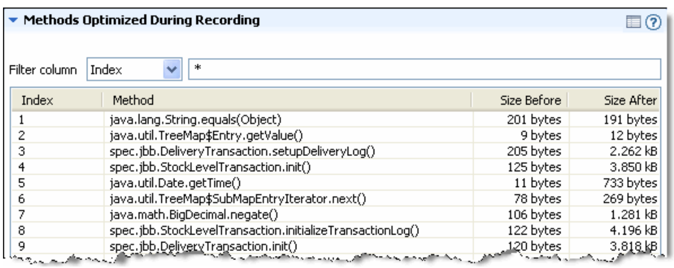


Viewing Methods Optimized During Recording Information

The **Methods Optimized During Recording** section (see Figure 10-3) lists all methods that were optimized during the JRA recording. Here you can study the size changes of each method that has been optimized.

Note: Some optimizations, such as inlining, causes the method size to increase

Figure 10-3 Methods Optimized During Recording section



Index	Method	Size Before	Size After
1	java.lang.String.equals(Object)	201 bytes	191 bytes
2	java.util.TreeMap$Entry.getValue()	9 bytes	12 bytes
3	spec.jbb.DeliveryTransaction.setupDeliveryLog()	205 bytes	2.262 kB
4	spec.jbb.StockLevelTransaction.init()	125 bytes	3.850 kB
5	java.util.Date.getTime()	11 bytes	733 bytes
6	java.util.TreeMap$SubMapEntryIterator.next()	78 bytes	269 bytes
7	java.math.BigDecimal.negate()	106 bytes	1.281 kB
8	spec.jbb.StockLevelTransaction.initializeTransactionLog()	122 bytes	4.196 kB
9	spec.jbb.DeliveryTransaction.init()	120 bytes	3.818 kB

Lock Profiling Information

The **Lock Profiling tab** (see Figure 11-1) shows comprehensive information about lock activity for both the application JRA is monitoring (Java locks) and JRockit itself (native locks). You need to enable the recording capability before you start the profiling of your application. If you have not enabled the lock profiling data recording, the lock profiling sections are left blank on the **Lock Profiling** tab. For more information on locks, please refer to About Thin, Fat, Recursive, and Contended Locks in BEA JRockit.

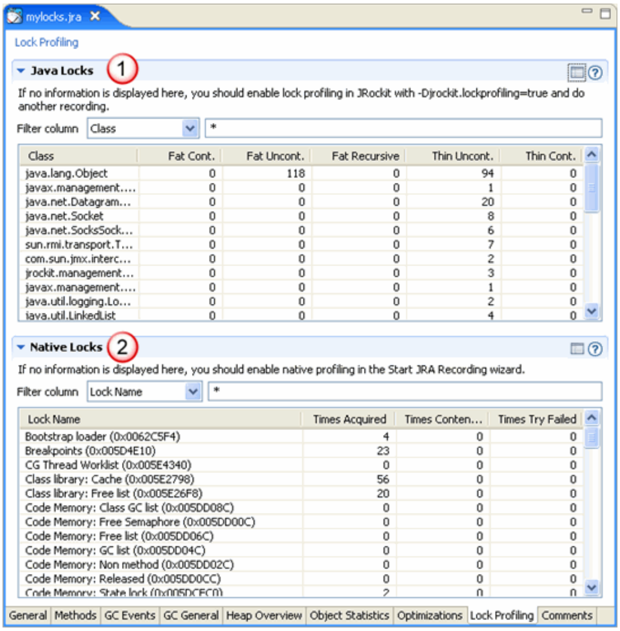
This section is divided into the following topics:

- Getting Familiar with the Lock Profiling Tab
- Java Locks Profiling
- Enabling Java Lock Profiling Data
- Native Lock Profiling
- Enabling Native Locks Information

Getting Familiar with the Lock Profiling Tab

The **Lock Profiling** tab displays lock information for both your application and JRockit (see Figure 11-1).

Figure 11-1 Lock Profiling tab



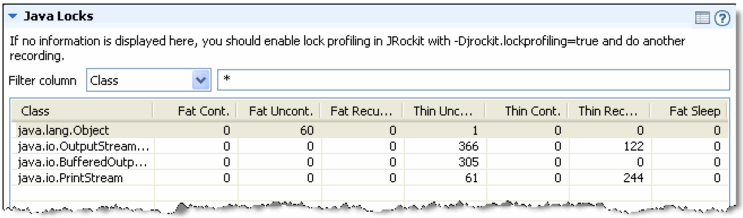
The **Lock Profiling** tab is divided into the following sections:

1. **Java Locks**—this section lists all locks in your application.
2. **Native Locks**—this section lists all locks in JRockit.

Java Locks Profiling

The information that is displayed under the **Java Locks** chart (see Figure 11-2) shows the number of locks of the threads in your application. You see information on the number of fat uncontended and contended locks, thin uncontended and contended locks, thin and fat recursive locks, and fat sleeping locks.

Figure 11-2 Java Locks



Enabling Java Lock Profiling Data

To record Java lock profiling data, you need to enable it from the command line when you start JRockit. If your the Java Locks section is blank, it is not enabled.

To enable Java lock profiling data

- Issue the command `-Djrockit.lockprofiling` at the JRockit command line.

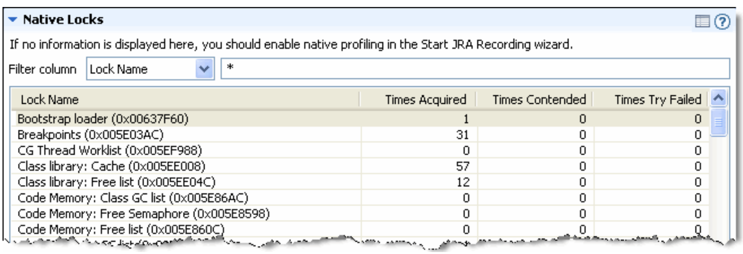
  For example:

```
java -Djrockit.lockprofiling=true -XXjra:<AnyJRAParam> -jar MyApplication.jar
```

Native Lock Profiling

If you are looking at a recording of JRockit J2SE 5.0 or later, the recording includes information about native locks (see Figure 11-3). Native locks are locks in the JRockit internal code and is nothing your application can control.

Figure 11-3 Native Locks



If you find high contention on a JRockit internal lock that might be causing issues for your application, either contact BEA support or contact JRockit through the BEA JRockit news group at the dev2dev web site.

Enabling Native Locks Information

Lock profiling data can only be generated from the command line. If you have no information displayed in the

To enable native locks profiling data

1. Select a JRockit from the JRockit Browser.
2. Click **Mission Control > JRA > Start JRA Recording**.
3. Select **Use native samples**.

Start and End Processes Information

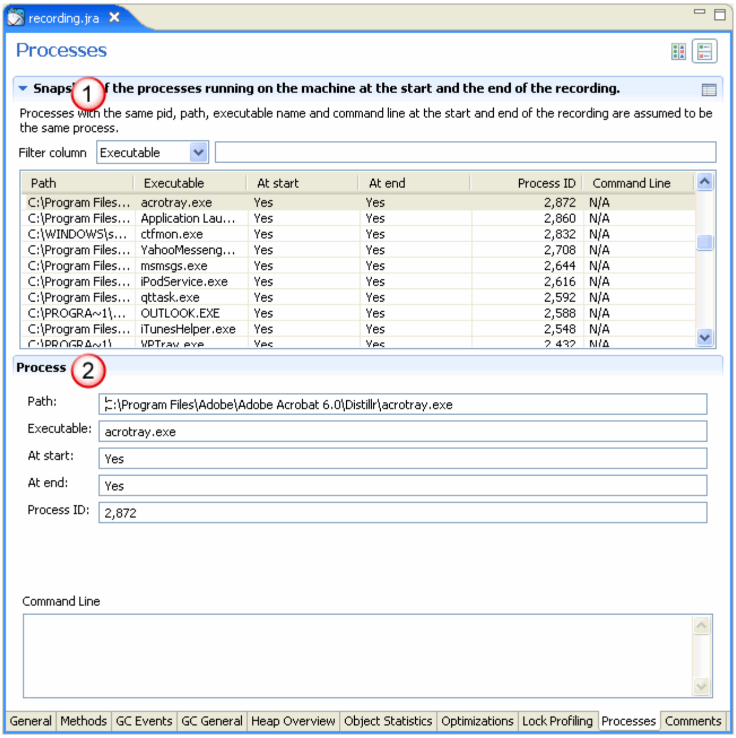
The **Processes** tab (see Figure 12-1) lists which processes were running during both the start and the end of the JRA recording.

This section is divided into the following topics:

Getting Familiar with the Processes Tab

The **Processes** tab displays start and end information of running processes (see Figure 12-1).

Figure 12-1 Processes tab



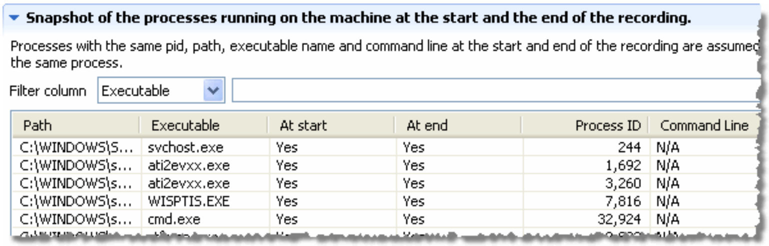
The **Processes** tab is divided into the following sections:

1. **Snapshot of the processes running on the machine at the start and at the end of the recording**—this section lists all processes that were active either during the start or the end of the recording or both.
2. **Process**—this section details the processes information.

Snapshot of Processes at Beginning and End of Recording

The information that is displayed under the **Snapshot** view (see Figure 12-2) shows all processes that were running at the start of the recording and at the end of the recording.

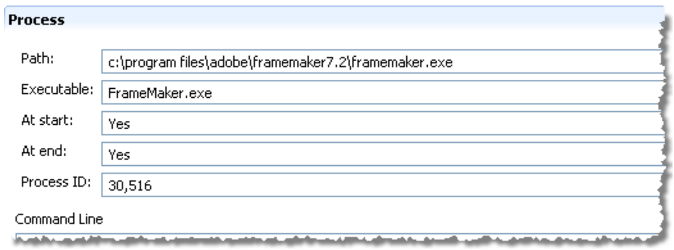
Figure 12-2 Snapshot view



Detailed Processes Information

When selecting a process in the **Snapshot** view, you see a listing of all details for that process at the bottom of the tab (see Figure 12-3). The path, the name of the executable, if the process was present during start and end, the process ID, and also if the process was started with a command-line option.

Figure 12-3 Detail process view



CHAPTER 13

Thread Latency Viewer Overview

The **Thread Latency Overview** tab (see Figure 13-1) displays an overview of all running threads during the recording.

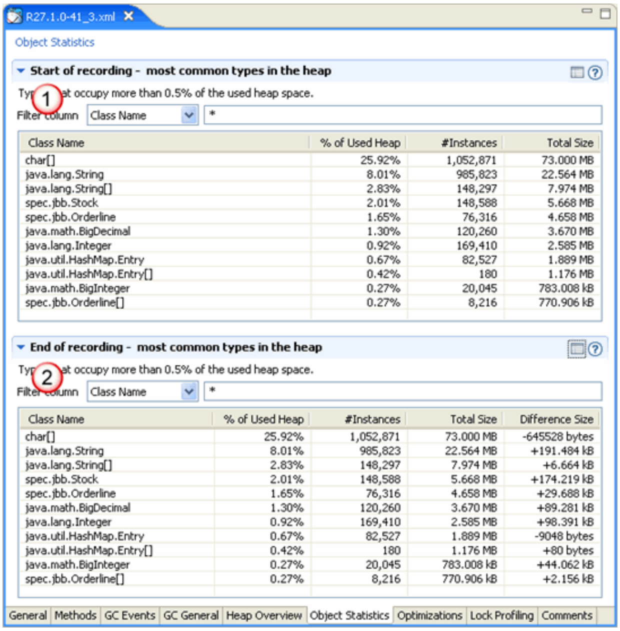
This section is divided into the following topics:

-
-
-

Getting Familiar with the Thread Latency Overview Tab

At the beginning and end of a recording session, snapshots are taken of the most common types and classes of object types that occupy the Java heap, that is, the types which instances in total occupy the most memory. The results are shown on the **Thread Latency Overview tab** (see Figure 13-1). Abnormal results in the object statistics might help you detect the existence of a memory leak in your application.

Figure 13-1 The Thread Latency Overview tab



The **Thread Latency Overview** tab is divided into the following sections:

1. **Start of Recording**—this section lists the most common types on the heap at the beginning of the recording.
2. **End of Recording**—this section lists the most common types on the heap at the end of the recording.

Event Graph Window

Workflow From Recording to Viewing

Drilling Down to Your Problem

Expanding and Collapsing Thread Nodes

Customizing Your View

Using the Filter Functions

Using the Time Scale on Top of Window

Getting Familiar with the Events Table Tab

About Selecting Events for “intressanta mängden”

Selecting Events for “intressanta mängden”

Deleting Events from “intressnta mängden”

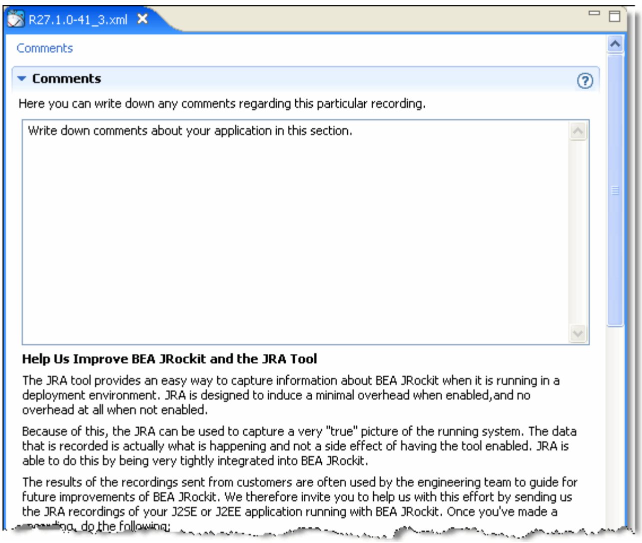
Getting Familiar with the Events Table Tab

Getting Familiar with the Stack Trace Tree Overview Tab

Adding Comments to a Recording

The JRA tool is equipped with a small text editor where you can add comments about the recording and your application. These comments will help the BEA JRockit engineering team to understand what has happened to JRockit and your application during the recording (see Figure 16-1).

Figure 16-1 The Comments Tab



To add a comment

1. Enter a description of you application in the text field.
2. Close the JRA recording.
3. Click **Yes**, when asked if you want to save the recording.